Fast and Interactive Analytics over Hadoop Data with Spark

MATEI ZAHARIA, MOSHARAF CHOWDHURY, TATHAGATA DAS, ANKUR DAVE, JUSTIN MA, MURPHY MCCAULEY, MICHAEL J. FRANKLIN, SCOTT SHENKER, AND ION STOICA

Matei Zaharia is a fifth-year PhD student at UC Berkeley, working with Scott Shenker and Ion Stoica on topics in computer systems, networks, cloud computing, and big data. He is also a committer on Apache Hadoop and Apache Mesos.
matei@eecs.berkeley.edu

Mosharaf Chowdhury is a PhD student at the University of California, Berkeley, working with Ion Stoica on topics in cloud computing and datacenter networks. He received his undergraduate degree at Bangladesh University of Engineering and Technology (BUET) and a Master's degree from the University of Waterloo in Canada.
mosharaf@cs.berkeley.edu

Tathagata Das is a PhD student at the University of California, Berkeley, working with Scott Shenker on topics in cloud computing, datacenter frameworks, and datacenter networks. He received his undergraduate degree at the Indian Institute of Technology, Kharagpur, India.
tdas@cs.berkeley.edu

Ankur Dave is an undergraduate at UC Berkeley and a Spark contributor since 2010. His projects include Arthur, a replay debugger for Spark programs, and Bagel, a Spark-based graph processing framework.
ankurd@eecs.berkeley.edu

Justin Ma is a postdoc in the UC Berkeley AMPLab. His primary research is in systems security, and his other interests include machine learning and the impact of energy availability on computing. He received BS degrees in Computer Science and Mathematics from the University of Maryland in 2004, and he received his PhD in Computer Science from UC San Diego in 2010.
jtma@cs.berkeley.edu

Murphy McCauley is a Master's student at UC Berkeley. His current focus is on software defined networking.
murphy.mccauley@gmail.com

Michael J. Franklin is the Thomas M. Siebel Professor of Computer Science and Director of the Algorithms, Machines and People Laboratory (AMPLab) at UC Berkeley. His current research is in the areas of scalable query processing, data integration, hybrid human/computer data processing systems, and cross-datacenter consistency protocols.
franklin@cs.berkeley.edu

Scott Shenker spent his academic youth studying theoretical physics but soon gave up chaos theory for computer science. Unable to hold a steady job, he currently splits his time between the UC Berkeley Computer Science Department and the International Computer Science Institute.
shenker@cs.berkeley.edu

Ion Stoica is a Professor in the EECS Department at the University of California at Berkeley. His research interests include cloud computing, distributed systems, and networked computer systems.
istoica@cs.berkeley.edu

The past few years have seen tremendous interest in large-scale data analysis, as data volumes in both industry and research continue to outgrow the processing speed of individual machines. Google's MapReduce model and its open source implementation, Hadoop, kicked off an ecosystem of parallel data analysis tools for large clusters, such as Apache's Hive and Pig engines for SQL processing. However, these tools have so far been optimized for one-pass batch processing of on-disk data, which makes them slow for *interactive* data exploration and for the more complex *multi-pass* analytics algorithms that are becoming common. In

this article, we introduce Spark, a new cluster computing framework that can run applications up to 40× faster than Hadoop by keeping data in memory, and can be used interactively to query large datasets with sub-second latency.

Spark started out of our research group's discussions with Hadoop users at and outside UC Berkeley. We saw that as organizations began loading more data into Hadoop, they quickly wanted to run rich applications that the single-pass, batch processing model of MapReduce does not support efficiently. In particular, users wanted to run:

- More complex, *multi-pass* algorithms, such as the iterative algorithms that are common in machine learning and graph processing
- More *interactive* ad hoc queries to explore the data

Although these applications may at first appear quite different, the core problem is that both multi-pass and interactive applications need to *share* data across multiple MapReduce steps (e.g., multiple queries from the user, or multiple steps of an iterative computation). Unfortunately, the only way to share data between parallel operations in MapReduce is to write it to a distributed filesystem, which adds substantial overhead due to data replication and disk I/O. Indeed, we found that this overhead could take up more than 90% of the running time of common machine learning algorithms implemented on Hadoop.

Spark overcomes this problem by providing a new storage primitive called *resilient distributed datasets* (RDDs). RDDs let users store data in memory across queries, and provide fault tolerance *without* requiring replication, by tracking how to recompute lost data starting from base data on disk. This lets RDDs be read and written up to 40× faster than typical distributed filesystems, which translates directly into faster applications.

Apart from making cluster applications fast, Spark also seeks to make them easier to write, through a concise language-integrated programming interface in Scala, a popular functional language for the JVM. (Interfaces in Java and SQL are also in the works.) In addition, Spark interoperates cleanly with Hadoop, in that it can read or write data from any storage system supported by Hadoop, including HDFS, HBase, or S3, through Hadoop's input/output APIs. Thus, Spark can be a powerful complement to Hadoop even for non-iterative applications.

Spark is open source at http://www.spark-project.org and is being used for data analysis both at Berkeley and at several companies. This article will cover how to get started with the system, how users are applying it, and where development is going next. For a detailed discussion of the research behind Spark, we refer the reader to our NSDI '12 paper [6].

Programming Interface

The key abstraction in Spark is *resilient distributed datasets* (RDDs), which are fault-tolerant collections of objects partitioned across cluster nodes that can be acted on in parallel. Users create RDDs by applying operations called *transformations*, such as map, filter, and groupBy, to data in a stable storage system, such as the Hadoop Distributed File System (HDFS).

In Spark's Scala-based interface, these transformations are called through a functional programming API, similar to the way users manipulate local collections. This interface is strongly inspired by Microsoft's DryadLINQ system for cluster

computing [5]. For example, the following Spark code creates an RDD representing the error messages in a log file, by searching for lines that start with "ERROR," and then prints the total number of error messages:

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(line => line.startsWith("ERROR"))
println("Total errors: " + errors.count())
```

The first line defines an RDD backed by an HDFS file as a collection of lines of text. The second line calls the `filter` transformation to derive a new RDD from `lines`. Its argument is Scala syntax for a function literal or closure. It's similar to a `lambda` in Python or a block in Ruby. Finally, the last line calls `count`, another type of RDD operation called an *action* that returns a result to the program (here, the number of elements in the RDD) instead of defining a new RDD.

Spark lets users call this API both from stand-alone programs and *interactively* from the Scala interpreter to rapidly explore data. In both cases, the closures passed to Spark can call any Java library, because Scala runs on the Java VM. They can also reference read-only copies of any variables in the program. Spark will automatically ship these to the worker nodes.

Although simply providing a concise interface for parallel processing is a boon to interactive analytics, what really makes the model shine is the ability to load data in memory. By default, Spark's RDDs are "ephemeral," in that they get recomputed each time they are used in an action (e.g., `count`). However, users can also *persist* selected RDDs in memory for rapid reuse. If the data does not fit in memory, Spark will automatically spill it to disk, and will perform similarly to Hadoop. For example, a user searching through a large set of log files in HDFS to debug a problem, as above, might load just the error messages into memory across the cluster by calling:

```
errors.persist()
```

After this, she can run a variety of queries on the in-memory data:

```
// Count the errors mentioning MySQL
errors.filter(line => line.contains("MySQL")).count()

// Fetch the MySQL errors as an array of strings
errors.filter(line => line.contains("MySQL")).collect()

// Fetch the time fields of errors mentioning PHP as an array
// (assuming time is field number 3 in a tab-separated format):
errors.filter(line => line.contains("PHP"))
    .map(line => line.split('\t')(3))
    .collect()
```

In-memory data provides a significant speed boost for these queries. For example, in one test, we loaded a 50 GB Wikipedia dump onto a 20-node Amazon cluster, and found that a full-text search through it took 20 seconds with on-disk data or with Hadoop. The same search took only 0.8 seconds with an in-memory RDD.

Fault Tolerance

Apart from providing in-memory storage and a variety of parallel operators, RDDs also automatically recover from failures. Each RDD tracks the graph of transformations that was used to build it, called its *lineage graph*, and reruns these

operations on base data to reconstruct any lost partitions. For example, Figure 1 shows the RDDs in the last query above, where we obtain the time fields of errors mentioning PHP by applying two filters and a map. If any partition of a dataset is lost, e.g., a node holding an in-memory partition of errors fails, Spark will rebuild it by applying the filter on the corresponding block of the HDFS file. Recovery is often much faster than simply rerunning the program, because a failed node typically contains multiple RDD partitions, and these can be rebuilt in parallel on other nodes.

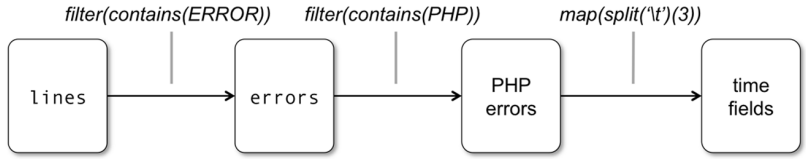

Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs, and arrows represent transformations between them.

Lineage-based fault recovery is powerful because it avoids the need to replicate data. This saves both time in constructing RDDs, as writing data over the network is much slower than writing it to RAM, and storage space, especially for precious memory resources. Even if *all* the nodes running a Spark program crash, Spark will automatically rebuild its RDDs and continue working.

Other Examples

Spark supports a wide range of operations beyond the ones we've shown so far, including all of SQL's relational operators (`groupBy`, `join`, `sort`, `union`, etc.). We refer the reader to the Spark Web site for a full programming guide [8], but show just a couple of additional examples here.

First, for applications that need to aggregate data by key, Spark provides a parallel `reduceByKey` operation similar to MapReduce. For example, the popular "word count" example for MapReduce can be written as follows:

```
val counts = file.flatMap(line => line.split(" "))
             .map(word => (word, 1))
             .reduceByKey((a, b) => a + b)
```

Second, to give an example of an iterative algorithm, the code below implements logistic regression, a common machine learning algorithm for classifying objects such as, say, spam vs. non-spam emails. The algorithm runs MapReduce operations repeatedly over the same dataset to optimize a mathematical function by gradient descent (specifically, to find a hyperplane that best separates the objects). Thus, it benefits greatly from storing the input data in RAM across iterations.

```
val points = spark.textFile(...).map(parsePoint).persist()
var w = Vector.random(D)   // Current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  }.reduce((a, b) => a + b)
  w -= gradient
}
println("Final separating plane: " + w)
```

To show the benefit of Spark for this algorithm, Figure 2 compares its performance against Hadoop for varying numbers of iterations. In this test, with 100 GB of data on a 50-node cluster, Hadoop takes a constant time per iteration of about 110 seconds. In contrast, Spark takes 80 seconds for the first iteration to load the data in memory, but only six seconds for each subsequent iteration.

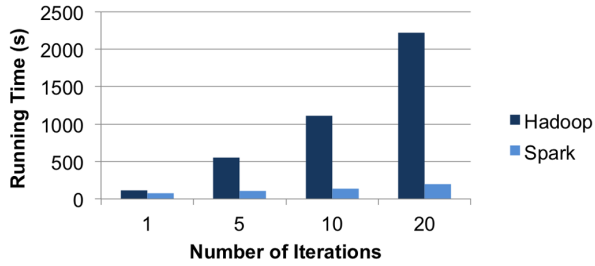

Figure 2: Performance of logistic regression in Hadoop vs. Spark for 100 GB of data on a 50-node cluster. Hadoop takes a constant time of 110s per iteration, much of which is spent in I/O. Spark takes 80s on the first iteration to load the data in memory, but only 6s per subsequent iteration.

User Applications

Spark is in use both at several Internet companies and in a number of machine learning research projects at Berkeley. Some of our users' applications are discussed below.

In-Memory Analytics on Hive Data

Conviva Inc., an online video distribution company, used Spark to accelerate a number of analytics reports that previously ran over Hive, the SQL engine built on Hadoop. For example, a report on viewership across different geographical regions went from taking 24 hours with Hadoop to only 45 minutes with Spark (30× faster) by loading the subset of data of interest into memory and then sharing it across queries [2]. Conviva is also using Spark to interactively search large collections of log files and troubleshoot problems, using both the Scala interface and a SQL interface we are developing (discussed in the next section) called Shark.

Interactive Queries on Data Streams

Quantifind, a startup that specializes in predictive analytics over time series data, used Spark to build an interactive interface for exploring time series. The system periodically loads new data from external feeds (e.g., Twitter streams), runs an entity extraction algorithm to parse the data, and builds an in-memory table of mentions of each entity. Users can then query this table interactively through a Web application that runs Spark queries on the backend [4].

Traffic Modeling

Researchers in the Mobile Millennium project at Berkeley [3] parallelized a learning algorithm for inferring traffic conditions from crowd-sourced automobile GPS measurements. The source data were a 10,000 link road network for the San Francisco area, as well as 600,000 position reports for GPS-equipped automobiles (e.g., taxi cabs, or users running a mobile phone application) collected every minute. By

applying an iterative expectation maximization (EM) algorithm to this data, the system can infer the time it takes to travel across individual road links.

Twitter Spam Detection

The Monarch project at Berkeley used Spark to identify link spam in Twitter posts. They implemented a logistic regression classifier on top of Spark, similar to the example in "Other Examples," above. They applied it to over 80 GB of data containing 400,000 URLs posted on Twitter and 10^7 features/dimensions related to the network and content properties of the pages at each URL to develop a fast and accurate classifier for spammy links.

Conclusion and Next Steps

By making in-memory data sharing a first-class primitive, Spark provides a powerful tool for interactive data mining, as well as a much more efficient runtime for the complex machine learning and graph algorithms that are becoming common on big data. At the same time, Spark's ability to call into existing Java libraries (through the Scala language) and to access any Hadoop-supported storage system (by reusing Hadoop's input/output APIs) make it a pragmatic choice to complement Hadoop for large-scale data analysis. Spark is open source under a BSD license, and we invite the reader to visit http://www.spark-project.org to try it out.

Our group is now using Spark as a foundation to build higher-level data analysis tools. Two ongoing projects that we plan to open source in 2012 are:

- **Hive on Spark (Shark):** Shark is a port of the Apache Hive SQL engine to run over Spark instead of Hadoop. It can run over existing Hive data warehouses and supports the existing Hive query language, but it adds the ability to load tables in memory for greater speed. Shark will also support machine learning functions written in Spark, such as classification and clustering, as an extension to SQL [1]. An alpha release is available at http://shark.cs.berkeley.edu.
- **Spark Streaming:** This project extends Spark with the ability to perform online processing, through a similar functional interface to Spark itself (map, filter, reduce, etc. on entire streams). It runs each streaming computation as a series of short batch jobs on in-memory data stored in RDDs, and offers automatic parallelization and fault recovery for a wide array of operators. A short paper on Spark Streaming appears in HotCloud '12 [7].

For more information on these projects, or on how to get started using Spark itself, visit the Spark Web site at http://www.spark-project.org.

Acknowledgments

Research on Spark is supported in part by an NSF CISE Expeditions award, gifts from Google, SAP, Amazon Web Services, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, and VMware, by DARPA (contract #FA8650-11-C-7136), and by a Google PhD Fellowship.

References

[1] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica, "Shark: Fast Data Analysis Using Coarse-Grained Distributed Memory," SIGMOD, 2012.

[2] D. Joseph, "Using Spark and Hive to Process Big Data at Conviva": http://www.conviva.com/blog/engineering/using-spark-and-hive-to-process-bigdata-at-conviva.

[3] Mobile Millennium project: http://traffic.berkeley.edu.

[4] K. Thiyagarajan, "Computing Time Series from Extracted Data Using Spark," Spark User Meetup presentation, Jan. 2012: http://files.meetup.com/3138542/Quantifind%20Spark%20User%20Group%20Talk.pdf.

[5] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," USENIX OSDI '08.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," USENIX NSDI, 2012.

[7] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient Programming Model for Large-Scale Stream Processing," USENIX HotCloud, 2012.

[8] Spark homepage: http://www.spark-project.org.