Universidade de São Paulo

Instituto de Matemática e Estatística

MAC5742 - Computação Paralela e Distribuída

Parallel applications in the cloud

Diana Naranjo
{dnaranjo@ime.usp.br}

June 25, 2015

Contents

1 Introduction

We are currently living in a time were data is not scarce at all. In fact, the amount of annual global data center IP traffic was 3.1 zettabytes (ZB) per year (255 exabytes (EB) per month) in 2013 and is estimated to reach 8.6 zettabytes (715 EB per month) by the end of 2018 [1], as can be perceived by Figure 1. These figures give us an idea of the amount of data-intensive computations that are and will be taking place in the near future. Some examples of applications that perform this kind of computations are: web-data analysis, click-stream analysis, data analysis of massive-scale simulations and sensor deployments[2]. One model that seems to be able to respond to the requirements of the mentioned applications is the on-demand cloud computing model. Paired with this model is MapReduce, a distributed computing framework that allows the programmer to execute data intensive computations in a large scalable cluster. One of its main features its is capability of taking care of the fault-tolerance and load balancing problems in a simple manner[3]. All the programmer needs to do is implement two functions: map and reduce. The first to transform the data and the second to aggregate it and produce a final result [2]. The problem with MapReduce is its design for batch-oriented computations and its low-level abstraction.

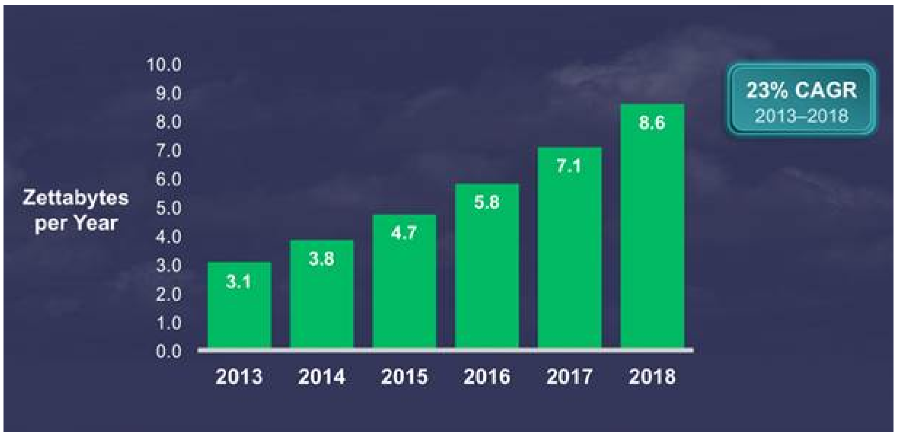

Figure 1: Cisco Global Cloud Index, 2013-2018 [1].

While MapReduce focuses in 1-step dataflows, there are many data analysis and scientific applications that require iterative processing. For example: the PageRank algorithm, which parses a web linkage graph many times in order to derive a page ranking score; recursive relational queries, that require n-step dataflows; dimensional scaling, which aim to visualize the similarity of sparse points; clustering; neural-network analysis; social network analysis; and many other machine learning algorithms[2][4][5]. These applications share one characteristic, they need to be executed in an iterative manner until some convergence is achieved or a stopping condition is reached. Because these applications employ huge amounts of data, a distributed programming model is required, the MapReduce framework could be an option, except that it does not support an iterative kind of implementation. Instead, the programmers must design and manage a series of MapReduce jobs that execute each step. These workaround generates a series of problems related to execution performance and an increase in development time.

In this presentation, three solutions to the problems presented by MapReduce are introduced: HaLoop, iMapReduce and Pig. The first two are related to the iterative applications. HaLoop focuses on loop-aware scheduling and data caching whereas iMapReduce proposes the concept of persistent tasks. Pig solves the problem related to the low-level abstraction. In the following sections we present a bit of background and describe the solutions previously listed.

2 Background

In the present section we give short definitions of the concepts that will be the foundation of the following sections.

2.1 Cloud Computing Technologies

There are many cloud technologies available to the public, we will start by defining Cloud Computing platforms. Cloud Computing platforms allow developers to write applications that run in the cloud and/or use services provided from the cloud. Some example of these platforms are: Google Cloud Platform [6], Microsoft Azure [7], among others.

Cloud computing frameworks, allow the developer to focus in the development of the application. A couple of the most popular being Hadoop and Dryad. In Table 1 we present some of the most important features of each.

Feature	Hadoop	Dryad
Programming Model	MapReduce	DAG based execution flows
Scheduling	Data locality/Rack aware	Data locality/Network topology based run time graph optimization
Failure Handling	Persistence via HDFS Re-execution of map and reduce tasks	Re-execution of vertices
Language Support	Implemented using Java/Other languages are supported via Hadoop Streaming	Programmable via C# DryadLINQ provides LINQ programming API for Dryad

Table 1: Table taken from [5]

2.2 MapReduce

As stated before, Hadoop is one of the most popular cloud computing frameworks. We are going to explain in greater detail the programming model that works at its core. MapReduce aims to satisfy the needs of large-scale data-intensive computations using commodity computer clusters [8]. Some of its most important features are:

- Easy-to-use programming model. Programmers need to implement only two functions: Map and Reduce.
- Fault-tolerance. A reasonable performance is expected even in the presence of faults for several applications due to its re-execution approach [9].
- Scalability. There is no restriction to the number of machines used in a deployment [8].
- Load-balancing. Each machine is assigned different tasks in order to maintain the dynamic load balance [9].
- Data locality-based optimizations. Location information is taken into consideration for the scheduling of tasks. New tasks are ideally scheduled to

machines that already contain the data that will be used as input; otherwise it is scheduled to execute in a machine in the same network as the one that contains the data [9].

2.2.1 MapReduce Execution model

MapReduce's execution model can be described by the following 6-steps [8] Presented in Figure 2.

1. Map. Takes as input a list of key/value pairs that are transformed and mapped to new a new list of keys/values. The keys and values of the input are completely unrelated to the key and values that are outputted.

2. Local sort. In each machine, a local sort by key is performed on the data produced by the map step.

3. Combine (optional). Generates a partial aggregation of the pairs by key.

4. Shuffle. Redistributes all the data among all the available machines by performing a global hash or ordering.

5. Merge/Combine. In each machine, all the data received is ordered and composed into a single stream. If the amount of streams is too large then a multi-pass merge operation is employed; a combine step may be executed in-between intermediate merge steps.

6. Reduce. The data streams associated with one key is processed in order to produce the final result. Generally some kind of aggregation is performed.

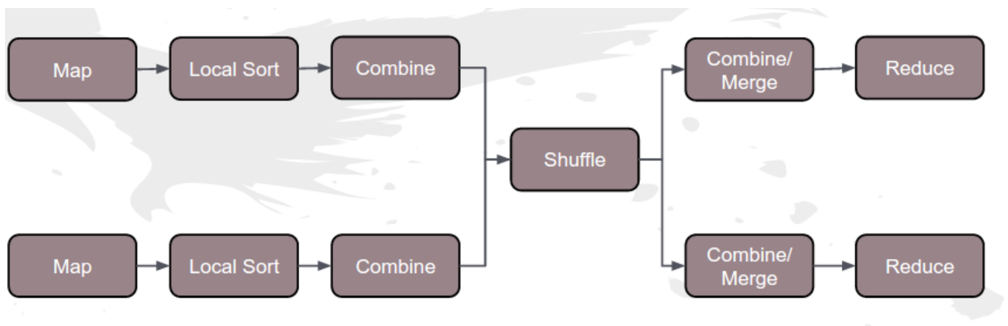

Figure 2: MapReduce execution model [8].

2.2.2 MapReduce Problems

Most ”pleasingly parallel” applications can be performed using a MapReduce technology. However, there are lots of industry and scientific applications that require more complex execution structures. One of these is the iterative computation, present in many machine learning algorithms and many relational data based applications. MapReduce does not provide a natural environment to the development of these applications. In order to obtain an iterative execution, the programmer must orchestrate a series of MapReduce jobs that execute the same work. Some of the main issues with this approach are the following:

- The input data must be reloaded and shuffled in each iteration, even though some part of it remains invariant across all iterations. This generates a waste of the I/O and CPU resources and the network bandwidth; which results in performance penalties.
- Since the work in each iteration is the same, the same tasks are created, scheduled and destroyed.
- In order to control the termination condition the user may need to create an extra MapReduce job for each iteration. This means the schedulers will have to control extra tasks, and may imply extra readings/transportation of data from the disk across the network.
- Also, in order to proceed with the execution of the map tasks, all reduce tasks of the previous iteration must be finished. Which is, sometimes, unnecessary.

Other issues with the MapReduce model is its low-level abstraction; which obliges users to repeatedly implement basic operations such as filtering, aggregation, and else. This slows down the production of code, introduces mistakes and makes them difficult to maintain.

In the following sections we introduce some solutions to the problems presented here.

3 HaLoop

HaLoop is a modified version of the open-source implementation of MapReduce framework, Hadoop. It is designed to facilitate the implementation of iterative programs. Most iterative applications have an important fraction of data that remains invariant across iterations and they will continue to process the information until a termination condition is reached. To obtain a good performance, HaLoop makes the scheduling process loop-aware and employs cache-mechanisms [2]. Some of HaLoop's assumptions are:

- MapReduce clusters will be able to cache the invariant data during the first iteration, so it can be later re-used in the following iterations.
- MapReduce clusters will be able to cache the reducer outputs so as to achieve a more efficient termination condition evaluation.

3.1 Architecture

Figure 3 illustrates the architecture of HaLoop. It has inherited much of its architecture from Hadoop. The system has two main parts: the master node and many slave nodes. The client submits jobs to the master node, which in return partitions it into tasks that are scheduled to the slave nodes. The tasks are executed in a parallel fashion and can be map or reduce tasks. To control the completion of a task, each slave node has a task tracker daemon process that communicates with the master node. Once a slave node completes a task, the master node assigns a new task to the now available node.

The changes applied to the architecture were the following:

1. Loop-control module. The master node will re-initiate the map-reduce steps that integrate the loop body of the iterations. This process will be repeated until a stopping condition is achieved.
2. Data-locality. The scheduler will seek to optimize data-locality by means of caching and indexing. Though Hadoop already performs data-locality optimizations. HaLoop aims to improve those rates.

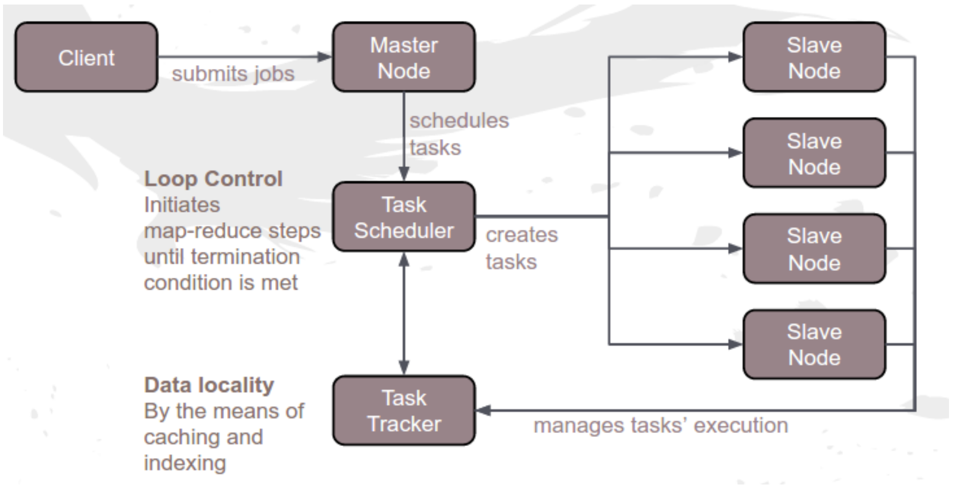

Figure 3: HaLoop Architecture [2].

In the following sections we aim to explain a little bit further the concept of data-locality and how is this managed.

3.2 Loop-aware task scheduling

The goal of having a task-aware scheduler is to place on the same machine new tasks that need access to the same data. This situation may arise in map/reduce tasks that evaluate the same data in different iterations and is called inter-iteration locality. The input data is divided into partitions (indexed by file URL in map task and hash value in reduce task), we say the schedule presents inter-iteration locality if for all iterations, except the first one, the task that consumes some partition d is assigned to the same machine [2].

To accomplish this goal HaLoop keeps track of all the partitions that have being employed in some task assigned to a machine. During the first iteration, the data partition needed by a task is cached in the slave node that will execute said task and the master node takes a note of the mapping between the node and the data. For the following iterations, the master node will assign to each node new tasks that employ one of the data partitions that are currently being cached there. In case the slave node is full, then the task is re-assigned to a new node that is nearby the one that holds the data needed for its execution.

3.2.1 Caching data

As stated before caching plays an important role in the data-locality optimization. HaLoop categorizes the data being cached into:

1. Reducer input cache. Aims to improve the performance of applications that execute joins against large invariant data; for example, PageRank. The idea is that as the tuple of key/values that come as input to the reduce function a list of invariant data values associated to that key is passed as well. By doing this, the invariant data is cached right before the reduce step is about to be executed and for the next iterations the data is already available. Since the number of reduce tasks remains the same in each iteration the default hash function used in the shuffle step can also be employed to retrieve the invariant data associated to the key.

2. Reducer output cache. Aims to reduce the cost of evaluating the termination condition. In the MapReduce model, the user had to create a new job to evaluate if the distance between the results of two iterations (end of reduce step) is smaller than a threshold. In HaLoop, the results of the previous iteration is cached. This way, the distance can be performed by each slave node; the results are then sent to the master node that is responsible of computing the final distance.

3. Mapper input cache. As Hadoop, HaLoop also aims to reduce the non-local reads. The approach differs in the fact that HaLoop uses the inter-iteration locality concept. And, it is in the non-initial iterations that the non-local reads are meant to stop.

4 iMapReduce

iMapReduce is also a modified version of Hadoop. It is designed to facilitate the implementation of iterative programs. The main difference with HaLoop are the concepts that aim to optimize these kind of applications. iMapReduce focuses on the employment of persistent tasks, that require the data to be loaded once and the use of asynchronous tasks [3]. Some of the assumptions made by iMapReduce about the applications to be developed are the following:

- The map and reduce operations employ the same key. This restriction generates a one-to-one mapping between the map tasks and the reduce tasks.
- Each iteration contains only one MapReduce job. This restriction covers most of the graph-based iterative algorithms.

The previously stated assumptions can be relaxed by the employment of some extensions to iMapReduce. However, those applications will not execute as fast as the ones that do respect these assumptions.

4.1 Persistent tasks

In MapReduce implementations of iterative computations, the tasks needed to perform each iteration are created, scheduled and destroy. The problem with this approach is that the tasks do not change from one iteration to the next one, the same work is being performed. However,resources are being wasted in each iteration to manage the tasks needed and also to load and save the input and output of each job into the Distributed File System, as can be appreciated in Figure 4. iMapReduce introduces the concept of persistent tasks, which are map/reduce tasks that are kept alive during the complete iteration process [3], as can be appreciated in Figure 4. While the data is being loaded or process, the tasks remain dormant awaiting for their input data to arrive. In the case of the map tasks, they await for the results of the reduce tasks executed in the previous iteration; meanwhile the reduce tasks await for the results of all the map tasks of the current iteration.

One thing to take into consideration is the fact that since all map/reduce tasks will be kept alive, then all of them will have to be created during the first iteration. To be able to create all of them the number of task slots available needs taken into consideration. The number of tasks scheduled cannot surpass the amount of tasks the system is able to accommodate simultaneously. This may in turn generate load balancing problems since one task may end with a heavier load in future iterations. This problem is solved by performing periodical migrations of tasks. The master node, monitors the velocity with which each machine is able to deliver a result. If one node seems to take more time to deliver then it will be classified as a strangler. If, on the contrary, it gives a high throughout, then it will be classified as a leader. The idea is to migrate new tasks from stranglers to leaders.

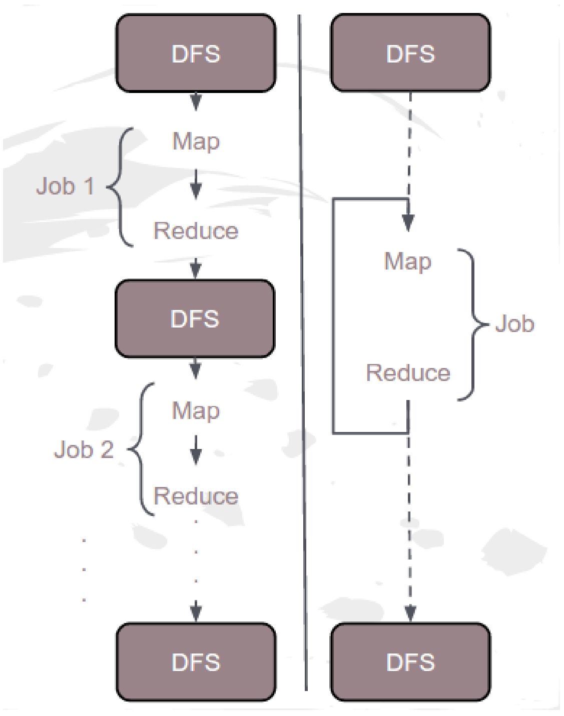

Figure 4: Persistent tasks [3].

4.2 Data Management

As in HaLoop, the data is view as the sum of two parts: the invariant, called static data; and the variant, called state data. Only the state data is shuffled during the shuffle step and the only input of the reduce step. The data that is sent from the reducer tasks to the map tasks of the next iteration is a combination of the state data and the static data, as can be seen in Figure 5. In order to pass the state data, a persistent socket is employed. Since, by the restrictions of iMapReduce, the map and reduce tasks are mapped one-to-one these is performed directly. To send the correct static data to the map task, the hash function employed in the shuffle state can be used; once again thanks to the one-to-one association.

Another extra optimization of resources is the same node assignment. The map and reduce tasks that associated are assigned to the same machine in order to reduce network cost otherwise needed to pass the state data.

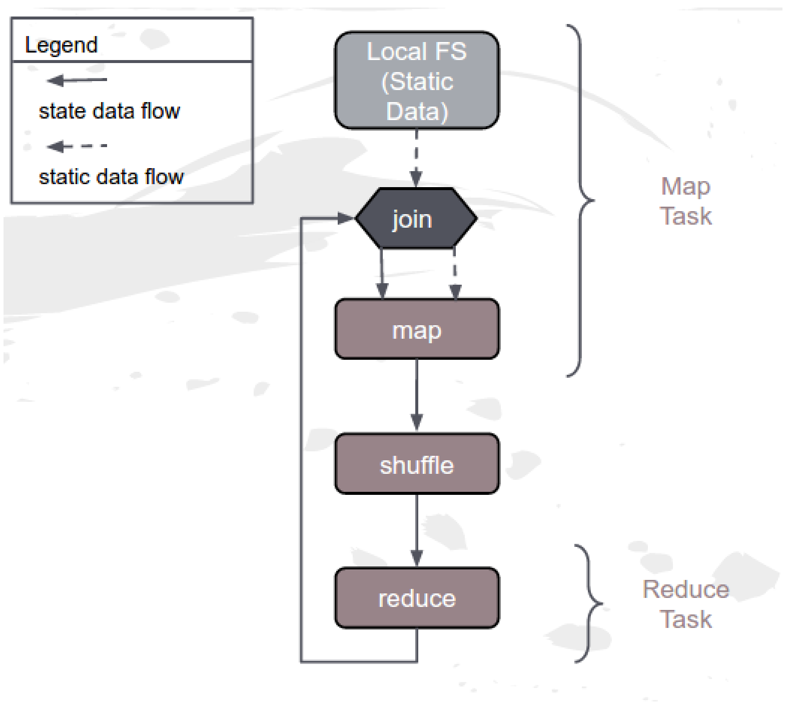

Figure 5: Static and state data dataflow [3].

4.3 Asynchronous execution map tasks

In the MapReduce implementation of iterative programs, for the map tasks of some iteration to start, all the reduce tasks of the previous iteration needed to be finished. This restriction is unnecessary in the iMapReduce model. A map task can initiate as soon as the state data from its corresponding reduce task is available, this means that the execution of the map tasks is asynchronous. This feature does come at a price, to maintain MapReduce's fault-tolerance the state data needs to be periodically saved. To do this a buffer is employed. When a reduce task finishes, the results are written into a local file; once the buffer reaches a threshold, the data is sent to the map tasks.

5 Pig

Pig aims to solve the MapReduce problems with low-level abstraction. It focuses on making the creation of programs easier for the user, by the creation of dataflow programs implemented in Pig Latin. This language provides constructs that allow high-level data manipulation as well as the ability to mix built-in relational-style operators, such as filter and join; with user-provided executables, such as scripts and pre-compiled binaries [8]. Pig Latin compiles these programs into MapReduce jobs, executed by Hadoop and controlled by Pig. Pig facilitates the creation of programs, but performance is also an important factor. In Figure 6, we can perceive that the evolution of Pig aims to optimize and reached a one-to-one correspondence with MapReduce generated code. In the following sections we will the detail the compilation and execution stages that a dataflow program must undertake and also some of the most important features that Pig presents.

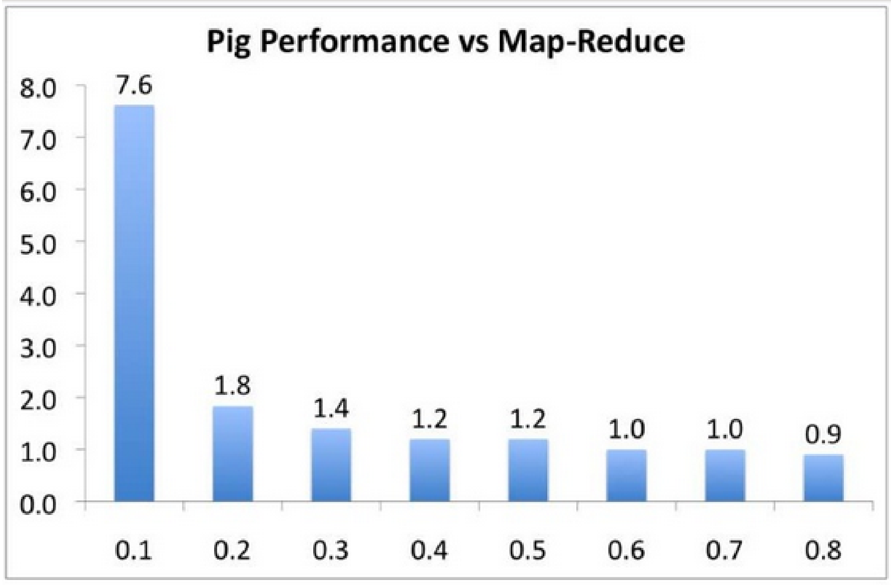

Figure 6: Pig performance [10].

5.1 Compilation and execution stages

A Pig program goes through a series of stages in order to be executed. These stages as depicted in Figure 7 and are explained in the following paragraph.

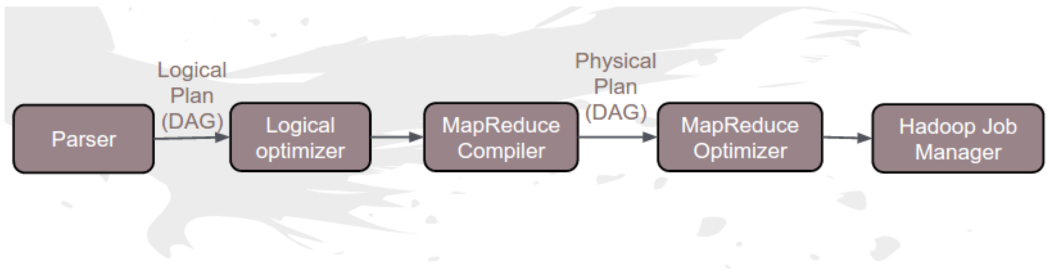

Figure 7: Pig compilation and execution stages [8].

1. Parser. In this first step, the code is evaluated to verify that the syntaxis is correct and that all variables are defined. Some of the other tasks undertaken by this step include: type checking; verification of insatiability of classes needed for user-defined functions and streaming executables existence confirmation. The output of this step is a logical plan that holds a one-to-one correspondence between Pig Latin statements and logical operators, arranged in a directed acyclic graph (DAG) fashion.

2. Logical optimizer. During this step a series of logical optimizations are executed. One of these optimizations is projection push-down, which aims to minimize the amount of data scanned and processed.

3. MapReduce compiler. After the optimizations have been performed, the logical plan is transformed into a series of MapReduce jobs, called the physical plan. In this stage, each physical operator is embedded into one of the MapReduce steps (map, local sort, combiner, shuffle, merge or reduce).

4. MapReduce optimizer. During this step a series of MapReduce optimizations is performed, One of these optimizations is the partition of distributive and algebraic aggregation functions, e.g. average, into three steps: initial, e.g. generation of (sum, count) pairs; intermediate, e.g. combination of n (sum, count) pairs into one pair; and final, e.g. combination of n (sum, count) pairs and computation of quotient. Each of these steps is mapped to the map, combine and reduce stages respectively.

5. Hadoop job manager. During this final stage the MapReduce jobs are topologically sorted and submitted to Hadoop in said order. Pig monitors the

execution status of the jobs and periodically reports the progress of the program to the user.

5.2 Important features

There are two important features that deserve a little extra explaining in Pig: data management and streaming.

5.2.1 Data Management

Pig was developed in Java; which means memory is not controlled by the user. Memory overflow is the most common problem that Pig users need to face. This problem arises due to the materialization of large bags of tuples between and inside operators, that is when complete bag needs to be operated on, e.g. median function. The sum of the tuple sizes may exceed available memory. To control this situation, Pig maintains a list of the bags ordered in descending manner according to their estimated size. The estimated size of a bag is computed by the evaluation of the size of n-tuples multiplied by the number of tuples in the bag. This list of bags and a low-memory handler will generate the removal of large bags once they are no longer needed. That way, the memory overflow situation can be evaded. Some issues with this approach is the fact that sorting the list can be time consuming; as well as the removal operation.

5.2.2 Streaming

As stated before, users are able to integrate user-provided executables. Streaming allows the incorporation of this executables into the program dataflow. These executables need not be programmed in Java and the only requirement is that they read from a input steam and write to an output stream. They are executed in an asynchronous manner with the employment of input and output queues.

6 Conclusion

This essay presents an exploration of all the possible solutions to the generation of parallel computing application in the cloud. It would be impossible to generate new applications that use the facilities that the cloud provides if the structure to support it had to be developed each time. That is why it is important to know and evaluate what framework works best with the necessities of the application. As was seen in the previous sections, HaLoop and iMapReduce are excellent alternatives to iterative applications that need to execute the same work in each iteration; whereas Pig is more oriented to n-step queries that are so common in the industries that work with huge amounts of data.

References

[1] Cisco and/or its affiliates. Cisco global cloud index: Forecast and methodology 20132018 white paper, 2014.

[2] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters.

[3] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. C.: Imapreduce: a distributed computing framework for iterative computation. In *In: Proceedings of the 1st International Workshop on Data Intensive Computing in the Clouds (DataCloud*, page 1112, 2011.

[4] Thilina Gunarathne, Bingjing Zhang, Tak lon Wu, and Judy Qiu. Portable parallel programming on cloud and hpc: Scientific applications of twister4azure," presented at the portable. In *Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure*, 2011.

[5] Jaliya Ekanayake, Xiaohong Qiu, Thilina Gunarathne, Scott Beason, and Geoffrey Fox. High performance parallel computing with cloud and cloud technologies.

[6] Google. Cloud platform, 2015.

[7] Microsoft. Get started with azure, 2015.

[8] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top

of map-reduce: The pig experience. In *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT*, pages 1414–1425, 2009.

[9] Jeffrey Dean and et al. Mapreduce: Simplified data processing on large clusters, 2004.

[10] Alan Gate. Pig-making hadoop easy, 2011.