Scaling Spark in the Real World: Performance and Usability

Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or,
Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, Matei Zaharia†
Databricks Inc. †MIT CSAIL

ABSTRACT

Apache Spark is one of the most widely used open source processing engines for big data, with rich language-integrated APIs and a wide range of libraries. Over the past two years, our group has worked to deploy Spark to a wide range of organizations through consulting relationships as well as our hosted service, Databricks. We describe the main challenges and requirements that appeared in taking Spark to a wide set of users, and usability and performance improvements we have made to the engine in response.

1. INTRODUCTION

Interest in MapReduce and large-scale data processing has led to the emergence of a wide array of cluster computing systems [3, 6, 7]. These systems use a variety of new APIs, often based on functional programming, to support both relational queries and more complex types of processing (*e.g.,* extract-transform-load work or machine learning).

Of these systems, Apache Spark [1] has become one of the most widely adopted, with, to our knowledge, over 500 production deployments, and the most active contributor community at Apache (over 400 contributors in 2014). Unlike previous specialized systems, Spark offers a *general* engine based on task DAGs and data sharing on which workloads such as batch jobs, streaming, SQL and graph analytics can run [14, 15, 2]. It has APIs in Java, Scala, Python and R.

As Spark transitioned from early adopters to a broader audience, we had a chance to see where its functional API worked well in practice, where it could be improved, and what the needs of new users were. This paper describes the major initiatives we have taken at Databricks to improve usability and performance of Spark. We cover both engine improvements and new APIs to make Spark accessible to non-experts, such as a table-oriented DataFrame API [2].

2. CHALLENGES AND REQUIREMENTS

Overall, Spark has been successful in its goal of supporting general analytics workloads: we and others have been able to implement libraries on top for SQL, streaming, graph processing and machine learning, often with comparable performance to specialized engines [15, 2, 5]. The generality of the Spark engine is important because most users *combine* multiple of these types of processing in their workloads.

Nevertheless, Spark's functional API did pose some challenges for both users and the system, given the heterogeneity of data types and computations supported. The most common challenges we saw were the following:

Functional API semantics. The core Spark API is based on collections of Java/Python objects, on which users run arbitrary functions written in these languages through operators like `map` or `groupBy` [14]. We found that users often had trouble selecting the best functional operators for a given computation. For example, one common problem is using Spark's `groupByKey` operator, which returns a distributed collection of (key, list of value) pairs, and then performing an aggregation on each list (*e.g.,* a sum). The `groupByKey` has to send each list of records to one machine because that is its return signature, but this computation would be much faster with Spark's `reduceByKey` operator, which can perform partial aggregation on each node.

Because the functions passed to Spark are arbitrary Java or Python code, it is also hard for the engine to analyze them and replace operators automatically. Some research has proposed static analysis of UDFs [11], but such analysis can be brittle for complex object-oriented programs.

Debugging and profiling. Distributed programs are inherently hard to debug, even with Spark's side-effect-free, functional API, because users have to worry about work distribution and skew. We found that the most challenging issues are in performance debugging: users often do not realize that their work is concentrated on a few machines, or that some of their data structures are memory-inefficient.

Memory management. "Big data" comes in a wide range of formats and sizes, requiring careful memory management throughout the engine. While external operations for aggregation and joins are well-understood, we found other sources of high memory use. For example, data records in some applications (*e.g.,* image processing) can be hundreds of megabytes each, requiring careful accounting as each record is read. As another example, Spark initially assumed that the data in each block of a file (typically 128 MB in HDFS) can all fit in memory at once, but for some highly compressed datasets, each block could decompress into 3-4 GB.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

Large-scale I/O. Spark clusters and workloads have grown significantly, with the largest cluster now being over 8000 nodes and individual jobs processing more than 1 PB [13]. We have invested significant engineering to make Spark's networking and I/O layers operate well at this scale.

Accessibility to non-experts. While early cluster computing systems like MapReduce were designed for software engineers, in most organizations, "big data" needs to be accessible to many other individuals, such as domain experts (*e.g.,* statisticians or data scientists) who are not developers. In addition, for all users, higher-level APIs are important because much data analysis is exploratory: users do not have time to write a fully optimized distributed program. To address these challenges, we have invested substantial effort in providing high-level data science APIs that mirror single-node tools, such as R's data frames, over Spark.

We next describe three key areas of work that tackle these challenges: core engine improvements, debugging tools, and a new DataFrame API.

3. ENGINE IMPROVEMENTS

Our main work in the execution engine falls into two domains: memory management and the networking layer. Both focus on making the engine more performant and robust for large-scale workloads.

3.1 Memory Management

To improve memory management, we studied causes of memory problems based on user reports and implemented a per-node allocator that manages all sources of memory usage within each node. Spark initially had a memory manager to track the size of "cached" data that the user chose to materialize in memory, evicting old data blocks when a cap was reached. The original manager did not explicitly track the memory usage for data processing (*e.g.,* scratch space used for joins or aggregations). As a result, a large fraction of the memory exhaustion problems came from processing large joins or aggregations. To address that, we implemented a second cap to track hash tables for joins and aggregation. This cap is allocated dynamically among the threads running these operations as they grow their tables, and threads that are not allowed to take more RAM spill to disk. Lastly, a third space was reserved for "unrolling" blocks that are read from disk to see whether the uncompressed data is still small enough to cache. In all these cases, we check memory usage every 16 records to handle skewed record sizes. With these controls, the engine runs robustly across a wide range of workloads.

3.2 Networking Layer

In Spark's networking layer, the largest challenge was supporting shuffle operations on many nodes. Shuffle operations need to move output data from map tasks to reduce tasks across the network, so that every node is sending some data to every other node. They are challenging to implement because each node may be serving data from multiple disks, multiple connections are generally required to saturate network bandwidth, and care must be taken to balance load (*e.g.,* if all reducers contact one node, it may get overloaded).

We previously wrote a custom network module that was based on Java's NIO. The module used the low-level Java NIO networking API directly and needed to maintain complex state machines internally. In addition, it created higher memory pressure from JVM garbage collection and higher CPU usage than needed due to unnecessary copies of network buffers.

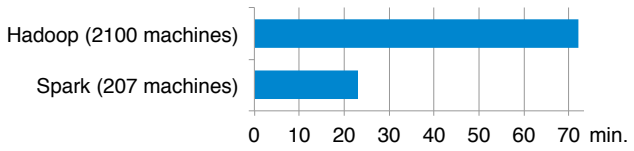

Figure 1: Completion times for 100 TB Daytona GraySort benchmark, comparing Spark's 2014 record to Hadoop's 2013 record.

In Apache Spark 1.2, we replaced the network module with a new implementation based on Netty (`www.netty.io`), a high-performance networking framework. Netty simplifies networking programming by providing a higher level asynchronous event-driven abstraction. Building on Netty, we have introduced a number of features to improve performance and scalability:

- Zero-copy I/O: Instruct the kernel to copy data directly from on-disk files to the socket, without going through the user-space memory. This reduces not only the CPU time spent in context switches between kernel and user space, but also the memory pressure in the JVM heap.
- Off-heap network buffer management: Netty maintains a pool of memory pages explicitly outside the Java heap, and as a result eliminates the impact of network buffers on the JVM garbage collector.
- Multiple connections: Each Spark worker node maintains multiple parallel active connections (by default 5) for data fetches, in order to increase the fetch throughput and balance load across the nodes serving data.

The implementation is able to saturate a full bisection bandwidth network between 200 machines with 10 Gbps links each. We used it to set a new record in the Daytona GraySort competition [12], by sorting 100 TB of on-disk data 3× faster than the previous Hadoop-based record using 10× fewer machines (Figure 1).

4. DEBUGGING TOOLS

To facilitate debugging, we added a wide variety of metrics to Spark's web based application monitoring UI. This UI shows metrics such as time taken to schedule, run, and receive the results for each task; input and output bytes; and memory sizes. The metrics are shown in tables that users can sort by each column to find outliers. They are also increasingly aggregated in graphs, such as a live dashboard with statistics about streaming jobs (Figure 2) and a visualization of the operator DAG (Figure 3).

Apart from metrics, one surprisingly useful feature we added was a "stack trace" button, which can be used to take a trace from any worker and see which functions it is currently running. This serves both as a simple sampling profiler and as a way to identify deadlocks.

Finally, although the current UI provides metrics for Spark's lowest-level functional operators, many programs increasingly use higher-level libraries such as Spark SQL, DataFrames

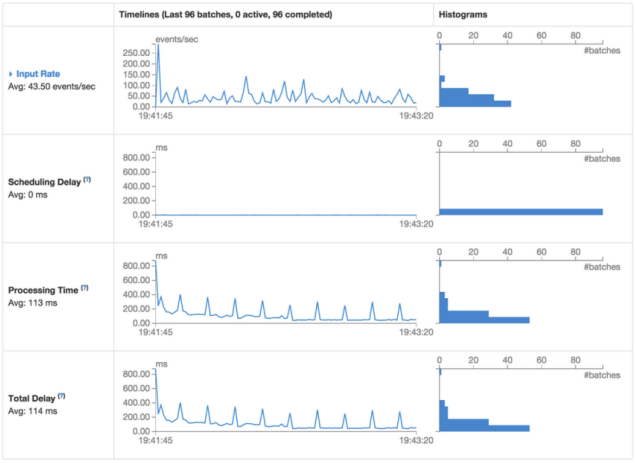

Figure 2: Metrics dashboard for Spark Streaming. The plots update in real time to show input rate, processing time and other information about the running application.

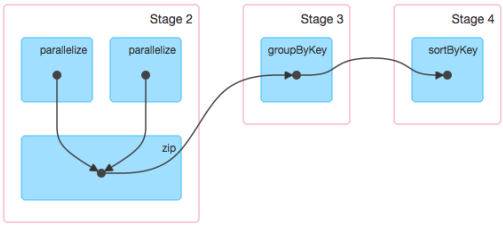

Figure 3: Visualization for operator DAGs in Spark's UI.

and Spark's machine learning library (MLlib). We are also extending the monitoring UI to capture these higher-level operations. In our experience, visibility into the system remains one of the biggest challenges for users of distributed computing.

5. DATAFRAME API

To make Spark more accessible to non-experts and increase the information visible to the engine for automatic optimization, we sought to develop a more declarative API. We chose an API based on data frames, a common abstraction for tables in Python and R. Data frames support operations similar to relational algebra, but expose them as functions in a procedural language (*e.g.,* Python), so that developers can use the control flow and abstraction features of the language around them to write complex programs.

Our DataFrame API implements this standard interface but compiles it using the relational optimizer in Spark SQL [2], enabling rich logical and physical optimizations based on the whole computation.[1] This lets Spark DataFrames handle transformations such as the `groupBy` problem mentioned in Section 2 automatically. To our knowledge, Spark's is the first data frame implementation to use a relational optimizer underneath—previous libraries such as R's are imperative.

The code below shows a short example of DataFrames in Python; the API is similar to `pandas` (`pandas.pydata.org`):

[1]In particular, Spark SQL supports standard logical optimizations, runtime code generation, and columnar storage.

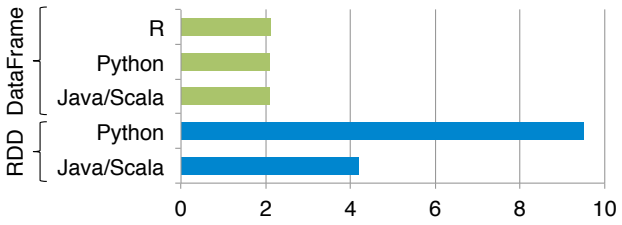

Figure 4: Running time (sec) of a simple aggregation query using DataFrames versus Spark's functional RDD API.

```
means = users.where(users["age"] > 20)
             .groupBy("city")
             .avg("income")
```

The API captures expressions like `users["age"] > 21` as abstract syntax trees to enable algebraic optimization, unlike the opaque user-defined functions passed to Spark's functional operators, such as `map`. Nonetheless, users can still easily invoke UDFs when needed, by passing inline functions like in the core Spark API.

As shown in Figure 4, DataFrame based computations can be 2–5× faster than the functional API. The speedups come from both runtime code generation and algebraic optimizations (*e.g.,* predicate pushdown).

Beyond offering DataFrames for basic data transformations, we are increasingly using them as the input and output format to Spark's libraries (*e.g.,* the machine learning library [8]). This allows us to easily expose Spark's libraries in all supported programming languages, without having to build API with language-specific types in each one. The recently added R bindings to Spark also support DataFrames and will access other libraries this way.

6. ONGOING WORK

We are continuing to improve Spark for both usability and performance. On the usability side, we and other members of the community are augmenting Spark with a large set of standard libraries containing scalable versions of common data analysis algorithms. For example, Spark's machine learning library, MLlib, grew by a factor of 4 in the past year. We have also designed a pluggable data source API that makes it easy to access external data sources in a uniform way using DataFrames or SQL [2]. Together, these APIs form one of the largest integrated standard libraries for "big data," and will undoubtedly lead to interesting design decisions to enable efficient composition of workflows.

On the performance side, under a new project codenamed Tungsten, we are implementing memory management outside the JVM and runtime code generation to bring the performance of DataFrames and SQL to the limit of the underlying hardware [10]. These optimizations will transparently speed up current user code and many of Spark's libraries.

We have also increasingly seen Spark used in research projects, including online aggregation [16], graph processing [5], genomic data processing [9], and large-scale neuroscience [4]. We hope that Spark's relatively small code size and wide array of built-in functions make it amenable to both systems and application-oriented projects.

All of the functionality described in this work is open source and available at `spark.apache.org`.

7. REFERENCES

[1] Apache Spark project. http://spark.apache.org.

[2] M. Armbrust et al. Spark SQL: relational data processing in Spark. In *SIGMOD*, 2015.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[4] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping brain activity at scale with cluster computing. *Nature Methods*, 11(9):941–950, Sept 2014.

[5] J. E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[6] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. *Eurosys*, 2007.

[7] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[8] X. Meng et al. ML pipelines: a new high-level API for MLlib. http://tinyurl.com/spark-ml.

[9] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, 2015.

[10] Project Tungsten. `https://databricks.com/blog/2015/04/28/`.

[11] K. Tzoumas et al. Peeking into the optimization of data flow programs with mapreduce-style UDFs. In *ICDE*, 2013.

[12] R. Xin et al. GraySort on Apache Spark by Databricks. `http://sortbenchmark.org/ApacheSpark2014.pdf`.

[13] R. Xin and M. Zaharia. Lessons from running large scale Spark workloads. http://tinyurl.com/large-scale-spark.

[14] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[15] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[16] K. Zeng et al. G-OLA: Generalized online aggregation for interactive analysis on big data. In *SIGMOD*, 2015.