Performance Analysis of a Cluster Management System with Stress Cases

Gergő Gombos, Attila Kiss, Zoltán Zvara

Eötvös Loránd University, Faculty of Informatics
Pázmány Péter u. 1/c, H-1117 Budapest, Hungary
ggombos@inf.elte.hu, kiss@inf.elte.hu, dyin@inf.elte.hu

Abstract: Cluster computing frameworks are important in the "Big Data" world. The famous common framework is the MapReduce that was introduced by Google. This framework is used by many of companies. However, this technique doesn't effectively solve all analytical problems. Some cases need another framework and these frameworks can work in the cluster. In this case, the cluster needs a manager that manages the framework. Therefore, the performance analysis of cluster management systems will be important. In this paper, we compare the performance of two most well-known cluster management systems (Yarn, Mesos) with stress cases. We analyze the resource usage techniques of the management systems.

Keywords: cluster management; resource sharing; scheduling

1 Introduction

For years, Big Data was confined to a group of elite technicians working for companies like Google and Yahoo, but the databases and the tools used to manage the data at that scale have been constantly evolving. At that time, Big Data was only a synonym to the leading tool, the Apache Hadoop [1], a MapReduce [2] implementation that was used as a data-processing platform for many years, exclusively. As Big Data continued to evolve, researchers found that MapReduce – though is still powerful for a large number of applications – was not as effective at solving many problems. Technicians were working on new cluster computing frameworks, and it became clear that no framework would be optimal for all applications. Researchers have been developing a wide array of data-centric computing frameworks and the need for a major computing platform emerged, powering both the growing number of data-intensive scientific applications and large internet services. It has become essential to run multiple frameworks on the same cluster, so data scientists can pick the best for each application.

As new analytic engines began to cover the ever growing space of problems, sharing a cluster between these frameworks started to get complicated. At the enterprise level, along with the need of batch processing, the need of real-time event processing, human interactive SQL queries, machine learning and graphic analytics emerged.

In a cluster, data are distributed and stored on the same nodes that run computations shared by frameworks. When the cluster is shared, statically, by frameworks, unnecessary data replication will appear, along with utilization issues. When a framework, for example a web-service farm, would be able to scale down at late hours, the MapReduce framework would perform better if it were able to use the resources released by the web-service farm. Sharing improves cluster utilization, through statistical multiplexing and avoids per-framework data replication and leads to data consolidation.

A cluster management system acts as a cluster-wide operating system by sharing commodity clusters between multiple and diverse cluster computing frameworks. Because reading data remotely, is expensive on a distributed file system, it is necessary to schedule computations near their data. At each node, applications take turns running computations, executing long or short tasks, spawned by different frameworks. Locality in large clusters is crucial for performance, because network bisection bandwidth becomes a bottleneck. [2] A cluster management system should provide a tool or interface, to design and implement specialized, distributed frameworks targeted at special problem domains. While multiple frameworks are operating cluster-wide, the operating system should take care of difficult problems, like cluster health, fault monitoring, resource arbitration and isolation.

Energy efficiency also becomes a critical matter for data centers powering large numbers of clusters [6] [7], since energy costs are ever increasing and hardware costs are decreasing. Minimizing the total amount of resources consumed will directly reduce the total energy consumption of a job.

Scalability, resource- and energy-efficiency are key metrics for a cluster management system, their performance matters for data-center operators, as well as for end users. [3] [4] [5]

Driven by the need of a cluster-wide operating system to share data among frameworks, two solutions appeared from the ground of The Apache Software Foundation that circulated widely in the Big Data community, to provide a resource management substrate for analytic engines and their applications. One such solution was designed and presented at U.C. Berkeley, called Apache Mesos and another one, originated from the Hadoop architecture, named YARN (Yet Another Resource Negotiator).

In this work we will show and demonstrate the differences of these two, open-source cluster-wide operating systems, by presenting an infrastructure, resource

management, scheduling overview and performance evaluations, on different scenarios together with load and stress testing. Both of these systems are used widely in production systems and by introducing different resource-management models, it is beneficial to analyze their performance. Using the performance evaluations we will demonstrate the advantages and disadvantages, of different configurations, use cases on both YARN and Mesos, with different analytical frameworks having diverse needs and routines on execution.

2 Design and Concepts

A cluster management system consists of two main components. A *master* entity, that manages resources, schedules framework's resource requirements and *slave* entities, which run on nodes to manage tasks and report to the master. These two components build up the *platform.* A *scheduler* is a singular or distributed component in the platform that schedules jobs (or applications) on the cluster expressed and written by end-users using a specific *framework* library. A cluster management system can be considered as a distributed operating system: it provides resources for frameworks and schedules their distributed applications.

Frameworks are more or less, independent entities, with their own scheduler and resource requirements, but there are dissimilarities among design philosophies on different systems. A live framework is expected to register itself with the cluster's master, by implementing a resource-negotiating API defined by the master. Apart from the global, cluster wide resource management, scheduling, other expectations, such as fault tolerance, job-level scheduling or logging are the framework's duty to provide.

The masters are made to be fault-tolerant on both Mesos and YARN by ZooKeeper [8]. In a cluster deployed with Mesos, a framework must be set up on a given node and it must register itself with the master to be able to negotiate for resources and run tasks on the nodes. YARN requires a *client* to submit the framework, as an *application* to the resource manager. The resource manager will eventually start the framework on a node, making it live, to be able to request resources and run tasks on the nodes.

2.1 Resource Management

As previously described, the master entity arbitrates all available cluster resources by working together with the per-node slaves and the frameworks or applications. The resource manager component of the master entity does not concern itself with framework or application state management. It schedules the overall resource profile for frameworks and it treats the cluster as a resource pool.

There are two methods for gathering resources from the cluster. Mesos *push*es, offers resources to frameworks, those implement a `Scheduler`, while applications, which implement the `YarnAppMasterListener` interface are expected to *pull*, request resources. Mesos offers resources to the `Scheduler` and it chooses to accept, or not, in contrast to the model used by YARN, where the `AppMaster` must request resources from the `ResourceManager` and it chooses to give resources or not.

Resource allocations in YARN are late binding, that is, the application or framework is obligated to use the resources provided by the container, but it does not have to apply them to a logical task on request. The framework or application can decide which task to run with its own, internal, second-level scheduler. In Mesos, task descriptions must be sent upon accepting a resource.

On Mesos and YARN the existing grammar of resource requests does not support specification of complex relationships between containers regarding co-location. Second-level schedulers must implement such relationships. Also, since Mesos offers resources to the framework it will hinder locality preferences, while YARN lets the framework request any node in the cluster, not only from a sub-cluster offered by the resource manager. To tailor and limit resource consumption of different frameworks, a pluggable allocation module in the master entity of Mesos can determine how many resources to offer each framework.

2.2 Scheduling

Given the limited resources in the cluster, when jobs cannot all be executed or resource requests cannot all be served, scheduling their executions becomes an important question, allocating resources to frameworks becomes crucial to the performance. A centerpiece of any cluster management system is the scheduler. Scheduler architecture design impacts elasticity, scalability and performance in many dimensions and data-localities within distributed operating systems.

2.2.1 Statically Partitioned

Statically partitioned schedulers lead to fragmentation and suboptimal utilization. It is not a viable architecture to achieve high throughput and performance, which is an elemental requirement amongst cluster management systems.

2.2.2 Monolithic

A monolithic scheduler uses a central algorithm for all jobs and it is not parallel, implements policies and specialized implementations, in one code base. In the high-performance computing world, this is a common approach, where each job must be scheduled by the same algorithm. The era of Hadoop on Demand (HoD), was a monolithic scheduler implementation. The problem with a monolithic architecture is that it puts too much strain on the scheduler from a certain cluster

size and it becomes increasingly difficult to apply new policy goals, such as, failure-tolerance and scaling.

2.2.3 Two-Level

An approach used by many systems is to have a central scheduler, a coordinator that decides how many resources each sub-cluster will have. This two-level scheduling is used by Mesos, YARN, Corona [9] and HoD. An offer-based two-level scheduling mechanism provided by Mesos, works best when the tasks release resources frequently, meaning that job sizes are also small compared to the total available resources. Since the Mesos master does not have access to a global view of the cluster state, only the resources it has been offered, it cannot support preemption. This restricted visibility of cluster resources might lead to losing work when optimistic concurrency assumptions are not correct. Mesos uses resource hoarding to group (gang) schedule frameworks and this can lead to a deadlock in the system. Also, the parallelism introduced by two-level schedulers is limited, due to a pessimistic concurrency control.

YARN, is effectively, a monolithic architecture, since the application masters usually don't provide scheduling, but only job-management services, like the Spark [10] master entity. An `ApplicationMaster` can in fact implement a second level of scheduling and assign its containers to whichever task is part of its execution plan. The `MRAppMaster` is a great example of the dynamic two-level scheduler as it matches allocated containers against the set of pending map tasks by data locality.

2.2.4 Comparison

Design comparisons, simulations present the tradeoffs between the different scheduler architecture approaches [11]. Increasing the per-job scheduling overhead (the time needed to schedule a job) will increase the scheduler business in the monolithic, single-path baseline case, linearly. The job wait time will increase at a similar rate, until the scheduler is fully saturated. On a multi-path implementation, average job wait time and scheduler activity decreases, but batch jobs can still get stuck in a queue behind service jobs, which are slow to schedule.

Scheduling batch workloads will result in a busier scheduler when using a two-level (Mesos) architecture instead of a monolithic architecture, as a consequence of the interaction between the Mesos offer model and the second-level scheduler in the framework. Because Mesos achieves fairness by offering *all* available cluster resources to schedulers, a long second-level decision time means that nearly all the resources are locked too long a time, making them inaccessible to other schedulers. Mesos predicts by making quick scheduling decisions and having small jobs within a large resource pool, which can cause aforementioned mentioned problems in a different scenario.

4 Experimental Evaluations

In this section we will demonstrate the two cluster management systems in operation, regarding scheduling and execution performance in different scenarios using two popular frameworks, the Hadoop MapReduce implementation and Spark. We test single job execution concerning startup overhead and scheduling efficiency, throughput along with node performances.

These evaluations were run on 5 computers, each equipped with an Intel Core i5 CPU and 4GB RAM. One computer was set up as a dedicated master, resource manager for both YARN and Mesos, history server and proxy server for YARN, but also as a name node and secondary name node for HDFS. The other 4 nodes were set up as data nodes and slaves to run jobs. In the case of Mesos, the frameworks (for example Hadoop `JobTracker`, Hama `BSPMaster`) were deployed and activated on the master node.

In these experiments the following cluster and framework versions were used: Hadoop YARN version 2.5.2 [12], Mesos 0.21.0 [13], with the Hadoop on Mesos library version 0.0.8 [14] and Spark 1.3.0 [15]. We observed no measurable performance differences between MRv1 and MRv2, apart from the overhead originated from launching TaskTrackers.

In each cluster, a total 32 virtual CPUs and 32768 MB of virtual memory were available while running these tests. Both YARN and Mesos were only able to isolate CPU and memory as resources. Disk usage or network bandwidth were managed by the underlying operating system (Ubuntu 14.04). The tests ran 5 times and the results were aggregated to calculate averages. We considered the resource use as use of CPU and memory.

While each cluster management system provides a REST API to query for the system, framework, job, task data as well as metrics, the Mesos API seemed to be a more detailed and precise regarding system parameters. While each job runs as a separate application in YARN, from a cluster point of view, finer grained snapshot becomes visible. Mesos does not know and neither concerns of the job granularity, it really does not know how many map-reduce jobs were ran by the connected `JobTracker` framework entity.

The deployment of YARN with the different frameworks mentioned above worked more like an out-of-box application, compared to Mesos, where permission problems were met several times while running map-reduce jobs along with the frequent node failures in case of handling too much executors at once.

4.1 Single Batch Job Performance

To examine the performance on accepting, scheduling and preparing a certain task, we’ve ran a long, batch-like job on each platform, an IO and memory heavy map-reduce program on a 40 GB dataset that is stored in HDFS with a replication

factor of 2. The cluster utilization was at 0% each time the program gets submitted. Since Mesos uses Hadoop MapReduce API and architecture of version 1 and YARN uses the next-generation version 2 API, differences are expected in map-reduce job execution schemes and performance.

While running the map-reduce job on Mesos, it completes in 1130 seconds with 374 maps and 1 reduce task and with total 358 data-local map tasks, that is 24 more maps ran, than in YARN. *Figure 1* shows that the first `TaskTrackers` of Hadoop version 1 to reach the staging status on an executor launched by Mesos required 8 seconds from the time the job submitted. Staging status refers to the state when the slot (container) is allocated and the setup of the executor has been started. To be able to set up `TaskTrackers`, the Hadoop v1 architecture needs to be distributed as a TAR throughout the slaves, stored on HDFS. After the executors get started, the Hadoop distribution gets downloaded from HDFS, so a working Hadoop must present on each slave with the capability to invoke the `hadoop` command and to communicate with the HDFS. If the Hadoop distribution in question, is not replicated at each node, the transfer time (of roughly 250 MB) heavily impacts the ramp-up time of the `TaskTrackers`.

In a scenario, where network bandwidth is a bottleneck, transferring Hadoop framework executors can keep many tasks on staging status for a long time. *Figure 1* shows that, with replication factor 4 it took 42 seconds to get the first few `TaskTrackers` to get to running state. Other mappers were also considered slow to start up. The reducer was created and launched in the 95th second, while map phase was at 7.6% completion. Reaching the maximum utilization, 1 virtual core and 1024 MB of memory was not used. Mesos set up a total of 11 `TaskTrackers` on the small cluster.

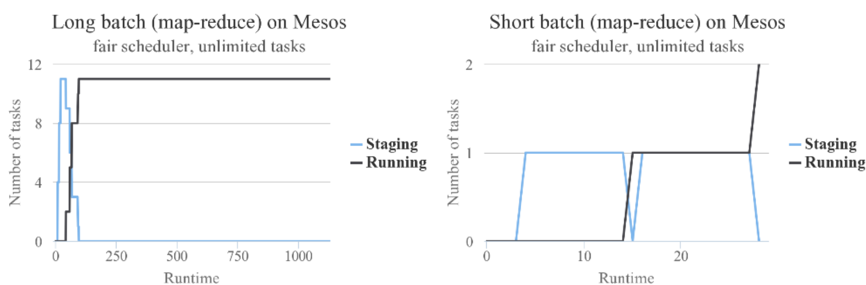

Figure 1

Number of staging and running tasks in case of long and short batch (map-reduce) jobs on Mesos

It became clear that `TaskTrackers`, executors are a huge deal to set up on first time and could mean a slow response from frameworks as elasticity is being harnessed. Burst-like jobs from different frameworks could mean too much work spent on setting up and launching executors.

A much smaller, micro map-reduce job on an 8 MB dataset was run for the same purpose. Mesos created two `TaskTrackers` on different nodes for 1 map and 1 reduce task, with 1 data-local map task. It took 29 seconds for the job to complete, while in contrast to the long batch job, the first task reached staging state on the 4th second and started to run on the 15 second mark. With smaller jobs, tasks get staged and run much faster.

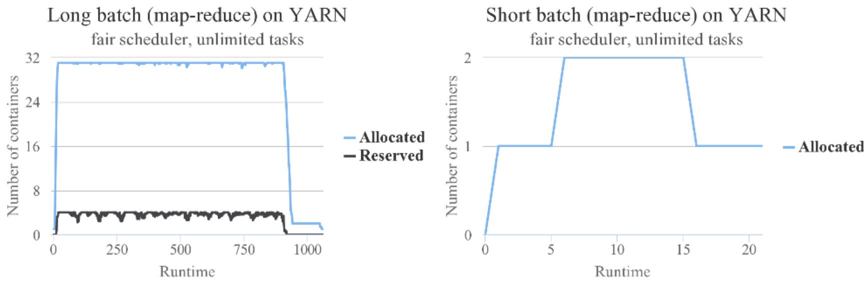

Figure 2
Number of containers allocated and reserved in case of long and short batch (map-reduce) job on YARN

YARN outperformed Mesos on both long and short term. As shown in *Figure 2*, YARN completed the short job in 21 seconds on average and the `MRAppMaster` was placed on container 0 in a second. Breaking with the Hadoop v1 design really pays off, as mappers and reducers are placed very fast on the designated nodes without the `TaskTrackers` to deploy. We found minimal differences in speed comparing the map-reduce implementation of Hadoop v1 and v2. As seen in *Figure 2*, YARN reserved containers to prevent starvation from smaller applications. This behavior appeared to be common on long tasks, but the `MRAppMaster` never reserved more than 4 containers, even on longer jobs, but one for each node.

YARN completed the long map-reduce job with 1 application master, 1 reduce and 360 map containers in total. The advantage of this granularity pays off, when new applications enter the scheduling phase and DRF wishes to free resources for them. Killing a few map tasks to be able to allocate cluster resources for new applications would not result in a major drawback for the map-reduce program running on YARN, since 360 mappers are reserved and used up linearly in the execution timeframe. On a long map-reduce job in case of Mesos, while `TaskTrackers` would allocate slots for a long time, an allocation module would kill a few of them, to place new frameworks' tasks on the cluster. Map-reduce is resilient to task failures, since work lost could be repeated, but on a long term this can hurt utilization and hinder completion time in a much greater aspect. Map-reduce on YARN, provides much better elasticity, along with, a faster execution.

Another issue to point out with Mesos, is that during the execution of the map-reduce long job it consumed relatively more resources on the execution timeframe than the MR implementation on YARN. As *Figure 2* shows after the 940th second, only one reducer was running on one container aggregating results from mappers, but in case of Mesos, until the very end of the job's last phase, all `TaskTrackers` were still running and the `JobTracker` freed them after the job was completed. As higher resource consumption directly affects money spent in a cloud environment, choosing Mesos might result in a higher bill than expected.

Running a micro Spark job on each system resulted in an average 4 second difference, in favor of Mesos. It worth mentioning, that the current Spark implementation does not support cluster deployment mode in for Mesos. Running a Spark job on YARN requires a Spark `ApplicationMaster` to be created on container 0, which impacts the startup time of the actual job. Spark jobs can run in client mode with YARN, but this setup did not yield a better result. The same job on Mesos was run by a Spark client on the master node, thus it was able to negotiate resources and launch containers without the time to deploy itself. Spark jobs on Mesos can run by using a predefined executor with the `spark.executor.uri` configuration parameter or by deploying packages manually to each node with the appropriate configuration.

It is evident, that the deploy-the-application approach introduced by YARN is more convenient from the client's point of view than the connect-the-framework concept. A client does not have to keep its instance of framework running and can disconnect from the cluster after the application got submitted. On the other hand, deploying a framework manually to a node could lead to uncontrolled resource consumption as the framework is not managed and isolated by the resource manager. Using Spark in client mode means that multiple Spark-framework instances will appear and act as tenants for the DRF scheduler, while one `JobTracker` runs multiple map-reduce programs. This approach will eventually make the tasks of the allocator module harder, when it tries to enforce organizational policies.

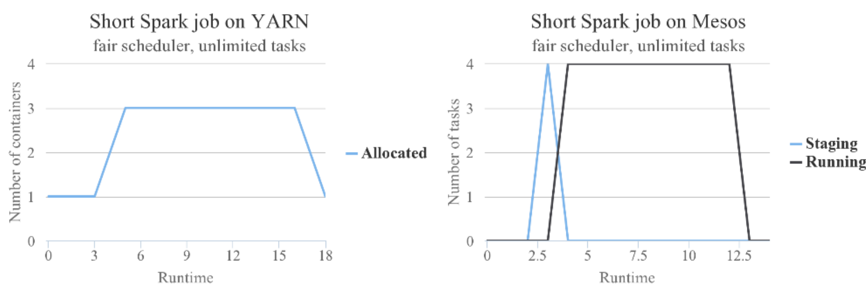

Figure 3

Number of containers allocated (YARN) and number of tasks staging and running (Mesos) in case of running the short Spark job

As seen in *Figure 3*, YARN completed the Spark job in 18 seconds using 3 containers (including the Spark master on container 0), while Mesos in 14 seconds using 4 containers. The running container 0, on YARN, required roughly 5 seconds to request, prepare and run the Spark executors, on the allocated containers. The executors were running for 11 seconds and the application master went offline after 2 seconds. The execution of Spark job on 4 containers resulted in a timeframe of 7 seconds.

Table 1
Summary of single job performances on YARN and Mesos

Case	Runtime	Maximum number of containers used
YARN, short map-reduce job	21 s	2 (including application master)
Mesos, short map-reduce job	28 s	2
YARN, long map-reduce job	1061 s	31 (including application master)
Mesos, long map-reduce job	1129 s	11
YARN, short Spark job	18 s	3 (including application master)
Mesos, short Spark job	14 s	4

4.2 Mixed Job and Framework Performance (Scenario 1)

To test the scheduling performance of Mesos and YARN, we've created a client that submits map-reduce and Spark jobs periodically. In this scenario, a micro map-reduce and a CPU Spark job was submitted every 10 seconds, and a long batch map-reduce job every 100 seconds. A total of 22 jobs were submitted.

Using YARN as a platform, with the fair scheduler and unlimited application preferences, we were able to encumber the system to a point, where the context switching and administration overhead turned each running application into a zombie as `NodeManagers` were overwhelmed. As seen in *Figure 4*, after completing 22 applications, the last 20 never reached a complete state, but actually did not make any progress in hours. The scenario became complicated for YARN, when the long-job entered the cluster and a huge portion of resources were allocated to it, rendering micro-job executions slower, causing them to pile up. It became evident that concurrent application limits are crucial for performance, after a certain threshold on YARN as context switching and parallelism overhead went out of control. For this system and with this scenario, it happened with 15 applications. As memory-intensive applications were still running and requesting a resource vector with memory greater than (1 CPU, 1 GB), in average, 12 virtual cores were never used. The reserved values on the dimension of memory were introduced by the long running map-reduce job. The characteristic leap of the allocated plot line refers to the time when the first long job appeared and started to acquire all available resources.

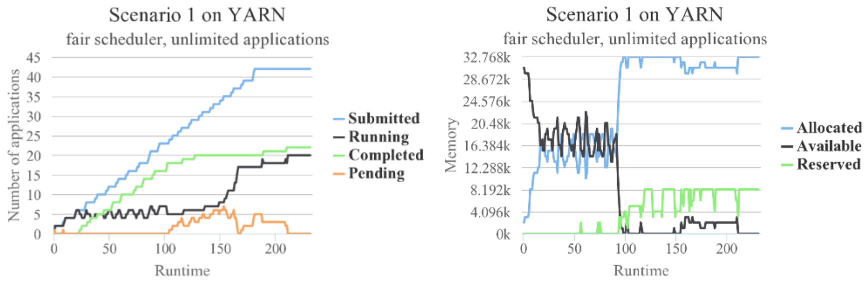

Figure 4

Evaluation of scenario 1 on YARN using fair scheduler with unlimited applications configuration in terms of number of applications and memory usage

Using YARN's capacity scheduler with a limit of 4 concurrent applications, this scenario was completed in 2168 seconds and as seen in *Figure 5,* compared to fair scheduler with an application limit of 10, was slower. Fair scheduler completed applications in 2132 seconds, while also performed with a better response-time as smaller jobs were able to run earlier. For larger job sizes, capacity scheduler provided a better response-time with a lower application limit. In the case of capacity scheduler no reservations were made for new containers. By not reserving containers, it seems a few containers were unused and scheduled on-the-fly after they became available.

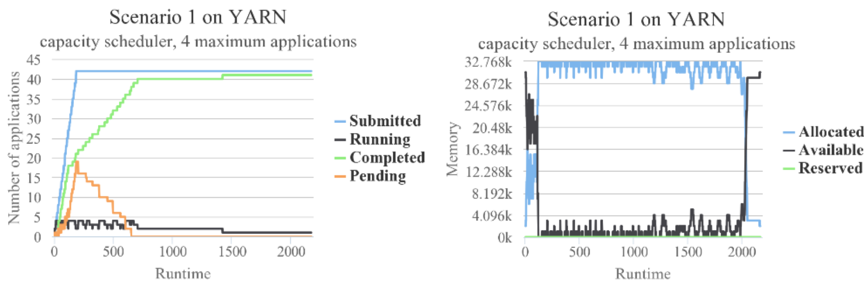

Figure 5

Evaluation of scenario 1 on YARN using capacity scheduler with maximum of 4 concurrent applications in terms of number of applications and memory usage

The FIFO scheduler with unlimited applications completed this scenario, on average 2153 seconds. This scheduling scheme can hurt smaller jobs and can cause starvation when a single, long job gets all resources as seen in *Figure 6*.

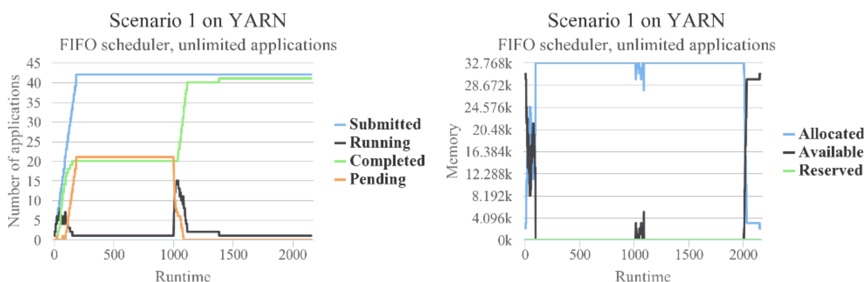

Figure 6

Evaluation of scenario 1 on YARN using FIFO scheduler with unlimited applications configuration in terms of number of applications and memory usage

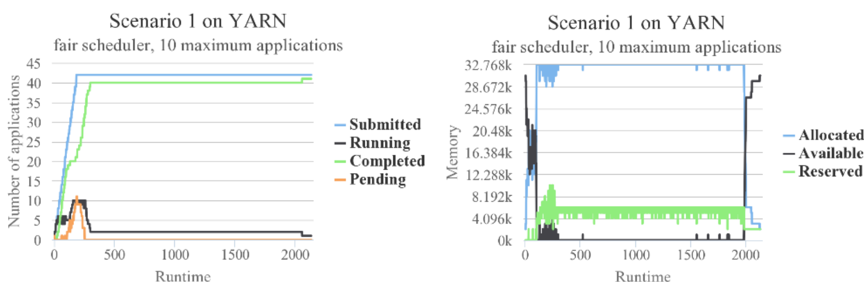

Figure 7

Evaluation of scenario 1 on YARN a maximumir scheduler with maximum of 10 concurrent applications in terms of number of applications and memory usage

Comparing the DRF implementation of Mesos to YARN's, YARN was able to perform much better and achieved a high utilization by allocating 100% of the available cluster resources for a long period of time, as shown on *Figure 7*. With the use of reserved amounts by the scheduler, containers were allocated and ran much faster achieving a higher utilization, than the capacity scheduler. Mesos, on the other hand, was not able to utilize all cluster resources. For a long time, the resource manager reported 4 CPUs and 6.1 GB memory idle, but the fine-grained, rapid tasks of Spark were utilizing the 4 CPUs as seen in *Figure 8*. Spark was set up in fine-grained mode in the first place, which means a separate Mesos task was launched for each Spark task. This allows frameworks to share cluster resources in a very fine granularity, but it comes with an additional overhead for managing task lifespans in a rapid rate. Focusing on the number of cores in the fine- and coarse-grained setup this behavior seems clear, as the fast task allocation pattern appears on the plot in the fine-grained case. A noticeable difference shows in the memory allocation pattern of different task-resolutions as (Spark) tasks with a lifespan measured in milliseconds allocated containers with <1 CPU, 128 MB> resource vectors instead of <1 CPU, 512 MB> or <1 CPU, 768 MB> as in the coarse-grained mode. In certain circumstances, it might be a good practice to

place and force very fast tasks of different fine-grained setup frameworks next to memory-intensive jobs to improve utilization and fairness with Dominant Resource Fairness.

The map-reduce jobs were managed by a single `JobTracker` and Spark jobs were submitted by multiple Spark clients. By increasing the number of Spark clients, the utilization improved. *Figure 8* shows the utilization best achieved while 6 Spark frameworks were active.

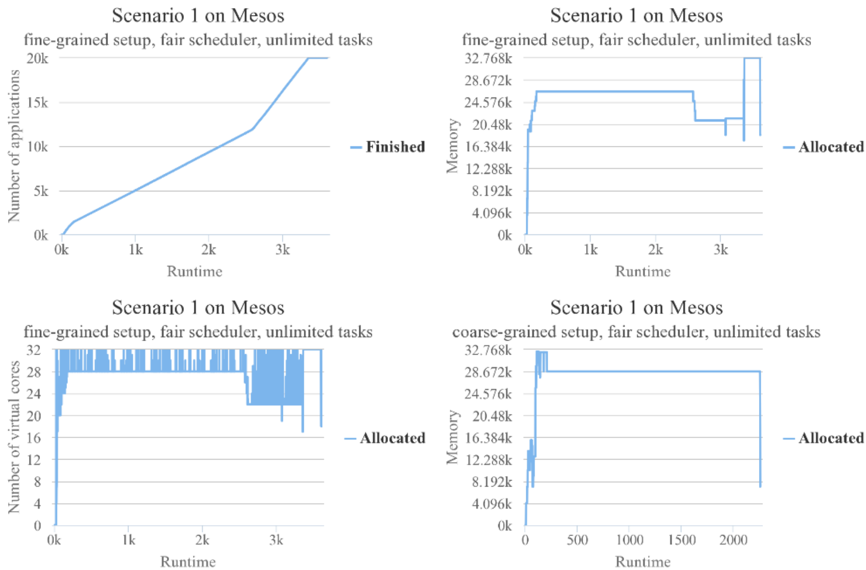

Figure 8

Evaluation of scenario 1 on Mesos using coarse- and fine-grained setup with fair scheduler and unlimited tasks configuration in terms of number of applications, virtual core and memory usage

In this scenario, multiple issues were found with Mesos. On some runs, an average of 24 `TaskTrackers` were lost and some of them were stuck in staging status, never reached running state. Also, one or two slaves were tended to disconnect from the master and froze in the first minutes of this scenario. The recorded and aggregated results were used, when Mesos did not lose a task.

The following table shows a summary of the results experienced from scenario 1.

Table 2
Summary of runtimes and average utilization experienced in scenario 1

Case	Runtime	Average utilization (CPU)	Average utilization (memory)
YARN, fair scheduler, unlimited	infinite	100%	100%
YARN, capacity scheduler, 4	2168 s	82.35%	89.87%
YARN, FIFO scheduler, unlimited	2153 s	83.27%	92.71%
YARN, fair scheduler, 10	2132 s	86.27%	92.90%
Mesos fine, fair scheduler, unlimited	3604 s	83.96%	78.46%
Mesos coarse, fair scheduler, unlimited	2256 s	90.81%	89.25%

The Spark implementation in fine-grained mode using Mesos spanned close to 20000 tasks in this scenario, it has put a strain worth mentioning on the master to schedule resources. While YARN performed about 1.7 times better than Mesos (with fine setup) with a relaxed (unlimited applications or tasks) DRF setting, due to the lightweight nature of Mesos it handled fine-grained tasks better as a first level scheduler. It has become clear that Mesos is more reliable and more suited for running large amounts of frameworks and tasks-per-framework with very fine-grained tasks.

4.3 Micro-Job Performance (Scenario 2)

In scenario 2, we've prepared a script, which submitted 4 micro-applications or jobs if you will, 2 map-reduce and 2 Spark job in each 10th second for 30 times. A total of 120 jobs reached the cluster. Our goal were to evaluate how fast short jobs can enter and leave the cluster on both systems and to see if there's any chance of overwhelming the slaves by increasing parallelism overhead to an undesirable level.

It has been shown in the demonstration of Mesos running a long batch job, `TaskTrackers` have a high startup overhead so our expectations were met about the difficulties these cases would produce. A standby TaskTracker might provide significant benefit regarding task start-ups, but it would also introduce data-locality problems, since a data might not be available where our TaskTracker has been deployed. Designing heuristics to keep TaskTrackers wisely on certain nodes, suggested by workload statistics, would not solve all of our problems on a long term. As seen on *Figure 9* YARN performed very well, by not letting pending tasks to reach 3 as applications were able to finish in a fast rate and were not interrupted by and overcrowded cluster. Applications were completed linearly with time and on average roughly 10 were running concurrently. In case of Mesos, as seen on the curve of staging tasks, in the first 60 seconds every Spark job entered the system were able to run and complete without unnecessary staging.

In the first 100 seconds only `TaskTrackers` were staging for a longer period of time, which meant that because of their startup time, those rapid map-reduce jobs arriving in every 10 seconds were not reached running status fast enough. Due to the fact that map-reduce programs were spawning on `TaskTrackers`, unnecessary parallelism appeared on slaves and about 20 map-reduce jobs were running concurrently along with the Spark jobs on the cluster.

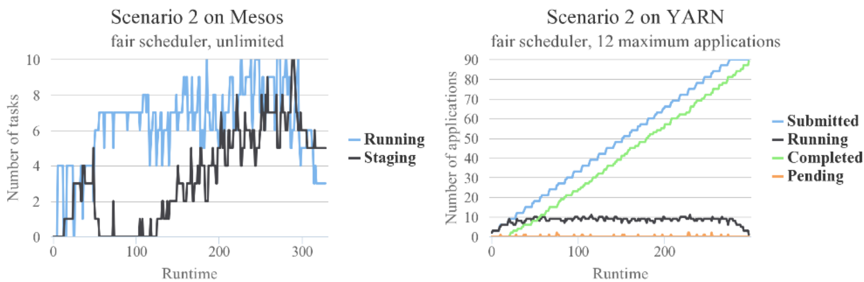

Figure 9

Evaluation of scenario 2 on Mesos (unlimited tasks) and YARN (maximum 12 applications) using fair scheduler in terms of number of tasks and application

On the memory footprint produced by tasks running on Mesos as shown in *Figure 10*, the `TaskTrackers` crowding the cluster are visible on the 30th to 110th second interval. After that point a few of them were broken down to be able to offer resources to Spark programs. YARN, in contrast, kept resource consumption constant as applications were not able to encumber the cluster.

The container reservations used by YARN's fair scheduler helps applications to receive and utilize containers faster than the resource offer approach introduced by Mesos. Mesos completed this scenario in 328 seconds, while YARN in 297 seconds. Again, issues were found, but this time with the `JobTracker` (Hadoop v1): in some cases map-reduce jobs were stuck and never reached running state on `TaskTrackers`.

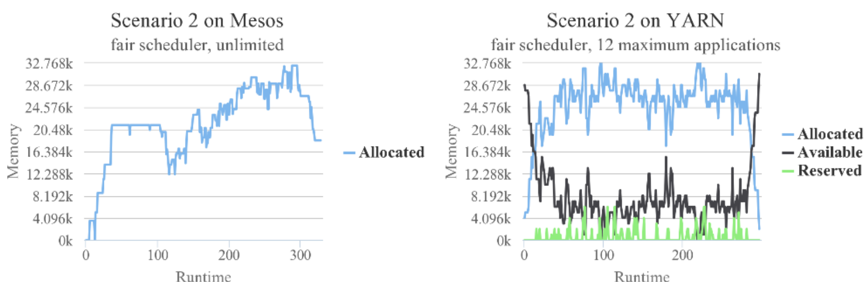

Figure 10

Evaluation of scenario 2 on Mesos (unlimited tasks) and YARN (maximum 12 applications) using fair scheduler in terms of memory consumption

4.4 Micro-Job Interruptions (Scenario 3)

To evaluate the problems related to granting fairness and job interruptions, we've prepared a scenario, where micro Spark programs were submitted on a long running map-reduce batch job. After the submission of the same long job, that was previously evaluated, for every 100 seconds a Spark CPU-heavy program was submitted, nine times longer overall.

Interrupting the map-reduce job's execution with micro Spark jobs on Mesos added, on average 9 seconds to the overall completion time, which became 1138 seconds. Recall the results of the same long job performance of YARN and Mesos from *Figure 1* and *Figure 2.* On YARN, the same scenario stretched the completion time of the map-reduce batch job, from 1061 to 1092 seconds.

On Mesos, 1 CPU was available with 1 GB RAM and the Spark client was able to initiate a start on a free container, where it completed in 48 seconds on average. Since YARN were utilizing all cluster resources during the execution of the map-reduce job most of the time, the Spark programs needed to wait on average 13.3 seconds to be able to progress to running state from pending state. Theoretically, every 2.6 seconds, a mapper finish (from the length map phase and number of maps) and its resources <1 CPU, 1 GB> frees up. On average, 3 containers were reserved by the `MRAppMaster` and the Spark job needs 2 containers (including the application master) to be able to run. When the Spark job reached the pending state, on average, 5 containers needed to free up, to be able to reach running state, which is roughly 13 seconds.

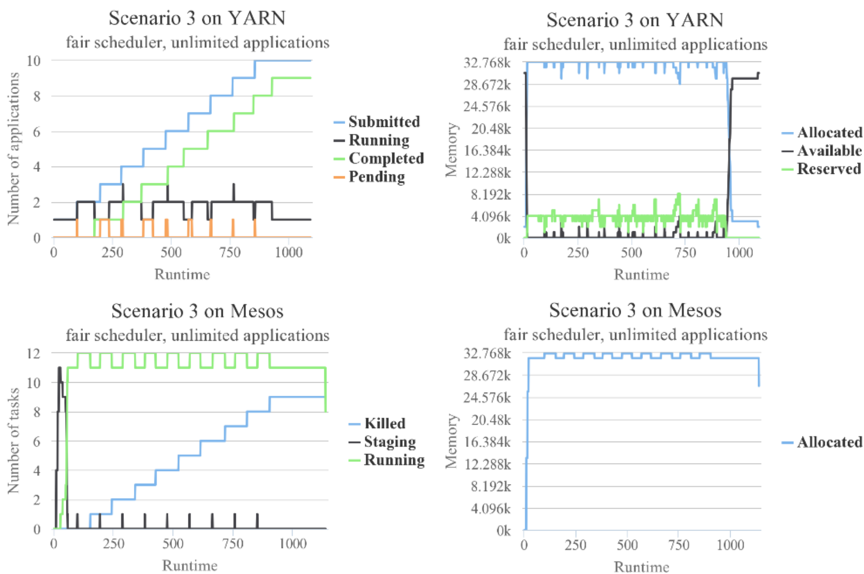

Figure 11
Evaluation of scenario 3 on Mesos (unlimited tasks) and YARN (unlimited applications) using fair scheduler in terms of number of tasks, application and memory usage

As seen in *Figure 11*, the same Spark job on YARN needed more time to start, mainly because the application master is necessary to be placed on a free container, which set back and slowed subsequent pending mappers. Utilization of YARN still proved to be better than in the case of Mesos.

Conclusions

Cluster management systems are the backbone of any Big Data analytical toolsets that are used by an Enterprise, their performance is determined significantly, by their design and greatly affects energy consumption for a data center. Evolution of Hadoop had the greatest impact in the motivation, concept and birth of these systems. We focused on the main two-level, open-source schedulers available, YARN and Mesos.

YARN is perfect for ad-hoc application deployment, as it ships the application to the requested node by carefully setting up the process in all cases. YARN has been made for an environment with higher security demands, as it protects the cluster from malicious clients and code in many cases. The differences in resource allocation methods showed that the push-method used by Mesos might hinder utilization and locality preferences in some cases, but proved to be faster than YARN's, which provides agility by using late-binding in opposite fashion. Applications running on YARN have the benefit of making better second-level scheduling decisions, because they have a global view of the cluster, whereas a framework have sight of only a subset of the cluster on Mesos. In a consequence of the resource allocation method, YARN supports preemption to prevent starvation. Restricted visibility of cluster resources might lead to losing work and resource hoarding used by Mesos can lead to a deadlock within the system. Mesos predicts outcomes to make quick scheduling decisions.

The functionality of YARN proved to be richer by providing convenient services for applications, but also supports more scheduling methods and algorithms. Capacity schedulers can work effectively when the workloads are well known. Fair scheduler introduces several problems with head-of-line jobs, but Delay scheduling addresses them and improves locality. HaSTE is a good alternative on YARN, when the goal is to minimize makespan in the cluster.

Regarding system parameters, the API provided by Mesos is more detailed and precise, but YARN gives a finer grained snapshot as each job runs as a separate application. Mesos does not know the job granularity of a connected framework, which can cause several problems. Deployment of YARN is usually more convenient, due to the higher level and more comprehensive interfaces available. It works more like an out-of-box product. During the evaluations, several issues were found with Mesos regarding permissions and node failures.

YARN has a wider and more diverse analytical toolset (frameworks) available than Mesos, but a practical decision between these platforms might include special requirements. The mainstream frameworks are mostly available on both systems.

As single job performance evaluations show, executors are difficult to set up the first time and can mean a slow response from frameworks like Hadoop MapReduce. YARN deploys mappers and reducers, much faster on the designated nodes and can provide better locality, also, this task-granularity provides better elasticity along with a faster execution. As TaskTrackers are expensive to deploy and they are long running, killing them to provide fairness is usually a significant drawback. Cached or standby TaskTrackers might provide significant improvements in task start-ups, but it would introduce other problems, for instance, a hindered data locality. Single job benchmarks also showed, that map-reduce jobs on Mesos run longer and consume more resources, which directly affects money spent in a cloud environment. YARN is 1.06 (in case of short map-reduce) and 1.33 (in case of long map-reduce) times faster than Mesos, but Mesos runs a micro-Spark job 1.28 times faster than YARN. It must be taken to account that YARN has to deploy the submitted application each time, while the framework's master runs separately on Mesos.

Multiple scenarios showed that the "concurrent application limit" is crucial for performance after a certain threshold on YARN. Using preemption, YARN performed about 1.7 times better than Mesos with the fine-grained setup, but Mesos handles large amounts of tasks better as a first level scheduler. Mesos is more reliable and more suited for running large amounts of fine-grained tasks. With the same setup, YARN provides a 1.05 times faster execution and 4.54% less CPU consumption. It is evident that the container reservations, used by YARN's fair scheduler, can utilize and provide containers to applications faster than the resource offer approach introduced by Mesos. Other scenarios showed that overall utilization on a YARN cluster is better, along with a 1.10 times faster execution.

References

[1] WHITE, Tom. Hadoop: The definitive guide. O'Reilly Media, Inc., 2012

[2] DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51.1: 107-113

[3] LIANG, Fan, et al. Performance benefits of DataMPI: a case study with BigDataBench. In: Big Data Benchmarks, Performance Optimization, and Emerging Hardware. Springer International Publishing, 2014, pp. 111-123

[4] JIA, Zhen, et al. Characterizing and subsetting big data workloads. arXiv preprint arXiv:1409.0792, 2014

[5] TAN, Jian; MENG, Xiaoqiao; ZHANG, Li. Performance analysis of coupling scheduler for mapreduce/hadoop. In: INFOCOM, 2012 Proceedings IEEE. IEEE, 2012, pp. 2586-2590

[6] CHEN, Yanpei, et al. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In: Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012, pp. 43-56

[7] KUMAR, K. Ashwin; DESHPANDE, Amol; KHULLER, Samir. Data placement and replica selection for improving co-location in distributed environments. arXiv preprint arXiv:1302.4168, 2013

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

[9] "Corona" 2013 [Online] Available: https://github.com/facebookarchive /hadoop-20/tree/master/src/contrib/corona

[10] ZAHARIA, Matei, et al. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010, p. 10-10

[11] SCHWARZKOPF, Malte, et al. Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013, pp. 351-364

[12] "Apache Hadoop 2.5.2," 2014 [Online] Available: http://hadoop.apache.org/docs/r2.5.2/

[13] "Apache Mesos" 2014 [Online] Available: https://github.com/apache/mesos

[14] "Hadoop on Mesos" 2014 [Online] Available: https://github.com/mesos/hadoop

[15] "Spark" 2014 [Online] Available: https://github.com/apache/spark