Stress-testing clouds for big data applications

Sutii, A.

Award date:
2013

Stress-Testing Clouds for Big Data Applications

Alexandru Şutîi

TU/e technische universiteit eindhoven

Stress-Testing Clouds for Big Data Applications

Master's Thesis in Computer Science

System Architecture and Networking Group
Faculty of Mathematics and Computer Science
Eindhoven University of Technology

Alexandru Şutîi

6th September 2013

Author
Alexandru Şutîi

Supervisor
prof. dr. ir. Dick H. J. Epema

Title
Stress-Testing Clouds for Big Data Applications

MSc presentation
13th September, 2013

Graduation Committee

prof. dr. ir. D. H. J. Epema	Eindhoven University of Technology
dr. R. H. Mak	Eindhoven University of Technology
dr. M. Pechenizkiy	Eindhoven University of Technology

Abstract

In the last decade, the world has been experiencing an explosion of data. On the one hand, larger data provide unprecedented opportunities to better understand the observed entities, but on the other hand, it becomes increasingly difficult to manage and process such large data. As processing Big Data on single machines has become infeasible, distributed systems have proven to be the only promising solution to tackling Big Data processing. In this thesis, we use Big Data applications to stress-test and benchmark one cluster system, the Distributed ASCI Supercomputer 4 (DAS4), and two cloud systems, OpenNebula and Amazon EC2.

We identify three Big Data domains: Graph Processing, Machine Learning, and Scientific Computing. For each domain, we select representative applications and execute them on the three systems with increasing amounts of input data. For graph processing, we run the Graph500 benchmark and three graph algorithms implemented with the Apache Giraph library. For machine learning, we run three algorithms implemented with the Apache Mahout library. For scientific computing, we execute the SS-DB benchmark on top of the SciDB data management system.

For each of the three Big Data domains, we compare the performance of the three systems, we identify the maximum amount of data that can be processed by clusters of up to 8 computing nodes, and we discuss the bottlenecks that stop the applications from scaling above certain limits. We find that the cluster system (DAS4) performs better than the two cloud systems: up to 11 times for graph processing, and up to 2 times for machine learning and scientific computing. We find that in one hour, a cluster of 8 nodes is able to process around 30GB of data (2^{27} vertices) for graph processing, around 100GB of input data (20 millions text documents) for machine learning algorithms, while for scientific processing, we estimate that up to 1TB of input data can be processed.

Preface

This report is my Master thesis for the conclusion of my Master program at the Faculty of Mathematics and Computer Science, Eindhoven University of Technology. I have worked on the thesis at the Eindhoven University of Technology, but I have remotely conducted my experiments on the Distributed ASCI Supercomputer 4 (DAS4), which is a six-cluster wide-area distributed system located in the Netherlands. Specifically, I have used two clusters of DAS4 for my experiments: the one located at the Delft University of Technology and the one located at the Vrije Universiteit Amsterdam.

I would like to express my deepest gratitude to my supervisor, Professor Dick Epema, for guiding me in the entire process of completing my Master thesis. He provided me with invaluable advice on how to approach the problem and how to rigorously present my findings in this report. I am confident that what I have learned from Prof. Epema is valuable not only for this thesis, but also for my future carrier as a software engineer.

I would also like to thank my colleague Ana-Maria Farcaşi for reviewing this report and providing me with feedback. I am thankful to the DAS4 administrators, Paulo Anita and Kees Verstoep, for helping me install SciDB on DAS4. Moreover, I would like to thank Aleksandra Kuzmanovska and Bogdan Ghiţ for helping me to solve the practical issues I encountered while running my experiments.

Alexandru Şutîi

Eindhoven, The Netherlands
6th September 2013

Contents

Chapter 1

Introduction

The continuous decrease in hardware prices enables us to collect increasingly larger amounts of data. The estimated amount of electronic data stored around the world increased from 130 exabytes in 2005 to 2,720 exabytes in 2012 and it will presumably increase to 7,910 exabytes in 2015. Currently, many companies mine datasets in the order of terabytes and even petabytes. The term *Big Data* is being used to refer to datasets of such magnitude. Because the data sizes increase continuously, it is hard to define what can and what cannot be referred to as Big Data. An informal definition [28] states that at a certain point in time, big data are those that are hard to analyze and manage with existing software and hardware solutions, and push the engineers to develop state of the art solutions that would make the processing easier.

We have long passed the point where large datasets can be processed with only one commodity computer. Because the prices for commodity hardware have decreased significantly in the last decade, it has become increasingly popular to create clusters of commodity computers and use them for applications requiring large processing power (e.g., Big Data applications). In cloud systems, users can create clusters by leasing virtual machines, and pay as they go. This made cloud systems popular for running Big Data applications. Despite their popularity, not much research has been done in order to assess how different cloud software systems (such as OpenNebula, OpenStack, Amazon Web Services) perform when processing Big Data.

The research question we address is what are the infrastructure limitations of cloud and cluster systems for processing Big Data applications. Specifically, we focus on assessing and comparing the performance of one cluster system, the Distributed ASCI Supercomputer 4 (DAS4) [9], and two cloud systems, OpenNebula [32] and Amazon Elastic Compute Cloud (EC2).

DAS4 is a wide-area distributed system located in the Netherlands. It consists of six clusters and has a total of about 1,600 cores. Unlike cloud systems, where the users run their applications on leased virtual machines, on DAS4, the users run their applications directly on the physical machines.

OpenNebula is an open-source cloud computing toolkit for managing heterogeneous distributed data center infrastructures. OpenNebula provides the ability to create clusters of virtual machines on top of existing physical heterogeneous clusters. For this thesis, we use the OpenNebula installation deployed on top of DAS4. Because OpenNebula virtual machines are leased directly on DAS4 physical machines, by comparing the performance of the two systems for a certain application, we identify the overhead introduced by CPU and I/O virtualization.

Amazon EC2 is a cloud service that, as OpenNebula, allows the users to lease virtual machines and pay as they go. It is interesting to compare OpenNebula and Amazon EC2 in order to understand how differences in their processing resources and communication systems contribute to differences in performance for the two systems.

The first step towards achieving our goal is to identify Big Data application areas. We identify three large areas: graph processing, machine learning, and scientific computing. For each of the three areas, we identify representative applications or benchmarks, and execute them on all the three systems with increasingly larger input data and increasing number of processing nodes.

For graph processing, we experiment with the Graph500 benchmark [33] and the Apache Giraph [1] graph processing library. The Graph500 benchmark consists of an implementation of the Breadth First Search algorithm on top the MPI system. We run the Apache Giraph implementations of the following three algorithms: Breadth First Search, Shortest Paths, and Page Rank. For machine learning, we run three classic machine learning algorithms implemented using the Apache Mahout [2] library, specifically: K-Means clustering algorithm, Naive Bayes classification algorithm, and Stochastic Gradient Descent classification algorithm. For scientific computing, we run the Standard Science DBMS Benchmark (SS-DB) [18] on top of the SciDB [17] database management system.

For each application, we start with small data size and increase it until we get to the point where the experiment either times out (i.e., runs longer than one hour), or the application crashes for some reason (e.g., the system runs out of memory). For graph processing and scientific computing, we use synthetically generated input data, whereas for machine learning, we use real-world datasets.

For machine learning and scientific data, we use fixed clusters summing 48 cores in total, whereas for graph processing, we repeat the experiments with clusters of 8, 16, 32, 48, and 64 cores, respectively. The metric we use in all our experiments is the execution time. For each experiment, we represent the execution time as a function of the input data size and the number of cores. We first identify the maximum data size that can be processed on each system using a specific number of cores. Second, we analyze the differences in execution between the three systems and try to identify what system characteristics cause one system to perform worse than another. Third, we discuss how the application scales with the number of cores and the input data size, and try to predict what data sizes would the application be able to process on clusters larger than the ones we run our experiments on.

From the results of our experiments, we find that the cluster system, DAS4, per-

forms significantly better than the two cloud systems: up to 11 times faster for graph processing, and up to 2 times faster for machine learning and scientific computing. We attribute the overhead mostly to the network virtualization and less to CPU virtualization. We find that a cluster of 64 cores can process graphs with about 130 millions vertices (about 30GB) and can run machine learning algorithms processing about 20 millions text documents (about 100GB). For scientific computing, we find that datasets of 100GB can be queried in less than one minute, and we estimate that the execution time for querying 1TB of data would also be in the order of minutes. We observe that some algorithms scale poorly with the number of cores and should one run them on larger clusters, they would not be able to process significantly larger data. We find that in some cases, this limitation comes from the algorithm's nature (e.g., Stochastic Gradient Descent algorithm), while in others, the parallel paradigm used for implementing them causes the limitation (e.g., Shortest Paths implemented with Giraph). We find also that such algorithms as Breadth First Search or Naive Bayes implemented with MapReduce, scale linearly with the number of cores, which means that by simply adding processing nodes, one would be able to process ever larger data.

In Chapter 2, we present the work related to the research question we address. We start by introducing Big Data and cloud computing. After that, we present the Big Data sub-domains. Then we present what has been done so far regarding stress-testing and benchmarking cloud systems with Big Data applications.

In Chapter 3, we start by presenting in general terms our approach to stress-testing and benchmarking clouds with Big Data applications. After that, we present in more details the three Big Data application domains: graph processing, machine learning, and scientific computing. For each domain, we start with a generic introduction. After that, we describe each representative application we selected for that domain, specifically, for graph processing, we describe Graph500 and the three algorithms implemented with Apache Giraph, for machine learning, we describe the three algorithms implemented with Apache Mahout, and for scientific computing, we describe the SS-DB benchmark and the SciDB database management system. After that, we present the experimental setup for each application, including the clusters we use and the datasets.

In Chapter 4, we present the results of running each selected application. For each application, we present charts of the execution time on each of the three systems. We identify the exact overhead between the systems, the maximum data size that can be processed by the application given a cluster size, we discuss the limitations of each application, and we try to identify the causes for the limitations.

In Chapter 5, we start by providing a summary of the report. Then we present the most important conclusions, and finally, we list a number of ideas for future work as a follow-up of this thesis.

Chapter 2

State of the Art

In the last several years, an increasing interest in the fields of Big Data and cloud computing has emerged in both academia and industry. In this chapter, we provide an introduction to the cloud computing and Big Data and provide an insight in what has been done in the direction of stress-testing and benchmarking clouds with Big Data applications. Having a clear view of the state of the art both helps us clearly define the problem we tackle in this thesis and provides us with a starting point on how to approach the problem.

In Sections 2.1 and 2.2, we provide an introduction to the fields of cloud computing and Big Data. In Section 2.3, we present Big Data application areas. In Section 2.4, we describe two major large-scale data analysis paradigms: MapReduce and Parallel Database Management Systems. In Section 2.5, we present the efforts that have been done so far in order to benchmark and stress-test clouds with Big Data applications.

2.1 Introducing Cloud Computing

Cloud computing has been a major shift in how software services are developed and how these services are offered to users. Traditionally, companies used two methods to offer software to their users. The first one involves providing the software as a package and the user executes the software on her local computer. In the second case, the company deploys the software on its own computing resources and offers the user access to the service via the Internet. With cloud computing, companies (i.e., cloud users) can rent computing resources from cloud providers and develop services on top of those resources. The important part is the cloud users can provision as many computing resources as they need at each certain moment and pay by hour. This way, the companies solve several problems regarding buying their own hardware. First, they do not risk buying more hardware than their business would actually scale up to. Second, they do not risk that their hardware may become insufficient in case of an unpredicted growth of user's demand. Third, companies do not have to setup and maintain their own node cluster. The cloud provider gives

the user the impression of an infinite pool of computing resources and assures the user of the resource availability.

There exist three types of cloud services:

- *Infrastructure as a Service (IaaS)* is the core service. It provisions computing nodes (often in form of virtual machines), storage, network bandwidth and tools for building an application from scratch. The cloud user can provision virtual machines on demand and pay-per-use. Although this service gives the highest flexibility, it gives the user the burden to deal with low level details, such as load-balancing the application and making it elastic by provisioning machines on demand. Examples of IaaS services are: Amazon EC2 and GoGrid for virtual machine provisioning, Amazon S3 and EBS for storage.

- *Platform as a Service (PaaS)* is the cloud service that is often built on top of IaaS. This service provides a higher level platform on top of which applications can be built. The cloud provider is responsible for managing, scaling-out and load-balancing the platform. Examples of PaaS services are Google AppEngine and Microsoft Azure. The former uses Python, Java and Go as application development languages, whereas the latter is based on .Net languages.

- *Software as a Service (SaaS)* is the highest level cloud service. It is special-purpose software available through the Internet. Examples of SaaS are Google Maps, Google Translate, Salesforce. These services are not suited for building user applications.

Armbrust et al. [8] present a larger view of cloud computing. Binning et al. [12] present the needs to create a cloud benchmark. These two papers present more in depth introduction of cloud computing and the motivation behind developing a benchmark for clouds.

2.2 Introducing Big Data

Big Data has attracted a lot of interest lately. However, the term has not been properly defined and it often causes confusion about what it actually means. Jacobs [28] discusses what Big Data is, what its challenges are, and how to deal with them.

2.2.1 Defining Big Data

Jacobs [28] states that the term *Big Data* is not new and he provides some examples of datasets that have been considered huge in the past. One of these examples is the IBM 3850 Mass Storage System, which was developed in the late 1980s and was able to store 100GB using thousands of magnetic tapes. This dataset size, as many others, are considered small nowadays. This is due to the evolution of hardware and software, which give us the power to process larger and larger data as the time

passes. This means that the definition of Big Data shifts continuously. The author tries to provide a meta-definition of Big Data. He states that 'Big Data should be defined at any point in time as data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time'. While the definition shifts, one fact will remain true: the most successful developers will be the ones who are able to look past the existing standard techniques and understand the true nature of the hardware and algorithms that are available to them.

Following his meta-definition of Big Data, Jacobs [28] states that nowadays Big Data are the datasets that are too large to fit into a relational database and to be analyzed using a desktop statistics package. Analyzing such data requires massively parallel software that runs on tens, hundreds, thousands or more computing resources.

Gantz and Reinsel [23] provide an analysis of the growth of the worldwide data. They state that the total amount of digital information created and replicated in 2009 was 800,000 petabytes. They estimate that by 2020 the data will be 44 times as big as in 2009. Although it is good to know the total amount of data in the world, one would like to know what is the size range of the datasets used in Big Data applications. Datasets in order of tens or hundreds of gigabytes are considered to be small for Big Data systems. For example, the Graph500 [33] benchmark generates datasets of custom sizes and it labels the dataset with the size of 1TB as *small*. Common Big Data datasets are in the range of terabytes and increasingly more in the range of petabytes. For example, Graph500 [33] generates datasets of up to 1.1PB. Another example is the Wikipedia dataset [6], consisting of more than 700GB of compressed text. For large companies, it is common to analyze datasets in the order of petabytes. Facebook's data warehouse was storing and analyzing 2.1PB of data in 2010 [38].

2.2.2 Big Data Issues

Jacobs [28] aims to identify what makes the Big Data so difficult to analyze. He presents four issues regarding Big Data analysis and provides ideas on how to deal with them. He calls these issues *pathologies* from which Big Data suffers.

The first issue is the poor handling of temporal dimension of the data by the nowadays databases. In order to understand this issue, one first needs to understand what makes Big Data so *Big*. The author states that this is due to the repeated number of observations over time and/or space, not due to the cardinalities of the observed entities. For example, a neurophysiology experiment uses only hundreds of sensors. However, due to the large sampling rate of the sensors, the generated dataset can result in gigabytes of data. Another example is the click stream of a set of Web pages. Although the number of Web pages may be relatively small, the Web log may record millions of visits every day. As a result of repeated observations, most datasets have a temporal dimension. Therefore, Big Data analysis will usually involve some ordering in order to understand the past and be able to predict the future. Most of the existing databases do not handle the temporal dimension

well. The most notorious databases that ignore temporal dimension are the relational databases, i.e., tuples are saved independently of the order in which they are inserted in a RDBMS table. When a query orders a RDBMS table by time, random accesses to disks happen, which results in degradation of performance compared to when the disk is accessed sequentially. In order to deal with this issue, one may want to develop a database that recognizes the concept of inherent data ordering down to the implementation level.

The second issue comes from the hard limits many applications have regarding the size of the data they can analyze, that is, they cannot scale above a certain limit. Such examples are the applications that load all the data in memory before processing it. Furthermore, some applications are not even able to use all the memory, such as 32-bit applications running on 64-bit platforms. They can only use at most 4GB of memory even if the computer may be equipped with more than that. Other examples are 64-bit applications that by their design can only access a part of the memory (e.g., using 32-bit integers to index array elements). In order to face this issue, developers should design their software such that it handles hardware improvements.

Any computer has some absolute limits: memory, disk space, processor speed, etc. When all these limits are exhausted, the only possible way to analyze large-scale data is to use multiple computing devices. The fact that the mainstream computer hardware is cheap and infinitely replicable further confirms that distributed computing is the most successful strategy for analyzing very large data.

The third issue Jacobs [28] describes, is nonuniform distribution of work across processing nodes. This happens when the data are wrongly distributed across nodes. When multiple queries are run on data, it is possible that distributing data in one way may distribute the work well for some queries and badly for others. Moreover, in distributed analysis, processing must contain a component that is local in the data and can be computed on each node without the need of accessing chunks of data residing on different nodes. Even if some aggregation is needed, it is important that it happens after all nodes have performed their local aggregations. Fortunately, the most common aggregations (e.g., summation) do fulfill this property.

The fourth issue is the reliability of the systems that analyze Big Data. For small data analysis software, one computer often suffices. In this case the probability of failure is small and regular backups solve the reliability issue. On the other hand, when analyzing large scale data, a large number of computing nodes need to be used, which increases the probability of node failures. Therefore, the analysis framework has to be able to recover from hardware failures transparently to the user. An idea is to replicate the data in order to prevent data losses when a computing node fails.

Gopalkrishnan et al. [24] present an important challenge: how to make Big Data analysis easier for companies to adopt. The authors state that while some big companies have adopted Big Data analysis, many others do not know whether they should switch from traditional data analysis to Big Data analysis and whether this

investment will bring them enough benefit.

Gopalkrishnan et al. [24] propose three questions companies should ask themselves before investing in Big Data analysis. First, what is the business problem and the organizational goal? Sometimes, the strategy of the business simply cannot benefit from Big Data analysis. For example, if an insurance company wants to update its fees daily based on daily fluctuations, then a real-time Big Data approach would help. However, if the insurance company wants to just set a six months insurance policy, then a traditional data mining would be enough for determining the best price. Second, given the goal, is the available data suitable? Maybe the owned dataset is too small or does not have enough information for Big Data analysis to mine valuable facts. Third, what is the return of investment on Big Data? Sometimes, implementing a Big Data approach is too costly compared to the benefit it would actually bring. These costs need to be carefully calculated before deciding whether to use Big Data analysis.

2.3 Big Data Application Areas

We have not found a clear classification of Big Data applications in the literature, but we gathered a list of domains where Big Data is widely used [33, 14, 24, 40]. These domains are not necessarily disjoint. The following are the application areas we identified:

- *Graph Processing.* There exist several subareas where graphs with different properties are used. Social Networks involve graphs where millions of users are connected using different kinds of relations. Medical Informatics use graphs of patient records for entity resolution.

- *Scientific Computing.* Mines data resulting from physics, medicine, astronomy, etc. These data tend to be structured and floating point operation intensive.

- *Machine Learning.* Data mining is used to return the best result for user's demand, based on his or other users' observed behavior. Machine learning is often used to recommend products or display relevant advertisement to users. Many machine learning systems need to return a result in real-time.

This classification lies at the ground of this thesis. For each area, we need to identify at least one representative application and run it on different cloud systems. The purpose is to both assess the performance differences between these systems and to detect what is the maximum dataset that can be handled by the application on each system. Moreover, we want to detect what specific components (e.g., CPU, memory, network) of the system does each application stress the most.

2.4 Big Data Analysis Paradigms

In this section, we start by describing what properties a large-data analysis system should have. Next, we present two major general purpose Big Data analysis paradigms: MapReduce and parallel databases.

2.4.1 Properties of Big Data Analysis Systems

Cohen et al. [16] discuss the traditional analysis techniques and their drawbacks considering the properties of nowadays large-scale datasets. They discuss the emergence of *Magnetic, Agile, Deep (MAD)* data analysis. They state that these are three important properties a Big Data analysis system should have.

Magnetic: In traditional Enterprise Data Warehouses and Business Intelligence, before mining the data, they first have to be moved from the databases operating directly with the user to the data warehouse. Next, the data are analyzed in the warehouse using On-Line Analytic Processing (OLAP) [15] and Data Cubes [25]. Before analysis, the data have to be cleansed and changed conforming to a certain schema. This analysis technique is unsuited for many datasets, because the data tend to be unstructured, uncertain and usually have information misses. Prominent examples are datasets collected from Web. As said above, the data warehouses reject new data sources until the data are cleansed and integrated. This results in a high effort to load the data in the system. A system is called magnetic, if it crops together any number of data sources, regardless of the data quality niceties. Ideally, the system would just be able to parse bulks of input data stored in conventional files.

Agile: Traditionally it takes a long time to carefully design and plan a data warehouse. Therefore, it takes a long time to modify the warehouse when new data sources need to be integrated. Furthermore, in enterprises, data warehouses are centrally administered by an assigned department, which makes it difficult for other departments to change the warehouse based on their changing needs. Considering the low cost of nowadays commodity hardware, any department would now be able to setup a cluster of servers and analyze its data locally. An agile data analysis system supports fast changes when new data sources are added. Moreover, it makes it easier for non specialized engineers to maintain the warehouse.

Deep: Modern data mining involves increasingly sophisticated statistical methods that go beyond traditional roll-ups and drill-downs performed on data cubes. Moreover, it has been claimed that SQL has a limited expressive power, making it difficult to implement some data mining methods. Therefore, a modern data warehouse has to be both a deep data repository and a sophisticated data analysis engine.

2.4.2 Major General Purpose Big Data Analysis Paradigms

Relational databases have been popular since the 1980s. In order to cope with large-scale datasets, already in the late 1980s, parallel implementations have been created that were able to run on clusters of shared nothing nodes. Lately, Google's MapReduce has been gaining significant adoption as a large-scale data analysis paradigm. Pavlo et al. [36] thoroughly compare the two paradigms. Below, we first introduce the two paradigms, and then we report the findings reported by Pavlo et al. [36].

Parallel Databases

All the parallel DBMSs support standard relational tables and SQL. Two key aspects enable parallel execution. First, the tables are partitioned across the nodes. Second, the system contains a query optimizer that plans the execution of a query across the cluster. The parallel nature of these systems is completely transparent to the developer, as she only needs to provide the query in the SQL high level language. The developer is not concerned about the underlying implementation details, such as indexing or joining strategies.

MapReduce

MapReduce [19] is a programming model introduced by Google. One of the most important properties that made it so popular, is its simplicity. A MapReduce program consists of only two functions: *Map* and *Reduce*. The developer defines these two functions using general purpose programming languages and the underlying framework is able to automatically parallelize the computation across arbitrary large clusters of shared-nothing nodes.

The input data are stored in a distributed file system deployed on the cluster and the framework injects the Map and Reduce functions to each node. The input data consists of a list of key/value pairs. Map and Reduce are the two phases of the computation. Both Map and Reduce phases spawn multiple Map and respectively Reduce instances on multiple nodes. Each Map instance function receives a part of the input list and generates an intermediary list of key/value records. Next, from this intermediary list, the framework creates groups with records that have the same key and feed each group to a Reduce instance. The Reduce phase generates the output list of key/value records. It is often the case that multiple MapReduce iterations are needed to run a computation.

Comparing MapReduce with Parallel DBMS

Pavlo et al. [36] start by providing in depth architectural details of the two data analysis approaches and use these details to explain the performance differences between the two. The following is an outline of the main architectural differences:

- MR does not comply to a schema, whereas DBMSs do. The burden to parse the input data in MR may result in the same effort as one needed to structure the data for DBMSs.

- MR uses generic programming languages, whereas DBMSs use SQL. In MR it is often needed to manually implement joins or filters. To solve this, higher level languages as Pig-Latin [34] and Hive [4] have been added on top of MR.

- Parallel DBMSs push the query to the data, whereas MR pushes the data to the computation tasks. Because MR lacks a query optimizer, often unneeded data are transfered in the network.

- Reduce functions use pull protocols to get the input data they need. If many Reduce instances need an input from the same Map instance, multiple accesses to the same disk are being performed, resulting in a poor performance. Parallel DBMSs use push protocols instead to solve this issue.

- MR supports programming languages that are more expressive than SQL. However, multiple extensions have been added to SQL to support such features as object-relational mappings or the ability to define procedures directly in SQL.

- Failures are better handled in MR. If a node fails, only the instances that were running on that specific node need to be restarted. Because parallel DBMSs support transactions, the whole query needs to be recomputed if at least one node fails. This proves to be an important problem in clusters with thousands of nodes, where failures are very probable.

Pavlo et al. [36] have performed several experiments in order to assess the performance of the two data analysis approaches. They compare Hadoop, the most popular open-source implementation of MapReduce, with two parallel DBMSs: DBMS-X and Vertica. They have run their experiments on a cluster of 100 nodes and a generated dataset of at most 2TB.

The authors run five tasks on each system. The first one looks for a certain word in the dataset. The other four tasks are four queries on a dataset of Web pages. Three of these represent a selection, an aggregation and a join, respectively. The last and the most complex task computes for each Web document the number of unique Web documents that link to it. This last task is called the UDF aggregation task and it is believed to be one of the most representative tasks for MapReduce.

Hadoop performs the best at loading the data, because this involves only copying the input files in the distributed file system, whereas for parallel DBMSs this also involves additional tasks such as indexing. On the other hand, the two parallel DBMS perform better at querying the data. Excepting the UDF aggregation tasks, Hadoop is performing the worst from the three systems. On average, DBMS-X was 3.2 times faster than Hadoop and Vertica was 2.3 times faster than DBMS-X.

The authors claim that in order to use parallel DBMSs on clusters of thousands of nodes, it is necessary to improve the fault tolerance of these systems in order to prevent performance decrease caused by more often failures. However, for datasets in the order of petabytes the same performance achieved by MR on a cluster of hundreds of nodes can be achieved by parallel DBMSs on just tens of nodes. For this cluster size, the parallel DBMSs perform well enough with their current failure tolerance. The authors provide examples of existing systems that are able to analyze datasets in the order of petabytes on clusters of tens of nodes [16].

This paper leaves an important open question: how would parallel DBMSs behave for datasets larger than a few petabytes on clusters larger than tens of nodes? Would MapReduce surpass them?

2.4.3 Specialized Big Data Analysis Systems

The two data analysis approaches presented in the previous section, MapReduce and parallel DBMSs, are general purpose systems, i.e., they can be used for a large range of applications. There exist however Big Data analysis systems specially designed for specific application areas. Some application specific systems have been implemented on top of MapReduce.

Graph algorithms are particularly hard to parallelize. Although graph algorithms have been implemented using MapReduce, there exist specialized frameworks for graph processing. Google's Pregel [31] is a vertex centric framework, where the program consists of a repeated iterations. The developer only has to implement a function that will be executed by each vertex. Parallel Boost Graph Library [27] is another graph processing framework, developed using MPI [22].

Monte Carlo Database (MCDB) [40] is a database implemented on top of MapReduce that is able to manage uncertain data. It is able to generate real-world data for tuples that have missing values for some attributes. Moreover, it can generate relevant data for missing tuples. When a query is executed, MCDB receives multiple possible results and the probability for each result.

2.5 Benchmarking Systems for Big Data Applications

Although lately Big Data applications have become increasingly popular, no benchmark has been defined yet that would assess how suitable are existing computation systems for various Big Data applications. Below, we present some recent work that concentrates on defining such a benchmark.

2.5.1 Towards a Big Data Benchmark

General Benchmarking Guidelines

In order to correctly design a benchmark, one needs to gain general knowledge about the process of benchmarking. Jain [29] provides practical guidelines for

analyzing the performance of computer systems. In Chapter 9 of his book, Jain [29] discusses *capacity planning* and *benchmarking*. Capacity planning is the problem of ensuring that the adequate computer resources will be available to meet the future workload demands. Benchmarking, the process of running benchmarks on different systems, is the most common method to compare the performance of the systems for capacity planning. A benchmark is the act of running one or more computer programs with a predefined set of workloads and measuring one or more performance metrics. In order to be reliable, a benchmark needs to have several characteristics:

- The workloads need to be representative for the real-world workload. Moreover, the workload should not only represent the average behavior. The chosen distribution of the workload events should resemble the distribution of the real workload.
- The monitoring overhead should be minimal. If this is not the case, the benchmark should properly treat this overhead rather than ignoring it.
- If the executed workload is non-deterministic (some randomness is involved), the benchmark should run the same workload multiple times and compute both the mean (or the median, as appropriate) and the standard deviation of the performance metric.
- The initial conditions should be the same for all the benchmarked systems.
- The benchmark should not ignore the skewness of the device demands. For example, some benchmarks erroneously consider that the I/O requests are evenly distributed across all I/O devices, therefore they are not able to reproduce the real-world scenarios when many I/O requests target only one I/O device, leading to queueing bottlenecks.

The Workshop on Big Data Benchmarking

The emergence of Big Data has introduced the need of a specialized benchmark for Big Data systems (BD benchmark). To develop such a benchmark, the *Workshop on Big Data Benchmarking (WBDB)* has been created. From May 2012 to July 2013, three editions of this workshop were held. The first workshop [11] focused on identifying the issues of Big Data and how would they influence the nature of the BD benchmark. The following seven issues have been discussed:

1. Existing benchmarks, such as TeraSort [35] and Graph500 [33], are designed for specific operations and data genres, and these benchmarks are therefore not suited for Big Data. The workshop participants agreed that a BD benchmark should focus on assessing end-to-end applications rather creating micro-benchmarks for separate components.

2. A BD benchmark should use a range of data genres, such as structured, semi-structured and unstructured data; graphs as they occur in different domains; streams; geospatial data; array-based data; and special data types such as genomic data. For each data genre, the core set of operations needs to be identified.

3. In non-BD benchmarks, the scale of the setup used for testing a system is often larger than the scale at which the customers actually use the system. In Big Data however, the system under test is actually smaller in size and configuration than the actual customer installation, given the high cost to setup an installation close to customer's. Therefore, a BD benchmark should be able to model a system's performance at a larger scale based on the measurements at a smaller scale.

4. Another key aspect to be considered is testing elasticity and durability, i.e., the ability to gracefully handle failures. The system needs to be tested under dynamic conditions, i.e., when the workload increases and nodes are added or removed due to hardware failures.

5. The workshop participants agreed that a BD benchmark should be technology agnostic. The benchmark should be specified in a language that can be easily understood by a non-technical audience. Using English provides much flexibility and a broader audience, while some specification parts would be defined in a declarative language like SQL.

6. Another important issue is what data sources to use. Using real-world data is unpractical, because it requires downloading and storing extremely large datasets. Moreover, real datasets may reflect only certain properties of Big Data and not others. Therefore, participants agreed that the dataset should consist of generated synthetic data. In order to generate such large datasets, parallel data generators [26, 7] need to be used.

7. The benchmark should include both performance and cost-based metrics. Cost-based metrics would measure price/performance ratio of the system. Moreover, other interesting cost metrics are: setup costs, energy costs, total system costs.

In the second and third Workshop on Big Data Benchmarking [10], the participants defined the characteristics of the BD benchmark and discussed two proposals for defining it. They found four characteristics:

1. *Simplicity*: The benchmark should be easy to implement and execute.

2. *Ease of Benchmarking*: The cost of benchmark implementation/execution and audits should be kept low.

3. *Time to Market*: The interval between benchmark versions should be short enough to keep pace with the rapid market changes in the Big Data domain.

4. *Verifiability of Results*: The BD benchmark should provide automatic means of result verification in order to make result audit cheap.

The following two proposals for the BD benchmark have been defined:

- *Deep Analytics Pipeline*: A broad range of data-driven industries try to learn *entity* behavior and their *events of interests* based on activity history of the entity. For example, online advertising industry tries to predict what online advertisement (the event of interest) a user (the entity) will click based on her previous activity. This type of use-cases involves first collecting a variety of datasets for the entity of interest and then correlate the past behavior with the outcome of interest. This can be modeled as a pipeline and can be used as the workload of the BD benchmark. The proposal enumerates seven steps the pipeline should have:

  1. Collect ”user” interaction data.
  2. Reorder events according to the entity of interest.
  3. Join ”fact tables” with other ”dimension tables”.
  4. Identify events of interest that one plans to correlate with other events in the same session for each entity.
  5. Build a model for target events based on previous behavior.
  6. Score the model on hold-out data from the initial dataset.
  7. Apply the model to the initial entities that did not result in the target event.

- *BigBench*: This approach is based on extending the TPC-DS benchmark [37] with semi-structured and unstructured data and altering this benchmark to incorporate queries targeting these types of data. The queries include both traditional SQL queries as well as procedural queries that are hard to define directly in SQL.

In our work, we make sure to comply with Jain's [29] general benchmarking guidelines and to learn from the discussions at the Workshops on Big Data Benchmarking [10] [11].

2.5.2 Previous Efforts Regarding Big Data Benchmarking

Although a BD benchmark has not been defined yet, some work has been done to assess the performance of Big Data applications. In this section, we present some of this work.

Evaluating MapReduce with Workload Suites

Chen et al. [13] introduce a method for assessing MapReduce installations using workload suites generated based on production MapReduce traces.

Existing MapReduce benchmarks, such as Pigmix [5] and Gridmix [21], stress test the cluster with large datasets using a small set of "representative" computations. These computations do not necessarily reflect the real-world MapReduce installations. Chen et al. [13] analyze two MapReduce traces, one from Yahoo! and one from Facebook. They use a machine learning clustering algorithm for detecting job categories. The job features they use for clustering are the following: the input data size, the output data size, the data size between map and reduce phases, map phase running time, reduce phase running time, and the total running time of the job. They detect several categories the jobs from these traces fall into. Some examples of the categories they found are: *aggregation jobs* - these have large input data and small output data; *small jobs* - these have a short total running time; *load jobs* - these have small input data and large output data. The authors argue that some of these job categories are not represented by any of the existing MapReduce benchmarks, which makes these benchmarks unsuited for real-world MapReduce applications.

The authors provide an algorithm for generating workloads from a MapReduce trace. They describe a synthesizer they developed using this algorithm. The algorithm divides the time interval of the trace into a sequence of non-overlapping segments. It randomly chooses a subset of these segments and puts them together in order to create a synthetic workload. The authors prove that the resulting workload is representative for the real-world trace and that their synthesizer introduces a low execution overhead.

Next, Chen et al. [13] demonstrate that workload suites provide cluster administrators with new capabilities. They present two new capabilities they found. First, by running particular workloads one can detect workload-specific bottlenecks. Second, one can identify what MapReduce scheduler suits better each specific workload. This approach has two drawbacks. First, one needs to have a trace at hand in order to generate workload suites. Second, the Map and Reduce functions implemented by the synthesizer do not have the same semantics as the functions that generated the trace.

Graph500

As said in Section 2.4.3, graph algorithms are particularly difficult to parallelize, because they exhibit poor data locality, requiring "global reach", and therefore result in many messages being sent across the cluster. Graph500 [33] is a benchmark for systems that analyze large-scale graphs.

The Graph500 benchmark was inspired by the LINPACK benchmark that defines the Top500 list. The Top500 benchmark assesses the CPU performance of the

systems, having the number of floating point operations per second as performance metric. Unlike the Top500, the Graph500 assesses the communication subsystem of the system. Its performance metric is the number of traversed graph edges per second. We run this benchmark in our experiments, therefore we discuss it in more detail in Section 3.3.2.

Chapter 3

Our Approach to Benchmarking Cloud Systems with Big Data Applications

The purpose of this project is three-fold. First, we aim to compare the performance of one cluster system and two cloud systems using different types of Big Data applications. Second, we aim to stress-test these systems, that is, we try to identify what is the maximum dataset size that can be processed by these three systems for each Big Data application type. Third, we try to identify what are the bottlenecks that limit each application from being able to process even larger datasets, e.g., is the nature of the algorithm not scalable, is the CPU, the memory or the IO bandwidth insufficient? In this chapter, we describe our approach towards achieving these goals. We present the systems we benchmark and the applications that we run on these systems. In Chapter 4, we describe the results of the experiments and discuss the outcomes.

In Section 3.1, we describe the three systems we benchmark. Section 3.2 describes our approach to stress-testing clouds with Big Data applications and enumerates the three Big Data areas we use for stress-testing. In Sections 3.3, 3.4, 3.5, we present how we stress-test using Graph Processing, Machine Learning, and Scientific Computing applications, respectively.

3.1 Benchmarked Cluster and Cloud Systems

The performance of an application is strongly influenced by the machine it is being run on. On a cloud system, not only does the hardware configuration of a machine count, but also the virtualization can lead to further decreases in performance. In our work, we aim to assess different types of clouds and see whether there are significant differences in performance for several types of Big Data applications. We ran our experiments on three systems: the *Distributed ASCI Supercomputer 4 (DAS4)*, OpenNebula virtual machines on top of DAS4, and Amazon Web Services.

In the following section, we describe each of the three systems and present our motivation for running our experiments on each of them.

3.1.1 DAS4

The Distributed ASCI Supercomputer 4 [9] is a wide-area distributed system located in the Netherlands. Its goal is to provide a common computational infrastructure for researchers involved in parallel, distributed, grid and cloud computing. DAS4 involves six participating universities and organizations. This divides DAS4 in six clusters. We performed our experiments on the Delft University of Technology (TUDelft) cluster and on the Vrije Universiteit Amsterdam cluster. The clusters consist of a number of standard computing nodes and some special purpose computing nodes (e.g., nodes with GPUs, nodes with extra memory). In this section, we present the configuration of the standard nodes. In case we use a non-standard node in one of our experiments, we present the configuration of that node type in that specific experiment's section.

The standard node has the following configuration: dual quad-core 2.4GHz CPU, 24GB of memory, 18TB of disk storage. In our experiments we use up to 16 nodes at a time. The connectivity between nodes is realized via Ethernet (1GB/s) and InfiniBand. The operating system is CentOS Linux.

As said before, DAS4 is not a cloud system, that is, the applications run directly on physical machines without any virtualization involved. Running our experiments on DAS4 before running on clouds provides us with a reference on the performance of the application on a physical machine. This allows us to assess the influence of virtualization on the performance of the application.

3.1.2 OpenNebula on top of DAS4

OpenNebula [32] is a cloud management system that can be used for leasing and managing virtual machines. DAS4 has an OpenNebula installation that can be used to lease up to eight virtual machines at the TUDelft site.

By default, the OpenNebula installation starts only one virtual machine per physical machine, that is, the virtual machine does not share the physical machine's resources with other virtual machines. One can configure how much memory and what percentage of the host's CPU to allocate to the virtual machine. In order to have a reliable comparison between the performance of the applications on raw DAS4 and on OpenNebula, we made sure that each virtual machine uses as much as possible of its physical machine's resources. This ensures that the virtual machine has almost the same hardware configuration as its physical machine. If differences in performance occur, they can only be caused by the virtualization layer. Therefore, we configured our virtual machines to use 100% of the host's CPU (dual quad-core 2.4GHz) and 20GB of memory (out of 24GB of the physical machine). The operating system we used on the virtual machines was Debian Linux.

3.1.3 Amazon Web Services

Amazon provides a large range of Web Services. Amazon Elastic Compute Cloud (EC2) is the service for leasing and managing virtual machines. It provides a large range of virtual machine configurations. The purpose of running our experiments on AWS was to compare the differences in performance between OpenNebula and AWS virtual machines. There are several possible sources of performance differences. First, there may be differences in the virtualization types used. Second, the OpenNebula virtual machines on DAS4 do not share physical machine's resources with other machines, whereas on AWS most probably more than one virtual machines are leased on the same physical host.

For our experiments, we used *m1.xlarge* instance types, because their configuration is the closest to the configuration of the DAS4 nodes. The *m1.xlarge* instance configuration is as follows: quad-core CPU with 2 EC2 compute units (ECU) each, 15GB of memory, 1680GB of disk storage. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. The operating system running on the virtual machines was Ubuntu 12.10 Linux.

3.2 Describing our Approach to Big Data Benchmarking

When defining our approach to stress-testing Big Data applications, we had as a reference the discussions [10] [11] at the Workshops on Big Data Benchmarking, that aimed to set the foundation of a Big Data benchmark (see Section 2.5.1).

One of the key points was to use real world end-to-end applications that use a wide range of data genres. Our first milestone was to identify the Big Data areas, such that we make sure we cover the most of the use cases in Big Data applications. In Section 2.3, we identified three Big Data areas: graph processing, machine learning, and scientific computing. For each of these areas, we searched for existing open source applications or benchmarks that we can run on all the three systems we listed in Section 3.1. It was important that each application comes with a dataset (either real-world or synthetically generated) whose size can be varied. For each application we increase the data size until the cloud system under test cannot handle it anymore (either the execution takes too long or the system fails for some reason). In the following sections, we describe the applications we chose to run for each Big Data area.

3.3 Graph Processing

In order to stress-test DAS4, OpenNebula and AWS for graph processing, we chose the Graph500 [33] benchmark and the graph applications that are shipped together with Apache Giraph [1] graph processing library. Several graph algorithms are employed by Graph500 and Giraph: Breadth First Search, Shortest Paths, and Page

Rank. Below, we first provide a general description of these algorithms, and then we describe Graph500 and Giraph.

3.3.1 Graph Algorithms

Breadth First Search This algorithm traverses a graph in breadth starting with a given source vertex and computes a traversal tree. In an unweighted graph, this algorithms finds the shortest path from the source vertex to any other vertex. Sequentially, this algorithm is implemented using a queue. First the source vertex is queued. The algorithm iteratively removes a vertex from the top of the queue, visits all its unvisited neighbors and adds them to the queue. The algorithm ends when the queue becomes empty.

On distributed systems, this algorithm can be implemented at least in two ways:

- *Message Passing model*: In this model, the computation consists of a number of processes that communicate between each other using messages. MPI [22] is the most notorious system used for developing message passing distributed programs. BFS is implemented on MPI using two queues for each process: an *old* queue and a *new* queue. Each MPI process is the owner of a subset of vertices and all the outgoing edges from those vertices. The algorithm executes in iterations. At the beginning of an iteration, each MPI process has an *old* queue with its own vertices that have just been visited. For each vertex in its *old* queue, the MPI process adds to the *new* queue all its unvisited neighbors that the process owns. For each unvisited neighbor vertex that it does not own, the process sends a message to the process owning that vertex. After the local process finishes processing the vertices in the *old* queue, the MPI process swaps the *old* queue with the *new* queue. Then, it receives messages that have been sent to it by other MPI processes. If the owned vertex, for which the process receives a message, is unvisited, the process marks that vertex as visited and adds it in the *old* queue.

- *Bulk Synchronous Parallel (BSP) model [39]*: A BSP computation consists of a number of *supersteps*. The developer implements a computation function to be executed by each vertex of the graph in each superstep. At each superstep, each vertex executes its computation function and optionally sends/receives messages to/from other vertices. The end of the superstep serves as a synchronization barrier. At the end of the superstep, the messages are delivered to their destination vertices. In the next superstep, the vertices can process the messages that have been sent to them in the previous superstep. In any superstep, a vertex can opt to halt. If a halted vertex receives a message, the system makes it active again. The computation ends when all the vertices halted and no messages exist in the system.

  BFS starts with all the vertices unvisited. In the first superstep, the source vertex marks itself as visited and sends its id to all its neighbors. At each

superstep, an unvisited vertex checks whether it received any messages from its neighbors. If it did, it reads one message, marks itself as visited, remembers the received id as its parent in the BFS tree, and sends its id to all its neighbors. The algorithm ends when no messages are sent anymore.

Shortest Paths This algorithm computes the shortest path from a source to any other vertex in a weighted graph. This algorithm is implemented in BSP as follows. At any point in time, each vertex knows the latest discovered shortest path from the source to it. The initial shortest path for the source is zero, while for the other vertices is infinity. In the first superstep, the source sends to each neighbor its shortest path (viz., zero) plus the weight of the edge between them. In each superstep, a vertex can receive a message from a neighbor with the shortest path to that neighbor plus the weight of the edge between them. If the newly received shortest path is smaller than the value stored by the vertex, it stores the new value and sends to all its neighbors its shortest path plus the weight of the edge between them. The algorithm finishes when no update of the shortest paths happens.

Page Rank This algorithm measures the relevance of each vertex based on the weight of the vertices that have an edge pointing to it. This algorithm was invented by Google to weigh the importance of each Web page in the World Wide Web. The algorithm runs in iterations. All the vertices begin with a fixed page rank. At each iteration, each vertex receives the page ranks of the vertices pointing to it, and computes its new rank r using the following formula:

$$r = 0.15/n + 0.85 \cdot (\sum_j r_j),$$

where n is the number of vertices in the graph and r_j are the ranks received from the previous iteration. The algorithm finishes when the ranks converge or when a maximum number of iterations has been reached.

3.3.2 Graph500

As said in Section 2.5.2, the Graph500 [33] benchmark stresses the communication subsystem of the system under test. The benchmark provides six reference implementations:

- serial high-level in GNU Octave
- serial low-level in C
- parallel C version with usage of OpenMP
- two versions for Cray-XMT
- basic MPI version (with MPI-1 functions)

- optimized MPI version (with MPI-2 one-sided communications)

We use the MPI implementation because it is the only one suited for distributed shared-nothing systems. We use the basic MPI version. Moreover, the MPI implementation has its core loops optimized using OpenMP pragmas. This means that one can run each MPI process in a multi-threaded fashion. When MPI slave machines have *x* cores each, one could run *x* single-threaded MPI processes per machine or run 1 *x*-threaded MPI process per machine. This ensures that each core runs one thread, making an optimal use of the machine.

The benchmark consists of three stages:

1. *Input Graph Generation*.

2. *BFS Execution*: The breadth first search (BFS) algorithm is run on the graph. This step computes the parent of each vertex in the resulting BFS tree. Only MPI messages are used for communication in BFS computation.

3. *Validation*: The benchmark validates that the resulting tree is a correct BFS search tree. Specifically, it checks that all edges connect vertices whose depths in the resulting tree differ by exactly one, and that each tree edge is an existing graph edge. When an MPI process needs to access an edge that is owned by another process, it initiates a remote memory access, that is, it accesses via network the remote process's memory that stores the edge. Exactly this operation is the one that stresses the communication subsystem of the system under test.

Steps two and three are run repeatedly for 64 unique randomly selected BFS source vertices on one and the same graph, because some source vertices may reside in connected components much smaller than the whole graph, thus resulting in the algorithm running much faster than on average. Running these two steps 64 times reduces the error introduced by such outliers. All the three stages are timed and at the end of the experiment the measured times are being displayed.

The benchmark gets two configuration parameters, the number of vertices and the average number of edges per vertex (the *edge factor*). Instead of actually specifying the number of vertices, one needs to specify the logarithm base two of the number of vertices. The benchmark calls this the *scale*, that is, $scale = log_2(\#vertices)$. The benchmark suggests to fix the number of edges per vertex to 16 and vary the scale. The benchmark defines six problem classes depending on the graph scale:

- *toy* - 17GB of input graph data, scale 26 (i.e., 2^{26} vertices),

- *mini* - 140GB of input graph data, scale 29,

- *small* - 1TB of input graph data, scale 32,

- *medium* - 17TB of input graph data, scale 36,

- *large* - 140TB of input graph data, scale 39, and
- *huge* - 1.1PB of input graph data, scale 42.

In the Graph500 list, the largest scale that has been reached is 40. The machine that is on the first place in this list has 1,048,576 cores. All the top 20 systems from the Graph500 list reach scale 35 and above. It is interesting to note that while some machines in the top 20 of the Graph500 list have hundreds of thousands of cores, others have only a few thousands of cores. Currently, there are 123 systems listed in the Graph500 top. There are many systems in the list that only managed to get to scales between 20 and 30. The list does not specify what implementation of the benchmark each system used. It would have been useful for us to know what systems have used the MPI implementation (such that we could compare them to our three systems).

Experimental Setup In our experiments we fix the edge factor to 16 and increase the scale until the system runs out of memory or the execution takes longer than one hour. Moreover, we repeat this procedure for increasing number of machines. On DAS4 we run for 8, 16, 32, 64, 128 cores, while on OpenNebula and AWS we run with 8, 16, 32, 64 cores (64 cores is the maximum we can lease in our OpenNebula installation). In Section 4.1, we present the results and discus the outcomes.

3.3.3 Apache Giraph

Apache Giraph is a highly scalable graph processing system built on top of Hadoop. It is the open-source implementation of Google's Pregel [31]. Giraph's model of computation is inspired by the Bulk Synchronous Parallel [39] model for distributed computation.

Giraph Design Insights Although Giraph runs as a Hadoop job, it does not use the MapReduce model in the traditional way. Instead, it uses Hadoop as a resource manager. Giraph system consists of a master and a number of workers. Each worker owns a subset of the input graph's vertices. Moreover, each worker is the one that executes the computation function for each of its owned vertices and sends and receives messages to and from vertices owned by other workers. The master is the one responsible for synchronizing the supersteps and recovering the dead workers. Giraph allocates a Map task for each worker and a Map task for the master. No Reduce tasks are used in a Giraph computation. The Map tasks allocated for master and workers terminate only when all the supersteps have finished, that is, only one MapReduce phase is necessary to run a Giraph computation.

Giraph Benchmarks Giraph source code ships with the implementation of two graph algorithms: Shortest Paths and Page Rank. It uses these two algorithms for benchmarking its own performance. Giraph provides a random graph generator,

which these two benchmarks make use of. For this thesis, we run these two benchmarks for different graph sizes and different number of workers. Moreover, we implemented the Breadth First Search algorithm ourselves and used it also as a benchmark. We run BFS with Giraph in order to compare its performance with the MPI implementation from Graph500.

Experimental Setup We run the Breadth First Search, Shortest Paths, and Page Rank programs on the Hadoop installation from DAS4. Initially, we planned to run these benchmarks on AWS and OpenNebula as well. However, installing Hadoop on these systems and running Giraph on it proved problematic for us (because of compatibility issues between Hadoop and Giraph), and therefore we decided to skip running Giraph benchmarks on these two systems due to the lack of time.

Hadoop is installed on 8 DAS4 machines with the following configuration: Intel "Sandy Bridge" E5-2620 (2.0 GHz) 6 core CPU, 64 GB memory. Note that each machine has a 6 core CPU, thus in total, the DAS4 Hadoop installation makes use of 48 cores. In our experiments, we refer to the *scale* of the graph size as the logarithm base two of the number of graph vertices. We use this in order to have the same graph size unit as in Graph500, which would make the comparisons easier. As said above, we use Giraph's built-in graph generation facility and, as in Graph500, we generate graphs with 16 edges per vertex. As in Graph500, we increase the graph scale until the system either times out or fails for some reason. We run the benchmarks with 8, 16, 32, and 48 workers.

3.4 Machine Learning

In Section 3.4.1, we provide a general description of machine learning algorithms. In Section 3.4.2, we introduce Apache Mahout [2], a machine learning library we use to stress test our systems for machine learning applications. In Section 3.4.3, we describe the datasets we selected for machine learning with Mahout. Finally, in Section 3.4.4, a description of the experimental setup is provided.

3.4.1 Machine Learning Algorithms

Data Representation in Machine Learning Machine learning algorithms usually get as input a set of labeled or unlabeled items and try to find structure in that set. The items are characterized by a number of independent variables, also referred to with the term *features*. Each item has a value assigned for each feature. Machine learning algorithms usually represent the items as vectors. Each vector dimension corresponds to a feature.

In order to choose our datasets, we needed to understand what makes the machine learning datasets large. The size of machine learning datasets are influenced by both the number of items and the number of features (aka dimensions) for each item. In some cases, the number of features is fairly small, whereas in others it can

be very large. An example of the latter is a dataset with text documents. In this case, first a dictionary is created with all the unique words present in the dataset. Each unique word in the dictionary becomes a dimension in the vectors representing the documents. The value for a dimension of a document is the number of times that particular word appears in the document.

Clustering Algorithms Clustering is a classical unsupervised learning approach. An unsupervised learner gets as input a set of unlabeled items and tries to find structure in them. Clustering in particular is the problem of grouping together subsets of the input set, such that the items in a group are more similar to each other than to items from other groups. As said above, the items are represented as vectors and in order to measure the similarity between two items, clustering algorithms use distance metrics between the vectors of the two items. Popular distance metrics are: euclidean distance, cosine similarity (basically measures the angle between the two vectors), Manhattan distance.

There exist several widely known clustering algorithms: K nearest neighbors, K-Means, Dirichlet clustering. There is no clear rule about which of these algorithms would provide the best accuracy. Usually the data scientists try several algorithms and pick the one that provides the best results for a given dataset. For our experiments, we chose the K-Means algorithm. This algorithm needs to be instructed how many clusters to create. Let k be the number of desired clusters. The algorithm starts by choosing k random points that will be the centroids of the groups to be created. Next, it iterates through all the points (items) in the dataset and assigns each point to the centroid that is closest to it. After the groups have been created, the algorithm computes the centroids for the newly created groups. With the new centroids it starts another iteration. The algorithm finishes when the centroids do not change from one iteration to another or when a maximum number of iterations has been reached.

Classification Algorithms Classification is a supervised learning problem approach. A classifier learns to assign newly observed items to a category from a well defined set of categories, based on their features. It trains from a dataset where each item has a number of features and a category already assigned to it, based on previous observations.

Widely used classification algorithms include: Decision Trees, Naive Bayes, Stochastic Gradient Descent (SGD). Naive Bayes computes the probability that an item containing a certain word is in a certain class. Naive Bayes better fits problems where the items have high dimensionality, such as text documents. In SGD, the class of an item is defined as a linear function of the values of its features. SGD performs better on items with few dimensions. SGD is know to perform very fast, but it is inherently sequential, which can be a bottleneck for very large datasets. As said above, in machine learning it is common to have both high dimensional and low dimensional datasets. Therefore, in our experiments, we decided to run both

Naive Bayes and SGD to cover both cases.

3.4.2 Apache Mahout

Apache Mahout [2] is a machine learning library for reasonably large data. Mahout runs most of its utilities and algorithms on top of Hadoop, using the MapReduce paradigm, but is not limited to using Hadoop or MapReduce (e.g., Mahout's SGD implementation does not use MapReduce, nor Hadoop). It provides an API for developing and customizing machine learning and data mining algorithms. More important, it provides implementations for the most widely used machine learning algorithms. Currently, Mahout provides algorithm implementations for four machine learning fields: clustering, classification, recommendation mining, and frequent item sets mining. Most of the algorithms implemented in Mahout get the input items in form of vectors. Mahout provides a handful of tools to transform custom data (e.g., text) to a vector format.

As said above, we choose to run the following three algorithms from Mahout: K-Means, Naive Bayes, and SGD. K-Means is implemented as a series of MapReduce iterations and one can specify the maximum number of iterations to run. Naive Bayes executes two MapReduce iterations exactly. SGD is implemented with no distributed computing in mind, thus, makes no use of Hadoop or MapReduce. It runs in a multi-threaded fashion on exactly one machine.

3.4.3 Selected Machine Learning Datasets

Unlike in the case of graph processing, where synthetically generated graphs resemble real-world graphs relatively well, in machine learning using randomly generated datasets does not reproduce real-world situations. For this reason, we had to find existing datasets that would be large enough to stress-test our systems. Moreover, we needed and a low dimensional dataset and a high dimensional dataset (e.g., text documents).

Forest Dataset The low dimensional dataset we chose is a forest cover type dataset [3]. The items in the dataset represent 30x30 meter cells from a forest area. The purpose of the dataset is to learn what tree species grow in a cell based on some features of the cell. The items have the following 12 features:

- Elevation
- Aspect
- Slope
- Horizontal Distance to Hydrology
- Vertical Distance to Hydrology

- Horizontal Distance to Roadways
- Hill Shade at 9am
- Hill Shade at noon
- Hill Shade at 3pm
- Horizontal Distance to Fire Points
- Wildness Areas
- Soil Type

The first ten features are quantitative measures, so each of them translates to exactly one vector dimension, whereas the last two are qualitative measures which in total transform to 44 vector dimensions. Therefore, the final vectors in this dataset have 54 dimensions. There are seven tree types used in this dataset to categorize the items. This dataset has 581,012 items and its size is 72 MB.

Wikipedia Dataset The high-dimensional dataset we created is a collection of Wikipedia articles. Wikipedia [6] provides large data dumps, but they are not classified. However, Wikipedia organizes all its pages in a category graph. It has 25 top categories. Each top category contains a number of pages and a number of subcategories. The same holds for subcategories. Unfortunately, the pages in the Wikipedia dumps are not tagged with the top category they reside in (more precisely, they are not tagged with any category).

We created a crawler that starts from the 25 top categories and goes down their subtrees and downloads all the pages in those subtrees. The crawler tags each downloaded page with the top category it resides in. The resulting dataset's size is about 4GB and contains 605,250 text documents. The documents are classified in 25 categories.

3.4.4 Experimental Setup

In our experiments, we run SGD with the Forest dataset, because SGD is better suited for low dimensional datasets. We run Naive Bayes with the Wikipedia dataset because it is more suited for high dimensional datasets. We run K-Means with the Wikipedia dataset.

The 4GB Wikipedia dataset proved to be too small for stress-testing our systems, so we decided to create multiple copies of the initial dataset and put them together in order to generate a larger dataset. This may change the accuracy of the algorithms we run, but the accuracy is not of our interest when stress-testing our systems. We are confident that creating multiple copies of the initial data to create a larger dataset conserves the realistic nature of our experiment.

On DAS4, we use the same Hadoop installation as with Giraph, i.e., 8 machines with Intel "Sandy Bridge" E5-2620 (2.0 GHz) 6 core CPU and 64 GB memory,

summing in total 48 cores. On OpenNebula, we use 6 virtual machines with dual quad-core 2.4GHz CPU and 20GB of memory, also summing to 48 cores in total. On AWS, we use 6 *m1.xlarge* virtual machines with 8 EC2 compute units (ECU) each and 15GB of memory. The SGD algorithm uses only one machine on each of the three systems. It is not a distributed algorithm, so it cannot use more than one machine.

Unlike with Giraph, one cannot limit the number of slave nodes to be used in a computation in Hadoop. The computation executes on all the available Hadoop slaves. Therefore, in our experiments, we only vary the size of the dataset, and not the number of computing nodes.

3.5 Scientific Computing

Scientific computing covers a large spectrum of scientific domains. Some scientific applications receive relatively small input data, whereas others have to handle very large input data. In our thesis, we focus on the latter ones. In Section 3.5.1, we present a scientific benchmark we will use to stress-test our systems. In Section 3.5.2, we describe the data management system used to implement this benchmark. Finally, in Section 3.5.3, we describe the experimental setup.

3.5.1 SS-DB: A Standard Science DBMS Benchmark

Traditionally, science users have managed scientific data using general-purpose relational DBMS. However, it is notorious that the format of scientific data is hard to model as a relational database. Specifically, the following characteristics of scientific computing make it hard to for RDBMS to manage scientific data:

- Scientific data is usually organized as multi-dimensional arrays, which are hard to represent in relational databases.
- Many scientific operations, like multi-dimensional windowing and matrix decomposition, are hard to implement in SQL.
- Required features like version control are missing in RDBMS.

Usually, scientific data management comprises the following phases:

1. Ingest raw data from sensors (e.g., images taken from a telescope).
2. Query the raw data (e.g., retrieve the images within a certain boundary).
3. Transform the raw data into a derived dataset that can be queried easier (e.g., extract all the stars from the images taken by a telescope). This operation is usually called ”cooking”.
4. Query the cooked dataset (e.g., get the stars within a certain boundary).

SS-DB [18] is a scientific benchmark that is meant to compare the performance of data management systems regarding their ability to store and efficiently query scientific data. Although the benchmark does not cover all the possible sub-domains of scientific computing, it captures a subset that can be encountered in many scientific sub-domains, specifically: querying multidimensional arrays.

The following are the stages performed by the benchmark:

1. *Raw data generation:* The benchmark generates a number of random images (i.e., two-dimensional arrays). Multiple domains use this kind of data: astronomy (i.e., images from telescopes), oceanography (e.g., satellite imagery), medicine (e.g., X-Ray images of the human body), etc. Each image is a square with a fixed size. Moreover, for each image, its coordinates in a global coordinate system are given.

2. *Detecting observations:* This step generates a cooked dataset with all the observations found in the raw images. Example of observations are: stars (astronomy), islands (oceanography), etc. An observation is defined by a center point *(x, y)*, a bounding polygon (which is a 1D array), and an observation specific value (avgdist, pixelsum).

3. *Cooking observations into groups:* Because the sensor or the observations move while the raw images are taken, the same observation can be seen at different times at different places (e.g., a star may move while images of the sky are taken). The purpose of this step is to group different observations of the same phenomenon. Knowing the maximum speed of an observation, one can detect when two observations represent the same phenomenon.

4. *Queries on data:* Nine queries are defined: three on the raw data (Q1-Q3), three on the observation data (Q4-Q6), and three on the observation groups (Q7-Q9):

    - *Q1 aggregation:* For a given slab, compute its average value from the images that intersect with that slab.
    - *Q2 recooking:* For a given slab, recook the images in that slab using a different cooking function than the initial one.
    - *Q3 regridding:* For a given slab, recompute the raw images in it such that their size decreases with the ratio 10:3.
    - *Q4 aggregation:* For the observations with the center in a given slab, compute the average value of the observation O_i, for a randomly chosen i.
    - *Q5 polygons:* Compute the observations whose polygons intersect a given slab.
    - *Q6 density:* Group the observations that intersect a given slab in D4 by D4 tiles and find the tiles with more than D5 observations.

- *Q7 centroid:* Find the group whose center falls in a given slab. The center of a group is defined as the average value of the centers of the observations in the group.
- *Q8 trajectory:* Define the trajectory to be the sequence of the centers of observations in group. For each trajectory that intersects a slab, produce the raw image of a D6 by D6 tile centered on each center for all images that intersect the slab.
- *Q9 trajectory-2:* An alternative definition of the trajectory is the sequence of polygons that correspond to the boundary of the observation group. For each trajectory that intersects a slab at some time t, produce the raw data of a D6 by D6 tile centered on each center for all images that intersect the slab.

The benchmark can be instructed how large should the input data be. Five levels of the dataset size are defined:

- *tiny:* 10 1000x1000 images with 0.44GB total size.
- *very small:* 40 1600x1600 images with 4.5GB total size.
- *small:* 160 3750x3750 images with 99GB total size.
- *normal:* 400 7500x7500 images with 999GB total size.
- *large:* 1000 15000x15000 images with 9.9TB total size.
- *very-large:* 2500 30000x30000 images with 99TB total size.

Only the data generator is directly provided by the benchmark's authors. The cooking operations and the queries are described in plain English. This may be a cause why the benchmark has not been widely adopted yet. The authors provide two implementations: one based on MySQL and another based on SciDB (a scientific DBMS). After experimenting with the two implementations, the authors conclude that SciDB runs orders of magnitude faster than MySQL. For this reason, we decided to also use the SciDB implementation to benchmark our systems.

3.5.2 SciDB

SciDB [17] is a scientific database management system with a multi-dimensional nested array model. Arrays can have any number of dimensions. Each combination of dimension values forms a cell. All the cells in a multi-dimensional array have the same type. A cell has a scalar and/or one or more arrays as values. The arrays are split into chunks and the chunks can be spread across multiple disks or machines.

Besides a rich set of built-in operators, SciDB provides a means to define user defined functions (UDF) in the C++ programming language. A UDF can modify the structure of an array, work on the content of the cells, or do both. UDFs operate

at a chunk granularity. Given that arrays are split in chunks, SciDB is able to automatically parallelize UDFs by running an instance of a UDF for each chunk. The parallelization can be done locally (by spawning threads on each CPU core) or across multiple machines.

A SciDB installation consists of a number of instances spread across a number of cluster nodes. One SciDB instance is the coordinator, whereas all the others are worker instances. The cluster node that runs the coordinator can also run one or more worker instances. The SciDB documentation recommends that an instance should use one hard drive and four CPU cores, in order to achieve maximum performance. This is easy to configure in a cluster of virtual machines.

3.5.3 Experimental Setup

On DAS4, we use a cluster of 6 processing nodes with the following node configuration: dual quad-core 2.4GHz CPU, 24GB of memory, 18TB of disk storage. On OpenNebula, we use the same virtual machine cluster configuration as for machine learning (see Section 3.4.4): 6 virtual machines with dual quad-core 2.4GHz CPU, 20GB of memory, and two hard drives of 300GB each. On AWS, we use 6 *m1.xlarge* virtual machines with 8 EC2 compute units (ECU) each, 15GB of memory, and 420GB storage. As said above, it is recommended by the SciDB documentation to have one instance per four cores, so in our case, two instances per machine would have been the best configuration. However, in our experiments, we realized that parallel data loading does not work when more than one instance per machine is used. Not using parallel data loading is out of question, for it is unpractical to load very large data from only one machine (e.g., for 100GB of data, the loading would take hours using only one machine). Therefore, on each virtual machine, we run exactly one SciDB instance. That is, we will have one coordinator and 5 worker instances.

Chapter 4

Experiments and Results

In the previous chapter, we have introduced the Big Data applications that we run on the three systems: DAS4, OpenNebula, and AWS. In this chapter, we present the results of the experiments, we discuss the encountered problems, and we draw conclusions based on the results.

In Section 4.1, we discuss the results of running the Graph500 benchmark on the three systems. In Section 4.2, we present the results of running Apache Giraph on DAS4. In Section 4.3, we compare the MPI implementation from Graph500 and the Giraph implementation of Breadth First Search. In Section 4.4, we discuss the outcomes of executing Mahout implementations of three machine learning algorithms. In Section 4.5, we describe the results of running the SS-DB benchmark in order to benchmark our cloud systems for scientific computing.

4.1 Graph Processing Results: Graph500

4.1.1 DAS4

We first run Graph500 on the raw DAS4. As said in Section 3.3.2, we can choose how many MPI processes to run on each MPI slave machine and how many threads should each MPI process execute. A DAS4 node has eight cores in total, so we had to choose between eight single-threaded MPI processes per machine or one eight-threaded MPI process per machine. We chose to use the former because we think that running independent single-threaded processes would make a better use of the eight cores.

We started with the scale of 10 (i.e., 2^{10} vertices) and increased it as much as possible. The same has been repeated for 8, 16, 32, 64 and 128 cores. Figure 4.1 shows the results. There are at least two interesting observations to make from this figure, which we discuss below.

The first interesting fact we noticed was that for the same data size, the execution time *increases* when increasing the number of MPI processes (i.e., the more cores). The reason for this is that the dataset is evenly spread across all the MPI processes,

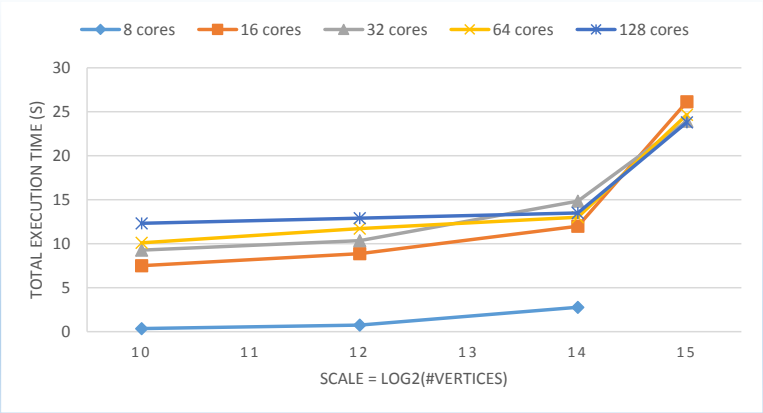

Figure 4.1: The execution time of Graph500 on DAS4 for 8, 16, 32, 64 and 128 cores (8 single-threaded MPI processes per machine).

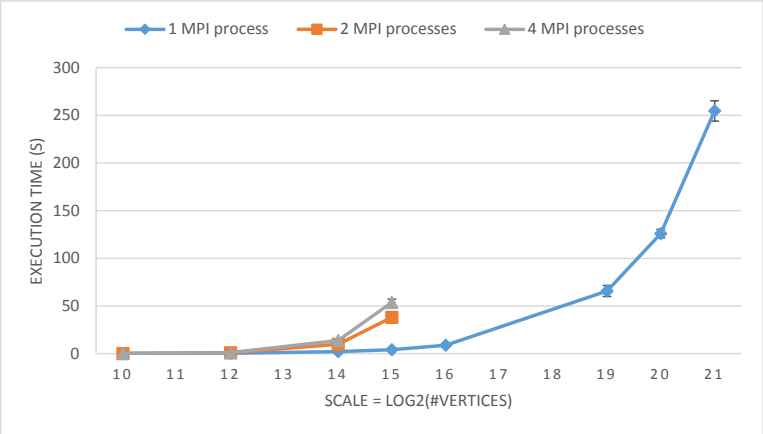

Figure 4.2: The execution time of Graph500 on DAS4 on 1, 2 and 4 machines (1 eight-threaded MPI process per machine).

and therefore, the more MPI processes, the smaller the chunk size owned by each MPI process. This results in increased network communication, which proves to be a bottleneck in this implementation. Moreover, note that for up to 2^{14} vertices, the execution with 8 MPI processes is between 4 to 35 times faster than when running on more than 8 MPI processes. The reason for this is that all 8 MPI processes reside on the same MPI slave machine, thus this machine does not have to communicate with other machines.

The second striking fact was that the benchmark was timing out (one hour) already at scale 16 (i.e., only 50MB of data), independently of the number of machines we used. When running using 8 MPI processes, the benchmark times out at already scale 15. In order to see whether this is a configuration problem, we ran the same experiments with 1 eight-threaded MPI process per machine. Fig-

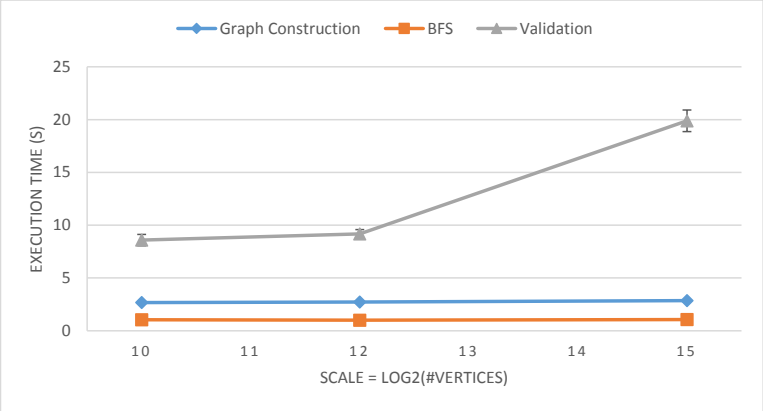

Figure 4.3: The execution time of Graph500 for graph construction, BFS execution, and validation. The execution has been performed on DAS4 using 128 cores.

ure 4.2 shows the results. For more than one machine the experiment fails again at scale 16. It is interesting though that when only one MPI process is used, the benchmark goes up to scale 21. At scale 22, the machine runs out of memory. This made us think that the communication between MPI processes is problematic and when only one MPI process is used, there is no communication involved, thus the experiment gets to higher scales.

We have also run the same experiments on OpenNebula and AWS. Most important was that we could not get to scales greater than 15 on these two systems as well. On these two systems, the execution times were comparable with the ones on DAS4, with negligible differences. This was expected because of the small data size. Therefore, we do not include charts for these two systems, as they do not provide much extra information.

4.1.2 Understanding the Graph500 bottleneck on DAS4

As said in Section 3.3.2, the Graph500 benchmark consists of three phases: graph construction, BFS computation, and validation of the created BFS tree. The benchmark displays the execution time for each of the three phases. In order to understand why the total execution time increases so much at scale 15, we checked which of the three phases is the bottleneck. Figure 4.3 displays the execution time for each phase when running on 128 cores. We notice that the validation time increases fast with the scale, whereas the BFS execution and construction time remain almost unchanged, thus, the validation is the bottleneck. During our experiments, it never happened that the validation found a wrong BFS tree. Therefore, it makes no sense to run the validation step when the implementation has been tested to be correct. This confirms that validation's main purpose was to stress the communication system rather than checking the validity of the BFS tree.

It is surprising that for only 50MB of graph data, the communication system

gets so much overloaded, considering the fact that the InfiniBand connection has a reasonably high bandwidth for a grid system. This makes us think that the implementation of the validation step is not suited for distributed shared-nothing systems that use high-speed Ethernet or InfiniBand network connections. It is possible that this stage would work better on systems that use shared-memory, considering that remote memory access has been chosen for its implementation.

Another question one may ask is: why executing on one machine with 1 MPI process manages to get to scale 21, whereas executing on one machine with 8 MPI processes hits the bottleneck problem at scale 15? When using only one MPI process, that process owns all the dataset and therefore does not communicate with any other process. When using 8 MPI processes, they still have to communicate, although they all reside on the same machine. At validation, instead of directly accessing each other's shared memory, the processes use remote memory access which is implemented on top of the network stack of the machine. This proves to be a bottleneck even if the machine does not communicate to other machines. It seems like a good idea to optimize the MPI remote memory access implementation such that it avoids using the network stack when sharing the memory between processes residing on the same machine.

Amazon EC2 provides a wide range of virtual machine types, so we tried to find a virtual machine type that has the largest network bandwidth, in order to see whether we can get to datasets larger scale 15. We chose to run the benchmark using two *hi1.4xlarge* virtual machines. These are storage optimized instances and have 10 Gigabit network connection, which is the largest bandwidth possible in Amazon EC2. Again, the benchmark did not get to datasets larger than scale 15. This was expected because 10 Gbps bandwidth is less than the bandwidth of InfiniBand available on DAS4. This proves that our results are consistent on all three systems we benchmarked.

4.1.3 Comparing DAS4, AWS and OpenNebula

In order to be able to make use of this benchmark for assessing the differences between the three systems, we disabled the validation step. Figure 4.4 shows the execution times for when using 8, 16, 32, 64 cores, respectively. We managed to reach graphs of 2^{27} vertices (34GB of data). The execution time is the smallest on DAS4. As a result, we get to higher scales on DAS4 compared to the other two systems. This is expected, as this system does not imply any virtualization, which makes the use of machines' resources more efficient. Moreover, the physical machine is not shared with other users. It is worth noting that OpenNebula performs up to 11 times slower than DAS4. Although an OpenNebula virtual machine is set to use the maximum CPU and almost all the memory of the physical machine and it does not share the physical machine with other virtual machines, the CPU and I/O virtualization introduces a significant overhead. We suspect that the I/O virtualization (specifically, the network virtualization) is the one that causes the most overhead. The execution runs up to 2.7 times faster on AWS than on OpenNebula,

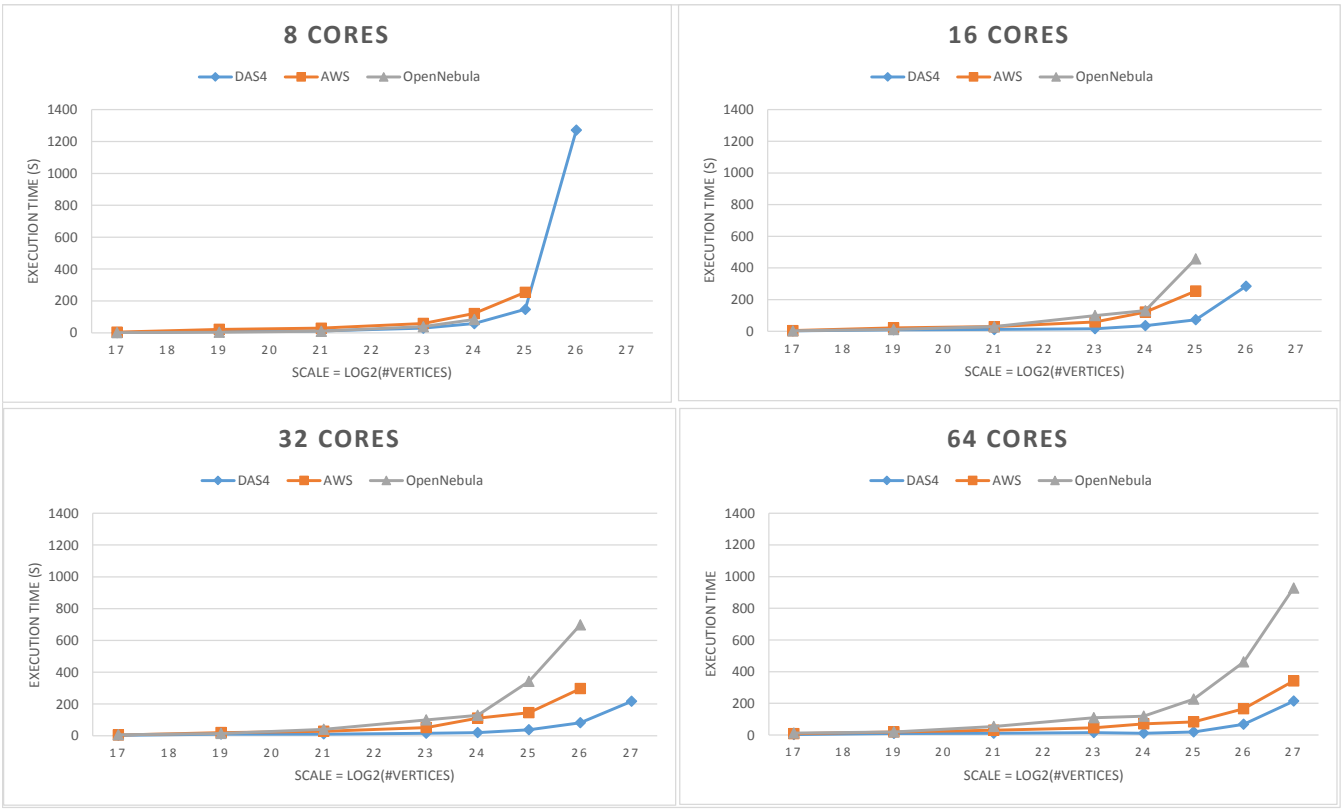

Figure 4.4: The execution time of Graph500 on DAS4, OpenNebula, and AWS (validation disabled).

in spite of the fact that there is a high probability that an AWS virtual machine shares the physical machine with other virtual machines. The fact that the chosen AWS instances have smaller memory (15GB) than OpenNebula virtual machines (20GB) does not affect the performance. This proves that this benchmark is not memory bound.

Another important observation is that the MPI implementation of BFS, as it appears in Graph500, scales linearly with the number of cores. Remember that the horizontal axis of the charts is a logarithmic scale of the number of vertices. As seen in Figure 4.4, when doubling the number of cores, the benchmark doubles the maximum data size it can process on all three systems: with 16 cores all three systems can process 2^{25} vertices, with 32 cores - 2^{26} vertices, with 64 cores - 2^{27} vertices. Moreover, for a fixed data size, when doubling the number of cores, the running times decrease almost two times (e.g., check the difference in running times for scale 25 in the 16 cores, 32 cores and 64 core charts, respectively). We estimate that this BFS implementation would scale as well if one would continue adding cores to the system.

4.1.4 Graph500 Conclusions

We reiterate four key conclusions resulting from our stress-testing with Graph500:

- The validation stage of BFS proves to be a bottleneck already for small datasets. While its purpose is to stress the communication subsystem of the system under test, we find it unsuited for distributed systems. The distributed systems use high speed Ethernet or InfiniBand for communication and this benchmark finds these communication systems too slow.
- The graph construction and BFS execution stages scale linearly with the number of used cores. Apart from the validation stage, we find this benchmark suitable for benchmarking distributed systems.
- OpenNebula performs up to 11 times worse than DAS4. The CPU and network virtualization introduces a significant overhead for the OpenNebula machines.
- OpenNebula performs up to 2.7 times worse than AWS.

4.2 Graph Processing Results: Giraph

As said in Section 3.3.3, we run the Giraph implementations of the Breadth First Search, Shortest Paths, and Page Rank algorithms only on DAS4. We use increasingly large graphs until the system either timed out (one hour) or fails for some reason. We run the three programs with 8, 16, 32, and 48 workers. Figure 4.5 shows the results.

The first question one may ask is why do we use 48 workers and not 64. As said in Section 3.3.3, the Hadoop installation from DAS4 has 48 cores in total. Still, we tried to run with 64 workers to see the outcomes. The result was that Giraph computations failed for graphs with only hundreds of vertices. In order to understand this problem, one needs to remember that in Giraph each worker runs in a Map task. In a regular MapReduce computation, it is possible to have some Map tasks in a queue waiting for other Map tasks from the same computation to finish. However, in Giraph, all the workers need to be active at the same time, i.e., all the Map tasks allocated for the workers. Because the DAS4 Hadoop installation has only 48 cores, it does not allow to have 64 Map tasks active at the same time, so the computation fails. Therefore, the maximum number of workers we could use was 48.

4.2.1 Scalability of BFS, Shortest Paths, and Page Rank with the Data Size

It is important to analyze the way these three algorithms scale with the dataset size, in order to predict whether adding more machines to the system will help increase the maximum dataset each algorithm can process.

Figure 4.5: The execution time of the Giraph implementations of Breadth First Search, Shortest Paths, and Page Rank algorithms on DAS4. Shortest Paths times out for smaller scales than the other two algorithms; for this reason it misses some chart values for scales for which the other two algorithms do have chart values.

Looking at Figure 4.5, the first observation to be made is that BFS has a significantly lower execution time than Page Rank and Shortest Paths. This is explained by the fact that BFS executes as many supersteps as the diameter of the graph. From our observations, in our experiments the diameter was no more than 10, regardless of the number vertices, because the diameter grows very slowly with the number of vertices when the number of edges per vertex is high. Page Rank, on the other hand, always executes exactly 30 supersteps, which is the commonly used value in the papers describing this algorithm. The number of supersteps executed by the Shortest Paths algorithm is equal to the maximum number of edges in any of the shortest paths in the graph (note that the algorithm creates the shortest path between the source vertex and all the other vertices). In the best case only, the maximum number of edges in any of the shortest paths is equal to the diameter of the graph (remember that Shortest Paths computes the shortest paths in *weighted* graphs, whereas BFS does not consider the edge weights). On average though, this number is larger than the diameter, therefore, the number of executed supersteps is usually much larger than the number of supersteps executed by BFS. This is also the reason why for a fixed number of workers, the maximum graph size Shortest Paths can process is two or four times smaller than what BFS and Page Rank can

	BFS			Shortest Paths			Page Rank		
Scale	16w	32w	48w	16w	32w	48w	16w	32w	48w
11	0.79	0.37	0.30	0.79	0.64	0.55	0.86	0.63	0.51
13	1.00	0.44	0.35	0.84	0.63	0.53	0.86	0.68	0.54
15	0.67	0.32	0.31	0.87	0.66	0.55	0.80	0.64	0.48
17	0.88	0.45	0.35	0.80	0.58	0.47	0.86	0.66	0.53
19	0.83	0.56	0.48	0.78	0.59	0.45	0.78	0.62	0.49
21	1.04	0.69	0.59	1.03	0.84	0.63	0.93	0.76	0.62
23	1.39	1.27	1.15	N/A	N/A	N/A	1.60	1.44	1.16

Table 4.1: Speedup over 8 workers of BFS, Shortest Paths, and Page Rank on 16, 32, and 48 workers, respectively.

process.

The conclusion we can draw from is: if the number of supersteps does not increase when increasing the graph size, one can be confident that one can scale to very large graph sizes by just adding machines to the Hadoop installation. With respect to this, BFS scales better than linear, Page Rank scales linearly (if the number of supersteps is kept constant), and Shortest Paths scales higher than linearly with the graph size.

4.2.2 Scalability of BFS, Shortest Paths, and Page Rank with the Number of Workers

We have computed the speedup of the algorithms for different number of workers, in order to understand whether increasing the number of workers results in a significant decrease of the execution time. Table 4.1 shows the speedups over 8 workers for each algorithm and scale, when running with 16, 32, and 48 workers, respectively. There are at least two striking observations to be made. First, for scales lower or equal to 21, the execution time is actually *longer* than in the case of 8 workers (i.e., speedup smaller than 1), and only for scale 23 some speedup is noticeable (i.e., speedup up to 1.6). Second, for each fixed scale, the speedup *decreases* when the number of workers *increases*.

There is exactly one cause for both. When increasing the number of workers, the graph gets spread across more workers, so each worker owns a smaller number of vertices. This increases the probability that neighbor vertices reside on different workers, thus resulting in increased network communication between workers, which results in increased execution time. At scale 23, a cluster of 8 workers is at the maximum it can process (see Figure 4.5), i.e., its resources are overloaded (it is close to exhausting its memory capacity), which results in a significant overhead and increased execution time. On the other hand, clusters of 16 workers or more are well below their capacity limit at scale 23, which makes their execution time smaller than in the case of 8 workers. This is why we notice a speedup larger than

1 for scale 23. Moreover, our explanation is supported by the fact that there is an increasing trend in the speedup from scale 19 to scale 23, which indicates that the cluster of 8 workers approaches its upper limit.

From this, we conclude that the Giraph implementation of graph algorithms scales insignificantly with the number of workers for fixed data sizes. To get shorter execution times for fixed graph sizes, one should use clusters as small as possible, as long as the cluster does not get overloaded.

4.2.3 Giraph Conclusions

To summarize, the following are the conclusions of our experiments with Giraph:

- If the number of supersteps performed by an algorithm increases with the graph size, the algorithm will scale worse than linearly with the graph size, therefore, adding more nodes to the system will hardly help in increasing the maximum size of the graphs that can be processed. Shortest Paths is an example of such an algorithm.
- Giraph implementations of graph algorithms do not scale with the number of workers for fixed graph sizes.

4.3 Comparing MPI and Giraph Implementations of BFS

When comparing the execution times of the MPI implementation from Graph500 (Figure 4.4) and the Giraph implementation (Figure 4.5) of BFS, we notice that:

- The MPI implementation performs 3 times better on average.
- The maximum graph size that can be processed by the MPI implementation is 4 times larger than the Giraph counterpart's.
- For a fixed graph size, the execution time of the MPI implementation decreases when the number of cores increases, whereas the Giraph implementation's execution time does not.
- Although the Giraph implementation performs worse, we find implementing graph algorithms in Giraph much easier than on MPI. It took us less than one hour to implement BFS in Giraph and the resulting source code was less than one hundred lines. The MPI implementation looks more laborious and requires expertise in implementing and understanding distributed algorithms.

4.4 Machine Learning Results: Mahout

In this section, we describe the results of executing the Mahout implementations of Naive Bayes, K-Means, and Stochastic Gradient Descent on DAS4, OpenNebula, and AWS. In Section 4.4.1, we describe how we preprocessed the Wikipedia

Operation	*Execution Time*
Uploading to HDFS	3h 16min
Creating sequence files	4h 58min
Creating vector files	11min

Table 4.2: Execution time for preprocessing operations for the Wikipedia dataset (4GB, 605,250 files).

dataset and the problems encountered in this process. In Sections 4.4.2, 4.4.3, 4.4.4, we present the results of running Naive Bayes, K-Means, and SGD, respectively. In Section 4.4.5, we summarize the main conclusions after running machine learning algorithms on cloud systems.

4.4.1 Preprocessing the Data

Before running Naive Bayes and K-Means on the Wikipedia dataset, we had to transform the raw text documents to vectors, because this is the format accepted by the Mahout implementations of Naive Bayes and K-Means. For this, we had to upload the Wikipedia dataset to the Hadoop Distributed File System, transform the documents to sequence files (the key-value file format of Hadoop), and transform the text documents to vector files (remember vector representation description from Section 3.4.1). Mahout provides tools to execute these operations. The preprocessing has been carried out on DAS4. For the other two systems, the final vector files were used directly. Table 4.2 displays the execution times of the preprocessing operations.

It is striking that the upload of only 4GB to HDFS takes more than three hours and the creation of sequence files takes almost five hours. The cause of this is the inability of Hadoop to efficiently handle large number of small files. The problem is well known and there are several works [30, 20] that try to solve it.

The preprocessing packs the 605,250 text documents in only 32 vector files. Therefore, the Naive Bayes and K-Means algorithms will work with a relatively small number of large files. As said in Section 3.4.4, we replicate these 32 vector files and put them together to generate an even larger dataset.

The SGD implementation from Mahout can read Comma Separated Values (CSV) files directly. The Forest dataset is exactly in this format, so no preprocessing was needed for this dataset.

4.4.2 Naive Bayes Results

Figure 4.6 shows the execution times of Naive Bayes on DAS4, OpenNebula, and AWS. In this case, AWS performs the worst and DAS4 performs the best. Naive Bayes runs on OpenNebula up to 1.28 times slower than on DAS4. We suspect that most of this overhead comes from CPU virtualization than from network virtualization, because MapReduce computations do not involve communication between

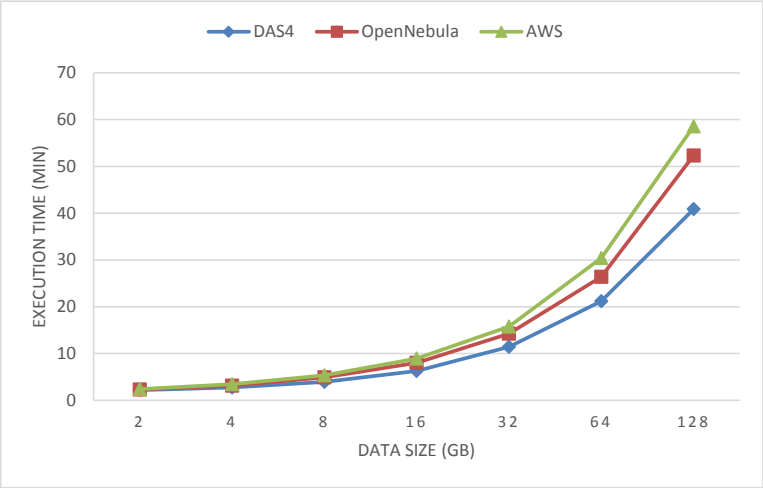

Figure 4.6: The execution time of Naive Bayes on DAS4, OpenNebula, and AWS on the Wikipedia dataset.

Map tasks or between Reduce tasks. The performance of AWS is not much different than the performance of OpenNebula. AWS performs only up to 1.15 times slower than OpenNebula.

Naive Bayes manages to process up to 128 GB of data in at most one hour on all the three systems using a total of 48 cores. At 128 GB, the dataset contains around 20 millions text documents. The algorithm scales linearly with the data size: when doubling the data size, the execution time increases on average 1.6 times for all three systems.

4.4.3 K-Means Results

Analyzing the results of K-Means execution was not as straight forward as in the Naive Bayes case. As said in Section 3.4.1, K-Means chooses the initial centroids randomly and runs a number of iterations until the centroids converge to stable locations. The closer the initial centroids are to the final centroids, the fewer iterations the algorithm runs. This randomness influences strongly the execution time of the algorithm. In our experiments, we noticed large variations in the number of iterations executed by the algorithm, and so, also in the execution time. Sometimes the algorithm would finish in four iterations, sometimes in ten iterations for the same dataset. Therefore, analyzing the total execution time of K-Means computations would not make much sense. However, we can analyze the execution time per iteration.

Before running the iterations, MapReduce decompresses the files from HDFS and feeds them to Map tasks. The running time of the decompression is a significant fraction from the total execution time of the computation. Therefore, in Figure 4.7, we present both the execution time of the decompression and the exe-

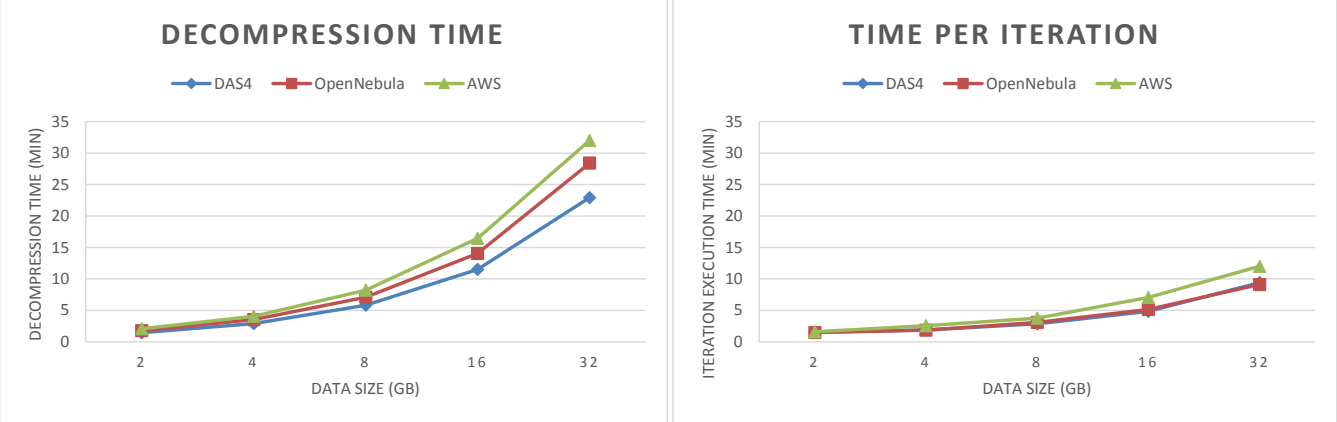

Figure 4.7: The decompression and iteration time of K-Means on DAS4, OpenNebula, and AWS on the Wikipedia dataset.

cution time per iteration, as a function of the data size.

The difference between the execution times of the three systems is not large. The decompression runs up to 1.23 times slower on OpenNebula than on DAS4 and up to 1.17 times slower on AWS than on OpenNebula. The time per iteration is up to 1.07 times slower on OpenNebula than on DAS4 and up to 1.17 times slower on AWS than on OpenNebula.

It is interesting to note the very small difference in iteration time between OpenNebula and DAS4. During an iteration very few I/O operations happen, because at that time the Map tasks already have the data loaded in memory. The only I/O happening in an iteration is the transfer of data from Map tasks to Reduce tasks. So it is safe to assume that the computation performed during an iteration is mostly CPU bound. So the only source of overhead between the two systems during an iteration is CPU virtualization. Considering the small difference in iteration time, we think that in general in the OpenNebula installation on DAS4, the CPU virtualization alone does not introduce much overhead. At decompression, we notice a larger performance difference between OpenNebula and DAS4. This operation is I/O bound (it fetches the data from disk), therefore we think that the overhead between OpenNebula and DAS4 in general comes more from I/O virtualization than from CPU virtualization.

Another observation to make is that the maximum data size K-Means can process on 48 cores is much smaller than the maximum data size Naive Bayes can process, viz., 4 times smaller. This is caused by the large number of MapReduce iterations K-Means uses. Remember that Naive Bayes uses only two MapReduce iterations.

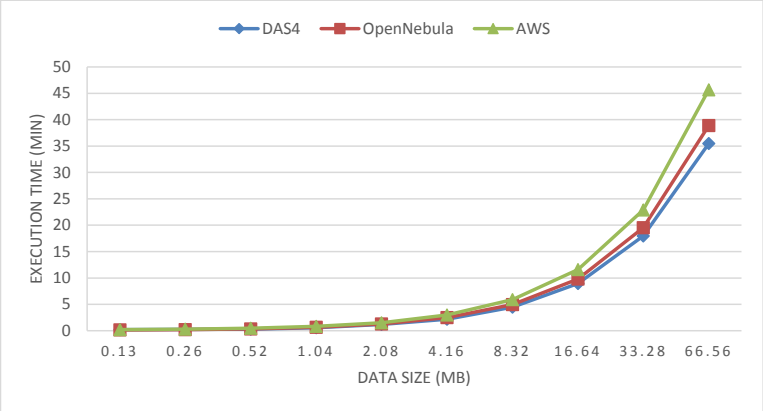

Figure 4.8: The execution time of Stochastic Gradient Descent on DAS4, OpenNebula, and AWS on the Forest dataset.

4.4.4 Stochastic Gradient Descent Results

We have executed SGD using the Forest dataset. Figure 4.8 shows the results on the three systems. OpenNebula performed up to 1.1 times worse than DAS4 and AWS performed 1.19 times worse than OpenNebula. Notice that the maximum data size SGD could process was 66MB. Although its physical size is not large, the dataset contains 580,000 items. It was expected that this algorithm would not be able to process very large datasets, considering that this is not a distributed algorithm (it only runs on one machine). Still, being able to process around 500,000 items is impressive for a non-distributed algorithm. Nevertheless, not being able to scale on multiple machines is an important drawback, which makes this algorithm unpractical for datasets with millions of items.

4.4.5 Machine Learning Conclusions

To summarize, the following are the most important conclusions regarding executing machine learning algorithms on cloud systems:

- The fact that Hadoop is not able to efficiently handle large numbers of small files can pose many problems to practitioners when uploading data to HDFS and preprocessing it.
- For all the three algorithms, DAS4 performs the best, OpenNebula the second, and AWS the last, yet the differences are small.
- In one hour, using 48 cores in total, Naive Bayes is able to process up to 128GB of data (20 millions items), K-Means can process up to 32GB (5 millions items), SGD can process up to 66MB of low dimensional data (500k items).

	Very Small (4.5GB)			*Small* (99GB)		
	DAS4	OpenNebula	AWS	DAS4	OpenNebula	AWS
Generation Time (min)	4.15	4.07	4.68	18.38	17.85	28.07
Loading Time (min)	14.17	14.60	16.57	72.60	74.17	79.87

Table 4.3: The data generation time and loading time for the *very small* and *small* workloads of the SS-DB benchmark on DAS4, OpenNebula, and AWS.

- We believe that the most overhead of OpenNebula over DAS4 comes from I/O virtualization, whereas CPU virtualization does not seem to introduce a large overhead.
- SGD is unpractical for very large datasets, because it is not able to scale with multiple machines.

4.5 Scientific Computing Results: SS-DB on SciDB

As said in Section 3.5.1, the SS-DB benchmark consists of six workloads of increasing size: tiny (0.44GB), very small (4.5GB), small (99GB), normal (999GB), large (9.9TB), and very large (99TB). In our experiment with SS-DB, we skipped the tiny workload, for it was too small. We managed to run the *very small* and the *small* workloads. Before running the cooking and the queries, the benchmark generates the data and loads them into SciDB. The generator creates chunks of the data in parallel on each node of the cluster. The loading also is performed in parallel. Table 4.3 displays the generation and loading time for the two workloads on DAS4, OpenNebula, and AWS. Figure 4.9 shows the execution time for observation detection, observation grouping, and the queries. Note that the times are missing for the query *Q3*. This query had a bug in its implementation in SS-DB (it has been defined for an older version of SciDB). We tried to fixed it, but we were unsure of our solution, so we decided to exclude it from our experiments in order to avoid misleading results.

In SS-DB, each query is executed five times with a different slab argument. Basically, each query involves an intersection of a given slab with the input images, observations, or groups. For some slabs the resulting intersection set can be small, while for others it can be large. The execution time of the query is influenced very much by the size of the resulting intersection set. Therefore, in Figure 4.9, we depict for each query the sum of the execution times of its five invocations.

When analyzing the loading time, we notice a clear ordering of the three systems regarding performance, whereas for cooking and querying, there is no clear best or worst performing system. For loading, DAS4 performs around 1.03 times better than OpenNebula, while OpenNebula performs around 1.1 times better than AWS. Although DAS4 performs better than OpenNebula, the differences in execution time are minor. As said above, the loading is performed in parallel. That is, each

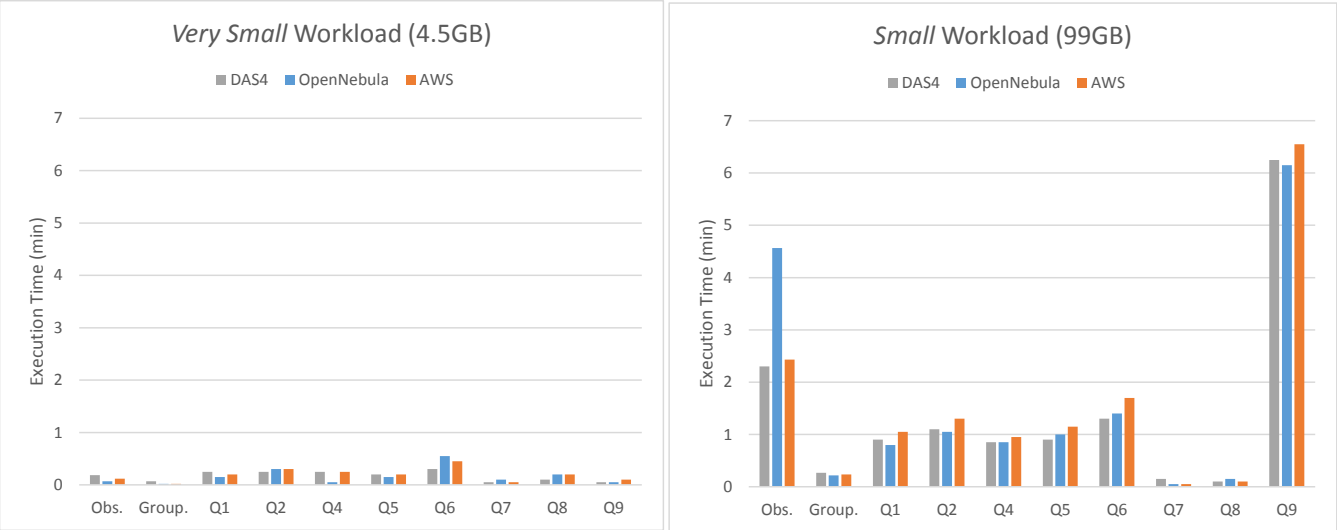

Figure 4.9: The execution time of the SS-DB benchmark with the *very small* and *small* workloads on DAS4, OpenNebula, and AWS.

cluster node has a chunk of the data and loads it into SciDB. Because the chunks have equal sizes on all the machines, the loading simply occurs internally on each node, so no data gets sent on the network. Therefore, we think that the network plays little role in the performance differences between OpenNebula and DAS4. So we think that the overhead can only come from CPU virtualization. Because of the small execution time differences between OpenNebula and DAS4, we conclude again that CPU virtualization introduces little overhead.

It is surprising in a pleasant way that while the data is large (99GB) and the loading time takes around 75 minutes, the querying execution time is very small (around one minute). Such high performance is due to SciDB distributing array chunks across multiple nodes and running the operations and user defined functions automatically in parallel on the processing nodes, resulting in little network communication. For such short execution times, we cannot reliably compare the three systems. As said above, we see that for some queries, one system performs the best, while for others the same system performs worse than the other two systems, and the differences are very small. We think that the short execution time differences for the queries come from slight random changes in the state of the processing nodes during the experiments (e.g., some random caching may influence the performance, or some other system processes may add some small additional load to the processing node for short periods of time). Therefore, we cannot draw a clear conclusion which system performs better for querying and what are the overhead sources.

We hoped that for larger data, the querying times would increase and we would be able to clearly see which system performs better than the others. That is why we

hoped that we would be able to run the *normal* workload (999GB). However, when trying to execute the *normal* workload, the generation took around four hours, whereas for loading, we waited 10 hours and it did not finish. This was unexpected, considering that in the paper describing SS-DB [18], the authors report that they manage to load the *normal* workload in around one hour. They used a slightly larger cluster than ours though, that is, they used 10 nodes, while we used 6. Moreover, they used a non open-source version of SciDB. Nevertheless, we find the slow loading a significant obstacle in the process of analyzing scientific datasets larger than 1TB. However, from the short querying execution times for the *small* workload, we estimate that SciDB would be able to query datasets in the order of hundreds of gigabytes (maybe even terabytes) in the order of minutes, assuming that the loading has been performed.

Overall, we think that the SS-DB benchmark reflects very well the real-world scenarios in the scientific computing world. However, its plain English specification is a major obstacle in it getting adopted by the scientific community. Moreover, its SciDB implementation is far from production quality and it required significant effort from us in order to be able to run it.

4.5.1 Scientific Computing Conclusions

To summarize, the following are the most important conclusions resulting from our experimenting with the SS-DB benchmark on top of SciDB:

- For loading, DAS4 performs 1.03 times better than OpenNebula, while OpenNebula performs 1.1 times better than AWS. The overhead of OpenNebula over DAS4 comes from CPU virtualization in this case, and it proves to be very little.
- The querying is performed in around one minute for 99GB, and we estimate that SciDB would be able to query datasets of hundreds of gigabytes in order of minutes on clusters of 6 processing nodes.
- Because of short querying execution times, we cannot make a performance comparison of the three systems regarding their ability to query scientific data.
- We find the SS-DB benchmark representative for scientific computing, but we find its lack of a good reference implementation a significant drawback.

Chapter 5

Conclusion

In this chapter, we present the conclusions of our work and propose some ideas for future work. In Section 5.1, we provide a summary of our work in this thesis. In Section 5.2, we present a list of the most important conclusions resulting from our experiments. Finally, in Section 5.3, we list a number of ideas for future work.

5.1 Summary

The accelerated increase of the world data has been posing significant challenges to store and process such large data. Distributed systems, and in particular clouds, have proven to be viable solutions to processing Big Data. Although clouds are being used for this purpose, little research has been done to stress-test and benchmark them with Big Data applications. This is challenging considering the very broad range of Big Data applications. The research question, introduced in Chapter 1, we address is: how cluster and cloud systems perform when processing Big Data? Specifically, we target one cluster system, the Distributed ASCI Supercomputer 4, and two cloud systems, OpenNebula and Amazon EC2.

In Chapter 2, we started by providing an overview of cloud computing and Big Data domains, and then, we identified three Big Data sub-domains: graph processing, machine learning, and scientific computing. In the same chapter, we presented several papers focusing on defining a generic Big Data benchmark. Moreover, we presented efforts that targeted benchmarking specific Big Data applications (e.g., MapReduce applications). We started from this previous work in order to define our approach to benchmarking clouds with Big Data applications.

In Chapter 3, we presented in depth the steps we took in order to benchmark and stress-test our three systems with Big Data applications. We started by presenting the three systems we benchmark: DAS4, OpenNebula, and AWS. To address our research question, we identified representative applications for each of the three Big Data sub-domains presented in Chapter 2. Specifically, we ran the Graph500 benchmark and Apache Giraph applications for graph processing, Apache Mahout applications for machine learning, and the SS-DB benchmark on top of SciDB for

scientific computing. For each Big Data sub-domain, we started by describing in general terms the algorithms, and then presented the selected applications.

The results of the experiments we ran, have been presented in Chapter 4. We ran the Graph500 benchmark, Apache Mahout applications, and the SS-DB benchmark on all the three systems, and discussed the performance differences. Apache Giraph could only be run on DAS4. We tried to identify the specification differences between the three systems that cause differences in the execution time. Moreover, we analyzed how the applications scale with the number of computing nodes (more exactly, the number of cores) and with the data size. We tried to identify the limits of each application and the causes for those limits. From the way the applications scale with the number of nodes, we tried to extrapolate what would their behavior be on clusters larger than the ones we used in our experiments.

5.2 Conclusions

We presented in Chapter 4 detailed comments on the results for each experiment we ran. Here, we reiterate the most important conclusions.

- The cluster system, DAS4, performs significantly better than the two cloud systems: up to 11 times faster for graph processing and up to 2 times faster for machine learning and scientific computing. The overhead of OpenNebula over DAS4 comes mostly from network virtualization, and less from CPU virtualization.

- Graph algorithms rely strongly on fast communication systems. This caused the kernel of the Graph500 that stress-tests the communication system (i.e., the validation step), to be unusable in our experiments. Moreover, the intense use of the network by the graph algorithms highlights the overhead introduced by network virtualization in clouds.

- Machine learning algorithms implemented with MapReduce perform most of their computation in memory and do not send many network messages, which makes DAS4, OpenNebula, and AWS perform very similar (i.e., very small differences in execution time).

- Array-based scientific data processing is highly parallel. Specifically, the most of the work can be done by each processing node locally and some aggregation is done at the end of the computation, thus resulting in little network communication and very short query times. This gives array-based database management systems the capabilities to query very large datasets (hundreds of gigabytes) in order of minutes. However, the user pays this price with very long loading times.

- In one hour, on a cluster of 8 computing nodes (8 cores each), graph algorithms can process up to 130 millions vertices (30GB of data), machine

learning applications can process tens of millions of text documents (100GB of data), while scientific processing can query multi-dimensional array data in the order of hundreds of gigabytes.

5.3 Future Work

In this thesis, we experimented with a large range of Big Data applications and created an overall picture on how clouds behave for each Big Data domain. However, we realize that we are far from having exhausted the research questions in this area. There is still much room left for research, so below we present three ideas for future work.

- *More metrics.* Because we covered a broad range of applications, we only had time to use one performance metric in our experiments, i.e., the execution time. One could try to narrow down the number of applications, say to only one application area, but use more metrics. For example, one could measure the CPU, memory, and I/O bandwidth usage.

- *Create a benchmark.* In our work, we focused on getting the performance results of the executed applications and not on how we run the experiments, that is, we did not focus on the reusability of the auxiliary software we created to automate the process of running the experiments. It would be interesting to develop some software that can automatically install the applications, and run the applications with increasing data size. This is a very challenging problem because of the very diverse nature of applications we used, but still some thinking can be done to push this forward.

- *Explore scientific computing deeper.* For graph processing and machine learning, we think we covered some of the most representative applications. For scientific computing though, we think there is still much room for exploration. First, the execution times for the SS-DB queries were very small, so it would be interesting to do something about the loading time such that further larger data can be queried. Second, one may try to experiment with other application types in scientific computing, e.g., high-energy physics applications.

Bibliography

[1] Apache Giraph. http://giraph.apache.org/.

[2] Apache Mahout. http://mahout.apache.org/.

[3] Forest Dataset. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype/.

[4] Hive. http://hadoop.apache.org/hive/.

[5] Pig Mix benchmark. https://cwiki.apache.org/pig/pigmix.html.

[6] Wikipedia dataset. http://dumps.wikimedia.org/enwiki/.

[7] Alexander Alexandrov, Berni Schiefer, John Poelman, Stephan Ewen, Thomas O. Bodner, and Volker Markl. Myriad: parallel data generation on shared-nothing architectures. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, 2011.

[8] Michael Armbrust et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] Henri Bal et al. The distributed ASCI supercomputer project. *ACM SIGOPS Operating Systems Review*, 34(4):76–96, 2000.

[10] Chaitanya Baru, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. Benchmarking Big Data Systems and the BigData Top100 List. *Big Data*, 1(1):60–64, 2013.

[11] Chaitanya Baru, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. Setting the direction for Big Data benchmark standards. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 197–208. Springer, Lecture Notes in Computer Science 7755, 2013.

[12] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, 2009.

[13] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating MapReduce performance using workload suites. In *Proceedings of IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2011.

[14] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for machine learning on multicore. *Advances in Neural Information Processing Systems*, 19:281, 2007.

[15] Edgar F. Codd, Sally B. Codd, and Clynch T. Salley. Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate. *Codd and Date*, 32, 1993.

[16] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD skills: new analysis practices for big data. In *Proceedings of the Very Large Data Bases Conference*, 2009.

[17] Philippe Cudré-Mauroux et al. A demonstration of SciDB: a science-oriented DBMS. In *Proceedings of the VLDB Endowment*, 2009.

[18] Philippe Cudre-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Samuel Madden, Michael Stonebraker, Stanley B. Zdonik, and Paul G. Brown. SS-DB: A Standard Science DBMS Benchmark. *(submitted for publication)*, 2012.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Bo Dong, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li, and Ying Li. A novel approach to improving the efficiency of storing and accessing small files on Hadoop: a case study by PowerPoint files. In *IEEE International Conference on Services Computing (SCC)*, 2010.

[21] Chris Douglas and Hong Tang. Gridmix3: Emulating production IO workload for Apache Hadoop. In *Proceedings of the Conference on File and Storage Technologies*, 2010.

[22] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Morgan Kaufmann, 2003.

[23] John Gantz and David Reinsel. The digital universe decade - are you ready? *External Publication of IDC (Analyse the Future) Information and Data*, pages 1–16, 2010.

[24] Vivekanand Gopalkrishnan, David Steier, Harvey Lewis, and James Guszcza. Big data, big business: bridging the gap. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, 2012.

[25] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[26] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *ACM Special Interest Group on Management of Data Record*, 1994.

[27] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing*, 2005.

[28] Adam Jacobs. The pathologies of Big Data. *Communications of the ACM*, 52(8):36–44, 2009.

[29] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, 1991.

[30] Xuhui Liu, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *IEEE International Conference on Cluster Computing*, 2009.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data*, 2010.

[32] Dejan Milojičić, Ignacio M. Llorente, and Ruben S. Montero. Opennebula: A cloud management tool. *Internet Computing, IEEE*, 15(2):11–14, 2011.

[33] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and Jim Ang. Introducing the Graph500. *Cray Users Group*, 2010.

[34] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the ACM International Conference on Management of Data*, 2008.

[35] Owen O'Malley. Terabyte sort on Apache Hadoop. Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop.pdf, 2008.

[36] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM International Conference on Management of Data*, 2009.

[37] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the ACM International Conference on Management of Data*, 2002.

[38] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of IEEE International Conference on Data Engineering*, 2010.

[39] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[40] Fei Xu, Kevin Beyer, Vuk Ercegovac, Peter J. Haas, and Eugene J. Shekita. $E = MC^3$: managing uncertain enterprise data in a cluster-computing environment. In *Proceedings of the ACM International Conference on Management of Data*, 2009.