Optimization of Machine Learning on Apache Spark

Rohit Taneja(IBM), Rajaram B. Krishnamurthy(IBM), Gang Liu(IBM)
11400 Burnet Road, Austin, Texas, USA 78758

Abstract - *Wide adoption of Spark for running big data analytics jobs not only requires focus on the functional performance of applications, but also on their operational efficiency with optimal usage of hardware resources. Since Spark has a large number of tunables, a bottom up approach to finding the optimal runtime by varying Spark executors and Spark executor cores can create an explosion of tuning runs for a given application because of the multiplicative nature of possible configurations. Instead, we use a hybrid top-bottom approach by first characterizing the application and then custom tuning the Spark environment to further enhance performance.*

We have experimented extensively with Spark's tunables and share our tuning methodology by providing a detailed walk-through of an Alternating Least Squares Based Matrix Factorization application. Using our methodology, we have been able to improve runtimes by a factor of 2.22X. We have successfully applied this methodology to complex Spark workflows consisting of Spark SQL and ML Pipelines, and achieved substantial performance improvements. Our methodology has been applied to a variety of cluster architectures to validate our approach.

Keywords: Apache Spark, Machine learning, Performance evaluation and optimization, Matrix Factorization

Submission: Regular Research Paper

1 Introduction

Data sets are growing more rapidly than ever: every day more than 2.5 exabytes of data is created, while at the same time the world's technological capacity to store information has roughly doubled every 40 months since the 1980s[1]. To leverage the power of big data, scientists and engineers have been working on innovative solutions for high performance big data engines. Apache Spark[2] has quickly emerged as one of the most popular in-memory big data processing engines for being easy to use, fast and exhaustive with its built-in modules for streaming, SQL, machine learning and graph processing.

For a big data analytic application running on the Spark framework, the efficient use of underlying cluster resources is vital and often requires a lot of time to apply the necessary tuning in order to achieve the efficiency. Determining the optimal performance point influences total cost of ownership and performance per dollar of applications. In fact, without understanding the characteristics of the workload, efficient parallel execution on a distributed computing platform may not be achieved. To minimize execution times and to achieve performance benefits from parallelism in a Spark cluster environment, characterization in terms of application's resource usage is very crucial. This also involves the identification of critical sections and hardware bottlenecks in order to fine tune the application before deployment on a distributed computing platform.

The rest of the paper is organized as follows: In Section 2, we have discussed about the abstraction that Spark provides and understood the vital parameters in deploying workloads on Spark. In Section 3, we have evaluated the popular machine learning model: Alternating Least Squares (ALS) based Matrix Factorization (MF)[3], on a Spark cluster. Finally we have described the various optimization iterations that we performed on the MF application and conclude our paper in Section 4.

2 Background

2.1 Apache Spark

Apache Spark is usually launched on top of an existing Hadoop Cluster with Hadoop file system spanning worker nodes, while master node drives the work-flow and provide management services. Spark provides distributed computing through working on collection of objects called resilient distributed dataset (RDD). RDDs can be created with a file in the Hadoop file system or by transforming the existing RDD. This abstraction allows in-memory caching of data, but at the same time makes spark more sensitive to data locality. RDDs are kept as partitions spread across the worker nodes in a cluster, and a driver program is initiated to manage the user application. Worker nodes execute the tasks on these partitions in parallel. These set of tasks are grouped into spark stages and various stages are grouped together to be called as a Spark Job, which is the highest layer of abstraction in Spark framework.

For the optimization process, the understanding of key spark parameters is necessary:-

- Spark executor Instances: This parameter decides the number of executors per node. A worker node can have more than one executor, but for the purpose of this paper, we use 1 executor per worker.
- Spark executor cores: Number of threads or logical CPUs per executor on a worker node.
- Spark executor memory: The maximum heap size per executor on a worker node.

- Spark shuffle location and manager: The shuffle operation in Spark writes the 'map' output before performing the 'reduce' on the data. Depending upon the available memory, intermediate shuffle data may remain entirely in memory or get spilled to shuffle location, which is usually HDDs or SSDs. Selection of the shuffle manager based on the characterization of an application can also provide substantial gain in performance. There are three types of shuffle managers: Hash based, Sort based and Tungsten-sort.
- RDD persistence storage level: Caching of RDD partitions across nodes in a cluster can be done only in memory, or stored on disk if RDD does not fit in memory. Serialized java objects, where only one byte array per partition is created, are more space-efficient, but more CPU-intensive to read.

The latest change in the Spark execution engine, called as Project Tungsten[4] improves memory and CPU efficiency through its explicit memory management, binary processing and better cache aware computation. We have evaluated the impact of Project Tungsten's features on ALS MF application.

2.2 Sparkbench Suite

Tailored for Apache Spark, Sparkbench suite[5] has four major categories of applications: Machine learning, SQL query, Streaming applications and Graph computation. This comprehensive suite makes extensive use of the built-in modules of Spark and exposes different bottlenecks with different application characteristics.

For the work described in this paper, we have selected a Machine learning model: ALS based MF to be evaluated and optimized. MF is used in Recommendation systems as it provides model based collaborative filtering and can be configured to use feedback from users. The memory hungry nature of this application with heavy shuffle operations makes it an important and interesting benchmark to evaluate and optimize.

2.3 Spark Cluster environment

We have created Spark environment for our experiments on IBM Power Systems S812LC[6] server machines, which comprises of POWER8 processors[7]. The high thread density of 8 per CPU core enhances the parallelism provided by Spark compute model. 10-core POWER8 3.54 GHz processor module has 512 KB of L2 cache per core, 8 MB of L3 cache per core, and a system memory of 512GB. We have deployed Hadoop file system for data storage to take advantage of the efficient distribution of data across worker nodes in a cluster.

3 Performance evaluation and optimization

3.1 MF application

We have evaluated the ALS MF application with synthetic dataset generated with help of Sparkbench testing framework, which allows for the easy reproduction of our experiments. For the experiments described in this paper, Table 1 provides details about the number of rows and columns and size of dataset. The dataset resides in the Hadoop file system once the data generation phase finishes. The Cluster environment details are mentioned in Table 2.

Data generation parameters	**Value**
Rows in data matrix	62000
Columns in data matrix	62000
Data set size	100 GB

Table 1: Parameters used for data generation in MF application

Spark parameter	**Value for MF**
Master node	1
Worker nodes	6
Executors per Node	1
Executor cores	80 / 40 /24
Executor Memory	480 GB
Shuffle Location	HDDs
Input Storage	HDFS

Table 2: Spark environment details for application evaluation

3.2 MF work-flow

The run phase of the ALS MF model training consists of a series of jobs, which are further divided into stages that perform Spark actions like 'first', 'count', 'saveasTextfile' etc, and transformations like 'map', 'join', 'cogroup' etc. We have highlighted the jobs and the associated APIs in Table 4 to better understand the actions and work flow of the MF application. We have depicted the order in which jobs of the MF application execute in Figure 1, based on Job Ids of Table 4. The reason for more than one API call in some jobs is due to the different stages within a job – and each stage calling different APIs.

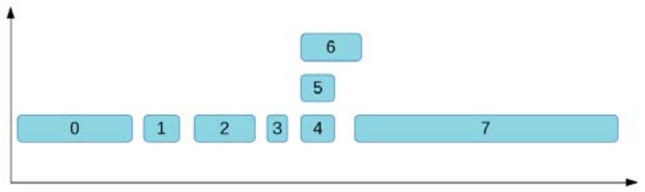

Figure 1. ALS MF jobs execution over time

Configuration	1	2	3	4	5	6	7	8	9	10	11
Spark executor cores	80	80	40	40	40	40	40	40	24	24	24
GC options	Default	Default	Default	ParallelGCthreads=40	ParallelGCthreads=40	ParallelGCthreads=40	ParallelGCthreads=40	ParallelGCthreads=40	ParallelGCthreads=24	ParallelGCthreads=24	Default
RDD compression	True	False	False	False	True	True	False	False	False	False	False
Storage level	memory_and_disk	memory_only	memory_only	memory_only	memory_and_disk_ser	memory_only_ser	memory_only	memory_only	memory_and_disk_ser	memory_and_disk_ser	memory_and_disk_ser
Partition numbers	1000	1000	1000	1000	1000	1000	800	1200	1000	1000	1000
Shuffle Manager	Sort based	Sort based	Sort based	Sort based	Sort based	Sort based	Sort based	Sort based	Sort based	Tungsten-sort	Tungsten-sort
Run-time (minutes)	40	34	26	24	20	25	26	27	21	19	18

Table 3: Various configurations tried in optimizing MF application on Spark

Job	Function	Description / API called
7	Mean at MFApp.java	AbstractJavaRDDLike.map MatrixFactorizationModel.predict JavaDoubleRDD.mean
6	Aggregate at MFModel.scala	MatrixFactorizationModel.predict MatrixFactorizationModel.countApproxDistinctUserProduct
5	First at MFModel.scala	ml.recommendation.ALS.computeFactors
4	First at MFModel.scala	ml.recommendation.ALS.computeFactors
3	Count at ALS.scala	ALS.train and ALS.intialize
2	Count at ALS.scala	ALS.train
1	Count at ALS.scala	ALS.train
0	Count at ALS.scala	ALS.train

Table 4. Description of jobs in MF application

3.3 Optimizing MF application

We have carried out several experiments to evaluate the performance impact with different Spark parameters. The Spark parameters that we have tuned for this work are listed in Table 3, in which each column is representative of an optimization iteration. We have tested configuration 1 and 2 from Table 3 for the purpose of characterizing the MF application, thus providing the maximum resources available in the cluster. For the purpose of a better understanding, we have plotted the CPU utilization over time for a node in Figure 2, and the memory footprint on a node in Figure 3. The low CPU utilization sections in the Figure 2 is representative of garbage collection (GC) pause times. We have confirmed that garbage collection happens during the low CPU activity sections by looking the history logs of stages and tasks.

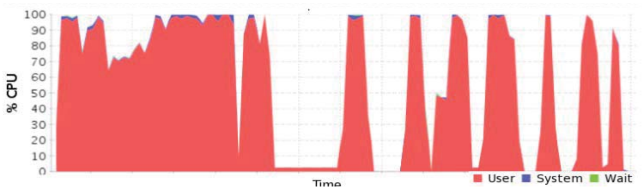

Figure 2. CPU utilization on a worker node (configuration 1 in Table 3)

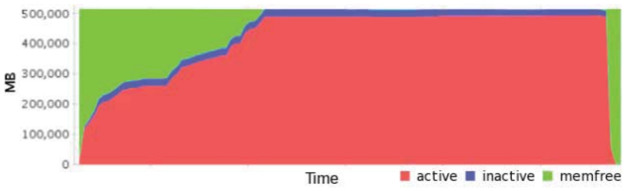

Figure 3. Memory utilization on a worker node (configuration 1 in Table 3)

In configurations 3, 4, 5 and 6 from Table 7, we selected the spark executor cores as 40 which improves the performance by providing more executor-memory per spark-executor-core and reducing contention in the last level cache with fewer spark executor cores. The difference in the above mentioned configurations is the storage level and the java garbage collection options. Performance with configuration 5 is twice that of configuration 1, which highlights the advantage by using same GC parallel threads as executor cores.

In the next set of experiments, we evaluated the performance impact by changing the number of partitions in the RDD. Configurations 7 and 8 show the partition numbers and the corresponding run times that we achieved. As the number of partitions decides the number of tasks that are spawned, careful selection of number of partitions to efficiently utilize CPU is vital to efficiently exploit the parallelism provided by the underlying compute resources. Configuration 7 and 8 did not yield any improvement over configuration 5, so we continued the optimization process using settings from Configuration 5.

In order to further reduce the last level cache contention, we configured Spark executor cores to 24 in configuration 9, 10 and 11, and it positively impacts the performance. Also, GC time is considerably reduced with 24 executor cores in configuration 9, but it still faces overhead from too many java objects created and garbage collection.

We evaluated the tungsten-sort shuffle manager from Project Tungsten in configurations 10 and 11. We disabled the java GC options in configuration 11, because tungsten-sort works directly against the binary data instead of java objects, thus alleviating the garbage collection overhead. This configuration gives us the best result with the experiments we performed for the scope of this paper. Using 24 out of 80 available threads along with tungsten-sort as the shuffle manager helps MF application to do more cache-aware computation, causes less last level cache contention and releases the pressure that comes from JVM object model.

The overall run time and memory footprint is greatly reduced as seen in Figure 5. We have highlighted the most time consuming stage in Figure 4, which is also the last stage of ALS MF application. We have compared the run time and GC time for this stage in Table 6, which shows the impact of using different values of spark executor cores and tungsten-sort on run-time and GC time.

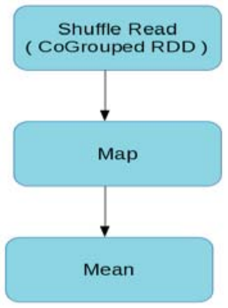

Figure 4. Directed Acyclic Graph of last stage in ALS MF

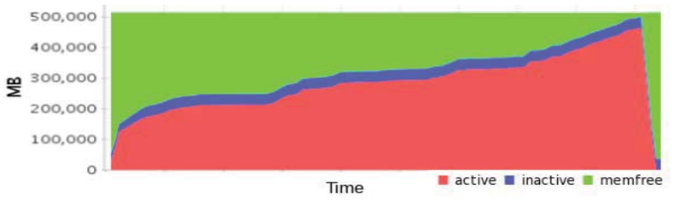

Figure 5. Memory footprint of configuration 11 from Table 3

Configuration	Run time of last stage	GC time of last stage
1	12 min	4.4 min
4	4.4 min	1.8 min
9	3.5 min	1.6 min
11	**47s**	**16s**

Table 6. Run time and GC time of Stage 68 for different configurations in Table 3

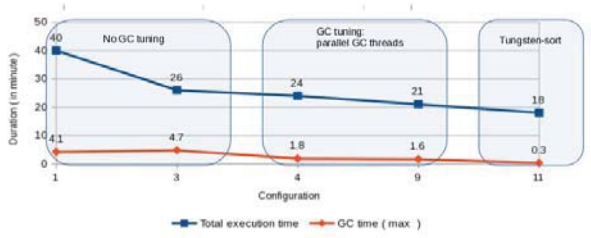

Figure 6. Run time and GC time for configurations in Table 3

4 Conclusion

The work presented in this paper explores the wide optimization space for the applications running on Spark. We use a hybrid top-bottom approach by searching the configuration space carefully after workload characterization through resource monitoring. This systematic methodology helps reduce the tuning iterations and achieving performance targets faster.

The shuffle heavy characteristic of ALS MF puts pressure on storage, network and also increases memory pressure from garbage collection due to JVM object model. Use of tungsten-sort feature alleviates these issues by doing explicit memory management and binary processing. We also observe that tungsten-sort does not bring advantage for all applications, and thus our hybrid “top-down” approach of workload characterization followed by searching configuration space becomes vital in order to apply custom optimization strategy.

Performance evaluation and optimization has resulted in a speed up of over 2x for the ALS MF application as well as other SQL and ML pipeline bases applications. The categorization of different applications and applying custom tuning for the different categories can not only improve operational efficiency, but also can be economical in the long run for an organization. We continue to evaluate more applications and improve our approach so as to reach best performance point with minimum optimization iterations for a Spark application.

References

[1] Hilbert, Martin; López, Priscila (2011). "The World's Technological Capacity to Store, Communicate, and Compute Information". Science 332 (6025): 60. doi:10.1126/science.1200970. PMID 21310967

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX NSDI, Berkeley, CA, USA, 2012.

[3] Haoming Li, Bangzheng He, Michael Lublin,Yonathan Perez (2015). "Matrix Completion via Alternating Least Square (ALS)". http://stanford.edu/~rezab/dao/notes/lec14.pdf

[4] Reynold Xin and Josh Rosen (2015). "Project Tungsten: Bringing Spark Closer to Bare Metal". https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[5] MinLi, Jian Tan, Yandong Wang, Li Zhang, Valentina Salapura (2015). "SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark". CF '15 Proceedings of the 12th ACM International Conference on Computing Frontiers, Article No. 53

[6] IBM Power System S812LC, https://www.ibm.com /marketplace/ cloud/big-data-infrastructure/us/en-us

[7] POWER8, https://en.wikipedia.org/wiki/POWER8