Using Torch on Hadoop

Roy E Lowrance

September 30, 2013

1 Problem

You want to run a streaming job on NYU's Hadoop cluster using torch as the programming language.

2 Solution

The solution is requires getting a lot of details right.

1. Get your data and torch scripts onto babar, the Hadoop cluster.
2. Make sure a suitable instance of torch is available on babar, the Hadoop cluster.
3. Break your map-reduce job into two torch programs. One will be the map command, the other the reduce command. A *command* is a program that receives no torch arguments (command line options).
4. Use the streaming interface to submit a map-reduce job. The job will read a single file in the Hadoop file system. It will create a directory of files in the Hadoop file system.
5. Copy the output directory from the Hadoop file system to the local file system for ease of examination and subsequent use.

3 Discussion

We provide a worked example of a streaming job written in torch. The code is short enough so that you can just type it in. It is also available in the repository that holds this TeX code that generated this document. How to get the source of this document is discussed in the section "See Also."

3.1 Move data and code to babar

Before working through the example, there are some preliminary steps. You need an NYU Hadoop logon. To get one, send a request in an email to Yann LeCun. You need to install torch in your home directory on the Hadoop cluster. We provide a recipe for torch installation in this cookbook.

Log into the the Hadoop cluster. In the example below, ID is your user id. babar is the name of the login node for the Hadoop cluster. (There is also `dumbo.es.its.nyu.edu`, but it doesn't have as much software installed on it.)

```
$ ssh ID@hpc.nyu.edu
$ ssh babar.es.its.nyu.edu
```

Now you should create a test file in the local file system. We provide the file `courant-abel-prize-winners.txt` which lists faculty of the Courant Institute who won the Abel Prize. Each row contains the year of the award, a tab character, and the name of the winner. Here's the file:

If you are using your own data and code, I recommend moving the data to babar using `rsync` and your code to babar using `github`.

3.2 Making a torch executable available

You need an accessible instance of torch. For this you can either install your own torch locally on babar in your user account or use someone else's instance. I like to install my own so that I know the torch I'm using on babar is exactly the same as the one on my development system. To install torch locally, follow the recipe for installing torch. At the time this note was written, Xiang Zhang's torch install was available at `/home/xz558/.usr/bin`. Note the dot before the `usr` directory. Xiang's torch install was up to date when this note was written.

Be careful of the installed torch executables already on babar. At one time, `dumbo`, an alternative Hadoop login node, had an installed version of torch, but it was old and had bugs that had been fixed in subsequent releases.

To find out which torch executable you have, enter `$ which torch` in a terminal. If you have installed torch and the `which` command doesn't bring it up, fix your `$PATH` variable by executing `EXPORT $PATH=<directory with your torch>:$PATH` in your `.bash_profile` script. If you edit your `.bash_profile` you will need to get bash to read it again. You do that by executing this command in a terminal: `$ source .bash_profile`.

3.3 Create the map-reduce commands

Create two commands that will form your map-reduce job. Each command is a torch source file that starts with a shebang an exclamation mark). The source file is invoked by the Hadoop run-time with no parameters, so it may not depend on command line arguments.

The map command reads stdin, which will be a file in the Hadoop file system. It writes to stdout a series of records. Each record is comprised of a key and a

value separated by a tab character. Hadoop may run multiple instances of your map command. In fact, you hope that it does, because that's where the speed up comes from. Each instance will produce a separate stdout key-value file.

Since you are using the streaming interface, you cannot specify a combiner.

When all the mappers finish running, their output files are combined and then split into one or more input files for the reducers to read. The split is such that each input to a reducer contains all the output records with a given key value.

The reduce command reads stdin and writes to stdout. The stdin file will look something like this:

```
aaa\t1
aaa\t2
bbb\t3
ccc\t4
ccc\t5
```

Several reduce programs may be run. Each will receive all of the records with a given key. Some may receive records with more than one key. You must program accordingly. The output of the reduce function is a file in the Hadoop file system. The file names have the form `part-NNNNN`.

The sample job `countInput` is a variant on the Unix program `wc`. It reads a file in the Hadoop file system and produces a directory in the Hadoop file system. The `part-NNNNN` files contain the number of records in the input, the number of bytes in keys in the input, and the number of byte in values in the input. Keys and values in the input file are separated by tab characters. If there is no tab character in an input record, the entire record is treated as a key. To fully mimic the unix `wc` command, the part files should be combined in some way, but that is not done here.

The job requires two commands, which I put into two torch files: `countInput-map.lua` and `countInput-reduce.lua`. Each file must begin with a shebang line to invoke the torch interpreter. The correct interpret to use is `torch-lua` as, when invoked in this way, it does not print a greeting to stdout. If the greeting were printed, it would appear in your output file.

The map program `countInput-map.lua` reads records from stdin. It attempts to break each record into a key and value. It counts the number of records, the number of bytes in keys, and the number of bytes in values.

The map and reduce programs share a function that splits an input record into a key and value.

Here is the source code for `countInput-map.lua`.

```
#!/home/rel292/local/bin/torch-lua
-- count number of records and bytes in stdin map function
-- stdin format: records consisting of pairs of strings
--   (keyname --> count)
-- stdout format:
```

```
--   (keyname --> sum of counts for that keyname)

require "getKeyValue"

local records = 0
local keyBytes = 0
local valueBytes = 0
for line in io.stdin:lines("*l") do
    if line == nil then break end
    records = records + 1
    local key, value = getKeyValue(line)
    keyBytes = keyBytes + string.len(key)
    valueBytes = valueBytes + string.len(value)
end
print("records" .. "\t" .. tostring(records))
print("keyBytes" .. "\t" .. tostring(keyBytes))
print("valueBytes" .. "\t" .. tostring(valueBytes))
```

Here is the source code for `countInput-reduce.lua`.

```
#!/home/rel292/local/bin/torch-lua
---- count number of records in standard in reduce function
-- this is the reduce
-- input is in some arbitrary order
--   ("records" --> <number of records>) ...
--   ("keyBytes" --> <number of bytes in keys>) ...
--   ("valueBytes" --> <number of bytes in values>) ...
-- output is total for each key value
-- NOTE: keys are arbitary strings, not restricted to value named above

require "getKeyValue"

local lastKey = nil
local count = 0
for line in io.stdin:lines("*l") do
    if line == nil then break end
    local key, value = getKeyValue(line)
    if lastKey == key then
        count = count + tonumber(value)
    else
        if lastKey then
            print(lastKey .. "\t" .. count)
        end
        lastKey = key
        count = tonumber(value)
```

```
    end
end
if lastKey then
    print(lastKey .. "\t" .. count)
end
```

Here is the source code for the `getKeyValue.lua`.

```
-- parse key and value from input record
-- ARGS:
-- s : string, the input record
--
-- RETURNS:
-- key   : string, bytes in s up to but not including the first tab (\t)
-- value : string, bytes in s after the tab (\t); potentially empty
function getKeyValue(s)
    local key, value = string.match(s, "^(.*)\t(.*)$")
    if key == nil then
        return s, nil
    end
    return key, value
end
```

3.4 Run your map-reduce job

The job is run by the `countInput-run.sh` shell script below. It is a bit complex and is described just after the listing. I wrote it with more comments than you may need, just in case you are a beginner. You run the script by first changing to the directory where your lua source code is and then running the command. In my setup, all the code is in one directory: both the lua scripts and shell scripts, so I

```
$ cd <source code directory>
$ ./countInput-run.sh
```

You will need to make the shell script executable be executing the command `$ chmod +x countInput-run.sh`.

I use a few naming conventions in the shell script.

- If the name of the job is `JOBNAME`, the name of the mapper is `JOBNAME-map.lau` and the name of the reducer is `JOBNAME-reduce.lau`.
- If the input file is `INPUT`, the output directory is `INPUT.JOBNAME`.
- The script copies the output directory the local directory `$HOME/map-reduce-output/INPUT.JOBNAME`.

Here is the shell script `countInput-run.sh`. It is followed with a description of each of the steps.

```
# run countInput map-reduce streaming job
# USAGE:
# Change to the directory that contains the source and shell files
# then run
#  ./countInput-run.sh

# 1: options to consider whilst debugging
#set -e     # stop at first non-zero return code
#set -x     # print each command before executing it

# 2: set control variables
INPUT_FILE="courant-abel-prize-winners.txt"
JOB_NAME="countInput"
USER_DIR="/user/$USER"
SRC=$PWD
MAPPER="${SRC}/${JOB_NAME}-map.lua"
REDUCER="${SRC}/${JOB_NAME}-reduce.lua"

# 3: input and output
# NOTES: $INPUT_FILE/$JOB_NAME as the output directory does not work because
# there is already a file $INPUT_FILE and there cannot be a directory with
# the same name
INPUT_PATH=$USER_DIR/$INPUT_FILE
OUTPUT_DIR=$INPUT_FILE.$JOB_NAME
LOCAL_OUTPUT_DIR=$HOME/map-reduce-output/$OUTPUT_DIR

# 4: system default streaming jar
HADOOP_HOME=/usr/lib/hadoop
STREAMING="hadoop-streaming-1.0.3.16.jar"

# 5: copy test file to the hadoop file system if it is not already there
hadoop fs -test -e $INPUT_FILE
if [ $? -ne 0 ]
then
  echo copying input from local file system to hadoop file system
  hadoop fs -copyFromLocal $INPUT_FILE $INPUT_FILE
else
  echo input file already in the hadoop file system
fi


# 6: delete output directory from previous run if it exists
hadoop fs -ls
```

```
hadoop fs -test -e $OUTPUT_DIR
if [ $? -eq 0 ]
then
  echo deleting output directory: $OUTPUT_DIR
  hadoop fs -rmr $OUTPUT_DIR
else
  echo output directory not in the hadoop file system
fi

# 7: run the streaming job; it will create the output directory
echo starting hadoop streaming job
echo mapper=$MAPPER
echo reducer=$REDUCER
echo input path=$INPUT_PATH
echo output dir=$OUTPUT_DIR
hadoop jar $HADOOP_HOME/contrib/streaming/$STREAMING \
 -file *.lua \
 -mapper $MAPPER \
 -reducer $REDUCER \
 -input $INPUT_PATH \
 -output $OUTPUT_DIR

# 8: copy output file to home directory
FROM=$USER/$OUTPUT_DIR
FROM=$OUTPUT_DIR
TO=$HOME/map-reduce-output/$OUTPUT_DIR
if [ -a ~/map-reduce-output/$OUTPUT_DIR ]
then
  # output directory exists, so delete it
  cd ~/map-reduce-output; rm -rf -- $OUTPUT_DIR # delete $TO directory
fi
mkdir -p $TO  # copyToLocal wants the directory to already exist
hadoop fs -copyToLocal $FROM $HOME/map-reduce-output/

# 9: list output dir
echo OUTPUT DIRECTORY
ls $TO

# 10: print main output file
echo FIRST OUTPUT FILE part-00000
cat $TO/part-00000
```

Here is a description of the shell script.

1. While debugging, I found these options useful.
2. Set up the name of the input file and the job name. The command names

are based on the job name.

3. Locate the input file and output directory.
4. Locate the hadoop streaming executable.
5. The input file must be in the Hadoop file system. If its not there, copy it from the local file system.
6. The output directory cannot exist when the streaming job is started. Delete it if it exists.
7. Run the streaming job. The lua source files need to be moved to the Hadoop execution nodes my naming them in the `-file` option. It's easiest to move all of the lua source code instead of just the source code files used.
8. The streaming job writes to a directory in the Hadoop file system. It's easier to examine output in the local file system, so the script copies the streaming output directory to the local file system.
9. I'm eager to find out what happened, so I list the local output directory.
10. I know the result will be in a specific file in the output directory, so I print that file.

The output directory will contain a `_SUCCESS` or `_FAILURE` file that acts as a signal for the success of the map-reduce job. If the job was successful, it will contain a `part-NNNNN` file for each reducer that ran.

4 See also

When you have many hadoop jobs to run, you will want to refactor this solution to increase code usage. For a solution, see the recipe for "I want to run many Hadoop jobs."

You will find that the debugging environment provided by Hadoop is terrible compared to less complex and parallelized environments. We provide some guidance on how to ease testing in the recipe "I want to test my Hadoop program."

This recipe is free documentation. You can modify it by visiting github for account "rlowrance" and forking the repo "torch-cookbook."