Analyzing Spark Cluster Performance in Amazon EMR

HY543: Big Data Project | 2024

Spyridon Tsagkarakis stzagkarak@csd.uoc.gr

Efthymios Papageorgiou papageorgiou@csd.uoc.gr

Irodotos Apostolou apostolou@csd.uoc.gr

1. Introduction

Over spring break our team conducted experiments in order to evaluate the performance of Apache Spark on Amazon EMR. This report contains a thorough analysis of the experiments conducted, their outcomes, and findings.

Apache Spark is a highly popular analytics engine for large-scale data processing. Using Apache Spark, programmers are able to program entire clusters with implicit data parallelism and fault tolerance. Being a general-purpose framework, it can be configured to process workloads, such as batch processing, interactive queries, real-time stream processing, machine learning, and graph processing. Spark integrates well with other big data tools and storage systems, such as Hadoop HDFS, Apache Hive, and Apache HBase. It can also work with cloud storage systems like Amazon S3.

Cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud all provide services that simplify the process of running large-scale data processing frameworks. In the case of Amazon, the service provided is called EMR (Elastic MapReduce). EMR allows users to set up, configure, and scale clusters of virtual servers to process vast amounts of data efficiently. One of the primary frameworks provided by EMR is Apache Spark, making it easy to run Spark applications in a scalable and managed environment.

In order for Amazon EMR to manage a cluster running Spark, two enabling technologies are required. A distributed storage system (1) and a resource management system (2). The distributed storage system that is commonly used with Spark and is bundled with EMR is HDFS (Hadoop Distributed File System). HDFS provides reliable and fault-tolerant storage to all cluster nodes. Nodes running Spark workloads use HDFS in order to read and write input and output workload data. For resource management, EMR uses a modified version of YARN (Yet Another Cluster Manager), a resource management layer for Hadoop that allows different data processing engines, such as Spark, to run and manage resources on a cluster.

2. Benchmark Suite

For all experiments conducted in this analysis, the workloads used were provided by HiBench. HiBench is a comprehensive benchmark suite designed to evaluate and measure the performance of big data frameworks and systems, such as Apache Spark. We used the HiBench Apache Spark module labeled as “sparkbench”. From “sparkbench”, 3 distinct workloads were used. The following numbered list contains a brief explanation of each of the 3 workloads.

1. Under the micro category, we chose the **Sort** (micro/Sort) workload. As the name suggests, the workload sorts a configurable, in size, array of random strings. In detail, the workload splits input data into a number of partitions using `HashPartitioner`. Then, using `sortByKeyWithPartitioner` and `RDD.save` it sorts and merges the partitions into one resulting RDD. For this workload we used the Gigantic dataset input, meaning that the initial generated array of random strings was about **32.5 Gigabytes**.

2. Under the machine learning category, we chose the **Random Forest** (ml/rf) workload. This workload is set to train and evaluate a Random Forest Regression Model. In more detail, the workload splits input data into a train and test set (70-30), using the `Spark.mllib.RandomForrest` implementation in order to train a classifier as well as make predictions using the test set. For this workload we used the Gigantic dataset input, meaning that **10.000 examples** were used, each having **300.000 features**. In total, the input data generation was measured to be around **24 Gigabytes**.

3. From the SQL category, we chose to include the **Join** (SQL/Join) workload. This workload is set to run a large amount of Join queries in a fabricated database using Hive. The input SQL script is read and split using Scala Spark primitives. Then, sequentially each query statement is executed using Hive. For this workload, we also used the Gigantic dataset as input. The dataset executes **100.000.000 queries** across **12.000.000 pages**. In

total the generated input data was measured to be around **19.1 Gigabytes** in size.

3. EMR Clusters

Amazon EMR categorizes cluster nodes into 3 distinct groups.

First, the Primary nodes are responsible for managing the cluster. They run primary components of distributed applications such as the YARN resource manager master, and the Hadoop HDFS Namenode service. The primary node is used to execute Spark Driver programs, aka Spark Jobs. Such programs create SparkContexts that schedule and submit tasks to the cluster.

Secondly, Core nodes run the Data Node daemon to coordinate data storage as part of the Hadoop Distributed File System (HDFS). By default, they are also configured to accept and run parallel Hadoop MapReduce tasks and Spark executors.

Lastly, Task nodes are used to perform parallel computation tasks on data, such as Hadoop MapReduce tasks and Spark executors. Task nodes don't run the Data Node daemon, nor do they store data in HDFS. They fetch RDD partitions from Core nodes when required, perform the analogous computations based on the tasks at hand and return the data back to Core nodes when they finish.

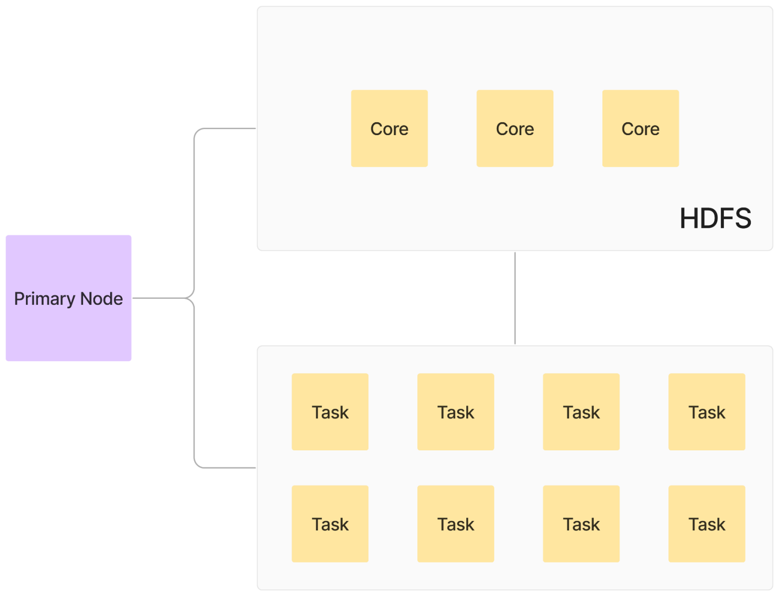


Figure 1: A sample EMR cluster configuration consisting of one Primary node, three Core nodes and 8 Task nodes

4. Experiment Cluster Setup

You can find instructions, plots, results as well as all available resources used in order to execute this project's experiments in this public repo.

4.1 Primary Node Configuration

For our experiments, we configured our clusters with one, capable, Primary node (capable in terms of memory, CPU, and network performance in order for it to not affect Spark Driver performance. We used an m5.xlarge instance). We initiated the execution of our workloads on these nodes.

4.2 Core Node (HDFS) configuration

For our experiments, we made sure that our cluster had a stable, static Core node layout as our goal was not to benchmark HDFS. We used 3 Core nodes, of instance type m5.xlarge, each one having 2 EBS volumes of 64 Gigabytes each. That brought our cluster's **total HDFS storage capacity to ~390 Gigabytes** (128 times 3). That amount of storage was enough to generate and prepare all workload input data into the HDFS before we executed the workloads.

4.3 Task Node Configuration

The number of Task nodes configured in the EMR clusters used on the experiments depended on the type of experiment. For experiments that did not evolve the changing the number of Task nodes, the cluster was set up with 8 Task nodes, all of the same instance type.

4.4 HiBench Setup

In each cluster, after waiting for the appropriate time for EMR to configure all requested machines, we "ssh" on the Primary node and execute a setup script in order to automatically configure and configure HiBench on the cluster. Afterward, if needed, we tweaked specific parameters under HiBench's conf folder, and using this simple script we initiated the experiments.

4.5 Result Aggregation and Plotting

We collected all metrics from HiBench's report file into a Google Sheet. We plotted experiment results using matplotlib python scripts. The plotting scripts as well as the plots can be found here.

4.6 Analysis Overview

We divided our analysis and experiments into 3 categories, all motivated by analogous questions we had about system performance and behavior under specific circumstances. We conducted experiments in order to answer those questions.

The first part of our performance analysis focused on the Spark resource configuration. Specifically, we wanted to identify the best Spark executor/core/memory configuration that resulted in the highest throughput and, thus higher performance.

The second part of the analysis focused on Spark's behavior and performance on varying cluster sizes. Specifically, we wanted to identify Spark behavior when the workload tasks are split into a greater number of machines.

The third and last part of the analysis investigated the role of CPU performance in Spark computation heavy workloads. Specifically, we wanted to identify if CPU performance majorly translates into higher throughput.

4.7 Average Result Deviation

We conducted initial experiments in order to identify the average execution time and throughput deviation of the selected Hibench workloads in an EMR cluster. The findings were as follows:

micro/sort bench	Time (in seconds)	Throughput (in Bytes)
AVERAGE	250	130347889,7
AVERAGE DEV	8,44	4661890

ML/rf bench		
AVERAGE	1.582	15166967
AVERAGE DEV	5,55	52226
sql/join bench		
AVERAGE	143	132483392,3
AVERAGE DEV	2	1718023

We can clearly see that all workloads have minor execution time and throughput deviations.

Q1: The “Best” Spark Configuration

How does the choice of the number of executors, cores per executor, and memory per executor affect the throughput of Spark jobs?

Q1: Experiment Setup

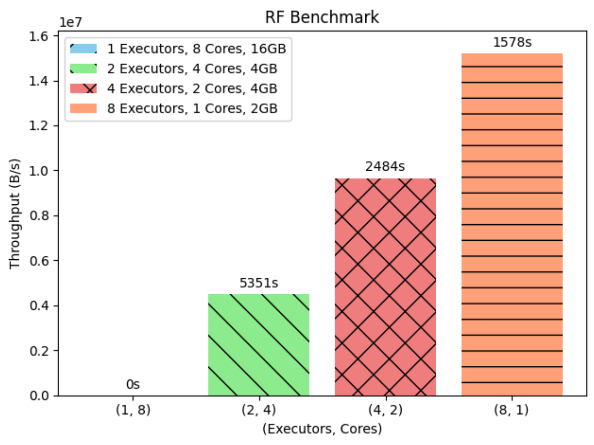
For this experiment, we kept the number and type of cluster nodes static. (meaning that, as stated above, the clusters contained 1 Primary node, 3 Core nodes, and 8 Task nodes. All m5.xlarge instances [4 virtual cores, 16 Gigabytes of RAM]).

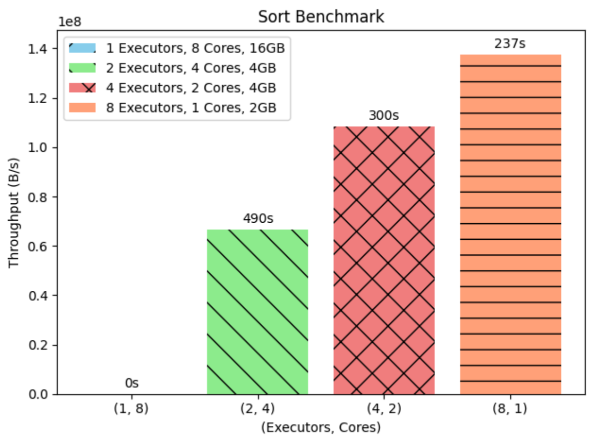
In order to tweak the number of executors, cores per executor, and memory per executor we tweaked the analogous Spark properties, disabling dynamic executor allocation in the process.

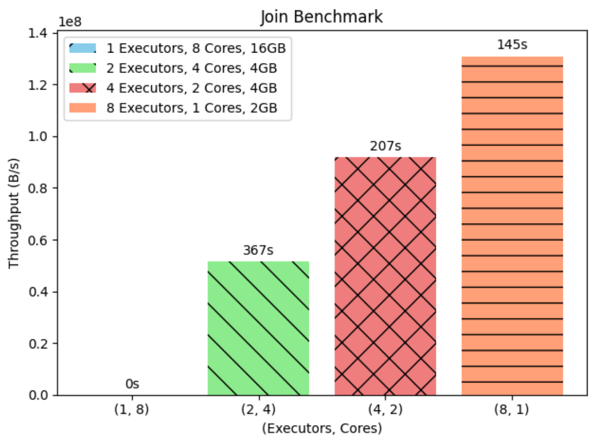
The property `spark.executor.instances` controls the number of executors (JVM worker binaries) that are initiated on each machine that runs map-reduce operations on the cluster. Overwriting this property disables dynamic executor allocation.
The property `spark.executor.cores` controls the number of cores given to each executor. The number of cores an executor holds affects the amount of Tasks that can be executed in parallel on that executor.
The property `spark.executor.memory` controls the amount of memory to allocate to each executor’s JVM process. The amount of memory allocated to each executor’s process affects its capacity to store intermediate data, cached data, and execution buffers.

In each experiment we allocated the available node resources differently to the 3 above properties, keeping in mind the finite amount of resources that the nodes contain. We ran our workloads on the selected configurations and we present our findings below.

Q1: Results and Insights







Note: We were unable to run the <1 Executor, 8 core, 16 GB> configuration as we were unable to change Yarn's pre-configured properties (specifically `yarn.scheduler.maximum-allocation-mb`). In order to change Yarn's properties in EMR clusters you need to do so programmatically using the aws cli. Our aws accounts did not have the necessary permissions to set up an AWS CLI session. Changing the property manually on the configuration files of the primary node did not actually alter the property's value as restarting the service or cluster resets the property to its default value (8 Gigabytes).

We observe that for all 3 workloads, the configuration that contains the **largest number of executors** clearly **outperforms** other configurations. Spark performance in EMR clusters increases when the amount of executors in a Spark Job is increased. For the selected benchmarks this also means that the **cores per executor, and memory per executor play a lesser role in overall performance**.

For I/O heavy workloads (sort and join) doubling the number of executors from 4 to 8 decreases the total compilation time by 1 minute, a significant improvement in a *relatively* fast workload for a cluster of this size. The computational heavy workload (rf) is also majorly sped up (reduced by ~16 minutes) when more executors are provided.

Q2: Cluster Task Node Variation

How does increasing the amount of Task/worker nodes affect workload performance?

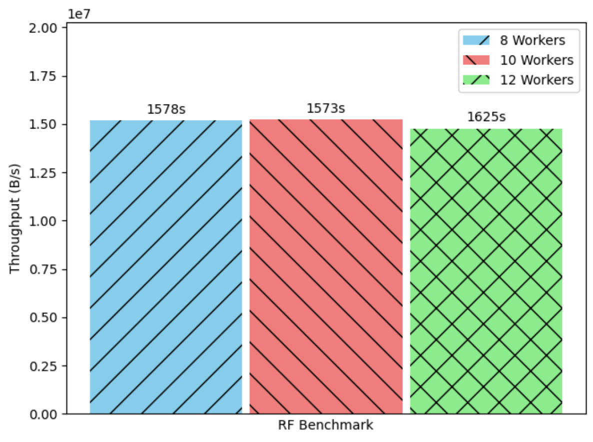
Q2: Experiment Setup

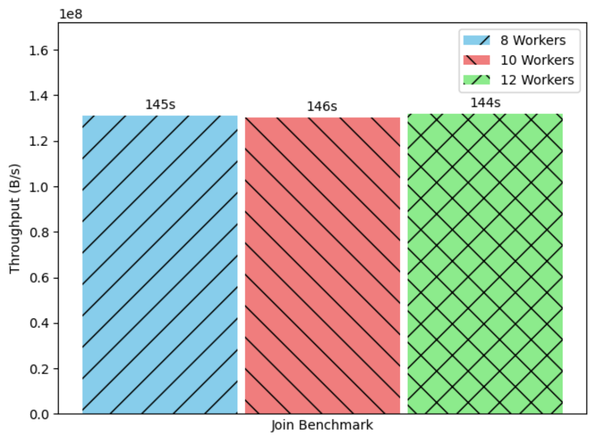
For this experiment we kept the cluster configuration static, only changing the amount of Task Nodes allocated to the cluster.

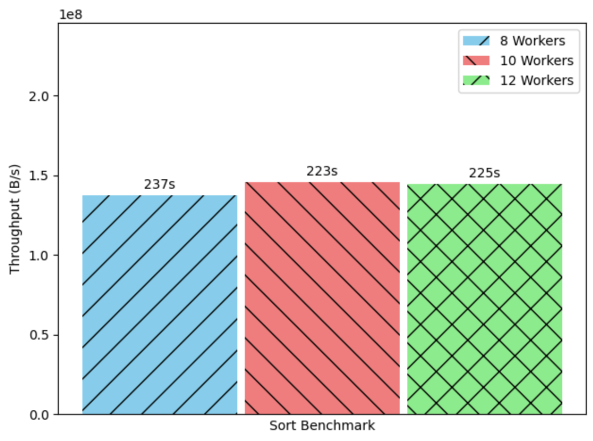
As previously stated, Task nodes are used to perform parallel computation tasks on data. They are not configured to run HDFS, they communicate with Core nodes in order to receive data to process.

Note: Admittedly, we had misunderstood the role of Core nodes in an EMR cluster. We were under the assumption that Core nodes did not perform map-reduce operations, something that is not the case. Nevertheless, increasing the number of Core nodes instead of Task nodes would also affect the cluster's HDFS setup, something that we shouldn't have strived for as it would have altered our results and converged from the initial goal of the experiment.

Q2: Initial Results and Insights







We clearly observe that **increasing the amount of Task nodes did not contribute to an increase in performance**. This trend is apparent regardless of the workload executed.

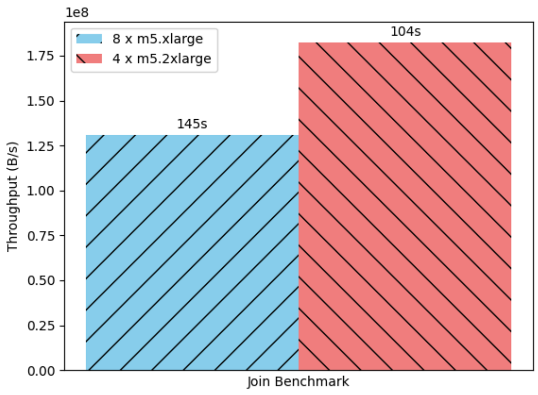
We had many initial hypotheses about why this was the case.

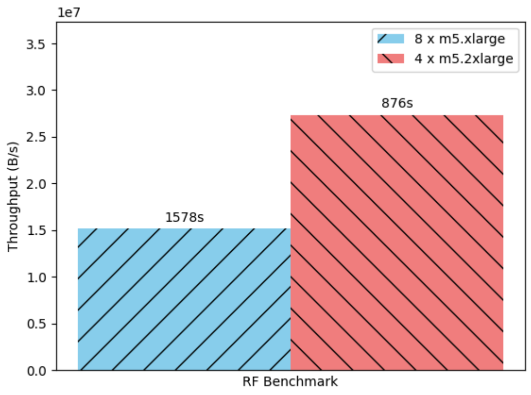
One of our hypotheses that was quickly dismissed was that the workloads were too small to be split into a greater number of parallel tasks. However, it is possible that this may not have been the case though as the number of tasks generated by Spark was proportional to the amount of Task nodes allocated in the cluster and the workload input was sufficiently big to be split into a larger amount of tasks.

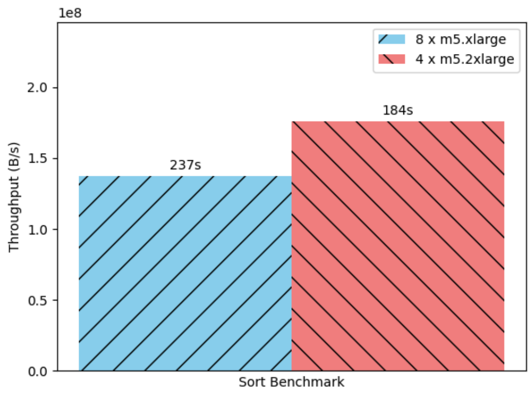
Another hypothesis that turned out to be valid was that data locality or better said, "data remoteness" was a major issue hindering the performance of our workloads when scaled to more task machines. In order to validate this claim we conducted a follow-up experiment.

Q2: Testing the Data Locality Hypothesis

This experiment focused on increasing data locality on Task nodes of the cluster. In order to test this hypothesis we cut the size of the Task nodes the cluster contained in half. We replaced the original 8 Task node instances with 4, double in available resources (virtual CPUs and available RAM), instances while keeping the same amount of total executors in the cluster. The Spark configuration of `<8 Executors, 1 Core per Executor, 1 GB of RAM per executor>` (8 * 8 = 64 total executors) was modified to `<16 Executors, 1 Core per Executor, 1 GB of RAM>` (4 * 16 = 64 total executors).







We clearly observe that by **increasing the executors in each machine** the **cluster's performance** on the selected workloads is **improved dramatically**.
Data locality is a major contributor to performance in EMR Spark clusters and, according to our beliefs, researchers and **organizations using Amazon EMR should optimize their clusters according to this**.
Another important thing to note is that renting fewer, more powerful machines comes at exactly the same cost in AWS. At the time of writing this, renting two m5.xlarge instances costs a total of 117.73 USD per month (spot instance pricing). Similarly, renting one m5.2xlarge costs exactly the same! Thus, using less but more powerful machines that can support a larger number of Spark executors rather than more, less powerful ones is a great strategy in order to improve a cluster's performance in non-interactive/static workloads (such as the ones used in this analysis).

Q3: CPU comparison

Does CPU performance majorly affect overall Spark performance?

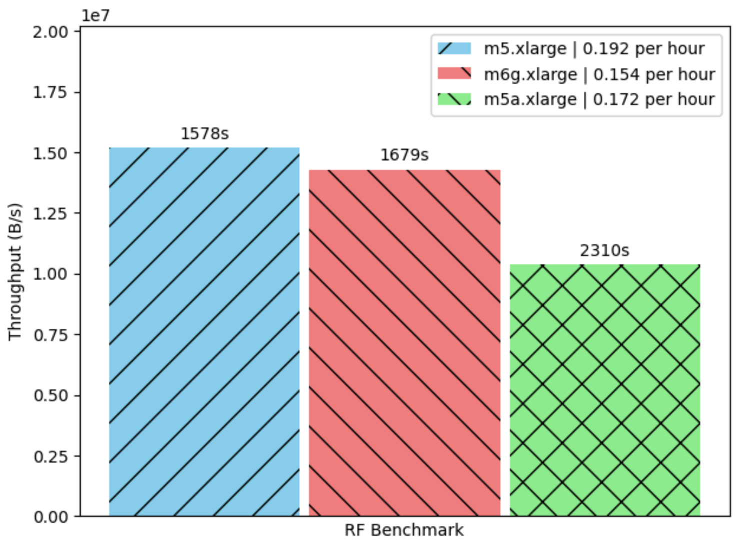
Q3: Experiment Setup

For this experiment we kept the cluster configuration static, only changing the instance type of Task Nodes allocated to the cluster.

We chose instance types with different CPU architectures but the same, comparable, CPU to RAM ratio (1-4).
For CISC-based CPU architecture, we picked x86-64 (ISA) CPUs from 2 different vendors, Intel and AMD. From Intel we chose the “common”/default `m5` performance Intel Xeon CPUs and from AMD we chose `m5a` EPYC 7000 series processors.
For RISC based CPU architecture, we picked Amazon’s ARM-based Graviton2 CPUs `m6g`. According to Amazon “AWS Graviton is a family of processors designed to deliver the best price-performance for your cloud workloads running in Amazon Elastic Compute”. Our experiment validates these claims.

We included results for the most computationally heavy workload, the Random Forest Classification. We also conducted the same experiments for the other workloads but, being majorly I/O driven workloads, we saw little difference in their performance that probably is not attributed to the type of CPU used.

Q3: Initial Results and Insights



In the Random Forest benchmark, we observed that Graviton2 m6g instances are only slightly slower (by **5.98%**) than Intel Xeon m5 cores, yet they offer a cost advantage, being **19.7%** cheaper to rent per hour. Additionally, our analysis shows that AMD Ryzen CPUs (m5a.xlarge) perform significantly worse in computation-heavy workloads, highlighting the efficiency and cost-effectiveness of the Graviton2 m6g instances despite their marginally lower performance compared to Intel Xeon m5 cores.

As stated above, the experiment clearly showcases that Graviton CPUs are indeed capable and cost-effective. We believe that researchers and organizations should choose such cores in their EMR clusters.

5. Conclusion

In conclusion, our findings indicate that maintaining a high number of executors while adjusting their cores and memory allocation accordingly results in optimal performance for the selected benchmarks. Also, simply increasing the number of worker nodes (Task+Cores) does not necessarily correlate with improved performance, highlighting the importance of data locality. Data locality is a main aspect to the performance of a spark cluster. Before you decide what is the optimal number of worker nodes with cores and memory you have to think about the data locality. Last but not least, different CPU architectures with same characteristics for cores and memory have also impacted the performance. There is also the trade-off between better performance and cost. Think twice when choosing CPUs for your cluster.