Python vs. Scala vs. Spark

17 JAN 2015 | PYTHON SCALA SPARK | (/2015/01/17/PYTHON-VS-SCALA-VS-SPARK/)

Update 20 Jan 2015

Thanks to a suggestion from a reddit comment, I added benchmarks for the python code running under PyPy. This makes the results even more interesting. PyPy actually runs the join faster than Scala when more cores are present. On the other hand, it runs the sort slower, leading to an approximately equal performance when there are more than 2 cores available. This is really good news for people (like myself) who are more familiar with python and don't want to learn another language just to execute faster Spark queries.

Apart from the extra data in the charts, the rest of this post is unmodified and thus doesn't mention PyPy.

The fantastic Apache Spark (https://spark.apache.org/) framework provides an API for distributed data analysis and processing in three different languages: Scala, Java and Python. Being an ardent yet somewhat impatient Python user, I was curious if there would be a large advantage in using Scala to code my data processing tasks, so I created a small benchmark data processing script using Python, Scala, and SparkSQL.

The Task

The benchmark task consists of the following steps:

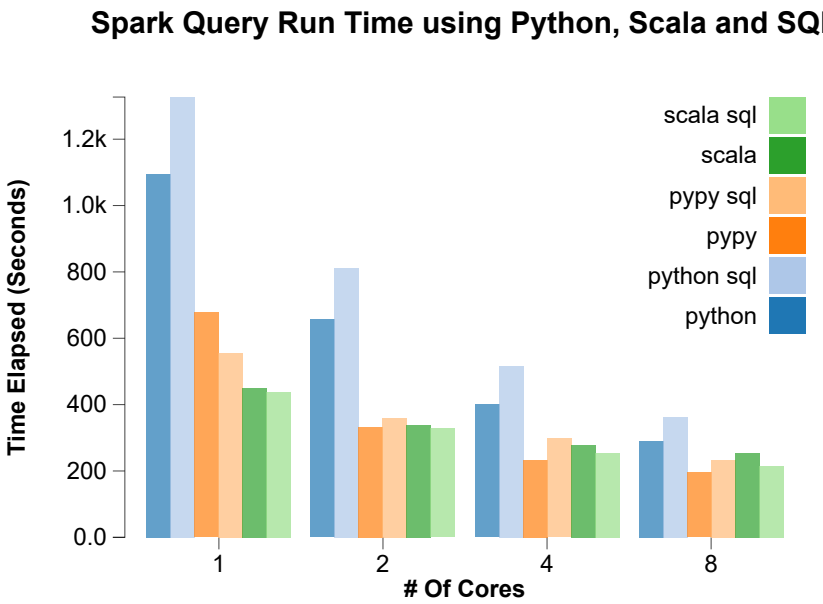
1. Load a tab-separated table (gene2pubmed), and convert string values to integers (map, filter)
2. Load another table (pmid_year), parse dates and convert to integers (map)
3. Join the two tables on a key (join)
4. Rearrange the keys and values (map)
5. Count the number of occurances of a key (reduceByKey)
6. Rearrange the keys and values (map)
7. Sort by key (sortByKey)

The Data

The dataset consists of two text file tables, weighing in at 297M and 229M.

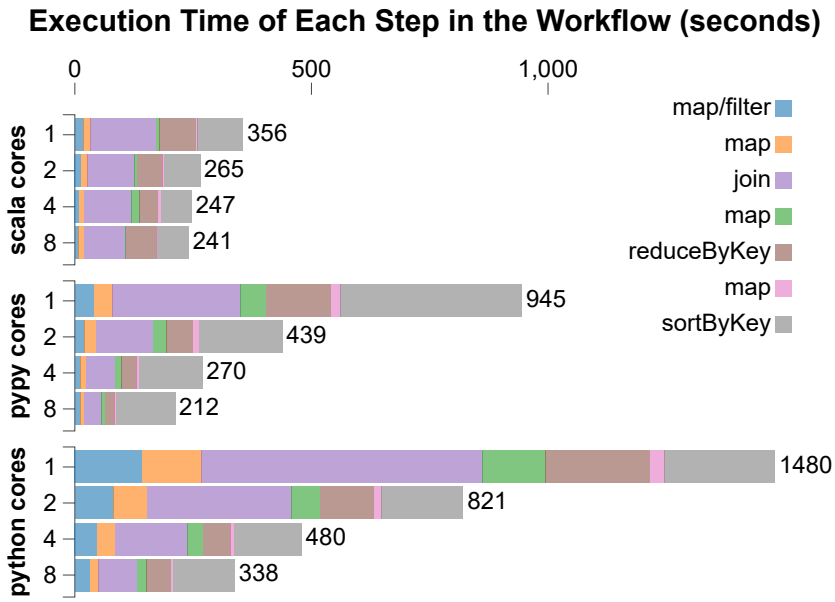
Total Running Time

Each of the scripts was run with a `collect` statement at the end to ensure that each step was executed. The time was recorded as the real time elapsed between the start of the script and the end. The scripts were run using 1,2,4 and 8 worker cores (as set by the `SPARK_WORKER_CORES` option).



The fastest performance was achieved when using SparkSQL with Scala. The slowest, SparkSQL with Python. The more cores used, the more equal the results. This is likely due to the fact that parallelizable tasks start to contribute less and less of the total time and so the running time becomes dominated by the collection and aggregation which must be run synchronously, take a long time and are largely language independent (i.e. possibly run by some internal Spark API).

To get a clearer picture of where the differences in performance lie, a `count()` action was performed after each transformation and other action. The time was recorded after each `count()`.



This data indicates that just about every step in the Python implementation, except for the final sort, benefitted proportionally (~ 8x) from the extra cores. The Scala implementation, in contrast, showed no large speedup in any of the steps. The longest, the join and sort steps, ran about 1.5 times faster when using 8 cores vs when using just 1. This can be either because the dataset is too small to benefit from parallelization, given Scala's already fast execution, or that something strange is going on with the settings and the operations performed by the master node are being run concurrently even when there are less worker nodes available.

This doesn't appear to be case as running both the master and worker nodes on a machine with only four available cores (vs 24 in the previous benchmarks) and allowing only one worker core actually led to faster execution. A more comprehensive test would require running the master node on a single core machine and placing the workers on separate more capable computers. I'll save that for another day though.

Conclusion

If you have less cores at your disposal, Scala is quite a bit faster than Python. As more cores are added, its advantage dwindles. Having more computing power gives you the opportunity to use alternative languages without having to wait for your results. If computing resources are at a premium, then it might make sense to learn a little bit of Scala, if only enough to be able to code SparkSQL queries.

Apache Spark

The code for each programming language is listed in the sections below:

Master and Worker Nodes

The number of workers was set in the `SPARK_WORKER_CORES` variable in `conf/spark-env.sh`

```
./sbin/stop-all.sh; rm logs/*; ./sbin/start-all.sh
```

Python Shell

```
./spark-1.2.0/bin/pyspark --master spark://server.com:7077 --
```

Scala Shell

```
./spark-1.2.0/bin/spark-shell --master spark://server.com:707
```

Benchmark Code

The following code was pasted into its respective shell and timed.

Scala

```
val pmid_gene = sc.textFile("/Users/pkerp/projects/genbank/da

// the dates were obtained by doing entrez queries and stored
// table of the form "date pmid"
        object ParsingFunctions {
                def parseDate(line: String): (Int,Int) = {
                    // extract the date information and the p
                    // and return it as tuple
                    val parts = line.split(" ");
                        // the year is in %Y/%m/%d format
                        val year = parts(0).split("/")(0).toI
                        val pmid = parts(1).toInt;
                        return (pmid, year);
                }
        }
val pmid_year = sc.textFile("/Users/pkerp/projects/genbank/da
val pmid_gene_year = pmid_year.join(pmid_gene)

val gene_year_1 = pmid_gene_year.map{ case (pmid, (gene, year
val gene_year_counts = gene_year_1.reduceByKey((x,y) => x+y)
val counts_gene_year = gene_year_counts.map{case ((gene,year)
val sorted_counts_gene_year = counts_gene_year.sortByKey()
val scgy = sorted_counts_gene_year.collect()
//blah
```

Scala SQL

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.createSchemaRDD
case class PmidGene(taxid: Int, geneid: Int, pmid: Int)
case class PmidYear(pmid: Int, year: Int)

val gene2pubmed = sc.textFile("/Users/pkerp/projects/genbank/

gene2pubmed.registerTempTable("gene2pubmed")

        object ParsingFunctions {
                def parseDate(line: String): PmidYear = {
                    // extract the date information and the p
                    // and return it as tuple
                    val parts = line.split(" ");
                        // the year is in %Y/%m/%d format
                        val year = parts(0).split("/")(0).to
                        val pmid = parts(1).toInt;

                        return PmidYear(pmid, year);
                }
        }

val pmid_year = sc.textFile("/Users/pkerp/projects/genbank/da
pmid_year.registerTempTable("pmid_year")
val geneid_year = sqlContext.sql("select geneid, year from pm
geneid_year.registerTempTable("geneid_year")
val result = sqlContext.sql("select geneid, year, count(*) as
val x = result.collect()
//blah
```

Python

```
# get the gene_id -> pubmed mapping
pmid_gene = sc.textFile("/Users/pkerp/projects/genbank/data/g

# the dates were obtained by doing entrez queries and stored
# table of the form "date pmid"
def parse_date_pmid_line(line):
    # extract the date information and the pmid
    # and return it as tuple
    parts = line.split()
    # the year is in %Y/%m/%d format
    year = int(parts[0].split('/')[0])
    pmid = int(parts[1])
    return (pmid, year)

# extract the dates and store them as values where the key is
pmid_year = sc.textFile('/Users/pkerp/projects/genbank/data/p
pmid_gene_year = pmid_year.join(pmid_gene)

gene_year_1 = pmid_gene_year.map(lambda (pmid, (gene, year)):
gene_year_counts = gene_year_1.reduceByKey(lambda x,y: x+y)
counts_gene_year = gene_year_counts.map(lambda ((gene, year),
sorted_counts_gene_year = counts_gene_year.sortByKey(ascendir
scgy = sorted_counts_gene_year.collect()
#blah
```

Python SQL

```
from pyspark.sql import *
sqlContext = SQLContext(sc)
# get the gene_id -> pubmed mapping
gene2pubmed = sc.textFile("/Users/pkerp/projects/genbank/data
schemaGene2Pubmed = sqlContext.inferSchema(gene2pubmed)
schemaGene2Pubmed.registerTempTable("gene2pubmed")
# the dates were obtained by doing entrez queries and stored
# table of the form "date pmid"
def parse_date_pmid_line(line):
    # extract the date information and the pmid
    # and return it as tuple
    parts = line.split()
    # the year is in %Y/%m/%d format
    year = int(parts[0].split('/')[0])
    pmid = int(parts[1])
    return (pmid, year)


pmid_year = sc.textFile('/Users/pkerp/projects/genbank/data/p
schemaPmidYear = sqlContext.inferSchema(pmid_year)
schemaPmidYear.registerTempTable('pmid_year')
geneid_year = sqlContext.sql('select geneid, year from pmid_y
geneid_year.registerTempTable('geneid_year')
result = sqlContext.sql('select geneid, year, count(*) as cnt
x = result.collect()
#blah
```