Distributed Vector Database with Concurrent Operations

Deepkumar Pareshkumar Raval
40231723
deep.raval.ca@gmail.com

Shivam Dipak Soni
40232364
shivam.soni.ca22@gmail.com

Jaymin Shantilal Suhagiya
40232368
jaymin.suhagiya.ca@gmail.com

Priyansh Jagdishbhai Bhuva
40269498
prjbhuva@gmail.com

Harshvardhansingh Rao
40268567
raoharsh2801@gmail.com

ABSTRACT

We aim to develop a distributed vector database system, deploying N instances of Vectordabase within their respective containers. These instances have the capability to store text data as embeddings and support similarity search queries. They expose gRPC interfaces for communication. A backend acts as a middleware, facilitating communication between the frontend/user and the database instances. This server not only caters to the frontend but also provides REST APIs for various operations. The system is designed for horizontal scaling of vector databases and incorporates fault tolerance. In the event of an instance failure, the impact on result quality is minimized. This system is particularly valuable for prompt augmentation, offering an efficient solution for current Large Language Models (LLMs).

1 INTRODUCTION

The proposed system consists mainly of four components: Dataset, Vector Database Instances, Middleware Backend, and Frontend. Figure 1 illustrates the high-level architecture of the system. Initially, each component is analyzed and explained. Subsequently, we will delve into performance analysis, exploring features such as scaling and fault tolerance.

2 SYSTEM

2.1 Dataset

The provided dataset is a small subset of the Project Gutenberg [2] dataset, which is a comprehensive collection of free and publicly accessible literary works. The subset comprises 3036 [3] manually cleaned books available in plain text format. The complete Project Gutenberg corpus includes over 70,000 books covering diverse genres and time periods. This extensive dataset, known for its public domain content, serves as a valuable resource for various projects involving natural language processing, text mining, and machine learning. The dataset size is 1.22 GB, and a script is employed to process and add the text to the Chroma DB database using a VDB instance's REST API endpoint (`/add_big_text`), with multithreading optimizing the ingestion process.

In the ingestion process, the book data is converted into small chunks of text, and the API endpoint itself specifies the limit for each chunk. This approach facilitates efficient data handling and enhances the overall ingestion performance.

This JSON example represents a POST request. It includes the book data, identified as <bookdata>, along with metadata specifying the source file name ("<file_name>") and a unique identifier for the file part ("<file_part>"). The "n_paragraph_sentences" parameter indicates a numeric value (7), representing the desired number of processed sentences or paragraphs.

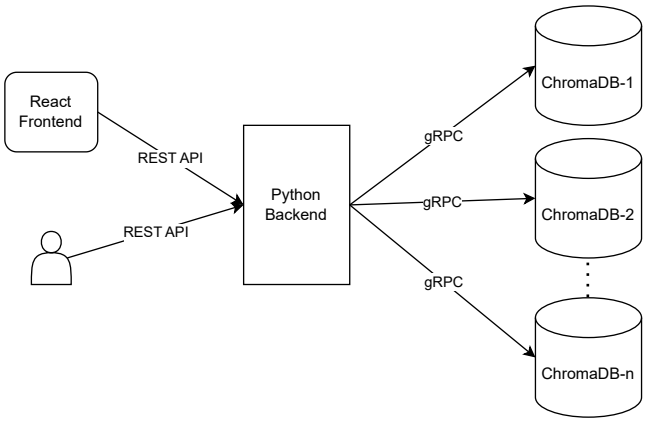

Figure 1: Architecture of the system.

```
{
    "document": "<bookdata>",
    "metadata": {
        "source": "<file_name>"
    },
    "id": "<file_part>",
    "n_paragraph_sentences": 7
}
```

Figure 2: Example POST request in JSON format.

2.2 Python Server

We employ a Python server that functions as a middleware backend, utilizing FASTAPI [4] to create REST endpoints. This server dynamically retrieves the list of available vector database instances and generates their corresponding gRPC [1] addresses. When a user invokes any endpoint, the backend internally queries the necessary instances using gRPC and subsequently delivers the results to the user. To enhance performance, multithreading is implemented for querying multiple databases concurrently. The server is containerized and additionally serves the Frontend along with the REST API. Backend have the following endpoints:

- `GET - /health`: Returns the health of all DB containers.

- `POST - /query`: Performs the similarity search across all online DB containers and returns the result.
- `POST - /add_text`: Adds text into DB containers respecting fault tolerance rules.
- `POST - /add_big_text`: Adds a long text (after converting into chunks) into DB containers respecting fault tolerance rules.

2.3 ChromaDB Instance

We utilize ChromaDB as our vector database and employ a persistent client to facilitate container restoration and data recovery in the event of a failure. Container has a gRPC interface for communication, offering the advantage of speed and efficiency. Additionally, since the database cotnainer operates internally without direct communication with the external world, security rules can be centralized at the initial point of contact, namely the server. This streamlined approach minimizes the footprint of our database instances, enhancing their speed. The database instance supports the following RPC methods:

- `rpc health (google.protobuf.Empty)`: Returns the database's health status, encompassing Used RAM, Total RAM, CPU usage (in %), and Average Response Time (in ms).
- `rpc addText (AddTextRequest)`: Adds text to the database as outlined in the `AddTextRequest` after converting it into embeddings.
- `rpc similaritySearch (SimilaritySearchRequest)`: Executes a similarity search in the database as specified in `SimilaritySearchRequest` and returns the results sorted from most relevant to least relevant.

2.4 React Frontend

We have designed a frontend to provide an intuitive user interface for system representation. The development utilized React and ChakraUI. The frontend communicates with the backend through a REST API. It is divided into two main sections: Health and Query. The Health section features a table displaying the health status of all database containers, while the Query section supports database querying. Figure 3 and Figure 4 depicts the frontend interface.

2.5 Deployment

All components, including the Frontend, Backend, and Vector Database, have been containerized for ease of deployment and management. Two Dockerfiles have been created: one for the Middleware Backend and another for the Vector Database. The Dockerfile for the backend also incorporates the deployment of the frontend. In both Dockerfiles, gRPC protos are compiled and then moved to their respective folders. For the backend, a production build of the frontend is generated and used for serving. The backend is executed with `uvicorn`, while the VectorDB container is simply run with a Python command for the gRPC server.

To streamline these processes, a shell script has been developed. This script takes the number of Vector Databases and a rebuild flag as input. The script automatically deploys containers using Docker Compose. Container images are rebuilt if the rebuild flag is provided.

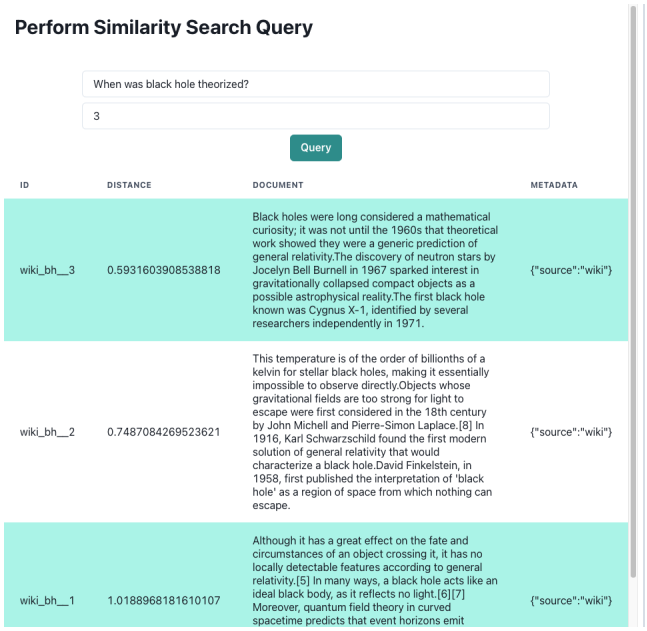

Figure 3: Screenshot of Frontend (Similarity Search).

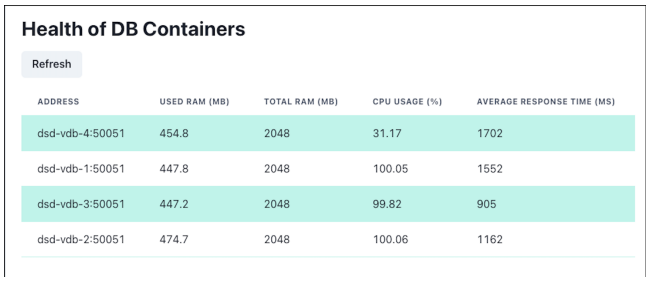

Figure 4: Screenshot of Frontend (DB Healths).

3 DEMO SCENARIO

3.1 Resources

For our experiments, we employed four Vector Database instances. Each instance was allocated 2GB of RAM and one CPU core from the host machine. It is worth noting that no specific limits were set for the Middleware Backend. The experiments were conducted using a MacBook Pro M3 with 16GB of RAM.

3.2 Fault Tolerance

For the current project, we have implemented a straightforward strategy to support fault tolerance. When a new piece of text needs to be added to the Vector Databases, it is ingested into any of 2 randomly selected instances. This strategy, while simple, proves to be highly effective for fault tolerance. In the future, this approach can be further optimized, and a more sophisticated strategy may be adopted. Our code has been designed to facilitate the integration of a new strategy with minimal effort.

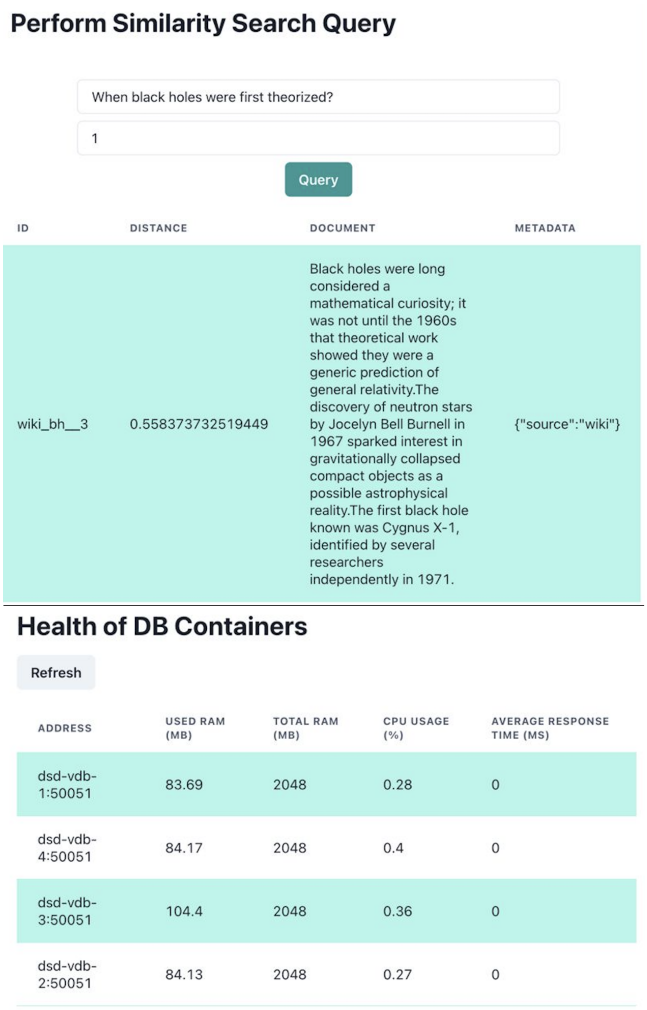

Figure 5: Screenshot of Query and DB Healths with all machines active.

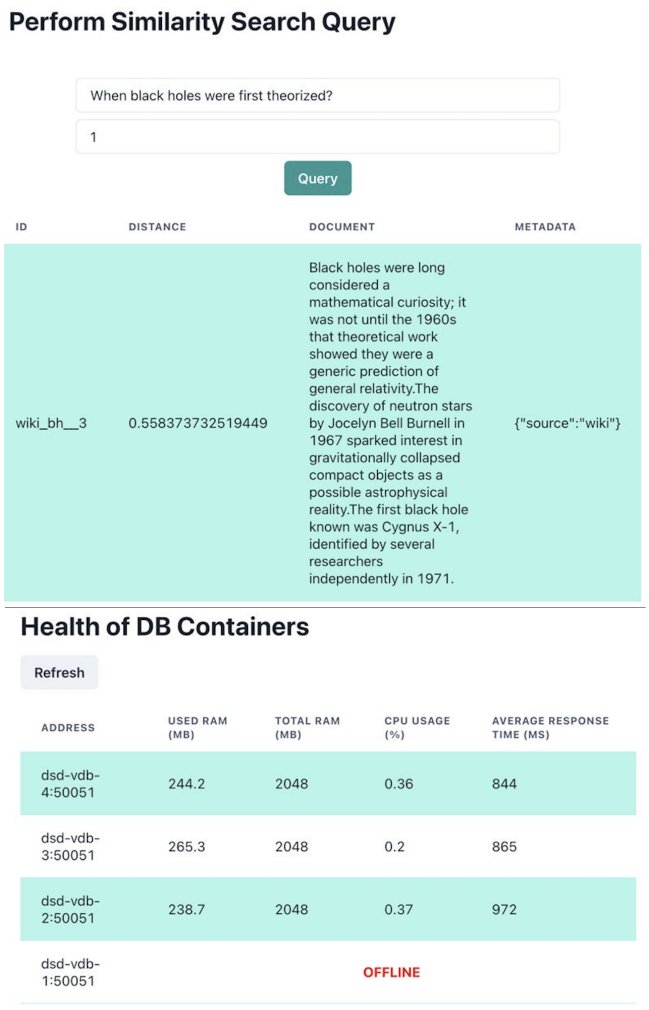

Figure 6: Screenshot of Query result which still remains the same (because of saving it to 2 machines) and DB Healths with machine-1 offline.

In the event that one instance of the database goes down, another instance retains the same embeddings, ensuring no impact on the results and maintaining result quality. Even if both instances experience an outage, some instances may still share similar text chunks (obtained from the same corpus), as two instances are randomly selected for ingestion. In such scenarios, there might be a slight degradation in quality, but relevant results will still be provided. The system will only fail if all instances of the database go offline.

For our experiment, we intentionally terminated one of the database instances and subsequently queried it again. As anticipated, this action did not impact the quality of the results. We then proceeded to terminate the second database (we picked the databse which was providing higest quality results) instance and repeated the query. While the results were affected this time, they still maintained an acceptable level of quality. Refer to Figures 5, 6 and 7 for illustrations of these scenarios.

3.3 Other Experiments

In the "Other Experiments" section, a comprehensive analysis of the health of DB containers was conducted, with a specific focus on response time metrics. The table below presents key performance indicators for each container at one specific instant of time:

ADDRESS	USED RAM (MB)	TOTAL RAM (MB)	CPU USAGE (%)	AVG RESPONSE TIME (MS)
dsd-vdb-3:50051	514.2	2048	101.84	883
dsd-vdb-2:50051	588.7	2048	40.43	1107
dsd-vdb-1:50051	546.3	2048	99.09	1268
dsd-vdb-4:50051	576.2	2048	98.79	1498

Table 1: Performance Metrics of DB Containers: RAM Usage, CPU Usage, and Average Response Time at some instant.

This analysis reveals several noteworthy insights:

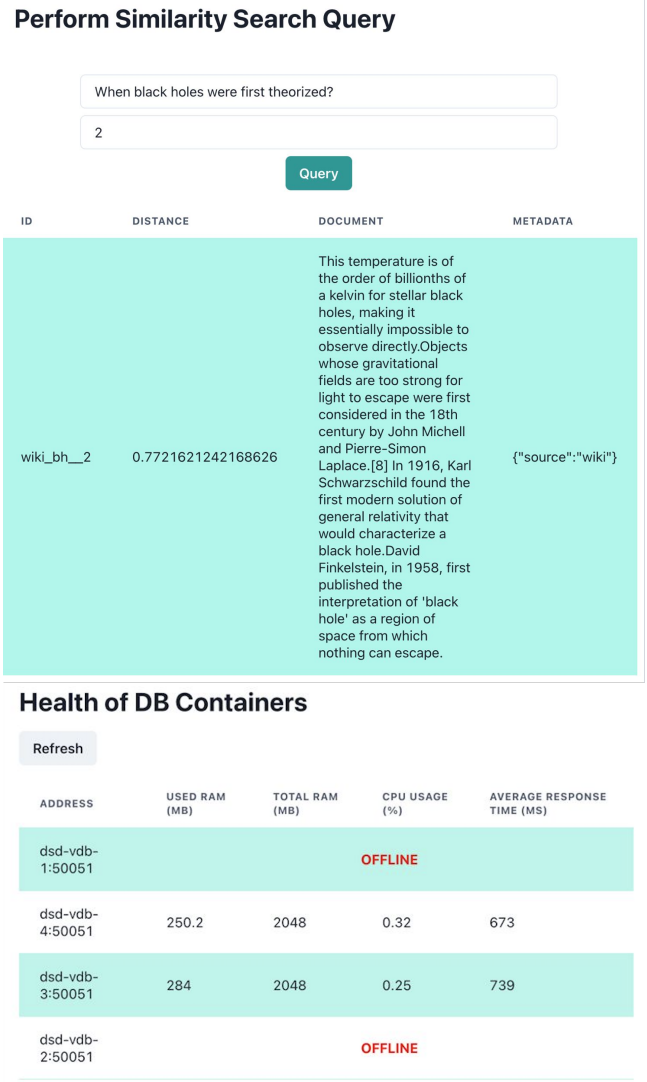

Figure 7: Screenshot of DB Healths with machine 1 and 2 offline. The answer is different this time since the 2 instances where data was stored are down but we still receive closest distance answer.

- **Average Response Time Variability**: Response times vary from 883 ms to 1498 ms, indicating potential workload distribution or container-specific processing disparities.
- **Performance Impact on User Experience**: Higher response times, as observed in "dsd-vdb-4," may affect user experience, necessitating further investigation to identify and address potential bottlenecks.
- **Correlation with CPU Usage**: Containers "dsd-vdb-3" and "dsd-vdb-1" show high CPU usage (101.84% and 99.09%, respectively) alongside relatively high response times, suggesting a potential relationship between CPU utilization and response time.
- **Optimization Opportunities**: Container "dsd-vdb-2" stands out with lower CPU usage (40.43%) and a response time of 1107 ms, indicating potential optimization opportunities for resource allocation.

In conclusion, a thorough examination of response time metrics provides valuable insights into DB container performance. Addressing observed variations and correlations is crucial for optimizing system responsiveness and enhancing overall user satisfaction.

4 CONCLUSION

Utilizing the proposed system offers several advantages:

- **Scalability:** The system allows for the creation of N number of Vector Databases, facilitating horizontal scaling.
- **Performance:** The use of gRPC and Multithreading for communication enhances speed and efficiency.
- **Fault Tolerance:** The nature of data storage ensures that the quality of results remains relatively high, even in the presence of offline DB instance(s). Acceptable quality can be maintained despite such instances.
- **Concurrency:** The independent and parallel processing of texts by parallel requests provides a high level of concurrency.

In conclusion, such a system proves valuable in real-world scenarios, particularly in conjunction with Large Language Models (LLMs) [5] for prompt augmentation and document-based text generation. These systems contribute to the facilitation of efficient, scalable, and fault-tolerant storage of embeddings.

REFERENCES

[1] gRPC. 2023. *gRPC: A high-performance, open-source universal RPC framework.* https://grpc.io
[2] Project Gutenberg. 1971-2021. *Project Gutenberg: A library of over 70,000 free eBooks.* https://www.gutenberg.org
[3] Shibamouli Lahiri. 2014. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics.* Association for Computational Linguistics, Gothenburg, Sweden, 96–105. http://www.aclweb.org/anthology/E14-3011
[4] Sebastian Ramirez. 2018. *FastAPI framework: High performance, easy to learn, fast to code, ready for production.* https://fastapi.tiangolo.com
[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).