WHITE PAPER

arm

Spark on AWS Graviton2: Best Practices

Masoud Koleini
Principal Solutions Engineer

Contents

Introduction 3

Spark Concepts and Components 4

Environment Setup 4

Spark Tuning Best Practices 5

- Cluster Tuning 5
- Executor Number/Cores/Memory 6
- Big-Small Principal 7
- Yarn Parameter-Naming Convention 9
- Spark/HiBench Configuration Parameters 10
- Yarn Configuration Parameters 11
- Spark Configuration Tuning 11

Spark Resource Monitoring 12

Spark Machine Learning Library 12

Spark K-Means Resource Tuning 13

- Introduction to K-Means Clustering 13
- Cluster Configuration 13
- K-Means Benchmark Parameters 13
- Caching 16
- Shuffling in K-Means 17

Conclusion 17

References 18

Introduction

This white paper focuses on how to tune a Spark application to run on a cluster of instances. We define the concepts for the cluster/Spark parameters and explain how to configure them given a specific set of resources. We use K-Means machine learning algorithm as a case study to analyze and tune the parameters to achieve the required performance, while optimally using the available resources.

Graviton3 only offers compute-optimized instances (C7g) at the time of writing, so we use Graviton2-based clusters because more instance types (M6g, R6g, C6g, etc.) are available based on requirements.

Spark Concepts and Components

We are assuming that the reader already has a basic familiarity with Spark concepts and components. The reader also should have some experience with Spark coding and an understanding of resource usage and execution-time analysis through the Spark Web UI.

Environment Setup

This white paper uses the HiBench suite for Spark performance analysis. HiBench is designed for big-data analysis of Hadoop, Spark, and streaming data engines. It can run different patterns of workloads including micro benchmarks, such as sort, word count, and eDFSIO. It can also run SQL benchmarks, such as scan, join, and aggregation. Lastly, it can run machine learning benchmarks for K-Means clustering, gradient-boosted trees, and many more. The benchmarks can run on different data sizes from tiny to big data.

HiBench workloads run in two phases:

1. Store the datasets into an HDFS cluster (data preparation).
2. Run the benchmarks on the data that is stored.

We set up the cluster in AWS in the following way:

1. An instance to run Spark with a Yarn Hadoop cluster in pseudo-distributed mode. In this configuration, Hadoop daemons run in separate java processes. Such a configuration allows all the executors to run on a single machine which simulates a full cluster. This configuration also reduces the latency of shuffling the data between executors when moving to subsequent stages.

2. An HDFS cluster with a single replica on a separate instance and in the same placement group as the Spark cluster. The HDFS cluster is then configured for memory storage to increase read/write bandwidth compared to the relatively limited bandwidth of NVMe SSD storage.

Spark Tuning Best Practices

Different workloads may require different tunings based on the nature of the compute. For instance, data analysis problems may be compute intensive or memory intensive. An example of this is the K-Means clustering algorithm in machine learning, which is a compute-intensive algorithm, while word count is more memory intensive. For this white paper, we explore tuning parameters to run K-Means clustering in an efficient way on AWS instances.

We divide Spark tuning into two separate categories:

1. Cluster tuning
2. Spark configuration tuning

Cluster Tuning

Tuning at the cluster/workload level involves choosing runtime parameters for a Spark computation given a specific number of instances with fix resources (CPU cores and memory). For example, on HiBench, the following parameters can be set to run the workloads:

1. Executor cores in Yarn mode
2. Executor number in Yarn mode
3. Executor memory

4. Driver memory
5. Parallelism
6. Shuffle partition number

Assuming that there are W worker nodes running on the cluster, each worker node has M gigabytes of memory and C CPU (vCPU) cores. The following explains how to calculate the above parameters:

Executor Number/Cores/Memory

There are multiple ways to consider the number of executors:

1. One executor per instance and assigning all available resources (memory and CPU cores) to that executor.
2. Multiple executors per instance and splitting available resources between them.

In principle, both approaches work in a similar manner when total resources and parallelism (number of parallel tasks) are set the same. However, there are some practical differences that make using a higher number of executors more suitable:

1. With a lower number of executors, if an executor fails, a higher number of parallelized running tasks will be interrupted.
2. Smaller executors are easier to get scheduled/rescheduled when instances with lower resources become available.

The downside for using a higher number of executors is that the broadcast data and caching are replicated on each executor. Therefore, if there are E executors on a node, the broadcast data/cache are replicated E times on the same node.

Big-Small Principal

To reduce the negative effect of data shuffling between executors, try running the executors on a few large-sized instances, instead of many small-sized ones.

The following are suggestions on how to calculate the different Yarn/Spark parameters:

- Executor cores: One (heuristic) rule of thumb suggested by several resources is to allocate five CPU cores to each executor, which means each executor can run five tasks or more in parallel. This is configured by setting yarn.scheduler.maximum-allocation-vcores in hadoop/etc/hadoop/yarn-site.xml. It is possible to set the minimum number of cores per container if necessary.
- It is common to assign the same number of CPUs and memory to the Spark driver, which must be considered in calculations. In Yarn mode, the Spark driver runs as the Application Master (in Kubernetes clustering, it runs as a separate container with permissions to launch executor containers).

Calculating the rest is straightforward:

- **Number of executors: floor((N * C – N) / 5)**
  - In the above formula, each node should have at least one core available for the operating system and other running processes (subtracting total number of cores by N). This calculation assumes that executors do not run on the driver (application master) node.
    - The calculations are slightly different when running HiBench in pseudo-distributed single-node setup, since the resources allocated to the driver running on the same node must be considered.

— **Executor memory: (M / executor per instance) * 0.9**

- This is the amount of memory that is assigned to an executor on the node. The total memory allocated by Yarn to each executor is executor memory plus memory overhead. The memory overhead is the non-heap memory used for interned strings, VM, and other types of overheads. This value is 10% of the total allocated memory by default.

— **Parallelism:**

- Defines the number of running tasks in parallel. The minimum value for this parameter should be the number of executor cores available. Some resources recommend the value to be equal to the total number of executor cores (the same as the default value for Spark). Spark documentation suggests that each CPU core can handle two to three parallel tasks, so, the number can be set higher (for example, twice the total number of executor cores). The input RDD is split into the same number of partitions when returned by operations like join, reduceByKey, and parallelize (Spark creates one task per partition).
  - This parameter only applies to the computations over raw RDDs. It is ignored when dataframes are used.

— **Shuffle Partition Size**

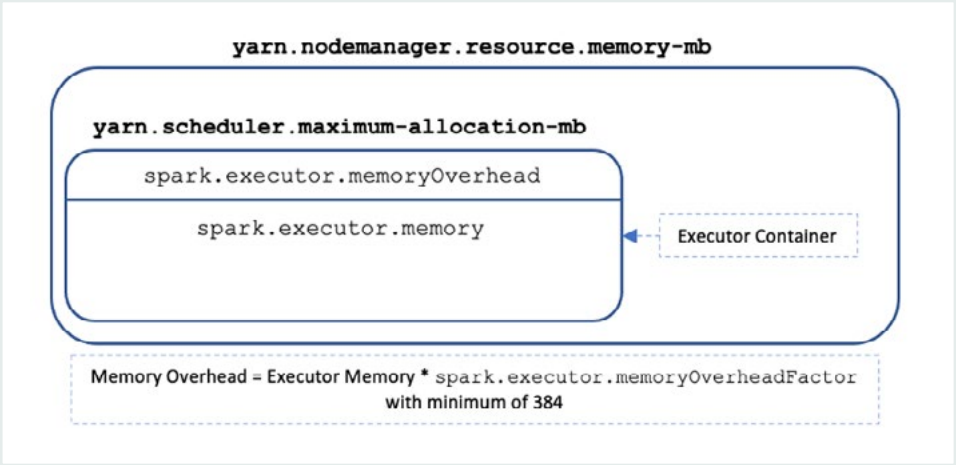
- This is like the parallelism parameter but applies to dataframes. The default value is 200, but users can set the value to a different number. One option is to set the number to twice the number of cores as we do for the parallelism. However, some take a different approach to set the partition size:
  - Best partition sizes for tasks are 128MB or 256MB. So, divide input data size into 128MB (or 256MB) to find the right number of partitions. Note that you must always set the number of partitions a factor of the CPU cores to keep the symmetry of the workload inside the cluster. The downside of this approach is that users must be aware of the input data size after deserialization. Also, the data size is different at each shuffle stage. So, another approach is to increase the number of partitions until the performance starts to drop.
  - When reading from bucketed HDFS files, the initial number of partitions (tasks) depends on the size of HDFS partitions (128MB by default). So, total number of partitions would be total data size divided by the default partition size. The shuffle partition size takes effect during the first data shuffle when moving to the next stage.

Yarn Parameter-Naming Convention

The parameters that start with yarn.nodemanager refer to node settings, while those that start with yarn.scheduler are for single containers (executors) running on the nodes.

Figure 1 shows how parameters define memory allocation for the cluster.

FIG. 1
Spark memory allocation parameters on Yarn cluster



Spark/HiBench Configuration Parameters

Spark parameters can be set inside the spark-defaults.conf file in the spark folder. For HiBench, spark parameters are set inside conf/spark.conf in the HiBench folder. The configuration parameters are as follows:

- Number of executors
  - spark.executor.instances/hibench.yarn.executor.num
- Executor memory
  - spark.executor.memory
- Executor memory overhead factor (default is 0.10)
  - spark.executor.memoryOverheadFactor
- Executor cores
  - spark.executor.cores/hibench.yarn.executor.cores
- Parallelism
  - spark.default.parallelism
- Shuffle partition size
  - spark.sql.shuffle.partitions

Similar parameters for the executors exist for the driver (application master).

Yarn Configuration Parameters

Yarn-specific parameters are defined in core-site.xml:

— yarn.nodemanager.resource.memory-mb
  - Memory available to Yarn executors on the (current) worker node

— yarn.scheduler.maximum-allocation-mb
  - Maximum memory allocated to executor container (including memory overhead)

— yarn.nodemanager.resource.cpu-vcores
  - Number of CPU cores available to Yarn executors on the (current) worker node

— yarn.scheduler.maximum-allocation-vcores
  - Maximum number of cores allocated to executor container

Spark Configuration Tuning

Spark can be fine-tuned depending on the application that is running. It is up to the developer to tune the memory, garbage collection, and serialization based on the code, data structures, and other parameters. See Tuning Spark document for more details.

Spark Resource Monitoring

Monitoring resources used for running a computation on a Spark cluster are highly important as they help find out if:

1. Enough resources are allocated to the job (or some tasks are failing),
2. Resources are more than required and a smaller cluster or instances can do the same job with almost the same processing time.

It is possible to analyze Spark performance and metrics using Web UI. For deep dive analysis, all the data during one or more runs can be collected and viewed using the Spark history server.

Other ways to collect metrics such as memory/disk/CPU from the machine directly (specifically when running pseudo-distributed cluster on a single machine) includes using tools like the system activity report (SAR).

Spark Machine Learning Library

The Spark machine learning library is called MLlib. It implements ML algorithms, data transformations, and pipelines in a distributed fashion. MLlib allows users to save the trained models and load them back in the prediction phase. The new library (also known as Spark ML) is based on the Spark Dataframe API and applies optimizations to the data pipeline. This article demonstrates K-Means clustering benchmarking as a case study for Spark resource allocation and tuning analysis.

Spark K-Means Resource Tuning

Introduction to K-Means Clustering

K-Means is an unsupervised clustering algorithm. Given K as the number of clusters, the algorithm first allocates K (semi)-random points (centroids) and iteratively refines their values until no further refinement is possible, or the maximum number of iterations is reached.

Spark implementations of K-Means run iterations over partitions independently and collects the results of each iteration back to the driver for centroid refinements. All iterations are over the same set of input data; so, it caches the data into the memory of each executor for faster computation.

Cluster Configuration

Our benchmarks run on m6gd.16xlarge EC2 instance, which has 64 vCPUs and 256GB of memory. Considering five cores per executor (scheduler maximum CPU allocation), the number of executors is set to 12, consuming 60 vCPUs. For K-Means, we assign only one vCPU for the driver (default) but set the driver memory to be the same as the executors. The memory per executor is set to 16GB (analysis of the metrics shows that this value can be increased). The Yarn scheduler and node manager parameters are set accordingly. For instance, the scheduler's maximum allocation of memory is defined to be 18022MB (1.1 * 16GB), and the node manager resources are set to 218264MB (12 executors + safeguard).

K-Means Benchmark Parameters

HiBench K-Means benchmarks default values are:

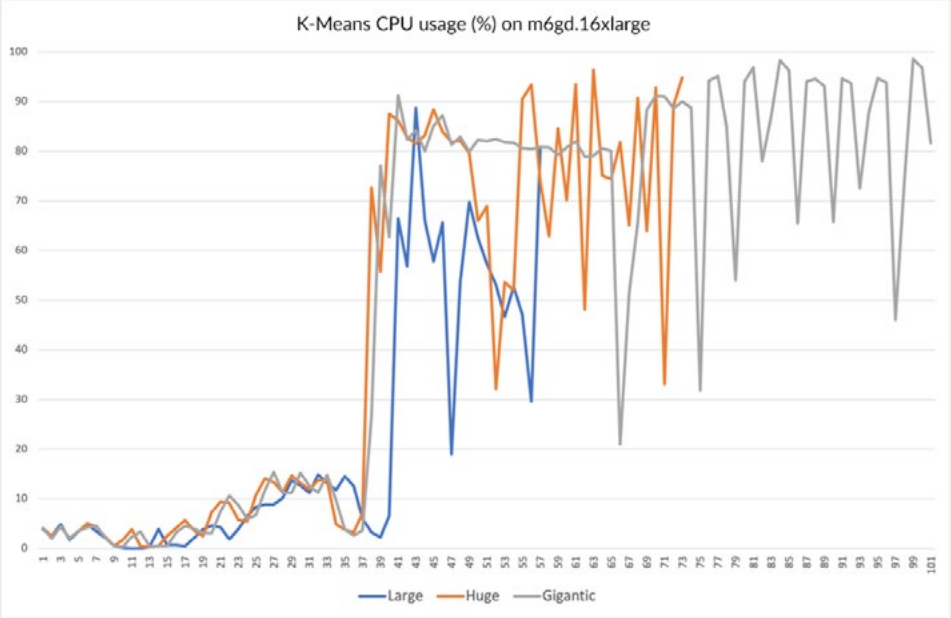
— Number of clusters: 5
— Maximum number of iterations: 5 (for up to the gigantic workload)

We run the K-Means algorithm on Hibench for three workloads of different sizes:

1. Large: 20m records, 3.7GB
2. Huge: 100m records, 18.5GB
3. Gigantic: 200m records, 37.1GB

K-Means is a compute-intensive algorithm. The following is the CPU usage of the K-Means algorithm running on large, huge, and gigantic data sizes of HiBench:

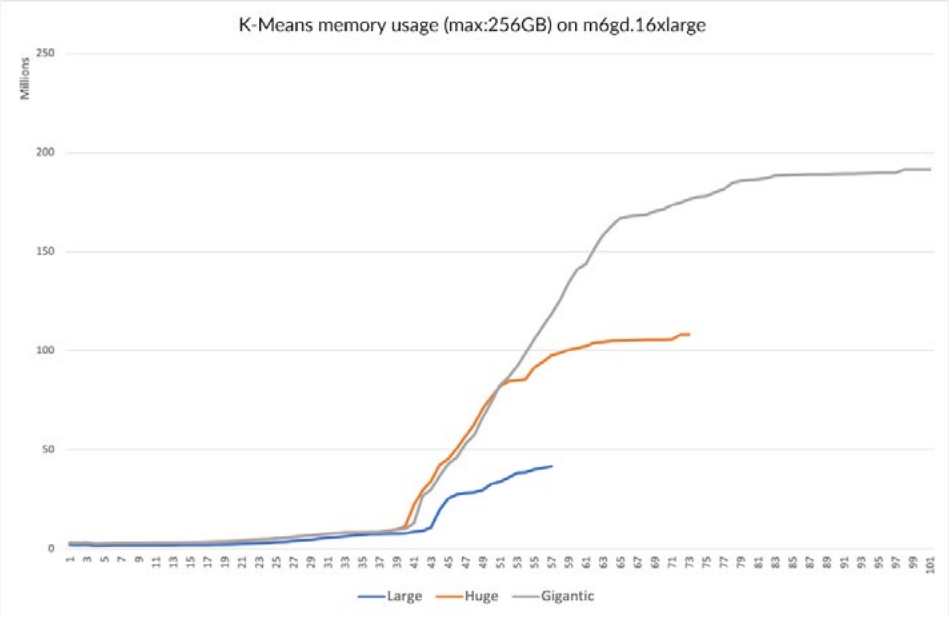
FIG. 2
CPU usage for large, huge, and gigantic workloads



The total CPU usage of all the executors is 93% based on the number of cores assigned per executor (60/64). Therefore, the gigantic workload uses most of the processing capacity.

Figure 3 shows the memory usage of all the three workloads.

FIG. 3
Memory usage for large, huge, and gigantic workloads



The memory usage for all the workloads is below the maximum amount set in the configuration. The two graphs (CPU and memory usage) show that one tuning possibility is to use compute-optimized instances to run the computation. These instances are cheaper than general purpose (M) and memory-optimized (R) instances of the same size.

It is also important to note that the total data size for the gigantic workload is only 37.1GB. Even when considering the memory used during the centroid computation, using close to 200GB of memory looks excessive for K-Means. The reason is the in-memory caching that HiBench K-Means benchmark enforces, which can change the way we tune the cluster to run the code.

Caching

As mentioned in previous sections, best practices suggest assigning five cores per executor, calculating the number of executors to use and assigning memory to each executor based on the available memory on the instance. However, in the case that the Spark program uses in-memory caching, all the caches replicate on all the executors. So, if you run E executors on the same instance, your cache consumes E times more memory compared to running a single executor.

The following chart compares gigantic workload memory usage for:

1. 12 executors, 5 cores, 16GB of memory per executor
2. 6 executors, 10 cores, 32GB of memory per executor

And all the executors running on the same instance (pseudo-distributed configuration).

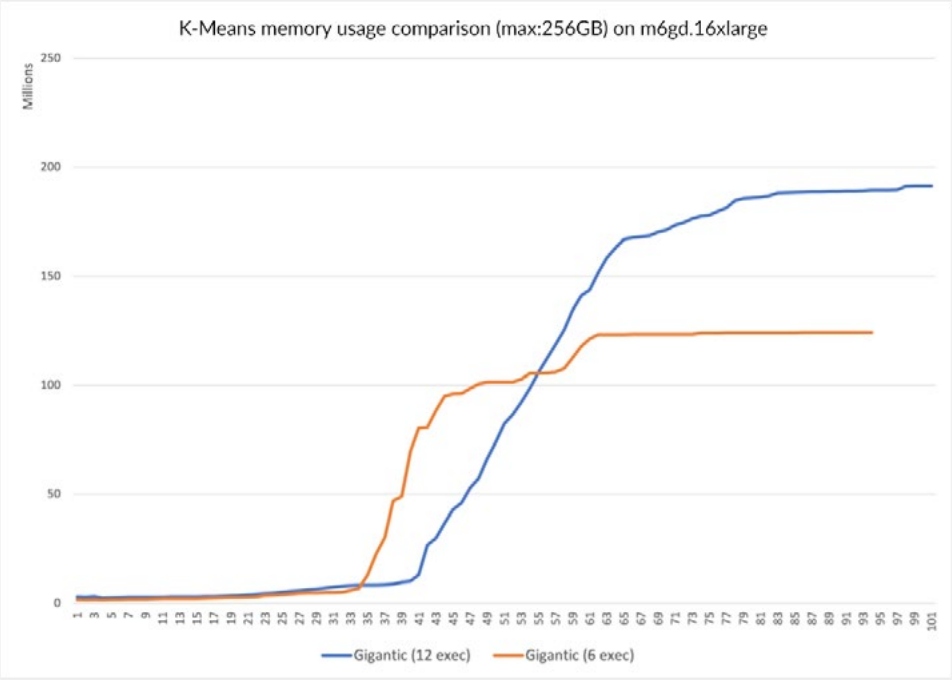


FIG. 4
Memory usage comparison for gigantic workloads for 12 and 6 executors running on the same machine

The second configuration clearly consumes less memory due to the smaller number of cache replications. So, when caching a large chunk of data, reducing the number of executors with bigger sizes can help decrease memory usage. Analysis shows that except for the initial phase for data pre-processing, processing times of K-Means stages do not differ.

Shuffling in K-Means

The implementation of K-Means only requires shuffling a small amount of data (which is an expensive operation in distributed computing). Different stages are separated by collecting centroid-based calculations over each partition into the driver for final centroid computation, which has a very small data size. Therefore, distributing the workload into a cluster of instances will not have a considerable impact on the performance.

Conclusion

Tuning Spark computations is application specific and depends on different parameters, such as data storage, caching, and shuffling. In cases like caching and storage, it is possible to use disk storage when memory could become a bottleneck (this might considerably degrade the performance on AWS since SSD read/write throughputs are throttled).

For applications like K-Means that are CPU intensive and do not involve much shuffling, most of the tuning is done through memory management. Using a smaller number of executors of bigger size, caching on both disk and memory based on availability, and using compute-optimized instances are options to consider for performance and cost optimizations.

References

[01] Tuning Spark

[02] HiBench Suite

[03] Hadoop: Setting up a Single Node Cluster

[04] Decoding Memory in Spark

arm