Bachelor's Thesis Computing Science

Radboud University Nijmegen

Implementing in-Database Similarity Search

Author:
Jaap Aarts
s1056714

First supervisor/assessor:
Prof. Dr. Ir. Arjen P. de Vries

Second assessor:
Prof. Dr. Hannes Mühleisen

August 16, 2024

Contents

Chapter 1

Introduction

Search is one of the most commonly used feature of computers these days. You may search your computer for a file, the internet for a web page, but that is not all. Search is in more places than you might think; YouTube searches for videos that match your viewing style, and Netflix recommends shows based on what you have watched before. All of these have one thing in common, they take in some query, like, "find file with name X", "find movie that is similar to Y", and result in a list of ranked results. With the best match first, second after that, and so on, assigning either a rank or score to every result, cutting off at given rank or score threshold. We will focus on systems that implement search using K-Nearest-Neighbor (KNN) [FH89] as their base.

KNN search algorithms can be expressed as an SQL query as such as Listing 1.1. More details on how this works in Section 1.2. One place such a query might be used, is for example in Retrieval-Augmented Generation (RAG) systems [Lew+20], which allows you to execute Question and Answer queries. An illustration of this model is provided in Figure 1.1. This is a popular model, and has proven to be very successful. We will not discuss this system as a whole in detail right now, additional detail will be discussed when appropriate.

Listing 1.1: SQL query expressing a KNN search. *id* is the id column, e is the column containing the points, q is the query point, *data* is the table containing the data, K is the number of desired results.

```
SELECT id, dist(q, e) as d FROM data ORDER BY d LIMIT K
```

An SQL query as in Listing 1.1 is good to get an idea about what is going on, however, using a naive implementation, it is unusably slow for large datasets. This method would require computing the distance for every possible result, a linear cost in the data size. Hence there are many implementations for efficient KNN-search. There are two that are most popular. The first is a well known data structure called the Inverted File index (IVF) [Knu73]. This can be used to efficiently query the nearest points, especially as data grows. More recently, the Hierarchical Navigable Small World (HNSW) [MY18] was discovered, providing accurate results and faster query times. Often, these are implemented in a vector database, dedicated to the purpose of executing KNN searches

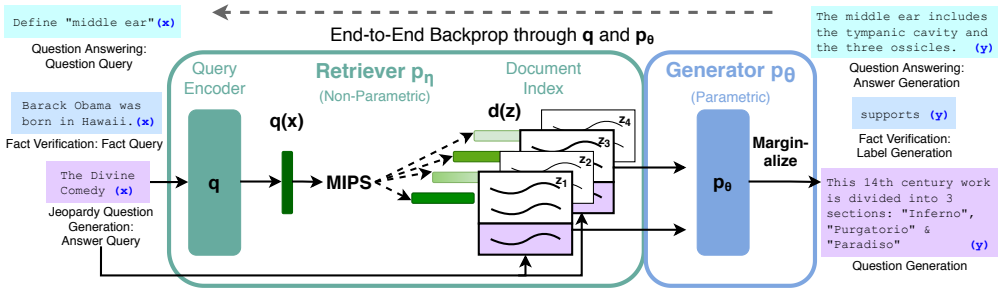


Figure 1.1: Overview of the RAG model. Executing a query x consists of encoding this query into a feature-vector, illustrated as $q(x)$, finding the nearest documents, here illustrated as **MIPS**. Then the documents go the Generator, which generates an answer from the selected documents. **MIPS** stands for Maximum Inner Product Search which is a kind of KNN.

on large datasets.

It can be useful to be able to apply a filter on such a query, such that it only returns results that satisfy other criteria as well. We will go more into an example explaining why in Section 1.1. One major problem with current implementations of KNN, such as IVF and HNSW however, is that they do not handle filtering well. If you wanted to execute a given query on a subset of the data this could lead to re-execution of queries with different cut-offs. This thesis explores the feasibility of combining a relational with a vector database to improve the performance of filtered KNN search. We will go more into why this is the case later in this chapter.

This thesis starts with a short running example. In section 2 we describe how search works in more detail, with a description of FAISS (a vector database). After that it is explained how these systems are currently setup and alternative setups, together with a comparison.

1.1 Running Example

For the purpose of the thesis, a running example was constructed, inspired by cases from existing literature. In this section we will introduce this example, and explain why it is interesting.

When doing research you often have to run analytical queries of various kinds, a natural part of the research process. If you want to differentiate 2 groups, you might look at the properties of one, and of the other. You can do this by querying, asking questions about these groups to find out their properties. In social science one would ask groups directly. If you want to know which country likes the color blue the most, you can survey a representative sample of the population. However, this is inconvenient; making a survey and getting people to answer is not an easy task. It involves making a good questionnaire, recruiting participants, and executing the survey. These steps are costly and take a lot of time. To make things easier, lets assume everybody already

posted their favorite color online on their personal webpage. We can use this to create a system which takes in all these webpages, their countries of origin, and allows you to run queries like "which country likes blue the most", and then spit out an answer. Although we will not discuss such systems in detail, a popular model is the RAG model [Lew+20], which would allow you to execute Question and Answer queries. An illustration of this model is provided in Figure 1.1.

We can also use this system for more complex questions. If we want to ask certain questions about scientific literature, we can make a large database of scientific articles, and ask questions about these. You could ask "is the earth flat", and you would likely find that the system gives back "no" as an answer. However, this might give an incomplete answer; the modern consensus is that the earth is not flat, however articles published between "600 and 1000 AD" could have a very different consensus. If you want to know what these people thought, you could add temporal a filter to your query. This indicates that you only want to include documents in this time range for your search. For the rest of this thesis this is the example that will be used, a simple question with a filter attached.

In the next section it will be explained how search is often implemented, and how such a filter could be applied at a lower level.

1.2 How search works

Although you may think that every search would be different, on modern systems most searches work relatively similar. This allows for some interesting search engines, like searching for images using text. The reason this is possible is because of one very opaque abstraction. Every document you want to search for is transformed into a format that is shared with the queries [Lin21]. In practice this representation is a long n-dimensional vector, just a list of numbers. Often the individual dimensions are thought of as features, or semantic aspects, and the vector is referred to as a feature-vector. This representation originates in a very simplistic model where each dimension represents some some aspect of the document. One dimension could represent how much an image looks like a dog[1], and the other how much it looks like a couch. Using this abstraction you can check if images are similar by checking if their vector representations are similar; if they both look like a dog and a couch, a conclusion could be that the images also look similar in general. At least more similar than two images, out of which one looks like a dog and one like a couch. With the advent of deep learning, these vectors have gotten to be much more abstract. One dimension no longer refers to just one semantic aspect, and one semantic aspect can be spread across many different dimensions.

In the following subsections we will discuss on a superficial level how these functions can be implemented, and why one can be chosen over another.

[1] How likely some model thinks there is a dog in the image.

1.2.1 Creating feature vectors

Creating good representations as feature-vectors, also called 'embeddings', is a complex task. These programs are called encoders, in our case specifically bi-encoders [Hum+20]. In short, a bi-encoder is a pair of encoders, one for documents and one for queries. Sometimes these are actually identical. There are multiple ways to construct bi-encoders, some of which we will cover in this subsection.

Direct method The first is to use the direct method, which means using a cleverly constructed algorithm to make a vector in such a way that similar documents get similar vectors. This can result in very fast encoding, but also is very difficult to write in most cases. If you want to compare images based on their brightness, you can create a very simple feature-vector. This feature-vector contains just one dimension, the brightness of an image, which we can easily do. Then images with similar brightness will be close to each other in that dimension. Using this representation you can very quickly search for images with the same level of brightness. However, not all aspects of images can be captures this easily, and sometimes it is not even known what aspects you would need to capture.

Indirect method Today, a much more commonly used method is to use machine learning to create a bi-encoder, this is the indirect method. Using machine learning allows for the creation of more abstract feature-vectors. For the rest of this thesis, the exact method used to create the vectors can safely be ignored. The exact model used to create a bi-encoder is dependent on the exact use case. Often, experimental results are the only way to create an appropriate bi-encoder. One example of such an encoder is BERT [Dev+19], a method that has proven to be successful in Q&A systems, such as RAG [Lew+20].

Using RAG as an example use-case, it would be difficult to use the direct construction method to create a suitable bi-encoder. This task is seemingly impossible due to the abstractness of the problem. However, creating a machine learning model to do this has been demonstrated to give adequate results [Lew+20].

1.2.2 Choosing a distance measure

A distance measure is the function used to calculate the distance between two points. One commonly used example is using a tape-measure, which measures euclidean distance. The distance measure of choice is not that relevant for the thesis. The most important thing to know is that there is a choice, and this affects what kind of results you get. Switching the distance method is not trivial, as documents previously very far could now be observed to be very close and the other way around. Construction of bi-encoders using machine learning also has to take into account the distance measure, so switching distance measure is usually undesirable. Some common distance functions are; Minkowski distances (l_p norm of the difference), Chebyshev distance, and Jaccard distance, but many more exist.

1.2.3 Filtering methods

Filtering is something that can be done in multiple ways, and each has their advantages. In this subsection we will discuss the post-filtered and pre-filtered method filtering methods.

First it is important to give some mathematical definitions. The distance function can be defined mathematically as a function $d(q, e) : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$, where q is the query vector and e the embedding of the data. Here the specific distance function is abstracted away. We can also define searching an index: $s(V, q, k) : \mathbb{P}(\mathbb{R}^n) \times \mathbb{R}^n \times \mathbb{N}$ is a search function, searching the set of vectors V for k embeddings close to q.

For filtered search we also need a filter, we can define this as such: $f(e) : \mathbb{R}^n \mapsto \mathbb{B}$, returning *True* when e should be included in the results, and *False* when it should not. In practice this filter will obtain information from an external source (like a relational database), and conclude based on its information whether or not the vector e passes the filter. The passing rate of this filter is $P := \frac{\|\{e \in V | f(e)\}\|}{\|V\|}$, where a filter with a low passing rate is considered to be selective.

Post-filtered method Similar to how we can represent a standard KNN search in SQL like Listing 1.1, we can also show how the post-filtered method works using SQL. Listing 1.2 shows how the post-filtered method would filter the results. This can also be represented mathematically as such: $\{e \in s(V, q, K) | f(e)\}$. First, a query is executed to select the K best results, and then each result is checked if it passes the filter. If it does not pass the filter it is discarded, if not, it is kept. This has a couple of downsides, one of which is that you do not know how many results you will get. You asked for K results, but how many will also pass the filter is an uncertainty. More details on why this is a major downside after after we explain the pre-filtered method.

Listing 1.2: SQL syntax to represent a post-filtered KNN search query-plan. *id* is the id column, e is the column containing the points, q is the query point, *data* is the table containing the data, K is the number of desired results, and ... is the filter to be applied.

```
SELECT * FROM (SELECT id, dist(q, e) as d FROM data ORDER BY d LIMIT K)
    WHERE ...
```

Pre-filtered method Once again we can express this is in an SQL query, as done in Listing 1.3. Note the one big difference; here the **WHERE** is executed on the data before the calculations are done. This can also be seen in the mathematical representation of this method: $s(\{e \in V | f(e)\}, q, K)$. The advantage of this is that it is guaranteed that there will be K results, unless it is impossible there exist no more candidates.

Listing 1.3: SQL syntax to represent a pre-filtered KNN search. *id* is the id column, e is the column containing the points, q is the query point, *data* is the table containing the data, K is the number of desired results, and ... is the filter to be applied.

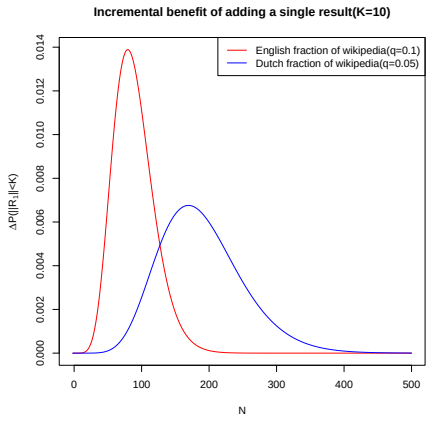
```
SELECT id, dist(q, e) as d FROM (SELECT * FROM data WHERE ...) ORDER BY d
    LIMIT K
```

To better explain the differences between the pre- and post-filtered, we will go through the process of executing the query *"Is the earth flat" between 600 and 1000 AD*. We will want exactly K results for this query, if at all possible, as this is a requirement of the RAG system. First, we will use the post-filtered method to execute the query: Here we have a clash with the requirements and what the post-filtered method can provide; the post-filtered method cannot guarantee exactly K results. The only solution is to over-ask: instead of requesting only the top K results, we ask for the top $K + M$ results. However, we do face a different problem, what should M be to provide enough results post-filter? One solution is to ask for every document in V, however this is really inefficient, and defeats the purpose of modern indices such as HNSW. It is much more efficient to set M to be a value such that there is a very high chance that the query results in at least K documents. If there still are not enough, we increase M and try again, repeating this process until we have enough results, or exhausted the dataset. Let us assume that any given datapoint e has a chance of $0 \leq p \leq 1$ to be filtered out, with a uniform distribution. We define the set of returned values to be $R_M = \{e \in s(V, q, K + M) | f(e)\}$. The probability of having less than K elements in R_M, and having to perform a follow-up search is:

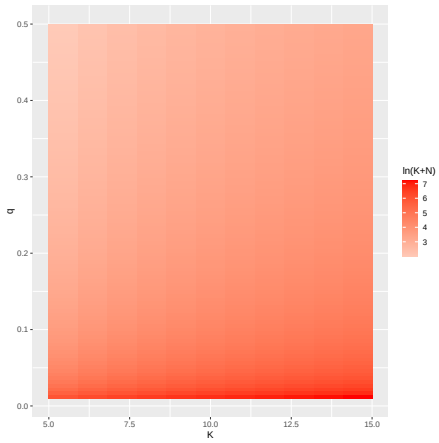
$$P(\|R_M\| < K) = \sum_{i=0}^{K-1} \binom{K+M}{i} p^i (1-p)^{K+M-i}.$$

You can see, that if we request one extra result, the chance that enough results pass the filter, $P(\|R_{M+1}\| < K)$, increases:

$$\begin{aligned}
P(\|R_{M+1}\| < K) &= \sum_{i=0}^{K} \binom{K+M+1}{i} p^i (1-p)^{K+M+1-i} \\
&= \sum_{i=0}^{K-1} \binom{K+M}{i} p^i (1-p)^{K+M-i} + \sum_{i=1}^{K-1} \binom{K+M}{i-1} p^i (1-p)^{K+M+1-i} \\
&= (1-p) P(\|R_M\| < K) + p \sum_{i=0}^{K-2} \binom{K+M}{i-1} q^i (1-p)^{K+M-i} \\
&= (1-p) P(\|R_M\| < K) + p \sum_{i=0}^{K-1} \binom{K+M}{i-1} p^i (1-p)^{K+M-i} \\
&\quad - \binom{K+M}{K-1} p^{K-1} (1-p)^{M+1} \\
&= (1-p) P(\|R_M\| < K) + p P(\|R_M\| < K) - \binom{K+M}{K-1} p^{K-1} (1-p)^{M+1} \\
&= P(\|R_M\| < K) - \binom{K+M}{K-1} p^{K-1} (1-p)^{M+1}.
\end{aligned}$$



(a) The incremental benefit in success-rate of requesting a single extra document, $K = 10$



(b) $log(K + M)$ with M such that the incremental benefit peaks at M. Brighter color means that the peak is at a higher M

Figure 1.2: Plots pertaining to the absolute increase in passing-rate, and the value at which this increase is the greatest.

When we plot the incremental part of this equation, i.e., the difference gained by increasing the limit M, in Figure 1.2a; we can see that initially increasing M gives a lot of value. However, it also shows that after a certain M it no longer provides much value. The first extra result will result in a higher probability increase than the later ones. If you are already requesting a lot of results, the likelihood that one extra will do the trick is small. Using this formula you could try to optimize M, and find a good limit for potential follow-up queries.

Figure 1.2a more clearly shows that there are diminishing returns. After a certain point, additional results only marginally increase the higher chance of getting enough data elements that pass the filter. When this happens depends on p and K. A smaller p means more results are necessary. For small enough p, it would be necessary to retrieve incredible amounts of data from the vector database to get a somewhat reasonable chance of success. As seen in Figure 1.2b it is not unrealistic to need orders of magnitude more query results for a good chance to get enough useful overall query results.

Let us now consider the pre-filtered method. Using this method it is not necessary to find some ideal M, since this method is guaranteed to result in K valid results, provided these exist in the dataset. There are no inherent complicated issues with using the pre-filtered method. However in practice the metadata and vectors are often stored in different places, vectors in an index and metadata in Relational Database Management System (RDBMS). The filter f however, is based on the metadata, the publishing date of the document. For pre-filtered methods to work, some information has to go from the relational database to the index, accelerating KNN searches. What information exactly

depends on the implementation, but in some setups this may be an issue.

1.2.4 FAISS

FAISS [JDJ17] is a library implementing a state of the art vector database, with many tunable parameters. This subsection reviews the relevant options that we take advantage off.

A vector database is usually build around one or multiple indices, data structures that are designed to perform KNN search queries really efficiently. What sets these indices (or access structures) apart is the specific details. Some take a long time to setup, but may be really space efficient, while others may be very fast to load all the data into, but are relatively slow to query. Each and every access structure proposed in the literature has its use case. An access structure may have parameters, for both the construction and querying the index. The combination of a access structure and some specified parameters will be referred to as an index configuration.

FAISS supports many configurations. Each configuration provides an implementation of a search function s. As not all access structures used by FAISS are exact, for these access structures it is not a perfect implementation of a search function; it is not guaranteed that there are no vectors closer to q but not in the results. It is also possible that 2 vectors may be equally distant from q, it is not clearly defined what happens in this case. For the case of simplicity we will assume these imperfections do not exist. It does not affect any part of the thesis.

Chapter 2

Possible Architectures of Filtered KNN

Just as there are multiple methods for filtered search, there are multiple architectures that allow for the combination of a RDBMS, and an optimized KNN search index. In this chapter we discuss the two relevant architectures, one that is commonly used, and another that we implemented for the thesis.

2.1 External layer

The most popular method of implementing a filtered similarity search, is to use two different databases as can be seen in Figure 2.1. This has some upsides, as it does not put any requirements on the individual database systems. Any RDBMS will be fit for purpose, and you have the freedom to choose any vector database as well. One of the major benefits is that there is no need to use a different RDBMS when adding search functionality. If the current setup works, extending it with vector search only requires the user to setup a vector database and connect them in software. Although the complexity cost of this overarching system can be high, switching from one database to another can also be a complex process [EA15]. Although this may seem attractive at first, there are also some major downsides. Main downside is that the movement of information is expensive. Getting results from the vector database to the relational database is cheap enough, because this is a small amount of data. However, using a pre-filtered method with this setup would not be feasible. Using a pre-filtered method would require sending information from the relational database to the vector database. Even the most minimal amount of data, one bit per row, would require approximately 122 KiB for just one million rows. Given that datasets can grow way beyond this size, it is easy to imagine that this can be prohibitively expensive.

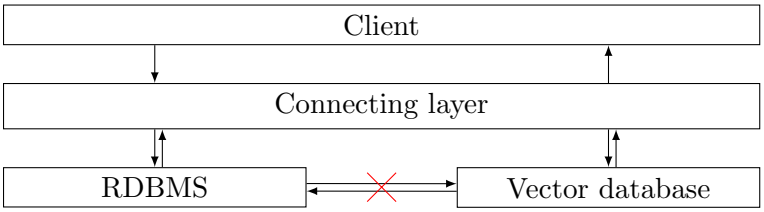


Figure 2.1: Diagram describing currently common architectures to implement filtered search. The overarching system communicates with both the RDBMS and the vector database. Notably there is no communication between the relational and vector database.

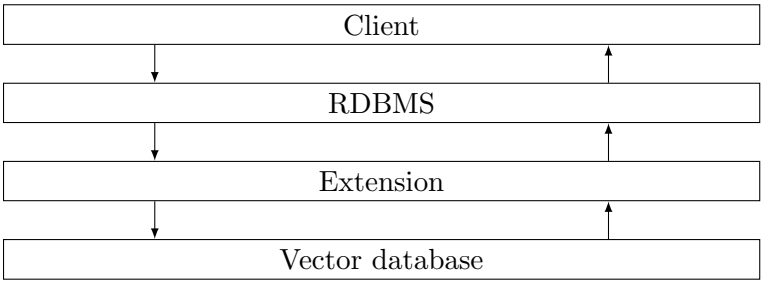


Figure 2.2: Diagram describing the architecture for a database integration. The client makes a request to the RDBMS, which passes information down to the extension layer, which passes it through to the vector database. Then the results are passed up from the vector database all the way, through the later back to the RDBMS. The RDBMS then finishes the query and sends it to the client.

2.2 Database extensions

Previously we saw that the main concern with the state of the art is having to either request a lot of records or sending over a lot of data to the vector database, and both options are costly or wasteful. Here we will describe a different way to provide similar functionality, by integrating an existing vector search database into an existing RDBMS, as depicted in Figure 2.2.

A database extension has much lower overhead when communication between the relational and the vector databases, meaning the cost of requesting for more results is cheaper. The biggest benefit is that it is no longer an issue to exchange data between the RDBMS and the vector database system. This is made much cheaper, since there is no networking involved. The extension can be small and efficient.

For the purpose of the thesis a Proof Of Concept (POC) was created based on FAISS. The POC implements everything we need, but without testing on edge cases[1]. This limitation is not relevant for the goal of the thesis, but will impact usability of the

[1]One example of this is that querying more than 2048 vectors at once is untested, but many more edge-cases exist.

POC for a more general purpose. The POC is based on an initial version created by Hannes Mühleisen, which implemented basic append and search functionality for vectors as an extension on DuckDB [RM19]. A large part of the initial code was rewritten and new functionality has been added. This is the final list of features:

- basic lifetime operations: creating/deleting an index, saving/loading an index;
- append vectors;
- functionality to manually train the index;
- querying the n closest vectors, including search parameters;
- querying the n closest vectors where a certain condition holds (a pre-filtered search);

For all these items, it is possible to set FAISS index construction and search parameters. Pre-filtered search is implemented in two ways, the first using a selection vector, and the second using an inclusion set. The inclusion set method allows more optimizations on the DuckDB side, since it can directly use an index, however it has to build a set. Building a set can be slow for large numbers of elements. The selection vector method scans the entire table and creates a large vector, where each element in the vector indicates whether the element at that location should be included or not. This is much faster, but you always have to create a very large vector, as you need to have one element per datapoint.

Most operations are one to one mappings from DuckDB to FAISS, with most code dedicated to data format conversion. The exception is the query with filter functionality. This operation needs to know for each row of the relevant table whether or not a certain condition applies. However, arbitrary functions can not take in tables as arguments. Thus we need to construct a query, execute this, and convert the result into a format FAISS understands. This process is more difficult when the row IDs are not sequential, as that complicates the implementation. Row IDs that are not sequential are also slower to process due to the increased code complexity.

2.2.1 Comparison against external layer

Compared to the external layer, the POC demonstrates a couple of advantages, stemming from the fact that we have deeper integration into the database, and run on the same machine.

The first advantage is reduced complexity of the connecting layer and infrastructure. The middle layer is traditionally complex, and also requires infrastructure. Although the decision on how many results to fetch still need to be made somewhere, this can now be done in SQL. Moving the middle layer to SQL means that everything happens in the RDBMS, and thus no need for any additional infrastructure. When the middle layer is in SQL, the programmer also has access to a much more abstract language. With

an external middle layer, the middle layer has to convert dataformats, generate correct SQL queries, and handle edge cases *etc.*. Using a middle layer written in SQL you do not have these problems. The data is already in the right format, no SQL queries are generated, and edge cases are handled by the database. This reduced complexity results in lower maintenance costs, and higher reliability of the overall system.

It is important to note that the basic problem with having a middle layer are still present. The communication overhead may be largely removed, but if more results from the vector query are required, the query still needs to be recomputed, and this logic has to be implemented somewhere. The complexity has been moved to a more convenient place, one where data movement is less costly and complexity reduced; but, moving a problem does not solve basic computational restrictions. To do that, a new approach would have to be developed.

Moving the complexity to the database has another big advantage though; recall from Section 2.1 the challenges of filtering a vector database search when the dataset is large. Since we integrated the vector store into the database we can quickly identify which rows pass a certain condition. This allows us to do a filtered vector search inside the vector database. The vector database can guarantee that all the results pass the filter, and thus there is no need to over-ask. This is not always beneficial, and depends heavily on the dataset and kind of filter. In a later section we will evaluate empirically the tradeoffs between both approaches.

2.2.2 Comparison against VSS extension

Recently, the DuckDB team released the VSS extension. It provides an interface much like pgvector[2], but in a more limited form. Since VSS is in its early stages, more features are to come, and it likely will get to feature-parity to pgvector. Extensions like VSS and pgvector integrate deeply into the database system, in this case DuckDB. VSS and the FAISS extension work in similar ways under the hood, but the exposed API introduces limits to both. For now, we will ignore inherent advantages of using FAISS or USearch [Var24] (used by VSS), just focus on the advantages of the interface itself.

VSS registers a DuckDB index on a (vector) column. This imposes a couple of restrictions, as the column with the vectors must thus exist in DuckDB. If this column were to be removed, so would the index. However the datapoints must also be retained in the HNSW, leading to data-duplication in memory (RAM or potentially spilled over on disk). When the user performs a top-N query, the extension attempts to optimize this to a search using the access structure. However the distance functions built into DuckDB, do not have the ability to take in arbitrary parameters. You cannot set search parameters per-query, only on the creation of an index. This limits the flexibility of the index. Another limitation is that using the index for KNN search requires the optimizer to detect the query pattern and optimize the query to use the index. This is currently quite limited, in practice, for example, it can only optimize when using a literal query, not one obtained from other query or using a join, preventing batching operations for

[2] `https://github.com/pgvector/pgvector`

example. A fully integrates extension, like VSS, should be noted that this last point can be resolved by simply making a better optimizer, or allowing manual access to the index using a custom function. It does however also have some advantages. Since everything in the index is managed by the database system, the index is also stored in the database file. This makes it easy to store the database and more it from machine to machine. The creation of the index is similar to normal DuckDB indices. All of these advantages improve the user-experience for the most common use-cases.

The FAISS extension developed for this Bachelors thesis offers a different interface towards its users. The interface is just a collection of functions, that have to be called manually. When using the FAISS extension, the user has to be deliberate on how the index should be used. Creating an index is done using a function call, passing an index identifier, which is just a string. Index parameters need be passed using a DuckDB MAP type instead of the nicer syntax provided by the **CREATE INDEX** syntax. Querying the index is performed with another custom function, instead of the distance functions in DuckDB, returning a list of results. This does not integrate as nicely with the database, forcing the user to explicitly perform a join on the metadata. This can be seen clearly in the long queries used for the thesis, later in the method section. Since the index is not registered within DuckDB, the user has to save/load the index from disk manually, storing a separate file for the FAISS index. However this also has some advantages. Notably you do not have to replicate the vector-data itself in DuckDB. Since the FAISS index is not linked to the column with the vectors, it is completely safe to remove the column with vector-data from DuckDB such that all the data resides in FAISS. This interface also allows for changing the search parameters for every search performed. Every search function takes in the search parameters, and passes them to FAISS. Storing the data separate files for the FAISS index and DuckDB database has the advantage that it is easy to distribute the FAISS index across machines. In summary, this API allows for more control and flexibility, at the cost of the user-experience.

Chapter 3

Method

To objectively evaluate the performance of the different designs for filtered KNN search discussed so far, we introduce a filtered vector search benchmark. In this Chapter, we specify the metrics to evaluate the configurations, the data for the comparison, index configurations, the precise selection methods, and the benchmark system.

3.1 Choice of metric

The first choice is to decide what to measure. This is crucial, as it can influence many choices down the line. The option chosen here was to benchmark the latency of executing a batch of 43 queries. The usage of batches was decided early on, as FAISS performs much better on batches than individual queries. The precise number of queries was coincidental, as it is the number of queries that exist for our dataset. We know these queries are fit for the data, and adding more could have affected the search process. For example, the data may have some natural, non-uniform distribution. Picking a query vector at random might land in an convenient/inconvenient spot affecting the results.

Since we do not have to make a new index for the different querying methods, we do not consider indexing time. Nor do we consider recall as a measure to compare the indices, as we focus on the query time between different methods. As we will describe later, we do consider recall for the selection of index.

3.2 Data and Queries

The dataset used for the benchmarks is based on the MSMARCO dataset, with queries from the TREC 2019 deep learning track. This mirrors research like [Lin+23], using their *openai-ada2* embeddings for data and queries. This potentially allows us to compare our results to theirs. The dataset contains approximately 8.8 million documents, and is commonly used for Q&A research. [Lin+23] focused on the retrieval of documents, which makes it easy to compare the methods discussed in the thesis for usage in Q&A systems like RAG [Lew+20].

3.3 Chosen Indices

Three common FAISS index configurations are included in the evaluation. The final decision was an IVF+HNSW index configuration, default parameters for HNSW and 2048 clusters, one IVF+HNSW configuration with default HNSW and 65536 clusters, *nprobe* of 32, and a HNSW configuration with *efConstruct* of 100, M of 32, and *efSearch* of 1000. These indices are specified in detail in Table 3.1 The first fits approximately 4000 documents in each cluster, which allows us to keep *nprobe* at 1. This allows for quick testing, and very quick training. The second option corresponds to the FAISS recommend to the FAISS recommended configuration[1], together with an *nprobe* sufficient to give enough results. Finally the last one was chosen because it matches the parameters used in [Lin+23].

The default HNSW index configuration from FAISS was not considered, because it was materially insufficient. In spite of its usage in the RAG paper [Lew+20]. Although the paper does not specify which configuration was used, the code uploaded to HuggingFace[2] leaves the HNSW unmodified when creating the index. Surprisingly, this configuration does not return sufficient results for our use case. Although we did not check in detail, the graph created for the HNSW is likely disconnected, which prevents it from returning enough results for our use case. Since in [Lew+20] the minimum value for K was 10, we want to match this in our queries.

The IVF65536+HNSW was configured with $nprobe = 32$ because it ensured enough results for every query on the most selective filter ($P = 0.01$). We saw empirically that increasing this parameter increases the recall, but with diminishing returns. This is why $nprobe = 32$ was chosen for this index configuration.

name	FAISS factory string	construction parameters	search parameters
IVF2048	IVF2048_HNSW32	-	-
IVF65536	IVF65536_HNSW32	-	$nprobe = 32$
HNSW128	HNSW128	$efConstruct = 100$	$efSearch = 1000$

Table 3.1: Specification of each index configuration.

3.4 Query Processing Methods and Selection Process

Three query processing methods were selected for the use in the comparison, two pre-filtered methods, and one post-filtered method. The pre-filtered methods are similar in that they give FAISS a filter and only return results that pass the filter. One uses the existence in a set to determine validity, which can benefit from database optimizations. Internally in FAISS this is converted to a bloom filter and a C++ unordered set. The cost of creating such a set heavily depends on the size of the set, and thus the selectivity

[1] https://web.archive.org/web/20240508155335/https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

[2] https://github.com/huggingface/transformers/tree/main/examples/research_projects/rag

of the filter. The other pre-filtered method uses a selection vector, where ith element is 1 if the document with id i should be included, and 0 otherwise. Creating such a vector does not depend on the selectivity of the filter, only on the size of the dataset. The third method, post-filtered, is the industry standard to retrieve a lot of documents. The target was set using a 99% probability that at least 10 documents would pass, if we applied the filter afterwards. This leaves a 1% chance that not enough results will pass. A higher chance of success would mean a skyrocketing amount of retrieved documents, and most systems might not require exactly 10 results. The amount of retrieved documents may be off by 1, however this makes no discernible difference in the results. This calculation was made under the assumption that all results are independent, that is to say if result number X were a valid result that does correlate with result number Y also having a higher probability being valid.

The post-filter method requires a join on the table containing the metadata. This makes it essentially free to also fetch the metadata. In many cases this is useful, and something you would want to do. To make the comparison fair, metadata was fetched for all queries. This means all queries performed a join on the metadata table. This keeps the results fair, and relatable to real use cases.

To control filter selectivity in a structured manner, a simple non-random process was followed. We use the selection constraint $(docid\%100) \leq P * 100$, where P is the passing rate of the filter. This process is not random, but it is uniform. Under the assumption that there are no repeating patterns in the dataset, this will create a uniform distribution.

name	type of filter	implementation
filterpost	post-filter	database join
filtersel	pre-filter	selection-vector
filterset	pre-filter	bloom-filter + exact set
VSS	post-filter	database join

Table 3.2: Specification of each index configuration.

Listing 3.1: Query used for *filterpost*. $K + M$ is the amount of results such that with 99% probability, 10 results can be obtained, $searchParams$ is the search parameters for the index, p is P, the passing rate of the filter.

```
SELECT * FROM
    (SELECT
        qid,
        UNNEST(faiss_search('flat', K+M, embedding, searchParams),
            recursive:=true)
        FROM queries)
    JOIN ids ON label=id WHERE sel<p
```

Listing 3.2: Query used for *filtersel*. $searchParams$ is the search parameters for the index, p is P, the passing rate of the filter.

```
SELECT * FROM
```

```
(SELECT
    qid,
    UNNEST(faiss_search_filter('flat', 10, embedding, 'sel<p', 'rowid',
        'ids', searchParams), recursive:=true)
    FROM queries)
JOIN ids ON label=id WHERE sel<p
```

Listing 3.3: Query used for *filterset*. *searchParams* is the search parameters for the index, p is P, the passing rate of the filter.

```
SELECT * FROM
    (SELECT
        qid,
        UNNEST(faiss_search_filter_set('flat', 10, embedding, 'sel<p',
            'rowid', 'ids', searchParams), recursive:=true)
        FROM queries)
    JOIN ids ON label=id WHERE sel<p
```

Listings 3.1, 3.2, and 3.3 specify the exact queries used. The search parameters for each index configuration are listed in Table 3.1. Each of these queries does a batch search, filtered or unfiltered, and then joins this on the table containing metadata (*ids*). In this join, there is also a condition, which DuckDB pushes down to the scan on *ids*. The queries are stored in the *embedding* column of *queries*, and the *sel* column of *ids* contains the values *docid*%100 as described above. *docid* is stored in the *id* column of *ids*. As can be seen in Listings 3.2 and 3.3 the filtered search functions take additional arguments. $sel < p$ is the filter to be applied, *rowid* is the *rowid* of each row, which in our case is equals *docid*. Due to internal optimizations in DuckDB, using the *rowid* is faster than using docid. In the future, DuckDB may allow the same optimization when using sequential IDs. The impact of this optimization is approximately 7ms per batch, which is significant. We chose to make use of this optimization because we do not want to be limited by implementation details of DuckDB, which are subject to change.

All benchmarks were performed on a P from 0.01 to 1 with steps of 0.01. This was chosen because there was no prior research on the performance of pre-filtered vs post-filtered KNN searches, and we do not know what range would be particularly interesting. We decided on a broad scan of the entire range.

3.5 Benchmark System

The benchmarks were performed on an Intel i9-9900k, with 64 GB RAM, on stock settings. This system has 16 total hardware-threads available. This is part of a shared homeserver, seeing little use at the time. For the comparison to VSS, more RAM was needed. This comparison was performed on a 2 x Xeon E5-2670 with 256GB RAM and hyperthreading disabled, provided by iCIS. This system also has 16 hardware-threads enabled.

The benchmarking code was written in C, using the GO benchmarking ecosystem. To prevent any overhead from the GO $\leftrightarrow$ C barrier, the executing of the queries was

done in C. This provides both the reliability and performance of using C but the benefits of using the go ecosystem. Each benchmark was run approximately 30 seconds.

All benchmarks were run in two settings: with a single thread and with all hardware-threads enabled. The reason for this is that running with just one hardware-thread enabled normalizes for CPU usage. The first is important because in most server applications the total throughput is most important. When limiting the query to one hardware-thread, it gives a clear throughput per hardware-thread number. Because in real applications multiple batches would be run in parallel, allowing each batch to also parallel tasks themselves would interfere with other batches. This would hinder the throughput. Thus when running on just one hardware-thread FAISS was also instructed to not use multiple threads. Disabling the lower level parallelism would also remove any communication overhead of parallelism within FAISS, actually improving throughput.

3.6 Comparisons to VSS extension

In its current implementation, the functionality offered by the new VSS extension is limited. It is currently not possible to execute batch queries for example. The number of rows supported is also insufficient for the full benchmarks[3]. For this reason, only a subset of data was used. We used the default VSS HNSW, and matched the configuration for the FAISS HNSW. The configurations can be seen in Table 3.3. The benchmarks were run on the server provided by iCIS. An non-indexed SQL search was also added to provide context to the results. Due to the limitations of VSS, only $1M$ rows were indexed in the queries.

name	FAISS factory string	construction parameters	search parameters
VSS	N/A	-	
HNSW32	HNSW32	$efConstruct = 128$	$efSearch = 64$

Table 3.3: Specification of each index configuration.

If we were to emulate a batch query using VSS, VSS would be disadvantaged as it would have to perform 48 individual queries. However, if we used individual queries, we would be disadvantaging FAISS as it is optimized for batch queries. Because there is no inherent advantage of either option from an application perspective, it was decided to report on both batch and individual queries.

[3] `https://github.com/duckdb/duckdb_vss/issues/16`

Chapter 4

Results

In this Chapter we discuss the validity of the indices, and report on the throughput of each index configuration individually. First we look at the quality of the results of the approximate KNN queries using index configurations.

4.1 Index validation

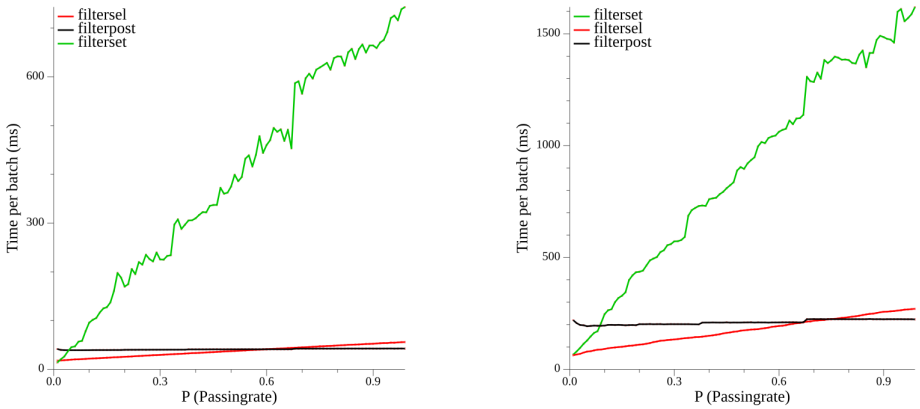
Validation the indices is important for multiple reasons. We select the test data to be representative, and in the real world a bad index configuration would not be chosen. Another important function is to verify the functionality of the extension. The validity of the filter functionality was checked manually, as recall is meaningless on such an artificial query.

Index configuration	Recall@1000
IVF2048	0.6676
IVF65536	0.6533
HNSW128	0.7867

Table 4.1: Recall@1000 per index configuration for dl19 passage using anserini's openai ada2 embeddings

From Table 4.1 it is clear the functionality of the extension works, as recall reaches 65 to 80%. We conclude that HNSW128 is superior to the other index configurations, with a much better recall score of 0.79, as opposed to 0.65. The other index configurations have worse recall performance, but they still show they are valid indices.

4.2 IVF2048



(a) All cores (b) One core

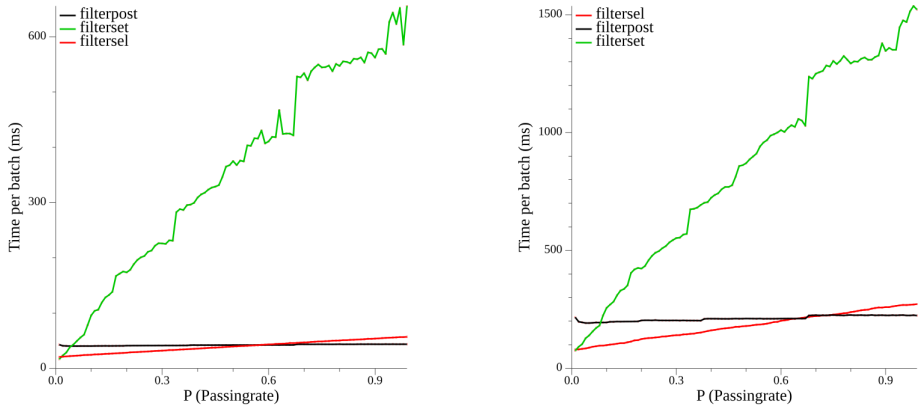
Figure 4.1: Time in ms per batch (of 48 queries) over the passingrate of the filter, using the IVF2048 configuration. Both using many cores and with just one available. Selectivity of n% means only n% of the data passes the filter.

Figure 4.1a shows clearly the time complexity of set filtering and selection-vector filtering. Set filtering has to build a set, where the costly operation is the creation of the set which is $O(m)$ where m is the amount of total datapoints that pass the filter. This leads to long query times when a lot of elements must pass the vector. The selection-vector filter method has to build a long bitmask with length n, the total amount of datapoints, leading to a complexity of $O(n)$ for building this vector. This shows in the graph as this method has an almost horizontal line. It does slowly go up, this is due to the final join getting bigger. This join is of size $m \times K$, and m grows linearly with the passingrate, making the query processing slightly slower when the passing rate is high. The *filterpost* method is more like the *filtersel* method, with a flatter curve. An explanation for this is that unlike the *filtersel* method, where the join is of size $m \times K$, the *filterpost* method has a join of size $m \times (K + M)$. Here m grows as the passingrate gets higher, but M shrinks, partially offsetting the growing of m.

When limiting the benchmark to just a single core as shown in Figure 4.1b, we can see something interesting happening. The post-filtered method slows down unilaterally, by a factor of approximately 5. This is due to the highly parallel nature of this method. However compared to the factor of 16 difference in thread count this is relatively small. The *filtersel* also slows down, however this is not the same across the board. Filtersel slows down similarly as the *filterpost* method at as high passingrate, and around 4 times at lower passingrates. The *filterset* method slows down most at low passingrates, but a lot less at the higher passingrates. This can be explained by the fact that at a higher passingrate this method is almost entirely sequential. This sequential code sees no slowdown when limiting the amount of cores.

For this index configuration the *filtersel* approach is faster than *filterpost* up to around the $P = 0.6$. However it it not slower by a lot at a high P. This makes filtering ideal if you expect a low P but are not sure you will always get it.

4.3 IVF65536



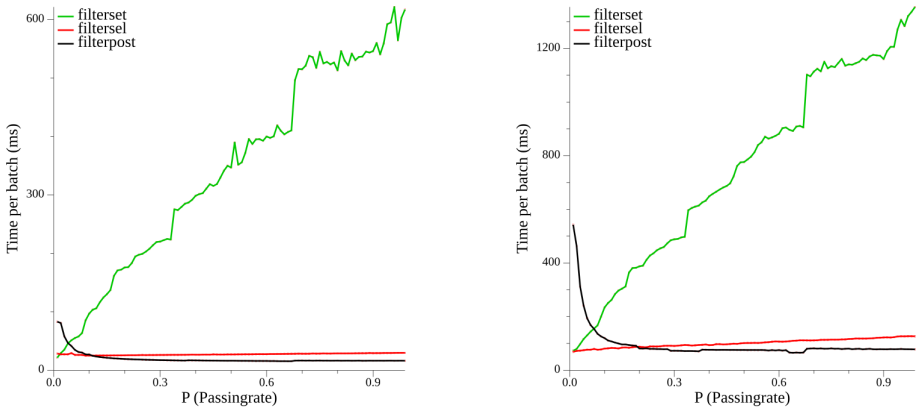
(a) All cores (b) One core

Figure 4.2: Time in ms per batch (of 48 queries) over the passingrate of the filter, using the IVF65536 configuration. Both using many cores and with just one available. Selectivity of n% means only n% of the data passes the filter.

Figure 4.2a looks almost the same as Figure 4.1a, not that surprising since only a few parameters differ. First they use a HNSW32 index configuration to search for the nearest clusters. Then both indices have to go through exactly the same number of points to check. This means that the only real difference affecting latency is that the IVF65536 has to combine the results from 32 clusters while IVF2048 does not, making the query processing slightly faster.

As shown in Figure 4.2b, the effects from limiting the system to a single core are much the same as compared to an IVF2048. One thing that is very visible in Figure 4.2b but not in Figure 4.1b is the shift of the crossing point from *filtersel* and *filterpost*. It lays much further to the right when limiting to just a single core. This would indicate that when limiting in a system where throughput is prioritized, running the pre-filtered methods is more lucrative than one where latency is important.

4.4 HNSW128



(a) All cores (b) One core

Figure 4.3: Time in ms per batch (of 48 queries) over the passingrate of the filter, using the HNSW128 index configuration. Both using many cores and with just one available. Selectivity of n% means only n% of the data passes the filter.

Figure 4.3a shows much of the same as before, except for higher P values. Near 0 it shows a great increase in latency. In Figure 4.1a and Figure 4.2a this may also be observable, but it is very clear in Figure 4.3a. More research could be done to understand and explain this.

Compared to Figure 4.1a it shows a smaller advantage for the pre-filtered methods. With $P \leq 0.1$ the pre-filtered methods are at an advantage. Especially the selection-vector method does very well. Also very notable is that the post-filtered method performs especially badly below 5% specificity. The uptick near $P = 0$ is much more noticeable in Figure 4.3a than in Figure 4.2a.

With a very low selectivity, the *filterset* method is the fastest. However due to the slowness of creating the set, it quickly becomes irrelevant. Much more promising is *filtersel*, where the performance does not depend on the selectivity of the filter. It remains faster than the *filterpost* method for much longer. However, the *filterpost* method does perform much better for the majority of the tested range.

Figure 4.3b shows similar effects as seen for the other indices. One thing that is more obvious in this one is that the crossing line for *filterset* and *filterpost* has shifted more to the right.

4.5 Post-filtered against VSS

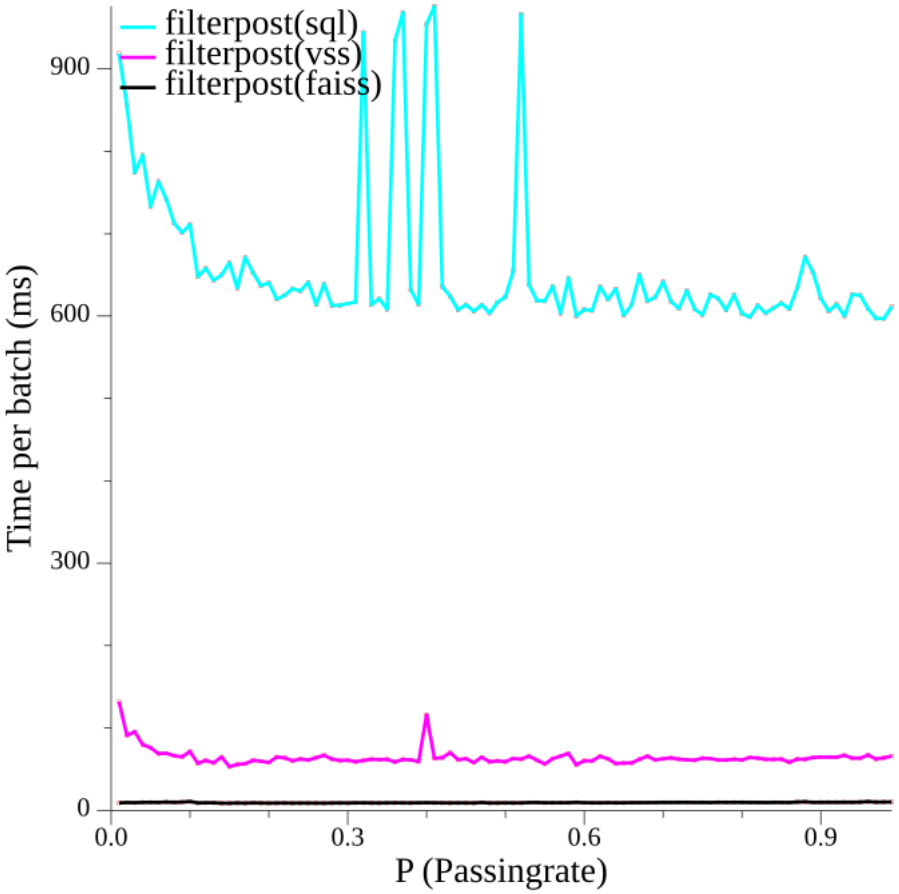


Figure 4.4: Time in ms per query over the passingrate of the filter, using the HNSW32 and VSS index configurations, and a non-indexed query. Selectivity of n% means only n% of the data passes the filter.

Here we compare the performance in terms of latency of the different unfiltered configurations. From Figure 4.4 it is clear that the pure SQL version of this query is the slowest variant. What is more interesting is that the VSS extension is slower than the FAISS extension. VSS uses the USearch library, which promotes itself on being up to 10X faster than FAISS. However this benchmark shows a very different picture, at least in this configuration. It is possible that the VSS extension layer adds a lot of overhead, however, the source code does not show any obvious indication that this might be the case. These results seem to raise doubts on the USearch claims.

Benchmarks released by USearch indicate that the performance varies a lot, depend-

ing on the number of documents[1]. Since we only have $100K$ documents in our index, Figure 4.4 is likely not an accurate representative for the relative performance that could be achieved on the full dataset.

[1] https://github.com/unum-cloud/usearch-benchmarks/blob/main/plots/100m-1536d-search-speed.png and https://github.com/unum-cloud/usearch-benchmarks/blob/main/plots/100m-96d-search-speed.png

Chapter 5

Discussion

Although the results clearly show that the pre-filtered methods can perform on-par with, and often even better than, post-filtered methods, there are still points unclear or up for discussion. In this section we will discus many points mentioned earlier, but also some unmentioned challenges. First there is a section on future research, and then we will discuss limitations of the research.

5.1 Future research

One of the assumptions made in the experimental setup of this work is that the document IDs have no correlation with the location of the document in the embedding space. This is of importance, because a correlation could affect the search performance of FAISS. If only few points around a query pass the filter, FAISS will have to search for longer to find appropriate results, and vise-versa. No research was done on a potential correlation between the document IDs and their location in embedding space. Because we evaluate using 48 different queries, it would be expected that any advantage or disadvantage would even out. Only if a large amount of query embeddings happen to be near/away from hotspots (places the filter passes more frequently), this would be observable. This chance was deemed too slim to take into account. It would have been possible to assign each document a random number and base the filter on this, instead of the document ID. Future research should probably use this method to avoid the possibility all together.

In Figure 4.3 we saw that near $P = 0$ the latency of the *filterpost* method increased dramatically. From these tests it is not possible to find out why. More research could be done to look into this. One possibility is that traversing the HNSW gets very inefficient if a large number of points are excluded, possibly related to cache-size or other factors. Future research could attempt to profile or debug the HNSW to investigate the cause of this inefficiency.

Sections 4.2, 4.3, & 4.4 all showed bad performance for the *filterset* method. This was due to building of the set within FAISS. FAISS uses the C++ unorderedmap internally, together with a bloom filter. The creation of this set consists of inserting every element into the set. Insertion performance of unordered_set, based on unordered_map, is

noticeably slow. In this case prohibitively slow, as can be seen in the graphs. A faster implementation could make this implementation much more viable. Implementations that offer an order of magnitude faster set insertion time exist [Mar22]. It was out of scope for the thesis to modify FAISS for this purpose, however it would be interesting to evaluate this in more detail.

These benchmarks use batched queries, and this may not be a realistic model for a practical deployment. However, the biggest cost for pre-filtered methods is the creation of the filter, and this does not depend on the query q. There are other ways to increase the efficiency of these queries. One example is to dynamically batch user queries that use similar filters, and apply the full filter using post-filter method. This hybrid approach could dramatically reduce the number of results fetched from FAISS, while also allowing for more batching. The filter could be constructed dynamically based on the queries that come in. This would make the system more complex, but could improve performance.

The results for the comparison against VSS are surprising. USearch is very proud of their performance compared to FAISS, but this benchmark does not reflect this. Instead of a 10X performance benefit for USearch, the results are close to a 10X performance deficit. Looking into why this is could prove valuable, for both USearch and VSS, as it could improve their performance.

5.2 Limitations

For the benchmarks we reported query latency instead of $\frac{batch}{t}$. This was done because the pre-filtered methods have a single-threaded component to them, making them not fully parallel. Therefore the setup we used, where batches are executed sequentially, would not allow for maximum $\frac{batch}{t}$ performance. As an example, the *filterset* method is almost entirely single-threaded at a high selectivity. Multiple batches could be ran at the same tome to increase total $\frac{batch}{t}$. The single-threaded parts can be parallelized, but further research must be done to determine the effectiveness of this. Together with a better implementation for unordered_set, this would be a much more interesting comparison. The benchmarks which were limited to just a single CPU core can be more easily converted to a throughput number, as they are effectively normalized for CPU usage. But other factors can still play a role, such as speed of components beyond the CPU.

As mentioned in the method, the performance numbers were obtained using a shared system, with multiple users. However at the time of benchmarking it was periodically checked that no other processes were interfering with the benchmarks. At the time the benchmarks were run there were no other users active on the system.

We used the go ecosystem for benchmarking, as mentioned in Section 3.5. This was done because it is more user-friendly than the alternatives. To verify that the GO $\leftrightarrow$ C interface did not interfere with the benchmarks, a part of the results were verified using the DuckDB benchmarking suite. The results from this verification run showed no significant differences between the DuckDB suite and our suite. This was not done for all benchmarks since it takes significantly more effort to use the DuckDB

benchmarking system; at least at the time the decision had to be made. The reason the DuckDB benchmarking suite was not used for the benchmarks, is that it does not equalize the runtime of each benchmark. It is also difficult to use with custom, and unsigned, extensions.

The recall of the VSS extension was not measured. As it is unable to load more than $300K$ documents, measuring recall would be ineffective. It is thus entirely possible that the VSS extension may not be working as intended. Due to a lack of time, this was not looked into further.

Bibliography

[Knu73] Donald Ervin Knuth. *The art of computer programming*. Second ed. Reading (Mass.) London Manila [etc.]: Addison-Wesley publ, 1973. ISBN: 978-0-201-03822-4 978-0-201-89685-5.

[FH89] Evelyn Fix and J. L. Hodges. "Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties". In: *International Statistical Review / Revue Internationale de Statistique* 57.3 (Dec. 1989), p. 238. ISSN: 03067734. DOI: 10.2307/1403797. URL: https://www.jstor.org/stable/1403797?origin=crossref (visited on 06/19/2024).

[EA15] M. Elamparithi and V. Anuratha. "A Review on Database Migration Strategies, Techniques and Tools". In: *World Journal of Computer Application and Technology* 3.3 (Dec. 2015), pp. 41–48. ISSN: 2331-4982, 2331-4990. DOI: 10.13189/wjcat.2015.030301. URL: http://www.hrpub.org/journals/article_info.php?aid=3392 (visited on 10/30/2023).

[JDJ17] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: (2017). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1702.08734. URL: https://arxiv.org/abs/1702.08734 (visited on 09/17/2023).

[MY18] Yu A. Malkov and D. A. Yashunin. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. Aug. 14, 2018. DOI: 10.48550/arXiv.1603.09320. arXiv: 1603.09320[cs]. URL: http://arxiv.org/abs/1603.09320 (visited on 06/19/2024).

[Dev+19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[RM19] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam Netherlands: ACM, June 25, 2019,

pp. 1981–1984. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3320212. URL: https://dl.acm.org/doi/10.1145/3299869.3320212 (visited on 06/20/2024).

[Hum+20] Samuel Humeau et al. *Poly-encoders: Transformer Architectures and Pre-training Strategies for Fast and Accurate Multi-sentence Scoring*. Mar. 25, 2020. arXiv: 1905.01969[cs]. URL: http://arxiv.org/abs/1905.01969 (visited on 12/25/2023).

[Lew+20] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

[Lin21] Jimmy Lin. "A Proposed Conceptual Framework for a Representational Approach to Information Retrieval". In: (2021). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.2110.01529. URL: https://arxiv.org/abs/2110.01529 (visited on 09/17/2023).

[Mar22] Leitner-Ankerl Martin. *Comprehensive C++ Hashmap Benchmarks 2022*. martin.ankerl.com. Aug. 27, 2022. URL: https://web.archive.org/web/20240604110905/https://martin.ankerl.com/2022/08/27/hashmap-bench-01/ (visited on 06/04/2024).

[Lin+23] Jimmy Lin et al. "Vector Search with OpenAI Embeddings: Lucene Is All You Need". In: (2023). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2308.14963. URL: https://arxiv.org/abs/2308.14963 (visited on 09/17/2023).

[Var24] Ash Vardanian. *USearch by Unum Cloud*. Version v2.12.0. Apr. 29, 2024. DOI: 10.5281/ZENODO.11082388. URL: https://zenodo.org/doi/10.5281/zenodo.11082388 (visited on 06/20/2024).