



# APACHE SPARK


A Modern Tool for Big Data Applications

WHITE PAPER

## ABSTRACT

*What is Apache Spark, and why it is currently changing the world of Big Data?*

This white paper provides a high-level overview of Apache Spark including an explanation for its rapid ascent, performance and developer advantages over MapReduce, and an exploration of built-in functionality for application types involving streaming, machine learning, and Extract, Transform and Load (ETL).



**Apache Spark** is an open source clustering framework for batch and streaming processing.

## Introduction

### WHAT IS SPARK?

Apache Spark is an open source clustering framework for batch and streaming processing. The framework originated at the AMPLab in UC Berkeley in 2009, became an Apache project in 2013 and in 2014 became one of the organization's top priorities. It is currently supported by Databricks which was founded by many of the original creators of Spark.

At the heart of Spark is the concept of the Resilient Distributed Dataset (RDD), a programming abstraction that is an immutable collection of objects, able to be split across a computing cluster. Operations on the RDDs can therefore also be split, leading to highly parallelizable processing. RDDs can be created from simple text files, SQL databases, NoSQL stores (e.g. Cassandra, Riak), Hadoop InputFiles, or even programmatically. Much of the Spark Core API is built on the RDD concept, enabling traditional map and reduce functionality, but also providing built-in support for joining datasets, filtering, sampling, and aggregation.

In addition, Spark comes complete with support for many different Big Data applications, all built around the RDD model:

**Spark Streaming:** Although Spark is at heart a batch-mode processing framework, it offers a Spark Streaming mode that continuously collects data in 'microbatches', effectively providing streaming support for applications that do not require low latency responses. The Spark distribution ships with support for streaming data from Kafka, Flume, and Kinesis.

**MLLib [Machine Learning for Spark]:** This library brings several machine learning algorithm implementations to Spark for off-the-shelf use by Data Scientists, including Naive and Multinomial Bayesian models, clustering, collaborative filtering, and dimension reduction.

**GraphX:** This provides graph algorithm support for Spark, including a parallelized version of PageRank, triangle counts, and the ability to discover connected components.

**SparkSQL (formerly known as Shark):** SparkSQL provides common and uniform access to various different structured data sources (e.g. Hive, Avro, Parquet, ORC, JSON, JDBC/ODBC), allowing the user to write SQL queries that run across the Spark cluster, and to mix these data sources together without all the need for complicated ETL pipelines.

### WHY SPARK

Spark is often considered to be the next step for Big Data processing beyond Hadoop. While Hadoop's MapReduce functionality is spread across storage, Spark uses in-memory techniques to provide huge improvements in processing times, in some cases up to 100x the equivalent task in Hadoop. [In a benchmark performed by Reynold Xin, Parviz Deyhim, Xiangrui Meng, Ali Ghodsi, Matei Zaharia with Databricks in October 2014](#), Spark came top in the Daytona GraySort Benchmark, sorting a 100TB set in 23 minutes with 206 nodes. In comparison, the equivalent MapReduce record of sorting a similar-sized set is 72 minutes with 2100 nodes. Being able to work faster with an order of magnitude fewer resources definitely adds to Spark's appeal.

While the benchmarks are undoubtedly notable, that is not the entire reason why Spark

is currently riding high in the Big Data world. As well as being fast, it provides developer-friendly features such as the Spark Shell for interactive development, plus native bindings for Java, Scala, Python, and R. In particular, support for Python and R opens up the developer pool to data scientists who previously would have had to navigate the more unwieldy landscape of MapReduce in Java. In addition, because Spark supports batch and streaming modes within the same architecture, it means the code can be re-used between applications running in different modes, and both modes can also take advantage of the additional libraries such as MLlib, GraphX, or SparkSQL. For example, doing a join with streaming data against data from Cassandra in Spark is just as simple as joining data from two text files

## Spark on Yarn

Spark clusters can either be run in standalone mode, or on top of a resource manager such as YARN or Mesos. We highly recommend on using a resource manager, and recommend YARN over Mesos at this time.

### LIMITATIONS OF SPARK STANDALONE CLUSTERS

While running Spark in standalone mode is fine for experimentation and PoCs, any serious deployment of Spark applications is going to be limited by the standalone cluster manager's first-in-first-out (FIFO) scheduler, and the default behavior of allocating the entire cluster's resources to an application. For instance, a long-running ETL application could grab hold of all the available resources and prevent any other applications from running. By using YARN or Mesos, resources will be used more efficiently (especially if dynamic allocation is used).

### WHY YARN?

While Mesos is a very capable scheduler -- and in a green-field deployment it would be a fine choice -- we recommend YARN due to its maturity and also because many enterprises already have a Hadoop installation, so using YARN will leverage your existing resources without much more setup. In addition, it will be able to interact with the security and auditing solutions active in the Hadoop cluster.

### RUNNING SPARK ON YARN

- Firstly, you'll need to download a binary distribution of Spark that's built against the version of Hadoop/YARN you're currently using. This is available via the [Spark website](#).
- Ensure that either **YARN\_CONF\_DIR** or **HADOOP\_CONF\_DIR** environment variables are set and point to the directory which contains the configuration files for the Hadoop cluster.
- Submitting a job to the YARN cluster is quite straight-forward:



One of the goals of Spark is to make developing Big Data applications a more approachable enterprise for both traditional developers and data scientists.

```
SPARK_HOME/bin/spark-submit --class path.to.your.Class
--master [yarn-cluster|yarn-client] [options] <app jar> [app
options]
```

- There are various options that can be passed to the YARN cluster to indicate how many cores and how much memory should be allocated to the driver (the main Spark application) and the remote executors. Executors can also be dynamically allocated and released within the Spark application itself. (Note that it's up to the YARN scheduler if these requests are honored).
- Spark can run on YARN in two modes: yarn-cluster or yarn-client. In yarn-cluster, the driver application is run on the cluster as well as on the executors (and so the submitting process can exit while the driver / application continues running on YARN), whereas in yarn-client, the driver runs on the local machine. This makes sense when running interactive programs like the Spark-Shell, where the driver needs to run locally for interactivity to be possible.

## Developing with Spark

One of the goals of Spark is to make developing Big Data applications a more approachable enterprise for both traditional developers and data scientists. To this end, there are language bindings for Scala, Java, Python, and as of Spark 1.4, R. Also, Scala, Python, and R have an interactive shell available for prototyping and data exploration without having to resort to the compilation cycle, and in the end are still able to harness the full power of the cluster. No matter which language you choose, however, you will use a SparkContext object to create RDDs and perform operations on them.

### BATCH ANALYSIS OF LOGS WITH SPARK

In this example we will examine a log file in the context of the Spark shell. To open a new Spark shell, simply issue this at your command-line:

```
$> SPARK_HOME/bin/spark-shell
```

**SPARK\_HOME** is where you have installed Spark. You should then see a set of logging messages as the shell initializes, along with this:

```
Welcome to
SPARK version 1.4.1
...
15/08/04 14:29:10 INFO SparkILoop: Created spark context...
Spark context available as sc.
...
scala>
```

The Spark shell is now active and Scala code can be entered directly into the shell. First, we are going to want to access our data: in this case a log file stored in S3. Spark has a `textFile()` method that allows us to grab text files from the local filesystem, HDFS, or in this case S3:



As well as being fast, it provides developer-friendly features such as the Spark Shell for interactive development, plus native bindings for Java, Scala, Python, & R.

```
scala> val log = sc.textFile("hdfs:///demo/testdata.txt")
log: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at
textFile at <console>:21
```

We now have a string-based RDD, with each logical unit in the RDD being a line from the log. As we are only interested in errors and the IP addresses associated with them, we need to filter the RDD so we are only processing lines that contain the 'ERROR' string:

```
scala> val errors = log.filter( _.split("\\s+")(1) == "ERROR" )
```

Here, we are supplying an inline function to the RDD's filter() method, which splits the line on whitespace and then does a comparison of the second element of the returned array with the string "ERROR". This could also be written as:

```
scala> def errorFilter(line: String): Boolean = { line.split("\\s+")(1) == "ERROR" }
scala> val errors = log.filter(errorFilter(_))
```

In this example, it is more readable to have the function inline, but as processing becomes more complex, and to facilitate code re-use between batch and streaming operations, your applications will likely use defined functions and pass them into transformations and actions.

Also note that because filter() is a transformation, rather than an action, no processing is performed at this point. Spark is building up the graph of operations that need to be run on the cluster, but it will not execute them until an action is added to the graph.

```
scala> val errorsByIP = errors.map( x => (x.split(" ")(0), 1L))
```

This is another transformation and one that is a familiar concept from MapReduce. On the filtered RDD we map over the contents, creating a new RDD that has a key (the IP address being the first element) and a value - in this case we are going to count the number of times an IP address appears, so the value is just 1, and then we will reduce these mapped values to count up all the times the IP addresses appear:

```
scala> errorsByIP.reduceByKey( (x,y) => x + y ).cache
```

At last we come to an action. Once this line has been entered into the shell, computation will occur, executing the previous transformations on the dataset and also carrying out the actions reduceByKey() and cache(). reduceByKey(), as you expect, performs a reducing aggregation on a per-key basis - in this example, it is simply adding all the values found on a key together (so all the 1s in previous step will be added up - you may also see it written in examples as reduceByKey(\_+\_). Then, the cache() method is called to persist the result to memory (so it doesn't need to be recomputed).

```
scala> errorsByIP.saveAsTextFile("errors")
```

Finally, we save the RDD to a file, now containing a list of IP addresses and counts of errors.

## STREAMING LOG ANALYSIS

Now we'll take the concepts in the previous example and extend it from batch operation to streaming. Unfortunately, streaming applications cannot be run in the shell. The Scala code for

the streaming version is below, and can also be found at <https://github.com/mammothdataco/spark-demo>. This needs to be compiled and submitted to the Spark cluster.

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._

object StreamingDemo extends java.io.Serializable {
  def main() {
    val ssc = new StreamingContext(new SparkConf(
      setAppName("StreamingLoggingDemo"), Seconds(1))
      val streamingLog = ssc.socketTextStream("localhost",
      8888)
      val errors = streamingLog.filter( x => x.split("\\
s+")(1) == "ERROR" )
      val errorsByIPMap = errors.map( x => ((x.split("\\
s+")(0)), 1L))
      val errorsByIPAndWindow = errorsByIPMap.
      reduceByKeyAndWindow( {(x:Long, y:Long) => x + y}, Seconds(30),
      Seconds(5))
      ssc.start()
      ssc.awaitTermination()
    }
  }
}
```

Most of this should be familiar from the previous example, but instead of a SparkContext, we require a StreamingContext. This will include an argument to detail the granularity of our batches and to note that we will be getting new batches of data every second. Next, we will require a streaming source for our data. The socketTextStream listed is a text-based socketstream which we can use to send data down via telnet or netcat (Spark comes with built-in support for Kafka, Flume, and Kinesis for enterprise-grade streaming solutions). reduceByKey has been replaced by reduceByKeyAndWindow, a powerful operation available in streaming that allows you to perform windowing operations with ease. Here, it constructs a window of 30 seconds, updating the values on the keys every 5 seconds within the window. Finally, we start() the pipeline on the cluster and awaitTermination() to clean up after the pipeline is terminated.

To run this on your Spark cluster you'll need to submit the jar using spark-submit:

```
$> spark-submit --master yarn-cluster --class com.mammothdata.
demos.StreamingLoggingDemo MammothSparkWhitePaper.jar
```

(Above is specifically for YARN. For a standalone cluster, you'll need to pass in the URI of the Spark Master).

# Machine Learning with Spark

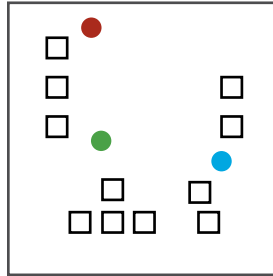
In addition to the building blocks of batch and streaming analysis, Spark also includes a scalable distributed machine learning library, Spark MLlib. It contains parallel algorithms for generating statistics, classification and regression models, pattern mining, collaborative filtering, and clustering. These algorithms can be brought into your batch and streaming applications with very little overhead to development.

## K-MEANS CLUSTERING WITH SPARK

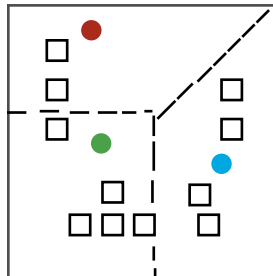
K-Means clustering is an unsupervised algorithm which splits a dataset into a number of clusters (k) based on a notion of similarity between points. It is often applied to real-world data to obtain a picture of structure hidden in large datasets, for example, identifying location clusters or breaking down sales into distinct purchasing groups.

K-Means Algorithm Explained

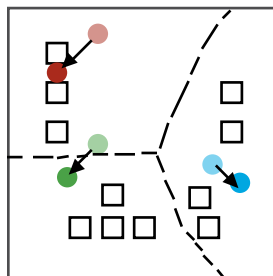
Parallel algorithms can be brought into your batch and streaming applications with very little overhead to development.



1. k initial "means" (in this case k=3) are randomly generated within the data domain (shown in color).



2. k (3) clusters are created by comparing each data point to the closest mean.





3. The centroid of each of these clusters is found, and these are used as new means. New clusters are formed via observing the closest data points to these new mean as shown in Step two. The process is repeated until the means converge (or until we hit our iteration limit)

Using K-Means clustering in Spark is incredibly simple, but also flexible and powerful. In the example below, we build a model against the logging data we saw previously:

```
import org.apache.spark.mllib.feature._

import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}

import org.apache.spark.mllib.linalg.Vector

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.rdd.RDD

object KmeansDemo extends java.io.Serializable {

  def featurize(s: String): Vector = {

    val tf = new HashingTF(1000)

    val bigram = s.sliding(2).toSeq

    tf.transform(bigram)

  }

  def build_model(text: RDD[String], numClusters: Int,
numIterations: Int ): KMeansModel = {

    // Caches the vectors since it will be used many times by
KMeans.

    val vectors = text.map(featurize).cache

    vectors.count() // Calls an action to create the cache.

    KMeans.train(vectors, numClusters, numIterations)

  }

  def main(args: Array[String]) {

    val conf = new SparkConf().setAppName("KmeansDemo")

    val sc = new SparkContext(conf)
```

```

val log = sc.textFile("S3URL")

val model = build_model()(log, 10, 100)

val predictedCluster = model.predict(featurize("This is a
test string"))

}

}

```

Almost all the models in MLlib operate on feature vectors, so before we can create a k-means model, we need to convert our dataset of strings into feature vectors. We do this in the featurize method, first forming a bi-gram collection of characters in the string using a sliding window, and then passing that into HashingTF, which produces a 1000-dimension feature vector.

In the `build_model` method, we first map featurize over our strings, and then cache the resulting feature vector RDD. This is an important step for performance as k-means is an iterative algorithm, so recalculating this RDD for each iteration would be an expensive process.

Finally, we train a k-means model by passing in the feature vector RDD, our value of k (10), and how many iterations of the algorithm we wish to run (here, we're running 100).

Having created the model, we can then use it to predict in which cluster future points in the dataset will fall, for example whether a customer will fall into a high-value or low-value group, or whether an incoming network trace should be marked as suspicious.

A powerful feature of Spark is the ability to train models such as k-means and then save them to storage (e.g. HDFS) to re-read in at another point. Or even by a different Spark application. Saving a model is as simple as:

```
model.save(sc, "hdfs:///models/kmeans")
```

And a model can be loaded in a similar manner:

```
val model = KMeansModel.load(sc, "hdfs:///models/kmeans")
```

This allows a Data Scientist to experiment and fine-tune the model in a Python Spark Shell before saving it out to disk, where it can be picked up by a production application developed in Scala, providing opportunities for upgrading and further fine-tuning as the dataset is explored in more detail and with simple operational deployment.



The fundamentals of software development still apply in the Big Data world - Spark can be unit-tested and integration-tested, and code should be reused between streaming and batch jobs wherever possible.

# Spark SQL

Spark SQL (previously known as Shark) is a component of Spark that enables structured data processing of multiple heterogeneous data sources in a common manner (e.g. Hive, JSON files, JDBC connections, etc). Spark provides the dataframe abstraction - a distributed collection of data organized into named columns (similar to a table in a SQL system). All operations on a dataframe are converted into RDD operations by Spark's Catalyst query planner and optimizer and then run on the cluster as normal.

## SPARK SQL OPERATIONS

In this example, we'll load data from a JSON file. Spark will automatically infer a schema from the structure of the JSON data and allow us to perform operations and transformations in a similar manner to working with RDDs (meanwhile, behind the scenes, the Catalyst optimiser will turn these Dataframe operations into RDD operations that execute across the cluster).

First, we'll create the dataframe inside a standard spark-shell. When the shell starts up, it creates a SQLContext object for you called sqlContext.

```
scala> val df = sqlContext.read.json("hdfs:///demo/dataframe.json")
```

We can then show the content of the dataframe:

```
scala> df.show()
```

```
+---+-----+
|age|  name|
+---+-----+
| 35|  Paul|
| 33|Quentin|
| 30| Leigh|
| 28|  Pip  |
| 32|  Matt|
| 30|  Maen|
+---+-----+
```

We can interrogate the dataframe to obtain the schema that Spark has created:

```
scala> df.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

Selection on columns:



## Additional Resources

Mammoth Data: <http://mammothdata.com>

\* Apache Spark: <http://spark.apache.org>

\* Databricks: <http://databricks.com>

\* Apache YARN: <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

\* Apache Mesos: <http://mesos.apache.org>

\* Apache Spark on Hortonworks Data Platform: <http://hortonworks.com/hadoop/spark/>

\* Apache Spark on Cloudera: <http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/spark.html>

\* Apache Spark on MapR: <https://www.mapr.com/products/apache-spark>

```
scala> df.select("name").show()
```

name
Paul
Quentin
Leigh
Pip
Matt
Maen

A more complex example which counts the occurrences of ages in the data and orders them in a descending (by age) manner:

```
scala> df.groupBy("age").count().orderBy("age").show()
```

age	count
28	1
30	2
32	1
33	1
35	1

Finally, we filter on people 35 or older and save back the results to disk in JSON (though you could save to Parquet, Hive, ORC, Avro, or through a JDBC connection):

```
scala> df.filter(df("age") >= 35).write.format("json").save("eligible torunfor president")
```

To wrap up this white paper on Apache Spark we have included [9 Tips for Best Practices with Spark](#) to get you started with this extremely useful tool.

If you have any questions regarding this white paper or a project incorporating Apache Spark, please check out our About Us page at the end of this document to contact us.

## 9 TIPS FOR BEST PRACTICES WITH SPARK

1. Spark is written in Scala, so new features will be first available for Scala (and Java). Python and R bindings can lag behind on new features from version to version.
2. If you are developing Spark in Java, use Java 8 or above if possible. The new lambda functionality in Java 8 removes a lot of verbosity required in your code.
3. The fundamentals of software development still apply in the Big Data world - Spark can be unit-tested and integration-tested, and code should be reused between streaming and batch jobs wherever possible.
4. When Spark is transferring data over the network, it needs to serialize objects into a binary form. This can have an effect on performance when shuffling or on other operations that require large amounts of data to be transferred. To ameliorate this, first try to make sure that your code is written in a way that minimizes the amount of shuffling that may occur (e.g. only use groupByKey as a last resort, preferring instead to use actions like reduceByKey which perform aggregation as in-place as possible). Second, consider using Kryo instead of java.io.Serializable for your objects, as it has a more compact binary representation than the standard Java serializer, and is also faster to compress or decompress. For further performance, especially when dealing with billions of objects, you can register classes with the Kryo serializer at start-up, saving more precious bytes.
5. Use connection pools instead of creating dedicated connections when connecting to external data sources - e.g. if you are writing elements from an RDD into a Redis cluster, you might be surprised if it attempts to open 10 million connections to Redis when running on production traffic.
6. Use Spark's checkpointing features when running streaming applications to ensure recovery from failures. Spark can save checkpoints to local files, HDFS, or S3.
7. With larger datasets (>200Gb), garbage collection on the JVM Spark runs may become a performance issue. In general, switching to the G1 GC over the default ParallelGC will ultimately be more performant. Although, some tuning will be required according to the details of your dataset and application (a process that Mammoth Data can easily assist with).
8. If possible, use dataframes over RDDs for developing new applications. While dataframes are in the Spark SQL package rather than Spark Core, Databricks has indicated that they will be dedicating significant resources to improving the Catalyst optimizer for generating RDD code. By adopting the dataframe approach of development, your application is likely to benefit from any optimizer improvements in the upcoming development cycles.
9. Remember, Spark Streaming is not a pure streaming architecture. If the microbatches do not provide a low enough latency for your processing, you may need to consider a different framework, e.g. Storm, Samza, or Flink.



## ABOUT US

Mammoth Data is a Big Data consulting firm specializing in Hadoop®, NoSQL databases, and designing modern data architectures that enable companies to become data-driven. By combining cutting-edge technologies with a high-level strategy, we are able to craft systems that capture, organize and turn unstructured information into real business intelligence.

Mammoth Data was founded as Open Software Integrators in 2008 by open source software developer, evangelist and now president, Andrew C. Oliver. Mammoth Data is headquartered in downtown Durham, North Carolina.

### MAIN OFFICE

345 W. MAIN ST. SUITE 201  
DURHAM, NC 27701

(919) 321-0119  
info@mammothdata.com  
@mammothdataco  
[mammothdata.com](http://mammothdata.com)