Why Apache Spark is a Crossover Hit for Data Scientists

by Sean Owen | March 03, 2014 | 10 comments

Spark is a compelling multi-purpose platform for use cases that span investigative, as well as operational, analytics.

Data science is a broad church. I am a data scientist — or so I've been told — but what I do is actually quite different from what other "data scientists" do. For example, there are those practicing "investigative analytics" and those implementing "operational analytics." (I'm in the second camp.)

Data scientists performing investigative analytics use interactive statistical environments like **R** to perform ad-hoc, exploratory analytics in order to answer questions and gain insights. By contrast, data scientists building operational analytics systems have more in common with engineers. They build software that creates and queries machine-learning models that operate at scale in real-time serving environments, using systems languages like C++ and Java, and often use several elements of an enterprise data hub, including the **Apache Hadoop** ecosystem.

And there are subgroups within these groups of data scientists. For example, some analysts who are proficient with R have never heard of **Python** or **scikit-learn**, or vice versa, even though both provide libraries of statistical functions that are accessible from a **REPL** (Read-Evaluate-Print Loop) environment.

A World of Tradeoffs

It would be wonderful to have one tool for everyone, and one architecture and language for investigative as well as operational analytics. If I primarily work in Java, should I really need to know a language like Python or R in order to be effective at exploring data? Coming from a conventional data analyst background, must I understand **MapReduce** in order to scale up computations? The array of tools available to data scientists tells a story of unfortunate tradeoffs:

- R offers a rich environment for statistical analysis and machine learning, but it has some rough edges when performing many of the data processing and cleanup tasks that are required before the real analysis work can begin. As a language, it's not similar to the mainstream languages developers know.
- Python is a general purpose programming language with excellent libraries for data analysis like Pandas and scikit-learn. But like R, it's still limited to working with an amount of data that can fit on one machine.
- It's possible to develop distributed machine learning algorithms on the classic MapReduce computation framework in Hadoop (see **Apache Mahout**). But MapReduce is notoriously low-level and difficult to express complex computations in.
- **Apache Crunch** offers a simpler, idiomatic Java API for expressing MapReduce computations. But still, the nature of MapReduce makes it inefficient for iterative computations, and most machine learning algorithms have an iterative component.

And so on. There are both gaps and overlaps between these and other data science tools. Coming from a background in Java and Hadoop, I do wonder with envy sometimes: why can't we have a nice REPL-like investigative analytics environment like the Python and R users have? That's still scalable and distributed? And has the nice distributed-collection design of Crunch? And can equally be used in operational contexts?

Common Ground in Spark

These are the desires that make me excited about **Apache Spark**. While discussion about Spark for data science has mostly noted its ability to keep data resident in memory, which can speed up iterative machine learning workloads compared to MapReduce, this is perhaps not even the big news, not to me. It does not solve every problem for everyone. However, Spark has a number of features that make it a compelling crossover platform for investigative as well as operational analytics:

- Spark comes with a machine-learning library, **MLlib**, albeit bare bones so far.
- Being **Scala**-based, Spark embeds in any JVM-based operational system, but can also be used **interactively in a REPL** in a way that will feel familiar to R and Python users.
- For Java programmers, Scala still presents a learning curve. But at least, any Java library can be used from within Scala.
- Spark's **RDD (Resilient Distributed Dataset)** abstraction resembles Crunch's **PCollection**, which has proved a useful abstraction in Hadoop that will already be familiar to Crunch developers. (Crunch can even be used **on top of Spark**.)
- Spark imitates Scala's collections API and functional style, which is a boon to Java and Scala developers, but also somewhat familiar to developers coming from Python. Scala is also a **compelling choice for statistical computing**.
- Spark itself, and Scala underneath it, are not specific to machine learning. They provide APIs supporting related tasks, like data access, **ETL**, and integration. As with Python, the entire data science pipeline can be implemented within this paradigm, not just the model fitting and analysis.
- Code that is implemented in the REPL environment can be used mostly as-is in an operational context.

- Data operations are transparently distributed across the cluster, even as you type.

Spark, and MLlib in particular, still has a lot of growing to do. For example, the project needs optimizations, fixes, and deeper integration with **YARN**. It doesn't yet provide nearly the depth of library functions that conventional data analysis tools do. But as a best-of-most-worlds platform, it is already sufficiently interesting for a data scientist of any denomination to look at seriously.

In Action: Tagging Stack Overflow Questions

A complete example will give a sense of using Spark as an environment for transforming data and building models on Hadoop. The following example uses a dump of data from the popular **Stack Overflow** Q&A site. On Stack Overflow, developers can ask and answer questions about software. Questions can be tagged with short strings like "**java**" or "**sql**". This example will build a model that can suggest new tags to questions based on existing tags, using the **alternating least squares** (ALS) recommender algorithm; questions are "users" and tags are "items".

Getting the Data

Stack Exchange provides **complete dumps of all data**, most recently from January 20, 2014. The data is provided as a **torrent** containing different types of data from Stack Overflow and many sister sites. Only the file stackoverflow.com-Posts.7z needs to be downloaded from the torrent.

This file is just a **bzip**-compressed file. Spark, like Hadoop, can directly read and split some compressed files, but in this case it is necessary to uncompress a copy on to **HDFS**. In one step, that's:

```
bzcat stackoverflow.com-Posts.7z | hdfs dfs -put - /user/srowen/Posts.xml
```

Uncompressed, it consumes about 24.4GB, and contains about 18 million posts, of which 2.1 million are questions. These questions have about 9.3 million tags from approximately 34,000 unique tags.

Set Up Spark

Given that Spark's integration with Hadoop is relatively new, it can be time-consuming to **get it working manually**. Fortunately, CDH hides that complexity by integrating Spark and managing setup of its processes. Spark can be **installed separately** with CDH 4.6.0, and is **included** in CDH 5 Beta 2. This example uses an **installation of CDH 5 Beta 2**.

This example uses MLlib, which uses the **jblas** library for linear algebra, which in turn calls native code using **LAPACK** and Fortran. At the moment, it is necessary to manually install the Fortran library dependency to enable this. The package is called libgfortran or libgfortran3, and should be available from the standard package manager of major Linux distributions. For example, for **RHEL 6**, install it with:

```
sudo yum install libgfortran
```

This must be installed on all machines that have been designated as Spark workers.

Log in to the machine designated as the Spark master with ssh. It will be necessary, at the moment, to ask Spark to let its workers use a large amount of memory. The code in MLlib that is used in this example, in version 0.9.0, has a **memory issue**, one that is already fixed for the next release. To configure for more memory and launch the shell:

```
export SPARK_JAVA_OPTS="-Dspark.executor.memory=8g"
spark-shell
```

Interactive Processing in the Shell

The shell is the Scala REPL. It's possible to execute lines of code, define methods, and in general access any Scala or Spark functionality in this environment, one line at a time. You can paste the following steps into the REPL, one by one.

First, get a handle on the Posts.xml file:

```
val postsXML = sc.textFile("hdfs:///user/srowen/Posts.xml")
```

In response the REPL will print:

```
postsXML: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at :12
```

The text file is an RDD (Resilient Distributed Dataset) of Strings, which are the lines of the file. You can query it by calling methods of the **RDD class**. For example, to count the lines:

```
postsXML.count
```

This command yields a great deal of output from Spark as it counts lines in a distributed way, and finally prints 18066983.

The next snippet transforms the lines of the XML file into a collection of (questionID,tag) tuples. This demonstrates Scala's functional programming style, and other quirks. (Explaining them is out of scope here.) RDDs behave like Scala collections, and expose many of the same methods, like map:

```
val postIDTags = postsXML.flatMap { line =>
  // Matches Id="..." ... Tags="..." in  line
  val idTagRegex = "Id=\"(\\d+)\".+Tags=\"([^\"]+)\"".r

  // // Finds tags like <TAG> value from above
  val tagRegex = "&lt;([^&]+)&gt;".r

  // Yields 0 or 1 matches:
  idTagRegex.findFirstMatchIn(line) match {
    // No match -- not a  line
    case None => None
    // Match, and can extract ID and tags from m
    case Some(m) => {
      val postID = m.group(1).toInt
      val tagsString = m.group(2)
      // Pick out just TAG matching group
      val tags = tagRegex.findAllMatchIn(tagsString).map(_.group(1)).toList
      // Keep only question with at least 4 tags, and map to (post,tag) tuples
      if (tags.size >= 4) tags.map((postID,_)) else None
     }
  }
  // Because of flatMap, individual lists will concatenate
  // into one collection of tuples
}
```

(You can copy the source for the above from **here**.)

You will notice that this returns immediately, unlike previously. So far, nothing requires Spark to actually perform this transformation. It is possible to force Spark to perform the computation by, for example, calling a method like count. Or Spark can be told to compute and persist the result through **checkpointing**, for example.

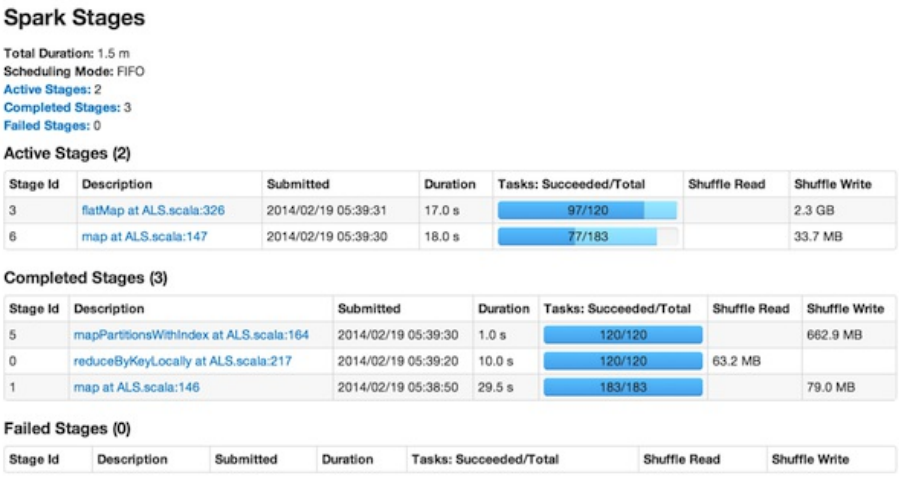
The MLlib implementation of ALS operates on numeric IDs, not strings. The tags ("items") in this data set are strings. It will be sufficient here to hash tags to a nonnegative integer value, use the integer values for the computation, and then use a reverse mapping to translate back to tag strings later. Here, a hash function is defined since it will be reused shortly.

```
def nnHash(tag: String) = tag.hashCode & 0x7FFFFF
var tagHashes = postIDTags.map(_._2).distinct.map(tag =>(nnHash(tag),tag))
```

Now, you can convert the tuples from before into the format that the ALS implementation expects, and the model can be computed:

```
import org.apache.spark.mllib.recommendation._
// Convert to Rating(Int,Int,Double) objects
val alsInput = postIDTags.map(t => Rating(t._1, nnHash(t._2), 1.0))
// Train model with 40 features, 10 iterations of ALS
val model = ALS.trainImplicit(alsInput, 40, 10)
```

This will take minutes or more, depending on the size of your cluster, and will spew a large amount of output from the workers. Take a moment to find the Spark master web UI, which can be found from Cloudera Manager, and will run by default at http://[master]:18080. There will be one running application. Click through, then click "Application Detail UI". In this view it's possible to monitor Spark's distributed execution of lines of code in ALS.scala:



Spark Stages

Total Duration: 1.5 m
Scheduling Mode: FIFO
Active Stages: 2
Completed Stages: 3
Failed Stages: 0

Active Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
3	flatMap at ALS.scala:326	2014/02/19 05:39:31	17.0 s	97/120		2.3 GB
6	map at ALS.scala:147	2014/02/19 05:39:30	18.0 s	77/183		33.7 MB

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
5	mapPartitionsWithIndex at ALS.scala:164	2014/02/19 05:39:30	1.0 s	120/120		662.9 MB
0	reduceByKeyLocally at ALS.scala:217	2014/02/19 05:39:20	10.0 s	120/120	63.2 MB	
1	map at ALS.scala:146	2014/02/19 05:38:50	29.5 s	183/183		79.0 MB

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

When it is complete, a factored matrix model is available in Spark. It can be used to predict question-tag associations by "recommending" tags to questions. At this early stage of MLlib's life, there is not even a proper recommend method yet, that would give suggested tags for a question. However it is easy to define one:

```
def recommend(questionID: Int, howMany: Int = 5): Array[(String, Double)] = {
  // Build list of one question and all items and predict value for all of them
  val predictions = model.predict(tagHashes.map(t => (questionID, t._1)))
  // Get top howMany recommendations ordered by prediction value
  val topN = predictions.top(howMany)(Ordering.by[Rating, Double](_.rating))
  // Translate back to tags from IDs
  topN.map(r => (tagHashes.lookup(r.product)(0), r.rating))
}
```

And to call it, pick any question with at least four tags, like "**How to make substring-matching query work fast on a large table?**" and get its ID from the URL. Here, that's 7122697:

```
recommend(7122697).foreach(println)
```

This method will take a minute or more to complete, which is slow. The lookups in the last line are quite expensive since each requires a distributed search. It would be somewhat faster if this mapping were available in memory. It's possible to tell Spark to do this:

```
tagHashes = tagHashes.cache
```

Because of the magic of Scala closures, this does in fact affect the object used inside the recommend method just defined. Run the method call again and it will return faster. The result in both cases will be something similar to the following:

```
(sql,0.17745152481166354)
(database,0.13526622226672633)
(oracle,0.1079428707621154)
(ruby-on-rails,0.06067207312463499)
(postgresql,0.050933613169706474)
```

(Your result will not be identical, since ALS starts from a random solution and iterates.) The original question was tagged "postgresql", "query-optimization", "substring", and "text-search". It's reasonable that the question might also be tagged "sql" and "database". "oracle" makes sense in the context of questions about optimization and text search, and "ruby-on-rails" often comes up with PostgreSQL, even though these tags are not in fact related to this particular question.

Something for Everyone

Of course, this example could be more efficient and more general. But for the practicing data scientists out there — whether you came in as an R analyst, Python hacker, or Hadoop developer — hopefully you saw something familiar in different elements of the example, and have discovered a way to use Spark to access some benefits that the other tribes take for granted.

Learn more about **Spark's role in an EDH**, and join the discussion in our brand-new **Spark forum**.

Sean is Director of Data Science for EMEA at Cloudera, helping customers build large-scale machine learning solutions on Hadoop. Previously, Sean founded Myrrix Ltd, producing a real-time recommender and clustering product

evolved from Apache Mahout. Sean was primary author of recommender components in Mahout, and has been an active committer and PMC member for the project. He is co-author of Mahout in Action.

Filed under:

- **Data Science**
- **Spark**
- **Use Case**

10 Responses

ANTONIO PICCOLBONI / MARCH 03, 2014 / 3:15 PM

Au contraire, R has exceptional tools for the manipulation of data prior to analysis, see
http://vita.had.co.nz/papers/tidy-data.pdf
Some of those are being "ported" to mapreduce, see this project **https://github.com/RevolutionAnalytics/plyrmr/**

JOWANZA JOSEPH / MARCH 03, 2014 / 3:27 PM

Thanks for posting. I have been messing around with SparkR, an R integration with Spark for a few days and I like it. I have great amount of hope for this framework.

FLORIAN LEITNER / MARCH 06, 2014 / 11:51 PM

You guys seem to never have heard of IPythonIPython, either. Indeed, it is very easy to do distributed computing with Python these days. So, at least two of your opening arguments are quite a bit moot.

SEAN OWEN / MARCH 09, 2014 / 7:57 PM

Thank you for the comments. The comments in the post are indeed generalizations and simplifications. It is not as if nobody has tried to clean data in R or parallelize Python. I believe data manipulation in R remains quite difficult compared to a modern programming language. IPython, which I have certainly heard of, provides a parallelization primitive among other things but does not compare seriously to Hadoop for distributed computation. To each his/her own tool but I would suggest these are exactly the audiences that should be looking at Spark.

MAJID ALDOSARI / MARCH 12, 2014 / 1:51 PM

Parallel programming and distributed data is more about paradigms than particular association with a programming language. It's possible to have a framework accessible from many languages.

What prompted me to say this is when you said Python is limited to processing in-memory. I'm sure I can search python interfaces (or even python native) to distributed data stores and processing.

SEAN OWEN / MARCH 13, 2014 / 7:45 AM

On re-reading, I agree that this is too dismissive of Python-related tools for distributed computation. To Python plus scikit, for example, it does require adding another different platform like IPython. And I suspect that for most people this paradigm is different again from where your other data and computation lives, like Hadoop. I would retreat to a weaker point then — and this is up for debate — that distributing the Python world is relatively hard to access outside of its specialist world, compared to, say, Hadoop. Really, the point was not that Python or its associated tools are deficient, but to create a simplistic sketch of the either/or tradeoffs your average organization faces when engaging these tools.

ALTON ALEXANDER / MARCH 13, 2014 / 11:37 AM

Very impressed with what I've seen from spark sofar. Looking forward to adding it to my arsenal including R and Python.

ALEX MCLINTOCK / MAY 09, 2014 / 3:37 AM

Hi Sean, I heard you give this talk last night at BigData London. I am wondering how SAS fits in to this picture. It seems to me that some people like SAS and use it a bit like R…. and that Cloudera and SAS talk to each other…. presumably through the Hive ODBC driver. How do I find out more?

> **JUSTIN KESTELYN (@KESTELYN)** / MAY 09, 2014 / 10:37 AM
>
> Alex,
>
> This post may help:
>
> **http://blog.cloudera.com/blog/2013/05/how-the-sas-and-cloudera-platforms-work-together/**

NEAL MCBURNETT / JANUARY 25, 2015 / 10:34 AM

Great post – thank you.
Python is indeed an awesome language for data scientists. Thankfully, Spark now supports pyspark – an official API binding to regular CPython 2.6 and up. This seems quite relevant re: the python discussion in the comments.

pyspark augments the other amazing multi-paradigm distributed processing opportunities provided by IPython parallel (**http://ipython.org/ipython-doc/2/parallel/index.html**) as well as scikit, and the wonderful world of literate

programming via IPython Notebooks, etc.

Leave a comment

Name REQUIRED

Email REQUIRED
(WILL NOT BE PUBLISHED)

Website

Comment

Leave Comment

Prove you're human! *

× one = 4

cloudera

Cloudera, Inc.
1001 Page Mill Road Bldg 2
Palo Alto, CA 94304

www.cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488