Laboratory 1: Impedance Mismatch

Explanatory document

Aashish Bhusal & Berta Ferré Segura

Introduction

The aim of this lab is to compare PostgreSQL and Chroma in terms of handling text and embeddings, analyzing the impedance mismatch and performance differences.

For this assignment, we needed to load the dataset *BookCorpus* from *HuggingFace* and split it into 10000 sentences. To achieve this, we created a script named *prepareData.py* that creates a CSV file named *bookcorpus10k.csv*, which we worked with.

All scripts and datasets that we used for this assignment can be found at our GitHub repository: https://github.com/bfs14/CBDE-L1

PostgreSQL

To do the PostgreSQL part, we used two schemas, one is bookcorpus(id SERIAL, text TEXT) for P0 as we only had to load the text chunks into the database and the other one is sentence_embeddings(sid INTEGER PRIMARY KEY, embedding JSONB) which we used to store the 384-D vectors as JSONB because we couldn't use *Pgvector*.

P0 Script

First of all, while writing the script for P0 we did the insertion with one sentence per INSERT and a commit per row to calculate the minimum, maximum, standard deviation and average time for storing the textual data.

Result of two executions for P0 script:

Insertion times (TEXT)
Min: 1.741 ms
Max: 38.478 ms
Average: 2.852 ms
Standard Deviation: 3.550 ms

Insertion times (TEXT)
Min: 1.795 ms
Max: 37.312 ms
Average: 3.122 ms
Standard Deviation: 4.332 ms

We obtained less stable results and a very high MAX and standard deviation values which we concluded to be because of the overheads caused by our choice of using commits per row.

P1 Script

For the script of P1 we encoded all sentences with all-MiniLM-L6-v2 and stored the vector embeddings as JSONB. As commit per row was not a suitable choice in P0 here we decided to use execute_values with batches of 200 and do one commit per batch which resulted in a lower standard deviation value and lower min max average values than P0.

Result of two executions for P1 script:

Time taken to encode 11239 sentences: 26.505 seconds

Insertion times (JSONB)
Min: 0.440 ms
Max: 0.692 ms
Average: 0.497 ms
Standard Deviation: 0.055 ms

Time taken to encode 22478 sentences: 40.904 seconds

Insertion times (JSONB)
Min: 0.248 ms
Max: 0.459 ms
Average: 0.285 ms

Standard Deviation: 0.045 ms

As we can observe, the batch insertion produces much more stable times compared to P0, showing how schema design and insertion method directly impact the performance and reduce some of the impedance mismatch issues when storing embeddings in PostgreSQL.

P2 Script

For P2 we had to compute the top-2 similar sentences for the 10 chosen sentences and as we already saved all sentences and it's embeddings in postgres during P0 and P1 scripts we loaded everything at once from the Postgres i.e the sentences id's, texts and their embeddings and we converted those embeddings into a PyTorch tensor so we could do the math quicker.(*The reference to this idea is mentioned in the reference section below*). Then for each of the 10 sentences we calculated the dot product for Cosine calculation and computed the euclidean distance from each sentence to every vector. Below are the results of execution of P2 script.

Result of two executions for P2 script:

Cosine query times (ms)
Min: 0.463 ms
Max: 3.762 ms
Average: 0.865 ms
Standard Deviation: 0.972 ms

Cosine query times (ms)
Min: 0.301 ms
Max: 3.703 ms
Average: 0.896 ms
Standard Deviation: 0.982 ms

Euclidean query times (ms)
Min: 1.557 ms
Max: 3.817 ms
Average: 2.097 ms
Standard Deviation: 0.741 ms

Euclidean query times (ms)
Min: 1.215 ms
Max: 12.524 ms
Average: 2.777 ms
Standard Deviation: 3.284 ms

We observed the cosine query times to be much faster than the Euclidean one which we think is because of more mathematical calculations while doing the comparisons.

PQ1 Questions

1) Are the insertion times for text and embeddings stable? What are your conclusions on this matter?

The insertion time for text was not very stable, we had a standard deviation of around 3.55ms or 4.33ms as we were inserting one row and doing one commit per row which created overheads. On the other hand for Embeddings we got a better result as we did the inserts in batches of 200 rows and a single commit per batch which resulted in a much faster insertion.

2) Are the querying times stable when computing the similarities? Did you find differences between the two distance metrics you used? What are the conclusions on this matter?

The querying times were stable when computing the similarities. We observed the differences between the two distance metrics as we found out that cosine was faster which was because of the use of already normalized vector from P1 and all we had to do was a dot product where as on the other hand the Euclidean metrics had to do much more calculations than the Cosine metrics for each comparison.

3) Can you think of any available insertion method, data structure or indexing technique available in PostgreSQL (Pgvector is still not allowed to be used yet) that would improve the performance of these operators?

After evaluating the results we think for faster insertion we could use batch inserts and a commit per batch instead of per row.

Chroma

For this part of the lab, we set up a local Chroma instance and connected to it via Python. The goal was to load the same chunks of the *BookCorpus* used in PostgreSQL, generate embeddings and perform top-2 similarity queries.

C0 Script

First of all, we employed a *DummyEmbedding* so that embeddings wouldn't be computed, isolating the storage cost of textual data. We also inserted batches of sentences and recorded execution times.

Result of two executions for C0 script:

Insertion times (TEXT)
Min: 0.149 ms
Max: 0.249 ms
Average: 0.189 ms
Standard Deviation: 0.022 ms

Insertion times (TEXT)
Min: 0.139 ms
Max: 0.237 ms
Average: 0.177 ms
Standard Deviation: 0.025 ms

As we can see, the results show extremely fast insertion times for text and standard deviations were also low, indicating a very stable performance.

C1 Script

This script handles the insertion of text and embeddings, where Chroma computes embeddings internally using *SentenceTransformerEmbeddingFunction*. We also inserted batches of 200 sentences in order to improve the throughput.

Result of two executions for C1 script:

Insertion times (ChromaDB)
Min: 1.855 ms
Max: 4.473 ms
Average: 2.609 ms
Standard Deviation: 0.573 ms

Insertion times (ChromaDB)
Min: 1.799 ms
Max: 4.841 ms
Average: 2.775 ms
Standard Deviation: 0.628 ms

The obtained results show a slightly higher insertion times compared to C0. Therefore, while embedding computation introduces additional time, insertion remains stable and efficient.

C2 Script

This script computes the top-2 similar sentences for the 10 chosen sentences, the same ones used in PostgreSQL scripts. For each sentence, we compute the Cosine similarity and the Euclidean distance to every vector.

Result of two executions for C2 script:

Cosine query times (ms)
Min: 1.188 ms
Max: 3.115 ms
Average: 2.625 ms
StdDev: 0.627 ms

Cosine query times (ms)
Min: 1.037 ms
Max: 8.853 ms
Average: 2.013 ms
StdDev: 2.286 ms

Euclidean query times (ms)
Min: 400.739 ms
Max: 430.898 ms
Average: 416.504 ms
StdDev: 10.046 ms

Euclidean query times (ms)
Min: 425.416 ms
Max: 507.847 ms
Average: 466.980 ms
StdDev: 21.585 ms

The results show a clear difference between both metrics. The Euclidean query times are significantly slower than the Cosine query times, showing that Chroma is highly optimized for Cosine similarity but requires manual computation for other metrics.

CQ1 Questions

1) Are the insertion times for text and embeddings stable? What are your conclusions on this matter?

The text insertion time was stable due to batch insertion, and the embedding insertion time was also stable, with variability mostly coming from embedding computation rather than storage. So we can conclude that Chroma handles batch insertion efficiently, and it's possible to separate text insertion from embedding generation.

2) Are the querying times stable when computing the similarities? Did you find differences between the two distance metrics you used? Is it possible, in Chroma, to measure the insertion of text and embeddings creation separately? What are the conclusions on this matter?

The times for cosine similarity queries were stable and fast, while the times for euclidean queries, computed manually, were much slower. Therefore, the choice of metric really affects query performance in Chroma.

3) Can you think of any available insertion method, data structure or indexing technique available in Chroma that would improve the performance of these operators?

Adjusting batch size or enabling disk persistence would improve the performance of these operators.

Discussion

PostgreSQL

While using the PostgreSQL, the impedance mismatch was high. As we saw during our execution that P0 where we used commits per row the stability was low and P1 was much more stable and faster per row because of batched inserts, P2 queries were stable too but we had to use PyTorch and almost all the work of calculations was not done in the database. So, in conclusion, it can store vectors as we did using JSONB, but it cannot compute the similarities efficiently on its own and needs external support for the calculation as we did using PyTorch to get the embeddings for calculations.

Pros: It is better for text and embeddings insertion and calculations if we make a better choice of schema, data structures and embedding techniques.

Cons: There is a high impedance mismatch and we need extra larger codes to get the data from the DB for calculations of similarities.

Chroma

The impedance mismatch is low in comparison to PostgreSQL, as it is a vector native DB used for storing and retrieving vector embeddings. As we saw in C0, the inserts were much stable. C1 had a slightly higher access time, but with the right approach this also seems to be avoidable. The cosine queries were much faster and stable in C2, whereas the Euclidean was higher.

Pros: We can say that Chroma works better with less code in comparison to PostgreSQL for the work we did. Also, we did not have to pull the embeddings data out as in pyTorch for the calculation of similarity.

Cons: Euclidean similarity queries are slow since Chroma is optimized for cosine similarity and other distance metrics require manual computation. Furthermore, Chroma provides less support for complex relational queries.

References

Video link to the python script watched to learn how to load the dataset from huggingface and split it into 10000 sentences or 10k rows necessary for the lab: https://www.youtube.com/watch?v=-svlg240JXk

Reference for *P0.py* code : https://www.dataquest.io/blog/loading-data-into-postgres/

Reference for *P1.py* code:
https://ai.google.dev/gemma/docs/embeddinggemma/inference-embeddinggemma-with-sentence-transformers
https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2?

Reference for *P2.py* code:
https://www.pinecone.io/learn/vector-similarity/
https://spotintelligence.com/2022/12/19/text-similarity-python/ (section of text similarity with pytorch)