Typesafe

BUILDING BIG DATA APPLICATIONS

Getting Started with Spark

Dean Wampler, Ph.D.
dean.wampler@typesafe.com

A Brief History of Big Data

The Rise of Petabyte-Scale Data Sets

Over the last twenty years, Internet giants like Amazon, Google, Yahoo!, eBay, and Twitter invented new tools for working with data sets of unprecedented size, far beyond what traditional tools could handle. They started the Big Data revolution, characterized by the ability to store and analyze these massive data sets with acceptable performance, at drastically reduced costs. The leading open-source tools for Big Data include Hadoop and NoSQL databases like Cassandra and MongoDB.

Hadoop has been a spectacular success, offering cheap storage in HDFS (Hadoop Distributed File System) of datasets up to Petabytes in size, with batch-mode (“offline”) analysis using MapReduce jobs.

New Trends in the Big Data Industry

However, recent trends have forced the industry to change:

- MapReduce provides a difficult programming model for developers and it suffers from a number of performance issues.
- While batch-mode analysis is still important, reacting to events as they arrive has become more important, even for applications where delayed, batch-mode analysis has traditionally been considered adequate. For example, a search index should reflect changes as soon as they are made, rather than lagging behind several hours until the next batch update. Similarly, you want your SPAM filter to learn as it goes. Patchwork solutions like aggregating events in HBase aren't sufficient for most needs.
- Advanced algorithms like those in machine learning and graph theory are increasingly important for extracting maximal value from data.
- Not all data sets are "big", but they would still benefit from the integration capabilities and low costs of Big Data technologies.

New Trends in the Big Data Industry

These trends have lead the industry in several directions, but there is an emerging consensus forming around Apache Spark as the next-generation, multi-purpose compute engine for Big Data applications.

Spark addresses the four trends as follows:

- Spark provides a more flexible, concise, easy to learn programming model for developers, with significantly better performance in most production scenarios.
- Spark supports traditional batch-mode applications, but it also provides a streaming model for Reactive applications.
- The functional-programming foundation of Spark and its support for iterative algorithms provide the foundation for a wide range of libraries, including SparkSQL, for integrated SQL-based queries over data with defined schemas, Spark Streaming, for handling incoming events in near-real time, GraphX for computations over graphs, and MLlib for machine-learning.
- Spark scales down to a single machine and up to large-scale clusters. Spark jobs can run in Hadoop using the YARN resource manager, on Mesos clusters, or in small, “standalone” clusters.

Reactive, Data-centric Application Development

Now you can write batch mode and streaming applications with one tool.

If you are a developer in a conventional Big Data team, you can write Spark jobs to do the work of older MapReduce jobs. Now you can also write streaming applications using the same tool, rather than introducing a totally separate tool or workaround to your environment.

However, if you are a developer of more “conventional” Reactive Enterprise and Internet applications using the Typesafe Reactive Platform, Spark opens up new opportunities for integrating sophisticated data analytics as part of your infrastructure. Because Spark is implemented in Scala, like Akka and Play, it presents a logically-consistent extension to the Typesafe Reactive Platform. Your Spark-based data analytics can deploy and scale with your existing environment, without the need to deploy a separate cluster dedicated to data analysis. However, when your projects grow to the point where you need a dedicated data cluster, Spark will grow with you and still interoperate with the rest of your Akka- and Play-based applications.

Let’s look at some representative architectures.

Hadoop with MapReduce and Spark

Schematically, a Hadoop cluster for a data-centric environment looks like figure 1.

MapReduce and Spark jobs are submitted to the cluster for scheduling and execution by the Resource Manager. Each job is divided into tasks (individual JVM processes) and run on the slave nodes under the control of the Node Manager and other service daemons (not shown). The nodes are chosen by asking the HDFS master service, the Name Node, which nodes contain blocks of data for the files the job should process. The Data Node service on each node manages the blocks the node holds.

The jobs usually write their results back to HDFS. Spark also reads and writes data from other HDFS-compatible file systems, like MapR-FS, the local file system, various databases, network sockets, and message queues like Kafka. Data exchange with databases is done with Sqoop. Other text sources, such as log files, are ingested using Flume.

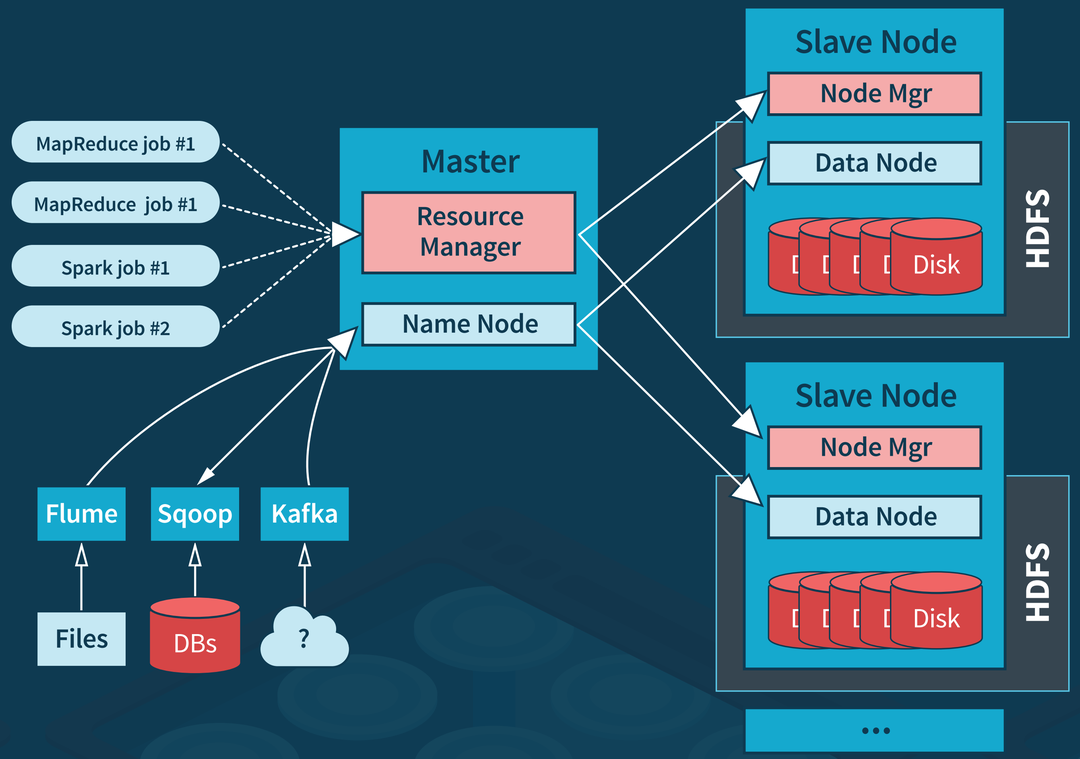

Figure 1

Event Streaming Reactive Applications

A Reactive application built with the Typesafe Reactive Platform and Spark would look like figure 2.

Working from the bottom up, all your services might be managed by Mesos, providing efficient allocation of cluster resources running on bare hardware or infrastructure as a service (IaaS), a public or private cloud environment, like Amazon EC2, Google Compute Engine, and others.

Play and Akka implement services like handling web requests, ingesting Reactive Streams of data from message queues and other sources (discussed below), interacting with databases, etc.

Akka streams data to Spark, whose streaming model works on time slices of event traffic. Spark is used to perform analytics, anything from running aggregations, like max N, average, etc., to machine-learning algorithms, like incrementally training recommendation engines, doing trend/threat analysis, etc. Spark may send data back to Akka. Spark and Akka may store data in local or distributed file systems or databases.

Additional batch-mode Spark jobs would run periodically to perform large-scale data analysis, like aggregations over long time frames, and ETL (extract, transform, and load) tasks like data cleansing and reformatting, and archiving.

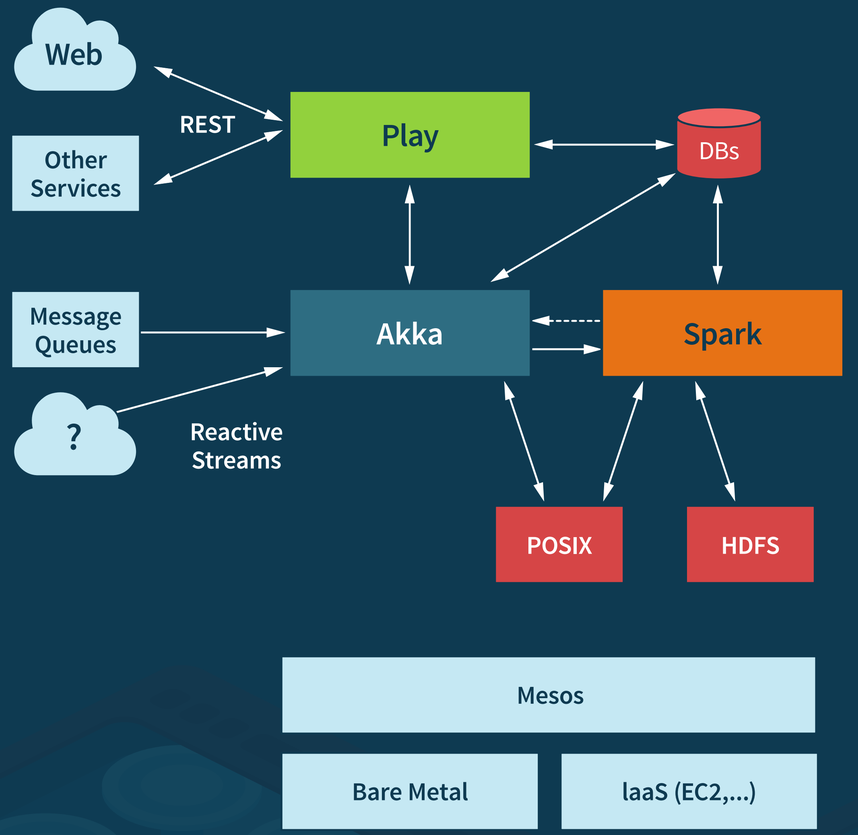

Figure 2

Akka and Spark Together

Let's look more closely at streaming data with a combination of Akka and Spark as shown in figure 3.

Reactive Streams support a mixed push and pull model combined with bounded queues. Bounded queues are important to prevent heap exhaustion when the stream consumer can't keep up with the rate of messages coming from the producer. When the consumer can handle the traffic rate, a normal push model is used. Otherwise, the rate of messages from the producer is controlled through feedback messages sent from the consumer to the producer.

An Akka application can use reactive streams to manage data ingested from various sources. It might also perform initial transformations, merge streams or route events to different consumers, etc.

For more advanced analytics, the data is streamed out of the Akka application over a socket to a Spark Streaming application (which doesn't support reactive streams).

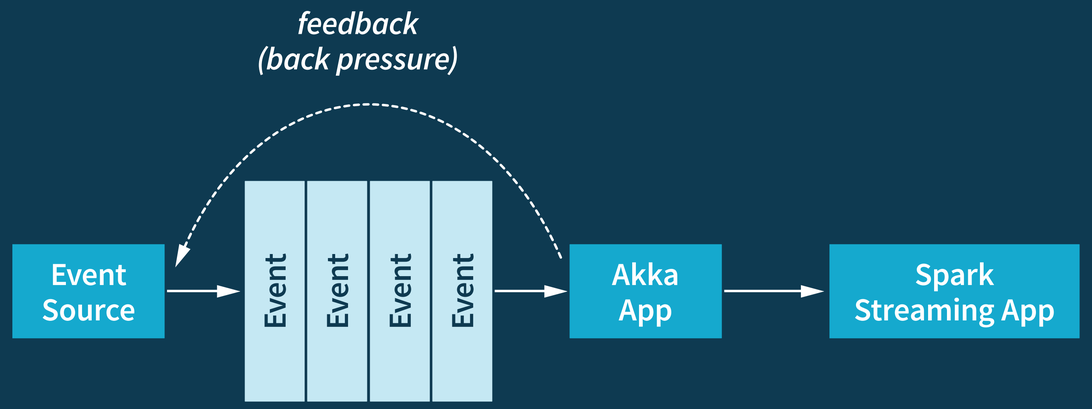

Figure 3

Akka and Spark Together

How Spark works.

Spark uses an in-memory data structure called an RDD, resilient, distributed dataset. RDDs are immutable, new ones are created as transformations are performed. RDDs are partitioned, each node in the cluster will have part of the RDD, as shown in the figure 4.

RDDs can be operated on in parallel, a transformation like mapping, filtering, etc. is performed simultaneously on all the partitions in parallel. RDDs are resilient, if a partition is lost due to a node crash, it can be reconstructed from the original sources.

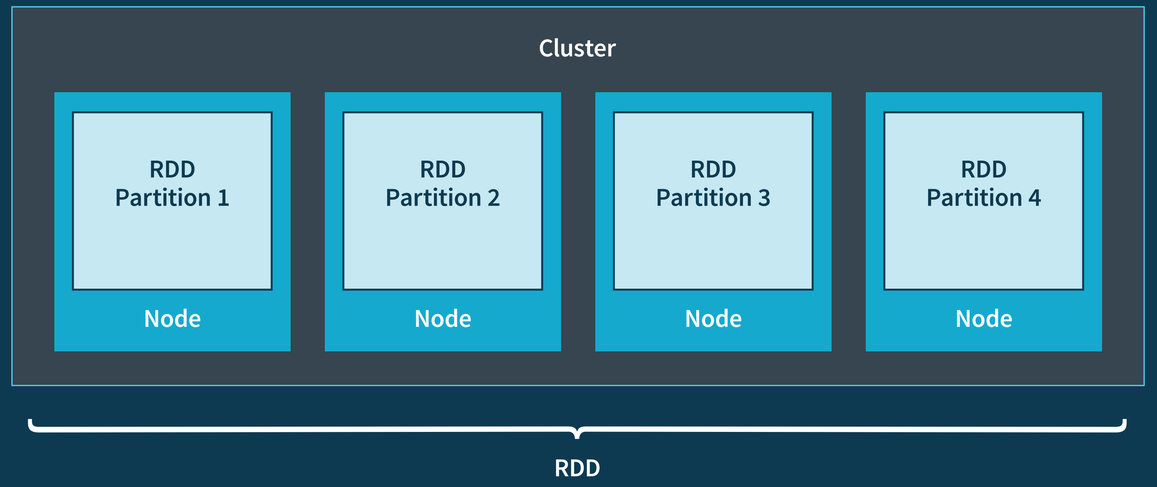

Figure 4

Akka and Spark Together

What happens inside Spark Streaming?

Spark's original implementation focused on batch-mode computations. However, Spark Streaming leverages RDDs using a clever DStream (discretized stream), where the stream of events is captured into batches, based on a user-configurable time slice (e.g., 1 second intervals). Each batch is stored in an RDD, enabling all the available RDD operations to be used on it, along with additional operations for working with windows of batches/RDDs.

Figure 5 illustrates the structure inside Spark Streaming.

The number of events in a given time-interval batch varies. The user configures the size of the sliding window and the number of batches to move the sliding window forward at each step. A window of three batches is shown and it slides one batch per interval. Common window aggregates include moving averages, such as the popular 50-day moving average for stock prices. Other common aggregates that are incrementally updated include max/min N (for some N - e.g., the top 10 trends on Twitter), averages, standard deviations, etc.

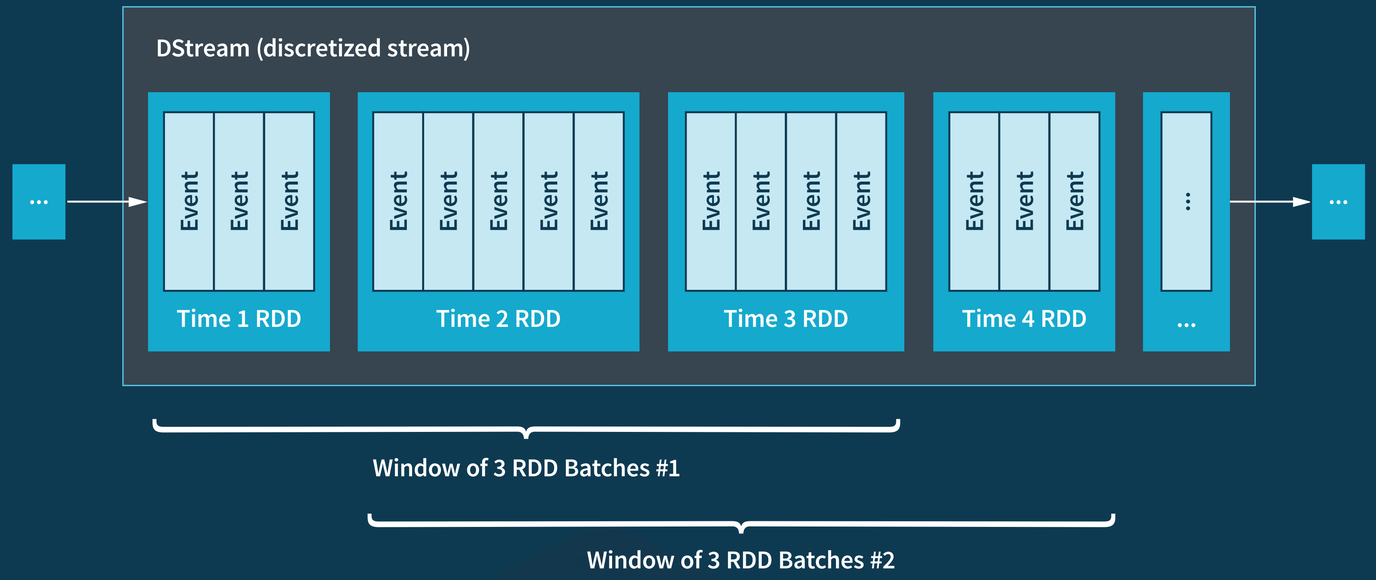

Figure 5

Spark and Scala

Functional programming provides a set of operations on data that are broadly applicable and compose together to build nontrivial transformations of data.

The Scala library implements these operations for data sets that fit in memory in a single JVM process or datasets that can be streamed through the process and transformed on the fly. Scala developers can write concise, expressive code, with high productivity.

Akka, Play, and Spark are implemented in Scala, so they enjoy these benefits. One way to think of the Spark API is that it scales up the idioms of the Scala library to the size of clusters, where the data is partitioned across the cluster in RDDs. Therefore, developers using Spark enjoy the same productivity benefits that other Scala developers enjoy.

Spark also uses Akka for some functionality. With its Actor model of computation and let it crash approach to resiliency, Akka provides many of the essentials tools for building distributed, fault-tolerant, event-based systems on the JVM.

*The Spark logo is the property of the Apache foundation.

Spark and Scala

Matei Zaharia, the creator of Spark and the co-founder of Databricks,
gave the following answer when asked why he chose Scala:

> "Quite a few people ask this question and the answer is pretty simple. When we started Spark, we had two goals — we wanted to work with the Hadoop ecosystem, which is JVM-based, and we wanted a concise programming interface similar to Microsoft's DryadLINQ (the first language-integrated big data framework I know of, that begat things like FlumeJava and Crunch). On the JVM, the only language that would offer that kind of API was Scala, due to its ability to capture functions and ship them across the network. Scala's static typing also made it much easier to control performance compared to, say, Jython or Groovy."

Matei Zaharia

Creator of Spark & Co-founder, Databricks

Spark and Scala

The Spark API is remarkably similar to the Scala Collections API.

To see the similarities between Spark and the Scala Collections API, let's look at a simple example, the famous Word Count algorithm, where we read in one or more documents, tokenize them into words, then count the occurrences of each word.

Listing 1 shows an implementation using the Scala Collections API, where we read all the text from a single file.

The comments provide the essential details. Ignoring the comments, note how concise this source code is!

```
import java.io._

object ScalaWordCount {
  def main(args: Array[String]) = {

    // The first command-line argument is the input file name.
    // We read each line, then "flatmap" the lines into words
    // by splitting each line on non-alphanumeric characters,
    // then output zero-to-many words, producing a stream of
    // words. Then we group-by the words to bring all the same
    // occurrences together, and finally map over the word-group
    // pairs and compute the size of the group.

    val wordsCounted = scala.io.Source.fromFile(args(0))
      .getLines.map(line => line.toLowerCase)
      .flatMap(line => line.split("""\W+""")).toSeq
      .groupBy(word => word)
      .map { case (word, group) => (word, group.size) }

    // Write the results to the file given as the second
    // command-line argument.
    val out = new PrintStream(new File(args(1)))
    wordsCounted foreach (word_count => out.println(word_count))
  }
}
```

Listing 1

Spark and Scala

The Spark implementation looks almost the same.

In listing 2 there are differences in handling of input and output, and in how the environment is set up and torn down, but the core logic is identical. It works for small data that fits in the memory of a single process, the Scala example, up to a large cluster, the Spark example.

Note that we didn't choose the most efficient implementations in each case. Both libraries offer pragmatic options for greater efficiency, but our purpose was to show how the ideas and even the specifics of the APIs translate across toolkits and across computation scales.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object SparkWordCount {

  def main(args: Array[String]) = {

    // Create a "SparkContext", which drives everything.
    val sc = new SparkContext("local", "Word Count (2)")

    // Except for how the files are read, the sequence of
    // API calls is identical.
    val wordsCounted = sc
      .textFile(args(0)).map(line => line.toLowerCase)
      .flatMap(line => line.split("""\W+"""))
      .groupBy(word => word)
      .map { case (word, group) => (word, group.size) }

    // Write the results and stop the context.
    wordsCounted.saveAsTextFile(args(1))
    sc.stop()
  }
}
```

Listing 2

Conclusions

Adding Apache Spark to the Typesafe Reactive Platform, including Scala, Akka, and Play, gives developers a comprehensive suite of tools for building Reactive applications with rich options for data analytics, all using similar, familiar APIs.

Are you interested in trying Spark? Please see our growing Typesafe Activator templates for Spark, especially the introductory Spark Workshop, which is our first Certified on Spark application.

See this Big Data Resources page to learn more about Typesafe's support for building Reactive, Big-Data applications.

Typesafe (Twitter: @Typesafe) is dedicated to helping developers build Reactive applications on the JVM. Backed by Greylock Partners, Shasta Ventures, Bain Capital Ventures and Juniper Networks, Typesafe is headquartered in San Francisco with offices in Switzerland and Sweden. To start building Reactive applications today, download Typesafe Activator.