Technology
Arts Sciences
TH Köln

Music Lyrics Semantic Search - A vector database approach

Max Hammer, 11153562
Dennis Gossler, 11140150

Module: Digital Science
Course: Modern Database Systems

Lecturer: Johann Schaible

June 28, 2023

Abstract

This report examines a web-based application designed to facilitate semantic song searching and provide users with recommendations of semantically similar songs based on their selections. The suitability of two databases, Weaviate (a vector-native database) and MariaDB (a relational database), is assessed in terms of their performance in various aspects crucial to this use case, including latency, scalability, query result quality, consistency, availability, and durability.

Contents

List of Figures

List of Tables

1 Introduction

The recent surge in the prominence of AI models has brought attention to the need for efficient data storage solutions tailored to these models. Many machine learning algorithms rely on numerical vectors as inputs to process their respective tasks. To accommodate the storage of large volumes of such inputs for models like Large Language Models, specialized databases have emerged with a focus on vector data. NoSQL databases like Weaviate and Pinecone have been developed as vector-native solutions to address this challenge.
This report aims to explore the application of semantic lyric search for songs, leveraging the embedding of lyrics and utilizing the distance between the embedded search term and the embedded songs in a vector database. Through this approach, users can search for songs based on semantic lyrics and search query similarity, and the stored embeddings enable recommendations of similar lyrics to be made.

Weaviate was selected as the NoSQL database for this project due to its open-source community and its support for vector search. The dataset used in this study was obtained from Kaggle and includes metadata that enables the playback of selected songs through a web application [14].
Furthermore, it is analyzed whether the selection of a NoSQL database, represented by Weaviate, is better suited for the use case at hand than a relational database. The relational database will be represented by a MariaDB instance. The requirements for the databases in order to fulfill the use case are outlined in the following section.

1.1 Use case

The use case for this project is an information retrieval application. The desired information is the similarity of songs based on a text search query or a chosen song. Since many search engine for songs already exists on streaming platforms like SoundCloud, Spotify, Apple Music, and many more, this use case is scoped to the semantic similarity of songs. Especially the semantic similarity of *lyrics*, *title*, *album name*, *genre* and *artist*. Because the application is supposed to retrieve information and not data, the following requirements can be stated.

First and foremost, the database should support a way to semantically search. Traditionally this is done by vector comparison. Therefore support for vector storage and vector search is optimal.

The application is very read-operations heavy and does not support any write operations within the current design. Consequently, read operations should be very fast, while write operations can perform slower to a greater extent.

Since we are not dealing with transactional data, ensuring data consistency becomes less crucial and is not a necessity. It is not of significant importance for the user if the data lacks consistency at all times.

Furthermore looking at ACID and BASE models for relational and NoSQL databases, you can say that for this use case, atomicity is not a necessity since

the application does not deal with transactional data. As mentioned before consistency is also not required. Each query should be isolated and not impacted by e.g. other search queries. Durability is a crucial attribute for disaster recovery and is universally advantageous for all databases.

While the data set we will use for a prototype of the application is rather small, the amount of data that could be stored inside the required database could be very large and linearly scale to the number of published songs worldwide. Consequently, scaling of the database should be as cost-efficient as possible. A large data set also can slow down the query time. NoSQL databases utilize sharding to break up the data across multiple nodes in a cluster, enabling more data to be stored at a cheap price. Therefore sharding is a good way to counteract the vast amount of theoretical data.
More important for this kind of application is the availability of data. Availability is preferred over consistency in this use case because it is tolerable to have eventual consistency and users may not retrieve the latest version of data instead of not retrieving data at all.

A core feature of search engines is low query latency which is also a requirement for the proposed use case. Particularly, a low parallel usage latency is important, since this application is turned towards a large possible user group.

The following requirement for a NoSQL database could be that it supports replication. By replication, the database can guarantee more availability and also can reduce concurrent query latency by splitting requests into several replications of the database. Therefore a NoSQL database should support replication.

1.2 Structure

The following section presents the objectives of this report in detail. You can split up the experiment into two sections, being a *benchmark* to compare query speed and quality of query results and a *prototype* to evaluate how the chosen database performs in a production environment. Firstly, a detailed description of the used dataset and the data-cleaning procedures that have been done is given. Additionally, the used data schemas for the evaluated databases and the corresponding decision process are presented. Subsequently, the report goes into detail how the benchmarking was set up. Based on that benchmarking a database is chosen and incorporated into a prototype.
Furthermore, the report discusses the rationale behind selecting Weaviate as the database for the prototype, elucidating its functioning and outlining the advantages it offers over relational databases and other NoSQL databases in terms of the use cases that have been specified in section 1.1.

2 Experiment

To be able to decide what database is best suited for the presented use case an experiment is conducted. The experiment includes three steps, **1)** to prepare a music data set to be able to benchmark and prototype possible solutions, **2)** a

benchmark setup to be able to compare relational databases with a vector-native database and finally **3)** a prototype to mimic a production environment.

2.1 Data

In this subsection, an overview of the used data set and the corresponding cleaning procedure to feed it into Weaviate and MariaDB is given. Furthermore, the selected database schemas are shown and the reasoning behind the chosen schemas is given. We selected a 13MB data set from Kaggle [14] that has 18452 entries with 25 columns, which are described in table 1. The data set is large enough to showcase differences in query speed and results. Additionally, it is possible to uncover possible complexity issues between the tested databases and still is small enough for us to flexibly experiment with it. Still, the data set is so small that in terms of database scalability, maintainability, and cost the tested databases could not be assessed. Therefore assessment of these factors is done theoretically.

2.1.1 Data Cleaning

Before importing the data set to the vector database the data was analyzed to sort out outliers and faulty data. 3339 song titles are duplicates, which is 18.07% of the total data set. All of these duplicates have no duplicate *track_id*, which shows they are individual songs. They are most likely remastered or live-recorded versions of songs. Therefore we keep all of these records.
The data set contains many properties that are extracted from the Spotify API that describe a song in numerical values like *loudness*, *speechiness*, *danceability*, *energy*, *mode*, *speechiness*, *acousticness*, *instrumentalness*, *liveness*, *valence* and *tempo*. In a future iteration to enhance the project these properties could be included.

2.1.2 Data Schema

In managing our SQL database, we adopted two distinct strategies. Initially, we implemented a single-table model named 'track_single_table'. This involved directly importing each data row from the .csv file into our database, an approach that conforms only to the first normal form.
We also utilized a more sophisticated model designed around the third normal form. This strategy required the division of the database into four separate tables, specifically *'track'*, *'album'*, *'artist'*, and *'playlist'*, as you can see in Figure 1. This design was adopted to uphold the recommended standards of data normalization.

Column	**Data Type**	**Description**
track_id	character	Spotify unique ID
track_name	character	Song Name
track_artist	character	Song Artist
lyrics	character	Lyrics for the song
track_popularity	double	Song Popularity (0-100) where higher is better
album_id	character	Spotify album unique ID
album_name	character	Album name
album_release_date	character	Date when album released
playlist_id	character	playlist ID
playlist_name	character	Name of playlist
playlist_genre	character	Playlist genre
playlist_subgenre	character	Playlist subgenre
language	character	Language of the lyrics
duration_ms	double	Duration of song in milliseconds
danceability	double	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
energy	double	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity.
key	double	The estimated overall key of the track. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1.
loudness	double	The overall loudness of a track in decibels (dB).
mode	double	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
speechiness	double	Speechiness detects the presence of spoken words in a track.
acousticness	double	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
instrumentalness	double	Predicts whether a track contains no vocals.
liveness	double	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
valence	double	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
tempo	double	The overall estimated tempo of a track in beats per minute (BPM).

Table 1: "Track" Data class schema - Highlighted Columns are embedded as a vector in the Weaviate database

Figure 1: Entity relationship diagram(multi and single table)

Weaviate provides a schema construction which is a formal description of data classes. While the database also offers options to store complex data types like images, the schema can define properties that should be used for embedding and further index configurations. Based on the presented use cases we decided to transcribe the data set to a singular class since the use case demands a fast information retrieval and not a structured database. All entries are indexed with a vector index, which is called Hierarchical Navigable Small World (HNSW). HNSW is a multi-layered graph that enables very fast query times at the downside of being relatively costly when adding new data [11]. In addition to the vector index an inverted index is added to all *character* type properties, e.g. *track_name* or *lyrics*, by default. The inverted index tokenizes the *character* properties by keeping all alpha-numeric characters, lowercasing them, and splitting by white spaces as seen in example 1 [10].
This adds options for syntactic querying to the vector search capabilities. Despite the fast query speed, this tokenization splits up text and therefore limits the capabilities to filter for exact snippets of a sentence since the properties are split up by whitespaces. This is a limitation that can be tolerated in this use case.

> **Example 1** (Inverted index tokenization by Weaviate)**.** "*What is love*?" → "*what*","*is*","*love*"

2.2 Prototype

As mentioned above this prototype introduces a semantic search for songs. The application especially should offer *Semantic Search for Songs* and *Semantic Recommendations for selected songs*. Furthermore, to make the application more

engaging the user should also be able to play songs from the application itself. The selected use cases profit very heavily from a vector-native database. For example, Weaviate has integrated vector search instead of the necessity of implementation on the application layer itself. The modular architecture of Weaviate makes it simple to plug in necessary embedding models that embed the data points on import to vectors and embed search queries.
The source code for the prototype is published in a GitHub repository [13]. Furthermore, in the appendix B screens of the implemented prototype are shown.

Figure 2: T-SNE demonstration of actual vector search in prototype database

In figure 2 you can see a T-SNE demonstration of how you can interpret how vector search works in a 2D space. In this prototype, the default distance metric, offered by Weaviate, was chosen which is *Cosine (angular) distance* [6]. It is defined as

$$1 - cosine\_sim(A, B)$$

with the distance specified as $0 \leq d \leq 2$ and

$$cosine\_sim(A, B) = \frac{A \cdot B}{\|A\|_2 \, \|B\|_2}$$

The application is configured and run as a multi-container application as seen in figure 3. All services are configured in a unified *docker-compose* YAML file. The application consists of the following services:

Web app: A web application user interface to enable users to query and play tracks. It resembles the core of the application since it communicates with all other services via REST calls. It is written with *Vite* and *React* as Frontend Stack.

Spotify Auth Server: A small *Node.js* server run to authorize communication with the Spotify API to enable access to song and album data.

Vectorizer: Weaviate needs an external service to embed data on import and when querying the database to have comparable vectors. The

architecture of Weaviate offers easy integration of so-called "modules". For this application, the simple *text2vec-contextionary* embedding model is used because it is open-source and small. While it is not fine-tuned for our specific use case, it shows sufficient results to show the concept of the application. Furthermore, the small size of the model enables us to run it locally with little computing resources. Better results could be expected when using a larger model with greater computing resource allocation.

Weaviate: The Weaviate container has two purposes. Besides logic to connect modules, business logic, and REST and GraphQL APIs, its core consists of two database components.

Vector Index: A vector index that stores all vectors with references to the corresponding documents in the document database. It is targeted with vector search queries to search for similar vectors based on *cosine (angular) distance*

Document Database: A document database that stores all data in classes with references to the vector index. In this application, it is used to retrieve song data like the title, artist, and lyrics of a certain song to be displayed in the user interface.

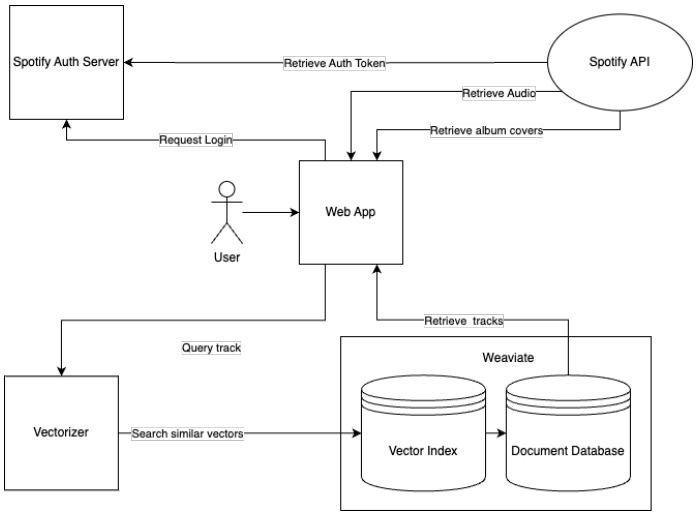

Figure 3: Architecture of prototype

2.3 Benchmark setup

To compare Weaviate and MariaDB, a speed benchmark is performed. Since Weaviate consists of a vector database and a document database, we decided that it would be best to compare both to MariaDB. It is important to acknowledge that comparing Weaviate and MariaDB in this particular context presents a significant challenge due to their inherent differences, as the results of the vector database are more like the results of a search engine and not like the output of the above two databases. While a search engine ranks the results and traditional databases filter for their results. Nevertheless, it gives a better

picture of the speed, strengths, and weaknesses of a relational database and Weaviate.
In order to perform the performance evaluation, we defined five different queries, each with a different goal in mind.

> **Benchmark 1**: This search criterion focuses on the identification of a specified term, named "love", within all text-based columns[1]. The sequence of other lexical units flanking the target term does not exert any influence on the search results.
>
> **Benchmark 2**: In this scenario, the search parameter extends to incorporate multiple lexical units, specifically "Love", "Cake", and "Fame". This inquiry permeates all textual columns.
>
> **Benchmark 3**: This benchmark delineates the search scope to locate a particular phrase, i.e., "What is love," dispersed across all textual columns. It is important to note that the results for this benchmark will vary a lot in terms of the results and query speeds. The integrated document database has an inverted index attached to all text properties, as explained in section 2.1.2. This inverted index queries the database in a way that instead of looking for the depicted phrase it looks for the word within the phrase, resulting in far more results. Furthermore, the inverted index enables a way faster query time than the query of a relational database, which loses its index when querying with the *"LIKE %"* operand. The integrated inverted index suits the use case a lot better and was therefore kept for this benchmark.
>
> **Benchmark 4**: This query focuses on searching for the word "love" in a specific column. The placement of other words before or after "love" doesn't bear any significant weightage. This search is specifically limited to the 'track_name' column.
>
> **Benchmark 5**: This search benchmark utilizes a vector to determine similar songs. The vector corresponds to the song "Queen - Get Down, Make Love - Remastered 2011".

Our systems are deployed through Docker containers, which encapsulate the MariaDB Server, a Vectoriser, and the Weaviate Databases, both Vector and Document. The Benchmarks were run on a Macbook M2 Pro.
Benchmarks are evaluated via a dedicated Python script. This script initiates each benchmark a total of 100 times. We then compute the average runtime across these repetitions to establish the definitive average benchmark time.
The benchmark Python script uses two libraries, named "mysql.connector" and "weaviate-client", to form the connection between our Python script and the databases.

The source code for the benchmark setup is published in a GitHub repository [13].

[1] In this context, the comprehensive suite of text columns encompasses "track_name", "artist_name", "album_name", "lyrics", "genre", and "subgenre"

3 Discussion

In the following section the results from the conducted benchmark experiment and their relation to the stated use case requirements in section 1.1 are discussed. Furthermore, insights from building a prototype with Weaviate are taken into account for the evaluation of the best-fitting database for the use case at hand.

3.1 Benchmark results

We run the benchmarks as mentioned in Section 2.3, using three distinct data retrieval limits: 10, 100, and 1000. The following diagrams in figure 4 illustrate that optimal database performance is contingent upon the specific use case in question.
Furthermore, it is critical to highlight the potential enhancement of document database/vector dataset performance through the deployment of CUDA. "Note that transformer models are Neural Networks built to run on GPUs. Running Weaviate with the text2vec-transformers module and without GPU is possible, but it will be slower." [7]. The query results are not discussed in this section since the use case heavily leans on support for information retrieval while relational databases like the tested MariaDB instance does only support data retrieval. Information retrieval is only possible through the vector index and inverted index in the tested databases. To be able to compare NoSQL variants to a relational database, the benchmarks are mostly analyzed in regard to their speed even though they support data retrieval instead of information retrieval.
Moreover, it is important to note that BM5 cannot be compared among the different database systems since they are not capable of comparing data as vectors. Therefore, the benchmark result is listed as a table.

ID	Database	Avg Query Time (s)	Limit
BM5	Vector	0.007215387	10
BM5	Vector	0.011124487	100
BM5	Vector	0.040241876	1000

Table 2: BM5 benchmark results for the vector database

Upon aggregating the outcomes and computing the average, a contrasting perspective emerges. Evidently, the vector database emerges as the most efficient alternative, succeeded by the document and single table SQL databases respectively. Significantly, operations adhering to the third normal form, which necessitates inner joins across diverse tables, were observed to operate four times slower compared to those constrained to a single table. It is worth mentioning that utilizing a materialized view stored within an in-memory table could further optimize our execution times [3] [2].

That vector database queries are the fastest, can be explained by the utilization of the HNSW graph as vector index, which is explained in section 2.1.2. The query will maintain to be relatively fast because of the compression of the data to an n-dimensional vector no matter the size of the database. The HNSW graph is an *approximate nearest neighbor* algorithm that offers fast query time with a trade-off for "approximation" instead of accurate results [9]. In this use

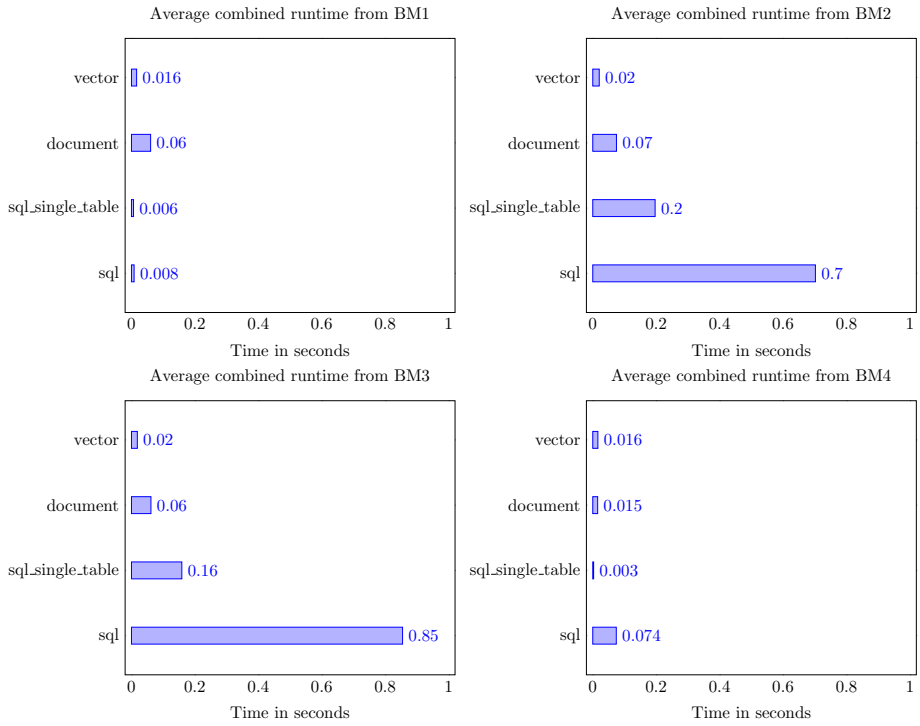

Figure 4: Benchmark results BM1-BM4

case, an approximation is good enough and expected.

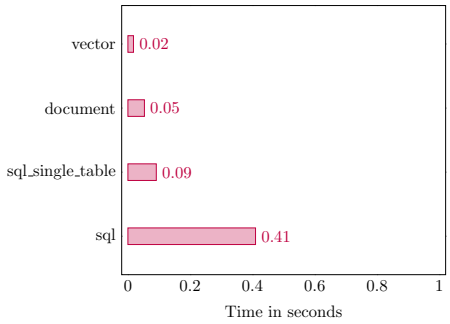

Figure 5: Average latency of BM1-4 results for every database

3.2 Prototype analysis

This section delves into a comparative analysis between traditional relational databases and the chosen NoSQL database - Weaviate. Our objective is to elucidate the distinctive characteristics of each database type, highlighting their strengths and potential limitations with a focus on our specific use case.

3.2.1 Scaling

Weaviate is designed to be horizontally scalable, meaning it can be run on multiple nodes in a cluster. This allows Weaviate to handle larger datasets, increase query throughput, and maintain high availability by distributing the load across multiple nodes. This horizontal scaling is achieved through two methods:

- **Sharding**: Dividing the dataset into smaller, manageable parts (shards) and distributing them across different nodes. This increases the maximum dataset size that can be handled and speeds up imports. However, sharding doesn't improve query throughput. Sharding is beneficial when the dataset is too large for a single node. Weaviate is using hash-based sharding, which has proven to be highly suitable for accommodating our specific use case. This choice is particularly advantageous due to the substantial volume of data we are responsible for managing. The sharding algorithm that Weaviate employed is a 64-bit Murmur-3 hash. [8]
- **Replication**: Replication involves duplicating data across multiple nodes to enhance query throughput and ensure high availability, which is both use-case requirements. Unlike sharding, replication does not directly accelerate data import speed. However, it becomes particularly useful when you encounter a high volume of queries that surpass the capacity of a single node or when maintaining like in our case an uninterrupted service, even in the event of a node failure [8]. In our specific use case, eventual consistency is sufficient since we are not dealing with transactional data.

  Leaderless Design: In Weaviate, replication is leaderless, meaning there is not a master-slave system or primary node that replicates to follower nodes. Instead, all nodes can accept writes and reads from the client, which improves availability by eliminating any single point of failure. However, a leaderless database tends to be less consistent as data on different nodes may temporarily be out of date. Consistency in Weaviate is tunable, but this occurs at the expense of availability [4]. There are three different options for consistency to select from ONE, QUORUM, ALL [5]. We picked QUORUM consensus option because the application does not require consistency across all nodes and requires as low latency as possible. Therefore QUORUM or ONE are the possible options for the use case, while QUORUM is the recommended consenus option from Weaviate.

  Replication Factor: Replication in Weaviate is enabled and controlled per class, and the replication factor (RF or n) determines how many copies of data are stored in the distributed setup. The replication factor can't be higher than the number of nodes. Any node in the cluster can act as a coordinating node to lead queries to the correct target node(s). A replication factor of 3 is commonly used as it provides a right balance between performance and fault tolerance [4].

In comparison to a traditional relational database like MySQL, MariaDB, or Oracle, horizontal scaling can be more challenging to implement in relational

databases because they are often designed with a monolithic architecture that assumes a single-node setup.
In contrast, Weaviate, as a vector database designed for horizontal scaling, automates many of these processes and is optimized for multi-node operation. Sharding and replication happen automatically, requiring only the specification of the desired shard count or replication factor.

3.2.2 CAP / PACELC

Relational databases like MySQL, MariaDB, or Oracle prioritize consistency over latency and availability. They are typically CP systems (Consistent and Partition-tolerant) according to the CAP theorem. This means that they ensure that data is consistent across all nodes, even in the event of a network partition, but at the cost of availability. They may not be able to serve all requests during a network partition. In the PACELC [12] context, they choose consistency over latency in normal operation, meaning they ensure that all data read will see the most recent write, even if it means increasing the latency.

On the other hand, Weaviate as a NoSQL database, is designed to provide high availability and low latency, often at the cost of consistency. It is typically an AP system (Available and Partition-tolerant) according to the CAP theorem. This means that it will continue to serve requests, even in the event of a network partition, but the data may not be consistent across all nodes. In the PACELC context, Weaviate would choose latency over consistency in normal operation, meaning it prioritizes serving data quickly, even if it means that not all reads will see the most recent write.

3.2.3 Query Language

Weaviate makes use of the GraphQL [1] query language as their interface. GraphQL is a very common NoSQL query language that can be used in other databases as well. Comparing it to the number of developers that most likely are familiar with the SQL query language, the number of proficient GraphQL developers will be smaller.

3.2.4 Vectorizer dependency

One problem we encounter with Weaviate is our dependency on a single selected vectorizer. If we were to switch to a new vectorizer, we'd have to reprocess all the data through the new system. This undertaking would require significant computational resources or extensive API usage. This process would be both time-consuming and expensive.

4 Conclusion

In conclusion, Weaviate emerges as the clear winner for the given use case of a semantic search engine for song lyrics. Its vector search function, which acts like a ranking rather than filtering, proves to be better suited for such an application. Additionally, Weaviate's low latency, thanks to its vector index, provides an efficient user experience. The horizontal scaling capabilities of

Weaviate, specifically through cost-efficient sharding, make it well-equipped to handle large volumes of data. Moreover, replication enables low parallel latency and high availability. However, it is important to note that Weaviate's performance is dependent on the vectorizer module, which may incur costs through API usage or computing resources.
However, it is important to acknowledge some limitations. The quality of results heavily relies on the vectorizer model used. This means that the effectiveness of the search engine is directly impacted by the choice of vectorizer. On the other hand, traditional SQL databases seem not to be well-suited for the depicted semantic search use case, highlighting the advantages of using Weaviate.
Looking ahead, there are a few aspects to consider. Firstly, it may be worth exploring Oracle as an alternative to Maria DB, as Oracle offers features such as materialized views that could turn out to be helpful in this use case. Additionally, experimenting with different vectorizer models can lead to improved embedding quality, but it may come at the cost of increased expenses or latency. It would be beneficial to evaluate specialized embedding models designed specifically for the use case of song similarity, as they may provide better results.

References

[1] *GraphQL*. https://graphql.org/, 2023. – Accessed: 2023-06-26

[2] *Introduction to Materialized Views*. https://docs.oracle.com/en/database/oracle/oracle-database/19/dwhsg/basic-materialized-views.html#GUID-A7AE8E5D-68A5-4519-81EB-252EAAF0ADFF, 2023. – Accessed: 2023-06-26

[3] *Populating Objects in the In-Memory Column Store*. https://docs.oracle.com/en/database/oracle/oracle-database/21/inmem/populating-objects-in-memory.html#GUID-C5F856BF-70E3-41C6-A1BA-1E94D7D230B8, 2023. – Accessed: 2023-06-26

[4] *Weaviate Documentation - Cluster Architecture*. https://weaviate.io/developers/weaviate/concepts/replication-architecture/cluster-architecture#leaderless-design, 2023. – Accessed: 2023-06-27

[5] *Weaviate Documentation - Consistency*. https://weaviate.io/developers/weaviate/concepts/replication-architecture/consistency, 2023. – Accessed: 2023-06-27

[6] *Weaviate Documentation - Distance Metrics*. https://weaviate.io/developers/weaviate/config-refs/distances, 2023. – Accessed: 2023-06-25

[7] *Weaviate Documentation - Docker-Compose Installation*. https://weaviate.io/developers/weaviate/installation/docker-compose, 2023. – Accessed: 2023-06-25

[8] *Weaviate Documentation - Horizontal Scaling*. https://weaviate.io/developers/weaviate/concepts/cluster#sharding-vs-replication, 2023. – Accessed: 2023-06-27

[9] *Weaviate Documentation - Indexing*. https://weaviate.io/developers/weaviate/concepts/indexing, 2023. – Accessed: 2023-06-27

[10] *Weaviate Documentation - Property Tokenization*. https://weaviate.io/developers/weaviate/config-refs/schema#property-tokenization, 2023. – Accessed: 2023-06-27

[11] *Weaviate Documentation - Vector Indexing*. https://weaviate.io/developers/weaviate/concepts/vector-index, 2023. – Accessed: 2023-06-27

[12] Abadi, Daniel: Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. In: *Computer* 45 (2012), Nr. 2, pages 37–42. http://dx.doi.org/10.1109/MC.2012.33. – DOI 10.1109/MC.2012.33

[13] Hammer, Max; Gossler, Dennis: *MDS_Spotify-Semantic-Search*. `https://github.com/MaxHam/MDS_Spotify-Semantic-Search`, 2023. – Accessed: 2023-06-27

[14] Nakhaee, Muhammad: *Audio features and lyrics of Spotify songs*. `https://www.kaggle.com/datasets/imuhammad/audio-features-and-lyrics-of-spotify-songs`, 2023. – Accessed: 2023-06-25

A Appendix raw benchmark results

ID	BM-DI	Database	Avg Query Time (ms)	Count Result	Limit
0	document	BM1	0.0528	10	10
1	sql_single_table	BM1	0.0005	10	10
2	sql	BM1	0.0008	10	10
3	vector	BM1	0.0040	10	10
4	document	BM1	0.0514	100	100
5	sql_single_table	BM1	0.0020	100	100
6	sql	BM1	0.0027	100	100
7	vector	BM1	0.0087	100	100
8	document	BM1	0.0770	1000	1000
9	sql_single_table	BM1	0.0168	1000	1000
10	sql	BM1	0.0214	1000	1000
11	vector	BM1	0.0362	1000	1000
12	document	BM2	0.0747	8	10
13	sql_single_table	BM2	0.1880	8	10
14	sql	BM2	0.2522	8	10
15	vector	BM2	0.0064	10	10
16	document	BM2	0.0726	8	100
17	sql_single_table	BM2	0.2105	8	100
18	sql	BM2	0.9517	8	100
19	vector	BM2	0.0113	100	100
20	document	BM2	0.0769	8	1000
21	sql_single_table	BM2	0.1912	8	1000
22	sql	BM2	0.9017	8	1000
23	vector	BM2	0.0435	1000	1000
24	document	BM3	0.0485	10	10
25	sql_single_table	BM3	0.1411	10	10
26	sql	BM3	0.8618	10	10
27	vector	BM3	0.0060	10	10
28	document	BM3	0.0524	100	100
29	sql_single_table	BM3	0.1662	12	100
30	sql	BM3	0.8481	12	100
31	vector	BM3	0.0106	100	100
32	document	BM3	0.0833	1000	1000
33	sql_single_table	BM3	0.1690	12	1000
34	sql	BM3	0.8538	12	1000
35	vector	BM3	0.0389	1000	1000
36	document	BM4	0.0073	10	10
37	sql_single_table	BM4	0.0005	10	10
38	sql	BM4	0.0007	10	10
39	vector	BM4	0.0036	10	10
40	document	BM4	0.0097	100	100
41	sql_single_table	BM4	0.0013	100	100
42	sql	BM4	0.0032	100	100
43	vector	BM4	0.0083	100	100
44	document	BM4	0.0293	807	1000
45	sql_single_table	BM4	0.0065	807	1000
46	sql	BM4	0.2186	807	1000
47	vector	BM4	0.0365	1000	1000
48	vector	BM5	0.0072	10	10
49	vector	BM5	0.0111	100	100
50	vector	BM5	0.0402	1000	1000

Table 3: Benchmark results [Macbook Pro M2 Pro]

B Prototype Screens

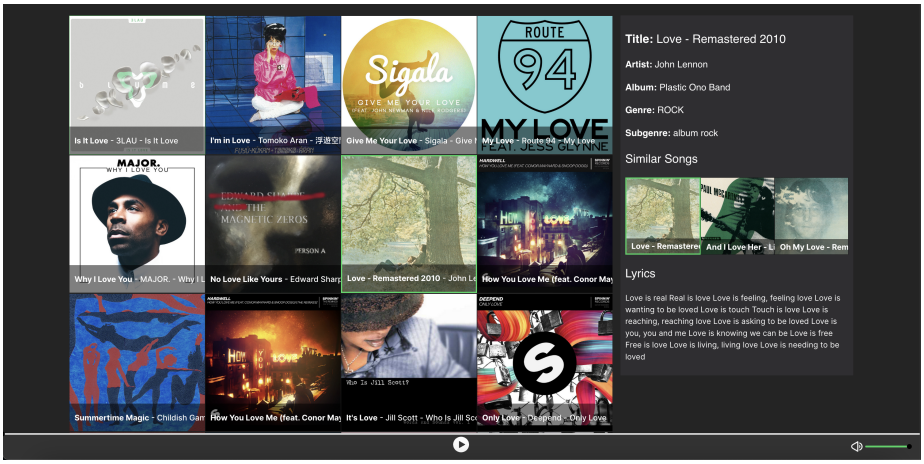

Figure 6: Prototype Screen 1 - Grid Overview of Search Results and Infobox for selected song

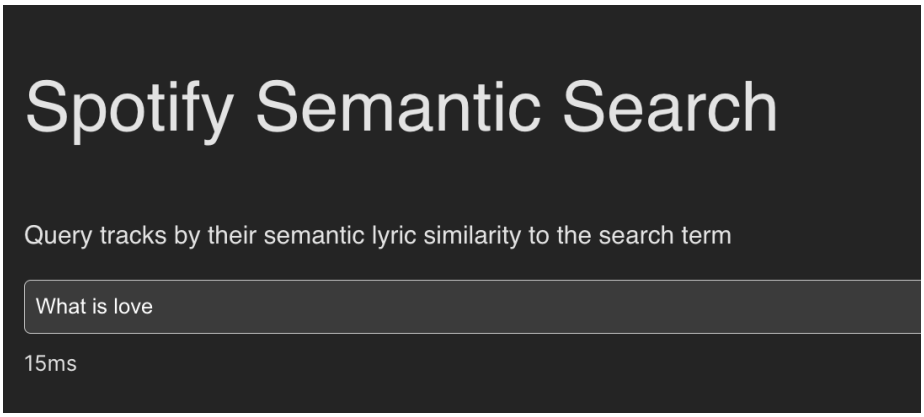

Figure 7: Prototype Screen 2 - Search bar interface

C Appendix Chat GPT usage

ChatGPT is used in the following chapters to enhance spelling and grammar:

Abstract

1. Introduction

2.3. Benchmark Setup

4. Conclusion