SparkBench – A Spark Performance Testing Suite

Dakshi Agrawal[1], Ali Butt[2], Kshitij Doshi[5], Josep-L. Larriba-Pey[3], Min Li[1](✉), Frederick R. Reiss[1], Francois Raab[4], Berni Schiefer[1], Toyotaro Suzumura[1], and Yinglong Xia[1]

[1] IBM Research, San Jose, USA
{agrawal,minli,frreiss,tsuzumura,yxia}@us.ibm.com, schiefer@ca.ibm.com
[2] Virginia Tech, Blacksburg, USA
butta@cs.vt.edu
[3] Universitat Politècnica de Catalunya BarcelonaTech, Barcelona, Spain
larri@ac.upc.edu
[4] InfoSizing, Manitou Springs, USA
francois@sizing.com
[5] Intel, Mountain View, USA
kshitij.a.doshi@intel.com

Abstract. Spark has emerged as an easy to use, scalable, robust and fast system for analytics with a rapidly growing and vibrant community of users and contributors. It is multipurpose—with extensive and modular infrastructure for machine learning, graph processing, SQL, streaming, statistical processing, and more. Its rapid adoption therefore calls for a performance assessment suite that supports agile development, measurement, validation, optimization, configuration, and deployment decisions across a broad range of platform environments and test cases.

Recognizing the need for such comprehensive and agile testing, this paper proposes going beyond existing performance tests for Spark and creating an expanded Spark performance testing suite. This proposal describes several desirable properties flowing from the larger scale, greater and evolving variety, and nuanced requirements of different applications of Spark. The paper identifies the major areas of performance characterization, and the key methodological aspects that should be factored into the design of the proposed suite. The objective is to capture insights from industry and academia on how to best characterize capabilities of Spark-based analytic platforms and provide cost-effective assessment of optimization opportunities in a timely manner.

1 Introduction

Spark's brisk evolution and rapid adoption outpace the ability of developers and deployers of solutions to make informed tradeoffs between different system designs, workload compositions, configuration optimizations, software versions, etc. Designers of its core and layered capabilities cannot easily gauge how wide ranging the potential impacts can be when planning and prioritizing software changes. While Spark-perf [16]

R. Nambiar and M. Poess (Eds.): TPCTC 2015, LNCS 9508, pp. 26–44, 2016.
DOI: 10.1007/978-3-319-31409-9_3

can be used to calibrate certain categories of operations, a Spark-specific, comprehensive and extensible performance evaluation alternative is essential for ferreting out inefficiencies and anomalies. This proposal is intended to be a starting point for a community driven development of such a testing suite. With this proposal we plan to open discussion and solicit feedback and participation from the community at the very beginning of designing such a performance testing suite.

1.1 Objective

The objective is to develop a far-reaching performance testing suite that enables performance comparisons between different levels of Spark offerings, including Spark libraries and Spark core. The suite is intended to facilitate evaluation of technologies and be relevant to Spark adopters and solutions creators. We anticipate that the implementation and execution of this suite will benefit from efforts of many groups of professionals – Spark operators, workload developers, Spark core developers, and vendors of Spark solutions and support services.

The following sections present the use cases, the fundamental requirements of the performance testing suite, the design of data models and data generators, the chosen workloads covering the Spark ecosystem, the execution and auditing rules, and the performance metrics. Finally, we conclude the proposal and indicate some areas for future work.

1.2 Related Work

Benchmarks and performance testing suites serve many different communities. They are valuable tools for software engineering teams to assess the performance impact of design trade-offs, to refine choices in system architectures, to inform implementation choices and to identify performance bottlenecks. They can be used by researchers to evaluate new concepts and algorithms. They are excellent vehicles for assessing the performance impact of new hardware or different hardware topologies. They can be used by users and system integrators to gain a deeper understanding of the capabilities offered by competing technologies. No one performance test can ever perfectly serve the needs of all constituencies, but the TPC and SPEC benchmarks, as well as open source benchmarks like DOTS [13] have proven track records in providing value to a broad spectrum of constituencies.

Overall, the focus of benchmarks and testing suites can span a spectrum from low-level (e.g. SPEC CPU2006 [7]) to high-level (e.g. TPC-E [6], SAP SD, LDBC SNB [11]) functions. In the big data application domain, existing performance testing suites and benchmarks can be grouped into three categories: component-level testing, technology-specific solutions and technology-agnostic solutions.

Component-level tests (sometimes called micro-benchmarks) focus on stressing key system primitives or specifically targeted components using a highly synthetic workload. Examples of big data component-level testing include the suite of Sort Benchmarks [17], YCSB [23] and AMP Lab Big Data [21].

Technology-specific solutions involve a set of representative applications in the targeted domains and generally mandate the use of a specific technology to implement the solution. The goal is to test the efficiency of a selected technology in the context of a realistic operational scenario. Examples of technology-specific solutions testing for big data are MRBench [29], PigMix [28], HiBench [18, 19] and SparkBench [24].

Technology-agnostic solutions aim at creating a level playing field for any number of technologies to compete in providing the most efficient implementation of a realistic application scenario within the targeted application domain. No assumption is made about which technology choice will best satisfy the real world demands at a solution level. Benchmarks such as BigDataBench [20], BigBench [22] and TPC-DS [6] fall into this category.

The Spark performance testing suite introduced in this paper is designed to fall into the category of technology-specific solutions. It aims at providing a Spark specific, comprehensive and representative set of workloads spanning the broad range of application types successfully implemented within the Spark ecosystem. While other benchmarks such as BigBench [22], BigDataBench [20] and HiBench [18] each cover a small number of Spark-enabled workloads, they are far from including a comprehensive coverage of the full set of application types supported under Spark. Spark-Bench [24] and Spark-perf [16] provide good initial starting points, yet they fall short of covering the full Spark picture. In particular Spark-perf is a performance testing suite developed by DataBricks to test the performance of MLlib, with extensions to streaming, SQL, data frame and Spark core currently under development. In contrast the Spark performance testing suite proposed in this paper incorporates a broader set of application types including text analytics, Spark R and ETL, with realistic and scalable data generators to enable testing them in a more real-world environment.

2 Targeted Dimensions

A distinctive aspect of the Spark performance testing suite proposed here is that it will simultaneously target the following three dimensions of performance analysis within the Spark ecosystem.

- **Quantitative Spark Core Engine Evaluation,** by enabling comparative analysis of core Spark system ingredients, such as caching policy, memory management optimization, and scheduling policy optimization, between baseline (standard) Spark release and modified/enhanced variations. It anticipates in-depth performance studies from multiple perspectives, including scalability, workload characterization, parameter configurations and their impacts, and fault tolerance of Spark systems.
- **Quantitative Spark Library Evaluation,** by allowing quantitative comparison of different library offerings built on top of the Spark core engine. These include the categories of SQL, streaming, machine learning, graph computation, statistical analysis, and text analytics. We envision interest in comparisons among different levels/versions of Spark libraries, as well as alternative libraries from vendors.
- **Quantitative Staging Infrastructure Evaluation,** by providing insight toward analysis relative to a fixed software stack, two examples of which are

(a) comparison across different runtimes and hardware cluster setups in private datacenters or public clouds, and with use of Spark data services, (b) gaining of configuration and tuning insights for cluster sizing and resource provisioning, and accelerated identification of resource contentions and bottlenecks.

In summary, the Spark performance testing suite is intended to serve the needs and interests of many different parties, and aims to cover the technology evaluation of the Spark ecosystem by exercising its key components comprehensively.

3 Requirements

Measurement is the key to improvement, in computing as in many other spheres. The Transaction Processing Performance Council [6] and the SPEC [7] are among the most prominent performance benchmarking organizations. Supplementing them are efforts like LDBC, for more specific yet significant areas like graph and RDF technologies benchmarking [27]. Application level benchmarks from vendors like SAP [14] and Infor Baan [15] play a key role in influencing solution choice, workload balancing and configuration tuning. Open source communities have created a rich variety of performance test suites, DOTS [13] being just one example. From these and other efforts we recognize an established set of core attributes that any new performance testing suite should possess.

From Huppler [3] we have the following attributes

– Relevant
– Repeatable
– Understandable
– Fair
– Verifiable
– Economical

In the context of a 21st century Spark performance testing suite we can further refine these timeless attributes as follows:

- **Simple, Easy-to-use and Automated**: The suite needs to be simple to understand, deploy, execute, and analyze in an automated fashion, requiring only modest configuration. Considering the rapidly evolving nature of Spark ecosystem, automation is essential.
- **Comprehensive**: The performance testing suite should be comprehensive and representative of the diversity of applications supported by Spark. Different Spark operations can put pressure on different resources, in different ratios. Since a benchmark suite cannot capture all such operations, it is important that the chosen representatives reflect both the diversity of Spark uses at the application level and the variant stresses put on the computing resources at the systems level. For example, the suite should include workloads that have high resource demands for specific system resources to test extreme cases for a provisioned system as these workloads will be one of several uses of Spark.

- **Bottleneck Oriented**: Frequently a role of performance testing is to spur technology advancement. The concept of bottleneck (or choke point) analysis appears with the LDBC benchmark effort and is a good means to shape workloads, and thereby provide impetus for innovation by drawing attention to tough, but solvable, challenges.
- **Extensible**: Due to the rapid evolution of Spark, the Spark performance testing suite needs to be able to evolve, which includes allowing users to easily add new or extend/expand existing capabilities. A successful Spark benchmark will successfully address the many parts of the Spark taxonomy and be flexible to extend to new capabilities that the community may develop. This is illustrated in Fig. 1 Spark Taxonomy, derived from Databricks [1], starting with the Spark Core Engine as a base with several workload-focused extensions on top.
- **Portable**: The benchmark suite should run on a broad range of open systems and be designed to be readily portable to other operating systems if required.
- **Scalable:** To allow scaling of tests to large distributed or cloud environments, the suite should facilitate generation of data that is sufficiently voluminous and varied that it exercises systems under test in statistically significant ways. The rate at which new data needs to be generated also needs to create meaningful stresses.

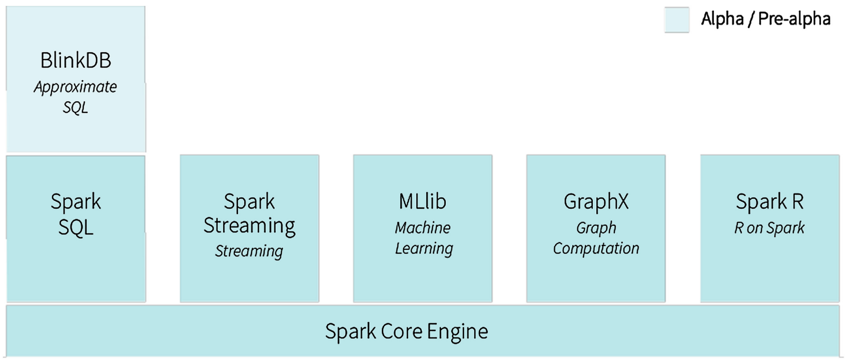

Fig. 1. Spark Taxonomy (https://databricks.com/Spark/about)

4 Data Model

Ideally we will want to develop a unified data model that allows integrating the multiple varieties of data, (relational tables, resilient distributed datasets, semi-structured data, graphs, arrays, and unstructured data such as text) that arise in Spark usages. A possible approach could be to start with an existing, largely traditional, relational data model and then extend it to the emerging domains. This has been popularized by the BigBench benchmark [8] which started with TPC-DS [6] and extended its data model to the SQL/MR and Mahout machine learning areas. An interesting possibility is to build on top of the LDBC Social Network Benchmark (SNB) [11] which already covers a number of the Spark ecosystem domains, and to extend it further.

We believe it would be effective to use this extension approach for the Spark performance testing suite. As the design and overall implementation of the Spark performance testing suite is refined, the choice between TPC-DS, SNB or some other option can be made. Since any kind of relational model would support SQL, the data model for that domain is quite straightforward. Streaming can use the relational tables both as sources and as targets. Work to recast the Mahout-based machine learning with Spark MLlib [2] is already underway. For graph computation, understanding the link between the Social Network Benchmark of LDBC and how it could contribute to the testing suite will be a challenge [25]. Finally, defining a data model across various specialized Spark features and spanning text analytics, and SparkR for a unified performance testing suite appears feasible in principle, but more investigation is needed.

5 Data Generator

Scalable, efficient, and realistic data generation plays a crucial role in the success of a data intensive testing suite. The built-in scalable data generator of TPC-H made it very compelling, as recounted in a retrospective of TPC-H [4]. TPC-DS further refined this notion and added non-uniform distributions and correlation. Multiple research efforts (e.g., Myriad [9] from TU Berlin, Parallel Data Generation Framework (PDGF) from Bankmark [10], and DATAGEN from the LDBC Social Network Benchmark [11]) are addressing these well appreciated needs for scalability and for reflecting real-world characteristics into synthetic data generation. We recognize similarly that while it may require significant development effort, rapid generation of representative data for use in large scale clusters will be critical in the adoption of the proposed Spark performance testing suite. We expect that the definition, design and validation of a powerful data generator is a key work item as we proceed with the implementation of this Spark performance testing suite.

5.1 Properties of Data Generator

A data generator must have multiple key attributes in order to be successful; in particular, it will need to be:

- Open source and transparent: to allow users to view and modify the code to enhance it or alter its behavior
- Scalable: to allow users to use it in a variety of environments, from small single node servers with only a few GB of data to the largest clusters with hundreds or even thousands of nodes and Petabytes of data.
- Parallel and distributed: as practical generation of petabytes of data will require many multi-core servers
- Incremental: to allow data to grow in iterations one must be able to generate data in "chunks" rather than in one monolithic start to finish method
- Realistic: Although synthetic, data must strive to be realistic; i.e., representative of real data sets. Following are some of the common properties of real life data:

  - Correlation
  - Non-uniform distributions representing high levels of skew
  - Fields/columns with both low and high cardinality
  - Varying numeric fields
  - Text fields of varying length
- Able to represent relationships: as capturing connections across multiple tables/data sets is essential for advanced analytics.
- Random but predictable: The data must be sufficiently random to challenge the system-under-test with unknown input sets. And yet, the output of data processing must be sufficiently predictable to permit validation of results and to determine which sets of input may trigger comparable levels of processing by the workload.

6 Workloads

6.1 Machine Learning

Several subtasks in the Machine Learning (ML) category are desirable for inclusion in the proposed suite, and are described further below. For each, the Spark performance testing suite should contain a reference implementation based on the most recent Spark MLlib capabilities. The suite should include a written specification for each machine learning subtask. It should be easy to substitute a different ML algorithm or implementation, provided that the replacing algorithm meets the specifications identically with reference implementation. The specification should therefore be at a sufficiently high level to permit alternative implementations, and be sufficiently strict to ensure that variant implementations produce useful outputs leading to quality results from a data science perspective.

It is important for the data generator for the machine learning tasks to cover a broad range of data characteristics which affect the behavior of ML algorithms. Input data should include dense, sparse, and hybrid (some attributes dense, others sparse) inputs. Algorithms should run against both small and large numbers of features.

In order to be able to draw broadly accepted conclusions and to drive innovation towards continuously improved machine learning, the generated data should also be robust and highly representative of real world data. In machine learning, processing speed is important but certain levels of quality are even more important. In a nutshell, generated data should not be so well-conditioned as to favor optimization algorithms that are not usable in practice with real-world data. We believe that substantial work is required to construct a robust, realistic data generator for the performance testing suite assessing Spark machine learning implementations.

Based on the above criteria, a Spark performance testing suite could augment the features tested in Spark-perf [16] and supplement [16] with extensions in the areas of logistic regression, support vector machine and matrix factorization. These are widely used regression, classification and recommendation algorithms for machine learning applications.

6.1.1 ML Subtasks

Logistic Regression. 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, sex, etc. The algorithm uses the stochastic gradient descent to train the classification model. The input data sets are kept in memory through RDD abstractions, and the parameter vector is calculated, updated, and broadcast in each iteration.

Support Vector Machine. A support vector machine (SVM) model is trained by constructing a set of hyper-planes 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 nonlinear classifications.

Matrix Factorization. Matrix factorization, typically used by recommendation systems, is a collaborative filtering technique that fills in the missing entries of a user-item association matrix. Matrix factorization in Spark currently supports model based collaborative filtering and can be configured to use either explicit or implicit feedback from users.

Random Forest Classification. A random forest classifier uses a large set of relatively simple decision trees to perform classification tasks. The classifier combines the results of the individual classifiers to produce a consensus result. Random forests have shown to be effective in a number of machine learning tasks beyond their primary uses in classification. Since building a random forest involves training many small models in parallel, the task involves different communication patterns from other types of training tasks. MLlib exposes a random forest classifier implementation via the `mllib.tree.RandomForest` API.

6.2 Graph Computation

Graph is a very widely utilized data model. Consequently, a comprehensive Spark performance testing suite needs to include graph processing. The graph packages supported under Spark include GraphX and Dato. Additional projects are underway.

Graph computations to be included in the testing suite need to be representative of common types of graphs and graph analytics operations, and graph properties should reflect those in practical applications. Therefore, the data generator should be able to generate graphs of different types, such as the social graphs and man-made graphs (e.g. road-network) where a sensitive metric (say, vertex degree distribution) can be varied to obtain a wide range of analytics impact. Where practical, we want to be able to link graph data with other data generated for other components of the Spark performance testing suite. We propose to draw considerably from the LDBC Social Network Benchmark [25, 26] and need to examine how best to adapt their benchmarks to the Spark ecosystem and the Spark performance testing suite infrastructure. Different types of work, such as static structure-based traversal, graph structure morphing/property updates, and the processing of property-rich graphs, are highly desirable to include in the graph analytics operations of the testing suite.

The following subtasks are proposed according to the above criteria.

6.2.1 Graph Generator Subtask

The Linked Data Benchmark Council (LDBC [27]) has created two benchmarks. One of them is the LDBC Social Network Benchmark [25, 26] (SNB) whose correlated graph generation, graph querying tests, complex graph dependencies and scalable benchmark drivers reflect landmark innovation in graph benchmarking. Its data generator (ldbc_snb_datagen) uses experimentally extracted metrics and correlations to produce scalable datasets mimicking real world social networks. LDBC introduced a new choke-point driven methodology for developing benchmark workloads, which combines user input with that from expert systems architects.

The SNB analytics workload [26] includes General Statistics, Community Detection, Breath First Search, Connected Components and Graph Evolution; a list that will grow in the near future with the addition of new algorithms. We propose to select workloads from this benchmark for the Spark performance testing suite and develop additional workloads to cover various aspects of graph computing as detailed in the next subsection.

Graph500 [12] is a graph benchmark focusing on data-intensive workloads and particularly on large graphs. It is based on a breadth first search in a synthetically generated large undirected graph with power-law property based on the Kronecker model with average degree of 16. It measures performance in TEPS (for Traversed Edges Per Second) and its problem size can be changed by varying a SCALE parameter that determines the total number of vertices as 2^{SCALE}. Thus its generated graphs can be of various sizes, suitable for benchmarking software or platforms at different scales. It consists of three phases: construction, computation, and validation.

A dataset generator for Belief Propagation should be included as it would make rich property graph analytics possible, and it should produce directed acyclic graphs (DAG) with (conditional) probability distributions of various scales.

6.2.2 Graph Analytics Subtask

Primitive operations for graph analytics, such as creating/reading/updating/deleting (CRUD) vertices, edges, and properties, are nearly universal. Tests calibrating these graph analytics building blocks are therefore essential to include in the suite. The metrics would cover throughput (e.g., number of edges traversed per second), latency, and scalability.

Graph construction for large scale property graph is another key subtask to cover. The metrics would be running time, and scalability, akin to a subset of Graph500 [12].

Graph query is of interest as it involves both structural information and property information [11].

Pagerank exercises graph structure traversal with fixed active working set; *Triangle counting* stresses similarly. In such graph computations, each vertex iterates through tasks of gathering information (say, rank score) from its (partial) neighbors (say, predecessors), updating local information, and propagating it to the other neighbors (say, successors); and the iterations continue until convergence or certain termination conditions are reached.

Breadth-first Search (BFS) represents another type of graph traversal where only the vertices on the traversal frontier are affected, and the workload can vary from one iteration to another.

Single Source Shortest Path (SSSP) with a maximum traversal depth represents a type of graph traversal similar to BFS (e.g., Bellman Ford algorithm), but it only touches a local subgraph, instead of engaging the entire graph. This workload can evaluate if a graph processing framework on Spark can efficiently address local or subgraph computations.

Belief Propagation on a Bayesian network represents property-rich graph processing, and is a type of graph analytics operation that occurs in many cognitive computing applications. For example, Loopy Belief Propagation on a Bayesian network traverses graph iteratively, but when vertex or edge properties are updated, it can become a multi-pattern and computationally intensive graph structure operation.

Graph Triangulation (a.k.a. Chordization) represents a type of graph processing workload where the structure is dynamically changed. It is used to find graph cliques (dense subgraphs) and/or the hyper graph representation. It is an iterative graph processing algorithm that modifies topology in each iteration. It can be used to determine whether graph dynamics can be efficiently captured by the system.

Collaborative Filtering finds a lot of application, especially in recommendation systems. It involves a number of local graph searches on a bipartite graph, possibly in parallel, and is suitable for evaluating the concurrent local traversal capacity of a graph analytic system.

Graph Matching and motif searching are similarly used extensively. When the target graph lacks an index, these operations are challenging and possibly involve significantly high local traversals.

Various *Graph Centrality* metrics, such as the betweenness, degrees, closeness, clustering coefficient should also be considered due to their wide use in many real graph processing solutions.

6.3 SQL Queries

SQL continues to be an enduring query language due to its ubiquity, the broad ecosystem of tools that supports it, and its ability to evolve and support new underlying infrastructure and new requirements, such as advanced analytics and data sampling.

One area where a different approach might be warranted is in the construction of the queries. Historically, different vendors have proposed queries that combined a variety of SQL processing constructs, such as the TPC-D/H/DS benchmarks. In such case, the coverage was often not obvious initially. There has been some good analysis of the TPC-H query set [6].

We propose that we introduce a set of elemental or atomic queries that assess basic scan, aggregation, and join properties, then a set of intermediate queries that add challenges both to the query optimizer and to a runtime engine, and finally some complex and very challenging queries, representing ROLAP concepts and advanced analytic processing.

6.4 Streaming Applications

Streaming applications can be characterized along three dimensions: latency, throughput, and state size. Ideally, the Spark performance testing suite would exercise each of these dimensions at three representative values - high, medium, and low - giving a total of twenty-seven use cases. However, guided by applicability in the real world scenarios, the number of use cases can be pruned down to a more manageable count initially, and grow as more diverse workloads migrate to Spark over time.

6.4.1 Streaming Subtasks

The following are some of the use cases covering a subset of the twenty seven combinations posed above.

Real-time Model Scoring. The emphasis in this use case is on small and medium latency ranges. Low latency is defined as response time in seconds and sub-second values[1]. An example is sending an SMS alert to a prepaid mobile customer notifying them of their leftover account balance and potentially inserting a marketing message in the SMS alert after a model evaluation. In this use case, a latency in the range of 20 ms to a few seconds is desired with lower latencies offering a larger payoff – for example, a 50 ms delay does not force the customer to take a second look at the phone screen to get the marketing message while a delay exceeding 10 s may lead to customer pocketing the phone without getting the marketing message. Other examples in this area are cybersecurity, fraud detection for online transactions, and insertion of ads in webpages, where latency requirements are considerably more stringent (possibly 100 ms or less).

In all use cases of real-time model scoring, state management is an independent dimension. The state could be as simple as a single quantity (e.g., in the example above, minutes of calls left) which gets updated based only on the current record, with the model scored on this simple state. Or, the state could be a very complex assemblage of hundreds of attributes across millions of entities, updated by incoming records; with the model evaluation proceeding over a selection of such entities (e.g., a fraud detection application which maintains a profile with hundreds of attributes for each customer, updates it based on incoming records and scores a model on the profile.)

Near Real-time Aggregations. Near real-time aggregates are required for a number of scenarios in which a physically distributed system is monitored for its health using the key performance indicators of its elements. Examples include monitoring of traffic congestion on roads, monitoring of communication networks and energy grids.

In these usages either sliding or tumbling window aggregates are computed from streaming records. Incoming records may be enriched by joining them with reference information. The aggregation window size could be from one minute up to an hour. In a typical case, records arrive out of order and are delayed, and contain a timestamp which should be used for aggregate computation.

[1] Current Spark Streaming is not recommended for sub-second response time, however, we discuss this here in the anticipation of future improvements.

For near real-time aggregations, throughput is an independent dimension. The volumes could range from a few hundred GB a day (enriched Twitter data) and range up to 500 TB a day (e.g., telecommunication call data records).

Another independent dimension is the number of aggregation buckets - which themselves can vary from 100's of millions (one bucket for each mobile user) to several thousands (monitoring of different metropolitan cities within US).

The two subtasks listed above could be used to produce four use cases that could become part of the Spark performance testing suite.

6.5 SparkR

R is a widely used language for statistical analysis, and the SparkR project will allow practitioners to use familiar R syntax in order to run jobs on Spark. In the short term we propose following SparkR subtasks for inclusion in the performance testing suite, with future additions as SparkR capabilities evolve.

6.5.1 SparkR Subtasks

Data Manipulation. This covers SparkR DataFrame functions, and operations that can be performed in a purely distributed fashion and includes all "record-at-a-time" transformations such as `log()`, `sin()`, etc.

Segmented or Subpopulation Modeling. This is a technique in which the data is broken down into subpopulations, such as by age and gender, and a separate model is built for each segment of the data. Assuming each segment is of a "reasonable" size, R's existing ML libraries can be used to build the models.

Ensemble Modeling. This is a technique in which the data is broken down into randomly selected sub-samples, each of which is a "reasonable" size. R's existing ML libraries can be used to build the component models of the ensemble; however, the code that constructs the ensembles has to be written. This code could be in Scala or maybe in R.

Scoring R Models. This is applying an existing model. In essence, it is a "record-at-a-time" transformation.

6.6 Spark Text Analytics

Text analytics is an extremely broad topic, encompassing all types of analysis for which natural language text is one of the primary inputs. To give the benchmark broad coverage of this domain, we propose including a wide variety of text-related subtasks described in the section that follows. For each subtask, the benchmark should include a reference implementation based on an open-source NLP software stack (e.g., Stanford NLP toolkit) consistent with the open-source license under which the performance test suite is released.

Different commercial vendors have proprietary implementations of these subtasks and will want to substitute their own implementations for the reference implementation. Each subtask should include a specification that is sufficiently detailed to permit vendors to perform such substitutions. For example, it should be possible to perform the "rule-based information extraction" subtask using IBM's System T engine. In general, proprietary implementations should be required to produce the same answer as the reference implementations. For tasks with an element of randomization, the result of a proprietary implementation should be of equal utility compared with the reference result. For example, in the "deep parsing" subtask, any deep parser that produces substantially the same parse trees as the reference implementation (say, 90 % or greater overlap) would be acceptable.

Data for the subtasks should consist of English-language documents that a human being could read and understand. The data generator should work either by taking a random sample from an extremely large "canned" collection of documents, or by mixing together snippets of English text drawn from a suitably large database. A range of document sizes from 100 bytes up to 1 MB should be supported.

6.6.1 Text Subtasks

Rule-based Information Extraction. Information extraction, or IE, is the process of identifying structured information inside unstructured natural language text. IE is an important component of any system that analyzes text. In some cases, IE is used to identify useful features for other NLP tasks. In other cases, it is the primary NLP component of a processing pipeline. Rule-based IE systems use a collection of fixed rules to define the entities and relationships to extract from the text. These systems are widely used in practice, particularly in feature extraction applications, because they deliver high throughput and predictable results. The rule-based IE task will stress Spark by producing large amounts of structured information from each input document.

Information Extraction via Conditional Random Fields. A number of supervised statistical techniques are used in NLP as an alternative to using manually curated rules. Conditional Random Fields (CRF) is currently the most popular of these techniques. A CRF is a graphical model, similar to a hidden Markov model, but with greater expressive power. CRF-based information extraction involves transforming each input document into a graph with missing labels; then a collection of labeled training data is used to compute the maximum likelihood estimate for each missing label. The CRF-based extraction task will stress Spark due to its very high memory requirements.

Deep Parsing. Deep parsing involves computing the parse trees of natural language sentences according to a natural language grammar. Deep parsing is an important component of advanced feature extraction tasks such as sentiment determination. The deep parsing task will stress Spark due to its high CPU requirements and large output sizes.

Online Document Classification. Automatically classifying a stream of incoming documents into two or more categories is a very common NLP task, arising in applications such as publish-subscribe systems and spam filtering.

Batch Topic Clustering. Topic clustering is a family of supervised learning techniques for identifying important topics within a corpus of text, while simultaneously classifying documents according to the topics. The resulting topics and clusters can be used to understand the corpus at a high level, or serve as features for other machine learning tasks.

6.7 Resilient Distributed Dataset (RDD) Primitives

Since the main programming abstraction in Spark is RDDs, offering RDD primitive facilitates end users to gain micro-level understanding of how RDD performs within Spark framework. The reference test suite implementation of RDD primitives should be based on the latest version of Spark core and make it easy to substitute a different RDD implementation, add new RDD operations and remove obsolete RDD operations.

While RDDs supports a wide variety of transformations and actions, the testing suite should cover the key operations broadly. In particular, the RDD primitives should include IO related, shuffle, set and compute RDD operations. We choose not to include set operations with `RDD.subtract and RDD.intersection` because their characteristics are a combination of compute and shuffle RDD operations.

The testing suite should provide a data generator which produces synthetic data sets to exercise the various RDD primitives. Considering that data skew is known to commonly exist in data analytics workloads, the data generator needs to be able to generate data sets with different types of statistical distribution representing different levels of data skew. Note that whereas this type of workloads is aimed at micro-level RDD performance, the data generator needs not to generate realistic data sets.

6.7.1 RDD Primitives Subtasks

IO Related RDD Operations. This set of operations identify how fast Spark reads and writes data from/to local or distributed file system and creates/removes RDDs for the targeted data set with various size. Examples of RDD actions include `SparkContext.textFile`, `RDD.unpersist`.

Shuffle RDD Operations. This set of operations focus on stressing the shuffle behavior of RDD operations. They quantify how fast shuffle RDD operations can perform given different data set sizes. Examples of RDD transformations include `RDD.union`, `RDD.zipPartition, RDD.reduceByKey, RDD.groupByKey, and RDD.treeAggregate`.

Compute RDD Operations. This set of operations exercise how fast the compute RDD operations can perform. Examples of RDD transformations include `RDD.map, RDD.flatMap`. We choose to specify trivial map function such as `sleep` within compute RDD operations so that we can isolate the evaluation of the overhead of Spark framework.

Check-pointing RDD Operations. This set of operations assesses how fast the check pointing RDD operations can perform. This is a key factor which helps encourage the adoption of Spark framework seeing that failure is a common phenomenon in large scale data centers and check-pointing and lineage are the fundamental failure recovery mechanisms within Spark.

The key evaluation metrics for RDD primitives are as follows: (1) throughput: how many RDD transformations and actions can Spark conducts within a given time window; (2) scalability: how does the execution time change when the RDD data set size increases; (3) efficiency of failure recovery: how fast can Spark recover from a RDD data partition lost.

7 Execution and Auditing Rules

In this section we discuss the outline of the proposed execution and auditing rules of the testing suite. These rules typically govern the preparation of the testing environment, the execution of the testing suite, and the evaluation, validation and reporting of the test results.

During test environment preparation, a user first identifies the targeted workload(s) and accordingly chooses a benchmarking profile. To reduce the performance testing overhead, the testing suite provides a set of benchmarking profiles. Each profile includes a subset of workloads from the entire testing suite, along with corresponding data generation configurations and sequence(s) of workload execution. For example, the testing suite has one benchmark profile for each workload described in Sect. 6. If the testing focuses on machine learning, the machine learning benchmarking profile can be used, eliminating the overhead of running the other workloads.

The execution rules also require both single user and multi-user execution scenarios. A single user scenario executes the workloads included in the benchmarking profile one after another with a focus on evaluating and collecting per-workloads metrics. A multi-user scenario runs multiple benchmarking profiles concurrently with profile launching time following a certain statistical distribution. The multi-user scenario also could support running the profiles against different data sets instead of reusing the same data sets. This gives the users a better understanding of the performance implication of the targeted system under a multi-user scenario.

Having selected a benchmarking profile, the testing environment can be set up. This includes provisioning a set of connected machines and installing the software stack needed for running the testing suite's profile.

Once the testing environment is ready, the testing suite's data generator is used to generate needed datasets and loading them into the storage component of the tested system. The user is then ready to proceed with running the benchmark with a workload execution sequence defined by the chosen benchmarking profile. To check whether a benchmark run is valid, all the workload execution should report successful return status and pass the validation phase.

The testing suite includes an output quality evaluation and validation phase to evaluate the correctness of the execution. While this varies by workloads, a user can get an initial result indicating the validity and performance level of a test run from the result log generated by the testing suite.

Another important aspect of the execution and auditing rules is the requirement to provide sufficient reporting about the testing to allow others to reproduce the results. The system details needed in the disclosure report includes the hardware configurations such as CPU, memory, network, disk speed, network controller, switches; and software information such as the OS name and version, other software names and versions relevant to the testing suite, and the parameters used to generate input datasets. The full set of result logs generated by the testing suite should also be provided online and in a format that is easy to reproduce.

8 Metrics

Whenever a set of somewhat independent measurements are performed, a question always arises – how should the results be aggregated into a single metric, a simple and comparable measure of composite "goodness"? Historically, geometric mean has been chosen for some benchmarks [8, 9], while it has been argued later that a geometric mean is inappropriate [6]. Several other options, viz. an arithmetic mean, a weighted arithmetic mean, a harmonic mean, a weighted harmonic mean, etc. may also be applicable candidates for devising a figure of merit.

Different components of the proposed suite have widely diverse origins, and are likely to be accorded dissimilar measures of importance by different people. Thus arriving at a consensus single metric is particularly challenging in this case. For the purpose of this paper we therefore defer any specific recommendations. We understand and accept that a simple single figure of merit for any set of measurements is highly desirable.

Overall, we believe that most tests are best characterized by multi-user throughput. However, as the community-based approach to evolve and finalize the ideas presented in this paper gets underway, we expect considerable open discussion before a final metric is settled upon.

9 Preliminary Work

Preliminary work [24] has been done in the design of a benchmarking suite focusing on targeted dimensions of quantitative Spark Core and the staging of infrastructure evaluation. In this work ten diverse and representative workloads were chosen, covering four types of applications supported by Spark – machine learning, graph computation, SQL and streaming workloads. The ten chosen workloads were characterized using synthetic data sets and demonstrating distinct patterns with regards to resource consumption, data flow and communication features affecting performance. The work also demonstrated how the benchmarking suite can be used to explore the performance implications of key system configuration parameters such as task parallelism.

10 Conclusion, Ongoing, and Future Work

As Spark is increasingly embraced by industries and academia, there is a growing need for a comprehensive set of Spark performance tools. Such tools should enable developers, integrators and end users within the Spark and big data community to identify performance bottlenecks, explore design trade-offs, assess optimization options and guide hardware and software choices with a focus on key workload characteristics. While early work has been done in this area, the Spark ecosystem, being a relatively new data analytics platform, lacks a far reaching set of performance tools. This paper introduces a framework for the creation of a comprehensive Spark performance testing suite to address this need. It identifies several key factors such a performance testing suite should consider, a set of Spark workloads consistent with those factors, and the requirements for their reference implementations and corresponding data generators.

Currently we are focusing on machine learning and graph processing workloads. More specifically, we are identifying real world data sets as seeds for data generator. They exemplify the data characteristics that need to be preserved in order to generate realistic data sets at selected scale factors. We are also looking into meaningful metrics for each workload with a focus on setting apart high performing algorithms and implementations from less efficient ones. In the future, we plan to add additional workloads to the identified set. For instance, as a necessary step between the data generation process and the analytics workflow, we identify extract-transform-load (ETL) as another key workload within the Spark ecosystem. We also plan to explore the possibility of supporting a Python interface within the performance testing suite. Moreover, we recognize the need for a formal definition of the testing suite's detailed execution and auditing rules, along with the selection of representative metrics that create an environment where true apples-to-apples comparisons can be made and alternative choices can be fairly evaluated.

Acknowledgements. The authors would like to acknowledge all those who contributed with suggestions, ideas and provided valuable feedback during earlier drafts of this document. In particular we would like to thank Alan Bivens, Michael Hind, David Grove, Steve Rees, Shankar Venkataraman, Randy Swanberg, Ching-Yung Lin, and John Poelman.

References

1. DataBricks. https://databricks.com/
2. Mahout. http://mahout.apache.org/
3. Huppler, K.: The art of building a good benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 18–30. Springer, Heidelberg (2009)
4. Boncz, P., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 61–76. Springer, Heidelberg (2014)

5. Jacob, B., Mudge, T.N.: Notes on calculating computer performance. University of Michigan, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science (1995)
6. Transaction Processing Performance Council. http://www.tpc.org/
7. Standard Performance Evaluation Corporation. https://www.spec.org/
8. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.-A.: BigBench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 1197–1208. ACM, New York, NY, USA (2013)
9. Alexandrov, A., Tzoumas, K., Markl, V.: Myriad: scalable and expressive data generation. Proc. VLDB Endow. **5**(12), 1890–1893 (2012)
10. Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.: A data generator for cloud-scale benchmarking. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 41–56. Springer, Heidelberg (2011)
11. Linked Data Benchmark Council Social Network Benchmark (LDBC-SNB) Generator. https://github.com/ldbc/ldbc_snb_datagen
12. Graph500 generator. http://www.graph500.org/specifications
13. DOTS: Database Opensource Test Suite. http://ltp.sourceforge.net/documentation/how-to/dots.php
14. SAP. http://www.sap.com
15. Infor LN Baan. www.infor.com/product_summary/erp/ln/
16. Spark-perf. https://github.com/databricks/spark-perf
17. Sort Benchmark. http://sortbenchmark.org/
18. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In 26th IEEE ICDEW, pp. 41–51, March 2010
19. Performance portal for Apache Spark. http://01org.github.io/sparkscore/plaf1.html
20. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zheng, C., Lu, G., Zhan, K., Li, X., Qiu, B.: Bigdatabench: a big data benchmark suite from internet services. In: IEEE 20th HPCA, pp. 488–499, February 2014
21. AMPLab Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/
22. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.-A.: Bigbench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD, pp. 1197–1208 (2013)
23. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM SOCC, pp. 143–154 (2010)
24. Li, M., Tan, J., Wang, Y., Zhang, L., Salapura, V.: SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, CF 2015, Article 53, ACM, New York, NY, USA (2015)
25. Erling, O., Averbuch, A., Larriba-Pey, J.L., Chafi, H., Gubichev, A., Prat, A., Pham, M.-D., Boncz, P.: The LDBC social network benchmark: interactive workload. In: Proceedings of SIGMOD 2015, Melbourne (2015)
26. Capotă, M., Hegeman, T., Iosup, A., Prat, A., Erling, O., Boncz, P.: Graphalytics: a big data benchmark for graph-processing platforms. In: Proceedings of GRADES2015, co-located with ACM SIGMOD/PODS (2015)

27. Angles, R., Boncz, P.A., Larriba-Pey, J.-L., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martínez-Bazan, N., Kotsev, V., Toma, I.: The linked data benchmark council: a graph and RDF industry benchmarking effort. SIGMOD Record **43**(1), 27–31 (2014)
28. PigMix. https://cwiki.apache.org/confluence/display/PIG/PigMix
29. Kim, K., Jeon, K., Han, H., Kim, S.,x Jung, S., Yeom, H.Y.: MRBench: a benchmark for MapReduce framework. In: IEEE ICPADS (2008)