Benchmarking Vector Databases on Code Embeddings

Vikram N. Subramanian
University of Waterloo

Raymond Chang
University of Waterloo

Yahya Jabary
University of Waterloo

Abstract

Vector databases are experiencing a major surge in interest thanks to their popularity in Large Language Model (LLM) based applications. However existing performance comparisons of vector databases are limited. This project contributes the first (to the best of our knowledge) large high dimensional embeddings data set for benchmarking. It is also the first data set generated from source code. Using our dataset we ran benchmarks on four popular vector databases: ChromaDB, MilvusDB, Weavite and Redis.

1 Introduction

Existing LLM applications use Retrieval Augmented Generation (RAG) to improve their performance. RAG involves providing to an LLM both a prompt and any semantically relevant documents. Determining whether a document is relevant is done by transforming both the prompt and any relevant documents into vector embeddings (embeddings) and comparing their similarity. Each embedding is an array of floats and is generated by feeding text, such as a prompt into an embeddings model.

Generally, the embeddings model will create embeddings such that if two documents are semantically similar, their corresponding embeddings will also be similar. Figure 1 shows an example of the query and retrieval process. For many LLM based applications, enormous amounts of embeddings must be stored and retrieval must be fast. Vector databases satisfy both of these requirements by offering efficient storage and retrieval of relevant embeddings [5].

The main contributions of this paper are:

1. A very large high-dimension (1024-dimension) data set
2. First code specific dataset to benchmark vector databases
3. Some preliminary benchmarking of 4 popular vector databases using above mentioned data set.

We didn't test established databases like Postgres because, as of December 2023, despite improvements like `pgvector` and `pg_embedding`, they lag in both performance and user convenience [3]. They struggle to meet the demands of the fast-evolving AI landscape.

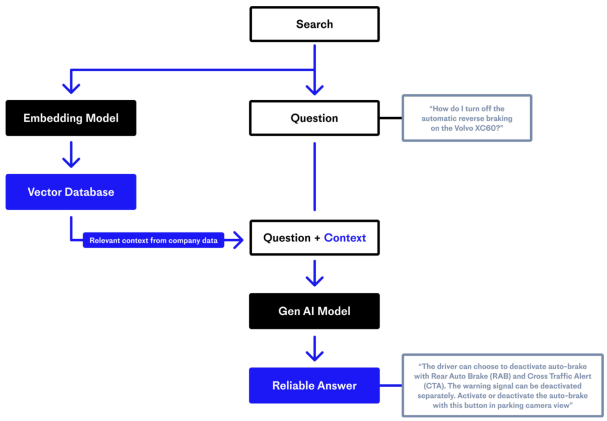

Figure 1: Retrieval Augmented Generation [5]

1.1 Why are mutations necessary?

Recall goes down as the load goes up in a vector database. This is because of the algorithm used to search in a vector database. This means, evaluating the accuracy of the results returned is an important metric in vector databases. To do this, we need access to the absolute truth of which entry is closest (in cosine similarity) to another entry in the dataset. The only way to do this is the manually compute the cosine similarity of every entry with every other entry, an $O(n^2)$ operation. With an embedding size of 1024 floats per vector, this results in around 3×10^{18} floating point operations, an extremely large amount. We estimate that this would take 100 Nvidia H100 GPUs an entire month to compute.

We solve this problem by using a novel application of mutations. We create mutated versions of snippets such that the snippets closest in cosine similarity to a code block are its mutations. See the mutation section for more information.

1.2 Determining similarity

A key requirement of vector databases is retrieving relevant embeddings given some query embedding q. This is done by measuring the similarity of q and the embeddings stored in the database. Several methods exist to determine similarity, however the one used in our benchmarks is cosine similarity. Equation 1 listed below describes how to calculate the cosine similarity of two vectors a and b.

$$CosineSimilarity = \frac{a \cdot b}{|a| * |b|} \tag{1}$$

1.3 Indexing

One algorithm used to create indexes in vector databases is Hierarchical Navigable Small Worlds (HNSW). In this algorithm, embeddings are represented as several layers of a proximity graph as shown in Figure 2. A search begins at the topmost layer at a designated entry node. The neighbors of the node at the current layer are then compared against the query vector and the node with the highest similarity is chosen as the next node. Eventually, the closest node in the current layer to the query vector is found (local minimum). The search moves down to the next layer and the search resumes. This process continues until the closest node to the query vector is found in the bottom layer [4].

The layers are constructed starting from the bottom layer. A probability function determines which layer each node belongs to. Most of the nodes will belong to the lowest layer and each successive layer will gradually have less nodes. Various parameters may be tuned during the construction process which can provide a trade off between recall and search time [4].

2 Design and Implementation

2.1 Generating a corpus of code documents

Our goal was to scrape at least half a million code files off the internet. As many open source projects are hosted on GitHub, we chose to source our corpus from there. Our first attempt to scrape GitHub involved using their API. However, the issue with this approach is the aggressive rate limiting imposed by GitHub. The rate limits are as follows:

- Anonymous client rate limit: 60 requests/h or over 8,333h in total
- Authenticated client rate limit: 5,000 requests/h or over 100h in total
- Enterprise client rate limit: 15,000 requests/h or over 33h in total

We tried different ways to get around Github's rate limits, but none worked:

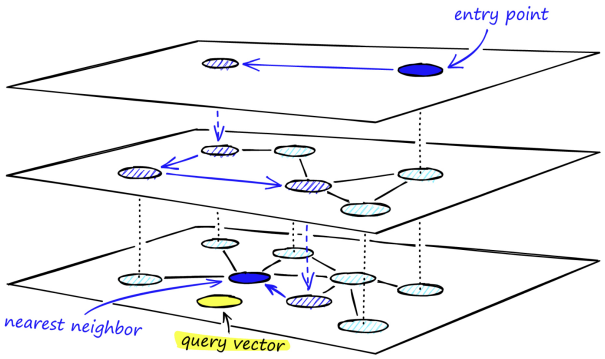

Figure 2: An example of an HNSW index. Each circle represents an embedding. Circles that are directly on top of each other represent the same embedding. Each plane in the diagram is a layer [4].

- Bandwidth Throttling: Set request limits and periods.
- Rotating IPs: Tried to exploit XFF vulnerability. Used rotating proxies from popular pools like the AWS API Gateway and BurpSuite IProtate.
- Fake Resource Paths: Used variations in resource paths like `%00`, `%09`, `%0a`, `%0c`, `%20`, etc., different endpoint naming formats and fake query parameters.
- Token rotation and ETag overwriting: Built requests on top of eachother.

We then resorted to web scraping and fortunately two alternative paths were found:

1. We used the `system size: > 100000` query to access the largest repositories on the GitHub "search" page. This was done manually, and around 500 links were collected before we concluded this approach was impractical.
2. We scraped by scrolling on different "topics" pages and clicking the "load more..." button. Each of these pages yielded 2000 repository links. This was equivalent to scraping 15 repository links per second.

Through the language "topics" page we were able to scrape around 5000 repository links. Next we cloned all of them, extracted relevant files and chunked them into blocks of 5000 files to commit to our public repository without being billed for GitHub-LFS. This process allowed for the successful scraping of over half a million files of code.

2.2 Mutating documents to create clusters

Next, we wanted to generate clusters of data through a process called mutation. The goal of mutations was to create semantically similar code blocks such that their corresponding embeddings will be similar.

The mutation process involves inserting blocks of dead code into each function. We generate five mutated versions for each function, with each successive mutation increasing in complexity. This can be achieved by mutating each function and inserting some dead code, thus preserving the semantics. For each function, we generate five mutated versions such that each successive mutant is more complex than the previous. Mutations have to be similar but not too similar to ensure recall drops.

Here's an example of a mutation:

```
# Original
def foo():
    ...

# Mutated
def foo():
    if False:
        i = 0
    ...
```

We were initially unsure about how complicated these mutations would have to be, so we built a fairly general solution. If the mutations were not complex enough, the database would not be challenged in identifying similar code snippets and our measurements would not be helpful. This is because a vector database's recall performance goes down as load increases (See HNSW section for more details). If the mutations were too strong, we would produce mutations with cosine similarities less than other code blocks and therefore would make our fundamental assumption wrong- the snippets closest in cosine similarity to a code block are its mutations.

We start by parsing each function and generating an Abstract Syntax Tree (AST). We then traverse the AST and upon finding a function, insert some dead code right at the beginning. We then dump these mutations and their originals into JSON files to be parsed later.

One of the measurements is how recall is affected by load on the database. We wanted to make sure that recall would actually decrease, so we had to make sure the mutations actually affected the cosine similarity. We did this by converting some mutations into embeddings and measuring them.

We first had to determine a baseline, which was the largest difference in similarity we would encounter. We did this by comparing the similarity of the two largest functions, which are likely to be the most different semantically, and got a cosine similarity of 0.68. We then checked the similarity of the smallest function with its mutations as well as the largest function and found that their similarity does actually differ. The smallest function had a similarity range of 0.82 to 0.92 with its mutations, while the largest function had a range of 0.85 to 0.91.

This analysis confirms that our mutation process is effective in generating semantically similar but distinct functions, which is crucial for our goal of data clustering.

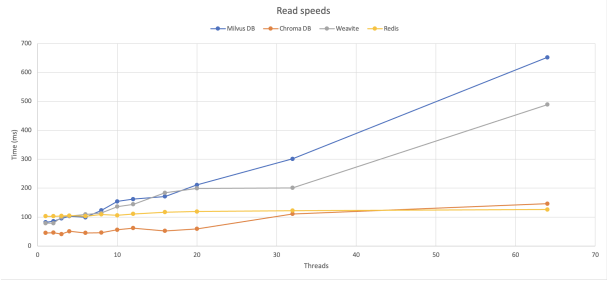

Figure 3: Read speeds

2.3 Generating embeddings from our mutations

We use the GTE-Large [2] embedding model to produce mutations. As of writing this paper, it is one of the top-performing embedding models in semantic search datasets/challenges such as the MTEB dataset [1]. It produces embedding of size 1024. We run this model on an Nvidia A6000 GPU. All the mutations and their code snippets took approximately 13 hours to be computed.

3 Evaluation

We benchmarked four databases: ChromaDB, MilvusDB, Weavite and Redis with our generated embeddings. We evaluated them on three different metrics: read speeds, write speeds and recall vs load.

3.1 Read speeds

Figure 3 shows the results of our read speed benchmarks. We noticed a clear distinction between the write speeds of the databases as the number of threads increases. Redis maintains a steady performance, ChromaDB shows minor fluctuations but remains relatively stable, MilvusDB's performance degrades gradually, and Weavite degrades significantly. Redis and ChromaDB are "on-memory" databases. This may explain their better scalability.

3.2 Write speeds

Figure 4 shows that write operations scale better for ChromaDB and Redis than for MilvusDB and Weavite. This is similar to read operations and the reasons for the observed behaviour should also be the same. We spin up a fixed number of threads. Each thread is assigned a fixed number of entries to write (100 in this case). And the threads are run and the start and end times are measured. We divide that number by 100 to get the times presented. Each of these writes were performed multiple times and averaged out to remove noise from our readings.

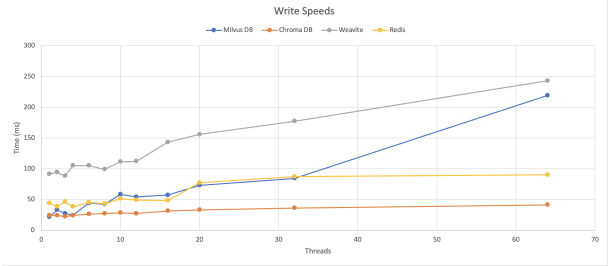

Figure 4: Write speeds

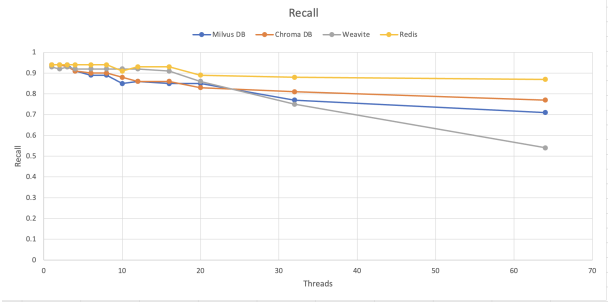

Figure 5: Recall vs Load

An interesting observation is that the write operations are noticeably faster than read operations. This is because data is "indexed" or structured in a queriable way at a later "indexing" stage. The write operation simply saves this data to memory without the need to go through an index.

3.3 Recall vs Load

As seen in Figure 5, Redis and ChromaDB seem to have higher recall when under heavy load (which means they are good at retrieving relevant items) than Weavite and MilvusDB. These observed behaviours could be due to better design and/or more efficient indexing. Also, the burstiness of the reads could affect recall. As the two databases have much faster read performance, they perhaps encounter less overall stress.

The big trade-off with ChromaDB is that it's purely an in-memory database. It offers no persistence. This severely limits it in its use cases. Redis does offer persistence but the operation of writing to disk costs time and is something to take into consideration while evaluating performance.

4 Threats to Validity

When the load on a database is very low, the recall is still less than 1. This means the data set isn't perfect. For example, given a code block x and its mutation x', there may exist another code block y that is closer in similarity to x than x' is.Databases under minimal load produced a recall of 0.99 with a perfect dataset. The same DB produced a recall of 0.94 with our dataset. This quantifies the error in our dataset. Results measured should have this fixed offset added to understand true database performance

5 Conclusion

In this project, we created a novel large high-dimensional data set for benchmarking vector databases. It is also the first code-specific data set for benchmarking vector databases. Determining the nearest neighbors of two entries in the dataset is expensive computationally. By generating semantically equivalent mutations of existing source code, we avoid this issue. Our benchmarks showed that Redis had the lowest fluctuations in read performance with ChromaDB coming in second. For write operations ChromaDB and Redis scale better for than MilvusDB and Weavite. We also found that Redis and ChromaDB provide higher recall when under heavy load.

Our source code and scraped source code/mutations can be found here: https://github.com/sueszli/vector-database-benchmark.

The raw data for the benchmark is stored separately here: https://uofwaterloo-my.sharepoint.com/personal/vnsubram_uwaterloo_ca/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Fvnsubram_uwaterloo_ca%2FDocuments%2FVDB-Code-benchmarking-dataset&ga=1

References

[1] Hugging Face. Mteb leaderboard. https://huggingface.co/spaces/mteb/leaderboard, 2023. Accessed: yyyy-mm-dd.

[2] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning, 2023.

[3] Jimmy Lin, Ronak Pradeep, Tommaso Teofili, and Jasper Xian. Vector search with openai embeddings: Lucene is all you need, 2023.

[4] Pinecone. Hierarchical Navigable Small Worlds (HNSW) | Pinecone.

[5] Zachary Proser. Retrieval Augmented Generation (RAG): The Solution to GenAI Hallucinations | Pinecone.