- 2014-01-18 00:00:00 +0000 UTC

Adam Drake

Jan 18, 2014

Command-line Tools can be 235x Faster than your Hadoop Cluster

Introduction

As I was browsing the web and catching up on some sites I visit periodically, I found a cool article from Tom Hayden about using Amazon Elastic Map Reduce (EMR) and mrjob in order to compute some statistics on win/loss ratios for chess games he downloaded from the millionbase archive, and generally have fun with EMR. Since the data volume was only about 1.75GB containing around 2 million chess games, I was skeptical of using Hadoop for the task, but I can understand his goal of learning and having fun with mrjob and EMR. Since the problem is basically just to look at the result lines of each file and aggregate the different results, it seems ideally suited to stream processing with shell commands. I tried this out, and for the same amount of data I was able to use my laptop to get the results in about 12 seconds (processing speed of about 270MB/sec), while the Hadoop processing took about 26 minutes (processing speed of about 1.14MB/sec).

After reporting that the time required to process the data with 7 c1.medium machine in the cluster took 26 minutes, Tom remarks

> This is probably better than it would take to run serially on my machine but probably not as good as if I did some kind of clever multi-threaded application locally.

This is absolutely correct, although even serial processing may beat 26 minutes. Although Tom was doing the project for fun, often people use Hadoop and other so-called *Big Data (tm)* tools for real-world processing and analysis jobs that can be done faster with simpler tools and different techniques.

One especially under-used approach for data processing is using standard shell tools and commands. The benefits of this approach can be massive, since creating a data pipeline out of shell commands means that all the processing steps can be done in parallel. This is basically like having your own Storm cluster on your local machine. Even the concepts of Spouts, Bolts, and Sinks transfer to shell pipes and the commands between them. You can pretty easily construct a stream processing pipeline with basic commands that will have extremely good performance compared to many modern *Big Data (tm)* tools.

An additional point is the batch versus streaming analysis approach. Tom mentions in the beginning of the piece that after loading 10000 games and doing the analysis locally, that he gets a bit short on memory. This is because all game data is loaded into RAM for the analysis. However, considering the problem for a bit, it can be easily solved with streaming analysis that requires basically no memory at all. The resulting stream processing pipeline we will create will be over 235 times faster than the Hadoop implementation and use virtually no memory.

Learn about the data

The first step in the pipeline is to get the data out of the PGN files. Since I had no idea what kind of format this was, I checked it out on Wikipedia.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia Yugoslavia|JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]
(moves from the game follow...)
```

We are only interested in the results of the game, which only have 3 real outcomes. The 1-0 case means that white won, the 0-1 case means that black won, and the 1/2-1/2 case means the game was a draw. There is also a - case meaning the game is ongoing or cannot be scored, but we ignore that for our purposes.

Acquire sample data

The first thing to do is get a lot of game data. This proved more difficult than I thought it would be, but after some looking around online I found a git repository on GitHub from rozim that had plenty of games. I used this to compile a set of 3.46GB of data, which is about twice what Tom used in his test. The next step is to get all that data into our pipeline.

Build a processing pipeline

If you are following along and timing your processing, don't forget to clear your OS page cache as otherwise you won't get valid processing times.

Shell commands are great for data processing pipelines because you get parallelism for free. For proof, try a simple example in your terminal.

```
sleep 3 | echo "Hello world."
```

Intuitively it may seem that the above will sleep for 3 seconds and then print `Hello world` but in fact both steps are done at the same time. This basic fact is what can offer such great speedups for simple non-IO-bound processing systems capable of running on a single machine.

Before starting the analysis pipeline, it is good to get a reference for how fast it could be and for this we can simply dump the data to `/dev/null`.

```
cat *.pgn > /dev/null
```

In this case, it takes about 13 seconds to go through the 3.46GB, which is about 272MB/sec. This would be a kind of upper-bound on how quickly data could be processed on this system due to IO constraints.

Now we can start on the analysis pipeline, the first step of which is using `cat` to generate the stream of data.

```
cat *.pgn
```

Since only the result lines in the files are interesting, we can simply scan through all the data files, and pick out the lines containing 'Results' with `grep`.

```
cat *.pgn | grep "Result"
```

This will give us only the `Result` lines from the files. Now if we want, we can simply use the `sort` and `uniq` commands in order to get a list of all the unique items in the file along with their counts.

```
cat *.pgn | grep "Result" | sort | uniq -c
```

This is a very straightforward analysis pipeline, and gives us the results in about 70 seconds. While we can certainly do better, assuming linear scaling this would have taken the Hadoop cluster approximately 52 minutes to process.

In order to reduce the speed further, we can take out the `sort | uniq` steps from the pipeline, and replace them with AWK, which is a wonderful tool/language for event-based data processing.

```
cat *.pgn | grep "Result" | awk '{ split($0, a, "-"); res = substr(a[1], length(a[1]), 1); if (res == 1) white++; if (res == 0) blac
```

This will take each result record, split it on the hyphen, and take the character immediately to the left, which will be a 0 in the case of a win for black, a 1 in the case of a win for white, or a 2 in the case of a draw. Note that `$0` is a built-in variable that represents the entire record.

This reduces the running time to approximately 65 seconds, and since we're processing twice as much data this is a speedup of around 47 times.

So even at this point we already have a speedup of around 47 with a naive local solution. Additionally, the memory usage is effectively zero since the only data stored is the actual counts, and incrementing 3 integers is almost free in memory space terms. However, looking at `htop` while this is running shows that `grep` is currently the bottleneck with full usage of a single CPU core.

Parallelize the bottlenecks

This problem of unused cores can be fixed with the wonderful `xargs` command, which will allow us to parallelize the `grep`. Since `xargs` expects input in a certain way, it is safer and easier to use `find` with the `-print0` argument in order to make sure that each file name being passed to `xargs` is null-terminated. The corresponding `-0` tells `xargs` to expected null-terminated input. Additionally, the `-n` how many inputs to give each process and the `-P` indicates the number of processes to run in parallel. Also important to be aware of is that such a parallel pipeline doesn't guarantee delivery order, but this isn't a problem if you are used to dealing with distributed processing systems. The `-F` for `grep` indicates that we are only matching on fixed strings and not doing any fancy regex, and can offer a small speedup, which I did not notice in my testing.

```
find . -type f -name '*.pgn' -print0 | xargs -0 -n1 -P4 grep -F "Result" | gawk '{ split($0, a, "-"); res = substr(a[1], length(a[1]
```

This results in a run time of about 38 seconds, which is an additional 40% or so reduction in processing time from parallelizing the `grep` step in our pipeline. This gets us up to approximately 77 times faster than the Hadoop implementation.

Although we have improved the performance dramatically by parallelizing the `grep` step in our pipeline, we can actually remove this entirely by having `awk` filter the input records (lines in this case) and only operate on those containing the string "Result".

```
find . -type f -name '*.pgn' -print0 | xargs -0 -n1 -P4 awk '/Result/ { split($0, a, "-"); res = substr(a[1], length(a[1]), 1); if (
```

You may think that would be the correct solution, but this will output the results of **each** file individually, when we want to aggregate them all together. The resulting correct implementation is conceptually very similar to what the MapReduce implementation would be.

```
find . -type f -name '*.pgn' -print0 | xargs -0 -n4 -P4 awk '/Result/ { split($0, a, "-"); res = substr(a[1], length(a[1]), 1); if (
```

By adding the second awk step at the end, we obtain the aggregated game information as desired.

This further improves the speed dramatically, achieving a running time of about 18 seconds, or about 174 times faster than the Hadoop implementation.

However, we can make it a bit faster still by using mawk, which is often a drop-in replacement for `gawk` and can offer better performance.

```
find . -type f -name '*.pgn' -print0 | xargs -0 -n4 -P4 mawk '/Result/ { split($0, a, "-"); res = substr(a[1], length(a[1]), 1); if
```

This `find | xargs mawk | mawk` pipeline gets us down to a runtime of about 12 seconds, or about 270MB/sec, which is around 235 times faster than the Hadoop implementation.

Conclusion

Hopefully this has illustrated some points about using and abusing tools like Hadoop for data processing tasks that can better be accomplished on a single machine with simple shell commands and tools. If you have a huge amount of data or really need distributed processing, then tools like Hadoop may be required, but more often than not these days I see Hadoop used where a traditional relational database or other solutions would be far better in terms of performance, cost of implementation, and ongoing maintenance.

Adam Drake leads technical business transformations in global and multi-cultural environments. He has a passion for helping companies become more productive by improving internal leadership capabilities, and accelerating product development through technology and data architecture guidance. Adam has served as a White House Presidential Innovation Fellow and is an IEEE Senior Member.