redislabs
home of redis

whitepaper

Getting Started With Spark and Redis

Author: Itamar Haber

Table of Contents

Executive Summary

This document outlines the initial steps needed to start using Apache Spark and Redis. We outline the steps for a basic Apache Spark install, followed by usage of the Spark-Redis package to expose Redis data structures to Spark. We use the “word count” example to demonstrate how to use Spark, Redis and the spark-redis package together.

Introduction

Redis Labs[1] recently published a spark-redis package for general public consumption. It is, as the name may suggest, a Redis connector for Apache Spark that provides read and write access to all of Redis' core data structures as RDDs (Resilient Distributed Datasets, in Spark terminology).

Since Spark was introduced, it has caught developer attention as a fast and general engine for large-scale data processing, easily surpassing alternate big data frameworks in the types of analytics that could be executed on a single platform. Spark supports a cyclic data flow and in-memory computing, allowing programs to be run faster than Hadoop MapReduce. With its ease of use and support for SQL, streaming and machine learning libraries, it has ignited early interest in a wide developer community. Redis brings a shared in-memory infrastructure to Spark, allowing it process data orders of magnitude faster. Redis data structures simplify data access and processing, reducing code complexity and saving on application network and bandwidth usage. The combination of Spark and Redis fast tracks your analytics, allowing unprecedented real-time processing of really large datasets.

So let’s explore how to get started with this powerful combination.

Setting up

There are a few prerequisites you need before you can actually use spark-redis, namely: Apache Spark, Scala, Jedis and Redis. While the package specifically states version requirements for each piece, we actually used later versions with no discernible ill effects (v1.5.2, v2.11.7, v2.8 and unstable respectively).

We start with setting up Spark on Ubuntu following this step by step guide, "Setting up a Standalone Apache Spark Cluster" published by Tim Spann @PaaSDev at @DZone.

Once you've fulfilled all the requirements, you can just git clone https://github.com/RedisLabs/spark-redis and build it by running sbt (install if you don’t already have it installed).

[1] Redis Labs and the talented Sun He @sunheehnus of the Redis community

Example Problem

For the purposes of getting started, we will use the equivalent of the “Hello World” example in analytics land, the problem of counting words. This simple problem will be used to illustrate how to use Spark and Redis together.

Step 1: Reading the data

For this example, we count the words in Redis' source code files (this commit specifically), also hoping to reveal some interesting facts in the process. With everything ready, start with spark-shell like the below:

```
itamar@ubuntu:~/src$ ./spark-1.5.2/bin/spark-shell --jars spark-redis/
target/spark-redis-0.5.1.jar,jedis/target/jedis-2.8.0.jar

Spark context available as sc.

SQL context available as sqlContext.

Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.5.2
      /_/


Using Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.7.0_91)

Type in expressions to have them evaluated.

Type :help for more information.


scala>
```

At the cursor, you can type in:

```
scala> val wtext = sc.wholeTextFiles("redis/src/*.[ch]")

wtext: org.apache.spark.rdd.RDD[(String, String)] = redis/src/*.[ch]
WholeTextFileRDD[0] at wholeTextFiles at <console>:24


scala> wtext.count

res0: Long = 100
```

The display shows there are exactly 100 Redis source files! Of course, doing `ls -1 redis.src/*.[ch] | wc -l` from the shell prompt would have displayed the same thing, but this way we can actually see the stages of the job being done by the standalone Spark cluster on the WholeTextFileRDD.

Step 2: Transforming file contents

The next step is to get the contents of the files transformed to words (that can later be counted). Unlike the usual examples that use the TextFileRDD, the WholeTextFilesRDD consists of file URLs and their contents, so we use the following snippet to split and clean the data (the call to the cache() method is strictly optional, but in keeping with best practices).

```
val fwds = wtext.
  flatMap{ case (filename, contents) =>
    val fname = filename.substring(filename.lastIndexOf("/") + 1)
    contents.
      split("\\W+").
      filter(!_.isEmpty).
      map( word => (fname, word))
  }
fwds.cache()
```

The variable names chosen are meant to be meaningful and short e.g. wtext represents WholeTextFiles, fwds is FileWords and so on.

Once the fwds RDD has clean filenames and all the words were neatly split, we are ready for some serious counting. First, we recreate the ubiquitous word counting example:

```
val wcnts = fwds.
  map{ case (fname, word) => (word, 1) }.
  reduceByKey(_ + _).
  map{ case (word, count) => (word, count.toString) }
```

Pasting the above into the spark-shell and following with take confirms success:

```
wcnts: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[5] at
map at <console>:31


scala> wcnts.take(10)

res1: Array[(String, String)] = Array((requirepass,15), (mixdigest,2),
(propagte,1), (used_cpu_sys,1), (rioFdsetRead,2), (0x3e13,1),
(preventing,1), (been,12), (modifies,1), (geoArrayCreate,3))


scala> wcnts.count()

res2: Long = 12657
```

A note about the results: take isn't supposed to be deterministic, but given that "requirepass" keeps surfacing these days, it may well be fatalistic. Also, 12657 must have some meaning but it is yet to be found.

Step 3: Writing RDDs to Redis

This is where we get started with powerful Redis. We use Redis to save the results so they can be used in later computations. Redis' Sorted Sets are a perfect match for the word-count pairs and also allow querying the data by score. It takes only one line of Scala code to do that (actually three lines, but the first two don't count):

```
import com.redislabs.provider.redis._

val redisDB = ("127.0.0.1", 6379)

sc.toRedisZSET(wcnts, "all:words", redisDB)
```

Once data is in a Redis sorted set we can use the cli to read it like so:

```
itamar@ubuntu:~/src$ ./redis/src/redis-cli
127.0.0.1:6379> DBSIZE
(integer) 1
127.0.0.1:6379> ZCARD all:words
(integer) 12657
127.0.0.1:6379> ZSCORE all:words requirepass
"15"
127.0.0.1:6379> ZREVRANGE all:words 0 4 WITHSCORES
 1) "the"
 2) "8164"
 3) "if"
 4) "6657"
 5) "0"
 6) "5396"
 7) "c"
 8) "4524"
 9) "1"
10) "4293"
127.0.0.1:6379> ZRANGE all:words 6378 6379
1) "mbl"
2) "mblen"
```

What else can we keep in Redis? The filenames are also perfect candidates, so we make another RDD and stored it in a regular Set:

```
val fnames = fwds.
  map{ case (fname, word) => fname }.distinct()
sc.toRedisSET(fnames, "all:files", redisDB)
```

Despite being very useful for science purposes, the content of the fnames Set is pretty mundane...so you can store the word count for each file in its very own Sorted Set as a more interesting example. We can do that with a few transformations/actions/RDDs:

```
fwds.
  groupByKey.
  collect.
  foreach{ case(fname, contents) =>
    val zsetcontents = contents.
      groupBy( word => word ).
      map{ case(word, list) => (word, list.size.toString) }.
      toArray
    sc.toRedisZSET(sc.parallelize(zsetcontents), "file:" + fname, redisDB)
  }
```

Back to redis-cli:

```
127.0.0.1:6379> dbsize
(integer) 102
127.0.0.1:6379> ZREVRANGE file:scripting.c 0 4 WITHSCORES
 1) "lua"
 2) "366"
 3) "the"
 4) "341"
 5) "if"
 6) "227"
 7) "1"
 8) "217"
 9) "0"
10) "197"
```

Step 4. Reading RDDs from Redis

Compared to storing word count data , a more practical use of the data in Redis is done with reads. Run the following code to take the per-file word counts and reduce them to basically the same output of the classic WC challenge:

```
val rwcnts = sc.fromRedisKeyPattern(redisDB, "file:*").
  getZSet().
  map{ case (member, count) => (member, count.toFloat.toInt) }.
  reduceByKey(_ + _)
```

Then back to spark-shell to test this code and get a grand total of all words:

```
scala> rwcnts.count()
res8: Long = 12657

scala> val total = rwcnts.aggregate(0)(
     |   (acc, value) => acc + value._2,
     |   (acc1, acc2) => acc1 + acc2)
total: Int = 272655
```

Lets double check using a Lua script:

```
local tot1, tot2, cursor = 0, 0, 0

repeat
  local rep1 = redis.call('SCAN', cursor, 'MATCH', 'file:*')
  cursor = tonumber(rep1[1])

  for _, ssk in pairs(rep1[2]) do
    local rep2 = redis.call('ZRANGE', ssk, 0, -1, 'WITHSCORES')
    for i = 2, #rep2, 2 do
      tot1 = tot1 + tonumber(rep2[i])
    end
  end

until cursor == 0

local rep = redis.call('ZRANGE', 'all:words', 0, -1, 'WITHSCORES')
for i = 2, #rep, 2 do
  tot2 = tot2 + tonumber(rep[i])
end

return { tot1, tot2 }



itamar@ubuntu:~/src$ ./redis/src/redis-cli --eval /tmp/wordcount.lua
1) (integer) 272655
2) (integer) 272655
```

Closing notes

Back in the days when data was small, you could get away with counting words using a simple wc -w. As data grows, we find new ways to abstract solutions and in return gain flexibility and scalability. Spark is an exciting tool to have and its core is extremely useful. And that's even without going into its integration with the Hadoop ecosystem and extensions for SQL, streaming, graphs processing and machine learning.

Redis quenches Spark's thirst for data. spark-redis lets you marry RDDs and Redis core data structures with just a line of Scala code. The spark-redis package already provides straightforward RDD-parallelized read/write access to all core data structures and a polite (i.e. SCAN-based) way to fetch key names. Furthermore, the connector carries considerable hidden punch as it is actually (Redis) cluster-aware and maps RDD partitions to hash slots to reduce inter-engine shuffling. The package is open source and has many more enhancements planned that should make Redis a default choice for use with Spark.