PA2577 Working with a Stream of Data

Samuel Jonsson

December 14, 2023

1 Question 1

Are you able to process the entire Qualitas Corpus? If not, what are the main issues (think in terms of data processing and storage) that cause the CodeStreamConsumer to hang? How can you modify the application to avoid these issues?

It would be difficult to process the entirety of the Qualitas Corpus, at least on my home PC. In fact, the program seems to stop processing around 23900 files read, likely due to the sheer size of the data needing to be processed more or less at the same time towards the end. I would say it's theoretically possible to do it, however not with the average PC's capabilities...

The way that the CloneDetector seems to be doing it right now is to compare line-to-line (or, to be more exact, chunk-by-chunk) in two files. This method brings upon it three main issue points:

1. Performance Issues: CloneDetector, or more specifically the CodeStream-Consumer component, suffers from performance problems. It hangs during processing, even when dealing with relatively small datasets. This issue can significantly impact its usability for handling actual Big Data workloads.

2. Memory Consumption: CodeStreamConsumer uses a FileStorage class, which keeps all processed files in memory. Additionally, it maintains a "chunkified" version of each file in memory as well. This approach effectively doubles the memory footprint for each file, which can lead to high memory usage and potential memory-related problems.

3. Suboptimal Resource Utilization: In many cases, the original file is not needed once the initial processing is completed.

The main points of utilisation I can see are related to the latter two. These can be improved in various ways, including:

- **Hashing**: Instead of comparing `SourceLines` directly, calculate hash values (e.g., checksums or cryptographic hashes) for each `SourceLine`. Then, compare the hash values first, and only if they match, perform a detailed line-by-line comparison. Hashing can significantly reduce the number of full text comparisons.

- **Tokenization**: In a way similar to hashing, one can tokenize the `SourceLines` into smaller units, such as words or symbols, using a lexer or tokenizer.

Then, compare the tokenized representations of the lines. Tokenization can help identify partial similarities without comparing entire lines.

- **Caching**: One can implement caching mechanisms to store previously compared chunks or lines. If the same chunks appear in different files, reuse the comparison results to avoid redundant work. To more efficiently one can implement an algorithm that sorts out the most commonly found clone elements, although it might not be too applicable to this program since it might be that every clone is unique.

2 Question 2

Comparing two chunks implies `CHUNKSIZE` comparisons of individual `SourceLines`. What can be done to reduce the number of comparisons?

A way I can consider is to implement hashing of the `SourceLines` to make them easier to compare and store, and then implement a hash table with the structure $<$ hash, file $>$. You need then only to look what hashes has more than one file source and investigate those.

3 Question 3

Studying the time it takes to process each file, do you see any trends as the number of already processed files grow? What may be the reasons for these trends (think in terms of the data processing algorithms)?

Other than what has already been mentioned in earlier questions, one can observe the graph over the average total processing time/number of files, as presented in Figure 1, one can see that the more files processed, the longer the average time becomes. This is natural considering the bubble sort-like way the algorithm is tackling the problem. Since every file is compared to every file before it, it will mean that for every file you have checked, there is another file to check, for every ten, there are ten more. The increase is (almost) linear, if you ignore the outliers, that are likely due to the algorithm having to look through more/less files before finding anything. There are also no early stop conditions for the algorithm as we want to look through all the code files to find any clones. This is another aspect that makes it like bubble sort.

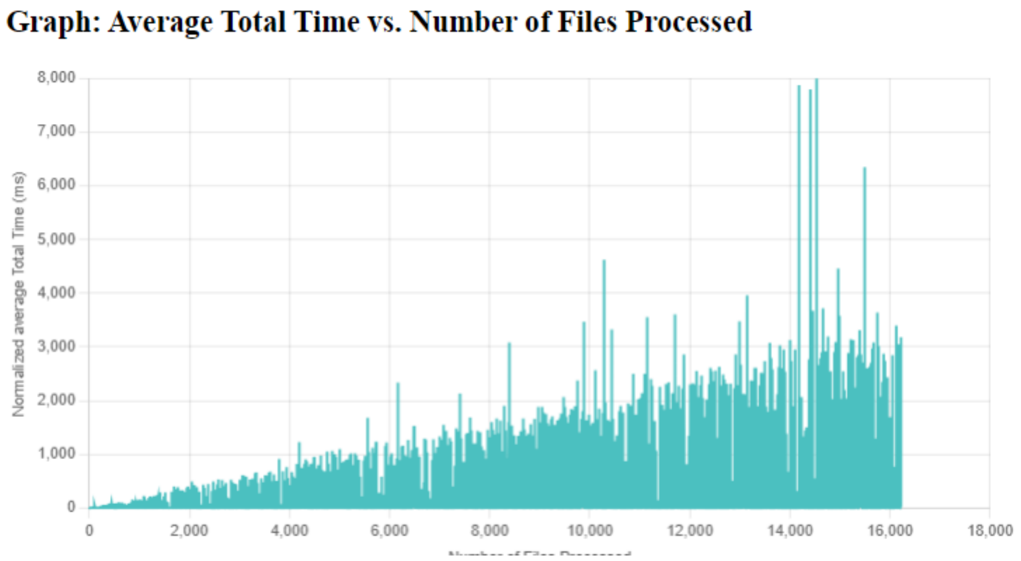

Figure 1: Plot for average match time over the number of files