Analyzing Apache access logs with Spark and Scala (a tutorial)

By Alvin Alexander. Last updated: February 15, 2021

Note: I originally wrote this article many years ago using Apache Spark 0.9.x. Hopefully the content below is still useful, but I wanted to warn you up front that it is old.

I want to analyze some Apache access log files for this website, and since those log files contain hundreds of millions (billions?) of lines, I thought I'd roll up my sleeves and dig into Apache Spark to see how it works, and how well it works. The short story is that I used Hadoop several

years ago, and I found the transition to Spark to be easy. Here are my notes.

Note: If you already know how to use Spark and just want to see how to process Apache access log records, I wrote this shorter article on How to generate a list of URLs sorted by their hit count from your Apache access log files.

Installation

To install Spark, just follow the notes at

named and default parameters

pass one function to another

pass a function to a function (swing)

files

open and read files

shell script example

command line and scripts

read command line arguments

execute (exec) system commands

prompting a user, reading input

make scripts run faster

show more methods in repl

show more info on classes/objects in repl

paste multiline commands in repl

database

jdbc connection, select

actors and concurrency

akka 'hello world'

akka ping-pong example

stop/quit an actor

stop actor and shut down

akka futures

akka ask, future, await, timeout

parallel collections, .par, and performance

akka remote example

akka remote - objects as messages

idioms

lift framework form examples

xml

create xml literal

generate dynamic xml

xml - pretty printing

xml - save to file

xml - serialize, deserialize

xml - load a file

xml - load a url

xml - xpath searching

searching xmlns namespaces, xpath

xml - extract data from nodes

xml - extract data from arrays

xml - parsing, tags

xml - using match expressions

xml - many examples

build, testing, and debugging

sbt documentation (pdf)

show sbt history

scalatest - installing

scalatest - writing tdd tests

scalatest - writing bdd tests

scalatest - given/when/then with bdd

scalatest - test suite

scalatest - expected, actual

scalatest - mark test as pending

scalatest - testing exceptions

scalatest - tagging tests

scalatest - disabling tests

scalatest - mock objects

scalatest - running in eclipse

categories

alaska (25)
android (138)
best practices (63)
career (50)
colorado (21)
cvs (27)
design (33)
drupal (120)
eclipse (6)
funny (3)
gadgets (108)
git (15)
intellij (4)
java (429)
jdbc (26)
swing (74)
jsp (9)
latex (26)
linux/unix (289)
mac os x (315)
mysql (54)
ooa/ood (11)
perl (156)
php (97)
postgresql (17)
programming (43)
ruby (56)
scala (640)
sencha (23)
servlets (10)
technology (84)
testing (13)
uml (24)
zen (47)

https://spark.incubator.apache.org/docs/latest/. As they say, "All you need to run it is to have Java to installed on your system PATH, or the JAVA_HOME environment variable pointing to a Java installation." I assume that's true; I have both Java and Scala installed on my system, and Spark installed fine.

As their notes say, you build Spark on your system with SBT by using this command in your Spark installation directory:

```
$ sbt/sbt assembly
```

At this point you might want to go make some coffee, tea, a large meal, walk the dog, or maybe go to a meeting, because it takes a while to build, especially on my old 2008 iMac.

Once that build process finishes, I recommend trying some of the examples at these pages:

- https://spark.incubator.apache.org/docs/latest/quick-start.html
- https://spark.incubator.apache.org/docs/latest/scala-programming-guide.html

For instance, just run the line count test on the *README.md* file to make sure it works as expected:

```
$ ./bin/spark-shell

scala> val textFile = sc.textFile("README.md")    // create a referen
scala> textFile.count                             // count the number
scala> textFile.first                             // print the first l
```

If that works, you're ready to dig in.

Getting started using my Apache logfile parser with Spark

I want to use my Apache logfile parser code, so I packaged it as a jar file named *AlsApacheLogParser.jar*. I quickly found out that Spark is a little weird about using jar files. For instance, I can't use `:cp` to include a jar file into the Spark REPL like I can with the regular Scala REPL. In short, the solution is to start Spark like this from your *nix command line:

```
// this works
$ MASTER=local[4] SPARK_CLASSPATH=AlsApacheLogParser.jar ./bin/spark-shell
```

Also, despite what you might read, these commands do not work with Spark 0.9:

```
// does not work
$ MASTER=local[4] ADD_JARS=AlsApacheLogParser.jar ./bin/spark-shell

// does not work
spark> :cp AlsApacheLogParser.jar
```

With my jar file properly loaded, I now paste in these lines of code into the Spark REPL to create an `AccessLogParser` instance:

```
import com.alvinalexander.accesslogparser._
val p = new AccessLogParser
```

Now I read my *accesslog.small* file, just like the *README*:

```
scala> val log = sc.textFile("accesslog.small")
14/03/09 11:25:23 INFO MemoryStore: ensureFreeSpace(32856) called with curMem=0, maxMem=309225C
14/03/09 11:25:23 INFO MemoryStore: Block broadcast_0 stored as values to memory (estimated siz
log: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at <console>:15

scala> log.count
(a lot of output here)
res0: Long = 100000
```

Now I have a few more basics working, including getting my Apache access log file parser library loaded into the REPL and creating the `log` reference, so it's time to start doing some analysis.

Analyzing Apache access log data

Because my parser returns an `Option[AccessLogRecord]` (attempting to parse records can fail occasionally, though much more rarely now), it's easiest to write little methods to work with the data. For instance, to find out how many `404` records are in the access log, I first create this method:

```
def getStatusCode(line: Option[AccessLogRecord]) = {
  line match {
    case Some(l) => l.httpStatusCode
    case None => "0"
  }
}
```

and then use it at the Spark command line like this:

```
log.filter(line => getStatusCode(p.parseRecord(line)) == "404").count
```

There are probably better ways to do that, but that approach works. It returns a count of records where the `httpStatusCode` is 404. In my case it's quite a bit — 9,381 records out of 3.5M — so I need to fix those.

> (As a quick aside, when I ran this code on my old iMac using Spark, it took about 35 seconds. I then timed a `grep '404' accesslog | wc -l` command, and it came in at 27 seconds. `grep` is faster in this small test, in part because Spark broke my input file into four separate files, ran my code on those four files, and then condensed the results. That doesn't help much on a single server, but can provide a great benefit on larger files running across many servers. (Also, my code does a lot more work than `grep` does, which will come in handy shortly.))

Finding broken URLs

This raises an interesting question: Which URLs (URIs, technically) are broken? To find that out, I want to get the "request" field from each 404 record. So in my algorithm I need to:

1. Filter the list of all records down to just the 404 records
2. Get the `request` field from the 404 records
3. Only return the distinct records (I don't want to see duplicates)

To do this, I created this method:

```
// get the `request` field from an access log record
def getRequest(rawAccessLogString: String): Option[String] = {
  val accessLogRecordOption = p.parseRecord(rawAccessLogString)
  accessLogRecordOption match {
    case Some(rec) => Some(rec.request)
    case None => None
  }
}
```

I pasted that method into the Spark REPL, and then ran these lines of code:

```
log.filter(line => getStatusCode(p.parseRecord(line)) == "404").map(getRequest(_)).count
val recs = log.filter(line => getStatusCode(p.parseRecord(line)) == "404").map(getRequest(_))
val distinctRecs = log.filter(line => getStatusCode(p.parseRecord(line)) == "404").map(getReque
distinctRecs.foreach(println)
```

This approach worked, but I forgot that it shows the *full* request field, which has records that look like this:

```
GET /foo HTTP/1.0
GET /foo HTTP/1.1
```

Because I don't care about the HTTP version, I decided to create another method so I'd only be looking at the actual URI portion of the request field, ignoring the leading "GET" and trailing HTTP version. To do this I created this method and pasted it into the Spark REPL:

```
// val request = "GET /foo HTTP/1.0"
def extractUriFromRequest(requestField: String) = requestField.split(" ")(1)
```

and then I created this code and pasted it into the REPL:

```
val distinctRecs = log.filter(line => getStatusCode(p.parseRecord(line)) == "404")
                      .map(getRequest(_))
                      .collect { case Some(requestField) => requestField }
                      .map(extractUriFromRequest(_))
                      .distinct
distinctRecs.count
distinctRecs.foreach(println)
```

If you haven't used the `collect` method before, it works like `map`, but nicely gets ride of the `None` elements it receives. I described how it works in the Scala Cookbook, and I'll write more about it here in the future. For the purposes of this code, the thing to know is that `getRequest` returns an `Option[String]`, and `collect` easily drops any `None` records it receives.

The results

As a summary of what I learned, the code I've shared so far showed that out of the 100,000 records in my small, sample Apache access log file, I had 303 "404" records, and of those, 96 of the URIs were unique. As it turns out, some of the 404 errors are my problem — things I screwed up — and others are from people attempting to hack the website.

Notes

As a first note, when I originally released this article I hadn't had much sleep, and I used the methods shown above. In part thanks to a Twitter comment (and some sleep), I took some more time to improve my approach. The improved approach is shown in the "Improved queries" section below.

Next, although I showed my custom methods first in these examples, the way my brain works, I actually wrote the code in the opposite manner. For instance, as I was thinking through how to solve that last problem, I wrote my algorithm down first:

1. Reduce the set to only the 404 records
2. Get the request field

3. Get the URI from the request
4. Get only the unique URIs

I then wrote my Scala code like this:

```
val distinctRecs = log.filter(line => getStatusCode(p.parseRecord(line)) == "404")
                      .map(getRequest(_))
                      .collect { case Some(requestField) => requestField }
                      .map(extractUriFromRequest(_))
                      .distinct
```

And finally, that code convinced me I needed this method:

```
def extractUriFromRequest(requestField: String) = requestField.split(" ")(1)
```

To me -- coming to Scala from a Java background -- it feels great to accomplish so much in so little code. And despite the power of what's happening in that code, it's still very readable.

Improved queries

After getting to know Spark and my data a little better (and getting some sleep), I started writing some better queries. (As a word of caution, I'm still learning the Spark API.) Without much introduction at this time, here are those queries:

```
get the URIs corresponding to 404 requests
------------------------------------------

create a "null object" AccessLogRecord for use with `getOrElse`
todo: adding the "GET" request here was a poor hack; get rid of it
 nullObject = AccessLogRecord("", "", "", "", "GET /foo HTTP/1.1", "", "", "", "")

a list of 404 requests, like "GET /foo-bar HTTP/1.0"
works, but requires double-parsing
 recs = log.filter(p.parseRecord(_).getOrElse(nullObject).httpStatusCode == "404")
          .map(p.parseRecord(_).getOrElse(nullObject).request)


find the URIs with the most hits
```

```
-------------------------------

sample map/reduce from docs:
 wordCounts = textFile.flatMap(line => line.split(" "))
                      .map(word => (word, 1))
                      .reduceByKey((a, b) => a + b)

create a series of URIs
 uriCounts = log.map(p.parseRecord(_).getOrElse(nullObject).request)
                .map(_.split(" ")(1))
                .filter(_ != "/foo")

use the previous example to get to a series of "(URI, COUNT)" pairs; (MapReduce like)
 uriCounts = log.map(p.parseRecord(_).getOrElse(nullObject).request)
                .map(_.split(" ")(1))
                .map(uri => (uri, 1))
                .reduceByKey((a, b) => a + b)
 uriToCount = uriCounts.collect   // (/foo, 3), (/bar, 10), (/baz, 1) ...

what i want: URIs sorted by hit count, highest hits first
ort scala.collection.immutable.ListMap
 uriHitCount = ListMap(uriToCount.toSeq.sortWith(_._2 > _._2):_*)   // (/bar, 10), (/foo, 3), (/b

this is a decent way to print some sample data
Counts.take(10).foreach(println)
```

I'll explain those when I have some more free time, but until then, I hope it helps to see them as they are.

One thing to say is that I can shorten these queries by adding a few methods to my Apache access log parser library. For instance, the parseRecord method currently returns an `Option[AccessLogRecord]`, but it could return an `AccessLogRecord` with a Null Object representation for records I can't parse correctly. This would let me skip the `getOrElse(nullObject)` calls in these queries.

Viewing sample data

When you're working with thousands or millions of records, you may want to see some sample data from your objects to make sure you're on the right track. A simple way to see some sample records is with the `take` method:

```
 uriCounts.take(10).foreach(println)
```

That prints the first ten lines from the `uriCounts` object. Another approach is to use the takeSample method to get a sample of your data:

```
// takeSample(withReplacement, numRecordsDesired, randomNumberGeneratorSeed)
uriCounts.takeSample(false, 100, 1000)
```

With my `uriCounts` object that returns an `Array[String]`, which I can then print.

Writing output to file

Once you have the data you want, you can write it to a file, or more accurately, a series of files. For instance, this code:

```
// this creates a directory named UriHitCount, with files in it like part-00000 and part-00001
uriHitCount.saveAsTextFile("UriHitCount")
```

creates a directory named *UriHitCount*, and writes a series of files to that directory, with the files being named *part-00000*, *part-00001*, and so on. (So the method name `saveAsTextFile` is misleading.)

The future: What do I want to know about my access log files?

In the future I'll either update this article, or write new articles, as I dig into some of the following questions I have about my Apache access log files:

- What unique server status codes are there? If there is anything troubling there, I'll dig into them.
- What are the most popular web pages?
- I get a boatload of comment spam. Most of it is eliminated by Mollom, but I'm curious, what percentage of hits are from comment-spam?

- In a related note, on what records is my Apache access log parser failing on, and why?
- Much more ...

Another part of the future is getting my many GBs of log files onto an Amazon cluster, and putting Spark through its paces on multiple servers, as it's intended to be used. But alas, my free time is up for this weekend.

Performance

It's hard to judge the performance of something like Spark on a single system. It's intended to be used on huge files spread across many servers, so as I mentioned earlier, for simple tests it's possible to use `grep` and get the results back faster. (The more complicated queries I ran can't be done with `grep` alone.)

One thing I did see is that Spark pegs the needles on both of my CPUs. By contrast, `grep` uses most of one CPU, and maybe 50% of the second CPU. Again, I don't want to say anything about Spark performance until I get it running across multiple servers, but it was fun to see the dials on the CPUs cranked to 100%.

Summary

In summary, if you are interested in using Apache Spark to analyze log files -- Apache access log files in particular -- I hope this article has been helpful.

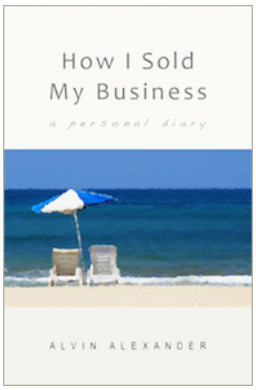
category: scala

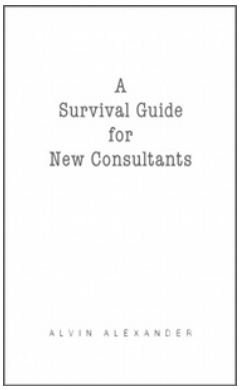
tags: access log apache log analytics scala hadoop spark analyze

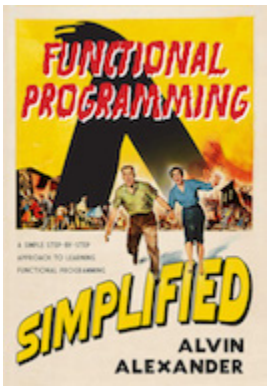
related

- Generating a list of URLs from Apache access log files, sorted by hit count, using Apache Spark (and Scala)
- Parsing “real world” HTML with Scala, HTMLCleaner, and StringEscapeUtils
- Install Apache Spark and fast log analytics
- My Scala Apache access log parser library
- How to set the SBT logging level

books i’ve written







what’s new

- Mast cell disease and the Covid vaccine (my experience)
- Love (a Virginia rest area sign)
- Functional Programming, Simplified
- In enlightenment, death has no relevance to one’s state of being
- sbt error: NoClassDefFoundError: org/fusesource/jansi/AnsiOutputStream

misc

- privacy policy
- terms & conditions
- subscribe
- unsubscribe
- wincvs tutorial
- function point analysis (fpa)
- fpa tutorial

Other

- contact me
- rss feed
- my photos
- life in alaska
- how i sold my business
- living in talkeetna, alaska
- my bookmarks
- inspirational quotes
- source code snippets

alvinalexander.com is owned and operated by Valley Programming, LLC