Checkmetrix: Auto-scaling of Hadoop Clusters based on Mesos metrics

Authors: Dominik Harz, Bouke Nederstigt and Stephan Plat, *Computer Science, EEMCS, TU Delft*
E-mails: [D.L.Harz, B.Nederstigt, S.J.Plat]@student.tudelft.nl
Course instructors: Alexandru Iosup and Dick Epema, *PDS Group, EEMCS, TU Delft*
E-mails: [A.Iosup, D.H.J.Epema]@tudelft.nl
Lab assistant: Bogdan Ghit, *PDS Group, EEMCS, TU Delft*
Email: B.I.Ghit@tudelft.nl

***Abstract*—Checkmetrix brings data-analytics and connectivity possibilities within the reach of small retailers. With this solution small businesses store their sales information centrally in a cloud environment and receive periodic as well as on-demand reports based on metrics. To enable the creation of reports, MapReduce jobs are created. As reports can be received any time, an automatic scaling and provisioning of virtual machines (VMs) is created to cope with the unpredictable demands. The auto-scaling is further used to reduce the number of VMs when the cluster is idle. We show in experiments that auto-scaling of Hadoop clusters is possible. We further elaborate that the time to execute MapReduce jobs is decreased by up to 13% within busy clusters due to auto-scaling. We further present that the cluster scales down during idle times while remaining a consistent Hadoop Distributed File System (HDFS).**

***Keywords*—Hadoop, MapReduce, Scaling, HDFS, Amazon**

I. Introduction

Note: We have conducted this project as discussed with Alexey Ilyushkin with an actual startup (Checkmetrix) and therefore do not reference to the WantCloud BV company as mentioned in the lab assignment.

Current situation: Small businesses transfer their sales information based on individual sale receipts to Checkmetrix. Checkmetrix stores this sales information in CSV files, each for one small business. These files contain (1) the name of the product sold, (2) the price of the product, (3) a time stamp when the product was sold and (4) a unique ID for each small business. Checkmetrix provides reports to small businesses about their individual top products sold, the overall top products sold as well as price comparisons, which compares the individual small business to the overall average. Checkmetrix plans to expand and therefore will add more and more small businesses. The reports are provided on a regular basis, but can also be created and collected on-demand. Checkmetrix uses basic scripts to perform the analysis, which are not scalable and thus not sustainable for future growth. They want to continue to provide their reports in time with an increasing number of customers and would like to have an expandable platform to easily create new reports.

Related work:

The existing Amazon Web Services (AWS) EC2 [2] infrastructure is used as the cloud environment on which the Apache Hadoop framework [4] runs the big data analysis jobs. The creation of a Hadoop cluster into an AWS EC2 environemnt is done in a completely automated manner, allowing easy reproduction of the created environment [10] [3]. We are not the first to build a platform trying to share resources in a cluster with heterogeneous applications [8]. Although we do not implement a redundant master in this paper, some ideas for future work are presented and their features explained [9]. Much work has been done in the area of automatically scaling a cluster depending on workload demand [13] [18]. Much less work has however been done trying to downscale a Hadoop cluster. Especially practical examples are missing. Some interesting ideas on how to do this are presented in [17], but this work fails to address the issue of downscaling. This is the main contribution of our paper.

System to be implemented: In order to enable and stimulate the future growth of Checkmetrix, and to react to the growing demand of their customers, a new system setup is required. Since Checkmetrix stores the sales information of small businesses in CSV files, this filetype functions as the input for our system. All data is therefore stored in a Hadoop Distributed File System (HDFS)[5], a file system that is tuned to support large datasets. The actual processing of the data takes place in a distributed manner through the MapReduce programming model, which enables the system to run a large number of tasks in parallel on a number of Virtual Machines (VM's). The system scales automatically depending on the CPU load of the system. The system output differs for three types of actors involved. Firstly the retailers (end-users) are able to request on-demand as well as periodic reports from the webserver. At the same time administrators can derive disk, CPU and memory metrics to monitor the systems performance. Developers are able to create new jobs resulting in different outputs for reports through job submission manager. The high scalability of this system allows Checkmetrix to scale according to their customers demands, which ultimately results in cost-efficiency for Checkmetrix.

Article structure: In the next section we present the application and its requirements. Section III describes the system's design followed by the experiments in section IV to validate the requirements with the system design. The results are discussed in section V and the article is concluded in section VI.

II. Background

The application generates metrics of large datasets (CSV format) that provide end-users with information about their business performance. End-users either request on-demand reports for their custom metrics or instantly access periodically generated reports, both through a front-end web server. The latter are created on scheduled tasks through Chronos, while the on-demand reports are created with the issuing of MapReduce tasks through Marathon. Just product level data of Checkmetrix sales transaction for a single venue can easily sum up to over hundred thousands of items a year. This means there is need for a solution equipped to manage large amounts of data.

Requirements: In order to provide a sustainable solution that meets the needs of the end-users, the system should adhere to the requirements below:

1) The system should create reports automatically based on defined MapReduce jobs.
2) The system should automatically scale up when high loads occur.
3) The system should automatically scale down when the cluster is idle.
4) Statistics and logs should be provided to analyze the performance of the system.
5) The distributed file system (HDFS) should remain consistent during scaling and in the meanwhile the data integrity should not be affected.

These requirements formed the starting point for the system design as will be described in the next section.

III. System design

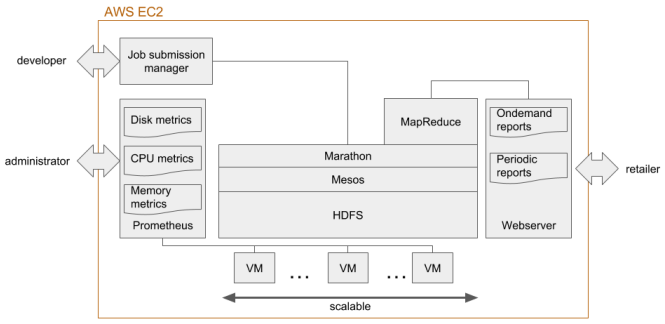

Fig. 1. Architecture overview

A. Overview

The system architecture can be characterized as a master and slave architecture managed by Mesos [8]. To enable automated and manageable configuration of the nodes Ansible is used[3]. Scaling of the slaves is managed by an auto-scaling script in Python developed during this project. By monitoring available resources (Memory, CPU and disk space) this script creates and terminates instances as needed. On top of this, actual analysis of the available data is done using MapReduce jobs written in Java. Additionally, the system exports the performance metrics data generated in Prometheus[15] through a Go script using the Prometheus API. The Go script is also developed during this project. An outline of the architecture is illustrated in figure 1.

B. Resource Management Architecture

Automation: To create a system that is manageable at scale, it is of utmost importance to automate as much as possible. This is the case during development as well as while operating the system. A key element in achieving this is to describe the infrastructure in code. This means that configuration of instance' operating system and it is applications is captured in configuration files. Using Ansible these configuration files can be used to automatically apply changes to all running instances, or recreate the entire infrastructure at the issue of a command [10].

Elasticity: Automation is an important first step in creating the manageable and reproducible system required by Checkmetrix. It is however not sufficient to use this approach for actually scaling the system while it is under pressure. Creation and configuration of a single node by Ansible can easily take over ten minutes, and it's hard to automatically trigger this task.

To be able to quickly respond to changes in resource demand it must be possible for slaves to join or leave the cluster within minutes. A basic first step to achieve this goal is creating images from the slaves. These images provide basic state and configuration, which means they can be used to quickly add more resources to the system. When taking this approach, there are obstacles. First of all, node specific configuration that needs to take place, including editing the hosts file and adding some Mesos configuration. Besides this, new Hadoop nodes must be empty and have a properly formatted file system. To achieve this we created a shell script that configures the node and starts all required services in the appropriate order on start up. This way, newly created nodes will automatically join the Mesos and Hadoop cluster.

Although this setup provides a starting point for automatically increasing resources, some challenges must be tackled when scaling the cluster down once resource usages decreases. Removing nodes from Mesos is as easy as just killing the node. Unfortunately this is not the case for Hadoop. Especially if the node contains data the entire Hadoop cluster might get corrupted if the node is removed without warning. This means nodes must be properly decommissioned, allowing the Hadoop master node to replicate any data to other nodes in the cluster, thereby preventing data loss or a corrupted file system [17].

All of the above comes together in a custom auto scaling application written in Python. The script monitors the available resources in the cluster using the Mesos API. This API provides a general overview of the CPU Disk and Memory usage in the cluster, as well as this data for specific nodes. In case resource demand becomes too high the script will increase the desired instance amount in an AWS scaling group. This will in turn trigger creating of a new instance, which will

due to the created image automatically join the cluster. If the cluster becomes underutilized the script randomly selects one or multiple nodes for removal. The selected nodes are then first decommissioned from Hadoop, then the decommissioned node is removed from the auto scaling group and finally terminated.

Performance: The solution distributes MapReduce jobs across Hadoop cluster consisting of one master (for Mesos, Marathon and Hadoop) and two to ten slaves to execute jobs in parallel and increase the systems performance.

Reliability: In the current situation reliability is mainly achieved by redundant slaves and replication of data within the Hadoop cluster. Because all data is replicated at least once it is not a problem if one of the slaves fails unexpected or gets corrupted. In case too many slave nodes die, the auto scaling application will also automatically recreate the slaves. The master and slave setup does unfortunately mean that the master becomes a single point of failure. In case the master fails, management of tasks across the cluster will become impossible, which could result in significant downtime. Luckily, Mesos is quite capable of dealing with this issue. It is possible to run the master in redundant mode as well, meaning that if one of the master instances fails, another master can take over. However, in this paper we chose for multiple reasons to not set up the cluster in such a way. First of all setting up Mesos in this manner requires the use of Apache Zookeeper. This meant adding yet another application to the already complex collection of applications in our system. Something that can obfuscate our understanding of the system while testing, and also put more pressure on our already limited time. Besides this a proper, redundant master setup requires at least three Mesos and Zookeeper masters, otherwise the system will not be able to select a new, leading master in the case of failure and/or inconsistencies [9]. This means cost of the system is increasing, since the master nodes are always up, whether tasks are running or not.

Monitoring: In order to monitor the metrics at specific nodes and the system performance as a whole, Prometheus [15] is configured on a separate node. The other nodes in the network have the node-exporter installed [19] to scrape metrics (CPU, memory, disk I/O) of tasks running on the master and slaves in the network. The node-exporter scrape mode is therefore configured in both master and slave mode. The Prometheus node stores time-series data while allowing querying of the data. A self-developed Go script pulls the data out of the Prometheus node and parses the data to a CSV file that can be processed.

C. System Policies

Currently some rudimentary policies for managing resources are implemented in the system. First of all a simple policy based on available CPU / Disk and Memory usage is implemented to deal with scaling the instances. This policy checks whether the amount of available resources is within a set threshold, and if this is not the case it will scale up or down. Although section IV shows this already returns good results it could be better. Especially in the case of high resource demand the system might not respond as quickly as possible. This is due to the fact that Mesos might not allocate resources to a framework if it demands too many resources. This problem can be dealt with by letting the auto scaling script also monitor the requested resources, and if these are above a certain threshold add instances to the instance pool.

Besides this the default Mesos setup uses Dominant Resource Fairness (DRF) to allocate resources to different users [7]. This algorithm allows allocating different resource types within a system, building upon the more generally known max-min fairness algorithm. In practice this comes down to comparing a framework's share of its dominant (most requested) resource to the dominant resource share in the entire cluster. This allows the algorithm to assign each instance its fair share. Thus, this approach is useful in a heterogeneous environment. It does offer Checkmetrix the possibility to easily implement additional features (such as machine learning) in a later stage, and in that case fully leverage the Mesos' fine grained resource sharing capabilities.

IV. Experimental results

A. Experimental setup

We implemented our Hadoop cluster with auto-scaling capabilities within Amazon Web Services [2]. Servers are either t2.micro instances with 1GB memory, 2 Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz CPU and 8GB SSD or t2.medium instances with 4GB memory, 2 Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz and 40GB SSD. We created one monitoring server, installed and configured prometheus-0.15.1 on a t2.micro instance [15]. One master server is running Hadoop 2.6.0, Mesos 0.21.0, Marathon 0.7.5 and MapReduce 2.6.0 on a t2.medium instance. It is configured as Mesos master server, Hadoop namenode and journalnode. Initially four additional t2.micro instaces are within the cluster and are configured as Hadoop datanodes as well as Mesos slaves. Each server is running on Ubuntu Server 14.04 LTS and has OpenJDK Runtime Environment 7.79 installed.

To enable auto-scaling a Python script runs on the master servers. As described in section III it issues on scaling decisions based on the CPU utilization metric as reported by the Mesos master.

Three standard MapReduce test jobs are used to evaluate different aspects of the cluster including CPU and HDFS[14] as well as our own CSV MapReduce job. *mrbench* runs a number of very small jobs in the cluster with little effect on the HDFS[11]. Thereby, high CPU loads can be generated and monitored. *TestDFSIO -write* writes a specified number of files with a defined size to the Hadoop cluster. On the other hand *TestDFSIO -read* reads a specified number of files with a defined size from the Hadoop cluster. Both are utilized to test the read/write performance of the HDFS[12]. Furthermore, the cluster is tested with our CSV parser. It reads the product names from CSV files and counts the products that appear the most. Based on this, the products are sorted according to their frequency and the most common products are written to a simple text file. We have written a Python script which creates near real world data. The CSV file created includes a random number of products with random prices, a random time-stamp

TABLE I. OVERVIEW OF MAPREDUCE JOBS USED TO TEST THE CLUSTER

Name	CPU [%]	Memory [MB]
mrbench 50 jobs	0.3	128
mrbench 500 jobs	0.6	256
TestDFSIO -write 500MB	0.3	128
TestDFSIO -write 5GB	0.6	512
TestDFSIO -read 500MB	0.3	128
TestDFSIO -read 5GB	0.6	512
CSV-parser 5 files	0.3	128
CSV-parser 50 files	0.6	512

of the sale and an ID for the venue where it was sold. The jobs are created in Marathon, which requires to assign them with a defined CPU share and a maximum amount of memory to use. Table 1 presents the parameters used to issue the MapReduce commands from Marathon.

B. Experiments

Firstly, tests are conducted to show that **auto-scaling** of the Hadoop cluster is working as designed. As the auto-scaler is monitoring the CPU utilization of the cluster, *mrbench* jobs are used to generate high CPU load on the cluster to trigger auto-scaling. In this test multiple jobs are issued and are executed on the cluster. Each jobs consists of one *mrbench* command, each resulting in 50 small tasks. As shown in figure 2, the cluster has initially four jobs running totaling to 200 tasks running in parallel. The auto-scaler checks the CPU utilization every four minutes, hence the number of jobs was stepwise increased to 32. The maximum number of slaves in the cluster is set to ten. Consequently, the number of jobs was reduced after ten slaves were operating in the cluster to trigger the down-scaling. In figure 3 the load was increased and decreased in bursty patterns from four to 32 and back. In both experiments the total number of slaves in the cluster and the execution time of jobs is monitored. Within the steady pattern *mrbench* executed the small tasks with a mean time of 10.3s and a standard deviation of 3.1s. In the bursty pattern setup *mrbench* executed the small tasks with a mean time of 10.6s and a standard deviation of 6.7s.

Secondly, experiments are conducted to evaluate the **Resource Utilization** of the cluster and the assignment of tasks. Again, the focus lays on the CPU utilization. Hereby, four *mrbench* jobs are issued with each 500 small tasks. The number of jobs is continuously increased to force the cluster to scale up to ten slaves. After ten slaves are created the number of jobs is continuously decreased to four again. Figure 4 and figure 5 show the CPU utilization in %, the available memory and the disk I/O of the master and the slaves respectively. Hereby, the CPU utilization of the slaves as well as the available memory is summed up over all slaves leading to a maximal possible percentage of 1000% with a cluster size of ten slaves. As the disk I/O is relatively constant due to using a *mrbench* job, this will not be further discussed. During the experiment a total of 14 slaves are used.

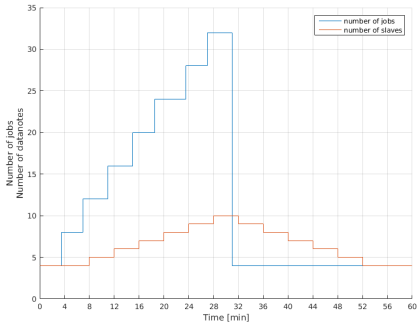

Fig. 2. Increasing and decreasing number of jobs to trigger autoscaling of the Hadoop cluster

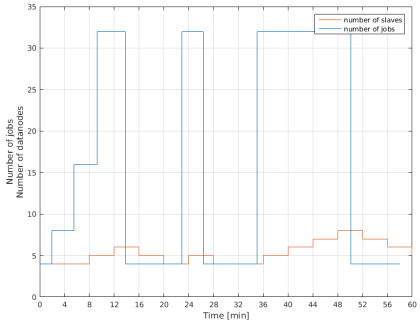

Fig. 3. Bursty increasing and decreasing number of jobs to trigger autoscaling of the Hadoop cluster

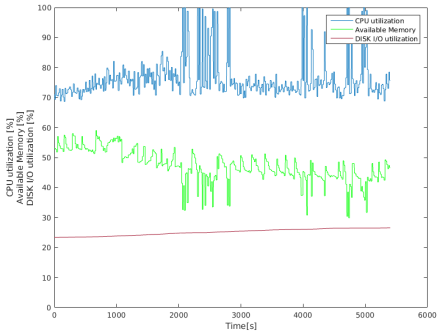

Fig. 4. CPU, memory and disk I/O of the Marathon, Mesos and Hadoop master

To understand the load distribution on the individual slaves figure 6 illustrates the mean CPU utilization, the 25% and 75% percentiles as well as outlier measurements for each of the 14 slaves. Only data points during the slaves active time (when joining the Mesos cluster until leaving it) are collected and considered for this figure.

Thirdly, experiments to determine the impact on the **MapReduce job runtime** by auto-scaling are executed.

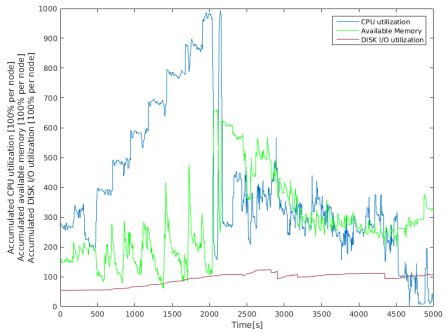

Fig. 5. CPU, memory and disk I/O of the Mesos and Hadoop slaves

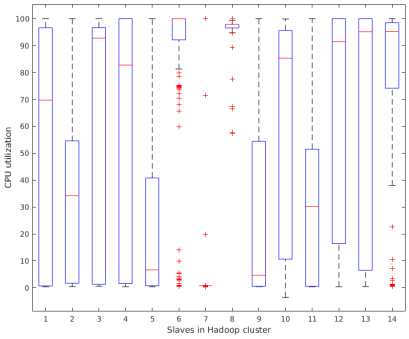

Fig. 6. Varying CPU utilization at Mesos and Hadoop slaves

Hereby, the cluster is tested with *TestDFSIO -write* and *TestDFSIO -read* jobs and our own CSV-parser job. As a first step, each job is run ten times with five small files (500MB). Each time the runtime is measured to determine the mean runtime and standard deviation of each of the ten jobs. The cluster has four slaves initially in both settings, auto-scaling disabled and enabled. Figure 7 shows the mean runtime and standard deviation of the three job types with auto-scaling disabled and enabled. The mean runtime is similar, whether auto-scaling is enabled or disabled. The experiments are repeated with running the jobs again ten times. This time the file size for *TestDFSIO -read* and *TestDFSIO -write* is set to 5GB. The CSV-parser is issued with 50 files each 500MB. Increasing the files and file size increases the runtime of jobs. The mean job runtime and standard deviation is shown in figure 8 for each of the three job types with auto-scaling and disabled. In these experiments four *mrbench* jobs with 500 tasks each are constantly run on the cluster (for both auto-scaling disabled and enabled) to prevent it from scaling down below four slaves, when auto-scaling is enabled. The large jobs complete up to 13% faster when auto-scaling is enabled.

The **charged time and charged cost** for our experiments are summarized in table 3. The master instance and the Prometheus server are continuously running. Hence, they are

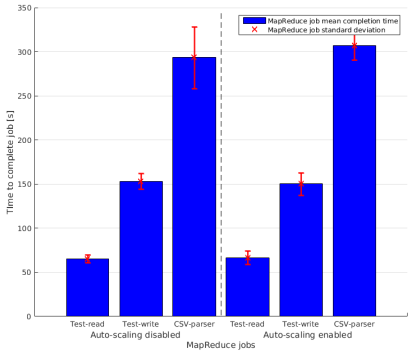

Fig. 7. Mean runtime and standard deviation of small jobs with auto-scaling enabled and disabled. Jobs are issued in both cases over Marathon. Test-read is a standard read TestDFSIO job with 5 files and 500MB data per file. Test-write is a standard write TestDFSIO job with 5 files and 500MB data per file. CSV-parser refers to our own CSV job which reads 5 files with each 500MB data and writes a few KB.

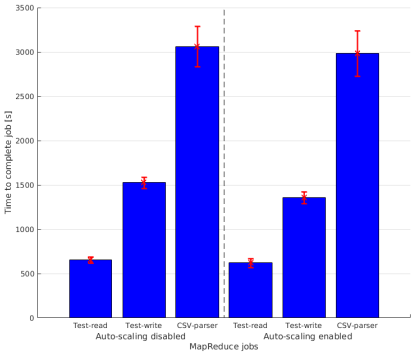

Fig. 8. Mean runtime and standard deviation of large jobs with auto-scaling enabled and disabled. Jobs are issued in both cases over Marathon. Test-read is a standard read TestDFSIO job with 5 files and 5GB data per file. Test-write is a standard write TestDFSIO job with 5 files and 5GB data per file. CSV-parser refers to our own CSV job which reads 50 files with each 500MB data and writes a few KB.

not scalable and therefore not considered in the calculation. The rows cover the additional time and cost for the instances used in the experiments. As auto-scaling provisions additional servers the costs are higher compared to a static cluster. However, improvements in terms of performance are only reached when jobs become larger. The economic impacts need to be further researched with running very large jobs (e.g. above 5TB of data) on large clusters. In small environments waiting a couple of seconds more is from an economic perspective more feasible.

V. Discussion

Auto-scaling as designed in Checkmetrix is working as shown in the experiments. During the experiments the HDFS remained healthy (i.e. *hdfs fsck /* returned status HEALTHY) during up-scaling and down-scaling. Two major points can be further improved. Scaling is not responding to bursty loads

TABLE II. OVERVIEW OF CHARGED TIME AND COSTS DURING THE EXPERIMENTS

Experiment	Time	Costs	Total
Auto-scaling continuously	6.4hrs	€0.10	€0.64
Auto-scaling bursty	5.1hrs	€0.10	€0.51
Resource utilization	10.7hrs	€0.10	€1.07
Runtime small files static	0.4hrs	€0.10	€0.04
Runtime small files scaling	0.4hrs	€0.10	€0.04
Runtime large files static	5.3hrs	€0.10	€0.53
Runtime large files scaling	6.7hrs	€0.10	€0.67

as the cluster is only slowly scaling up. However, changing a large percentage of datanodes in a Hadoop cluster is relatively difficult, as the HDFS might become inconsistent due to missing replications or blocks[4]. Secondly, the current auto-scaler does not take into account memory or disk loads. Hence, jobs with high disk I/O impact (i.e. *TeraSort* or *TestDFSIO*) will not result in a fast scaling of the cluster.

Utilization of resources as examined in the experiments differs among the slaves in the cluster. While the overall load of the cluster constantly increases or decreases the master has to take care of assigning the different job instances to individual slaves. The slaves start the specified Hadoop job and execute it. From the results presented in figure 6 the resource scheduling of Mesos with having just one type of job is not optimal. One group of slaves has a load above 70% and the other group below 30%. After investigating the logs slaves 5, 7 and 9 were provisioned and shortly joined the Mesos cluster, but were removed shortly after since their appeared issues in their configuration. Hence, they were replaced by newly provisioned slaves. These provision failures occured during the downscaling of the cluster, when there were still bursty job loads affecting the CPU utilization metric in Mesos. Thus, the auto-scaling needs to be more robust against bursts when it is downscaling the cluster.

Within the experiments it is shown that the **MapReduce job runtime** is only affected when the job is actually long running. Hereby, other jobs can be assigned to newly provisioned slaves and reduce the workload of one slave. As the jobs were started from Marathon a CPU and memory share was already entered when starting the task. The running MapReduce job only takes that portion of available resources. Thus, gains in job runtime compared to non-scalable environments can only be made, if there are more jobs scheduled than the cluster resources can serve.

Suggestions for **future work** would be to build a solution for task scheduling and/or utilise Chronos in order to schedule tasks within the cluster. The challenge in the current systems configuration is that when CPU measurements take place at exactly the same moment when a lot of jobs have just finished, the clusters seems to be empty and might scale down (although a lot of jobs are still running/remaining). The scaling script should therefore initialize which slaves it wants to shutdown and inform Mesos that it should not assign jobs to those slaves any longer. Another interesting contribution to this field of research would be to extend the functionality of the cluster by differentiating between on-demand and long running tasks. To derive the on-demand metrics even faster one could think of prioritizing them over the long running tasks (periodic reports) that could be generated in the idle time (i.e. over night). Besides that, the reliability of the system could be increased by also running the masters in distributed way so that another master node could take over the operation of the master in case it fails. In order to do so, Zookeeper should be configured and set up properly within the network. Furthermore the impact of the system's scalability could be even further increased by storing the data from the slaves on long term storage (i.e. Amazon Simple Storage Service [1]). In that way even nodes that still contain data (but are underutilized regarding to the CPU) can be removed without any risk on data loss.

VI. CONCLUSION

Auto-scaling of Hadoop clusters with job management over Marathon and Mesos is possible. In the system design the basis to run MapReduce jobs within fixed resource boundaries on a automatically scalable platform is set. Thereby, existing products, libraries and code is enhanced and connected to create a new platform. In experiments it is shown that the current implementation of auto-scaling works best with continuous changes of loads. Furthermore, the cluster needs to run a large number of long-running jobs to make advantage of the auto-scaling feature. The platform can be further enhanced by integrating platforms for machine learning (e.g. SciKit with Hadoop streaming API [6]). However, with the given time the here implemented and tested auto-scaling solution in itself was quite demanding to realize. There are numerous issues in keeping the HDFS consistent especially during downscaling. Furthermore, the number of different applications used and the required configurations as well as the code written was prone to various errors, which took a high bug-fixing effort. Overall, we are confident that we create herewith a showcase, which can be inspiring to others to continue our work. As the project was realized for the Amsterdam based startup Checkmetrix the code and knowledge will definitely be of further use.

APPENDIX
TIMESHEETS

TABLE III. TIME SPENT ON PROJECT

Parts	Bouke	Dominik	Stephan
total-time	62 hours	57 hours	54 hours
think-time	8 hours	4 hours	6 hours
dev-time	30 hours	25 hours	22 hours
xp-time	8 hours	16 hours	10 hours
analysis-time	7 hours	5 hours	5 hours
write-time	4 hours	4 hours	4 hours
wasted-time	5 hours	3 hours	7 hours

REFERENCES

[1] Amazon Web Services, *Data Archive*, https://aws.amazon.com/archive/, 2015.

[2] Amazon Web Services, *EC2*, https://aws.amazon.com/de/ec2/, 2015.

[3] Ansible, *Ansible*, https://github.com/ansible/ansible, 2015.

[4] Apache, *Hadoop*, https://hadoop.apache.org, 2015.

[5] Apache, *HDFS Architecture Guide*, https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2015.

[6] Cloudera, *SciKit*, http://blog.cloudera.com/blog/2013/01/a-guide-to-python-frameworks-for-hadoop/, 2013.

[7] Ghodsi, Ali, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In: *NSDI*. 2011. p. 24-24.

[8] Hindman, B, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI* (Vol. 11, pp. 22-22).

[9] Hunt, Patrick, et al., ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: *USENIX Annual Technical Conference*, 2010. vol. 8, p. 9.

[10] Huttermann, Michael, Infrastructure as Code. In: *DevOps for Developers*, Apress, 2012. p. 135-156.

[11] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin-gyu Kim, Hyungsoo Jung and Heon Y. Yeom, *MRBench : A Benchmark for Map-Reduce Framework*, 14th IEEE International Conference on Parallel and Distributed Systems, 2008.

[12] Lynch, Daniel, *Running DFSIO mapreduce benchmark test*,

[13] Mao, Ming; Li, Jie; Humphrey, Marty. Cloud auto-scaling with deadline and budget constraints. In: *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE, 2010. p. 41-48.

[14] Noll, Michael, *Benchmarking and Stress Testing an Hadoop Cluster*, http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench, 2011.

[15] Prometheus, *Prometheus*, http://prometheus.io/docs/introduction/overview/, 2015.

[16] Prometheus, *Prometheus Mesos Exporter*, https://github.com/prometheus/mesos_exporter, 2015.

[17] Romer, Toomas, *Autoscaling Hadoop Clusters*. 2010. PhD Thesis. MSc thesis, University of Tartu.

[18] Varia, J. (2011). Best practices in architecting cloud applications in the AWS cloud. Cloud Computing: Principles and Paradigms, 459-490.

[19] Prometheus, *Prometheus Mesos Exporter*, https://github.com/prometheus/mesos_exporter, 2015.