Log Extractor: Project Description

Concepts involved

We assume that the DB is file-based. We then perform 2-level indexing of database to process queries quickly. The first level of indexing stores the timestamp of the first log line for each log file. For any new data that gets stored in the database, its index can be inserted in this index table accordingly. This practice helps the program identify log files corresponding to start and end timestamps of the query range, using binary search on timestamps. We use dense indexing for this level since the total number of log files is 18203 and the binary search will take O(log (18203)) time in the worst case.

In the second level of indexing, we use sparse indexing for each log file. We provide the functionality that enables the user to vary the size of each index block. We maintain an index table for each log file where we store one representative timestamp from each log block and its corresponding row number.

The log file corresponding to start timestamp is obtained via 1st-level indexing as explained above. Then, we use the 2nd-level index table to carry out a binary search to reach the desired block, which contains the start timestamp. Linear search will then be used to traverse the block to find the exact query line corresponding to (or just higher than, if exact start timestamp value doesn't have a corresponding log in the log file) the start timestamp. This method is much faster than iterating through the log file linearly and searching for the line corresponding to the start timestamp. Thus, 2nd-level of indexing saves a lot of time in searching for the exact row for the start timestamp in the log file.

We need to access multiple log files in the database to output log lines. For conquering this and minimising latency, we use multi-threading. The program will use one thread per log file to query logs concurrently, thus reducing engineer's wait time.

Performance Analysis

Test data used: 20 log files, each log file with 1000 log lines. Each file is of 82 KB, and the entire DB becomes 16.4 MB.

Test machine specs: Lenovo B50, Intel Core i5-4200U; 2 CPU cores with Hyper-threading. Hence, a maximum of four threads can run in parallel.

Performance: Querying entire DB (2 x 10^5 log lines) on my local machine takes ~ 12.55 seconds.

Typically, a log extractor would run on a server. On a server with 40 CPU cores, at max 80 threads can be run in parallel. Hence, total output time would become **0.6275 seconds**.

Challenges

We have a 285 TB database of log files, majorly stored in archives, with only the recent logs in a DBMS for faster access. Therefore, this would typically involve a distributed systems architecture. In such a system, there are multiple servers, each writing data into their respective databases independently. To query logs in a specific time range, we can have a master server where the query would be sent. The master server then queries all other servers to process their respective logs in the queried time range. To output log lines, one option is all servers can output their logs on servers itself, independently. The second option is to request queried logs from all servers to the master server and then use a streaming service to output logs concurrently on the master server. Implementation of above-described architecture is beyond the scope of the current project. Therefore, for the current scenario, it is safe to assume the entire database exists on a single server. This distributed form of computation can be enabled in the future versions of the application.

Moreover, the indexing methodology can be made more robust, handling different forms of query fields and scaling to a higher degree of throughput.