Hadoop Streaming

Jim Harner

1/6/2020

4.6 Hadoop Streaming

The following are two ways to submit work to a Hadoop cluster:

- *Streaming*: Write the Map and Reduce operations as R scripts (or in another scripting language). The Hadoop framework launches the R scripts at appropriate times and communicates by standard input and standard output.
- *Java API*: Write the Map and Reduce operations in Java.

The Java code runs `Runtime.exec()` to invoke the R scripts. The appropriate method depends on several factors, including your understanding of Java versus R, and the particular problem you're trying to solve.

- Streaming tends to win for rapid development.
- Java API is useful for working with data such as images or sound files.

4.6.1 Ex1: Simple Hadoop Streaming

Setting: The input data is several million lines of plain-text phone call records. Each CSV input line is of the format:

```
{id}, {date}, {caller_num}, {caller_carrier}, {dest_num}, {dest_carrier}, {length}
```

The plan is to analyze each call record separately, so there's no need to sort and group the data. The job is run with only a mapper R script and a follow-up analysis R script.

mapper.R

```
#! /usr/bin/env Rscript

input <- file( "stdin" , "r" )

while( TRUE ){

    currentLine <- readLines( input )
    if( 0 == length( currentLine ) ){
        break
    }

    currentFields <- unlist( strsplit( currentLine , "," ) )
    result <- paste( currentFields[3] , currentFields[7], sep="\t" )
    cat(result, "\n")
}

close( input )
```

Hadoop Streaming sends input records to the `mapper.R` script via standard input. A Map script receives one or more input records in a single call (the default for `readLines` is one record at a time); we read from standard input until there's no more data, i.e., when `0 == length( currentLine )`.

Ideally, the incoming lines are split among the nodes to gain parallelism. This did not happen here due to hdfs's block size since `short.csv` only has 100 observations. Also, see the `output` directory which only has one part.

Each comma-separated input line is split into a list using `strsplit`, but is then unlisted and becomes a character vector. The third (`call_num`) and seventh fields are pasted together to form the key-value pair, which in this case are:

key: `call_num`
value: `length`

`cat` then write the key-value pairs to standard output, which could be send to a reducer or, as in this case, to further analyses.

Run the Hadoop job from the command line.

The Hadoop job can be run from the shell line-by-line, but it is more efficient to build a `bash` script. Before running this scrip, you need to be in the `RhadoopEx1` directory.

The Java code runs `Runtime.exec()` to invoke the `mapper.R` R script. The shell script to execute the job is given in `ex1.sh`. The output is voluminous and thus should be run from R's Shell in the Tools menu.

```
# set the working directory to RhadoopEx1
cd /home/rstudio/rspark-notes/c4_hadoop/s6_streaming/RhadoopEx1
./ex1.sh
```

The `ex1.sh` script removes `output` from both the local filesystem and hdfs. Then we `put` the data file `short.csv` into hdfs. Next we run `hadoop` with the `jar` command. When using `NLineInputFormat` as the `inputformat`, Hadoop treats each line as a split and spreads the work evenly throughout the cluster. The `input`, `output`, and `mapper` files or directories are also options. There is no `reducer` file in this example.

Review the output.

A typical Hadoop job will create several files, one for each Reduce operation. Since this is a Map-only job, there is just one file. If the output file is compressed, you will need to uncompress it, e.g., using `gunzip`.

Now process the output file by computing the mean call length for each call number. Again, `analysis.R` should be run in R's Shell.

```
# set the working directory to RhadoopEx1
cd /home/rstudio/rspark-notes/c4_hadoop/s6_streaming/RhadoopEx1
./analysis.R
```

We get a names vector with the call number as the name and the average length as the value.

Prototyping A Hadoop Streaming Job

For streaming jobs, you can chain the scripts with pipes to simulate a workflow. For Ex1 the command sequence is:

```
# set the working directory to RhadoopEx1
cd /home/rstudio/rspark-notes/c4_hadoop/s6_streaming/RhadoopEx1
cat short.csv | ./mapper.R | sort | ./analysis.R
```

Of course the computations are done sequentially, but is does allow your logic to be checked. The results are identical to those of the streaming job.

4.6.2 Ex2: Processing Related Groups using Full Map/Reduce

Setting: You want to collect related records and operate on that group as a whole. Returning to the "phone records" example, let's say you want to analyze every number's output call patterns. That would require you to first gather all of the calls made by each number (Map phase) and then process those records together (Reduce phase).

The code: As noted above, this will require both the Map and Reduce phases. The Map phase code will extract the caller's phone number to use as the key.

mapper.R:

```
#! /usr/bin/env Rscript

input <- file("stdin", "r")

while(TRUE){

    currentLine <- readLines(input, n=1)
    if(0 == length(currentLine)) break

    currentFields <- unlist( strsplit(currentLine , ","))
    result <- paste(currentFields[3], currentFields[7] , sep="\t")
  cat(result, "\n")
}

close(input)
```

The third field in the comma-separated line is the caller's phone number, which serves as the key output from the Map task, whereas the 7th field is the call length, which serves as the value.

The Reducer code builds a `data.frame` of all calls made by each number, and then passes the `data.frame` to the analysis function to compute the average call length. This example is the same as Example 1 except that the average length is computed by the reduce phase rather than an external file, i.e., outside of Hadoop.

In a Reducer script, each input line is of the format:

```
  {key}{tab}{value}
```

where `{key}` and `{value}` are a single key/value pair, as output from a Map task.

The Reducer's job is to collect all of the values for a given key, then process them together. Hadoop may pass a single Reducer values for multiple keys, but it will sort them first. When the key changes, then, you know you've seen all of the values for the previous key. You can process those values as a group, then move on to the next key.

reducer.R

```
#! /usr/bin/env Rscript

input <- file("stdin" , "r")
lastKey <- ""

tempFile <- tempfile(pattern="hadoop-mr-demo" , fileext="csv")
tempHandle <- file(tempFile , "w")

while(TRUE){

    currentLine <- readLines(input , n=1)
    if(0 == length(currentLine)){
        break
    }

# break this apart into the key and value that were
# assigned in the Map task
    tuple <- unlist(strsplit(currentLine, "\t"))
    currentKey <- tuple[1]
    currentValue <- tuple[2]

    if(currentKey != lastKey))){
# a little extra logic here, since the first time through,
# this conditional will trip

        if(lastKey != ""){
# we've hit a new key, so first let's process the
# data we've accumulated for the previous key:

# close tempFile connection
            close(tempHandle)

# read file of accumulated lines into a data.frame
            bucket <- read.csv(tempFile , header=FALSE)

# process data.frame and write result to standard output
            result <- apply(as.matrix(bucket[1]), 2, mean)

# write result to standard output
            cat(lastKey , "\t" , result , "\n")

# cleaup, and start fresh for the next round
            tempHandle <- file(tempFile , "w")
        }

      lastKey <- currentKey

    }

# by now, either we're still accumulating data for the same key
# or we have just started a new file.  Either way, we dump a line
# to the file for later processing.
  cat(currentValue , "\n" , file=tempHandle, append=TRUE)

}

## handle the last key, wind-down, cleanup
close(tempHandle)

bucket <- read.csv(tempFile , header=FALSE)
result <- apply(as.matrix(bucket[1]), 2, mean)
cat(currentKey , "\t" , result , "\n")

unlink(tempFile)
```

```
close(input)
```

This MapReduce job is doing the exact same computation as Ex1 except a reducer is used rather than an analysis Rscript. The dataset is the same as in Example 1 so the results can be compared. Once data frames are formed for each call, the analysis could be much more complex.

The bash script for running the job centers around the `hadoop jar` command.

```
# set the working directory to RhadoopEx2
cd /home/rstudio/rspark-notes/c4_hadoop/s6_streaming/RhadoopEx2

./ex2.sh
```

As before the `output` directories are removed from both hdfs and the local filesystem. The call data (`short.csv`) is then uploaded to hdfs and the `hadoop` command it issued. The `output` directory is then downloaded.

The results for both examples 1 and 2 are identical except that the format is different. A larger input file, `cdat2.csv` is also available for experimentation.

Caveats:

Typically, the Map phase is very lightweight (since it's just used to assign keys to each input) and the heavy lifting takes place in the Reduce operation. To take advantage of the parallelism of the Reduce stage, you'll need to meet two conditions:

1. A large number of unique keys output from the Map phase;

2. Each key should have a similar number of records (at least, no one key should clearly dominate).

Hadoop splits the work across a cluster, sending each unit of work to a different node even though R itself is single-threaded, this allows the use of tens or hundreds of CPUs.

4.6.3 Java Word Count

For the Java API, the Map and Reduce operations are written in Java. The Java code runs `Runtime.exec()` to invoke R scripts, if needed. The canonical example is counting the number of occurrences of each word in one or more files. The `WordCount.java` file contains the source for the mapper and the reducer. The `data` directory contains four files containing words.

```
# set the working directory to JavaWC
cd /home/rstudio/rspark-notes/c4_hadoop/s6_streaming/JavaWC

./cmd.sh
```

The output file contains a count of the number of occurrences for each word.

4.6.4 Working with Binary Data

Hadoop Streaming can only be used for text input and output. This doesn't preclude you from working with binary data in a streaming job; but it does preclude your Map and Reduce scripts from accepting binary input and producing binary output. Note that the Java API does work with binary data.

Setting: Imagine that you want to analyze a series of image files. Perhaps they are frames from a video recording, or a file full of serialized R objects, or maybe you run a large photo-sharing site. For this example, let's say you have R code that will perform image feature extraction.

The code: Hadoop Streaming can only handle line-by-line text input and output. One option would be to feed your Hadoop Streaming job an input of pointers to the data, which your R script could then fetch and process locally. For example:

- Host the data on an internal web server, and feed Hadoop a list of URLs;
- Use an NFS mount;
- Use `scp` to pull the files from a remote system;
- Make a SQL call to a database system.

HDFS does't work well with small files.

```
#! /usr/bin/env Rscript
input <- file( "stdin" , "r" )
while( TRUE ){
  currentLine <- readLines( input , n=1 )
  if( 0 == length( currentLine ) ){ break
  }

  pulledData <- url( currentLine ) )

  result <- imageFeatureExtraction( url( currentLine ) )
  cat( result , "\n" , sep="" )
}
close( input )
```

Run the Hadoop job as before.

Review the Output.

if your job yields binary output, you can use the same idea as you did for the input, and push the output to another system:

- Copy it to an NFS mount;
- Use an HTTP POST operation to send the data to a remote web server;
- Call `scp` to ship the data to another system;
- Use SQL to push the data to an RDBMS.

Caveats

A Hadoop cluster is robust, i.e., on the software-side framework and the required hardware layout, you are protected from hard disk failures, node crashes, and even loss of network connectivity.

Everything required for the job must *exist within the cluster*. Map and Reduce scripts, input data, and output must all live in HDFS (S3 if you are using Elastic MapReduce).

For systems or services outside of the cluster, you lose in four ways:

- Loss of robustness
- Scaling

- Overhead
- Idempotence/risk of side effects

The `rmr2` package in RHadoop allows R developer to perform statistical analysis in R via Hadoop MapReduce functionality on a Hadoop cluster.