Performance Tuning and Evaluation of Iterative Algorithms in Spark

Janani Gururam
Department of Computer Science
University of Maryland
College Park, MD 20742
janani@cs.umd.edu

Abstract. Spark is a widely used distributed, open-source framework for machine learning, relational queries, graph analytics and stream processing. This paper discusses key features of Spark that differentiate it from other MapReduce frameworks, focusing specifically on how iterative algorithms are implemented and optimized. We then analyze the impact of Resilient Distributed Dataset (RDD) caching, Serialization, Disk I/O, and other parameter considerations to improve performance in Spark. Finally, we discuss benchmarking tools currently available to Spark users. While there are several general purpose tools available, we do note that only a few Spark-specific solutions exist, and do not fully bridge the gap between workload characterization and performance implications.

1 Introduction

Spark is an open-source, in-memory distributed framework for big data processing. The core engine of Spark underlays several modules for specialized analytical computation, including MLlib for machine learning applications, GraphX for graph analytics, SparkSQL for queries on structured, relational data, and Spark Streaming [25]. Spark provides several APIs for languages like Java, Scala, R, and Python.

The base data structure in Spark is a Resilient Distributed Dataset(RDD), an immutable, distributed collection of records. There are two types of operations in Spark: *transformations*, which perform operations on an input RDD to produce one or more new RDDs, and *actions*, which launch the series of transformations to return the final result RDD. Operations in Spark are lazily evaluated.

A logical execution plan, or lineage graph, provides the series of transformations to be performed on an input data set once an action has been called. This plan determines the dependencies between the RDDs in the set of transformations. The DAGScheduler then converts this logical plan into a physical execution plan: a directed acyclic graph of stages. Each stage is a set of tasks based on partitions of the input data which can be run in parallel. For debugging, the logical execution plan is readily available to the user with the debug package (queryExecution.logical) in Spark [14]. Narrow dependencies in the lineage graph like *map* allow for pipelined execution of tasks, whereas wide dependencies like *join* indicate stage boundaries [32] in the graph. The DAGScheduler, in addition to scheduling tasks on each worker node, keeps track of the RDDs that have been materialized in order to avoid recomputing results and improve performance. At run time, parallel execution tasks (stages) are processed in batches: all partitions of a particular stage are executed before the downstream dependent stages are started.

Spark is used for big data applications, including analysis of mobile data usage patterns, ETL and log processing, predictive analytics, and business intelligence [1]. Spark's wide adoption drives the need for the right set of tools to be able to optimize Spark's performance in various application domains. To this end, we will discuss in this paper some aspects of performance tuning, including:

1. Fault tolerance in Spark and how it compares to other popular data processing frameworks
2. Implementation of Iterative Algorithms
3. Performance Tuning and the impacts of RDD caching, Serialization, and Disk I/O
4. Benchmarking tools for Spark and similar frameworks

2 Fault Tolerance

Spark differentiates itself from other large-scale distributed, shared memory systems in the way it handles fault tolerance. Instead of incurring the overhead of replica creation and management, the Spark driver saves the lineage graph as course-grained transformations on input data. Lineage information is tracked at the level of partitions, and RDDs can be rebuilt from lineage graph information, on node failure [33]. We will compare fault tolerance mechanisms in Spark with five other open-source, big data processing frameworks.

Google MapReduce For fault tolerance, Google's MapReduce [7] implementation uses a pull model as opposed to a push model to move data between mappers and reducers. On node failure, the push model forces re-execution of map tasks. One disadvantage to the pull model is that it forces the creation of multiple smaller files saved to disk. However, the impact of these are mitigated by batching and sorting intermediate results. In contrast, Spark provides the option of caching intermediate results in memory, reducing disk overhead costs.

Dryad [12] is a parallel data processing engine suited for coarse-grained transformations on data. Dryad enables the developer to build efficient distributed applications by handling resource scheduling and concurrency optimizations, in addition to an API for graph construction and modification. Jobs in Dryad are directed acyclic graphs representing a logical computation to be mapped to physical resources at run time. Spark uses a similar execution model. Dryad uses batching to improve execution time, and classifies vertices into those that need to be run sequentially, and those that can use a thread pool to be run in parallel, for efficient pipelined execution. Dryad has a re-execution-based fault tolerance policy. On a read error, Dryad terminates the current process, and re-creates the vertex causing the error. Subsequent vertices are not impacted. Vertex execution failures are reported by either sending an error through a daemon before exiting, or by informing the job manager through a heartbeat timeout.

Apache Tez [24] is a framework that provides the components to build scalable dataflow processing engines, without loosing the customizability of specialized systems. Tez is tightly integrated with the YARN [27] resource management layer to improve performance for dataflow processing engines. Tez allows users to model computations as directed acyclic graphs similar to Dryad. Tez uses a combination of checkpointing and lineage graphs for re-execution based fault tolerance. If a task produces missing data, an InputReadError is produced as a control statement. The DAG dependencies are used to determine recursively the stage of failure or latest checkpoint, and re-executes only the portion impacted by the failure.

Pregel [18] is a graph database framework that inspires the underlying structure of Spark's graph processing module GraphX. Pregel's data computation model is based on *supersteps*, where each superstep contains vertices that can be executed in parallel. Pregel achieves fault tolerance through checkpointing. At each superstep, each worker saves its data partition to persistent storage based on a predefined checkpoint frequency. At node failure, the master node reallocates the partitions to the existing set of nodes, but the missed supersteps must be recomputed before the next iterations can continue. Logging outgoing messages, in addition to checkpointing, allows Pregel to only recover lost partitions, improving latency and minimizing consumed resources.

Apache Flink [6] is an open-source framework primarily for data-stream processing. Flink provides an alternative to systems that use micro-batching to process data, where querying incurs high latency and produces approximate results by design. Flink, in coordination with durable messaging systems like Apache Kafka [28], instead processes data through a versatile windowing mechanism and by maintaining different types of state information. For fault tolerance, Flink uses both checkpointing and partial re-execution. Flink uses Asynchronous Barrier Checkpointing such that taking a "snapshot" of the execution does not interfere with execution. Flink manages this by inserting control events, checkpointing barriers, into the input stream, which indicate snapshot boundaries. Upon failure, all operator states are returned to the point of the last valid snapshot, and re-executed from that point onwards. Flink is primarily a data stream processing framework, while Spark is a batch processing framework. However, Flink can run batch-processing applications, and Spark can handle streaming data through Spark Streaming.

3 Iterative Algorithms

The evaluation and optimization of iterative algorithms present a particularly interesting chalenge in dataflow systems like Spark. In certain iterative applications, like PageRank, lineage graphs can become quite large. Spark is implemented in Scala [19], which is a functional and object-oriented programming language which runs on the Java Virtual Machine (JVM) and integrates seamlessly with existing Java programs. Spark saves data objects and DAG information associated with each stage to the JVM heap on the driver node. Once the heap is filled, the JVM is forced to do full garbage collection. This event increases in frequency as lineage graphs expand with more iterations. In addition, recomputing results from a large lineage graph in case of node failure may extend execution time significantly. In this scenario, manual checkpointing, while not necessary for fault tolerance, can improve performance because the DAGScheduler uses materialized RDD results for computation. Zaharia et. al [32] show that persisting the intermediate result of a map operation in logistic regression over multiple iterations can lead to 20x speedup in execution time. A short conference paper by Zhu, Chen, et al. [34] present an algorithm for adaptive checkpointing in Spark to reduce the overhead of garbage collection. The authors determine the need to cache intermediate results based on the rate of utilization of the heap space. Reaching a threshold rate indicates that the current RDD should be checkpointed.

Dataflow systems are an abstraction based on directed acyclic graphs, originally designed for indexing, filtering, aggregation, and transformation tasks. This abstraction is also well-suited for iterative tasks such as machine learning algorithms and graph analytics. Ewen et al [8] present two types of iterative operations: *Bulk Iterations*, which compute a completely new result from input data, and *Incremental Iterations*, where each iteration result only modifies or adds to some small subset of the input data. Because Spark is a batch processing framework, its dataflow model works well with bulk iterations. Spark is not well-optimized to address incremental iterations, where a mutable state must be updated and carried to the next iteration. The immutability of RDDs significantly impacts the computational efficiency of algorithms with incremental iterations, since it cannot exploit the sparse dependencies of these tasks.

However, the ability to cache intermediate results in memory presents an advantage to using Spark for iterative algorithms as opposed to MapReduce frameworks like Hadoop [7]. Iterative algorithms in Hadoop are treated as multiple MapReduce computations, where memory is freed at the end of each MapReduce cycle [10]. The overhead of reading input data and writing to HDFS after each iteration is costly. In Apache Flink, iterative algorithms are dealt with by creating a unique iteration step which contains a DAG within itself. The head and tail nodes of the graph are connected implicitly by a feedback loop, and iteration barriers are sent as control events in the loop to indicate the beginning and end of a superstep in the bulk synchronous parallel model of execution. In addition, Flink's Graph API, Gelly, handles sparse computational dependencies using delta iterations.

We will now analyze select iterative algorithms, and their implementation and performance in Spark.

3.1 Page Rank

PageRank is a bulk iteration algorithm that iteratively updates a rank for each node of an input graph by summing rank contributions of adjacent nodes. This implementation of PageRank is based on the Pregel programming model [18], which uses vertex-based partitioning to split the input graph across the worker nodes, and exchanges updates to each rank through shuffle operations. Pregel also uses combiners to aggregate by key on each worker node before a cogroup operation. Zaharia et. al [31] found that RDD caching improved performance over Hadoop by 2x, and hash-based vertex partitioning and combiners further improves the execution time.

Gu et al. [10] conduct a performance comparison between Hadoop and Spark for PageRank on synthetic and real graph datasets. PageRank is computed on a cluster running Spark with 1 master node and 7 worker nodes, each with 3GB memory. The paper presents a modification of the naive PageRank algorithm, to deal with two scenarios: *spider trap*: groups of nodes with no edges to other nodes outside of the group, and *dangling nodes*: nodes in the graph without any edges.

```
val lines = spark.read.textFile("….txt").rdd
val links = lines.map(…).distinct().groupByKey().cache()
var ranks = links.mapValues(v => 1.0)
for  (iter <- 1 to n) {
          val contribs = links.join(ranks).flatMap{
                    (page, (urls, rank)) => urls.map(u  =>(u, rank / urls.size))}
          ranks = contribs.reduceByKey(_ + _).mapValues(sum => a + (1-a) * sum)
}
```

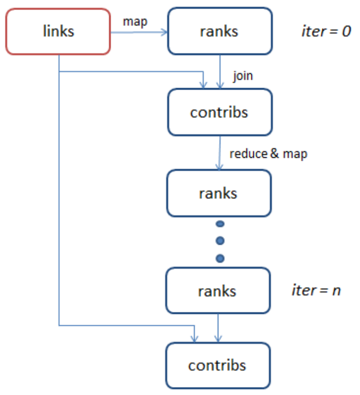


Figure 1: The code presented is an implementation of the naive PageRank algorithm in Spark. The diagram represents the lineage graph for this code (adopted from [32]). The bold text represents transformations on the data leading to the creation of new RDDs, and the links RDD (in red) is cached in memory.

Each iteration in PageRank requires a shuffle for aggregating values to compute ranks, and then another shuffle to update the ranks on the edge partitions. SparkBench [16], a benchmkaring tool for Spark, ran PageRank on a unstructured Google web graph dataset. Because of the relatively small size of the data, disk and network I/O overhead were non-obvious. However, the increasing size of the intermediate results over the set of 100 iterations, suggests that PageRank is a memory-bound workload. Each iteration of the computation creates a new RDD which stores the temporary rank values for each node in the graph. GraphX [30] is a distributed graph processing module built on top of Spark, which presents a series of optimizations to improve the performance of graph queries, on par with graph processing systems like Giraph [5] and GraphLab [17]. Some of Spark optimizations in GraphX include horizontal partitioning of the vertex and edge collections and index reuse. Index reuse allows derived aggregated vertex and edge collections to share indices with the original vertices collection. This technique in particular is useful in reducing the execution time of each iteration in PageRank by over 50% on the Twitter graph dataset. In addition, GraphX handles data skew better than the naive PageRank algorithm due to these optimizations [26].

3.2 K-means Clustering

K-means clustering is a machine learning algorithm which partitions N observations into K clusters, and is representative of the bulk-iteration type of iterative algorithms. An observation, n, belongs to the cluster k with the nearest mean. At each iteration, the algorithm uses a training model of N elements to update parameters of each of the K centroids. Generally, since $K < N$, data to be shuffled across the network among partitions is low. Therefore, intermediate results between iterations can more easily be cached in memory. [26] finds that K-Means in Spark is

CPU bound, where parsing the input text into Point objects causes significant CPU overhead, and where RDD caching makes little impact on subsequent iterations. Logistic Regression is another machine learning algorithm similar to K-Means clustering. Each iteration in logistic regression is less computationally intensive than iterations of K-means, so this algorithm is more sensitive to serialization and I/O.

3.3 Connected Components

The connected components algorithm represents the incremental iteration type of iterative algorithm, unlike Page Rank and K-means. In Connected Components, each node is labeled with the lowest vertex ID of the group of connected components; only a subset of the vertex properties are updated. GraphX uses Incremental View Maintenance to optimize performance of graph queries in this scenario. Only modified vertices are sent to their respective edge partition sites after a graph operation, and unmodified mirrored vertices are reused. Results show that incremental view maintenance has a significant impact on reducing communication data for this algorithm, as the number of ‘active vertices’ converges rapidly [30].

4 Performance Tuning in Spark

In Spark, the DAG, or physical execution plan, is organized into stages, where stage boundaries indicate shuffle dependencies. This execution model impacts performance in Spark because it determines which tasks are pipelined, and how tasks are parallelized. Each stage in the DAG can have its own unique resource consumption characteristics and complex shuffling patterns [16]. These factors present challenges to understanding performance characteristics of workloads in Spark.

We present the impact of RDD caching, serialization and disk I/O improvements on performance in Spark. Then, we present some general observations on the topic of performance tuning.

4.1 RDD Caching

Most machine learning and graph computation algorithms run in Spark consist of a number of stages proportional to the number of iterations. While caching RDDs can offset the cost of Disk I/O, the challenge lies in choosing the optimal memory cache size for RDDs, which varies for different workloads. For example, for logistic regression, caching all RDDs within memory produces a 5.1x factor improvement in performance. However, RDD caching is not a universal solution. For algorithms such as Triangle Counting, SQL-based queries and RDD map reduce queries, caching RDDs is shown to increase execution time [16]. There is a tradeoff between the amount of memory allocated to caching RDDs and task memory, which is used to store shuffle data during shuffle stages. If all memory is allocated to RDD caching, a specific stage that incurs a large amount of shuffle data is forced to store it all on disk, causing a performance slowdown. Therefore, [16] suggests dynamic RDD cache allocation, and more work on the developers side to characterize their workloads: certain stages can be classified as *RDD Caching friendly* and *non RDD Caching friendly.*

In addition, Spark provides 11 storage levels for RDD caching such as in memory, on disk, on disk and in memory, and Tachyon [15]. Shi et al. [26] consider the impact of storage levels and memory limits on the effectiveness of RDD caching for the K-means algorithm. Results show that storage level only impacts the execution time of the first iteration. While the storage level has no impact on subsequent iterations of the algorithm, the number of cached RDD partitions does impact the performance. In Spark, the memory allocated to storing cached RDD partitions is configurable by setting *MemoryStore*.

The effectiveness of RDD caching for a specific application is also dependent on whether the workload is CPU or disk bound. If it is disk bound, RDD caching can have significant performance improvements. If the application is CPU bound, like in the scenario of K-means clustering, where the main overhead was caused by converting text into Point objects, this will not have significant impacts on performance. However, it can minimize CPU overhead because intermediate results are now cached [26].

4.2 Serialization

Li et al. [16] tested various combinations of data compression (Snappy, LZF, LZ4) and serialization (Kyro Serialization, Java serialization) offered in Spark. Serialization and Data compression methods present a tradeoff between I/O and computation time. Accordingly, the study found that for applications that are CPU intensive, like Logistic Regression, using no form of data compression or serialization is best. For the K-means algorithm, RDD caching without serialization is about 1.1x faster than caching with serialization, as this algorithm is already CPU-bound. For PageRank, the impact of serialization is seen more significantly: while serialization reduced the size of the RDD by 20%, the CPU overhead for serialization and de-serialization is higher than Disk I/O overhead without serialization in GraphX [26].

4.3 Disk I/O

Disk I/O happens at four different points in Spark: reading input data, reading and writing writing shuffle data spilled to disk, and writing output data, not including the time spent on serialization and de-serialization [21]. Shi et al. [26] found that the number of disks had minimal impact on Disk I/O for intermediate results, but can have a significant impact in the case of an unbalanced cluster (several CPU cores with minimal number of disks). If an algorithm is disk-bound, an option is to increase JVM heap size, to avoid spilling data to disk. However, as the JVM heap size increases, execution time may not improve, due to increased overhead for garbage collection and OS page swapping. Finally, serialization formats, as discussed previously, incur additional computation time costs despite the savings in Disk I/O. Analysis of Disk and network I/O had only minimal impact on the execution time, of 19% and 2 % on average respectively [21].

4.4 Other Performance Observations

1. Queries in Spark are largely CPU bound rather than disk bound [21, 26]. One cause for this is the decision to use Scala, which forces data read from disk to be serialized as Java objects

from a byte buffer. To demonstrate the impact of this decision, a sample query written in C++ reduced the CPU time by a factor of 2 [21].

2. Another performance consideration in Spark are “data stragglers”, or tasks with execution time much longer than average, which singularly increase the total computation time. These stragglers are attributed to data skew, where one task is forced to compute on a larger set of data than other tasks. Ousterhout et al. [21] provide an addition reason for data skew: Java’s just-in-time compilation optimizes tasks repeated over a certain threshold. Task stragglers could be tasks run for the first time.

3. Li et al. [16] make the observation that for the workloads tested by their benchmarking suite, resource consumption is quite stable throughout execution, so dynamic cluster configuration which could allocate necessary CPU and memory resources may not be effective.

4. Spark’s thread-based model prevents context-switching overhead caused by these process-based models like MapReduce [26].

5 Benchmarking in Spark

Users have a need for debugging tools in Spark, because the current functional API does not provide an understanding of the performance implications of their applications. Performance debugging is especially important for users to understand the efficiency of certain data structures or how the work is partitioned across nodes [4]. Current benchmarking tools evaluate performance on a variety of different dimensions, including CPU, network, and disk usage. Here we survey some framework-specific and framework-agnostic tools for big data processing frameworks in general and those specific to Spark.

5.1 Framework-Specific Tools

Certain benchmarking solutions target a specific framework, running a series of realistic tests representing the full range of functionality of the framework. We will highlight some of these framework-specific tools here.

HiBench [11] is a benchmarking tool specifically to test Hadoop on synthetic and real datasets for a diverse set of workloads, including microbenchmarks such as sorting, machine learning algorithms, and web search. These workloads are evaluated based on CPU, memory, and disk usage, throughput, and execution time.

Spark-perf [2], developed by Databricks, is focused primarily on testing machine learning applications in Spark, but does not cover the full range of modules in Spark (currently under development).

Spark Performance Analysis Project [21] presents blocked time analysis as a simplified method for evaluating the performance of Spark applications. Blocked time analysis analyzes the execution time of each task subtracting blocking due to disk and network I/O. This is a solution to the challenging task of determining task completion time, as tasks are pipelined and executed in parallel in Spark. A simulation of the tasks are then used to determine total execution time given these shorter tasks. The advantage of this tool is in its simplicity. However, this method requires additional instrumentation to be added to Spark, and does not measure CPU usage.

5.2 Multiple Framework Tools

Multiple Framework tools are more generalized benchmarking solutions which allow users to run applications on different frameworks to evaluate and compare performance on different workloads.

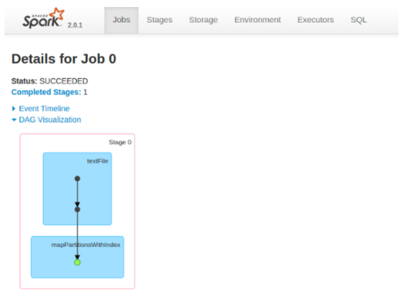
BigDataBench [29] targets different application scenarios on both real and synthetic datasets. Application types include real time and offline analytics and online services. The workloads represent both real world and generated synthetic data in structured, semi-structured, and unstructured format. Examples include Amazon Movie Reviews, Google Web Graph, Facebook Social Network, ProfSearch Person Resumes, and E-Commerce Transactional Data. The workloads form typical relational database queries such as Select, Aggregate, and Join, iterative computations such as PageRank, Connected Components, and K-Means, and scanning and sorting jobs. In total, 19 workloads are represented, tested on several frameworks including Spark, Hadoop, Cassandra [13], and traditional RDBMS systems.The performance is measured in both user-definable metrics such as the number of processed requests, the number of operations, and data processed per second and architectural metrics: MIPS and cache MPKI.

TPC-DS [23] is a Decision Support benchmark created by the Transaction Processing Performance Council. This benchmark runs complex business intelligence queries such as data extraction, reporting, iterative OLAP, and Ad-Hoc DSS queries on structured data in a snowflake schema. It differs from previous TPC benchmarks in that it incorporates both real time static decision queries, and queries that reflect long-term planning.

BigBench [9] extends TPC-DS to include benchmarking for structured and semi-structured data, and aims to address the *volume*, *variety*, and *velocity* trinity of big data systems. BigBench uses Parallel Data Generation Framework (PGDF) to generate synthetic non-structured datasets, and bases workloads on real world applications related to marketing and merchandising, from sentiment analysis to inventory management. The main metric for performance comparisons is execution time.

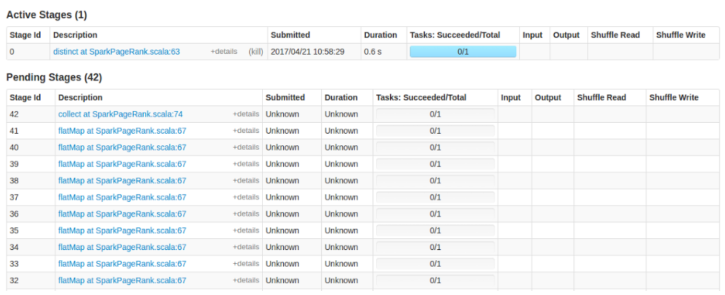
5.3 Spark Benchmarking

Agrawal et al. [3] present a set of guidelines for comprehensive testing of Spark. These dictate comprehensive testing of Spark requires testing the Spark core engine, library modules, and different cluster configurations. In addition, tests must be repeatable, scalable, and determine bottlenecks within the program. Finally, data generation must be realistic and scalable. We have previously discussed Spark-perf, which has not been maintained since 2015 and does not provide functionality for a majority of the algorithms in MLlib, core RDD functions, and SparkSQL. We also discussed the Spark Performance Analysis Project, which provides a thorough analysis of Spark Performance, but is limited by its blocked time analysis metric of performance; it requires additional hardware configurations and does not provide an understanding of CPU usage unless profiled separately. Finally, technology-agnostic solutions such as BigDataBench do provide a large range of workloads on which to test Spark, but is especially hardware-focused in its performance analysis. A promising benchmarking tool currently in development is SparkBench [16], which is built largely on the



Details for Job 0

Status: SUCCEEDED
Completed Stages: 1



Active Stages (1)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	distinct at SparkPageRank.scala:63	+details (kill)	2017/04/21 10:58:29	0.6 s	0/1				

Pending Stages (42)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
42	collect at SparkPageRank.scala:74	+details	Unknown	Unknown	0/1				
41	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
40	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
39	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
38	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
37	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
36	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
35	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
34	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
33	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				
32	flatMap at SparkPageRank.scala:67	+details	Unknown	Unknown	0/1				

Figure 2: Images above are snapshots of the web UI provided by Spark. The right image shows the DAG visualization of Job 0 of the PageRank implementation in GraphX. The left image shows the active and pending stages of a naive PageRank example, including the number of tasks completed and duration of each stage.

principles set forth by [3]. It stands out from other benchmarking suites in the space due to its ability to evaluate Spark performance in the context of the workload characterization.

5.4 Run Time Debugging Environments

One of the biggest challenges in Spark is the ability to profile and debug code, specifically to identify inefficient data structures or skewed data partitioning. Currently, Spark provides a web UI for DAG visualization, and information about active/completed/failed jobs, stages, and tasks for the running application. It also provides environmental information such as available storage and metrics for SQL queries. While this UI provides fundamental information to characterize a running application, other big data processing systems have rich UIs with additional functionality to enable iterative and intuitive development. One of these systems is Pig [20], an abstraction of Hadoop specifically to bridge the gap between SQL queries and MapReduce, through a dataflow language, PigLatin. To enable performance debugging, Pig provides Pig Pen, an interactive debugging environment. This UI allows queries written by the user to be run on a smaller *sandbox* dataset, presenting the schema and results at each step. This environment encourages the user to develop and test programs incrementally. Another UI developed by Pimentel et al. [22] integrate IPython with noWorkflow to enable data provenance in interactive notebooks, through graph visualization. The advantage of this UI is that provenance information is integrated with the notebook rather than separate to it, and is presented in a visually appealing, comprehensible manner. Spark currently lacks similar tools to enable users to debug their applications with an interactive and intuitive GUI environment, presenting an opportunity for future work.

6 Conclusion

Thorough workload characterization is necessary to understand the performance implications of Spark applications. Understanding whether a workload is CPU-bound or disk-bound is essential to understanding whether or not RDD caching, serialization, and changes to heap size will have intended impact on performance. Zaharia et al. [32] noted that users have control over ”persistence and partitioning” of Spark RDDs, and benchmarking is key to allow users to make those decisions correctly. SparkBench [16] is one of the more recent, Spark-specific tools for benchmarking these applications, along with several other technology-agnostic benchmarking solutions. Pig Pen has emerged as a useful and intuitive debugging framework for Pig. We have yet to see a similar fully-developed solution for Spark.

References

[1] Project and product names using "spark".

[2] Spark-perf: spark performance tests.

[3] Dakshi Agrawal, Ali Butt, Kshitij Doshi, Josep-L Larriba-Pey, Min Li, Frederick R Reiss, Francois Raab, Berni Schiefer, Toyotaro Suzumura, and Yinglong Xia. Sparkbench–a spark performance testing suite. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 26–44. Springer, 2015.

[4] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. Scaling spark in the real world: performance and usability. *Proceedings of the VLDB Endowment*, 8(12):1840–1843, 2015.

[5] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11, 2011.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, 38(4), 2015.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5(11):1268–1279, 2012.

[9] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208. ACM, 2013.

[10] Lei Gu and Huan Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 721–727. IEEE, 2013.

[11] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[13] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[14] Jacek Laskowski. *Mastering Apache Spark 2*.

[15] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[16] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, pages 1–15.

[17] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[18] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[19] Martin Odersky et al. The scala programming language. *URL http://www. scala-lang. org*, 2008.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[21] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[22] Joao Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. Collecting and analyzing provenance on interactive notebooks: When ipython meets noworkflow. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, Edinburgh, Scotland, 2015. USENIX Association.

[23] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 582–587. ACM, 2002.

[24] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.

[25] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, pages 1–20, 2016.

[26] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[27] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[28] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.

[29] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.

[30] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical report, Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, 2011.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[33] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[34] Wei Zhu, Haopeng Chen, and Fei Hu. Asc: Improving spark driver performance with automatic spark checkpoint. In *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, pages 607–611. IEEE, 2016.