HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

Software-Defined Storage for Big Data on top of the OpenStack platform

D5.1 System Monitoring Design and Preliminary Evaluation

Due date of deliverable: 31-12-2015
Actual submission date: 31-12-2015

Start date of project: 01-01-2015 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.1
Number of pages	23
WP/Task related to this document	WP5 / T5.1
WP/Task responsible	EUR
Leader	Francesco Pace
Technical Manager	Pietro Michiardi
Quality Manager	Marc Sánchez
Author(s)	Francesco Pace, Daniele Venzano, Marco Milanesio, Pietro Michiardi
Partner(s) Contributing	URV, GDP, BSC and IBM
Document ID	IOSTACK_D5.1_Public.pdf
Abstract	This deliverable describes the monitoring tools architecture that will be used to define models to reason about deployment strategies. It also presents preliminary results on the system performance when analytic applications frameworks are deployed according to naive strategies.
Keywords	Monitoring, Analytics, Benchmarking

History of changes

Version	Date	Author	Summary of changes
1.0	31-12-2015	Francesco Pace, Daniele Venzano, Marco Milanesio, Pietro Michiardi	First version
1.1	15-11-2016	Francesco Pace, Daniele Venzano, Marco Milanesio, Pietro Michiardi	Improvement of Section 2 and 2.1 according to project reviewers.

Table of Contents

Executive summary

The underlying vision of the IOStack project promotes the separation of compute and storage layer, however several issues related to data access performance arise: by breaking the principle of data locality, I/O bottlenecks (both network and disk) become predominant. To address this problem we have to tackle an important aspect of the IOStack project: the monitoring system.
Measurements from the monitoring system can used both to benchmark the overall IOStack system and to continuously inform the service deployment modules for application placement and migration. This deliverable is divided in two parts: *(i)* monitoring system architecture and *(ii)* evaluation results of different deployment strategies.

After an introduction of Analytics as a Service in section 1, this deliverable describes the architecture of the monitoring system in section 2. We decided to pursue the design and implementation of the system performance monitoring along two different path. The first is by re-using generic open source tools that are already well known by the community and the second is to develop specialized tool for monitoring specific applications and workloads. Both monitoring system have been selected and developed to be as efficient as possible, in order not to pollute the measurements. More details on the open source tools are in section 2.1, while in section 2.2 we describe the software we developed to complement the open source solutions.

In section 3 we present the initial results of a study conducted to pinpoint possible bottlenecks of different deployment strategies. We ran analytic applications in different deployment configurations, using the monitoring system to record resource utilization metrics.

In the last section, 4, we present a Swift-specific monitoring service that can help the IOStack SDS controller in maintaining a correct bandwidth allocation when the number of clients increases.

1 Analytics as a Service

The objectives of this work package are to define, design and build deployment strategies for data-analytics as a service. The focus of this work is on big data frameworks like Apache Spark, which enable users to define both batch and latency-sensitive analytic jobs. IOStack promotes the separation of compute and storage layers allowing increased flexibility in meeting the variable demand of computation resources, while keeping the data storage system in a stable state. However, by separating the layers, several performance issues arise: by breaking the principle of data locality, I/O bottlenecks (both network and disk) can become predominant.

It is thus important to define appropriate application deployment strategies that aim at mitigating the aforementioned I/O problems.

- System performance measurements, which deal with the design and implementation of a framework for the continuous monitoring of system performance. Such measurements are used both to benchmark the overall IOStack system, and to automatically and continuously inform the service deployment modules for application placement and migration. The outcome of this task is a distributed software, that can eventually be deployed as an IOStack service;
- System deployment strategies. Taking into account the measurement results obtained from the previous task, it will be possible to inform the system on the performance of analytic application deployments so that different policies can be compared and chosen correctly.
- System deployment tools, which deal with the instantiation of deployment strategies. Such tools must address the problems related to an asynchronous and distributed cloud environment (and in particular, its scheduling component).

In this deliverable we present work done, mainly, over virtual machines. We are well aware of containers as a different virtualization choice that is gaining traction in the community and we are already starting to expand our work on that direction.

2 Monitoring system

Monitoring system performance involves the study of an entire system, including all physical components and the full software stack: anything in the data path, software or hardware, must be included, because it can play a crucial role in determining the overall performance. As such, a monitoring system is fundamental to characterize, from the early stages to consolidated versions of the IOStack architecture and system, the possible I/O bottlenecks and to pinpoint the root causes of performance degradation.

Today, little is known on the impact on application-level performance of placing data-intensive framework components (such as Hadoop or Spark) in different ways. Indeed, deployment strategies are bound to be ephemeral: an optimized deployment selected at a given point in time depends on the system state, which is dynamic, as dictated by the workload imposed on the system. As a consequence, the measurement system discussed in this deliverable needs to be flexible and responsive to accommodate system state variability, which may trigger re-deployment and re-configuration of currently running analytic applications. Finally – as observed for any measurement framework in general – it is important to minimize the intrusiveness of the monitoring infrastructure: this means that the measurement framework should avoid perturbing the system state.

There are two basic approaches to server monitoring: Passive and Active. Active monitoring, also referred to as synthetic monitoring, performs regular, scripted checks that simulate end-user behavior. These tests can be run from inside the network or from multiple points on the Internet. Running these checks externally replicates an end-user experience as closely as possible. Passive monitoring takes an internal approach to monitoring the performance from within the distributed cluster system. Since we want to minimize the intrusiveness, we opted to use a passive monitoring system with a centralized architecture: all the metrics are shipped to a single monitoring host. We measured the overhead introduced by such architecture on the distributed cluster and confirmed that is negligible and has no impact over the running applications.

We built a system performance monitoring solution combining off-the-shelf open source software and custom tools developed to cater to IOStack specific needs.

2.1 Open source tools

Our choice fell on three open source projects: *(i)* collectd [1], *(ii)* Carbon/Graphite [2] and *(iii)* Grafana [3]. These projects tackle three specific aspects of the monitoring system, respectively: *(i)* metrics gathering, *(ii)* metrics storage and *(iii)* metrics visualization.

Our choice has been to configure collectd to gather metrics and send them to Carbon/Graphite for storage. Grafana uses the Carbon/Graphite APIs to create rich dashboards that can be used to visualize and understand the recorded data. The criteria that we used to choose such pipeline are the following: *(i)* opensource, *(ii)* maintenance/support, *(iii)* flexibility, *(iv)* performance and *(v)* usability. Furthermore, this pipeline of services has been recently adopted by the community [4] to monitor distributed systems clusters. It is worth saying that some tools in their respective area are very similar between each other; in that case, we opted to use the most familiar, thus reducing the learning and deployment curve.

collectd is a small daemon that runs on each host to be monitored, collects statistics about the system and provide mechanisms to forward the samples via the network to be centrally aggregated. Collectd holds all the criteria that we were looking for: *(i)* it is opensource, *(ii)* it is written in C for performance and portability, *(iii)* it uses a plugin system that makes it very flexible and *(iv)* it is actively developed, supported and well documented. In our monitoring architecture, collectd, is installed on physical hosts. Virtual machines and containers can be monitored from the outside with the help of specialized plugins. Measurements for the intrusiveness of collectd showed that the overhead introduced by this tool is negligible on the distributed cluster system, in light of the previous discussion about active vs passive monitoring.

The second aspect of monitoring system is storage for samples and metrics. The most interesting projects in this field are InfluxDB[5] and Carbon/Graphite. While InfluxDB is still at the early stages of development, with unstable APIs, Carbon/Graphite has a long and strong tradition in production systems; it also allows to store metrics, both standard and custom, in an easy and efficient way.

Once metrics are gathered and stored, the third step is access, process and visualize them. This role is covered by Grafana, an open source dashboard tool used for creating complex visualizations of time series data. It provides a powerful and easy way to create, share, and explore data and dashboards from metric databases. Grafana features pluggable panels and data sources, supports Carbon/Graphite, but also InfluxDB and OpenTSB and let the user define arbitrary mathematical and statistical operations to transform the data prior to visualization.

Grafana does not come with prepared templates, so we created the necessary dashboards needed to monitor the different aspect of the IOStack platform. Figure 1 shows an overview of the monitoring system built with opensource tools; both Carbon and Grafana are hosted on a separate and dedicated machine.

2.1.1 Metrics

Collectd is a powerful tool that, with the help of plugins, is able to monitor a wide range of data sources, some of which are very important for the IOStack project. For this reason we spent time to understand the meaning of each metric generated and what information they convey about the status of the infrastructure.

With the help of collectd we measure three different resources at one minute interval: *(i)* CPU, *(ii)* Disk and *(iii)* Network. Taking samples at one minute of granularity allows Grafana to create a time series analysis of the utilization during the duration of an analytic application, without overloading the monitoring system, to better understand when the demand of resources is higher.

In the following part, we define the formula for calculating the utilization of every resource taken in consideration.

CPU utilization CPU time is accounted by the kernel according to the categories listed in table 2.1a. The time counters start from zero at system boot and are measured in jiffies, that, in the default Linux

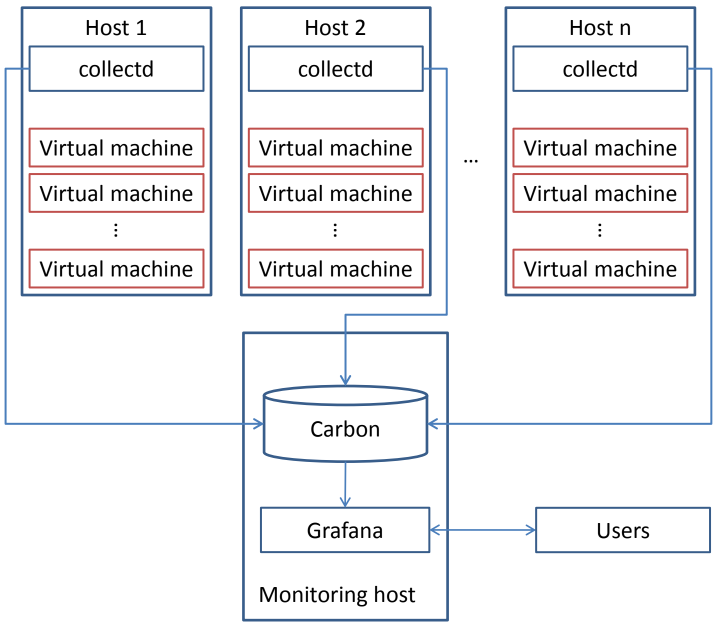


Figure 1: Monitoring Architecture.

user	time spent running user code
nice	time spent running user code for low-priority processes
system	time spent running in kernel mode
idle	idle time
iowait	time spent waiting for I/O operations to complete
irq	time spent managing interrupts
softirq	time spent managing soft interrupts
steal	time spent waiting for the physical CPU in virtualized environments

Table 2.1a: CPU accounting by the Linux kernel.

kernel configuration, correspond to hundredths of milliseconds.

We define CPU utilization as follows:

$$\text{all} = user + nice + system + idle + iowait + irq + softirq + steal \tag{1}$$

$$\text{busy} = all - (idle + iowait) \tag{2}$$

$$\text{busyTimeDelta} = busy_{t+1} - busy_t \tag{3}$$

$$\text{allTimeDelta} = all_{t+1} - all_t \tag{4}$$

$$\text{cpuUtilization} = 100 * \frac{busyDelta}{allDelta} \tag{5}$$

The last formula produces a percentage of the time the CPU has been busy in a given interval of time.

Network utilization Given the networking hardware bandwidth (for the platform at Eurecom it is 1GB/s, for example) and two counters for bytes sent and received by the network interface (respectively *bytes_sent* and *bytes_recv*, we define network utilization as follows:

$$
\begin{aligned}
\text{maxThroughput} &= \text{hardware bandwidth} && (6)\\
\text{bytes} &= bytes\_sent + bytes\_recv && (7)\\
\text{bytesTimeDelta} &= bytes_{t+1} - bytes_t && (8)\\
\text{netUtilization} &= 100 * \frac{bytesTimeDelta}{maxThroughput} && (9)
\end{aligned}
$$

If the time delta is different than one second, *maxThroughout* must be adjusted accordingly.

Disk utilization Disk utilization cannot be calculated in the same way as CPU and network utilization. The maximum throughput of a hard disk as specified by the manufacturer is reported for the disk functioning in optimal conditions, using the fastest sectors on the platters, without RAID controllers, etc. The real maximum throughput is highly dependent on the system and the location of the data on the disk platters and varies over time in an unpredictable way.

For this reason, we decided to use the same metric used by the well-known tool *iostat*[6]. They define the disk utilization as follows:

> [...] percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100%".

Mathematically, the definition is as follows, refer to table 2.1a for an explanation of the CPU time counters.

$$
\begin{aligned}
\text{numCpus} &= \text{number of cores in the system} && (10)\\
\text{iotime} &= \text{time spent doing I/O} && (11)\\
\text{cputime} &= user + nice + system + idle + iowait + irq + softirq + steal && (12)\\
\text{ioTimeDelta} &= iotime_{t+1} - iotime_t && (13)\\
\text{cpuTimeDelta} &= cputime_{t+1} - cputime_t && (14)\\
\text{disk} &= 100 * \frac{ioTimeDelta}{\frac{cpuTimeDelta}{numCpus} * 100} && (15)
\end{aligned}
$$

In this formula cputime is the sum of the times of all cores in the system. To make this quantity comparable to iotime we need to divide it by the number of cores available in the system.

cputime is taken by reading the */proc/stat* file, while iotime is taken from the */proc/diskstat* file.

2.1.2 Carbon/Graphite

Carbon is the data aggregator component we have chosen, it uses a specialized on-disk format, called Whisper, derived from the venerable RRD format. Graphite is the frontend component, that users can use to extract the data of interest from Carbon.

Carbon can be distributed in a limited fashion and comes with helper daemons that can help filter and distribute metrics incoming from the network. This tool is very well known and it is very widespread, but it has also well-known limitations caused by the amount of I/O load it can generate when then number on incoming metrics is too high. For this reason we are also keeping track of the InfluxDB project that promises a more modern approach to metric storage and querying.

Graphite is a web application that provides an API to access data stored in Carbon. It also has a limited web interface to build plots and dashboards, but for this task Grafana offers much better flexibility.

2.1.3 Grafana dashboards and panels

Grafana is a web application that is quickly becoming a standard in monitoring interfaces. It has the concepts of dashboards that group related plots. Each plot can contain a number of time series, with transformations (moving average, derivative, etc.) applied. Plots are drawn in rows of one to 4 plots each, see figure 2 for an example. An interactive editor included in Grafana can be used to build the plots and the dashboards.

We created a number of dashboards to monitor metrics related to the IOStack needs.

Compute This dashboard contains one row for every host belonging to the compute layer of analytic applications. Each row contains plots with the following time series:

1. UNIX load (short term, medium term and long term)
2. CPU utilization
3. network utilization
4. disk I/O time
5. disk throughput
6. disk utilization

Object storage This dashboards monitors the hosts assigned to run the object storage service. Each row contains plots with the following time series:

1. UNIX load (short term, medium term and long term)
2. CPU utilization
3. network utilization
4. disk I/O time
5. disk throughput
6. disk utilization
7. free disk space

Volumes This dashboards has one row for every host running the volume service. Each row contains plots with the following time series:

1. UNIX load (short term, medium term and long term)
2. CPU utilization
3. network utilization
4. disk I/O time
5. disk throughput
6. disk utilization
7. free disk space

Depending on the type of volumes in use (e.g. CEPH) more metrics can be added for more detailed service monitoring.

Tenants This dashboard is more user-oriented. There is one row of plots that shows aggregated information for a general overview and then more detailed rows, one per tenant in the IOStack platform. The plots contain:

1. virtual cores used
2. virtual cores allocated
3. network utilization
4. disk utilization

Figure 2 shows the tenants dashboard.

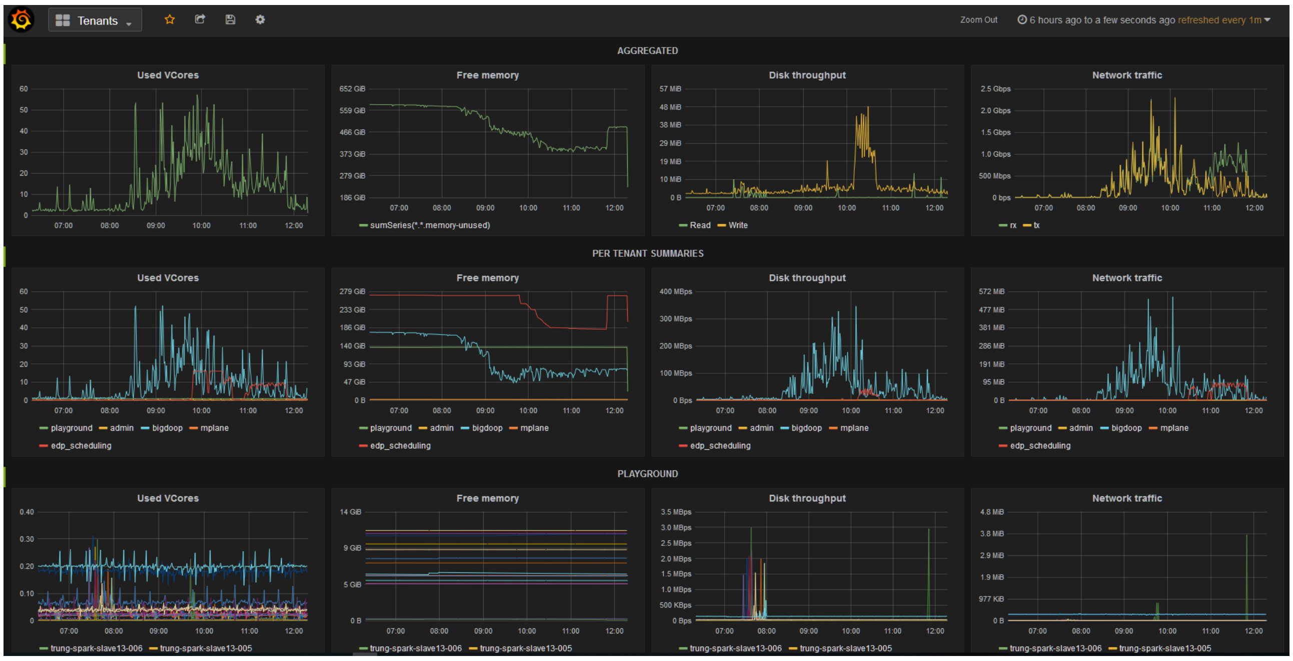


Figure 2: The top of the tenants dashboard. The first row contains aggregated information for all tenants, the second row per-tenant details and the third row a detailed view of the virtual machines of a single tenant.

2.2 In-House Tools

While using off-the-shelf open source tools satisfies the need to have a solid environment to monitor the platform in its entirety, there is also the need for tools to provide very specific information linked to the analytic application being run in the virtual environment. To *(i)* make sure we satisfy our needs and *(ii)* we fully understand the measurement being recorded, we decided to develop two IOStack-specific tools.

The first tool records system and application metrics at one second interval. It records the data in a very simple csv format, while the analytic application is running. It has been developed to have the smallest resource usage possible, so that measurements would not be perturbed by its activity.

The second tool is a parser and aggregator. It reads the data recorded by the first tool and combines it with profiling data generated internally be the application to produce time series that can later be analyzed and plotted.[1]

3 Preliminary performance analysis of Analytics applications

During the last few years, more and more scalable computing frameworks are being deployed on virtual environments, shifting from clusters on top of bare-metal machines to clusters on top of virtual machines. The reason behind this choice is connected to the benefits of virtualization; virtual machines are more flexible, cheaper, can be provisioned easily and in different sizes and can be destroyed once their use comes to an end, freeing resources and saving money.

Many big companies offer the possibility to use a virtual cluster to run a variety of data-intensive applications: Amazon Web Services (AWS) [7], DataBricks Cloud[2] [8] , Cloudera Cloud[1] [9] and Google Cloud Hadoop [10] are all noteworthy examples.

A typical distributed analytic application is composed of two major layers: the compute layer and the data layer. Both are composed by a number of hosts running a data-intensive framework (the compute layer) and a distributed filesystem or object storage service (the storage layer). In

[1]For example Spark can be configured to generate a "Spark events" stream containing a very detailed view of Spark-specific details about I/O, task and job performance.

[2]Solution hosted on Amazon Web Services (AWS) [7]

virtualized environments, like the ones listed above, compute and data layers are often separated.

As a result, the traditional concept of data locality, moving computation to the data, is challenged. Taking the example of Amazon Elastic MapReduce service, data is stored in S3 and the computation happens in EC2 instances, that run in completely different physical hosts.

Today, little is known on the impact on application-level performance of breaking the data locality rules. With this study we aim to fulfill the need for a rigorous study of how analytic applications behave when different deployment strategies are applied.

We define Compute-to-Data path as the path that a compute instance has to follow in order to retrieve the data needed from storage. We analyzed different deployment strategies, taking measurements with the tools described in the previous sections, to understand how data-intensive application performance is affected by different Compute-to-Data paths. We found that indeed a correct placement of compute and data layer leads to a lower analytic application run time.

In the rest of this deliverable we go into the details of the measurement study, the methodology, the scenarios, the workloads and the results.

3.1 Related work

The work done by Kay et al.[11], among other considerations, questions the importance of data-locality, by virtue of the low I/O utilization of some workloads. Their finding are based on a series of test performed on Amazon AWS [7] in a configuration that hosts data and compute layer on the same virtual machines. They used two workloads, TPC-DS[12] and BDBench in two different configurations: disk-intensive and memory-intensive.

With the help of *(i)* an analysis performed on network and disk block time and *(ii)* resource utilization, they find that analytic jobs are CPU-bound and that the contributions of disk or network to the overall run-time are small. We expand on their work, taking into consideration deployment scenarios where layers are not co-located. We also take into account a wider set of workloads, leading to different considerations on the analytic job run-time bounds.

Venzano et al.[13] explore how virtual machine placement can have an impact on network throughput. They use a metric called Bulk Transfer Capacity (BTC) to calculate the maximum throughput of a network path; with this metric they find that virtual machine placement plays a crucial role in determining application-level performance, more in particular they evidence the dramatic throughput difference between co-location on the same physical host and on different physical hosts. We take a much more general approach, testing different configuration of virtual machine placements, running analytic applications and using different metrics.

3.2 Methodology

In the following section we describe the methodology used in this study to analyze the performance of analytic applications in different configurations of compute and data layers. This section is divided into four part: the platform, the scenarios, the workloads and the metrics.

3.2.1 Experimental Platform

To run the measurements, the OpenStack private cloud platform hosted at Eurecom has been used. At the time of this study, the platform is composed of twelve hosts, belonging to two different generations.
The difference between the two lies only in the CPU model. The older generation has Intel Xeon E5-2650L at 1.80GHz CPUs, the other one has Intel Xeon E5-2630 at 2.40GHz CPUs. Both sets have hyper threading enabled for a total of 32 cores (16 physical and 16 virtual), 128 GB of RAM, ten 7200 RPM disks of 1 TB each and 1GB/s Ethernet interfaces. Having a cluster composed of an heterogeneous set of machines is quite common and reflects a realistic scenario in private and public clouds.

Data layer OpenStack is installed on all machines and is used to deploy virtual clusters. We use the following storage solutions: Swift, HDFS over Volumes and classic HDFS on local disks.

Swift[14] is built using the so called SAIO (Swift All in One) solution; all processes are run on the same host. The machine that we use for Swift has: one socket with Intel Xeon L5320 at 1.86GHz with hyper threading active for a total of 8 Cores (4 physical and 4 virtual), 16 GB of RAM, five 7200 RPM disks of 1 TB each (one for the OS and four for Swift) and a network link of 1 Gb/s. Swift configuration contains 1 proxy server and 4 storage nodes.

OpenStack is configure to provision volumes from a distributed CEPH[15] cluster. CEPH is distributed over four disks spread across two of the twelve hosts that compose our platform. CEPH already performs three-way replication, so we disabled HDFS replication when setting up HDFS over CEPH volumes.

The last storage solution we used, HDFS, is built over a virtual cluster of five machines (one Namenode and four Datanodes) across four physical hosts; all Datanodes are on different hosts. The virtual machines that compose the cluster have: 2 cores, 4 GB of RAM and 80 GB of disk.

Compute layer The compute layer is composed of five virtual machines and uses Spark 1.3.1 [16] as scalable computing framework; four worker nodes, spread across different hosts, and one master. The application containers have: 4 cores, 8 GB of RAM and 80 GB of disk.

Having the components that do the actual work, HDFS datanodes and Spark workers on different virtual machines allows for better analysis of the metrics gathered, instead the Spark master and HDFS namenode can share the same physical host because the architecture of both system exclude them as sources of bottlenecks.

We decided not to use cluster providers such as Amazon [7], to improve our analysis on the results by having full control and knowledge of the virtualization platform on which the virtual machines run.

All the experiments were repeated five times and there was no interference caused by other users running some other applications.

3.2.2 Deployment scenarios

We define six different compute scenarios:

1. Guest Coloc: Compute and HDFS layers are hosted on the same virtual machine. This is the traditional configuration that maintains data locality.

2. Host Coloc: Compute and HDFS layers are hosted on the same physical host, but on different virtual machines.

3. Host Coloc V: Compute and HDFS layers are hosted on the same physical host, but on different virtual machines. The HDFS layer is on top of volumes.

4. No Coloc: Compute and HDFS layers are hosted on different physical hosts.

5. No Coloc V: Compute and HDFS layers are hosted on different physical hosts. The HDFS layer is on top of volumes.

6. Swift: in this scenario we use Swift as the data layer instead of HDFS.

Putting HDFS on top of volumes, while counter-intuitive, is actually a configuration proposed by Amazon, where EBS volumes are proposed as a low-cost long-term storage solution for virtual machines.

The scenarios we propose all cover valid use-cases and each scenario has different Compute-to-Data paths, as defined in section 3.

In the Guest Coloc scenario the Compute-to-Data path is internal to the application container that hosts both the Compute layer and the Data layer. In this scenario, HDFS has been configured to use Unix Sockets to send data from the Data to the Compute layer reducing the overhead caused by extra communication required when using the TCP/IP protocol. Thus, the Compute-to-Data path is just composed by the Unix Socket when reading data, the loopback interface when writing to HDFS, and the physical network link when HDFS replicates data across its nodes. This scenario should have the lowest run time and we expect to see network activity only during application phases that write on HDFS.

In the Host Coloc scenario the Compute-to-Data path is composed by a virtual network link when reading data and by a virtual and a physical network link when writing data because of the HDFS replication mechanism; they both use TCP/IP as communication protocol. In this scenario we expect: higher network utilization and a slower run time both caused by the longer Compute-to-Data path. Like for Guest Coloc, Data and Compute layer are deployed on the slowest area of the platform.

The No Coloc scenario has compute and data instances on different hosts, thus the Compute-to-Data path is composed, for both reads and writes, by a physical network link, with the TCP/IP protocol. Our expectations for this scenario are similar to the Host Coloc, with the difference that the run time should be slightly slower because the communication, in all cases, are over a physical link and this can be a source of increased latency.

Volumes So far, we discussed scenarios that had, as Data Layer, HDFS over a filesystem hosted on a block device local to the virtual machine. The following two scenarios use volumes instead. Volumes are provisioned with the help of OpenStack Cinder and are hosted on CEPH.

Both Host Coloc V and No Coloc V have the same initial Compute-to-Data path of the respective scenarios without the volumes, but, the Data layer itself has to retrieve data from a remote location, so it has to use a physical network link and the TCP/IP protocol. Thus, we have a longer and more complex Compute-to-Data path. We expect to see an increased disk utilization and the same network utilization as the respective scenarios without the volumes and an increased run time caused by the longer Compute-to-Data path.

Swift Swift is very similar to the No Coloc scenario with the difference that we use Openstack Swift as data layer instead of HDFS. In this scenario our expectation is that of a longer run time caused by the architecture of Swift and a resource utilization similar to the No Coloc scenario.

The scenarios that we address can be the foundation of more complex scenarios. An example is an application composed of several jobs, each job performs some transformation on the input and creates a model, then use the model to predict behaviors; the model and other intermediate data are saved on an HDFS layer that is co-located with the compute layer and the final results on Swift.

3.2.3 Workloads

We used three different workloads, all of them written in Scala and run on Spark as the scalable computing framework:

WordCount : Hello World application for the parallel computing. Very simple application that counts the words of a text file. The text file used is composed by the concatenation of books of the Project Gutenberg [17], limited to 20 GB. This application is read intensive and is composed by 158 map task and 158 reduce task. The output size is 225.6 MB, 676.8 MB when counting the replication factor.

DFSIO-3 : Write intensive application. It is composed by just one map stage that writes a sequence of zeros to the storage. We have map tasks that write a small portion of the file because we are limited by the amount of memory that Spark workers allow the user to use and we did not want to modify that limit to prevent a degradation of Spark's execution. This results in having 1280 map tasks that write 10 MB each one for a total of 12.5 GB, 37.5 GB when counting replications.

TPC-DS : Transaction Processing Performance Council's decision-support benchmark test [12]. It is designed to model multiple users running a set of different decision-support SQL queries including OLAP, interactive, reporting and data mining. The benchmark uses pre-generated data that mimic the database of a retail product supplier about product purchases. We set the input size to 20 GB, using the default configuration of storing the data in Parquet[18], disabling Spark caching system. We use a set of 5 queries made available in Spark-Sql-Perf library[19] by DataBricks. The output size is 17.9 KB, 53.8 K when counting replications and contains just the benchmark results.

Clustering : This application runs an algorithm developed by Lulli et al. [20]. Their algorithm works as text clustering and had 2 phases. In the first phase, it builds an approximate k-NN graph of text items. The first phase concludes with a pruning stage, which strives at eliminating spurious links between items with low similarity. In the second phase, they use a parallel approach to identify connected components in the k-NN graph, which are a proxy for clusters of similar items. This workload exploits the cache mechanism implemented in Spark. We chose this workload for two reason: first because it represent a real workload used by data scientists and second because it will allow us to check the performance on the various scenarios when we have a small input size and Spark's cache layer active. The output size is 285.6 MB, 856.9 MB with replications.

3.2.4 Performance metrics

To compare results between scenarios and workloads, we selected the following metrics:

Run time It is the amount of time required by the application to finish its activity. The run time is an high-level metric that gives and indication of global performance and makes comparing the same workload across different scenarios very easy. On the other hand it does not give any indication on the resource usage and needs to be complemented by other metrics.

CPU utilization Utilization of the CPU across all virtual cores available, calculated according to the formula defined in equation 5, measured at one second intervals.

Network utilization Utilization of the network interface, calculated according to the formula defined in equation 9, measured at one second intervals.

Disk utilization Utilization of the disk subsystem, calculated according to the formula defined in equation 15, measured at one second intervals.

Taking samples at one second of granularity allow us to have a detailed time series of the metrics during the course of the application run, to better understand when the demand of resources is higher; we monitor each component and machine of the cluster.

3.3 Results

Host Coloc	218.96 ± 3.57
No Coloc	221.08 ± 3.31
Guest Coloc	222.33 ± 2.50
Host Coloc V	231.65 ± 17.52
No Coloc V	284.61 ± 14.13
Swift	803.88 ± 51.99

(a) WordCount

Guest Coloc	180.87 ± 3.21
No Coloc	186.28 ± 4.07
Host Coloc	199.54 ± 4.13
No Coloc V	393.39 ± 67.34
Host Coloc V	413.04 ± 50.74
Swift	2348.06 ± 233.48

(b) DFSIO-3

Guest Coloc	2376.33 ± 24.39
Host Coloc V	3916.68 ± 135.43
No Coloc V	4218.30 ± 129.09
No Coloc	4999.28 ± 12.76
Host Coloc	5939.94 ± 9.79
Swift	41181.35 ± 1773.68

(c) TPC-DS

Host Coloc V	288.40 ± 12.45
Guest Coloc	406.39 ± 9.74
No Coloc	408.57 ± 12.63
Host Coloc	419.88 ± 8.45
No Coloc V	433.15 ± 6.87
Swift	1675.92 ± 71.62

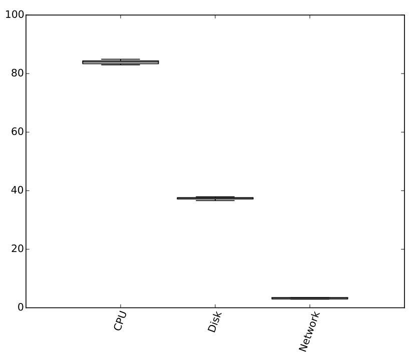
(d) Clustering

Table 3.3a: Run times for all scenarios and workloads. Measurements are expressed in seconds.

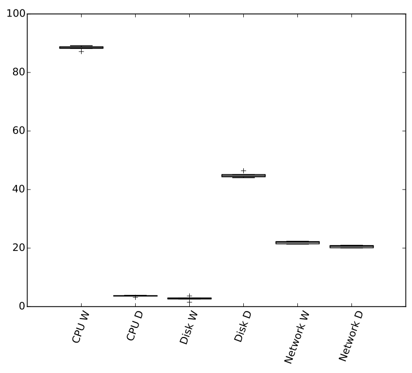
In this section, we discuss the results from the workloads obtained using the metrics described in section 3.2 and 2.1.1.

3.4 Run times

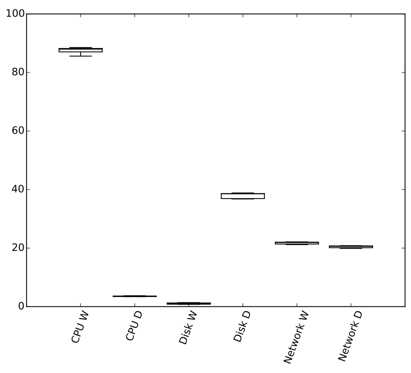
WordCount Table 3.3a shows us that the fastest scenarios is not Guest Coloc, as could be expected, but Host Coloc, also the run times of Guest Coloc, Host Coloc and No Coloc are almost identical. This result goes against our intuition; scenarios that contain network links inside their Compute-to-Data path, and thus an overhead caused by both the communication protocol and network link itself, are faster than scenarios without network links.
Regarding the other scenarios' run times we are in line with the assumptions mentioned before. More in particular, we find that Host Coloc V and No Coloc V have an overhead over the run time of the



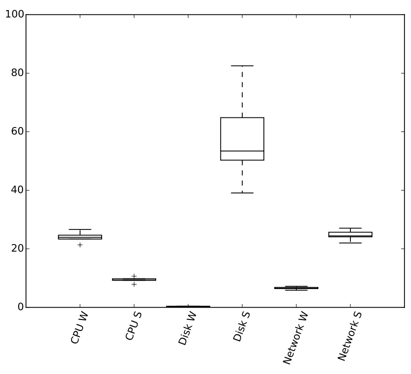
(a) Guest Coloc



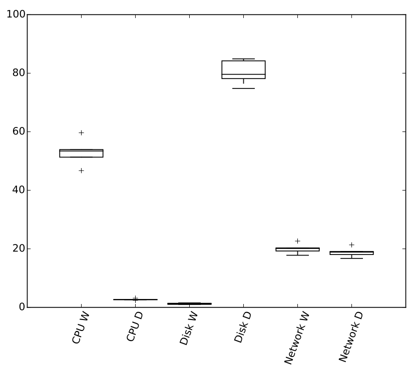
(b) Host Coloc



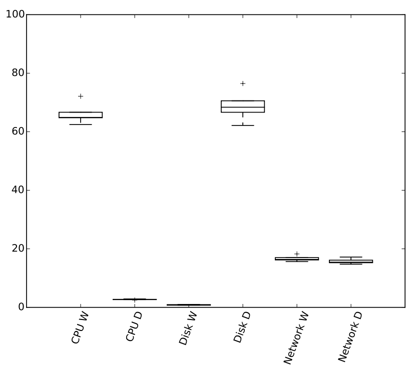
(c) No Coloc



(d) Swift



(e) Host Coloc V



(f) No Coloc V

Figure 3: Resource utilization for the WordCount workload in all scenarios. The labels on the X axis that end with “W" stand for “Worker" and are the instances of the compute layer, labels that end with “D" stand for “DataNode" and are the instances of the data layer, finally labels that end with “S" stand for Swift. The resource utilization reported in this figure is averaged across all instances.

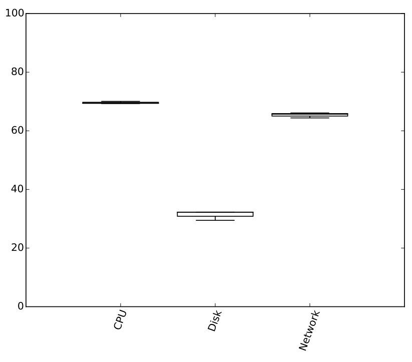
fastest scenario of 16% and 42% respectively[3]. Swift seems to be the slowest scenario, with run times more then three times slower than Guest Coloc.

DFSIO-3 This workload was conceived to stress the data layer. Results show us that the fastest scenario is Guest Coloc, followed by, in order, No Coloc, Host Coloc, No Coloc V, Host Coloc V and, finally, Swift. In this scenario, we are in line with our initial expectations regarding the Guest Coloc being the fastest. Results also show that scenarios with volumes are much slower compared to the ones that use a local disk; we have a 117% and 128% overhead compared to the Guest Coloc scenario. Swift is more than eleven times slower in this case compared to the fastest scenario.

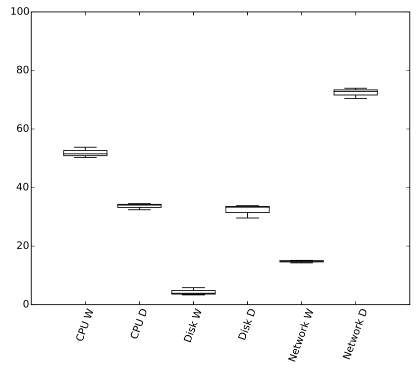
TPC-DS By looking at table 3.3a we can see that Guest Coloc is the fastest scenario followed by Host Coloc V, No Coloc V, No Coloc, Host Coloc and Swift. The results of this workload are not in line with both our initial considerations and the previous workloads; the only scenario that respects them is the Guest Coloc. Volumes scenarios, that so far have been the slowest (excluding Swift), now are in second and third place. Host Coloc, that should be ranked as second fastest scenario, is before the last. The overheads, compared to the fastest scenario, are the followings: 64% for Host Coloc V, 77% for No Coloc V, 110% for No Coloc and 150% for Host Coloc. Swift is eighteen times slower than Guest Coloc.

Clustering Here the results show a consistency in our preliminary analysis of the expected performances of the different configurations. The only outlier is Host Coloc V, we can assume that the

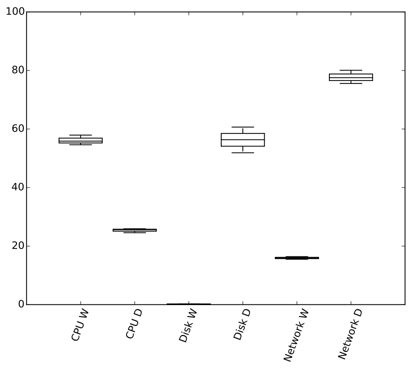
[3]Since run time variations are so small between scenarios, we run the same test on a bigger data set (100 GB), obtaining the same percentages.



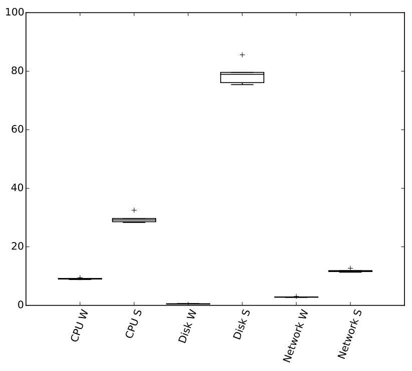
(a) Guest Coloc



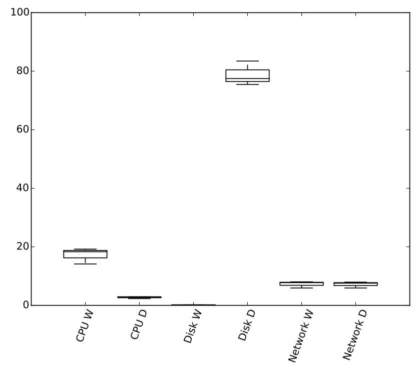
(b) Host Coloc



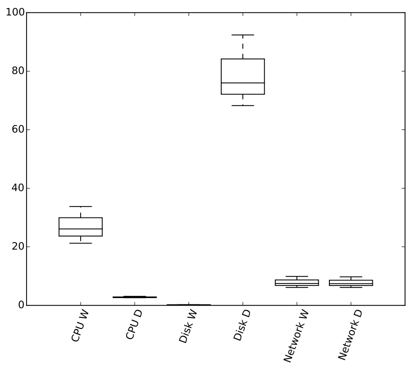
(c) No Coloc



(d) Swift



(e) Host Coloc V



(f) No Coloc V

Figure 4: Resource utilization for the DFSIO-3 workload in all scenarios. The labels on the X axis that end with "W" stand for "Worker" and are the instances of the compute layer labels that end with "D" stand for "DataNode" and are the instances of the data layer, finally labels that end with "S" stand for Swift. The resources utilization reported in this figure is averaged across all instances.

workload is heavily using the CPU.

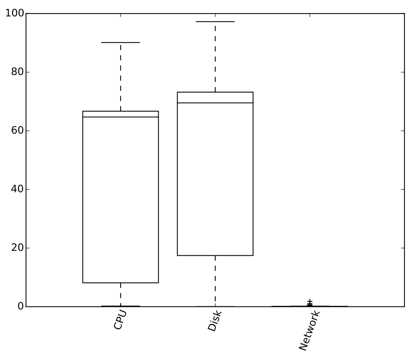
By looking at run times we can say that the Guest Coloc scenario always performs the same or better, we will add more to this observation by analyzing the resource utilization in section 3.5.

Swift considerations Swift consistently has the worst performance of all tested scenarios. We dug in Spark logs to find out why Swift was performing so poorly and we found that, especially for DFSIO-3, there is an increased time in the operations that Spark perform after the completion of the job; merging temporary files, copying them to a different folder and deleting the temporary folder. This "post-job" process is done by the Spark Master and is taking a lot of time because of a lack of optimization in Swift. Renaming objects in Swift requires rewriting the data with a different name and deleting the old object, instead of just changing the object directory. Due to Spark architecture this final work of temporary file management is done sequentially by a single node, the Spark Master, further slowing down the process.

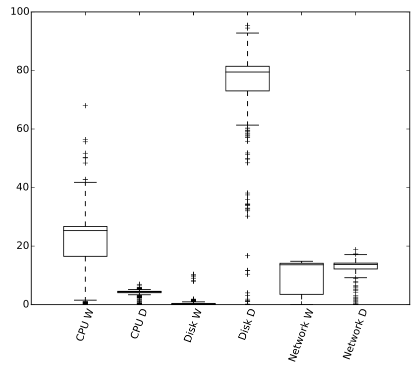
If we not consider this "post-job" process, DFSIO-3 takes 840 seconds, instead of 2230 seconds: the "post-job" part takes 140% of the job run time. On the other hand, even if we do not consider the "post-job" process, DFSIO-3 takes 464% more compared to Guest Coloc therefore something else is happening that we cannot see by just looking at run times.

3.5 Resource utilization

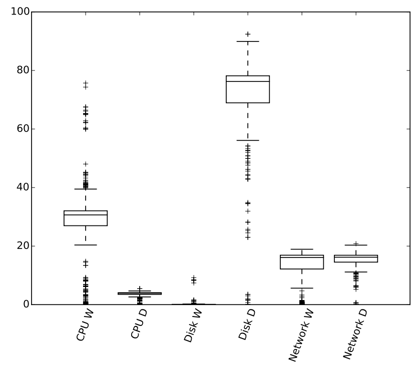
Wordcount Figure 3 shows the resource utilization for WordCount in all the scenarios. The first consideration that we make is that in all scenarios, the resource utilization is steady during the whole workload duration. By comparing Guest Coloc, Host Coloc and No Coloc, we see that the resource



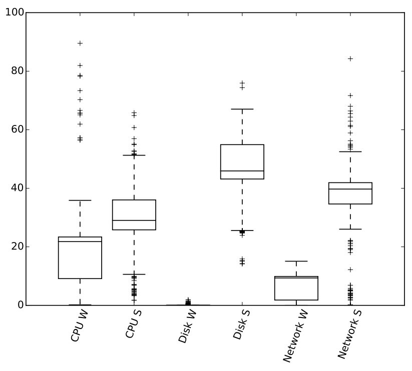
(a) Guest Coloc



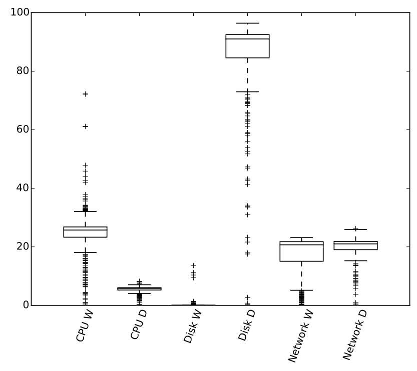
(b) Host Coloc



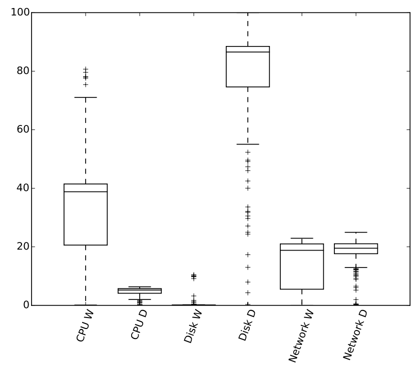
(c) No Coloc



(d) Swift



(e) Host Coloc V



(f) No Coloc V

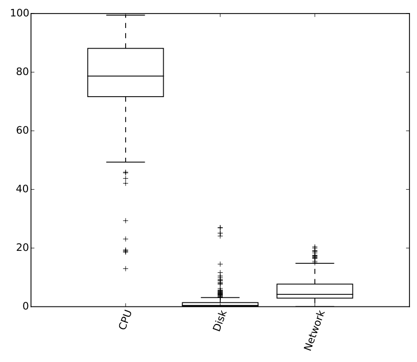
Figure 5: Resource utilization for the TPC-DS workload in all scenarios. The labels on the X axis that end with "W" stand for "Worker" and are the instances of the compute layer labels that end with "D" stand for "DataNode" and are the instances of the data layer, finally labels that end with "S" stand for Swift. The resources utilization reported in this figure is averaged across all instances.

utilization is pretty much the same between Guest Coloc and No Coloc but is higher on Host Coloc and, thus, this is the reason why Host Coloc is faster[4]. Host Coloc is able to exploit the CPU and disk better than Guest Coloc because there is no resource contention between compute and data layers since they are on different instances.
Guest Coloc and No Coloc have pretty much the same resource utilization because, even if No Coloc has the same Network usage and is able to exploit CPU and Disk better, like in the Host Coloc, there is an higher latency in the Compute-to-Data caused by the physical network link; this higher latency forces the CPU to wait a bit longer to process the data requested and thus reduce the usage[5]. In the case of Host Coloc V and No Coloc V we have a higher Disk and lower CPU utilization compared to the previous scenario. CPU is lower because data takes more time to reach the Compute layer due to the longer and more complex Compute-to-Data path. On the other hand, Disk is higher not because of better efficiency, but because of increased access times caused by the network volumes.

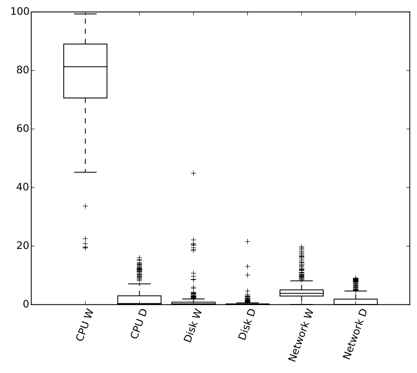
DFSIO-3 In the Guest Coloc we can see that, similarly to WordCount, the most used resource is the CPU, followed by Disk and Network; compared with WordCount we have less usage on the CPU and slightly more on the Network. Network is used more, compared to WordCount, because the "Short-Circuit" option in HDFS is not active during writes and the TCP/IP protocol over the loop-back interface is used; Network is also used more because HDFS has to replicate more data compared to WordCount and thus it has to transmit more data to the other DataNodes.

[4]An higher utilization of a resource not always means a bottleneck. It can be considered a bottleneck when it is always at 100% and this is not our case.

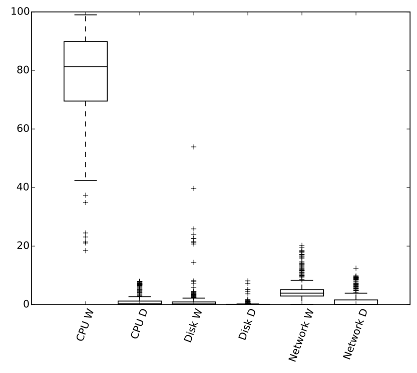
[5]All requests from the compute layer are synchronous



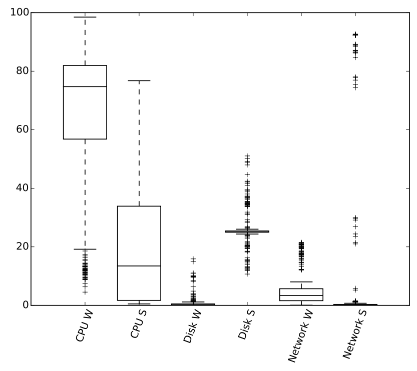
(a) Guest Coloc



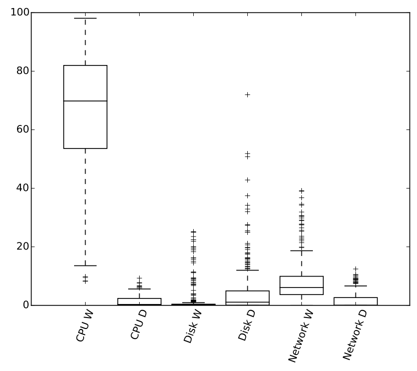
(b) Host Coloc



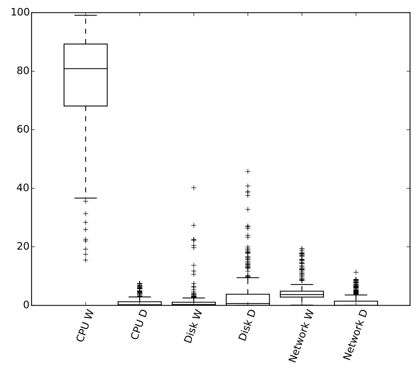
(c) No Coloc



(d) Swift



(e) Host Coloc V



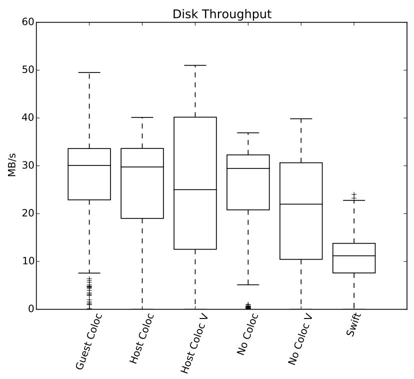
(f) No Coloc V

Figure 6: Resource utilization for the Clustering workload in all scenarios. The labels on the X axis that end with "W" stand for "Worker" and are the instances of the compute layer labels that end with "D" stand for "DataNode" and are the instances of the data layer, finally labels that end with "S" stand for Swift. The resources utilization reported in this figure is averaged across all instances.

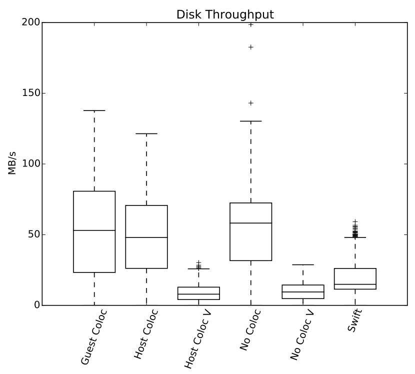
Since the Guest Coloc resource utilization is what we expected, we are now going to compare resources between Host Coloc and No Coloc. By observing the CPU of Spark's Worker, we can see that No Coloc was able to use it slightly better because the Disk usage was higher thus allowing Spark's Worker to process the next write faster. The disk of Host Coloc was a bit more busy because, in our platform, the Compute layer and Data layer were using the same physical disk. Like in WordCount, we can see similar variations in resource utilization when using Host Coloc V and No Coloc V compared to the scenarios without volumes; CPU utilization is half, like in WordCount, and Disk utilization is doubled when using No Coloc V and more than tripled when using Host Coloc V because writing is a slower process than reading and thus has a bigger impact.

If the overhead of using volumes in the DFSIO-3 is similar to the one in WordCount, why the overhead in the run times is so different[6]? We found the answer to this question by looking at the number of task for each workload. WordCount has 158 mapper tasks and 158 reducer tasks for a total of 316 tasks, but we are going to consider just the mappers because the total reducer run time is much smaller compared to the mappers. On the other hand, DFSIO-3 has 1280 mappers that write data. On all scenarios the number of task that can be executed concurrently is 16, this means that in WordCount and DFSIO-3 we have 9 and 80 waves respectively. Assuming there are no stragglers we can calculate the average run time of a single wave in the fastest scenario and we find 22 second for WordCount and 2,26 second for DFSIO-3. Shorter tasks are more sensitive to latency, therefore we can say that the overhead of using volumes in DFSIO-3 is higher because tasks are faster and more

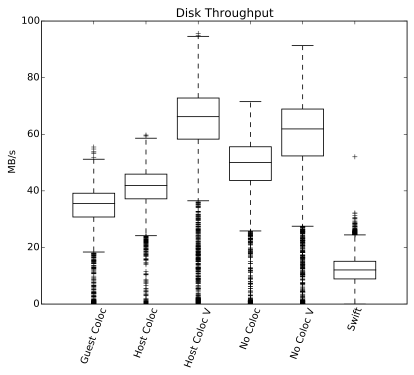
[6]16-42% in WordCount and 117-128% in DFSIO-3



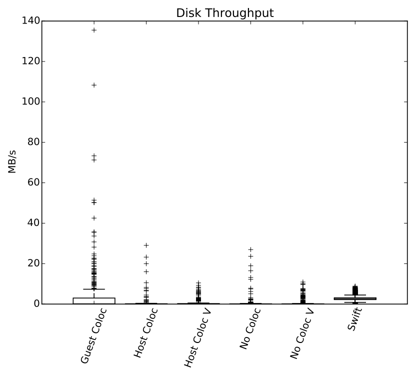
(a) WordCount



(b) DFSIO-3



(c) TPC-DS



(d) Clustering

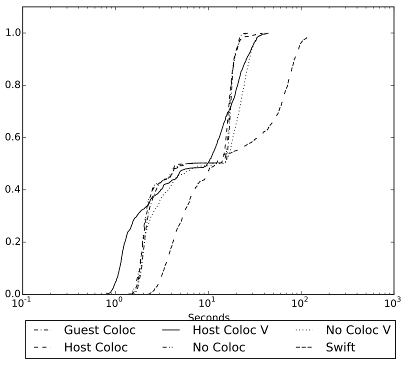
Figure 7: Comnparison of disk throughput across workloads and scenarios. When the compute and data layer are separate, the disk throughput considered here refers to the data layer only.

sensible to latency caused by the network and volumes infrastructure.

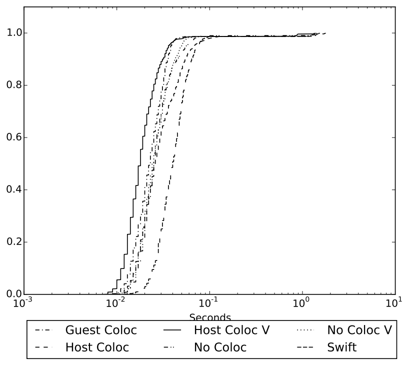
TPC-DS From figure 5 in Guest Coloc the disk has an higher usage compared to the CPU, as opposed to the previous two workloads; this result is in contrast with work from Kay et al.[11] that state that the most used resource was the CPU and that disk could not be a source of bottlenecks. The reason why our results differ from Kay et al. is that: *(i)* the query set is different, they run 20 queries, we run 5 and *(ii)* we use 4 Spark's Workers and they use 20. The different size of the cluster can also explain the higher disk utilization; distributing the data across more nodes increases the parallelism leading a single task to read less data from disk. In case this assumption holds this would mean that increasing the number of instances reduces the benefit of data locality, in favor of other configurations, like the one that we tested. In both Host Coloc and No Coloc we see that the most used resource is the disk and that CPU is relatively low compared to Guest Coloc, as for the other workload we can say that the CPU is lower because of the delay introduced by the relatively longer Compute-to-Data path.

When we compare Host Coloc and No Coloc with the two scenarios that use volumes, we see that the overhead is completely different from the previous two workloads; CPU is the same or higher, in case of No Coloc V and run time is lower. How is that possible if so far we said that an higher latency means lower CPU usage and if volumes so far were slower compared to scenarios with HDFS? The answer is that the infrastructure of the volumes contains a cache level, thus, since the Compute-to-Data path is entirely made by network links (if we consider an access to the cache instead of the physical disk), the bytes transferred are higher than the case without volumes. Figure 7 shows that this is indeed the case; the two scenarios that use volumes have higher disk throughput.

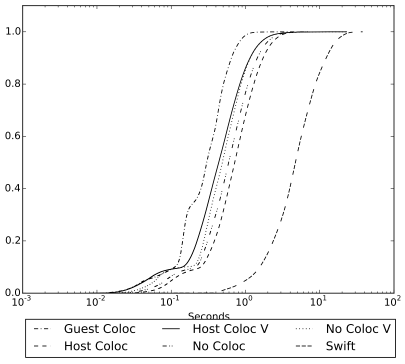
Like in the DFSIO-3 case, we can attribute the difference between run times of the various scenar-



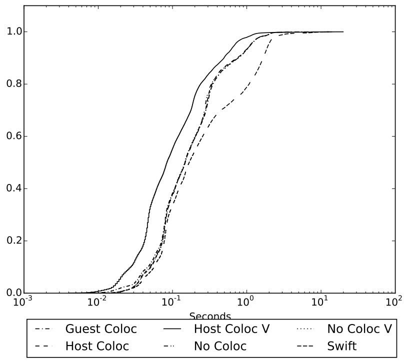
(a) WordCount



(b) DFSIO-3



(c) TPC-DS



(d) Clustering

Figure 8: Distribution (CDF) of task run times for all workloads and scenarios. Tasks are the smallest unit of work generated by Spark and distributed across the compute layer for computation.

ios to the task run time. Workload TPC-DS is executing 5 queries for a total of 103 jobs. The queries spawn 4, 13, 25, 28, and 31 jobs respectively for a total of 101 jobs. The two missing jobs are: *(i)* an initial job launched by Spark to plan the execution of the queries and *(ii)* a final job to collect some metrics. Considering the number of jobs, stages and task, we calculate that the number of waves is 6584 and we find that, in the fastest scenario, a single wave takes 360ms to finish, thus, for the same considerations made during the analysis of DFSIO-3, TPC-DS is more susceptible to latency. A 1ms latency increases the run time by almost 7s in TPC-DS, 128ms in DFSIO-3 and 9ms in WordCount[7]. From figure 8 we can see the difference in run times across different scenarios when using a specific workload. If we take a closer look at the TPC-DS workload, we can see that, on the slowest scenario (not considering Swift), the tasks last twice as long as the task on the fastest scenario: we can attribute this behavior to the longer Compute-to-Data path, more in particular a latency is added and since all our data requests are synchronous, we delay the transformations and actions performed by our analytic framework.

3.5.1 General remarks

After the previous analysis we feel confident to answer to our initial question: what happens when we break data locality? There is a drop in the performance. This drop is caused by longer Compute-to-Data path that adds latency. On the other hand we saw that, depending on the workload, using a configuration that guarantees data persistence, the difference can be negligible and thus a data scientist could consider breaking the data locality principle.

We also would like to point out that the resource utilization of a workload can change dramatically by changing the scenario on which it is run. We extended the work done by Kay et al. [11],

[7]This calculation are not 100% accurate because we haven't take into account stragglers

studying the behavior of TPC-DS on more scenarios, with different input size and query set, resulting in different resources utilization patterns.

Another interesting point that we saw is that, when using analytic applications that re-iterate over the same data, using the cache at data layer leads to better results. This line of research has already been approached by other systems like Tachion[21] and we confirm that indeed it helps. First a cache at data layer level enables multiple applications to exploit hot data [8] and, second, cluster managers can exploit this to decrease the average run times of applications even when a data scientist does not enable a cache mechanism in their analytic framework, or for those frameworks that do not support it, resulting in an overall better resources utilization.

We also observed how object storage behave poorly compared to block storage, in our case Swift vs HDFS. We believe that, on top of a non optimized implementation of the connector[22] between Hadoop[23] and Swift, the architecture introduces too much overhead. This overhead is caused by the higher level protocol used to communicate (HTTP) and the introduction of a proxy server that act as coordinator for the request, changing the architecture of the storage solution from distributed to centralized. Using a proxy server as coordinator enables cluster managers to easily add control flow to the object storage. Recent works from IBM [24] shows that some control flow and data transformation can be added closer to the storage nodes. IBM work allows to rethink Swift's architecture making it more distributed reducing the possibility that the proxy server could be the cause of bottlenecks. While running tests on Swift we verified that the proxy component has no scheduling system for the incoming requests, just admission control.

3.6 Summary

The configurations that we study are currently exploited by different data analysts and by the IOStack project. Starting from a configuration that was considered standard and best performing, with no separation between data and compute layers, we increasingly move toward more complex solutions that provide more and more flexibility. We span six configurations using three different data layer: HDFS, HDFS on top of volumes and Object Storage (OpenStack Swift). With the help of 4 different workload, both real and benchmark, we find the following:

1. Data locality plays an important role in the performance of an analytic application. In general it is not true that data locality is becoming more and more irrelevant, different workloads have different resource usage patterns and are more or less sensitive to data layer access throughput and latency.

2. Caching hot data, at data layer level, is helpful. Spark contains a cache layer that it is used when requested by the data analyst and that it is not shared across applications. Adding a caching system to the data reduces the run time.

3. Object storage should be considered carefully before being used for analytic applications. Moreover, in this study, we found bugs and design choices made in Swift that impact the run time of analytic applications.

4 Swift Monitoring for Bandwidth differentiation

Related to the Bandwidth differentiation service (see deliverable 2.2), we need to know to which object server we should send the request to, to maintain a correct bandwidth. There are some problems for that:

1. A mechanical device does not scale correctly when we increase the number of clients. The bandwidth obtained depends of the workload (random, sequential) and the different interferences. As a result, we cannot use a maximum bandwidth value.

2. SSD devices are more scalable, as they do not have seek time.

[8]now Spark only support cache at intra-application level

For that we need to keep track continuously of local maximums and minimums (1, 2 and 3 minutes) and use them to assess if the new request will fit the disk bandwidth or not. As we have explained, the bandwidth obtained depends on the workload and interferences; therefore, the bandwidth differentiation service does not guarantee that the bandwidth will be obtained, but guarantees that the relations between different workloads are maintained.

To monitor such behavior we created a swift middleware called bwinfo [25](at the object server) to inform the SDS controller (Figure 9, based on a NSDI'15 paper [26]) about different values such as:

1. Immediate BW of the different devices
2. Running mean, minimum and maximum of the BW of the last 1, 2, and 3 minutes

For example:

```
{'/dev/sda': {'mean2min': xxx, 'mean1min': xxx, 'insta': 0.0,
'mean5min': xxx,'max2min': xxx, 'max1min': xxx, 'max5min': xxx,
'min2min': xxx, 'min1min': xxx, 'min5min': xxx}}
}
```

The SDS controller also receives the request being made at each object server to detect overloaded servers, this information, osinfo, contains the following values:

1. Object server id
2. Disk
3. Thread serving the request
4. Policy assigned
5. Account
6. Needed BW per account
7. Obtained BW per object request, range and Object ID.

For example:

```
{"127.0.0.1:6010":
{"sdb1":
{"0":
{"policy": "gold", "account": "AUTH_8d1e0ed1be62",
"identifier": "object", "needed_BW": -1,
"objects":
[{"range": "0 - end", "oid": "/AUTH_8d1e0ed1be62/files/file3.dat",
"oid_calculated_BW": 7.56},
{"range": "0 - end", "oid": "/AUTH_8d1e0ed1be62/files/file.dat",
"oid_calculated_BW": 9.55},
{"range": "0 - end", "oid": "/AUTH_8d1e0ed1be62/files/file2.dat",
"oid_calculated_BW": 6.50}]
}}}
}
```

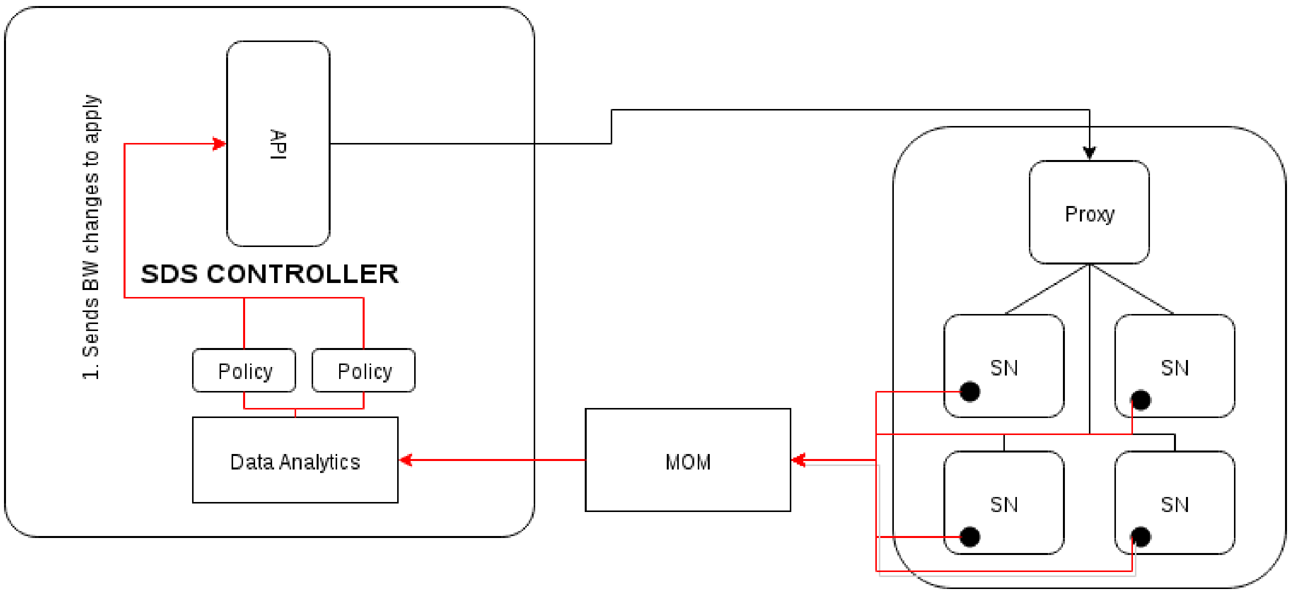


Figure 9: Swift and SDS internal monitoring.

5 I/O detailed analysis

To understand the implications of the middleware building blocks and behaviour into the I/O Stack, we used blktrace [27] that creates a log file of the requests going to the disk that is included in the Linux Kernel. Converting the log file using blktrace2paraver [28] and the paraver [29] tool we can visualize the I/O behaviour on a single node to find problems. Such analysis showed problems in the threading model of Swift and the relation with the disk scheduler on mechanical devices.

An example of such analysis can be seen on Figure 10, where we have the different processes of the system on y-axis (including devices), and the timeline on x-axis. On this example, each color represent the number of operations (0, light blue, 2 white) issued to the disk, and the lines represent what thread is sending the request. We can observe how even having 1 single client, the sequential requests are going to the disk from two different threads.

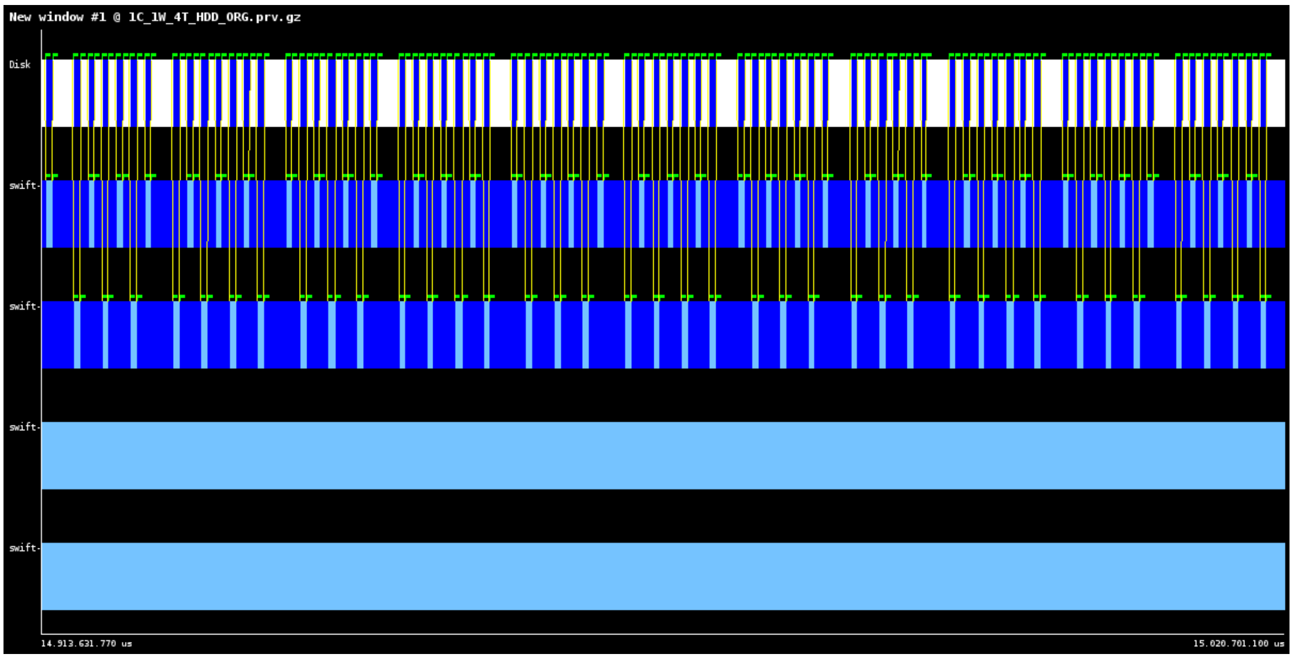


Figure 10: Paraver analysis for one client with a Swift configuration of 4 Threads and 1 Worker.

6 Conclusion

This document presented the work done in WP 5: IOStack's monitoring system architecture and a study conducted to pinpoint possible bottleneck in different application containers deployment strategies.

The monitoring system is composed by two sets of tools. The first is composed by open source tools that are already well known by the community and the second set is made of two tools for resources monitoring developed internally. They allow us to have two types of monitoring systems with different granularity and control. The open source tools monitor generic resources over the entire platform. The IOStack-specific tools are used to extract specific and raw information across a subset of virtual machines in order to have a better insight on the possible bottlenecks that can afflict a specific deployment configuration.

In the measurement study, we analyzed different workloads and scenarios that lead to different Compute-to-Data paths. We found the following: *(i)* a correct placement of Compute and Data layer leads to lower analytic application run times, *(ii)* caching hot data, at data layer level, is helpful and *(iii)* object storage should be considered carefully before being used for analytic applications.

Finally, we presented the monitoring tools developed specifically to address the bandwidth differentiation problem in Swift, as part of the overall architecture detailed in deliverable 2.2.

References

[1] CollectD, "Collectd." https://collectd.org/.

[2] Graphite, "Graphite project." https://github.com/graphite-project/.

[3] Grafana, "Grafana." http://grafana.org/.

[4] J. Dixon, Monitoring with Graphite. O'Reilly Media, 2016.

[5] InfluxData, "Influxdb." https://influxdata.com/time-series-platform/influxdb/.

[6] iostat, "iostat." http://linux.die.net/man/1/iostat.

[7] Amazon, "Amazon web services." http://aws.amazon.com/.

[8] DataBricks, "Databricks cloud." https://databricks.com/product/databricks-cloud.

[9] Cloudera, "Cloudera cloud." http://www.cloudera.com/content/cloudera/en/solutions/partner/Amazon-Web-Services.html.

[10] Google, "Google cloud hadoop." https://cloud.google.com/hadoop/.

[11] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pp. 293–307, USENIX Association, May 2015.

[12] TPC, "Tpc-ds." http://www.tpc.org/tpcds/.

[13] D. Venzano and P. Michiardi, "A measurement study of data-intensive network traffic patterns in a private cloud," in DCC 2013, Workshop on Distributed Cloud Computing, co-located with the IEEE/ACM Conference on Utility and Cloud Computing (UCC), 9 December 2013, Dresden, Germany, (Dresden, GERMANY), 12 2013.

[14] OpenStack Foundation, "Swift." http://docs.openstack.org/developer/swift/.

[15] R. Hat, "Ceph." http://ceph.com/.

[16] Apache, "Spark." http://spark.apache.org/.

[17] G. Project, "Gutenberg." https://www.gutenberg.org/.

[18] Apache, "Parquet." https://parquet.apache.org/.

[19] DataBricks, "Spark-sql-perf." https://github.com/databricks/spark-sql-perf.

[20] A. Lulli, T. Debatty, M. Dell?Amico, P. Michiardi, and L. Ricci, "Scalable k-NN based text clustering," in BIGDATA 2015, IEEE International Conference on Big Data, October 29-November 1, 2015, Santa Clara, USA, (Santa Clara, ÉTATS-UNIS), 10 2015.

[21] t. . Takyon Project.

[22] OpenStack Foundation, "Hadoop-Swift Connector." https://github.com/openstack/sahara-extra/tree/master/hadoop-swiftfs.

[23] Apache, "Hadoop." http://hadoop.apache.org/.

[24] S. Rabinovici-Cohen, E. Henis, J. Marberg, and K. Nagin, "Storlet engine: performing computations in cloud storage," tech. rep., IBM Technical Report H-0320 (August 2014), 2014.

[25] R. Nou, "Swift bandwidth differentiation (hard-limiter branch)." `https://github.com/iostackproject/IO-Bandwidth-Differentiation-Client.git`.

[26] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems,"

[27] J. Axboe and A. D. Brunelle, "Blktrace User Guide," 2007.

[28] R. Nou, "blktrace to Paraver trace converter." `https://github.com/mavy/blktrace-utils`.

[29] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," WoTUG-18, pp. 17–31, 1995.