CS250B Homework 1: Optimizing Vector DB

Due February 28

In this homework, you will attempt to apply what we discussed during the lectures to implement the best possible performance of vector databases. You are free to try whatever approach you want, while being unsure of what machine your code will eventually be deployed to. (So, a typical development scenario)

The provided code, among some support functionality, implements a simplified version of vector DB search. The simplified operation of the target application is the following:

The program is given a file of key-value pairs which should form the database, and another file of key-value pairs which is the query. The task is to construct a data structure to store the database in memory, and then process the queries. A working version of this functionality is already implemented in the provided code.

Querying consists of two parts. Each query is a range query, consisting of [start, end] key pairs and a value array. The algorithm should check all values in the database with the appropriate key, and then perform distance calculation between the query's value array, and only keep the minimum distance per query. The sum of all distances should be returned for printing. The distance function to use is Manhattan distance, for simplicity.

Again, a working version of this code is provided in the provided code, so please consult its behavior.

Implementation Details

Run "make". This will generate two binaries: datagen and traverse. datagen is used to generate the workload, and traverse is the actual program.

datagen.cpp implements the datagen program. It takes parameters to specify the scale and distribution of the data. You can run the command without any parameters for the default dataset.

main.cpp executes the traversal function, which is implemented in **tree.cpp.** Multithreading is also implemented. You are expected to primarily work inside tree.cpp. No other file (probably) needs to be edited. If you must edit other files (e.g., Makefile) please include that in your submission.

Please do not change main.cpp. If you feel you must, let's discuss!

Deployment System

The only assurance given about the deployment/evaluation system is that it is an x86 server new enough to support AVX2. (So, perhaps not the best idea to develop on an ARM machine.) There is no assurance on DRAM performance or details about the cache hierarchy. Although it is likely it will also share some

typical characteristics such as the size of the L1 cache. Off-chip DRAM bandwidth may be fast (e.g., 4 DDR4 cards) or slow (e.g., 1 DDR4 card).

Recommended Approaches

The default run will involve running

./obj/datagen 24 7

This will generate two files: “bfile_24_64.dat” (database) and “qfile_24.dat” (query)

Then you can run

./obj/traverse ./bfile_24_64.dat 64 ./qfile_24.dat 8

To specify the input files, the size of the value array (the vector in the vector database) to be 64, and the number of threads. Do not change the value array size from 64 unless you changed it in datagen. The generated dataset’s value array size is specified in the generated filename. (e.g., “bfile_24_**64**.dat”)

You can improve the performance of this program on two fronts: Parallelism and Cache access.

Parallelism is probably the easiest to get started on, since implementing Manhattan distance with AVX should be straightforward.

Then, the cache behavior of the tree traversal must be improved, based on what we discussed during class. Keep in mind that high cache optimization is necessary for the off-chip memory to not bottleneck the system with many threads. A less cache-optimized version may function competitively with a single thread, but may cause the backing DRAM to become the bottleneck with many threads.

What to Submit

Only submit tree.cpp, and a short document briefly describing your approaches.

Caveats and Grading

This project is new, and I don’t (yet) have clear knowledge about the upper limit on the performance we can achieve. Bear in mind I may change many things about the generated data during evaluations, for example, range of the range queries, sparsity of the dataset, etc.

Grading will be done via an anonymized derby, meaning we will compare the performance of all submissions (and some of my reference implementations) with names hidden. But don’t worry, as long as there is reasonably effort the grades will be favorable. The derby is less for grading on a curve, and more for comparing the effects of various optimization attempts using concrete implementations.