Blogpost - Optimising searching with grep

In a world with exponential amount of stored data, data searching is becoming more of a bottleneck. This leads to an increased amount of required compute power and time used on searches. However, in most cases you end up with subpar solutions not utilizing the tools and machines full potential. Addressing this issue can result in quicker and cheaper searches when you need to search through data.

We have approximately 37400 books in which we need to search through and find out what books have which cities mentioned. We have approximately 48900 cities to search for. So we wanted to find the quickest way to perform this task, since its 1.8 billion combinations, where each combination holds thousands of words to search through. So we will dive into how to optimize our grep search, to obtain speeds and a clean results set.

Please notice

Everything we cover in this blog post is mainly aimed at improving timings in certain use cases, and may not be the best approach for all use cases.

All of our test and findings are done on a server with the following specs, 2x6 core CPU (3,33 GHz), 96GB RAM, a RAM Disk and a clean Ubuntu 18.04 install. This is both the timings and the benchmarks.

Prerequisites for this blog

The prerequisites both covers if you want to play around with the commands and just general knowledge of what this is.

- Knowledge about what is grep
- A UNIX system if you want to test along

If your knowledge about grep is non existing, we would recommend to read this[1] blog post.

For the UNIX based system we would recommend Ubuntu for simplicity, or if you rather want to stick to Windows it should be Windows 10 with WSL. It can be read about here[2].

Terms used

- keyword : The specific word we are looking for

grep

Our grep command started out looking like this, `grep 'keyword'`.

Search in many files

First off we wanted to search a whole directory full of text files and not just a specific file for our keyword, and since grep is used to search text, and we want to search multiple files. We would need a way to read all the files in a directory recursively, and lucky for us this is baked right into grep as `-R, -r or --recursive`.

Now our grep command looks like this `grep -r 'keyword'`, and using it gets us a search time that is 25.8 seconds. This is searched through our total of 37400 books. When looking at the time, 25.8 seconds through

37400 books it is not bad, but we have a total of 48900 cities to search for. That totals to 14.6 days of searching which is horribly slow.

Optimising search in files

Now that we enabled our self to search in our whole directory of files, we found out how long it took to search for a city, and all of our cities. We needed a way to make this search quicker, so we looked through the man page[3] for grep. Here we found a option called `-l or --files-with-matches` that had the following 2 attributes, suppress output; instead of printing the line of where the match is, it now only prints the file name. The second attribute which in our case was one of the biggest improvements, it will stop searching after it matches in the file.

Now our grep command looks like this `grep -rl 'keyword'` and using it gets us a search time that is 9.8 seconds, this is a improvement on 16 seconds / a 62% decrease in search time. Furthermore it also gives us a better format to save to files for further data processing.

Cleaning our result set

There is a problem with grep that needs to be addressed in our use case. It matches the keyword as long as its in the text without checking if its in another word. So e.g. if we search for `London` it will also match on `Londonderry`, which is not what expected to get as return.

We tried to fix this problem by making a regex pattern to match on `grep -rl -e '[\ \n,.<(\ []London[.,!?<>;:"]'`, it covers most use cases, however there are some it does not catch but its edge cases. We tested this command to see the timings, it gave a average runtime of 42.07 seconds which is a 329.29% decrease in performance compared with our 9.8 second timings we did with `grep -rl 'keyword'`. However we get a cleaner result set, which we also need in this case.

This new timing on 42.07 seconds is horrible and we are at the point now, where it again takes multiple weeks to search through our books. Once again we went diving into the man page[3] hoping for natural implemented way to fix our problem. We found the option `-w or --word-regexp` which did the same as our regex search, while also fixing the worst of our edge cases. We timed our new command `grep -rlw 'keyword'` and got timings down on our `grep -rl 'keyword'` timings again, which is around 9.8 seconds.

Case sensitivity

We did not do anything about case sensitivity since city names supposed to be capitalised, but if its something you would like to use yourself it can be read about here.[4]

grep benchmarks

The way performed our benchmarks is by running it multiple times. Our single threaded test has been run 100 times on each city. Since we had to test so many amount of threads in our multi threaded benchmark, and we saw the results from our single threaded benchmark, we decided to lower the amount of test to 10 on each.

We also decided to test our parallelization 10 times each.

We choose a best case city 'London' something in the middle 'Berlin' and a worst case 'Odense'.

All of our benchmark data is located in this folder[5]

Single threaded

The numbers used for the graph is in the run_time_data folder.

On the image we can see 3 test groups. Group 1 is London, Group 2 is Berlin and Group 3 is Odense.

The data shown in the graph display a very consistent run time with a standard deviation of 0.12 in Group 1, 0.23 in Group 2 and 0.23 in Group 3. If we look at the results of the different grep commands we can see that London is mentioned in far more books than Berlin is, and Berlin is mentioned in far more books than Odense is - this correlate well with our runtimes as we skip to the next book after the first match due to the `l` flag in grep, and therefore it skips searching more text in the less often mentioned city names.

Multi threaded

One way of improving the performance is to utilize the resources more efficiently. By default grep will only use one thread at a time. There is a way to force grep to use more threads by using `xargs`. You can read more about `xargs` here[6]

In order to force grep to utilize more than one thread we append `find . -type f -print0 | xargs -0 -P $threadCount` in front of our grep command.

We benchmarked our multi threaded command line to figure out if it was worth it to use more threads on the same task.

Here group 1 to 10 represent the amount on threads and 11 is 20 threads. As we can see there is no real performance gains to get from using 4 to 20 threads.

As seen on the graph we get some substantial performance gains by threading our process up until 4 threads. Going from 1 to 2 threads gave us a 41.1% decrease in search time. Going from 1 to 3 threads gave us a 58.1% decrease in search time. Going from 1 to 4 threads gave us a 69.8% decrease in search time.

The results show a significant diminishing returns for each thread we add to the process, and no performance gains above 4 threads (this point might shift based on the clock frequency of the given CPU)

Parallelization

Another way of utilize the resources more efficiently, is to run multiple instances side by side i.e. parallelization. But does parallelization reduce the performance of the single instance by creating a bottleneck on the disk?

We benchmarked parallelization to figure out what effect running multiple searches at the same time would have on our data set.

Running 2 in parallel increased the search time by 1.1%, 9.91 seconds. Running 3 in parallel increased the search time by 1.9%, 9.98 seconds. Running 4 in parallel increased the search time by 2.1%, 10 seconds. Running 24 in parallel increased the search time by 24.5%, 12.2 seconds.

As seen above we lose a bit performance by running it in parallel, however the lost performance is not worth crying over when looking at the speeds you finish searches. At 24 threads we get 24 searches, but losing 24.5% search time on each. In the case of running 24 parallel searches with our 48900 cities would the total compute time 165.8 hours, spread across 24 threads would give a 6,9 hours.

Running 12 searches in parallel with 2 threads allocated to each instance takes 6.8 seconds which is quite fast, but compared to running 24 threads with 24 searches which takes 12.2 seconds it would take 13.6 seconds to obtain the same result. In the case of running 12 searches in parallel with 2 threads allocated to each instance with our 48900 cities would the total compute time 93.1 hours, spread across 12 parallel instances would give a 7,76 hours.

Thereby we save 0,85 hours by only using a single thread 24 cores compared to running 12 parallel searches with 2 threads allocated to each.

How to reproduce our benchmarks

We created bash scripts for easier execution and reproducibility, however we couldn't get the bash script with multithreading to work properly so its tested manually by running the command X amount of times in quick succession.

For single thread test run this[7] bash script, and for multi threaded take the command from this[8] bash script, and put the amount of threads where it says `$i`. For parallelization test run this[9] bash script. Remember to change `24` to the amount of threads you want to test with, in both occurrences.

As seen on our specs the server we had at our hand had a high RAM amount so we decided to create a RAM disk since we only had a HDD to remove that bottleneck. This might skew the results a bit compared to a HDD or SSD test.

Conclusion

As we seen its worth it to take time looking through the man page (or just a manual) of the tool you are using. We managed to save 16 seconds by using built in options compared to our own implementations.

By our benchmark we learn several things, if we have a lot of things to search for (in our case cities) it is better to have many single threaded greps running in parallelization, but if we have less things than threads on the system it is beginning to be worth it to multi thread our greps. As long as we can fill out the total amount of threads its worth it to multi thread, however it is not worth it to put more then 4 threads (in our case) on a single grep search at a time as seen in the multi threaded section.

Written By

David Carl & Tjalfe Møller

https://github.com/DavidCarl/UFO/

Reference

1: https://www.maketecheasier.com/what-is-grep-and-uses/

2: https://www.computerhope.com/issues/ch001879.htm

3: https://linux.die.net/man/1/grep

4: http://droptips.com/using-grep-and-ignoring-case-case-insensitive-grep

5: https://github.com/DavidCarl/UFO/tree/master/run_time_data

6: https://shapeshed.com/unix-xargs/

7: https://github.com/DavidCarl/UFO/blob/master/run.sh

8: https://github.com/DavidCarl/UFO/blob/master/RunThreads.sh

9: https://github.com/DavidCarl/UFO/blob/master/parallelization.sh