

SPARKBENCH: A Comprehensive Benchmarking Suite For In Memory Data Analytic Platform Spark

Min Li, Jian Tan, Yandong Wang, Li Zhang, Valentina Salapura
IBM TJ Watson Research Center
{minli,tanji,yandong,zhangli,salapura}@us.ibm.com

ABSTRACT

Spark has been increasingly adopted by industries in recent years for big data analysis by providing a fault tolerant, scalable and easy-to-use in memory abstraction. Moreover, the community has been actively developing a rich ecosystem around Spark, making it even more attractive. However, there is not yet a Spark specific benchmark existing in the literature to guide the development and cluster deployment of Spark to better fit resource demands of user applications. In this paper, we present SPARKBENCH, a Spark specific benchmarking suite, which includes a comprehensive set of applications. SPARKBENCH covers four main categories of applications, including machine learning, graph computation, SQL query and streaming applications. We also characterize the resource consumption, data flow and timing information of each application and evaluate the performance impact of a key configuration parameter to guide the design and optimization of Spark data analytic platform.

General Terms

Performance; Evaluation.

Keywords

Cloud computing; In memory Data Analytics; Spark; Benchmarking.

1. INTRODUCTION

The popularity of large-scale data processing frameworks, such as MapReduce [10], Giraph [8], Spark [32], has grown rapidly in recent years. This trend continues as the amount of data created by web search engines, social networks, business web click streams, and scientific research keep increasing at an unprecedented rate. Among the various types of big data programming models, Spark has attracted considerable attention because of the concept on Resilient Distributed Datasets (RDD). RDD is an easy-to-use, in-memory abstraction of data sets. It allows programmers to easily cache intermediate data into memory between computation tasks,

eliminate significant amount of disk IOs and thus reduce data processing time. Moreover, Spark community is actively building an ecosystem on top of Spark by supporting various types of computations [4]. For example, Spark currently supports four types of commonly used workloads, including machine learning, graph computation, SQL query and streaming applications.

While Spark has been evolving rapidly, the community lacks a comprehensive benchmarking suite specifically tailored for Spark. The purpose of such a suite is to help users to understand the trade-off between different system designs, guide the configuration optimization and cluster provisioning for Spark deployments. Existing big data benchmarks [1–3, 5–7, 9, 12, 13, 15, 23, 25, 27, 28, 31] are either designed for other frameworks such as Hadoop or Hive, or are too general to provide enough insights on Spark workload characteristics.

This paper aims to provide a Spark specific benchmarking suite to help developers and researchers to evaluate and analyze the performance of their systems and the optimization of workload configurations. To this end, we design SPARKBENCH by carefully choosing a comprehensive and representative set of workloads that cover different application types currently supported by Spark. Since various jobs can put pressure on different resources, a complete set of typical workloads can stress all computing resources of the whole cluster. That enables SPARKBENCH users to test extreme cases for a provisioned system. To help researchers and developers understand the Spark application behaviour and guide the Spark optimization and cluster provisioning, we characterize all the applications in SPARKBENCH through extensive experiments on synthetic data sets. We quantitatively analyze the workloads with respect to the CPU, memory, disk and network IO consumption during job execution, the data access patterns and task statistics. We then identify and evaluate a critical Spark configuration parameter that impacts the workload performance to help guide the Spark job parameter tuning.

Our main contributions are as follows:

- We propose SPARKBENCH, which covers a comprehensive set of workloads for Spark including four types of representative applications on synthetic data sets.
- We identify the distinct features of each application in terms of the resource consumption, the data flow and the communication pattern that can impact job execution time. We also explore the impact of a key Spark parameter, task parallelism, on Spark performance.

The rest of the paper is organized as follows. In Section 2, we discuss the difference between Spark and Hadoop and explain in more detail why Spark is increasingly adopted by industry. Then, Section 3 presents SPARKBENCH and describes the chosen workloads. We characterize all chosen workloads and summarize the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CF'15, May 18–21, 2015, Ischia, Italy.

Copyright 2015 ACM 978-1-4503-3358-0/15/05 ...\$15.00.

<http://dx.doi.org/10.1145/2742854.2747283>.

observed application characteristics in Section 4. In Section 5, we evaluate the performance impact of a key Spark system parameter. Finally, we differentiate related work from SPARKBENCH in Section 6 and discuss the future plan of our work in Section 7.

2. BACKGROUND

Apache Spark [4, 32] is a new programming model for processing big data which offers up to two orders of magnitude performance increase compared to MapReduce Hadoop. The significant performance increase comes from the way data is accessed and moved during processing, making repeated accesses to the same data much faster. Spark programming model is rapidly being adopted by many companies [22].

In Spark, there are four major differences from MapReduce Hadoop. Firstly, Spark provides an easy-to-use memory abstraction implemented as various resilient distributed datasets (RDDs). A RDD is a collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. RDDs allow programmers to store intermediate data into memory instead of storing it into disks, and retrieving it from disks, as it is implemented in the map and reduce processing phases of MapReduce. This approach avoids significant portion of slow disk IOs that incur in MapReduce framework. This is one of the fundamental reasons why Spark in many cases outperforms Hadoop. Big data Sort Benchmark [23] results comparing Spark and MapReduce demonstrated three times performance advantage for Spark [30].

Secondly, Spark supports directed acyclic graph (DAG) scheduling. In contrast to the simple programming model of map phase and reduce phase in MapReduce framework, a Spark job usually consists of multiple stages. That makes the resource provisioning of Spark jobs much more difficult than provisioning for MapReduce jobs. The stages within a Spark job can be scheduled simultaneously as long as they do not have any dependencies. Moreover, each stage can exhibit different resource consumption characteristics and more complicated shuffling patterns. In Spark, in addition to all-to-all shuffle, other communication patterns are supported such as union, group By Key, join with input co-partitioned, join with input not co-partitioned etc.

Thirdly, allowing data residing in memory through RDDs makes Spark more sensitive to the data locality compared with MapReduce framework since accessing data from remote memory can be 10 times more expensive in Spark while accessing data from remote disk can be 1 – 3 times more expensive in Hadoop.

Finally, unlike MapReduce, Spark supports a wider range of applications, from graphs processing and machine-learning libraries, enabling usage of one platform to meet data analytics needs instead of using different platforms through out the data processing process.

In this paper, we characterize the performance impact of the important system design factors of Spark on a representative set of workloads. A significant momentum in adoption of Spark programming model, and rapid development of new applications surface the need for a new Spark-specific benchmark. The benchmark can help both Spark developers and users to compare applications in a standardized application scenario.

3. THE SPARKBENCH BENCHMARKING SUITE

In this section, we present the SPARKBENCH benchmarking suite that incorporates a diverse set of applications to stress different cluster resources and the performance metrics reported by SPARKBENCH which enable SPARKBENCH users to quantitatively

Application Type	Workload
Machine Learning	Logistic Regression
	Support Vector Machine
	Matrix Factorization
Graph Computation	PageRank
	SVD++
	TriangleCount
SQL Queries	Hive
	RDDRelation
Streaming Application	Twitter
	PageView

Table 1: SPARKBENCH Workloads

compare between different Spark implementation, different cluster environments or configurations. It comprehensively covers the representative workloads currently supported by Spark. More specifically, as shown in Table 1, we classify the workloads into four categories including machine learning applications supported by ML-Lib of Spark, graph computation applications supported by GraphX of Spark, SQL applications supported by Hive on top of Spark as well as Spark native SQL service, and streaming applications supported by DStream of Spark.

Machine Learning: We choose three workloads including logistic regression [18], support vector machines (SVMs) [18] and matrix factorization (MF) [14]. They are widely used regression, classification and recommendation algorithms for machine learning applications. Logistic regression, as a machine learning classifier, can be used to predict continuous or categorical data. For example, it is used to predict whether a patient has a given cancer based on measured characteristics such as various blood test, family disease history, age and gender. The algorithm uses the stochastic gradient descent to train the classification model. The input data set is kept in the memory through RDD abstractions and the parameter vector is calculated, updated and broadcast in each iteration. SVMs train models by constructing a set of hyperplanes in a high, or even infinite, dimension space for classification. Compared with linear and logistic classification, SVMs can implicitly map inputs into a high dimensional feature space and efficiently conduct non-linear classifications. MF, typically used by recommendation systems, is a collaborative filtering technique that fills in the missing entries of a user-item association matrix. MF in Spark currently supports model based collaborative filtering and can be configured to use either explicit or implicit feedback from users.

Graph Computation: PageRank [24], SVD++ [20] and TriangleCount [19] are representative and popular graph computation algorithms. The intensive resource consumption of these three algorithms can also help explore Spark performance bottlenecks. PageRank is the first algorithm used by Google web search engine to rank pages by measuring the importance of website pages based on the number and quality of links to a page. SVD++ is a collaborative filtering model that takes both explicit and implicit feedback by users into account and improves the quality of recommendation. The algorithm updates bias terms and weight vectors of every edge during each iteration. TriangleCount is a fundamental graph analytics algorithm that counts the number of triangles in a graph. It is commonly used in complex real world graph applications, e.g. detecting spam and unveiling the hidden thematic structures in graphs of web pages and links. Due to the intensive computation load, it can be used to exercise the system performance boundary. Both machine learning and graph computation algorithms consist of various number of stages depending on the number of iterations each

Workload	Input	Shuffle R/W	Stages/Tasks
Logistic Regression	51 G	0 G/0 G	10/4399
Support Vector Machine	48 G	34 G/32 G	17/7842
Matrix Factorization	275 G	83 G/92 G	865/39802
PageRank	5 G	17 G/20 G	13/5224
SVD++	51 M	46 G/29 G	15/12000
TriangleCount	113 M	26 G/29 G	8/6400
Hive	7.5 G	10.2 G/11.3 G	12/9805
RDD Relation	7.5 G	2.0 G/2.0 G	12/9805
Twitter	0	4 M/4 M	3451/8109
PageView	0	78 M/86 M	475/22532

Table 2: Data Access Pattern of SparkBench Workloads

algorithm performs. Spark caches the data in RDDs to avoid disk IOs. However, configuring the right memory size for RDDs is not an easy task and the optimal values vary with applications.

SQL Engine: We have written two SQL query applications to query the E-commerce data set [27] which includes two tables that record orders and order-product information for an E-commerce website. The two SQL query workloads uses Hive on Spark and the Spark native SQL query support accordingly. We perform three common actions that are widely used operations with SQL language, namely select, aggregate and join [25]. More specifically, we select orders that have sale price larger than a certain amount. Then, we aggregate, count the number of orders that happened within a certain duration, and join the two tables with the same order ID to exercise the system performance of Spark in terms of SQL query support.

Streaming Applications: Streaming is a major representative use case that differs Spark from Hadoop framework. We choose two streaming applications, Twitter popular tag and PageView. Twitter popular tag retrieves data from the Twitter website through a Twitter Java library, Twitter4j [26], and calculates the most popular tags every 60 seconds. PageView is a streaming application that receives synthetically generated user clicks and counts various statistics such as active user counts and the page view counts every 60 seconds.

In addition to the carefully chosen ten workloads, SPARKBENCH defines a number of metrics facilitating users to compare between various Spark optimizations, configurations and cluster provisioning options. SPARKBENCH currently reports *job execution time* (seconds) measuring the job execution time of each workload, *data process rate* (MB/second) defined as input data size divided by job execution time. *Job execution time* is the most important performance indicator of Spark system. Any Spark optimization should lower down the average job execution time of Spark workloads in order to claim performance improvement. *Data process rate* reflects the data processing capability of Spark system. In the future, SPARKBENCH plans to report additional metrics such as the shuffle data size, input / output data size, average resource consumption enabling users to have in-depth understanding of workloads.

4. APPLICATION CHARACTERIZATION

In this section, we characterize the applications with respect to the data access pattern, job execution time, and resource consumption using SPARKBENCH. We conduct our experiments on a 11-node VM cluster hosted on one of SoftLayer [17] data centers. Each node on our 11-node cluster has 4 VCPU cores, 8 G memory, two locally attached 100 G virtual disk, connected through 1 Gbps network. One node is configured to work as the master node and the others work as slaves for both HDFS and Spark. One of the two

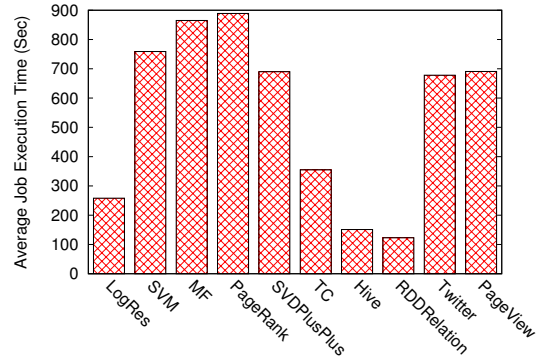


Figure 1: Workload execution time. LogRes denotes logistic regression.

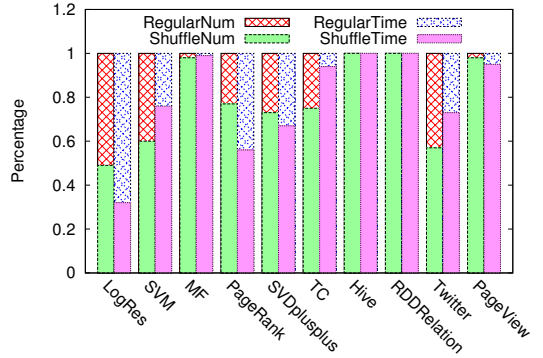


Figure 2: The ratio of execution time of shuffle tasks to regular tasks.

virtual disks is only used by HDFS and the other is used for spark local temporary files. Without mentioning explicitly, we configure each slave node to have one executor, with each executor having 6 G memory and leaving other memory space for the OS caches and data node of HDFS. Moreover, for the iterative machine learning and graph computation algorithms, we set the iteration limit to three in order for the experiments to finish in a reasonable amount of time without sacrificing the accuracy of application characterization. Except Hive and RDDRelation SQL query applications, we generate the input data sets for chosen workloads using built-in Spark data generator. We run each experiment for four times and report the average value for job execution time of each workload. Although we ran the experiments on a virtual machine cluster, we observed that the variation of results such as job execution time was usually within 5%.

Table 2 shows the data access patterns of all applications with respect to the total size of input data, shuffle write and read data and the number of stages and tasks. Note that the size of input data counts for the amount of data read by all the stages from HDFS thus it can be much larger than the size of the input data set of each workload. Figure 1 illustrates the average execution time of all workloads. We use synthetic data generator to control the size of the input data sets and finish each workload within 2 to 15 minutes. The average spark workload execution time is 549 seconds (9.1 minutes) across all the workloads.

Spark has two task types, namely the *ShuffleMapTask* and *ResultTask*. ShuffleMapTasks are generated by Spark DAG scheduler whenever the shuffle dependencies between stages are detected. Shuffle dependencies are established when operators such

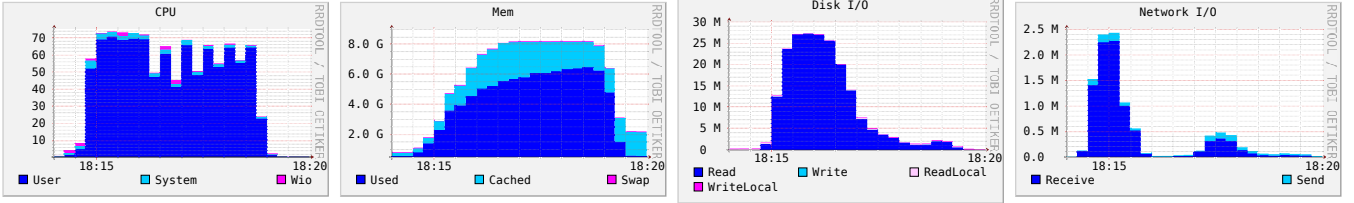


Figure 3: Resource Consumption of LogisticRegression.

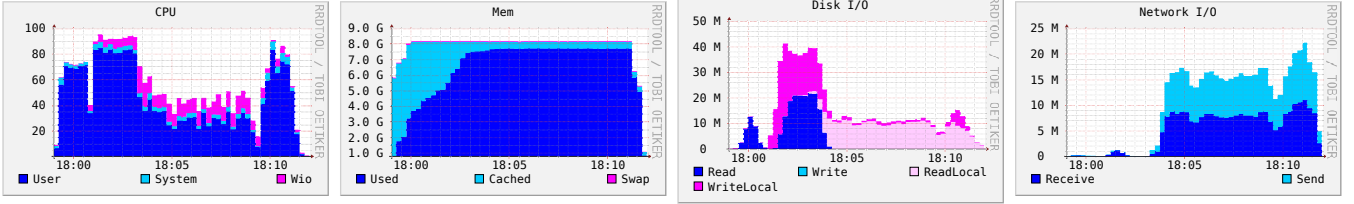


Figure 4: Resource Consumption of SVM.

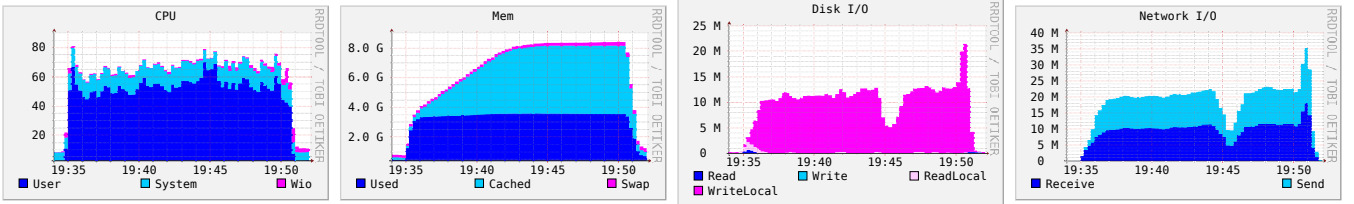


Figure 5: Resource Consumption of MatrixFactorization.

as repartition, combineByKey, GroupByKey are used. A shuffle operation in Spark is expensive since it usually involves data partitioning, data serialization and deserialization, data compression and disk IOs. Understanding the percentage of time the applications spend on shuffling helps to understand the application behaviour. ResultTask is similar to reduce task of the MapReduce framework, but it does not perform any shuffling and only outputs data to storage if there is any. Figure 2 shows the ratio of ShuffleMapTasks to the ResultTasks in terms of the number of tasks and the execution time. We observe that except logistic regression, all other workloads spend more than 50% of the total execution time for ShuffleMapTasks. Especially for MF, Hive, RDDRelation and PageView, nearly 100% of the tasks are MapShuffleTasks. This is because the operations of selection, aggregation and join of Hive and RDDRelation all incur the shuffle dependencies between stages. MF spend 99% of execution time computing user feature vectors by first joining products with out-links, generating messages to each destination user block, then grouping the messages by destination and joining them with in-link information. PageView summarizes and reports user statistics through “groupByKey” RDD operations thus producing 98% MapShuffleTasks.

Figure 3,4,5, 6,7, 8,9, 10, 11,12 illustrate the resource consumption of each application in terms of CPU, memory, disk and network IO utilization. For CPU resource consumption, we depict the CPU utilization of Spark tasks, system processes and CPU usage of waiting IOs (Wio). For memory utilization, we demonstrate the memory usage of Spark tasks, OS buffer cache and page cache and the memory size that has been swapped. Note that the disk IO includes the amount of data accesses to and from HDFS denoted by “Read”, “Write” legends while the amount of disk IOs generated by MapShuffleTasks are denoted by “ReadLocal” and “WriteLocal” legends. We also record the send and receive network IO for each slave node. All the number are calculated as the average of all ten worker nodes.

As shown in Figure 3 and Table 2, logistic regression consists of 10 stages with 4399 tasks, inputs a total amount of 51 G data, and shuffles 4 G read data and 0 G write data. It trains a classification model for the input data set using stochastic gradient descent. This job has an average of 63% CPU utilization and an average of 5.2 G memory usage. It also has a high disk IO utilization at the beginning of the workload execution and relatively low network utilization since only the parameter vectors are been shuffled during the iteration.

SVM possesses 17 stages with 7842 tasks, reads 48 G data from HDFS, and incurs 34 G shuffle reads and 32 G shuffle writes. SVM maintains a bimodal resource usage pattern, an average memory utilization of 7.5 G during the execution and high Disk IOs at the beginning and high network IO accesses at the end at the end of execution.

Figure 5 depicts the resource usage of MF workload. MF consists of 865 stages with a total of 39802 tasks. It reads 275 G data from HDFS and shuffles a large amount of data (83 G shuffle read and 92 G of shuffle write). The significant portion of shuffle IOs can also be observed from the high disk and network IO utilization of Figure 5. The memory usage stays at 2.7 G during the job execution. The SPARKBENCH user can use this workload to exercise the shuffle performance of Spark framework. Although, this workload consumes high CPU memory and IO utilization, the utilization can be further increased by increasing the parallelism of the tasks until one of the resources is saturated.

PageRank incurs 5 G HDFS IOs, 17 G shuffle reads and 20 G shuffle writes. It involves 13 stages with a total of 5224 tasks. It also achieves an average of 30% CPU utilization despite that there are some CPU peaks through out the job execution. While PageRank demands fewer CPU and IO resources, it saturates the memory resource during the execution. The stable usage of memory is because PageRank caches the vertices and edges in the memory, serving as the input for each algorithm iteration.

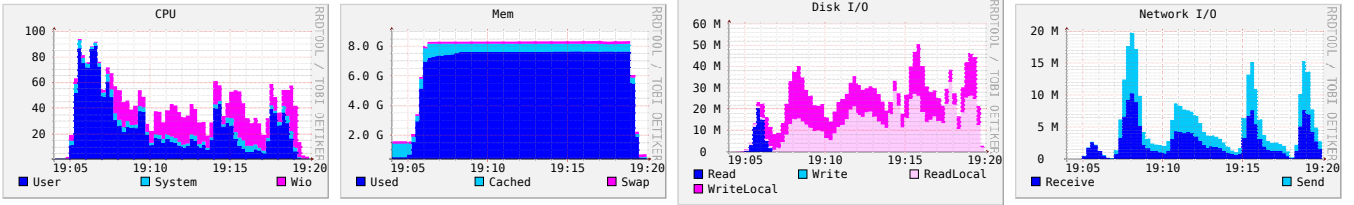


Figure 6: Resource Consumption of PageRank.

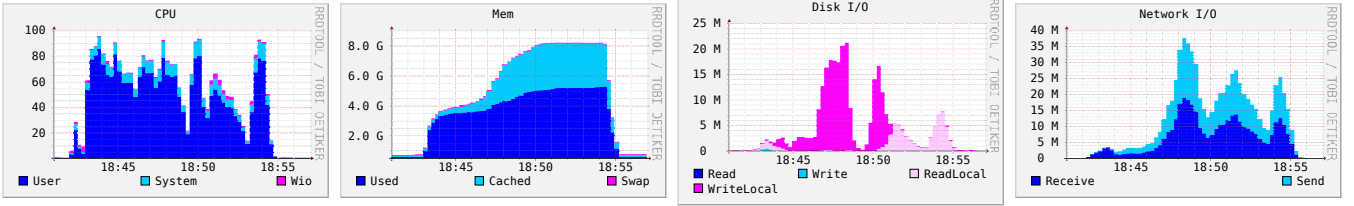


Figure 7: Resource Consumption of SVDPlusPlus.

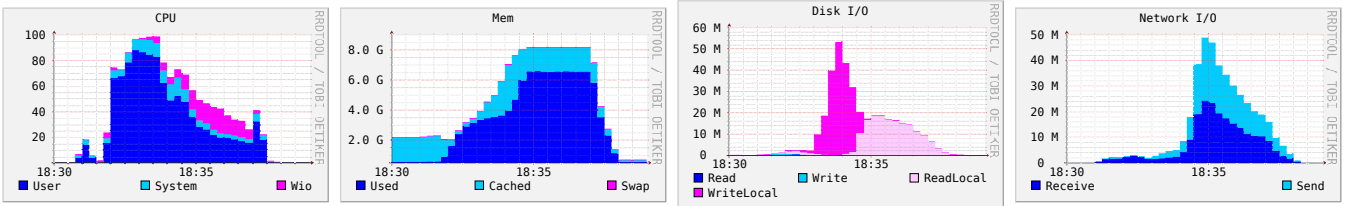


Figure 8: Resource Consumption of TriangleCount.

As illustrated in Figure 7, SVD++ contains 15 stages with 12000 tasks, reads 51 M, shuffle reads 46 G and shuffle writes 29 G of data. IO utilization exhibits three peaks at run time while the memory usage steadily and slowly increases.

TriangleCount includes 8 stages with 6400 tasks, reads 113 M HDFS data, shuffle-reads 26 G and shuffle-writes 29 G data, as listed in Table 2. TriangleCount has bursty resource consumption behaviour. It reaches 100% CPU utilization in the beginning then drops down later. Disk and network IOs show peaks in the middle of workflow execution.

Hive and RDDRelation workloads have similar resource consumption patterns. As show in Figure 9,10 and Table 2, both workloads comprise 12 stages with 9805 tasks and both consume 7.5 G input data. They have 10.2 G, 2 G shuffle read and 11.3 G, 2 G shuffle write data respectively. They hit nearly 100% utilization at the end of job execution since this is a join operation of two data tables. Spark computes the match of the two order IDs in the two tables and counts the number of the joined entries. The selection and aggregation operations use less CPU resources, but instead take up all the available memory.

The two streaming applications demonstrate similar resource usage patterns in Figure 11,12 as well as Table 2. We run both workloads for around 10 minutes and collect the statistics. Note that the two applications read zero data from HDFS since the Twitter workload retrieves input data through Twitter4J while the PageView workload receives data from a PageView Generator. The number of stages and tasks increases as the workloads keep running. However, majority of the tasks are light weight consuming less resources than other types of workloads. Except memory, both applications have light CPU, Disk, Network IO consumption. The memory usage is steadily increased at run time. Below are the differences between the two workloads. PageView consumes more memory as the workload keeps running and there are some local disk IOs at the beginning of job execution. Based on the resource consumption

graph, we can scale down the cluster without sacrificing the performance because none of the resources is saturated.

Based on the above workload characterization, we derive some interesting observations across all the workloads. Memory becomes a precious resource for Spark framework due to the extensive use of the RDD in memory abstraction. It is also important for Spark to optimize the shuffle operations since the majority of workloads spend more than 50% of execution time for ShuffleMapTasks. Although the resource consumption and data access patterns of the workloads can vary with respect to cluster setup and Spark platform parameter configuration, program parameter configuration, the input data content and data size, the provided characterization can still be valuable in that it reflects the resource demand of a given workload is usually stable and the exhibited resource consumption can be inferred based on our provided profiles.

5. IMPACT OF PARAMETER CONFIGURATION

We identify three key Spark system parameters that impacts the application resource utilization and workload execution time. In this section, we present the quantitative result of the impact of the number of concurrently executed tasks on application performance. We choose three applications, logistic regression, PageRank and Hive from SPARKBENCH each belongs to a different application type. We also plan to quantitatively study the impact of RDD cache size, and the executor configurations in the near future.

We configure each VM to possess one executor, each with 6 G memory, varying the number of allocated virtual CPU cores from 1 to 6. Since we only have 4 virtual CPU cores, assigning 6 virtual CPU cores per executor means the CPU resources are over provisioned and CPU intensive Spark workloads might compete CPU resources. Controlling the assignment of number of virtual cores enables us to control the number of concurrent tasks running simultaneously. Spark framework launches tasks whenever there are

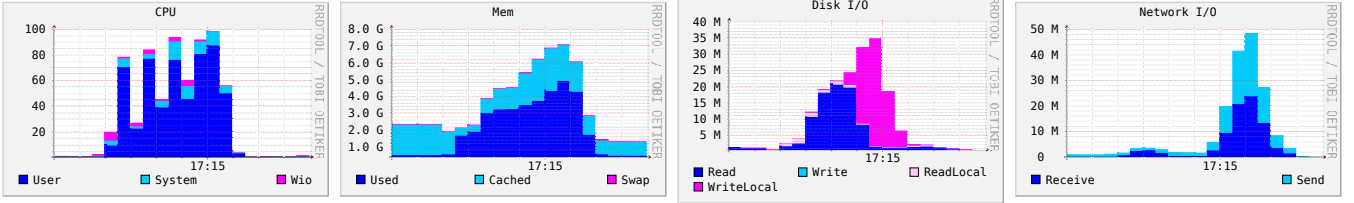


Figure 9: Resource Consumption of HiveSQL.

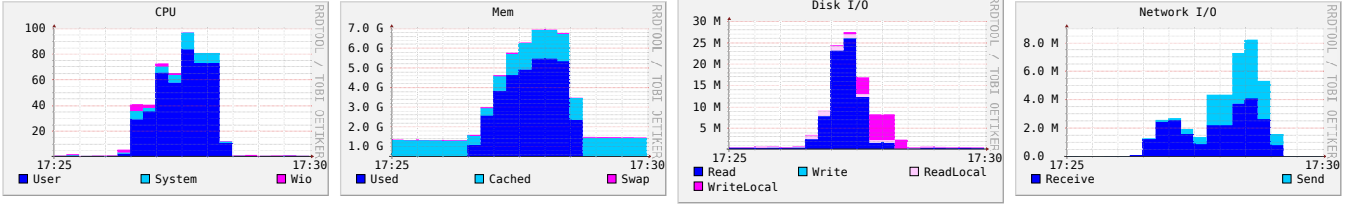


Figure 10: Resource Consumption of RDDRelation.

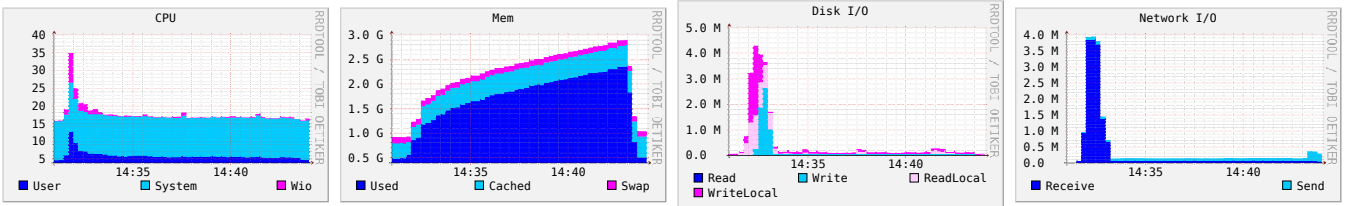


Figure 11: Resource Consumption of TwitterStreaming.

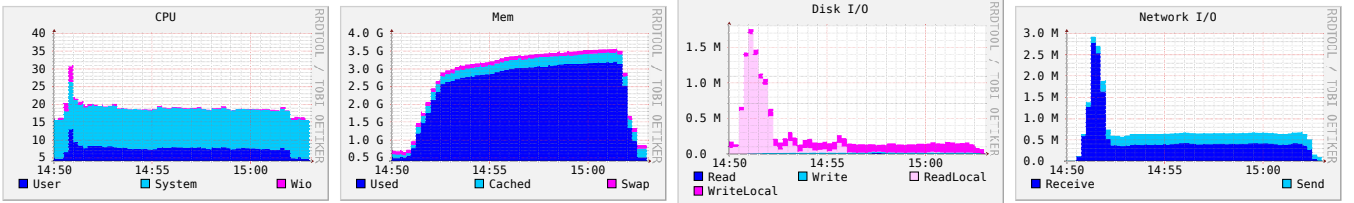


Figure 12: Resource Consumption of PageViewStreaming.

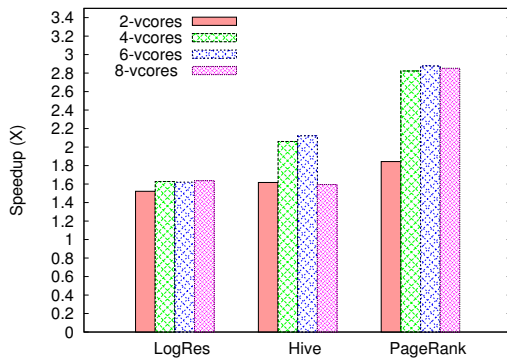


Figure 13: The impact of task parallelism on average job execution time. Y-axis shows the speed-up normalized to the job execution time with task parallelism of 1-vcore.

free virtual CPU cores and by default each task consumes one CPU core. We ran this experiment for four times and report the average results. Figure 13 illustrates the average job execution time under different number of virtual CPU cores per executor. The job execution time decreases as the number of virtual CPU cores increases up to 4 cores since Spark can be able to launch more tasks con-

currently. We also observe a corresponding rise on resource consumption as the number of concurrent tasks increases. However, over-committing CPU resources might not necessarily lead to performance improvement. When the CPU is a bottleneck resources, over-committing CPU actually lead to performance degradation.

6. RELATED WORK

Developing benchmarks for big data frameworks plays an important role in understanding the system performance, validating design choices, and optimizing parameter configurations [5–7,15,25]. The work on benchmarks can be classified into two categories: framework specific benchmarks and multiple-framework benchmarks that include workloads for various frameworks. Framework specific benchmarks refer to benchmarks that are tailored to a particular framework and investigates the characterization of typical applications using this framework.

SparkBench belongs to the first category. Other Hadoop specific benchmarks, which also belong to first category, include HiBench [15], GridMix [5], PigMix [6], Hive performance benchmarks [25], LinkBench [7] etc. These benchmarks cannot be used directly on Spark without any modifications. On the other hand, SPARKBENCH provides a diverse and representative set of Spark workloads that help guide the optimization and deployment of Spark.

Multiple-framework benchmarks usually provide one to multiple workloads that can run on different software stacks. Listed below are some typical benchmarks, including Gray sort, cloud sort, terabyte sort etc [23], BigDataBench [27,28], AMP Lab Big Data Benchmark [3], CloudSuite [12], YCSB [9], BigBench [13]. While these benchmarks aim to provide comparisons between different data analytic frameworks SPARKBENCH targets to deliver in-depth insights of Spark performance implications.

7. DISCUSSION AND FUTURE WORK

In this paper, we present SPARKBENCH, a Spark specific benchmarking suite that comprises diverse types of workloads using synthetic data sets. We then characterize all the workloads to shed lights on various application behaviours running on Spark. We observe that memory becomes a precious resource due to the use of RDD abstraction. In addition, all chosen machine learning workloads have intensive resource demand for CPU. While resource demand of graph computation workloads varies, the streaming applications have relatively light resource demand. In addition, we study the impact of a key configuration parameter, the parallelism of workloads, on application performance. Our evaluation shows that while increasing task parallelism to fully leverage CPU resources can reduce the job execution time, over-committing CPU resources can lead to CPU contention and adversely impact the execution time. We have open sourced SPARKBENCH, which is publicly available [21].

We are currently working on characterizing SPARKBENCH applications using real world data sets such as Wikipedia data set [29] and Facebook social graph [11]. Although synthetic data sets in some extent help to understand the resource demand of each application, they might not accurately reflect the actual resource demand and usage patterns in real world. The workload characteristics can vary with input data set content. Using real world data sets allows SPARKBENCH to provide more useful and meaningful evaluation results since the veracity of 4V properties of big data [16] is being preserved.

Moreover, we plan to use SPARKBENCH workloads to study more performance related Spark configuration parameters. For example, we plan to quantitatively study how different memory sizes used by RDDs impact performance of workloads by changing “memoryFraction” ratio and how different executor configurations impact performance by changing resource configuration of Spark executors.

8. ACKNOWLEDGMENT

We thank David P. Grove, Joshua Milthorpe and the reviewers for their feedback and appreciate their support for our work.

9. REFERENCES

- [1] TPC-DS. <http://www.tpc.org/tpcds/>, 2014.
- [2] TPC-H. <http://www.tpc.org/tpch/>, 2014.
- [3] AMPLab. Big Data Benchmark. <https://amplab.cs.berkeley.edu/benchmark/>, 2013.
- [4] Apache. Spark. <http://spark.apache.org/>.
- [5] Apache. GridMix. <http://hadoop.apache.org/docs/r1.2.1/gridmix.html>, 2013.
- [6] Apache. PigMix. <https://cwiki.apache.org/confluence/display/PIG/PigMix>, 2013.
- [7] T. G. Armstrong, V. Ponnemanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD*, pages 1185–1196.
- [8] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.
- [9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM SOCC*, pages 143–154, 2010.
- [10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
- [11] FaceBook. Social Network Graph. <http://snap.stanford.edu/data/egonets-Facebook.html>.
- [12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th ACM ASPLOS*, pages 37–48, New York, NY, USA, 2012.
- [13] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD*, pages 1197–1208, 2013.
- [14] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *Proceedings of the Eighth IEEE International Conference on Data Mining*, pages 263–272. IEEE, 2008.
- [15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *26th IEEE ICDEW*, pages 41–51, March 2010.
- [16] IBM. Big Data and Analytics Hub. <http://www.ibmbigdatahub.com/infographic/four-vs-big-data>.
- [17] IBM. SoftLayer. <http://www.softlayer.com/>.
- [18] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*. Springer, 2013.
- [19] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012.
- [20] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD*, pages 426–434, 2008.
- [21] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. SparkBench: A Comprehensive Spark Benchmarking Suite. <https://bitbucket.org/lm0926/sparkbench>.
- [22] S. Neumann. Spark vs. Hadoop MapReduce. <https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/>.
- [23] C. Nyberg, M. Shah, and N. Govindaraju. Sort Benchmark. <http://sortbenchmark.org/>, 2014.
- [24] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.
- [25] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD*, pages 165–178, New York, NY, USA.
- [26] Twitter. Twitter4j: a Java Library for the Twitter API. <http://twitter4j.org>.

- [27] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench.
<http://prof.ict.ac.cn/BigDataBench/>, 2014.
- [28] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *IEEE 20th HPCA*, pages 488–499, Feb 2014.
- [29] Wikipedia. Data Dumps.
<http://dumps.wikimedia.org/enwiki/>.
- [30] A. Woodie. Spark Smashes MapReduce in Big Data Benchmark.
<http://www.datanami.com/2014/10/10/spark-smashes-mapreduce-big-data-benchmark/>.
- [31] W. Xiong, Z. Yu, Z. Bei, J. Zhao, F. Zhang, Y. Zou, X. Bai, Y. Li, and C. Xu. A characterization of big data benchmarks. In *IEEE International Conference on Big Data*, pages 118–125, Oct 2013.
- [32] 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.