

Vrije Universiteit Amsterdam



Universiteit van Amsterdam



Master's Thesis

Toward Efficient Filtered Vector Similarity Search in Analytical Databases

Author: Simon Leonardus Johannes van Noort (2758494)

1st supervisor: Prof. Dr. Peter Boncz
daily supervisor: Leonardo Xavier Kuffo Rivero MSc.
2nd reader: Prof. Dr. Hannes Mühleisen

*A thesis submitted in fulfillment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science*

July 25, 2026

Abstract

Vector similarity search (VSS) finds objects in a dataset that are similar to a given query object across data types such as text, images, audio, and video. VSS is widely used and increasingly powers retrieval-augmented generation (RAG) for large language models (LLMs). When this semantic search is combined with SQL predicates, the result is filtered VSS — a capability that many relational OLAP DBMSs now support by integrating a VSS index, thereby becoming vector-extended systems. These integrations, however, are often loosely coupled: only the system is adapted to fit the index. Furthermore, due to the systems' predicate flexibility, filters that eliminate most tuples are frequent, making integrations like post-filtering inefficient. Additionally, because their workloads are often write-heavy, VSS indices that are expensive to construct and maintain become impractical.

In this thesis, we aim to achieve efficient filtered vector similarity search in vector-extended relational OLAP DBMSs. We claim that these DBMSs should not only adjust the system to integrate the index but also redesign the index to fit that specific system. This expands the design space, enabling optimizations such as storage-model-aligned index partitioning and intra-query parallelism.

We propose PDXearch, a DuckDB extension tightly integrated with the system's internals. At its core is a pre-filtered IVF index, aligned with DuckDB's morsel-driven parallelized execution and storage model: PDXearch maintains one IVF index per storage block, that is, one per DuckDB row group. Searching these IVF indices in parallel yields intra-query parallelism, and filtered query execution is made efficient by skipping IVF indexes that have been filtered out (i.e., row group pruning). The cost of pre-filtering is partly offset by the OLAP DBMS's optimized filter evaluation. For IVF index performance that rivals graph-based indices, we build on the state-of-the-art Super K-Means and PDXearch libraries.

On an Apple M4 Pro MacBook, compared to the DuckDB VSS extension's USearch HNSW index, the pre-filtered IVF row group approach is approximately 33 to 56 times faster to construct and uses roughly 30 % to 50 % less memory. It achieves competitive (filtered) search performance and demonstrates its filtered search row group pruning mechanism on large datasets. Lastly, the design is promising for write-heavy workloads and for scaling to datasets that do not fit in memory. The extension is available at: <https://github.com/Noorts/PDXearch>.

Contents

Acronyms	iii
1 Introduction	1
1.1 Contributions	5
1.2 Research Questions	5
1.3 Outline	6
2 Background	7
2.1 Vector Similarity Search	7
2.1.1 Embeddings	7
2.1.2 Approximate vector search	9
2.2 Approximate Indices	9
2.2.1 Graph-based HNSW	9
2.2.2 Partition-based IVF	11
2.2.3 The Super K-Means and PDXearch libraries	13
2.2.4 Graph-based HNSW vs. partition-based IVF	14
2.3 Quantization	15
2.4 Filtered VSS	16
3 Systems	19
3.1 DuckDB	19
3.1.1 Internals	19
3.1.2 VSS	22
3.1.3 VSS extensions	22
3.2 PostgreSQL	23
4 PDXearch: Our Extension for VSS in DuckDB	24
4.1 Two Approaches	24
4.2 PDXearch Global	26
4.3 PDXearch Row Group	26
4.4 Summary	28
5 PDXearch Row Group Implementation	30
5.1 Software and Dependencies	30
5.2 Components	30
5.3 Important Consideration: Late Materialization	31
5.4 Index Creation	33
5.5 Non-Filtered Search	36
5.6 Filtered Search	39
5.7 Integrated Filtering	44

CONTENTS

5.8	Limitations of the Implementation	45
5.8.1	Filtered search complexity	46
5.8.2	Row id metadata	46
5.8.3	DuckDB planner and optimizer overhead	47
6	Evaluation	49
6.1	Experimental Setup	49
6.1.1	Hardware	49
6.1.2	Systems	49
6.1.3	Datasets	52
6.1.4	Index parameters	52
6.1.5	Metrics	54
6.1.6	Artifacts	55
6.1.7	QPS–recall plots	55
6.2	Index Overhead	57
6.2.1	Exp. 1 – In-memory size	57
6.2.2	Exp. 2 – Construction time	60
6.3	Non-Filtered Search	62
6.3.1	Exp. 3 – Normal interface	62
6.3.2	Exp. 4 – Blob interface	64
6.3.3	Exp. 5 – Varying the number of clusters	68
6.4	Filtered Search	70
6.4.1	Exp. 6 – Filtering spectrum	70
6.4.2	Exp. 7 – Row group pruning	72
7	Conclusion and Future Work	77
7.1	Answers to the Research Questions	78
7.2	Limitations of the Evaluation	80
7.2.1	Workloads	80
7.2.2	Data gathering	80
7.2.3	PDXearch	81
7.3	Future Work	82
	References	84
A	Dataset Origins	93

Acronyms

AMX	Apple Matrix Coprocessor	60
ANNS	approximate K-nearest neighbors search	9, 16, 18, 22, 23
CI	confidence interval	54, 56
CWI	Centrum Wiskunde & Informatica	13, 30
DBMS	database management system 2–5, 8, 15, 16, 18, 19, 21, 23, 24, 44, 47, 77, 78, 80	
ENNS	exact K-nearest neighbors search	9, 16, 22, 23, 30, 31, 33, 36, 41, 46, 70, 93
FANNS	filtered approximate K-nearest neighbors search	16
GEMM	general matrix multiply	13
HNSW	hierarchical navigable small world2, 4, 7, 9–11, 14–16, 23, 41, 53, 57–61, 67, 70, 72, 77, 94	
IVF	inverted file 2–5, 7, 9, 11–15, 23, 24, 26, 27, 30, 31, 34, 39, 41, 44, 46, 53, 56–59, 61, 67, 70, 72, 77, 79, 81, 82	
JSON	JavaScript Object Notation	55
KNNS	K-nearest neighbor search	1, 7, 8
LLM	large language model	1, 7
MVCC	multiversion concurrency control	3, 82
NSW	navigable small world	9
OLAP	online analytical processing	19, 21, 80
OLTP	online transaction processing	19
OOM	out of memory	73, 94
PCA	principal component analysis	94
PDX	Partition Dimensions Across	4, 13, 36, 82
PQ	product quantization	15
QPS	queries per second	28, 54, 55, 63, 64, 67, 69, 70, 72, 76, 77
RAG	retrieval augmented generation	1, 7
SIMD	single instruction multiple data	13, 45, 49, 50

ACRONYMS

SOTA	state-of-the-art	4, 5, 13, 24, 30, 82
SQ	scalar quantization	12, 15, 24, 36, 49, 59, 64, 68, 72, 73, 76, 79, 80
VDBMS	vector database management system	2, 15, 18, 54, 80
VSS	vector similarity search 1–3, 5, 7, 8, 13, 15, 16, 18, 22–24, 28, 45–47, 49, 52, 54, 55, 77, 82, 83	

1

Introduction

Vector similarity search (VSS) is a technique that, given a query object (e.g., an image), can find other objects (e.g., images) in a dataset that are similar to it [1, 2]. This makes it possible for a system to answer the query: “What are the five images most similar to the given image?” This similarity search is not restricted to images; it can involve any data, including text, video, and audio. VSS is used in many systems, including for recommendation, multimedia retrieval (e.g., reverse image search), voice recognition, missing value imputation, and large language models (LLMs) (e.g., retrieval augmented generation (RAG)) [1, 3, 4, 5].

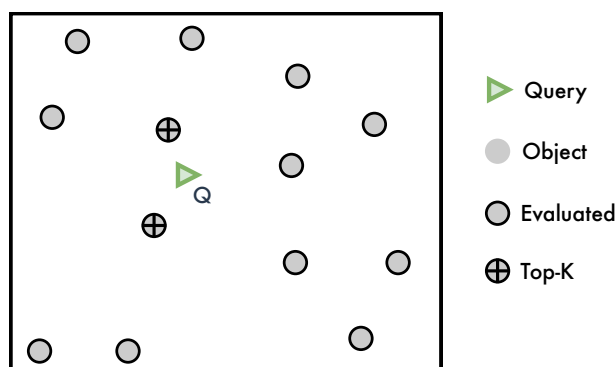


Figure 1.1: Simple 2-dimensional space, where each object (e.g., an image) in the dataset is represented by a point. The query object is also a point. The distance between any two points indicates the similarity of those two points. The smaller the distance, the higher the similarity. We assume the user requests the two most similar objects (i.e., $K = 2$), and thus the closest two neighbors are returned as the answer to the query.

The algorithm that powers vector similarity search is K-nearest neighbor search [2, 6, 7]. In short, all objects in the dataset (e.g., images) are represented as points in a high-dimensional space. Figure 1.1 demonstrates this using a simple 2-dimensional space. Besides the dataset objects, the query object is also a point in this space. To find the most similar objects, the algorithm computes the distances between the query object and each dataset object. The k objects closest to the query object (i.e., the neighbors) are most semantically similar and are thus returned as the answer. The number of neighbors to return (known as the top- k) depends on the workload; in benchmarks, k is often set to 10 [8, 9, 10, 11]. Chapter 2 discusses the details, including distance metrics, dataset sizes, and how to convert objects to points.

1. INTRODUCTION

Approximate VSS indexes make < 10 millisecond latency searches on datasets with billions of objects possible [4]. Without indexes, computing the distances between the query object and all the objects in the dataset, known as *exact search*, is computationally demanding, resulting in high search latency. Approximate VSS indexes reduce this computational cost by only computing distances to the most *promising* objects in the dataset. These indices return approximate answers; that is, they return neighbors that are not necessarily the closest to the query object [2, 6, 12]. Fortunately, approximate answers are sufficient for many workloads, and thus these approximate indexes are commonly used. The index data structure speeds up the search but is not free; it must be built, maintained as the underlying dataset changes, and takes up space. There are multiple types of indices, each with their strengths and weaknesses. Widely used index types are graph-based indices and partition-based indices [2, 3]). Graph-based indices (e.g., an hierarchical navigable small world (HNSW) index [13]) deliver high search performance but consume a large amount of memory and take long to construct and update. Partition-based indices (e.g., an inverted file (IVF) index) are smaller and faster to build and update but deliver slower search performance. We discuss these indices in Section 2.2.

VSS implementations consist of VSS algorithms and indices. These are packaged as search libraries (e.g., *PDXearch* [9], *USearch* [14], *Faiss* [15]). Vector database management systems (VDBMSs) are software systems that, in addition to standard database management system (DBMS) capabilities (e.g., data storage, query optimization, access control, transactions), also offer vector similarity search (VSS) [1, 2]. We distinguish between two types of VDBMSs: specialized *native* systems that are purpose-built for vector search, and *vector-extended* systems, where vector search capabilities have been added on top of an existing DBMS.

In the past six years, by integrating a VSS index, many DBMSs have become vector-extended systems [2]. Although these systems might provide less performance than native systems, many businesses already have a DBMS that can be extended [2, 16, 17, 18]. Hence, using a vector-extended DBMS avoids the operational toil of managing a separate native system. An example of a vector-extended DBMS is DuckDB paired with its VSS extension [19, 20].

The extension of DBMSs to support vector search introduced many common SQL operators into the vector search pipeline. Filtered vector search combines VSS and metadata filters (i.e., SQL *WHERE* predicates on structured data). This allows one to find k objects similar to a given object, which all pass the SQL predicate [2, 11, 21]. These are known as hybrid queries or as filtered queries. This common workload answers the more specific query: “What are five images similar to the given image, that were uploaded in 2026?”

In this thesis, we provide the community with a new way to build indexes and perform efficient filtered VSS in vector-extended relational analytical DBMSs. Our solution incorporates three key design decisions:

-
1. **Adapt not only the system, but also the index.** The VSS community has designed many indexes [2, 3, 4, 6, 11]. Vector-extended DBMSs integrate such an index, exposing VSS capabilities to the user. **Unfortunately, many of these systems are loosely coupled; that is, only the system is adapted to fit the index.** We believe that database engineers should also consider redesigning the index for that specific system’s characteristics. This tight coupling expands the design space, enabling new optimizations, such as using the DBMS’s storage layout to inform the index’s (IVF) partitioning, exploiting the DBMS’s parallelized execution model during index construction and index search, integrating with the DBMS’s memory facilities to make the index workload-aware (e.g., evicting unused partitions), and coordinating with the DBMS’s transaction support (e.g., multiversion concurrency control (MVCC)) to efficiently represent concurrent index versions.
 2. **Predicate flexibility makes filters that eliminate most tuples common; a pre-filtering integration is optimal for those.** Vector-extended relational analytical DBMSs allow the user to combine any predicate (WHERE clause) with their vector search (e.g., equality checks, range filters, joins). These predicate-agnostic queries sometimes result in few rows passing the filter. In this thesis, the performance of these filtered search queries is a priority. There are multiple ways to combine the predicate evaluation and vector search [8]. In the literature, we see many (hybrid) indexes that are specialized for a subset of filters [11]. These might achieve high search performance on specific workloads but are impractical for predicate-agnostic vector-extended DBMSs workloads (e.g., data science) [22].¹ Another approach, post-filtering — where the system first executes the vector search, which yields k items, and then filters those items — is optimal for queries where most tuples pass the filter [8]. Pre-filtering is the converse; the system filters first and then runs the vector search only for those items. It is optimal for queries where few tuples pass, but the predicate evaluation becomes costly on large datasets. Fortunately, vector-extended analytical DBMSs have fast predicate evaluation, which partially mitigates the cost of this pre-filtering. Hence, these vector-extended analytical systems should consider a pre-filtering integration. We discuss pre-filtering, post-filtering, etc. in detail in Section 2.4.
 3. **Expensive to build and update (e.g., graph-based) indices are impractical for vector-extended DBMSs. We opt for a lightweight partition-based IVF index.** In practice, many vector systems continuously receive new data (e.g., uploads on YouTube, new products on e-commerce sites) [23]. The VSS index is incrementally updated (i.e., maintained) to incorporate this new data and to address data distribution drift. Alter-

¹As discussed in Section 2.4, there are also indexes built around a single attribute that can use in-filtering to evaluate arbitrary predicates. Unfortunately, these indexes are also suboptimal for queries where few tuples pass [8].

1. INTRODUCTION

natively, the index is periodically rebuilt from scratch. Indexes such as the graph-based HNSW index provide fast search but are expensive to construct (taking hours to days) and maintain [4, 24, 25]. This makes them impractical for large datasets with frequent updates [9, 15, 26]. We think these data management aspects are important, even if it comes at the cost of search performance. A lightweight IVF index is more suitable for large datasets with continuous ingest. Hence, we use a lightweight partition-based IVF index.¹ Finally, the fast index construction also makes the IVF index appealing for interactive data scientist workloads.

Taken together, this thesis’ main claim is that vector-extended relational analytical DBMSs 1) should not only adjust the system to integrate the index but also design the index to fit the system and to integrate it tightly, and 2) should consider integrating a lightweight predicate-agnostic IVF index through pre-filtering.

To support our claim, we contribute the new **PDXearch DuckDB extension**, which uses a state-of-the-art (SOTA) IVF index integrated through pre-filtering. We propose a **row group approach**, which integrates with DuckDB tightly. Our extension incorporates the aforementioned three key design decisions and uses lossy compression (quantization). In short, the implementation:

1. Integrates a lightweight SOTA IVF provided by the PDXearch library [27].² This IVF index provides search performance competitive with top-of-the-line HNSW performance, while being significantly faster to construct and update, and lower in in-memory usage. The PDXearch library is built around Partition Dimensions Across (PDX) [9], a data layout that transposes vectors in a block-column-major order, enabling efficient pruning of dimensions thanks to optimized data access patterns. To cluster the objects into IVF partitions, we use the SOTA K-means clustering library called Super K-Means [28, 29].
2. Uses a pre-filter approach and exploits the vector-extended analytical DBMS’s optimized predicate evaluation for fast and predicate-agnostic filtered vector search.
3. Adjusts the index to fit the system. Our row group approach tightly integrates with the storage and parallelism model of DuckDB by using a separate **micro IVF index per storage block**.
4. Uses lossy compression (quantization) on the dataset objects to reduce memory use and improve search performance at the cost of compression and a slight loss of result quality.

¹In the first point we said: “adapt not only the system, but also the index”. The IVF index is easy to adapt and highly suitable for tight integration. Other lightweight indexes might also be suitable but are out of scope.

²Both the extension and library are called PDXearch. We use the term PDXearch to refer to the extension.

In the evaluation, we use medium-scale filtered vector similarity search workloads that are common in VSS benchmarking and that can be run on a modern laptop. We compare the performance of our new DuckDB PDSearch extension to core DuckDB, the DuckDB VSS extension, and the popular PostgreSQL pgvector extension.

1.1 Contributions

Conceptual

- We contribute a design optimized for relational analytical DBMSs's predicate-agnostic write-heavy workloads (Chapter 4). The design **introduces a storage block and parallelism model-aligned IVF index**, called the *row group* approach.
- We analyze the benefits and drawbacks of the design (Chapter 4).
- We evaluate our implementation against the DuckDB core, the DuckDB VSS extension, and the popular PostgreSQL pgvector extension (Chapter 6). The results indicate that the row group approach has superior data management characteristics (index construction and index size) compared to DuckDB VSS' HNSW index and that it offers competitive search performance. The experimental setup, benchmarking scripts, measurements, and plots are open source (<https://github.com/noorts/vss-benchmarks>).

Open-source artifacts

- We contribute the open-source SOTA IVF-based DuckDB PDSearch extension (<https://github.com/noorts/pdsearch>). This aims to offer the database community a lightweight but fast (filtered) VSS solution.
- We extend the VectorDBBench benchmarking framework with new DuckDB clients and new datasets, and improve the framework's user experience and result output (<https://github.com/noorts/vectordbbench>).

1.2 Research Questions

To guide our work, we rely on the following research questions.

- (Q1) How can we design a tightly integrated vector search architecture that provides efficient filtered search within a relational analytical vector-extended DBMS?
- (Q1a) How can we leverage the parallelization model of the DBMS to improve vector search performance?
- (Q1b) How should we handle queries with predicates that eliminate almost all tuples?

1. INTRODUCTION

(Q2) What does the design offer?

(Q2a) How do the features and performance of the baseline global approach compare to the other systems?

(Q2b) How do the features and performance of the tightly integrated row group approach compare to the other systems?

(Q2c) How does PDXearch's scalar quantization affect the performance?

1.3 Outline

The next chapters, Chapter 2 and Chapter 3, describe the required background information. We discuss the design of the extension and its implications in Chapter 4, and describe the current implementation in Chapter 5. Chapter 6 covers the evaluation of the extension, compares it to core DuckDB, DuckDB VSS, and PostgreSQL pgvector through seven experiments, and analyzes the results. We conclude the thesis with the conclusion, and a discussion of the limitations and future work in Chapter 7.

2

Background

This section discusses the required vector similarity search (VSS) background knowledge, including the IVF and HNSW index types, and the filtered VSS integration approaches. Chapter 3 will introduce the systems that are part of the evaluation.

2.1 Vector Similarity Search

2.1.1 Embeddings

Vector similarity search (VSS) is used in many systems, including for recommendation, multi-media retrieval (e.g., reverse image search), voice recognition, missing value imputation, and LLMs (e.g., retrieval augmented generation (RAG)) [1, 3, 4, 5]. The technique behind this is K-nearest neighbor search (KNN).

KNN operates on vectors, also known as embeddings, which are fixed-length D -dimensional vector/array representations (e.g., $[0.11, 0.21, \dots]$) of the original data (e.g., images) [2, 6]. These embeddings can be visualized as points in a D -dimensional space. Proximity within this space denotes semantic similarity; the shorter the distance, the higher the semantic similarity. Proximity is defined through a mathematical similarity metric (also known as a distance function, e.g., Euclidean distance, cosine similarity, and inner product).

Figure 2.1 shows the embedding/vectorization workflow. Given a dataset with objects (e.g., images), a pre-processing step transforms these objects into *vector embeddings* [1].

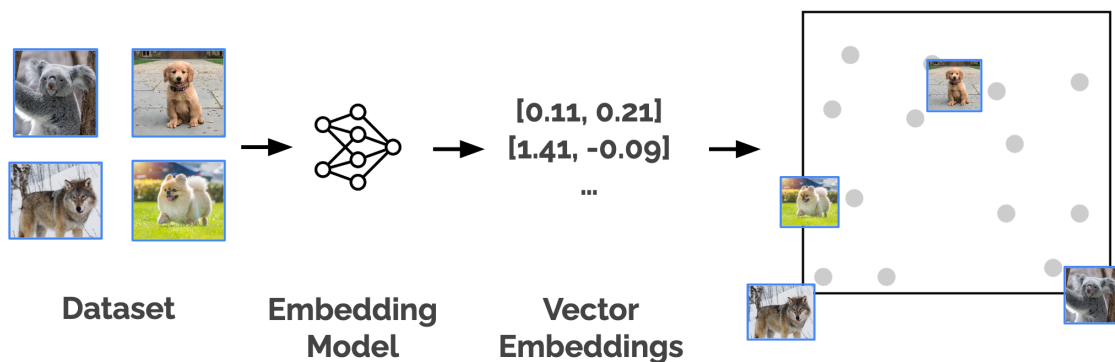


Figure 2.1: Overview of the VSS embedding/vectorization workflow.

2. BACKGROUND

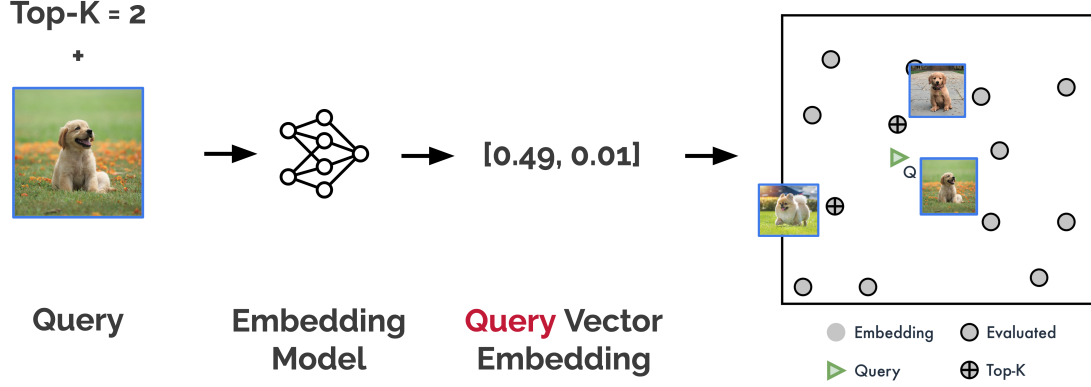


Figure 2.2: Overview of a vector similarity search (VSS).

This transformation is also known as *vectorization*.¹ These vector embeddings are stored in a system (e.g., in a LIST, ARRAY, or BLOB column in a DBMS). An identifier is often stored alongside each vector embedding to refer to the original object [1].

As an example, assume that we have a collection of images of animals that we want to perform a vector similarity search on. We transform these images into embeddings using a function, which is often a machine learning model. For our example, this embedding model has been trained such that all images of dogs are mapped to points close to each other in the D -dimensional space. The use case determines the embedding model to use. Pretrained embedding models for a range of modalities are available to download online. Examples include word2vec (text; [31]) and CLIP (text and images; [32]). The key idea is that the embedding model and distance function combined capture the semantic similarity that the user is interested in. Transforming the original data into embeddings typically occurs at the application layer, outside the vector system [1].

Now that we have the embeddings of our animal images, we can perform a vector similarity search, as visualized in Figure 2.2. Given a query image, we use the same embedding model to turn the image into a query embedding. Next, given the query embedding, KNN computes the distance between the query embedding and each of the embeddings in the dataset. KNN returns the K neighboring embeddings that are closest to the query vector [2]. This is known as the top- k . The retrieved vectors are then linked back to their original form (e.g., images) using their identifiers, and those objects are returned to the user.

The number of elements in the vector embedding array is referred to as the dimension of the vector. The number of dimensions commonly ranges from 128 to 4096 dimensions [6, 8]. Each element in the array is usually stored as a 32-bit float type. Therefore, an embedding

¹The terms *vectorization* and *vectorized* in a VSS context are not to be confused with the DBMS concept of *vectorized query execution* [30] nor DuckDB’s internal *Vector* concept [19]. To avoid confusion, we will mainly use the term *embedding* (instead of *vector*) to refer to the array representation of an object.

with 2048 dimensions takes up 8 KiB. The high dimensionality combined with the size of the dataset (e.g., one billion objects) results in a computationally demanding exact vector similarity search (also known as exact K-nearest neighbors search (ENNS)).

2.1.2 Approximate vector search

In many use cases, it is acceptable if an approximate K-nearest neighbors search (ANNS) is performed, where the search might miss a few of the true neighbors and return neighbors that are slightly less similar (i.e., further away) to the query [2]. Relaxing the requirement for exact results makes it feasible to process datasets with billions of objects within a 10-millisecond latency bound [4]. Recall@k is a commonly used performance metric that quantifies the quality of the top- k result [8, 9, 33].

We define recall@k as:

$$\text{recall@k} = \frac{|\text{top-}k^{\text{alg}} \cap \text{top-}k^{\text{ground truth}}|}{k}$$

Where $\text{top-}k^{\text{alg}}$ represents the embeddings returned by the algorithm (i.e., index) and $\text{top-}k^{\text{ground truth}}$ are the true k closest neighbors (i.e., the result of ENNS). k depends on the use case but is commonly set to 10 [8, 9, 10, 11]. Recall ranges from 0%–100%. The recall percentage to aim for depends heavily on the use case [9]. Many benchmarking papers focus on the upper 70%–100% range [9, 11, 33]. Next, we discuss two techniques that make the similarity search approximate: approximate indices (Section 2.2) and quantization (Section 2.3).

2.2 Approximate Indices

In this section, we discuss the HNSW index and IVF index in detail.

2.2.1 Graph-based HNSW

A hierarchical navigable small world (HNSW) index is a graph-based index [4, 13]. It combines key concepts from the original navigable small world (NSW) index [34, 35] and the probabilistic skip list structure [36].

The HNSW index has inspired numerous other graph-based indexes [4]. The index stores the embeddings as vertices in a proximity graph structure, as shown in Figure 2.3. Edges connect vertices that are similar; that is, based on the distance metric, they are close to each other in the D -dimensional space. The graph has multiple layers. A probability function determines what the highest layer is that an embedding appears in. The embedding will exist in that layer and all layers below it. Hence, it creates a hierarchy; the number of nodes increases as one traverses down the layers.

2. BACKGROUND

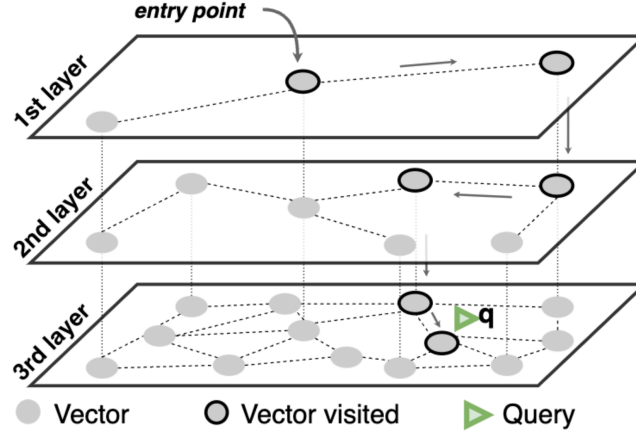


Figure 2.3: Hierarchical navigable small world (HNSW) index with three layers. The search procedure starts at the top layer and greedily finds the node (i.e., vector/embedding) closest to the query embedding on that layer. Then it repeats the procedure starting from that node on the layer below. The K closest neighbors found on the bottom layer are the query result.

A HNSW index search works as follows [13, pp. 4–5]. We start at some entry point on the top layer and greedily traverse that layer to find the node with the shortest distance to the query embedding. We then use that node as the entry point on the next layer and repeat the greedy search until we find the nearest neighbor on the bottom layer. During the traversal of the bottom layer, we keep a larger candidate list with the (at most `ef_search`) nearest neighbors found closest to the query embedding. Once the search has completed, we return the K closest neighbors from that candidate list. This hierarchical greedy index search is usually single-threaded and has a logarithmic search complexity.

The HNSW index is built incrementally. Given an embedding and a (partially built) HNSW index, we first use the probability function¹ to determine this embedding’s (topmost) insertion layer. Then we use the aforementioned search procedure to greedily traverse the layers to find the closest neighbor up until the insertion layer. From this layer onward, on each layer during traversal, we maintain a larger candidate list, this time with the `ef_construction` closest found nearest neighbors. When the greedy search on a layer completes, we create the embedding on this layer by adding the node and adding bidirectional edges to the M^2 closest³ neighbors in the candidate list. The algorithm ensures no node exceeds M connections, shrinking connections as necessary. The construction continues on the layer below it, using the previous layer’s candidate list as the greedy search’s starting point. This repeats, layer by

¹There is a related m_L parameter, which normalizes the probability using a level multiplier. This parameter is not exposed to the user in pgvector nor in DuckDB VSS, so we do not mention it further.

²Malkov et al. refer to M as M_{max} . There is also the separate $M0$ (M_{max0}), which configures the connectivity (M) of the bottom-most layer. DuckDB VSS sets $M0$ to $2 \cdot M$ by default [37].

³Alternatively, Malkov et al. [13] suggest using a heuristic to create connections to nodes in diverse directions instead of simply the closest. They show that this benefits search performance.

layer, until the embedding has been inserted in the bottom-most layer. This incremental index construction can be accelerated by inserting embeddings in parallel [13].

The construction parameters `ef_construction` (size of the candidate list during construction) and M (number of edges per node) affect the index construction time, index size, search latency, and recall. They are commonly set to 64–128 and 16, respectively [37, 38]. The `ef_search` parameter (size of the candidate list during search) determines the search latency vs. recall tradeoff at runtime. It is commonly set in the range of 40–64.

HNSW indexes support maintenance (i.e., inserts, updates, and deletes). Inserts can be handled incrementally through the aforementioned index construction process. Unfortunately, deleting an embedding is expensive in many HNSW implementations. The index either has to be completely rebuilt, or the index requires a periodic maintenance or vacuum step to clean up embeddings that are marked for deletion (i.e., tombstones) [14, 37, 38]. This resolves the performance degradation induced by the tombstone nodes and addresses data distribution drift (e.g., a stale top-level entrypoint).

2.2.2 Partition-based IVF

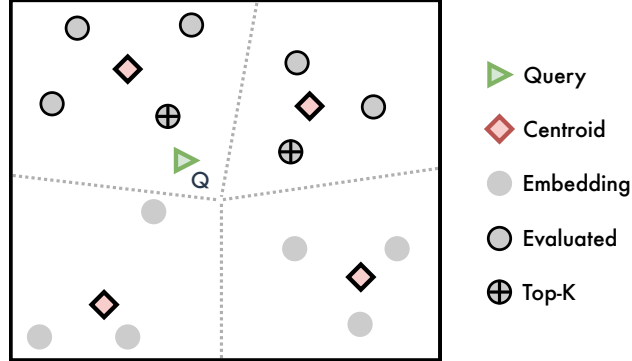


Figure 2.4: Inverted file (IVF) index with 13 embeddings partitioned into 4 clusters. Each cluster has a representative cluster centroid. During an index search we first compute the distance from the query to the centroids. This determines which partitions are closest. Then we probe the closest two partitions, which means we compute the distance from the query to the objects inside the probed partitions. Assuming $k = 2$, we return the two evaluated objects closest to the query as the query answer.

An inverted file (IVF) index partitions/clusters the D -dimensional embedding space into L partitions/clusters [9, 15, 23, 39, 40, 41]. Figure 2.4 shows a simple example of a 2-dimensional space with 4 clusters. During index construction, given a number of lists L (e.g., $L = 4$), the first step is to run a K-means clustering algorithm (e.g., the Lloyd-Max algorithm [42, 43]). This will process all embeddings and create L clusters. Each partition is represented by a centroid, which, similar to the dataset embeddings, is a point in the D -dimensional space. After determining the partitions and their centroids through clustering, the next step is to

2. BACKGROUND

assign each dataset embedding to its closest partition. The closest partition is determined by computing the distance from the embedding to each of the centroids. These embedding-partition assignments are stored as inverted lists; that is, for each partition we store a list with the identifiers of the embeddings that belong to it.

An IVF index stores only two pieces of metadata: 1) the centroids of the partitions (so a D -dimensional array for each partition), and 2) the inverted lists storing the mapping from partition ID to the identifiers of the embeddings that were assigned to that partition.

Many steps of IVF index construction process can be parallelized. This includes computing the distances during clustering, assigning embeddings to clusters, and quantizing (e.g., scalar quantization (SQ)) the embeddings.

Using the IVF index to search for a query embedding's top- k neighbors works as follows [9, 15]. First, we compute the distance between the query embedding and all the centroids. This allows us to prioritize the partitions. Next, given several partitions to probe (i.e., $n_probe = 2$), for the closest n_probe partitions we compute the distance between the query embedding and the embeddings in the partitions we probed. The inverted lists state the embeddings that belong to those partitions. These embeddings are prioritized by their distance descendingly. Some systems (e.g., pgvector [38]) use a sorted list for this, while others (e.g., PDXearch [27]) use a max heap data structure that stores only the top- k closest items. From the probed partitions, the k embeddings closest to the query embedding are returned as the answer.

The key idea behind the IVF index is that it avoids computing the distance to all the embeddings by 1) using centroids as representatives of the embeddings and 2) only probing a subset of the partitions. This allows it to significantly decrease the number of distance calculations, leading to reduced latency. However, as shown in the example in Figure 2.4, the result is approximate, as it misses the neighbor in the partition that was not probed.

IVF index search is usually single-threaded, but it can be parallelized.¹ Namely, the distance calculations workload can be spread across multiple threads. For example, the system could probe in parallel, using one thread per partition, and then merge the results into a shared sorted list or heap.

The number of lists/partitions L (construction parameter) and number of lists to probe (runtime parameter; n_probe) affect the index construction time, index size, search latency, and recall. The number of lists L is commonly set to $\sqrt{N}$, where N is the number of objects in the dataset [38]. As the experiments (Chapter 6) will show, setting n_probe to probe only 5 % of the partitions is often sufficient to reach 90 % recall.

IVF indexes support maintenance (i.e., inserts, updates, and deletes). After index construction, the embeddings are not necessarily spread out in equal numbers across the partitions. Some partitions will likely be assigned slightly more embeddings than others, as shown in Fig-

¹Most systems offer inter-query parallelism to saturate their cores. Note that our baseline PDXearch IVF approach is single-threaded, and the row group approach is multi-threaded.

ure 2.4. To add a new embedding to the index, the system computes the distance between the embedding and all the centroids, and then assigns the new embedding to the closest centroid's partition. In the worst case, many new embeddings are all assigned to the same partition, leading to a state where the clusters no longer represent the embeddings' distribution (i.e., data distribution drift). If left alone, the index will become unbalanced and search performance will degrade [41]. To address this drift, instead of rebuilding the index from scratch, recent IVF implementations such as *Ada-IVF* [44], *Quake* [41], and *SPFresh* [23] support incremental updates that manage the drift automatically.

2.2.3 The Super K-Means and PDXearch libraries

Our PDXearch extension relies on two IVF related libraries: *Super K-Means* and *PDXearch*.

PDX is a column-major data layout for vector embeddings developed at Centrum Wiskunde & Informatica (CWI) [9]. The layout supports dimension pruning, which means that not all dimensions (e.g., only 256 out of 1024 dimensions) are evaluated during distance calculations. This can significantly speed up distance calculation throughput.

Super K-Means is a SOTA K-means clustering library from CWI [28, 29]. It uses the PDX layout and carefully interleaved general matrix multiply (GEMM) routines and dimension pruning kernels to achieve fast K-means clustering on CPUs.

CWI's **PDXearch**¹ is a VSS library also built around the PDX data layout [9, 27]. PDXearch consists of IVF index data structures (e.g., centroids, inverted lists) and (filtered) search algorithms. These algorithms exploit the data layout, dimension pruning, and use single instruction multiple data (SIMD) instructions to provide fast partition-based vector search on CPUs.

Setting up an IVF index in the PDXearch library requires a dataset of embeddings (and their identifiers) and the partitioning metadata (i.e., centroids and assignments) from Super K-Means. Given a query embedding and k , one can initialize and run a vector search. This will determine the partition probe order/priority, probe the first n_probe partitions, and finally return distance-identifier pairs of the top- k embeddings.

PDXearch filtered search is similar. During filtered search initialization, the library creates a selection vector (i.e., bitmask) that indicates which embeddings should be filtered out. If all embeddings in a partition are filtered out, then as an optimization, that partition is skipped entirely during the search.

¹The PDXearch library [27] is not to be confused with the PDXearch extension [45]. The PDXearch extension uses the PDXearch library internally. We use the term *PDXearch* to refer to the extension.

2. BACKGROUND

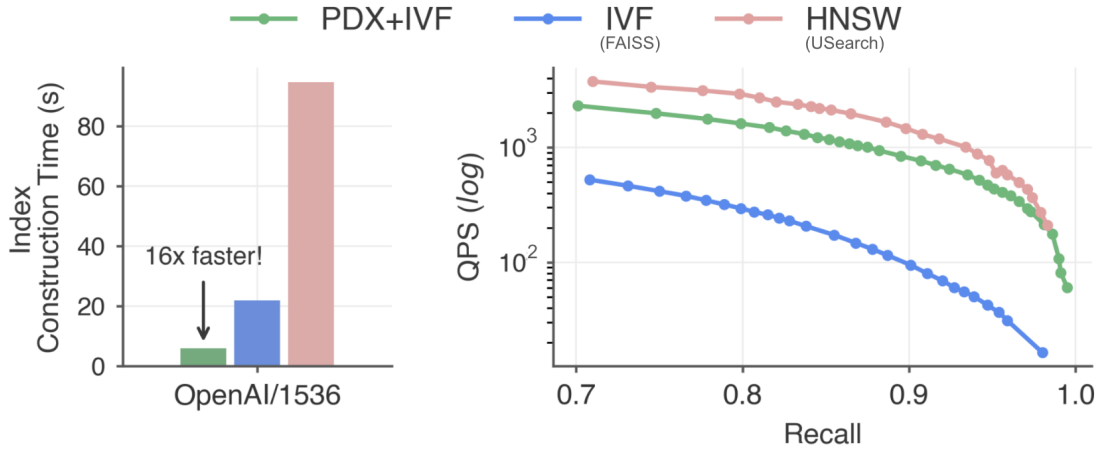


Figure 2.5: Performance of the *Super K-Means* and *PDXearch* libraries compared to Faiss’ IVF index and USearch’s HNSW index. Adapted from Kuffo [46]. Section 6.1.7 explains how to interpret the QPS–recall curve plot shown on the right.

2.2.4 Graph-based HNSW vs. partition-based IVF

Traditionally, graph-based indexes (e.g., HNSW) have been the de-facto standard due to their dominant search performance [4, 24, 25]. However, compared to partition-based indexes (e.g., IVF), graph-based indexes take longer to construct, longer to update (i.e., maintenance), and have higher memory usage [26, 41, 47]. Due to these long construction times and memory requirements it is infeasible for graph-based indexes to scale to larger datasets on commodity hardware [9, 15, 26]. Fortunately, as shown on the right in Figure 2.5, recent IVF indexes, such as the one offered by the *PDXearch* library, make IVF index search competitive to graph-based HNSW indices, while remaining lightweight [9, 11]. Furthermore, the adaptable partition-based nature of the IVF index type make them more suitable for tight integration into data management architectures [26, 39, 44, 47]. Table 2.1 summarizes these characteristics.

Table 2.1: Characteristics of graph-based and partition-based indexes [7, 12, 33].

	Graph-based	Partition-based
Search throughput	Highest.	High (competitive when using <i>PDXearch</i> IVF; Figure 2.5).
Index construction time	High.	At least 10x faster.
Index size (memory usage)	Large (graph nodes and edges).	Small (centroids and inverted lists).
Inserts and deletes (maintenance)	Expensive (the index is usually rebuilt from scratch).	Cheaper (the index is usually updated incrementally) [23, 41].

2.3 Quantization

In addition to the approximate HNSW and IVF indexes, we can also use quantization to make the vector search faster and more manageable in terms of memory footprint.

Quantization is a lossy type of compression that transforms embeddings composed of floating-point numbers into lower-precision embeddings composed of integers. There are many quantization techniques, including the widely used product quantization (PQ) [48] and scalar quantization (SQ). We focus on scalar quantization, which is used by the PDXearch IVF index.

Scalar quantization (SQ) maps each floating-point embedding element to an integer [48, 49]. PDXearch maps from a 32-bit floating-point value to an 8-bit integer, reaching a compression ratio of 4.¹ The 8-bit integer represents 256 equally sized buckets. To quantize, first we find the minimum and maximum floating-point values across all dimensions and dataset embeddings², and determine the base (minimum value), range (max - minimum), and bucket/step size (range / 256). Second, we use the base and step size to map the floating-point values of all dimensions and embeddings into the buckets.

Quantized embeddings use less space, which increases efficiency (CPU caches, RAM use, SIMD throughput), leading to faster index construction and search [2, 15]. The gains of quantization are not free. It introduces computational overhead, as the system has to quantize the embeddings. The dataset embeddings are quantized once during index construction, and the query embedding is quantized once during index search. Furthermore, quantization leads to a loss of information in the embeddings, resulting in less accurate distances, and hence lower recall. Fortunately, as we will see in Chapter 6, quantization significantly improves search throughput, with only a slight loss of recall. For many VSS use cases, this tradeoff is desirable; therefore, quantization is common in VDBMSs [2, 3, 21]. Lastly, it is common for the VSS index to store the quantized embeddings, with the original full precision embeddings available (as backup) in the vector-extended DBMS's table storage.

Approximate indexes are frequently combined with quantization techniques. In this thesis, we will see IVFFlat (IVF with non-quantized full precision embeddings) and IVF U8 (also known as SQ8: scalar quantization to 8-bit unsigned integers).

¹The PDXearch library's quantization logic can be found in: <https://github.com/Noorts/PDXearch/blob/cbed8a5b35dd0ee8e6697ade0c92679b66ee8d40/src/include/pdxearch/quantizers/scalar.hpp#L43>.

²The PDXearch library uses the simple approach discussed here, with a single global base and range. A different SQ approach is to use D bases and ranges, one per dimension [49].

2. BACKGROUND

2.4 Filtered VSS

Filtered vector similarity search is also known as filtered approximate K-nearest neighbors search (FANNS) and filtered queries are known as *hybrid queries* [2, 11, 21].

There are multiple ways to integrate the vector search and predicate evaluation. Systems can filter-then-search (pre-filtering), search-then-filter (post-filtering), or filter-during-search (in-filtering). There are also hybrid indexes, which combine embeddings and attribute(s) inside the index. We discuss each approach in more detail.

Filter-then-search (pre-filtering) As shown in Figure 2.6a, use a DBMS or traditional attribute-column index (e.g., B-tree) to efficiently evaluate the filter, then run ANNS (with a VSS index and/or quantization) or exact K-nearest neighbors search (ENNS) on the remaining items. Some implementations [27, 50] use a bitmask to pass the filter result to the ANNS.

When few rows pass the filter, pre-filtering solutions perform best. In that scenario, it is necessary to evaluate the filter on all (or most) rows to find the passing rows. Pre-filtering makes this efficient by evaluating filters in bulk and then executing the top- k search on only the relevant items. This is not the best approach when most rows pass the filter, as the top- k search will be the one to reject the majority of rows. In this case, evaluating the filter on all rows is redundant, resulting in computational overhead.

Search-then-filter (post-filtering) As shown in Figure 2.6b, run ANNS to compute the top- k , and then filter out the items. Unfortunately, this is fragile, as it is likely to end up with $< K$ (e.g., 0) items. There are two common approaches: either set k higher to increase the probability that k items are left, or implement iterative fetches. That is, run ANNS, filter, and then if there are $< k$ results left, fetch more items in the ANNS, filter, etc..

Post-filtering is suitable when most rows pass the filter. As the number of rows that pass decreases, post-filtering's performance degrades because achieving high recall requires additional strategies such as performing backtracking on the index search in hopes of finding enough points that match the predicate.

Filter-during-search (in-filtering) In-filtering approaches search the VSS index and evaluate the filter as they traverse the index. Common in-filtering examples are graph-based (modified HNSW) indexes, where nodes are filtered out as the graph is explored [51, 52].

Similar to post-filtering, this performs well on queries where most rows pass. As the number of rows that pass decreases, in-filter solutions will become less efficient or start to return low recall results (e.g., in some approaches the graph will become disconnected [51]) [8].

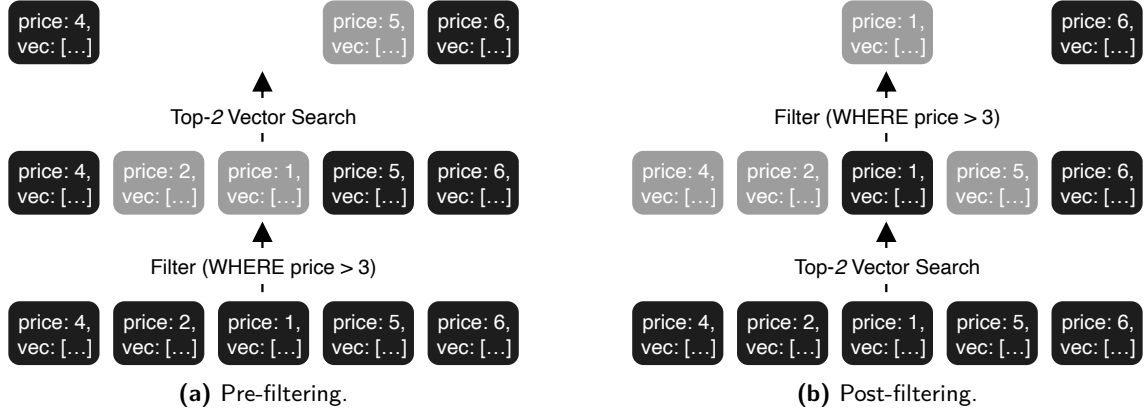


Figure 2.6: Pre-filtering and post-filtering. Adapted from Iff et al. [8].

Hybrid index During index construction, a hybrid index will use both (a subset of) attribute values and the embeddings themselves to build the index. For example, one could build an index for each combination of attribute values and then, during a search, use the graph that matches the filter [53]. Another hybrid index approach is to mathematically incorporate the attributes into the D -dimensional space [54]. Some hybrid indexes [55, 56] include a subset of the attributes (e.g., 1 attribute) in the index at construction time, and propose using in-filtering or post-filtering to evaluate the other attributes.

Iff et al. [8] recommend hybrid indexes for queries where $\sim 50\%$ of the rows pass the filter. In addition to this, we think these specialized hybrid indexes are most suitable when optimizing for a specific workload (e.g., a range filter on an integer attribute). As for this thesis' predicate agnostic workloads, there are two downsides: first, in-filtering or post-filtering enhancements might allow a hybrid index to cover all workloads, but it is likely less efficient than pre-filtering. Second, normal indexes require maintenance when the embeddings change. In addition, these hybrid indexes might also require maintenance when metadata attributes change.

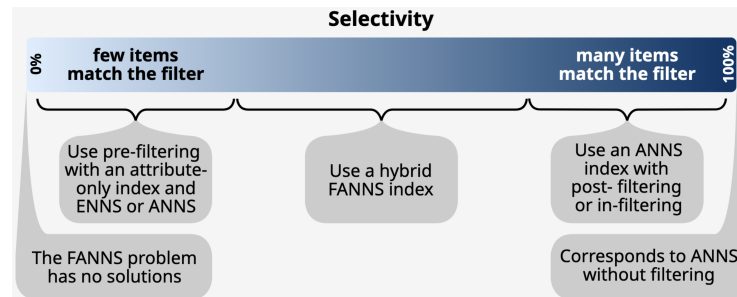


Figure 2.7: Filtering percentage spectrum. The optimal integration depends on the percentage of rows the filter eliminates in a given dataset. Borrowed from Iff et al. [8].

2. BACKGROUND

According to Iff et al. [8] and as depicted in Figure 2.7, there is no optimal approach for all workloads. The approach that is most suitable depends on the percentage of tuples eliminated by the query and the query type (e.g., equality filter, range filter). That is why some systems implement multiple of these approaches. For example, the *Milvus* VDBMS implements both pre-filtering and (iterative) post-filtering and allows the user to choose which to use at runtime [57]. Another example is *AnalyticDB-V*, a system that automatically chooses the best approach for a query using a cost model [58].

Competitors The systems we benchmark against, DuckDB VSS (Section 3.1), pgvector IVF, and pgvector HNSW (Section 3.2), all use post-filtering. As shown in Figure 2.7 — this positions them to perform best on queries where most rows pass the filter. pgvector implements iterative post-filtering, “ensuring” that it returns k items if there are k items in the dataset that pass the filter. As of the writing of this thesis, DuckDB VSS has not implemented an iteration mechanism. Unfortunately, for queries with filters where few tuples pass, this means it runs an ANNS, filters the top- k , and then often returns 0 results as all items were filtered out. This limitation makes the DuckDB VSS extension impractical to use for filtered vector similarity search. It is possible to work around this, but we have used the DuckDB VSS extension as is.¹ Ideally, the DuckDB VSS extension will address this problem in the future.

Our design In Chapter 1 we reasoned that vector-extended DBMSs should consider a pre-filtering approach. We argued that queries that eliminate most tuples are common and that we can exploit the vectorized query execution engine to quickly evaluate the predicates. For these reasons, we use an approach similar to pre-filtering for PDSearch’s filtered VSS query execution. Section 5.7 provides more details on the integration. For simplicity we refer to the integration as “pre-filtering” throughout the thesis.

¹As noted above, the user could request k' results (where $k' > k$) to increase the likelihood that k items are left, and then add a DBMS LIMIT k operator on top. Or DuckDB VSS could implement iterative post-filtering, similar to pgvector (Section 3.2).

3

Systems

This section introduces the systems that are part of the evaluation. Namely, DuckDB with the VSS extension, and PostgreSQL with the pgvector extension.

3.1 DuckDB

DuckDB is an open-source in-process relational DBMS developed by DuckDB Labs [59]. It has become popular in the community [60], with more than 40 million downloads per month [61, 2:02]. DuckDB was designed with both data science and data engineering workloads in mind [62]. The system’s columnar-vectorized execution engine is optimized for complex analytical workloads on large datasets. These analytical workloads are known as online analytical processing (OLAP). On the other side of the spectrum are DBMSs (e.g., TigerBeetle [63] [61, 25:13]) that are specialized for online transaction processing (OLTP) workloads [64, 65]. These OLTP systems optimize for low latency processing of a high volume of (concurrent) simple queries.

3.1.1 Internals

In this section, we introduce DuckDB concepts that are relevant to the PDXearch extension’s design and implementation. Note that we are using DuckDB v1.4.4.

Figure 3.1 depicts the query execution of a fictional query plan¹ that finds the name and age of the top 10 oldest people that are 30 years or younger: `SELECT name, age FROM table WHERE age <= 30 ORDER BY age LIMIT 10;`. The Filter and Top-N operators are a simplified version of DuckDB’s implementation. In the query plan, the sequential scan fetches rows, the filter operator evaluates the condition, and then pushes the remaining rows into the Top-N operator. The Top-N collects the rows, storing them in a Top-N heap (where $N = 10$ in our case). Finally, the result is emitted to the user. In the following paragraphs, we use this diagram to explain important aspects of DuckDB.

¹For this simple predicate, DuckDB would push the filter down into the sequential scan. We use a separate filter operator on top for illustration.

3. SYSTEMS

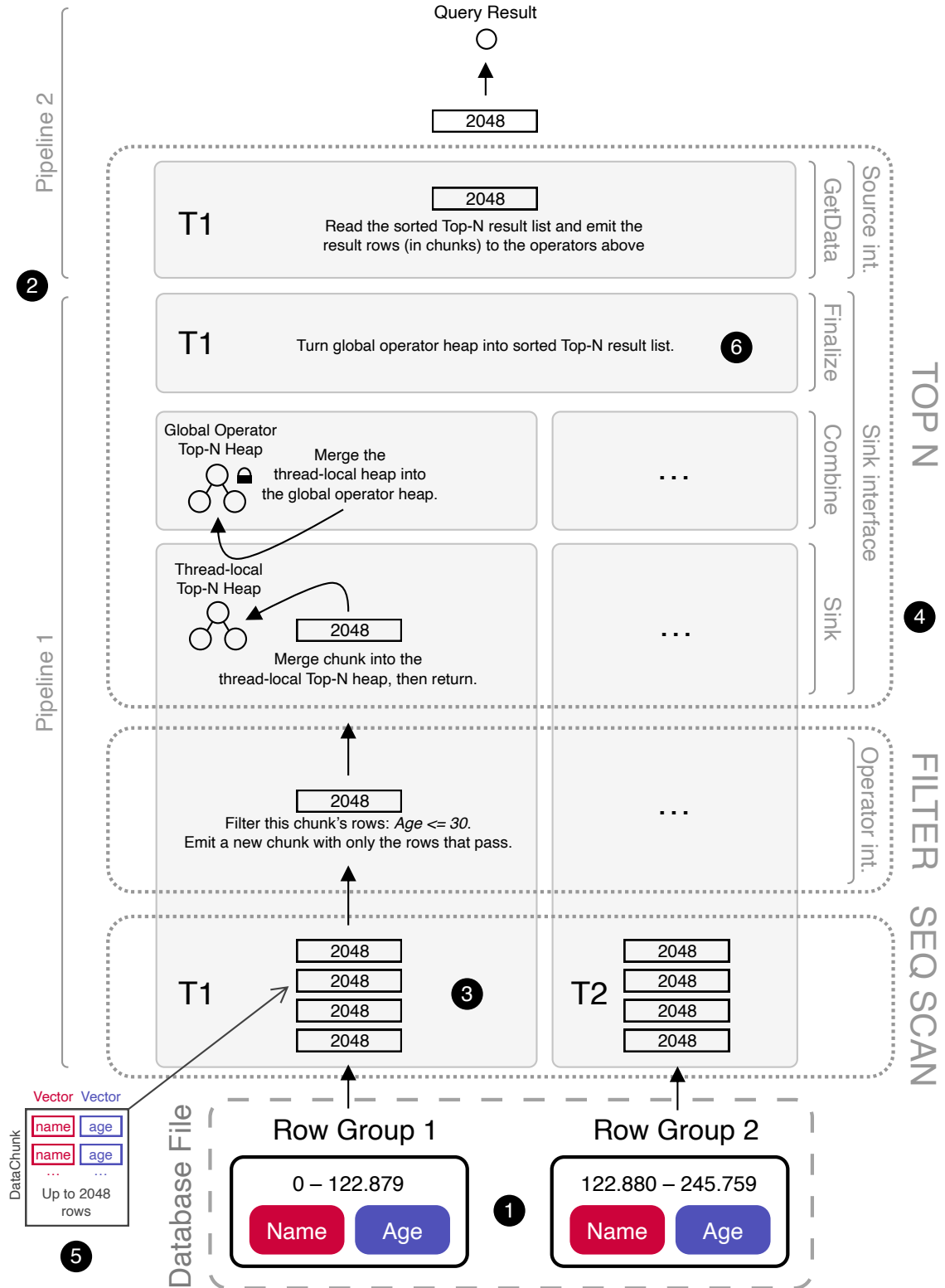


Figure 3.1: DuckDB execution of an example query: `SELECT name, age FROM table WHERE age <= 30 ORDER BY age LIMIT 10;`. The query plan and implementation are simplified for educational purposes. The example dataset contains two full row groups. T1 and T2 indicate the threads that execute the steps.

As shown at ❶¹ in Figure 3.1, DuckDB stores all data (e.g., tables, indexes) in a single database file [19, 66]. The table rows are split into groups of at most 122 880 rows each². We refer to these groups as **row groups**. They do not have to be full; insertion behavior and DuckDB’s compaction determine the size of the row groups. DuckDB uses columnar storage, which entails that, inside a row group, all column values are stored contiguously. In our example, the data would be stored as ["Bob", "Alice", ...] and then [30, 79, ...]. Columnar storage is common in OLAP DBMSs as it makes aggregate data access efficient [30, 67, 68]. All rows have a unique row identifier, stored in the virtual `rowid` column.

A query plan is split into one or more pipelines [69]. An operator such as the Top-N operator is a so-called **pipeline breaker**. The query execution must finish all work in the pipeline’s operators below it before the pipeline breaker (❷) can start emitting rows itself. A Top-N operator must have evaluated the entire input before it can guarantee the right output. In the diagram, this breaks the entire query plan into two pipelines.

For query execution, DuckDB uses the **morsel-driven parallelism** framework [70]. The morsel (the data unit) in DuckDB is a row group (i.e., 122 880 rows) [66]. In this framework, a task consists of processing all steps in a pipeline for a given row group. DuckDB assigns these tasks to the threads in its thread pool. In the diagram (❸), thread 1 is assigned the “pipeline 1 – row group 1” task and thus fetches the rows from storage, filters them, and accumulates them in a thread-local Top-N heap. Thread 1 can pick up another task (e.g., for row group 3) when it is ready. The essential idea is that a task consists of multiple steps in the pipeline and uses the row group as the unit of work. This allows multiple threads to process the row groups in parallel, enabling intra-query parallelism.

DuckDB’s physical operators process data during execution. These operators can implement a single or a combination of interfaces, based on the desired behavior. There are three main interfaces (❹): Operator, Sink, and Source³. Some complex operators, such as DuckDB’s hash join, implement all three. As discussed in Chapter 5, the PDXearch extension implements new physical operators that use the Sink and Source interfaces.

The **Sink interface** marks the end/top of a pipeline and consists of three main methods: Sink, Combine, and Finalize. The **Sink method** is called constantly with new input from the operator below, as long as new input is available. Note that this method can be called in parallel. The **Combine method** is called when a single thread has completed execution of its part of the pipeline (e.g., it has processed multiple row groups and there are no row groups left to assign). The **Finalize method** is called once all threads have called Combine.

¹We use these numbers (❶, ❷, ...) to refer to the numbers in the diagram.

²The default number of rows per row group is defined in: https://github.com/duckdb/duckdb/blob/v1.4-andium/src/include/duckdb/storage/storage_info.hpp#L22-L23.

³The physical operator interfaces are defined in: https://github.com/duckdb/duckdb/blob/8e22ef8e576cfb0b462bc8b2ac4faf05e8c19e31/src/include/duckdb/execution/physical_operator.hpp.

3. SYSTEMS

The **Source interface** marks the beginning/bottom of a pipeline and consists of the `GetData` method. The **GetData method** is repeatedly executed to produce chunks of data for the next operator. The method's return value signals once it is done.

The **Operator interface's Execute method** processes one chunk at a time. One chunk in, one chunk out. This is what our example's Filter operator uses.

Internally, DuckDB does not process the entire row group in one chunk; instead, it splits up the row group into **DataChunks**. A DataChunk (⑤) holds the data of up to 2048 rows and consists of one or more **Vectors** [71]. Each Vector represents a single column of data (for those up to 2048 rows) [72].¹ This decomposition of the row group into DataChunks allows the assigned thread to process them in a streaming manner (③). As shown in Figure 3.1, thread 1 fetches row group 1's first chunk of 2048 rows from table storage. It filters that chunk efficiently by evaluating the compact Age Vector, and then merges the passing rows into the thread-local Top-N heap. Thread 1 will iteratively execute these steps until at some point there is no row group left to be assigned. It will execute the Combine method and merge the thread-local heap into the global operator heap. Once all threads have finished, the single-threaded Finalize method (⑥) runs once, after which the Source interface's `GetData` starts emitting the result rows to the pipeline above.

3.1.2 VSS

DuckDB has VSS capabilities built in by default. First, one stores the embeddings in a table column with the `FLOAT ARRAY` type. Then, to perform an exact vector search (ENNS), the user executes an SQL query that combines a VSS similarity-metric based `ORDER BY` and `LIMIT` [73]. For example, the following is a Euclidean distance search in DuckDB where $K = 5$: `SELECT * FROM table ORDER BY array_distance(embedding_column, [0.1, 0.2, 0.3]::FLOAT[3]) LIMIT 5;`. This exact search is expensive because it computes the distance between the query object and all other objects in the dataset.

3.1.3 VSS extensions

Extensions make it possible to add functionality to DuckDB, ranging from official core extensions (e.g., JSON support) to community extensions. These are widely popular, with a total of 160 million extension downloads per month [61, 2:02]. At the time of writing, there exist three VSS extensions that add ANNS capabilities to DuckDB. The official VSS extension [20, 37], *Faiss* extension [50] (which incorporates the index types from the Faiss library [15]), and the *Lance* extension (which supports vector search on the Lance lakehouse format [74]). In this thesis, we explain and benchmark only the VSS extension. Evaluating against the other two extensions is part of future work (Section 7.2).

¹As explained in Section 2.1.1, a DuckDB Vector is not to be confused with vectors/embeddings in vector similarity search.

The official VSS extension developed by DuckDB Labs is called VSS [20, 37]. It was downloaded an estimated 62742 times in the week of April 13th, 2026¹, indicating the demand for VSS functionality in DuckDB [75]. This extension integrates an HNSW index into DuckDB's SQL by using post-filtering and the *USearch* VSS library [14]. After a user creates the VSS index (`CREATE INDEX idx ON dataset USING HNSW (vector_column);`), it will be used to optimize the aforementioned top- k ORDER BY query. DuckDB VSS' HNSW index has multi-threaded index construction and single-threaded search.

3.2 PostgreSQL

PostgreSQL [76, 77] is an open source object-relational DBMS. It has been in development for decades [77] and is one of the most widely used² DBMSs. It is a general purpose DBMS [61, 25:13] and uses row-oriented storage. It was not designed specifically to process analytical queries, but it has features to speed up their execution, including query optimization [78] and parallel execution [79].

pgvector is an extension that adds vector search capabilities to PostgreSQL [38]. The pgvector extension is widely used, with the Python package alone reaching 18 million downloads per month³. It offers ENNS, and ANNS via HNSW and IVF indexes. The indexes are combined with filtering via post-filtering. It has an iterative index scan mechanism to be able to fetch additional items from the index when processing queries with filters that eliminate most rows (as discussed in Section 2.4). Index construction is multi-threaded [38]. In terms of search, ENNS (without an index) can use multiple threads, but the ANNS index searches (HNSW and IVF) are single-threaded⁴.

pgvector indexes support multiple levels of precision (i.e., quantization): vector (32 bits, with support up to 2000 dimensions), halfvec (16 bits, up to 4000 dimensions), and bit (1 bit, up to 64 000 dimensions) [38]. The number of dimensions per embedding are capped because by default the embeddings are persisted on 8 KiB pages, and they do not allow an embedding to span across multiple pages.

¹Extension download statistics are available at: <https://extensions.duckdb.org/downloads-last-week.json>.

²According to the 2025 Stack Overflow survey, PostgreSQL is the most used database among professional developers. See: <https://survey.stackoverflow.co/2025/technology#most-popular-technologies-database-database-prof>.

³pgvector package downloads as of April 20th, 2026, according to: <https://pypistats.org/packages/pgvector>.

⁴The following comment discusses a pgvector parallelized IVF implementation: <https://github.com/pgvector/pgvector/pull/897#issuecomment-3320601437>.

4

PDXearch: Our Extension for VSS in DuckDB

PDXearch is the name of our proposed extension for VSS and filtered VSS inside DuckDB.

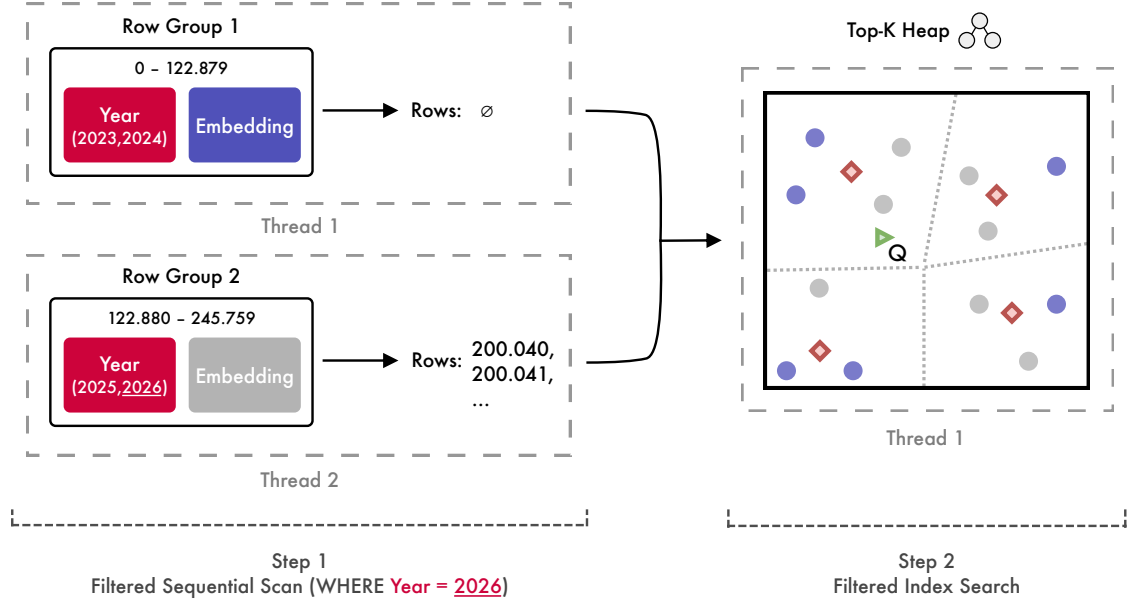
4.1 Two Approaches

Chapter 1 introduced the four decisions that shape the design of the extension. We revisit them and mention why they are well-suited for DuckDB.

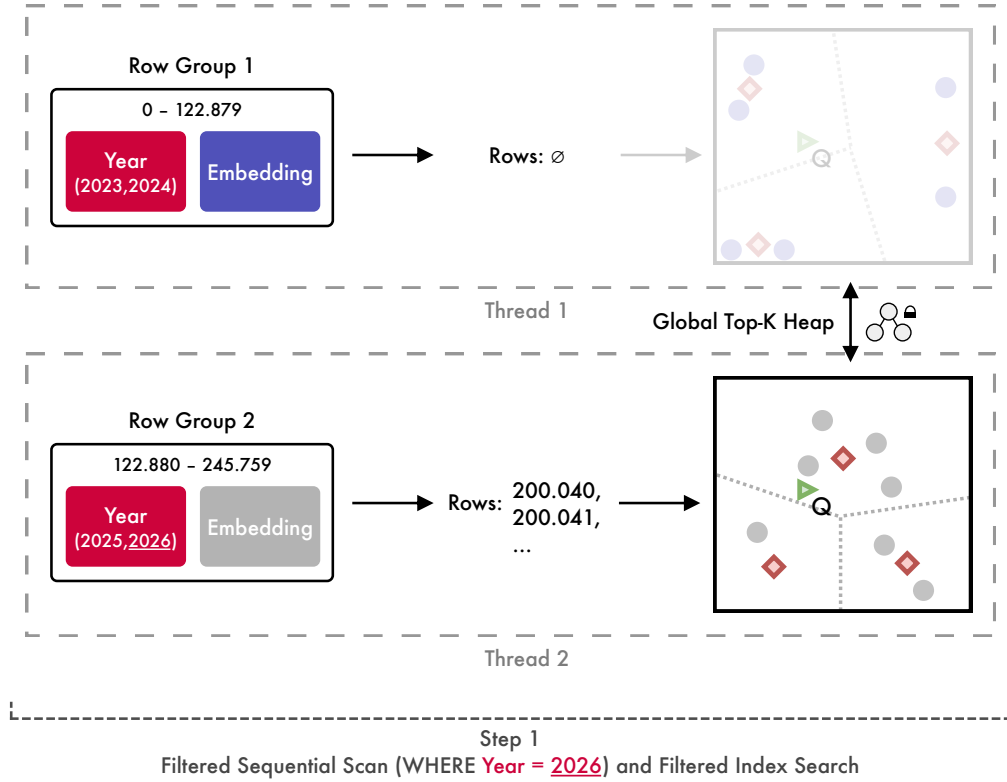
1. Use a SOTA IVF: its lightweight memory footprint and fast construction are ideal for analytical databases, such as DuckDB, in which users expect setup times to be minimal.
2. Integrate pre-filtering for filtered VSS: DuckDB's optimized predicate evaluation (i.e., vectorized execution) makes the overhead of evaluating the predicate much more manageable than other DBMSs [22].
3. Adjust the index to fit the system: integrating tightly with DuckDB's storage and execution model to unlock new tradeoffs.
4. Use quantization: to reduce the memory requirements of the index, which aligns well with DuckDB's ability to be deployed on resource-constrained edge devices [80].

We have designed and built two variants of the PDXearch extension. The *global* variant applies (1), (2), and (4). This is a standard IVF approach, with one IVF index for the entire dataset. This global approach is used mainly as a baseline for the experiments. The second approach is the *row group* variant, which incorporates all four decisions. Instead of a single IVF index for the entire dataset, it has a separate micro IVF index for each row group (i.e., one index for every storage block of at most 122 880 rows). The design of this row group variant is one of the main contributions. Both variants support scalar quantization (SQ). The next two sections provide high-level introductions to the two variants.

4.1 Two Approaches



(a) Design of the PDSearch global variant. The embeddings from all row groups are indexed by a single global IVF index.



(b) Design of the PDSearch row group variant. Each row group has a representative micro IVF index. The row group pruning optimization allows thread 1 to skip searching row group 1's micro IVF because all of row group 1's rows are filtered out. All row groups share a single global top- k heap.

Figure 4.1: Designs of the PDSearch global and PDSearch row group variants.

4. PDXSEARCH: OUR EXTENSION FOR VSS IN DUCKDB

4.2 PDXearch Global

Figure 4.1a shows a filtered search flow using the *global* IVF index integrated into DuckDB. This is a standard pre-filtering IVF approach. Assume that we have a dataset with precisely 245 760 rows, stored across two full row groups. On the right there is a single global IVF index for this dataset, which indexes 245 760 embeddings. The filtered search proceeds in two steps.

First, DuckDB’s morsel-driven parallelism evaluates a filtered sequential scan (i.e., `SELECT rowid FROM table WHERE Year = 2026`). In this case there are two row groups, so we use 2 threads to search in parallel. This scan produces a set of row ids of the rows that pass the filter. In the example, only a subset of the rows in row group 2 pass the filter.

In the second step, the IVF index receives the row ids, uses them to create a selection vector / bitmask, and then uses that bitmask to perform a single-threaded filtered IVF index search. As explained in Figure 2.4, the index computes the distance between the query embedding and each centroid, determines the closest partitions, and then probes the closest n_{probe} partitions. Finally, we use the top- k heap to return the desired rows to the user.

There are multiple downsides to this pre-filtered global IVF approach. First, the index search cannot start until the predicate is evaluated in every row group. Second, inserting or deleting embeddings in any row group will require the global IVF to be updated. Finally, large IVF indexes require out-of-core processing support to create the index and execute a search on memory-constrained systems.

4.3 PDXearch Row Group

We propose the row group approach, shown in Figure 4.1b. It constructs a separate micro IVF for the embeddings in each row group. With this approach the search is no longer blocked until all row groups’ filters have been evaluated. Instead, a thread is assigned a row group, evaluates the filter on those rows, and then immediately searches that row group’s index. This allows many row groups to be filtered and searched in parallel. A single top- k heap is shared between all threads. Compared to the global approach, there are a handful of tradeoffs:

A reduced number of clusters per IVF leads to faster index construction, but increases index size. First, in aggregate, the indexes in the row group approach are faster to construct than the single global index. The construction complexity of Lloyd–Forgy’s K-means clustering algorithm¹ is $O(ncdi)$, where n is the number of d -dimensional embeddings, c is the number of clusters, and i is a constant number of clustering iterations [42, 81, 82, 83]. Hence, the global approach’s complexity is $O(ncdi)$. Assume we have a dataset where every row group is full; we have $r = \frac{n}{122880}$ row groups. The number of clusters c is commonly

¹In our argument we only consider K-means clustering, as its complexity dominates the embedding assignment $O(ncd)$, and other index creation processing steps (e.g., quantization $O(nd)$).

set to $c = \log(n)$. Assuming that we have more than one row group ($r > 1$), the row group will use a lower number of clusters per IVF (c') than the global approach (c). This entails that the construction complexity of the row group approach is $O(122880 \cdot rc'di)$ which simplifies to $O(nc'di)$, where $c' < c$. Hence, in terms of complexity, the row group approach is faster to construct than the single global index. The reduced number of clusters implies that each individual micro IVFs stores less metadata (centroids and assignments) in memory, but the aggregate memory use for metadata will be larger than that of the global IVF: $O(\frac{n}{122880}(c'd + 122880)) > O(cd + n)$. Hence, the in-memory index size of the row group approach is larger than the global approach.

Row group based maintenance is more efficient for common workloads. In general, IVF indexes are already fast to update. Additionally, for the row group approach, inserts and deletes across row groups will only affect their micro IVF indexes, not one large global IVF index. We expect this to be beneficial when maintenance operations are grouped together. For example, in common bulk insert scenarios, new rows (with embeddings) land in dense new row groups, and hence only those micro IVF indexes receive updates. In the worst case, when updates are scattered across many row groups, each of those row groups will need to be updated. Fortunately, we can trivially parallelize the maintenance operations across those row groups. Furthermore, the row group approach is immune to drift when the data distribution is correlated with the insertion order. As new data is inserted, new row groups are created. Each row group's micro IVF runs its own K-means clustering, capturing the row group's data distribution. In the worst case where all new embeddings drift significantly from the existing data's distribution, the global approach would have to completely restructure its partitions, whereas the row group approach handles this automatically by clustering per row group. The performance impact of row-group based maintenance will depend on the implementation and workload.

Trivial intra-query index search parallelism. The row group approach enables trivial intra-query index search parallelism, as shown in Figure 4.1b. Note that a single global IVF search can also be multi-threaded, but this is uncommon (as discussed in Section 2.2.2). Our baseline global IVF implementation is single-threaded¹. Because of this difference in index search parallelism, we expect to see the row group approach exceed in throughput as we scale up the number of threads (though with diminishing returns due to overhead).

Row group pruning opportunity for more efficient filtered search. The row group approach exploits DuckDB's **row group pruning** to make our index search more efficient. In Figure 4.1b, the filter `YEAR = 2026` does not match any rows in row group 1. Thus, the filtered sequential scan there returns an empty row ids set ($\emptyset$). Given that no rows pass, there is no reason to search that row group's micro IVF, hence we skip searching it. This is what we call row group pruning. We expect this to increase search throughput most when the

¹Our global IVF implementation being single-threaded is a thesis limitation, as mentioned in Section 7.2.

4. PDXSEARCH: OUR EXTENSION FOR VSS IN DUCKDB

percentage of rows that pass the filter is low and the filter attribute is correlated with the row ids¹. Figure 4.1b gives a clear example of this — the timestamp (i.e., year) correlates with the row insertion order. This optimization is applicable on datasets with such a correlation and will always benefit the row group approach when there are a large number of row groups and the query’s filter eliminates most rows.

The row group can be used as a unit to achieve out-of-core index construction and search. The row group approach simplifies the implementation of out-of-core index construction and out-of-core index search. A single thread can filter and probe a row group at a time, keeping memory consumption bounded. Row group pruning ties in with this. This is mainly a code complexity benefit; a global IVF could also support out-of-core processing, but it could be more complex if it fetches and searches one partition at a time.

Higher index search latency than the global approach, which intra-query parallelism and row group pruning can partially compensate for. On the other hand, the row group approach’s major downside is increased overhead. First, the implementation has to track the row groups², increasing code complexity. Second, the current design incurs search overhead. As mentioned, the row group approach probes the first n_probe partitions of each row group (in parallel). Assuming the embedding distribution is the same across the row groups, the global IVF will be able to find the relevant query embeddings in X clusters, while the row group has to probe $\frac{n}{122880} \cdot X$ smaller clusters. Hence, the row group approach has to probe more clusters to achieve the same recall; this incurs coordination overhead, thus increasing query latency. Fortunately, the aforementioned row group pruning can help to reduce the number of row groups to search, and intra-query parallelism can spread the work across multiple threads. As we will see in Section 6.3.2, on the dataset sizes that we have tested, the row group approach, when no filters are applied, achieves up to 58.7 % less queries per second (QPS) than the global approach. In future work, billion-scale datasets should be tested. Due to the aforementioned reasons, we expect a higher performance degradation. Additional optimizations might be required to satisfy QPS requirements on the row group approach.

4.4 Summary

We summarize the design decisions and their reasoning and implications in Table 4.1. We aim to confirm these claims in the evaluation (Chapter 6).

¹DuckDB automatically creates min-max indexes (known as zone maps) for all general-purpose data types [84]. In this scenario, beside the reduced VSS cost (due to skipping the index search), an OLAP DBMS such as DuckDB will use row group pruning itself to efficiently evaluate the filter using these zone maps [85].

²Tracking DuckDB row groups is discussed in more detail in Section 5.8.2.

Table 4.1: Design decisions and their implications. Based on Chapter 2 and Chapter 4.

We choose...	Because...	And we gain...
Pre-filtering	Vector-extended relational OLAP DBMSs support agnostic predicates. This flexibility sometimes results in queries where few rows pass, which our design prioritizes. Pre-filtering is optimal for those queries.	• Efficient filtered query execution when few rows pass.
To use the DBMS for the predicate evaluation	We can leverage the OLAP DBMSs' optimized query execution.	• Predicate agnostic query support (support for queries of any complexity, which opens the door to new workloads). • Fast and efficient predicate evaluation. • A filter evaluation that incorporates decades of database theory and optimizations and that receives free updates.
Scalar quantization (SQ)	For most use cases the latency–recall tradeoff is desirable.	• Significantly reduced in-memory size. • Reduced search latency. • Computational overhead for quantizing the embeddings. • Slight decrease in recall.
A SOTA IVF	We design for vector-extended relational OLAP DBMSs with write-heavy workloads and filters that eliminate almost all tuples.	• Search performance competitive with HNSW indexes. • Fast to construct. • Low memory use. • Efficient to update.
One IVF per row group instead of one global IVF	Global has downsides: • Index search cannot start until the predicate is evaluated in every row group. • Updates in one row group affect the entire index. • Complex out-of-core support is required due to the large index.	Upsides of the row group approach compared to the global approach: • Faster index construction times (even with only a single thread). • More efficient maintenance for common workloads. Bulk inserts affect only a few micro IVFs and the row group approach is immune from data drift when the drift correlates with the insertion order). • Trivial intra-query index search parallelism. • Row group pruning opportunity for more efficient filtered search. • A simple out-of-core processing implementation. Downsides of the row group approach compared to the global approach: • Slightly increased total index size. • Increased code complexity for row group tracking. • Overhead of searching all row groups degrades performance for non-filtered queries and filtered queries that eliminate few tuples.

5

PDXearch Row Group Implementation

This chapter explains how we implemented the row group design inside DuckDB. The PDXearch global implementation is similar to the row group implementation. We do not include details on the global approach's implementation in this thesis.

5.1 Software and Dependencies

Unless explicitly mentioned, assume that we are on DuckDB v1.4.4 with PDXearch extension commit cbed8a5¹. We have added end-to-end sqllogic tests and use DuckDB's clang format and clang tidy. Compile-time language features (`#ifdef`) allow the extension to be compiled for one approach (global or row group) at a time. We use CWI's SOTA *Super K-Means* library (introduced in Section 2.2.3) for clustering. The parallelized index creation operator will cluster the micro IVFs in parallel. Super K-Means' hierarchical clustering and sampling features speed up the clustering without compromising the cluster quality [29]. We integrate CWI's SOTA *PDXearch* library (introduced in Section 2.2.3) for the IVF data structures and (filtered) search algorithms. The library has been modified to fit our specific needs.

5.2 Components

Our extension consists of the following components: a DuckDB index (`PDXearchIndex` class), an optimizer rule, and miscellaneous functions (`ScalarFunctions` for the blob interface (Section 5.8.3) and a `TableFunction` for an index metadata pragma). When the extension is loaded, these are all registered through DuckDB's extension loader endpoint.

As mentioned in Section 3.1, DuckDB has exact search (ENNS) capabilities built in. Given a table with a fixed-size `ARRAY` column with embeddings, one can use array distance functions such as `array_distance` together with a query embedding and `ORDER BY` and `LIMIT` clauses to run a vector similarity search query (e.g., `SELECT * FROM table ORDER BY array_distance(embedding_column, [0.1, 0.2, 0.3]::FLOAT[3]) LIMIT 5;`). Instead of adding new VSS syntax (e.g., like the DuckDB *Faiss* extension [50]), we overwrite

¹The PDXearch extension's thesis version can be found at: <https://github.com/Noorts/PDXearch/tree/cbed8a5b35dd0ee8e6697ade0c92679b66ee8d40>.

5.3 Important Consideration: Late Materialization

the execution of the existing SQL syntax. After creating an index, the user can simply use the standard DuckDB top- k query syntax, and their query will be optimized.

The `PDXearchIndex` extends DuckDB's `BoundIndex` class. It stores row group metadata and the micro IVFs' centroids and assignments. The extension also introduces an index creation operator and new non-filtered and filtered search operators. We have added a single optimizer rule that detects both non-filtered and filtered VSS queries. After DuckDB's built-in optimizer rules (e.g., predicate pushdown, late materialization) have run, our new optimizer rule is executed. This will pattern match on the optimized logical query plan and replace the inefficient ENNS (i.e., sequential scan, order-by, and limit operators) with our custom IVF index scan operators. We show how this transforms the query plan in Section 5.5.

5.3 Important Consideration: Late Materialization

DuckDB VSS and DuckDB PDXearch's optimizers do not take into account DuckDB's late materialization optimizer rule. Hence, for these extensions we disable late materialization to avoid inefficient query plans.

DuckDB has a late materialization optimization, which works as follows [86]. Assume a user has a query that projects five columns and applies a filter on one column (e.g., `SELECT col1, ..., col5 FROM t WHERE col1 > 5`). Late materialization, as the term says, attempts to materialize the column data as late as possible. For our example, it means that the execution fetches only the filter column (`col1`) data from storage at first, evaluates the filter, and then fetches the remaining four columns' values for only the rows that passed from storage. Hence, this avoids fetching/materializing and keeping around all five columns from the start, for many rows that will be discarded. There is a tradeoff between the cost of multiple trips to storage (late materialization) vs. fetching and moving multiple columns of data around (eager materialization). By default¹, DuckDB enables late materialization when the number of rows that are emitted are ≤ 50 (so `LIMIT 50`).

Unfortunately, the DuckDB VSS and DuckDB PDXearch implementations' optimizers do not take the late materialization into account, leading to an inefficient query plan. Both the non-filtered and filtered search query plans are affected by this.

Figure 5.1a shows the intended query plan for a non-filtered search² using DuckDB VSS. The index scan operator fetches only the columns that the user requested (e.g., `SELECT id, embedding`) for the top- k rows, so late materialization is already in action inside the operator. Figure 5.1b shows the inefficient plan with late materialization turned on when $k \leq 50$. The extension's optimizer was not programmed to handle this. The index scan correctly emits

¹The default DuckDB late materialization configuration is defined at: <https://github.com/duckdb/duckdb/blob/a8d8e49b32233c1762026c5fc88b321093099489/src/include/duckdb/main/settings.hpp#L837-L845>

²Specifically, Figure 5.1 shows the result of `SELECT id, embedding FROM t ORDER BY array_distance(embedding, repeat([1,2,3,4], 192)::FLOAT[768]) LIMIT 10`; on the Cohere1M dataset.

5. PDSEARCH ROW GROUP IMPLEMENTATION

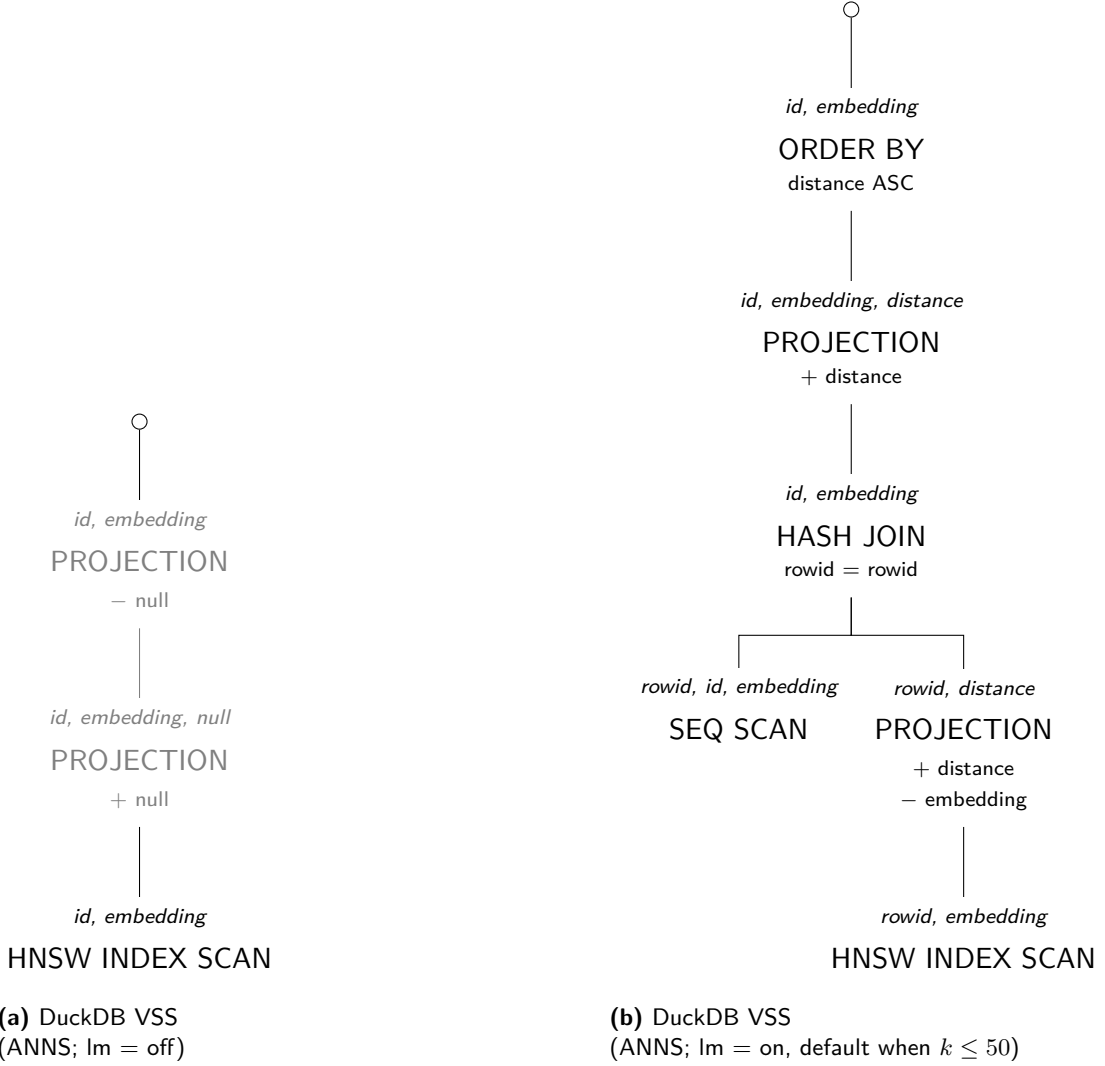


Figure 5.1: Impact of late materialization on query plans. Figure 5.1b shows an inefficient optimized query plan for a $K = 10$ non-filtered search using DuckDB VSS, where DuckDB is configured with the default `late_materialization_max_rows = 50`. Both the DuckDB VSS and DuckDB PDSearch implementations do not account for the late materialization optimization yet, leading to this inefficient query plan. A filtered search query (not shown) leads to a similarly inefficient query plan. In the thesis benchmarking, we work around this by manually disabling the late materialization optimization, which results in the query plan shown in Figure 5.1a.

the row id, but also emits the embedding for the redundant projection on top. As expected in late materialization, the leftmost sequential scan materializes the desired columns (id, embedding) for the top- k rows, reorders the rows, and returns the result to the user. We explain the query tree plans in more detail in the upcoming sections.

Going forward, we have explicitly disabled¹ the late materialization optimization for both DuckDB VSS and DuckDB PDXearch. This means that optimal query plans are used in the following sections and for the performance evaluation in Chapter 6. Note that, as just mentioned, by their design, internally the indexes apply late materialization. We leave late materialization enabled for DuckDB’s plain ENNS². In the future, DuckDB VSS and DuckDB PDXearch should be adjusted to 1) to embrace late materialization, e.g., by having the index scan operator emit only row ids, removing the projection on top, and letting DuckDB’s late materialization operators handle the rest, or 2) disable late materialization transparently (without user intervention) and let the index scan fetch all column values for the top- k rows.

5.4 Index Creation

This section, focused on index creation, discusses the user’s interaction with the index, the index creation query plan, and finally the implementation of the index creation operator.

Interface When the user has a DuckDB table with embeddings stored in a D -dimensional ARRAY column, the user can create a PDXearch index on the table. From the user’s perspective, only one database index is created. This database index has the micro IVF indexes stored inside. A user creates an index with: `CREATE INDEX idx ON table USING PDSEARCH (embedding_column) WITH (metric = 'l2sq', quantization = 'U8');`. The WITH clause allows the user to specify index construction parameters (i.e., the distance metric and the quantization to use). The number of partitions that the indexes use is determined internally based on the number of embeddings per row group.

Query plan Executing a PDXearch index creation query produces the query plan shown in Figure 5.2. DuckDB VSS creates the same query plan. The sequential scan operator fetches the row id and embedding columns for all rows from the table. These rows flow through a projection and filter operator until they reach a custom PDXearch index creation operator (EXTENSION). The projection executes any expressions on the key columns. The filter operator filters out rows where `embedding = NULL`. We now explain the index creation operator (EXTENSION) in detail.

¹The following query disables the late materialization optimization:
`SET late_materialization_max_rows = 0;`

²We decided to leave late materialization enabled for DuckDB’s exact search. ~20 ad hoc single-threaded filtered search queries on Cohere768D1M showed that enabling late materialization on those specific queries increases query latency by only approximately ~3%.

5. PDSEARCH ROW GROUP IMPLEMENTATION

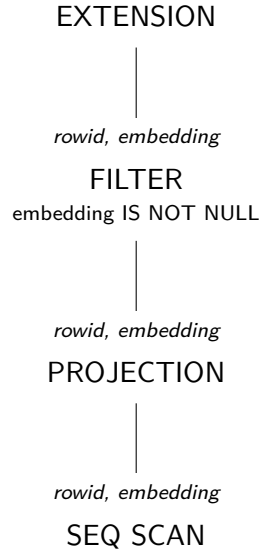


Figure 5.2: Index creation query plan of the DuckDB PDSearch and DuckDB VSS indexes. *Italic text above an operator indicates the columns it projects. Small text below an operator provides details (e.g., the filter expression).*

Operator implementation As discussed in the design section (Chapter 4), the PDSearch row group index creation is parallelized. In parallel we create a separate micro IVF index for each row group (a chunk of at most 122 880 rows). These micro IVFs and their metadata are bundled into a single DuckDB index.

As shown in Figure 5.3, the sequential scan, projection, filter, and PDSearch index create operators take the following steps to create the PDSearch row group index:

We start with the sequential operator at the bottom of the diagram (❶). The sequential scan fetches row id–embedding pairs from table storage. We need the row id as metadata, and the embedding for it to be stored inside a micro IVF index. We will process all rows in the table. The rows are provided in chunks of at most 2048 rows (this is a DuckDB DataChunk). This chunk contains both the row id and embedding column for those rows. Given a full row group of 122 880 rows, the sequential scan will emit 60 DataChunks.

The sequential scan is parallelized, so multiple threads fetch data at the same time. DuckDB provides an important guarantee: a row group is only ever processed by one thread at a time. That is, a single thread is responsible for processing a row group. As shown in the figure, assuming thread 1 starts fetching row group 1, then it will fetch all chunks of that row group. Hence, thread 2 *cannot* also fetch chunks for row group 1. Once a thread has processed a row group, it is assigned another one (if any are left).

The PDSearch index create operator implements the Sink interface, with the Sink, Combine, and Finalize methods. Assume the sequential scan just emitted the first 2048 rowid–embedding pairs of row group 1. This chunk moves the control flow into the *Sink* method,

which is executed for every chunk. Given that this is the first row group, we answer *no* to the if-statement and go to ②. We take the chunk of at most 2048 rows and rotate the embeddings¹, and then accumulate the row ids and preprocessed embeddings in a thread-local in-memory buffer. Now control flow returns to the sequential scan which likely emits another chunk.

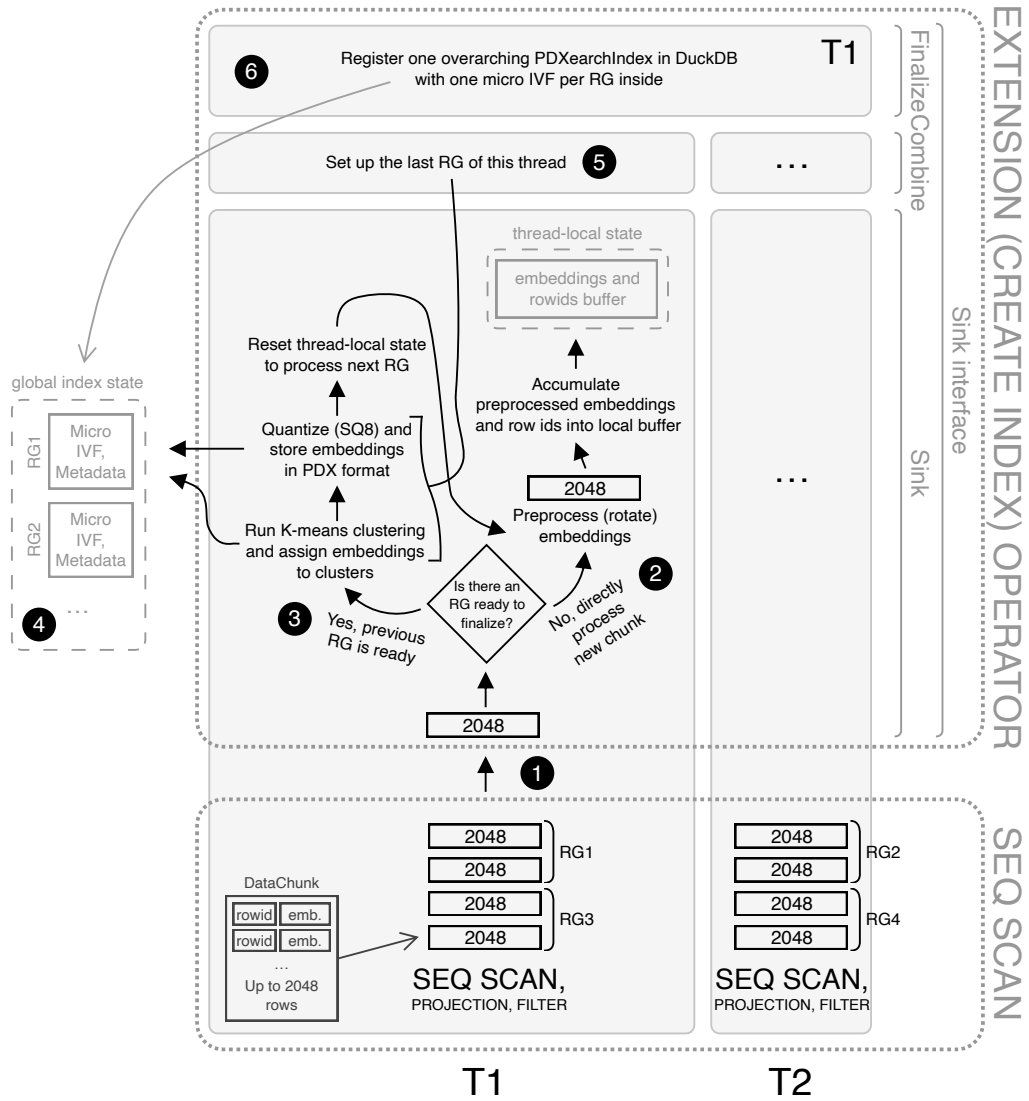


Figure 5.3: Implementation of DuckDB PDXearch's row group (RG) parallelized index creation. Execution flow is from bottom to top. We use the following query: `CREATE INDEX idx ON table USING PDSEARCH (embedding_column);`. T1 and T2 indicate the threads that execute the steps.

¹Our pruning IVF search algorithm requires the embeddings to be multiplied by a rotation matrix. We rotate them here at index creation time to prepare for later index searches.

5. PDXSEARCH ROW GROUP IMPLEMENTATION

This *preprocess and accumulate* process continues until we have processed the entire row group (at most 60 Sink method invocations). When the first chunk of the next row group arrives, we execute the logic in ❸. That is, we set up the micro IVF index for the previous row group that is ready in our thread-local buffer. Setting up the index entails running K-means clustering, assigning the embeddings to clusters, setting up the necessary data structures (i.e., centroids and inverted lists), quantizing (scalar quantization (SQ)) from Float32 to UInt8 if necessary, and moving the (quantized) embeddings to new buffers (one buffer per IVF partition/cluster) that use the columnar-decomposed PDX data layout. The resulting micro IVF and metadata are stored in the global operator state (❹). The thread then resets its state to be able to process the next row group, and jumps to ❷ to process the first chunk (at most 2048 rows) of the new row group it received. This process repeats in parallel until all row groups have been processed.

Once there are no more row groups for the thread to process, the thread's Sink invocations stop, and the thread-local *Combine* method (❺) is called. If there is still an accumulated row group left in the thread's local buffer, then we jump to ❸ to create the index for this thread's last row group. Afterward, the thread returns to the thread pool.

Once all threads have executed their Combine method, we run the single-threaded *Finalize* method once (❻). This “barrier” takes the micro IVF indexes and metadata, and registers them in DuckDB storage as a single DuckDB index. This index is persisted in memory and ready to be used for an index search.

5.5 Non-Filtered Search

Interface A non-filtered vector search query in DuckDB looks like: `SELECT id, embedding FROM table ORDER BY array_distance(embedding_column, [0.1, 0.2, 0.3]::FLOAT[3]) LIMIT 10;`. By default, DuckDB will process this as an exact search (without an index). The PDXearch extension implements and registers an optimization pass in DuckDB. Before query execution, this optimization pass will rewrite the traditional exact search query plan to a query plan that uses a new PDXearch non-filtered search operator, as shown in Figure 5.4. This requires a matching¹ PDXearch index to have been created on the table beforehand. Running `SET pdxearch_n_probe = 5` beforehand allows the user to configure the number of partitions probed during the search.

Query plans We start with the query plan (Figure 5.4a) of DuckDB's non-filtered exact K-nearest neighbors search (ENNS). The right subtree determines the row ids of the top-*k* rows through a parallelized brute force exact search. DuckDB's late materialization adds the

¹A DuckDB PDXearch index is bound to a specific table and column. It will only optimize queries that use the same distance metric and quantization settings that the index was created with. It is possible to create multiple PDXearch indexes on the same table to optimize multiple scenarios.

left subtree and the remainder of the tree on top. The leftmost sequential scan and hash join efficiently fetch the exact columns the user projected (including the embedding column) for the top- k rows. Since the hash join does not retain the row order, the rows are reordered at the top.

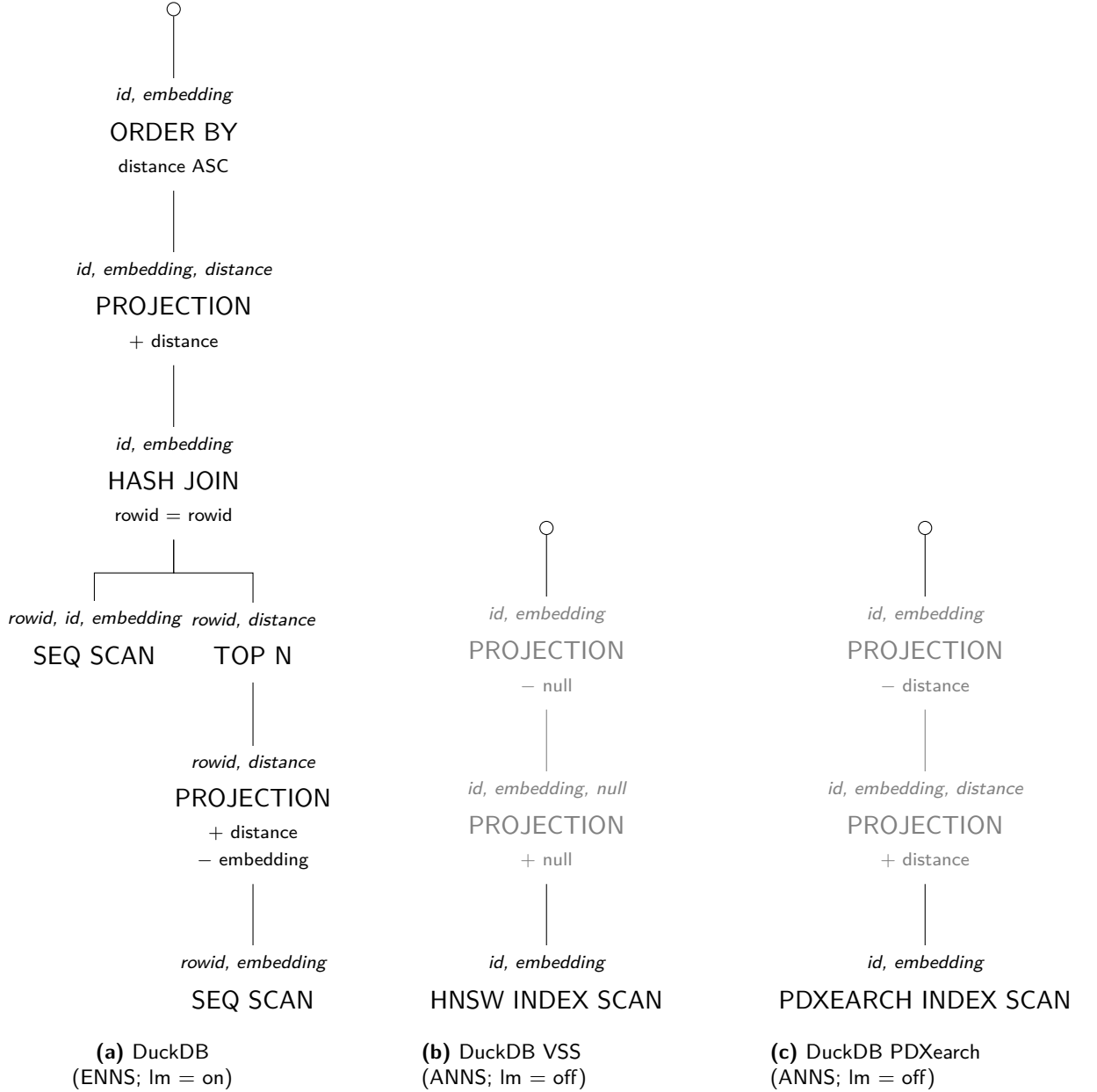


Figure 5.4: Non-filtered search query plans for DuckDB, DuckDB VSS, and DuckDB PDXearch, where $K = 10$ and we project `id` and `embedding`. *lm* indicates whether late materialization is enabled. It is disabled for DuckDB VSS and DuckDB PDXearch (Section 5.3). *Italic text* above an operator indicates the columns it projects. Small text below an operator provides details (e.g., a change in the projected columns).

5. PDXSEARCH ROW GROUP IMPLEMENTATION

Figure 5.4b shows DuckDB VSS' HNSW index scan query plan, with late materialization disabled. This is an efficient VSS index scan that emits the user's requested columns for the top- k rows. The gray projection operators on top are unnecessary. They were left behind by DuckDB VSS' optimizer when it transformed the original optimized logical plan. Fortunately, these operators have negligible performance overhead.

The PDXearch query plan (Figure 5.4c) is similar — an efficient PDXearch index scan that materializes only the top- k rows. Due to a bug¹ the gray projections on top are slightly more expensive. Fortunately, the overhead remains negligible², and thus the benchmarking experiments are not affected.

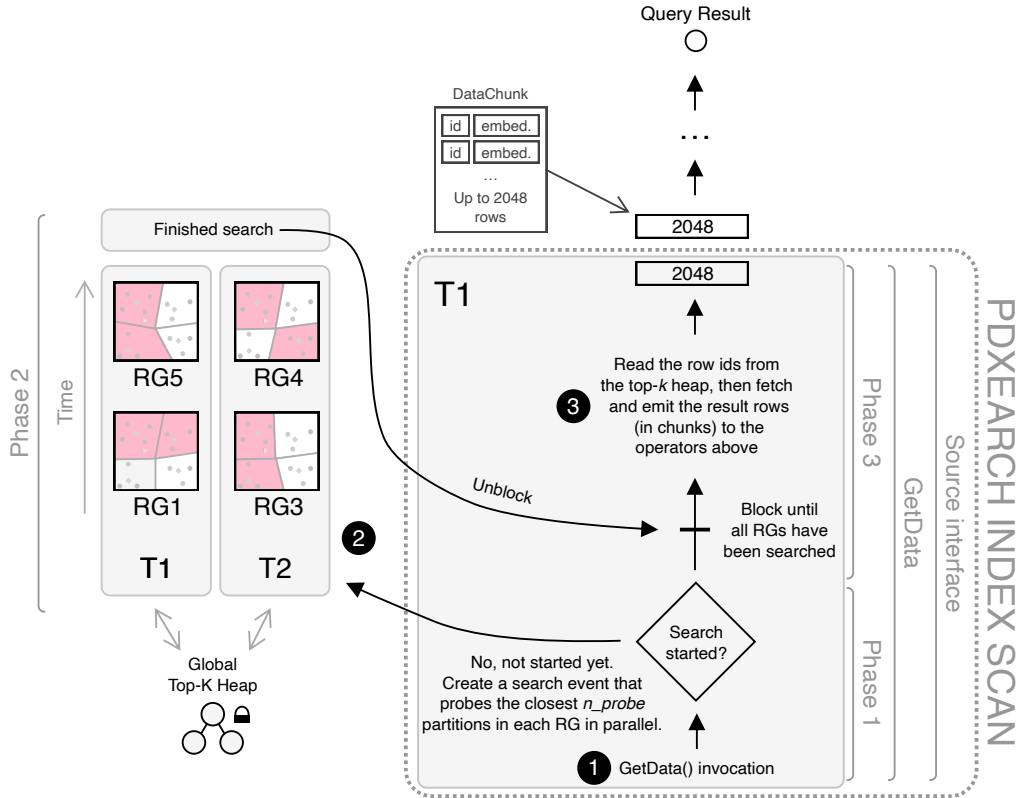


Figure 5.5: Implementation of DuckDB PDXearch's row group (RG) parallelized index search. Execution flow is from bottom to top. We use the following query: `SELECT id, embedding FROM table ORDER BY array_distance(embedding_column, [0.1, 0.2, 0.3]::FLOAT[3]) LIMIT 10;`. T1 and T2 indicate the threads that execute the steps.

¹As shown, the projection computes the distance for the top- k items and then discards them. This bug has since been fixed. See: <https://github.com/Noorts/PDXearch/commit/4898adf8c44efddd1be5c2a34ecd9ff3d995f63a>.

²An ad hoc experiment of ~30 single-threaded queries showed that the combined time spent in the projection operators is 60 microseconds or less.

Operator implementation The row group index has a micro IVF for each row group. The non-filtered search operator searches these IVFs in parallel using multiple threads and a single shared global top- k heap. After the closest n_probe partitions of each micro IVF have been probed, the heap contains the row ids of the top- k rows. The index scan operator uses DuckDB's local storage API to fetch the column data of these top- k items and emits them in DataChunks of 2048 rows at a time. These pass through the aforementioned projection operators and are then provided to the user as the query result.

We wanted the emission of top- k results to be single-threaded, while the index search should be multi-threaded. Furthermore, the index scan is the bottom-most operator of the query plan; we cannot elegantly use a parallel Sink interface to search the IVFs. We settled on a single-threaded *Source* interface combined with custom DuckDB *Events* and *ExecutorTasks* to make parallel searching possible. We set up the Source interface's *GetData* method to execute in 3 phases. As shown in Figure 5.5, the PDSearch row group non-filtered search operator works as follows:

Phase 1 The *GetData* method starts off in phase 1, where the first invocation (❶) of this single-threaded method initializes a global heap, and creates and schedules a custom search task for each row group. To be precise, the micro IVF search *ExecutorTasks* are bundled in an *ExecutorEvent*. The initialization thread then blocks until the search is complete.

Phase 2 While the *GetData* method is blocked, DuckDB uses its global thread pool to pull the *ExecutorTasks* from the queue. Each thread (❷) computes the distances to all the centroids in its assigned micro IVF, prioritizes the partitions based on centroid distance, and then probes the first/closest n_probe partitions in that micro IVF. The top- k heap is shared across threads. Once all row groups have been probed for n_probe partitions, all *ExecutorTasks* have been processed. This triggers the completion of the overarching *ExecutorEvent*, which unblocks the *GetData* method.

Phase 3 With the search completed, we now have a global heap with the row ids of the top- k embeddings. The unblocked single-threaded *GetData* method is invoked again (❸) and can now use the global heap's row ids together with DuckDB's local storage API to fetch the column data (*SELECT **) of the top- k rows that the user requested. As mentioned, this is in DataChunks of 2048 rows to support the rare case where $k > 2048$. This data is passed directly to the next operator in the query plan until it ends up in DuckDB's output operator.

5.6 Filtered Search

Interface A filtered vector search in DuckDB looks like: `SELECT id, embedding FROM table WHERE year = 2026 ORDER BY array_distance(embedding_column, [0.1, 0.2, 0.3]::FLOAT[3]) LIMIT 10;` Note that this is the same as the non-filtered query, except that a *WHERE* clause has been added.

5. PDSEARCH ROW GROUP IMPLEMENTATION

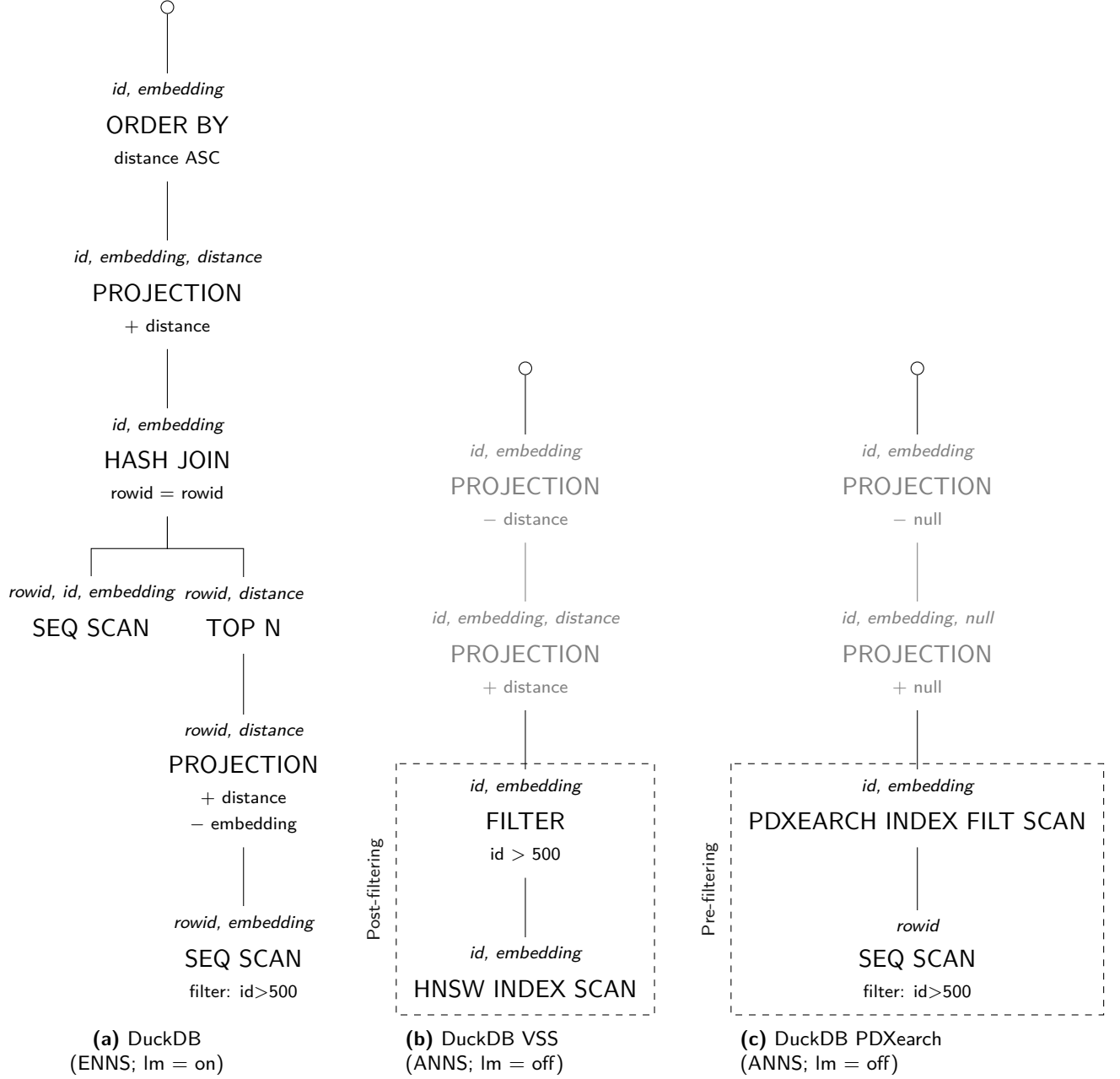


Figure 5.6: Filtered search query plans for DuckDB, DuckDB VSS, and DuckDB PDSearch, where $K = 10$ and we project *id* and *embedding*. lm indicates whether late materialization is enabled. It is disabled for DuckDB VSS and DuckDB PDSearch (Section 5.3). *Italic text* above an operator indicates the columns it projects. Small text below an operator provides details (e.g., a change in the projected columns).

Query plans DuckDB's filtered ENNS query plan is shown in Figure 5.6a. This is the same as the aforementioned non-filtered variant, except that the bottommost sequential scan has a pushed down filter. This ensures that the right subtree only emits top- k rows that pass the filter. The remainder of the plan materializes the remaining columns of the top- k rows.

Figure 5.6b shows DuckDB VSS' classic post-filtering setup. The HNSW index scan operator produces the top- k rows, which are filtered by the operator on top. It becomes immediately clear that this can cause the aforementioned $< k$ results scenario, where there are k passing rows in the dataset, yet the query returns $< k$ or even an empty result. An iteration mechanism (such as the one used by pgvector) would allow the filter to fetch more items from the index scan in case any are filtered out.

In Figure 5.6c, we see the PDXearch index scan operator again, but this time it receives the row ids of the rows that pass the filter from the sequential scan below. The PDXearch index filtered scan operator accumulates these row ids to create a selection vector / bitmask for its pre-filtered index scan. Similarly to DuckDB ENNS, the filter is pushed down into the sequential scan. In Figure 5.6b and Figure 5.6c again we see the unnecessary but negligible projection operators. This time, the redundant distance calculation was left in the DuckDB VSS plan.

Operator implementation As discussed, the PDXearch filtered search row group variant combines pre-filtering with a parallelized micro IVF search.

The filtered search operator implements both the Sink and Source interfaces. The sequential scan and parallel Sink interface handle the filtering and searching of the index. Similarly to the non-filtered index scan, the single-threaded Source interface emits the results (the top- k rows) to the next operators, and eventually to the user.

As shown in Figure 5.7, the PDXearch row group filtered search operator works as follows:

We start with the parallel sequential scans (❶), like the index creation operator (Section 5.4). This time, however, the sequential scan evaluates the pushed-down filter/predicate and emits the row ids of the rows that passed the predicate. As usual, these are grouped in 2048 row DataChunks. DuckDB guarantees that only one thread processes a given row group.

Assume that the sequential scan emitted the first DataChunk of row group 1. The thread's control flow reaches the *Sink* method and jumps to ❷, merging the passed row ids into the thread-local row id buffer. The Sink method is now done, and control returns to the sequential scan, which will likely emit another DataChunk. Assuming thread 1 processes another row group, at some point the DataChunk it receives will belong to another row group. That is when the mechanism concludes it will receive no more row ids for the previous row group, and hence that row group is ready to be searched.¹

¹If no row ids pass the predicate, then the sequential scan will not emit a row id for that partition, and subsequently it will not be searched by the PDXearch filtered search operator. This is the *row group pruning* optimization in action.

5. PDXSEARCH ROW GROUP IMPLEMENTATION

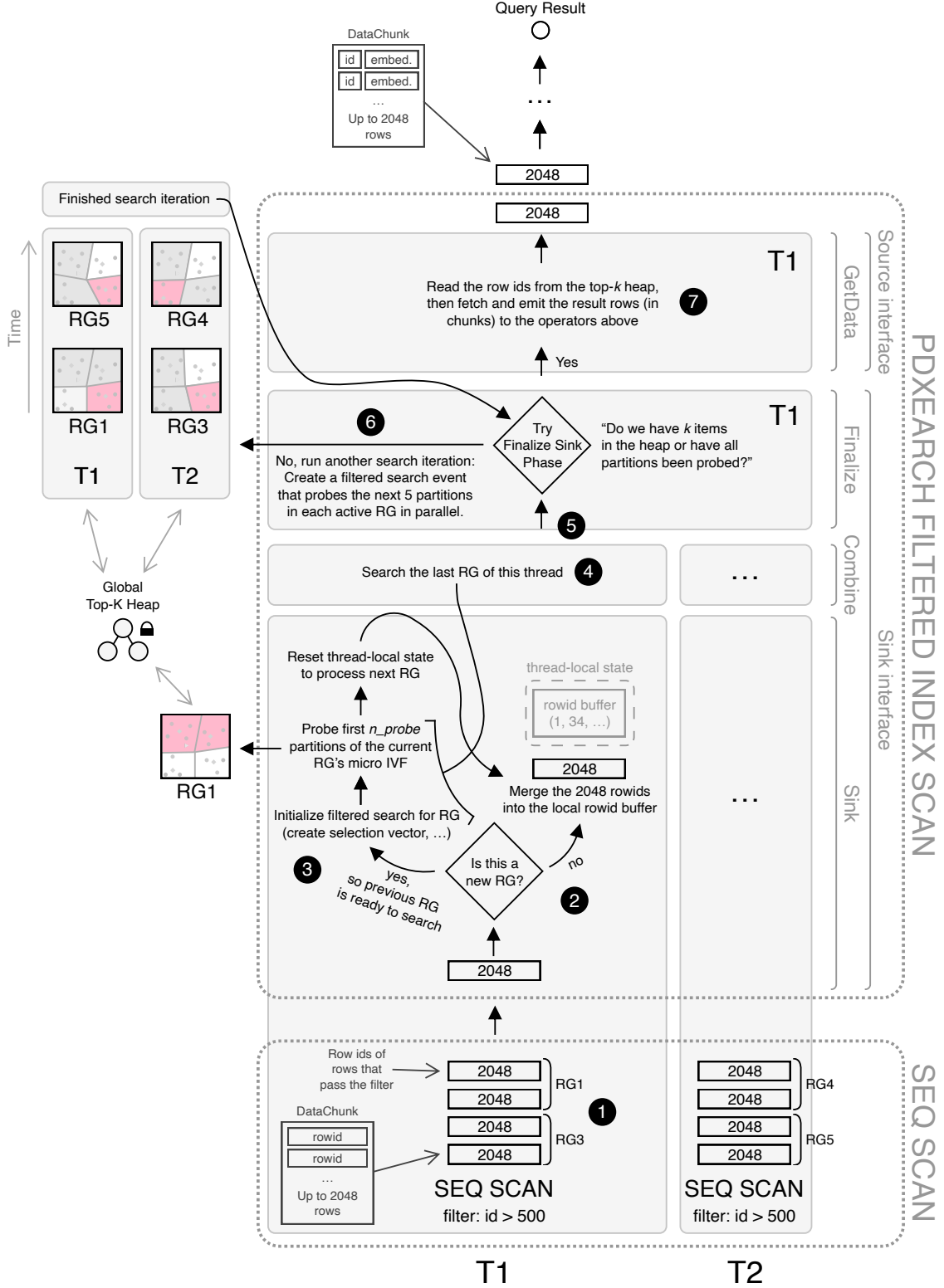


Figure 5.7: Implementation of DuckDB PDXsearch's row group (RG) parallelized index filtered search. Execution flow is from bottom to top. We use the following query: `SELECT id, embedding FROM table WHERE year = 2026 ORDER BY array_distance(embedding_column, [0.1, 0.2, 0.3]::FLOAT[3]) LIMIT 10;` T1 and T2 indicate the threads that execute the steps.

The thread will jump to ③ and first initialize the search by creating a selection vector / bitmask¹ from the thread-local row id buffer and compute the distances from the query embedding to the centroids to determine the partition probe order/priority. After initializing the search, it will probe the first `n_probe` partitions in that row group. This populates the global heap. We call this the *initial probe iteration*. After the search, it will reset its local state and continue by processing the DataChunk it received for the new row group (②). The threads repeat this process in parallel until all row groups have been processed.

Once there are no new row ids for the thread to process, the thread's Sink invocations stop. The thread-local *Combine* method (④) is then called. If there are accumulated row ids for a row group left in the thread's local buffer, then we jump to ③ to search the index for this thread's last row group. Afterward, the thread returns to the thread pool.

Once all threads have executed their Combine method, we run the single-threaded *Finalize* method once (⑤). This method is complex. At this stage, all row ids from the sequential scan have been consumed — each relevant² row group will have had its first most promising `n_probe` partitions searched. Ideally, similarly to the non-filtered search, we could move from the Finalize method to the Source interface's *GetData* method and start emitting the top- k results. However, imagine the following scenario: the user set $n_probe = 1^3$, $k = 10$, and there are 5 row groups in total. It might be that only 1 row passes the filter in each row group's probed partition. Hence, the heap now has 5 elements, not 10. At this point we could emit the results, but because of low recall this leads to an unsatisfying user experience. If we probed more partitions, then those would contain at least 1 row that passes the filter each⁴. Instead of requiring the user to increase `n_probe` manually and rerunning the query in hope of higher recall⁵, we have added an iteration mechanism.

The iteration mechanism starts with the question: “Can we finalize the sink phase and move on to emitting the results?” It does this using the following condition: `is_heap_filled_with_k_results || have_all_partitions_been_probed`. If *no*, then we are in the scenario discussed above; our heap is not full, and if we probe additional partitions, then there is a chance to add more items to the heap. In this case, we jump to ⑥, which creates and schedules an *ExecutorEvent* and underlying *ExecutorTasks*. This new event blocks the operator from transitioning to the *GetData* method. Each task is assigned one row

¹The PDSearch search library uses this bitmask to only search the rows that passed the predicate.

²The exception here are the pruned row groups; those will not have been searched.

³`n_probe` indicates the number of partitions to probe per row group. So the total number of explored partitions is: `n_probe × num_row_groups`.

⁴If the bitmap indicates that none of the embeddings in a cluster pass, then the PDSearch search library will skip that partition. Note that in the current implementation it will not search another one in its place. Finally, for clarity: the PDSearch filtered search operator 1) skips searching a complete row group's micro IVF if no rows pass in that row group at all. And 2) the PDSearch micro IVF search will skip a partition in that IVF if no rows pass for that partition.

⁵In non-filtered search, users can use the `n_probe` runtime parameter to tweak the throughput vs. recall tradeoff. However, for filtered search this seems unreliable and toilsome, as the dataset and the specific filter affect the per-cluster filtering percentage.

5. PDXSEARCH ROW GROUP IMPLEMENTATION

group that we previously searched in ③¹. The task will take the micro IVF's search state and probe the next 5 partitions². Similarly to the non-filtered search, in parallel — DuckDB's threads will pick up and execute these search tasks. As shown in Figure 5.7, once this iteration of search tasks is complete, a single thread will re-evaluate the condition. The mechanism can trigger multiple times until after a finite number of search iterations³ the question's answer is yes and we finalize the Sink phase.

Finally, we execute the GetData method (⑦), which emits the results into the pipeline above. We use the global heap's row ids together with DuckDB's local storage API to fetch the column data (SELECT *) of the heap's rows that the user requested. As mentioned, this is in DataChunks of 2048 rows to support the rare case where $k > 2048$. This data is passed directly to the next operator in the query plan, where in this case, it ends up in DuckDB's output operator.

5.7 Integrated Filtering

Section 2.4 introduced the most common ways to integrate/combine the DBMS's filter evaluation with vector search (e.g., pre-filtering, post-filtering). In this section, we clarify the PDXsearch extension's integration.

Although the intricate details of the PDXsearch algorithm are out of scope for this thesis⁴, we briefly explain the idea behind the algorithm's pruning mechanism. Suppose that our distance metric is monotonically increasing (e.g., Euclidean distance) and that we are interested in the 3 embeddings closest to the query embedding (i.e., $k = 3$). Furthermore, assume that, during initialization, we compute the exact distance between the query embedding and any 3 embeddings from the dataset that have not been filtered out. The distances of these 3 embeddings make up our initial top- k max heap. The distance at the top of this max heap acts as a pruning threshold. Any embeddings that want to enter the top- k must have a distance lower than this threshold; otherwise, they would not be closer to the query embedding. When the PDXsearch algorithm computes the distances between the query embedding and the remaining dataset embeddings, it will use this pruning threshold to abort computing the distance for an embedding as soon as it recognizes that the distance will be larger than the threshold. For example, given some embedding, if after evaluating only the first 10 dimensions the partial distance already exceeds the pruning threshold, then the algorithm aborts any further computations for that embedding, as those could only increase the partial distance. This dimension pruning mechanism increases the PDXsearch algorithm's efficiency (i.e., reduced

¹Hence, pruned row groups, those with no passing rows in a micro IVF, are skipped.

²5 partitions per iteration attempts to balance between query latency and recall, and iteration overhead. Future experiments will have to determine this empirically.

³Either we reach k items in the heap, or we exhaust/probe all partitions in all the initialized micro IVFs.

⁴For more details about the PDXsearch algorithm, refer to the introductory blog post [87], the PDX paper [9], or the PDXsearch library [27].

memory access and reduced calculations) by avoiding computing the full distance (i.e., taking all dimensions into account) for many embeddings, without affecting the correctness and quality of the final top- k .

The PDXearch extension tightly integrates the PDXearch algorithm and dynamically adjusts the number of VSS distance calculations based on the filtering percentage of the predicate. When most embeddings pass the filter, then the algorithm's dimension pruning aggressively reduces the amount of data access and number of computations. In the opposite scenario, when most embeddings are eliminated, the extension's iteration mechanism (Section 5.6) increases the number of clusters to probe to prevent returning less than k results when possible.

Although this thesis refers to our integration as pre-filtering — as we have seen, it is slightly more complex. The definition of pre-filtering introduced in Section 2.4 suggests a clear boundary: evaluate the predicates and then compute the distances between the query embedding and only the dataset embeddings that passed the filter to determine the top- k . The more rows that are eliminated, the fewer embeddings remain for which distance calculations must be computed.

Our tight integration does not have this clear boundary. Although a row group's predicates are evaluated by DuckDB up front, we do not implement strict pre-filtering. First, besides dimension pruning, the PDXearch algorithm heavily relies on block-based sequential memory access and SIMD instructions to achieve high performance. If the integration adhered to strict pre-filtering, then it would not compute distances for filtered-out tuples (i.e., by skipping over tuples through branching logic). Kuffo et al. have shown that this tuple skipping is detrimental for the algorithm's performance when $> 20\%$ of rows pass the filter [9]. Hence, for efficiency, the algorithm will still include filtered-out embeddings in the initial phase of its calculations. Second, the integration does not only filter before the search but also during the search, through dimension pruning. Both of these points show that our tight integration is not strict pre-filtering.

This intertwined and tight integration, with its dynamic increase and decrease of the number of embeddings searched based on the filtering percentage, is another example of the thesis' focus on VSS-DBMS co-design, resulting in improved efficiency and result quality. For simplicity, in the remainder of the thesis we continue to refer to our integration as "pre-filtering".

5.8 Limitations of the Implementation

In this section, we discuss limitations of the current implementation. Similarly to the late materialization inefficiency we discussed in Section 5.3, these are not limits of the design, but limits of the current implementation. Section 7.3 provides a full overview of the extension's future work.

5. PDXSEARCH ROW GROUP IMPLEMENTATION

5.8.1 Filtered search complexity

The DuckDB VSS and DuckDB PDXsearch implementations are currently not predicate agnostic. They do cover the filter types (e.g., equality, range) that are used in the experiments. Making PDXsearch predicate agnostic is future work.

As of DuckDB v1.4.4, the DuckDB VSS [20] and DuckDB PDXsearch implementations are not predicate agnostic. Filtered VSS queries that are not yet optimized include filtered queries with a disjunction spanning multiple columns (e.g., `SELECT * FROM test_table WHERE col1 > 500 OR col2 < 50 ORDER BY array_distance(embedding, [0.1, 0.2]::FLOAT[2]) LIMIT 10;`). For these queries, DuckDB will fall back to DuckDB's ENNS. Fortunately, these queries are not part of the evaluation. Making the DuckDB PDXsearch implementation predicate agnostic (support for the aforementioned disjunction filters, JOINS, etc.) is permitted by the design but requires the PDXsearch optimizer to be rewritten, which is part of future work¹.

5.8.2 Row id metadata

The DuckDB PDXsearch row group approach requires row-id-to-row-group-id metadata. This seems inaccessible in DuckDB v1.4.4. We work around this by ensuring row groups are full, making the metadata mapping trivial. Supporting non-full row groups is part of future work.

For the row group approach, we need metadata about the row groups themselves. The rows of a row group are pushed into our operators in chunks (DuckDB DataChunks). Given a row id, we need to know the row group it belongs to. DuckDB's row groups need not be filled completely (e.g., 122880, 40000, 122880), which further complicates the implementation. As of DuckDB v1.4.4, it appears that the DuckDB metadata API we need is not available².

In the current implementation, we assume that all but the last row group have 122880 rows (e.g., 122880, 122880, 40000). We use the following formula to determine the row group id based on the row id: `row_group_id = floor(row_id / 122880)`. In the tests and experiments, we insert the rows in batches of 122880, which under DuckDB v1.4.4 behavior results in full row groups. Supporting arbitrarily sized row groups and compaction is part of future work.

¹For more information about predicate agnostic support, see Section 7.3 or the following code comment: https://github.com/Noorts/PDXsearch/blob/e92d178843390a38298e2756432c7f1c270617fa/src/index/search/pdxsearch_scan_optimizer.cpp#L28-L210.

²We have looked at this extensively and might have to expand DuckDB's API upstream. Concerns include accessing this information during index maintenance (the `BoundIndex`'s `Append` method), dealing with transactions, support for data sources (e.g., Parquet), and synchronizing state after DuckDB's row group compaction (DuckDB v1.5.1). Alternatively, we could drop the row group association and instead map blocks of rows to micro IVFs (potentially spanning multiple row groups), but that approach has other tradeoffs, and concerns remain (e.g., handling compaction).

5.8.3 DuckDB planner and optimizer overhead

DuckDB has significant overhead in the preprocessing of vector search queries. We introduce a new “blob interface” syntax in the DuckDB PDXearch and DuckDB VSS extensions to avoid this overhead and reveal the true performance in the experiments.

As discussed earlier, DuckDB is a DBMS designed to process analytical query workloads. These workloads consist of processing large amounts of data (i.e., many rows), which is why DuckDB spends a significant amount of time optimizing the query plan before executing. Assume we have a top- k query: `SELECT * FROM table ORDER BY array_distance(emb, [0.1, 0.2]::FLOAT[2]) LIMIT 10;`. Unfortunately, the current DuckDB planner and optimizer spend a significant amount of time parsing, typing, and constant folding each individual element of the `[0.1, 0.2]` query embedding, which here has 2 elements/dimensions ($D = 2$), but which is usually in the range of 512 to 4096 dimensions. It spends so much time preprocessing the query that more time is spent on this query preparation than on the VSS index scan. Hence, the top- k query throughput is significantly reduced. We quantify this overhead in an experiment discussed in Section 6.3.1.

This overhead affects both the DuckDB VSS and new DuckDB PDXearch extensions, as they both use the top- k SQL interface. DuckDB Labs’ second blog post [88] written about the VSS extension acknowledges this overhead, stating that the 1:N vector similarity search suffers significant overhead. Instead of trying to reduce the time spent optimizing, the DuckDB Labs team added support for M:N similarity joins¹. This workload increases the amount of processing and thus amortizes the overhead across more work.

The DuckDB overhead is a roadblock for this thesis’ performance evaluation, which is why we modified the DuckDB PDXearch extension² and forked the DuckDB VSS extension³ to support a new *blob* interface. The key idea behind the blob interface is to ensure that DuckDB’s query planner and optimizers treat the query embedding (e.g., `[0.1, 0.2]`) as a single unit, instead of D individual elements. The extensions now support a query such as: `SELECT * FROM table ORDER BY array_distance(emb, "[0.1, 0.2]") LIMIT 10;`. In this query, the `[0.1, 0.2]` is replaced with a stringified/blobified `"[0.1, 0.2]"`. This single unit passes right through the query preprocessing. All the user has to do when running a query is to stringify/concatenate in their application-level environment (e.g., Python) and then pass it as

¹The similarity searches discussed thus far find k embeddings similar to a single query embedding. This is achieved through an exact or approximate 1:N comparison between the query embedding and the N embeddings in the dataset. Another not-yet-discussed VSS workload is called a similarity join. This is a batch processing workload where the search is performed for M query embeddings at once (where $M > 1$), leading to an M:N join [2, 88]. It is also known as a lateral join in databases and is supported by DuckDB’s VSS and FAISS extensions.

²The pull request that added blob interface support to the PDXearch extension can be found at: <https://github.com/Noorts/PDXearch/pull/16>.

³The commit that added blob interface support to our DuckDB VSS fork can be found at: <https://github.com/Noorts/duckdb-vss/commit/9464e55b02be616ca9bfc7024c917bd3de661fe0>.

5. PDXSEARCH ROW GROUP IMPLEMENTATION

a string to DuckDB. This is a simplified explanation — in reality, we currently use the BLOB type along with a 16-bit integer encoding. This blob interface is a temporary workaround. In the future, we will want to address the root problem.

Both the standard interface (with the overhead) and the new blob interface will be used during the benchmarking experiments. The experiment discussed in Section 6.3.1 quantifies the DuckDB overhead.

6

Evaluation

In this chapter, we evaluate the cost and (filtered) search performance of the indexes. We focus on the performance of the baseline global implementation and the row group implementation, and evaluate the scalar quantization. In the first section, we define our benchmarking setup.

6.1 Experimental Setup

6.1.1 Hardware

Measurements were taken on a 16-inch Apple MacBook M4 Pro, equipped with 10 performance cores (up to 4.5 GHz), 4 efficiency cores (up to 2.6 GHz), 48 GiB of RAM, and 1 TB of storage [89, 90]. It runs macOS Sequoia 15.6.1 and supports ARM NEON for SIMD processing. During benchmarking, macOS' high-power mode is enabled, and we use an external power supply.

In terms of multithreading, we use 1 and 14 threads in the experiments. Using 1 thread is common for performance comparisons in VSS [8, 10, 91] and systems evaluations [92]. We include 14 threads as well to evaluate the systems' intra-query parallelism. The machine will use 4 efficiency cores and 10 performance cores. Gaining insight into the speedup factor by trying an intermediate number of threads (e.g., to target 4 performance cores) is interesting but is out of scope.

6.1.2 Systems

We use six VSS systems in our evaluation. Table 6.1 provides an overview of their characteristics. Although pgvector supports quantization (Section 3.2), we do not use it in the evaluation to reduce the number of configurations to test. In future evaluations, we will include more systems and configurations (Section 7.2). We have optimized the systems to the best of our knowledge, following their optimization guidelines and recommendations.

6. EVALUATION

We use DuckDB’s v1.4.4 Python client. For the DuckDB VSS extension, we rely on the precompiled version¹, available through the `INSTALL VSS; SQL statement [37]`. The DuckDB PDXearch extension is built using LLVM Clang 18 and the default release mode on the same machine used for benchmarking.

We start the PostgreSQL (pgvector) server process using the 0.8.2-pgvector (PostgreSQL 18) Docker image, running locally on the benchmarking machine. The Docker container and PostgreSQL and pgvector server parameters were tuned using PG Tune² and the pgvector authors’ advice³. The complete setup details can be found in this thesis’ benchmarking repository⁴.

¹Ad hoc experiments have shown that the performance difference between the precompiled and locally compiled version is negligible. Furthermore, enabling SIMSIMD and OpenMP inside VSS’ USearch did not have a meaningful performance impact on the Apple M4 machine (ARM NEON SIMD). The extension’s build configuration uses optimization level `-O3` by default, which enables auto-vectorization.

²See the `postgresql.conf` file for the adjusted settings: <https://github.com/Noorts/vss-benchmarks/blob/de02bfd00bb81ca7ed472be5d854f53b7203ffbb/experiments/postgresql.conf#L886-L912>.

³The pgvector advice we used to tune the experimental setup is summarized in the following commit’s description: <https://github.com/Noorts/vss-benchmarks/commit/a0adc20fe2e89fdd201d6221c55d23ea429dda13>.

⁴See the `docker-compose.yml` and `postgresql.conf` files in the repository for the full benchmarking configuration: <https://github.com/noorts/vss-benchmarks>.

Table 6.1: Characteristics of all the systems used in the evaluation. Compiled from the previous chapters.

	pgvector IVF [38]	pgvector HNSW [38]	DuckDB [19, 59]	DuckDB VSS [20]	DuckDB PDXearch Global [45]	DuckDB PDXearch Row Group [45]
System type	Client-server, row-oriented, general-purpose	Client-server, row-oriented, general-purpose	In-process, columnar, OLAP	In-process, columnar, OLAP	In-process, columnar, OLAP	In-process, columnar, OLAP
Index type	IVF	HNSW	–	HNSW	IVF	IVF per row group
Integration	Post-filtering with iteration (pre-filtering with partial index or partitioning)	Post-filtering with iteration (pre-filtering with partial index or partitioning)	Pre-filtering	Post-filtering (without iteration)	Pre-filtering	Pre-filtering (with row group pruning)
System description	Mature feature-rich VSS system with an iterative post-filtered lightweight IVF index.	Mature feature-rich VSS system with an iterative post-filtered heavy HNSW index.	Portable analytical VSS system with a brute force exact vector similarity search.	Portable analytical VSS system with an experimental post-filtered heavy HNSW index.	Portable analytical VSS system with an experimental pre-filtered global lightweight IVF index.	Portable analytical VSS system with an experimental pre-filtered lightweight per row group IVF index.
Optimal use-case	For non-filtered or filtered queries that eliminate few tuples serving production workloads on up to the largest datasets where data management is important (the IVF index can handle the scale efficiently).	For non-filtered or filtered queries that eliminate few tuples serving production workloads on small to large datasets where throughput is important and large index size and long index construction are feasible.	For filtered queries that eliminate almost all tuples filtered data analytics workloads where the fast predicate evaluation can discard rows until only a few are left on which to run the expensive (ENNS) exact search.	For non-filtered or filtered queries that eliminate very few tuples data analytics workloads on small to medium datasets where the HNSW index can deliver high throughput without being hindered by the large index and lack of iterative index scans (leading to recall loss).	For data analytics workloads with filters that eliminate most tuples on up to large datasets where data management (e.g., maintenance) and portability are important (the IVF index can handle the scale efficiently).	For data analytics workloads with filters that eliminate most tuples on up to large datasets where data management (e.g., maintenance) and portability are important (the IVF index can handle the scale efficiently; however, search will likely be hindered by too many RGs, so RG pruning is key).
Parallelize a single query (intra-query parallelism)	Construction: Parallel Index search: Serial	Construction: Parallel Index search: Serial	Construction: – Index search: Parallel	Construction: Parallel Index search: Serial	Construction: Parallel Index search: Serial	Construction: Parallel Index search: Parallel (per row group)
Limitations of the current implementation	• Low storage efficiency due to fragmentation • Limited dimensionality at full precision	• Low storage efficiency due to fragmentation • Limited dimensionality at full precision	• Only supports exact search (ENNS)	• No iterative index scans (hence post-filtering frequently leads to low recall) • No index quantization • Does not support arbitrary predicates	• No persistence • No maintenance • No concurrency • Does not support arbitrary predicates • Row groups must be full	• No persistence • No maintenance • No concurrency • Does not support arbitrary predicates • Row groups must be full

6. EVALUATION

6.1.3 Datasets

Table 6.2: Benchmarking datasets. * = custom dataset only available in the VectorDBBench fork [93], and which is discussed in more detail in Appendix A.

Name	Dims	Embeddings	Size in GB	Distance Metric	Type	Attributes
SIFT* [94, 95, 96]	128	4 999 000	2.56	L2	Image	Synthetic
Cohere [96, 97]	768	1 000 000	3.07	Cosine	Text	Synthetic
Agnews* [33, 98, 99, 100]	1024	769 382	3.15	L2	Text	Synthetic
OpenAI [96, 101]	1536	500 000	3.07	Cosine	Text	Synthetic
arxiv-for-fanns* [8, 102]	1024	1 200 000	4.92	L2	Text	Real
Cohere6M* [96, 97]	768	6 021 120	18.50	Cosine	Text	Synthetic

Table 6.2 shows the datasets used. We focus on medium-scale workloads that are commonly used for VSS benchmarks and that can run on modern laptops [8, 9, 21, 103]. We use a set of datasets that covers a range of dataset properties (e.g., number of dimensions, number of embeddings, embedding type). Although realistic dataset attributes are preferable [8], we have had to use datasets with synthetic attributes for more control over the filtering percentage.

Two of the six datasets are available out of the box in the VectorDBBench tool (introduced in Section 6.1.5). The other four, marked with a “*” in Table 6.2, were assembled for the thesis and are available in the thesis’ VectorDBBench fork [93]. We discuss how they were assembled in Appendix A.

The search queries and the pre-computed top- k ground truth come included with the datasets. The distance metric is either Euclidean distance or cosine distance and was chosen based on the dataset’s precomputed ground truth.

6.1.4 Index parameters

As discussed in Chapter 2, indexes have parameters (e.g., `ef_construction`, `ef_search`, `lists`, `n_probe`), which affect the construction time, index size, (filtered) search latency, and search quality (recall). These parameters are strongly intertwined and dataset dependent, and the combination of these parameters creates a large parameter space.

Benchmarks sweep over this parameter space. Some argue that a thorough exploration of the entire parameter space (including construction parameters) is the key to reliable evaluations [8, 12, 21]. They depict the Pareto frontier to show the algorithm/index’s characteristics fully. However, many take a simpler approach [11, 24, 103]. They use one index construction configuration (e.g., size of the graph, number of partitions) to build the index once. Then they only explore the runtime parameter space (e.g., number of vectors to examine in the graph, number of partitions to probe) to find the desired balance between latency and recall.

6.1 Experimental Setup

In this paper, we use this simpler approach, setting the construction parameters based on heuristics (i.e., recommendations). We believe that this approach is sufficient for this thesis as it mimics practical data science workloads. We explore the construction parameter space in future work (Section 7.2).

To increase fairness, we use the same parameters for the same index types in different systems. For example, the IVF indexes of PDXearch (global) and pgvector use the same number of partitions.

For the pgvector IVF and PDXearch global IVF, we determined the number of lists to use based on advice from pgvector [38]. Given N embeddings in a dataset, we define the number of lists as:

$$\text{number of lists} = \begin{cases} \frac{N}{1000}, & \text{if } N \leq 1\,000\,000 \\ \sqrt{N}, & \text{else} \end{cases}$$

PDXearch’s row group approach uses 1 partition for every 256 embeddings, resulting in 480 partitions per full row group.¹ We set this to 256 embeddings based on the default maximum per partition in Super K-Means [28], which mimics the approach used by the Faiss library [15]. Table 6.3 shows the number of lists this results in across the datasets.

Table 6.3: IVF index properties and the number of lists used across the datasets.

Dataset	Dims	Embeddings	Number of lists used by...	
			pgvector IVF, PDXearch Global	PDXearch Row Group
SIFT	128	4 999 000	2236	41 row groups $\times$ 480 = 19 680
Cohere	768	1 000 000	1000	9 row groups $\times$ 480 = 4320
Agnews	1024	769 382	770	7 row groups $\times$ 480 = 3360
OpenAI	1536	500 000	500	5 row groups $\times$ 480 = 2400
arxiv-for-fanns	1024	1 200 000	1096	10 row groups $\times$ 480 = 4800
Cohere6M	768	6 021 120	2454	49 row groups $\times$ 480 = 23 520

At runtime we sweep over PDXearch’s $n_probe = 1, 2, 3, 4, 5, 8, 16, 32, 64, 128, 256, 384, 512, 768, 0^2$. The pgvector IVF is slower and therefore uses a smaller n_probe sweep, $n_probe = 1, 2, 3, 4, 5, 8, 16, 32, 64, 128$. Fortunately, that is sufficient as pgvector reaches close to 100 % recall in the experiments.

For the HNSW indexes — both pgvector HNSW and DuckDB VSS use DuckDB VSS’ defaults. The construction parameters are $ef_construction = 128$ and $m = 16$. Both systems set the connectivity of the base layer (m_0) to $m_0 = 2 \times m = 32$.

¹In some of our benchmarking datasets, the last row group is partially full. This row group also uses 480 partitions. Newer versions of the extension improve this behavior by dynamically setting the number of partitions per row group based on its size.

²Setting $n_probe = 0$ in PDXearch means that it will probe all clusters in the index.

6. EVALUATION

6.1.5 Metrics

We use **VectorDBBench** [96]. This is a widely used [3, 96] open-source VSS benchmarking tool offered by Zilliz [104] — the company behind the Milvus VDBMS. One of the major reasons we use VectorDBBench instead of alternatives [105, 106, 107, 108, 109, 110] is because it is widely used, and supports filtered search workloads and many systems (e.g., pgvector) out of the box. We extend VectorDBBench¹ to provide benchmarking support (i.e., a client) for DuckDB itself and the VSS and PDXearch extensions, add support for the datasets discussed above, and make several improvements.

For the index cost experiments, we measure the index construction time and the index size. The **index construction time** is the time in seconds that it takes from the submission of the `CREATE INDEX ...` statement to it returning, indicating that the index is ready to use. The **index size** is the in-memory size of the index reported by the system itself.

For the (filtered) search experiments, we report VectorDBBench’s **serial throughput**, that is, the **queries per second (QPS)** that is achieved when a single database client sends search queries to the VSS system synchronously one by one. We define serial QPS as the multiplicative inverse of the median query latency:

$$\text{serial QPS} = \frac{1}{\text{median latency}}$$

The latency is the median (of the dataset’s 1000 queries) wall clock time in seconds that it takes for a search query to be executed and answered in a Python environment with a pre-established database connection. This serial search is an alternative to VectorDBBench’s default multi-process concurrent search runner, which will try to saturate a VSS system by sending queries from multiple database connections concurrently. In practice, the concurrent search runner can achieve much higher throughput for systems such as pgvector². Nevertheless, we report serial search because it is common in VSS benchmarking literature [8], and because our PDXearch extension currently lacks support for concurrency. In addition, we include the **bootstrapped 95 % confidence interval (CI)** of this median latency³. Finally, we show the mean **recall@10** ($K = 10$).

¹The extended VectorDBBench fork can be found at: <https://github.com/noorts/vectorddbbench>.

²In ad hoc testing on the OpenAI (999K, 1536) dataset, pgvector achieved 1289 QPS when searching concurrently and only 129 QPS when searching serially.

³We use confidence intervals, but we do not use statistical *tests* to reinforce our findings. The use of such tests is rare in the VSS literature.

We ensure that the table and a system’s index are loaded into memory before we run any (filtered) search benchmarks. Loading the table and index from persistent storage is thus not included in the first measurement, avoiding outliers. This applies to all systems and increases fairness¹.

6.1.6 Artifacts

In addition to the PDSearch extension [45] and VectorDBBench fork [93], we have open sourced the scripts that sweep over the construction parameter space and collect the measurements, the measurements themselves (in JavaScript Object Notation (JSON) files), and the plotting notebooks². In addition to the measurements, these JSON files also store the specific index parameters and the software versions that were used (including a commit hash of the PDSearch extension).

6.1.7 QPS–recall plots

In this section, we briefly explain the QPS–recall curve plots that we will use in the evaluation. Figure 6.1 shows an example plot where $K = 10$ (hence “Recall@10”). These plots are common in VSS benchmarking. They provide insight into the query throughput (QPS) vs. result quality (recall) tradeoff [8, 21].

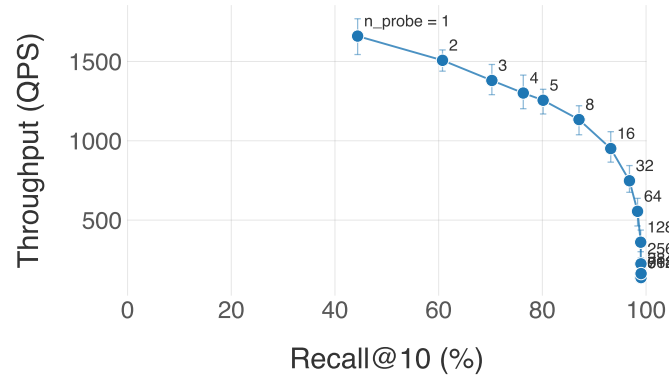


Figure 6.1: QPS–recall curve example plot. The numbers next to the points indicate the `n_probe` parameter value that was used during that benchmarking run. The vertical error bars indicate the 95 % CI for the serial throughput (QPS).

¹PDSearch currently does not implement persistence. Hence, the PDSearch index will always be constructed before a (filtered) search benchmark run. This is not the case for the other systems, which could suffer a “load from persistence” time penalty on the first search query, thus skewing the results in PDSearch’s favor. DuckDB’s VSS extension incurs 0.5–1 seconds of load time depending on the index size. We added the `-force-load-index` option to VectorDBBench, which will execute a dummy query on the table before the benchmark run. This is enabled for all experiments.

²The experimental setup and measurements are archived at: <https://github.com/noorts/vss-benchmarks>.

6. EVALUATION

Each point represents a benchmark run of 1000 queries under one configuration of index parameters. Both the x-axis and y-axis are outputs of a benchmark run. Simplified, one could see it as $f(\text{construction_effort}, \text{runtime_search_effort}) = (\text{QPS}, \text{recall})$. The top-leftmost point in the plot is where the index was configured to spend as little computational effort as possible during the index search (i.e., $n_{\text{probe}} = 1$, that is, probe only one cluster in the IVF index). This leads to high throughput (low latency), but low recall. As we increase the computational effort (i.e., n_{probe}), the throughput decreases, but recall increases. As we get close to 100 % recall the increased effort shows diminishing returns.

As discussed in Section 6.1.5, each point vertically indicates the throughput (computed based on the median latency of 1000 queries) and horizontally indicates the recall (the mean recall of the 1000 queries). We also see the aforementioned 95 % CI for the serial throughput as vertical error bars. In the experiments, these error bars are often small enough to be hidden behind the main point. Now that we have discussed the experimental setup, in the next sections, we dive into the index cost, non-filtered search, and filtered search experiments.

6.2 Index Overhead

Indexes speed up the search but are not free; they come at a cost. In the next two experiments, we evaluate this cost in terms of in-memory size and construction time.

6.2.1 Exp. 1 – In-memory size

Setup This experiment investigates the indexes' size in memory. The size was manually requested from each system, using a system-specific metadata SQL statement (e.g., `CALL pdxsearch_index_info()`). We collected only a single measurement per index type, which we argue is sufficient as index size variance should be insignificant. Beware that the reported index size is an approximate lower bound in bytes and is highly dependent on the system's implementation.

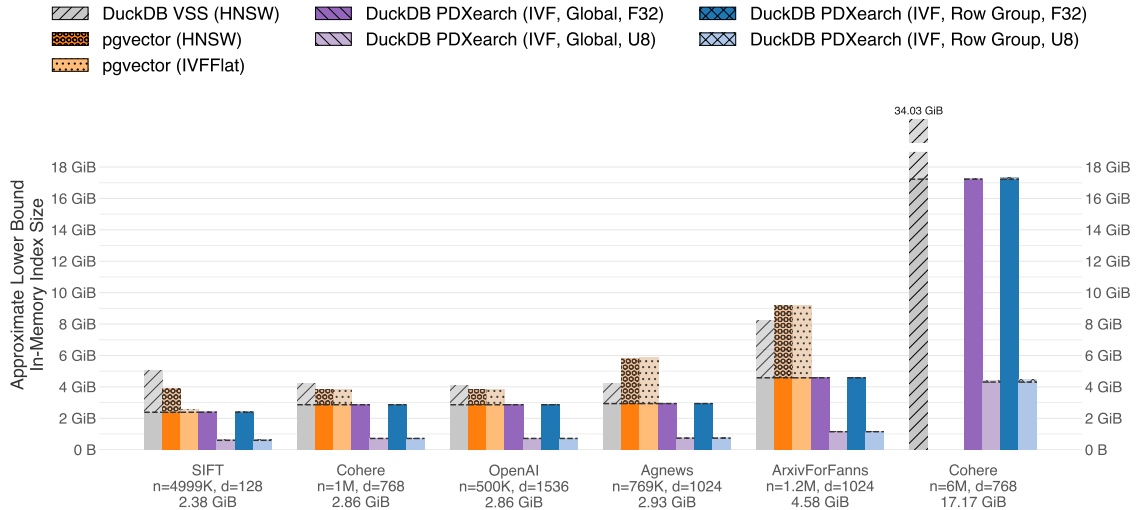


Figure 6.2: Exp. 1 – Index size. An approximate lower bound in-memory index size in bytes. The measurements are what the system reports through an index metadata query (e.g., `CALL pdxsearch_index_info()`). Each bar represents one measurement. The bar up to the dashed line shows the amount of memory the embeddings themselves take up in the index. The bar's length above the line is known as the index's overhead (total size - size of embeddings). The DuckDB VSS Cohere6M measurement was shortened, with the total size included as a text label. Table 6.4 contains the individual measurements.

Analysis Figure 6.2 depicts the results. The dashed line, along with the portion of the bar below it, indicates the memory consumed by the raw embeddings within the index. The remainder of the bar, above the line, is overhead that the index type or implementation requires (e.g., graph edges). The raw measurements are available in Table 6.4.

pgvector's IVF suffers from high memory usage. As mentioned in Section 2.2.4, an IVF index is expected to consume significantly less memory than an HNSW index. A look at our

6. EVALUATION

results and the pgvector source code revealed that both pgvector’s IVF and HNSW index suffer from internal fragmentation. pgvector persists the embeddings on 8 KiB pages. Each page stores metadata and a single full-precision (4-byte) embedding with at most 2000 dimensions. In our experiment, this affects the ArxivForFanns and Agnews datasets the most. Those datasets have 1024 dimensions, which means that an embedding is just slightly too large to fit two on a single page. This leads to roughly 49 % internal fragmentation, effectively doubling the amount of memory pgvector’s indexes use¹.

Table 6.4: Approximate lower bound of in-memory index size across datasets and systems.

System		SIFT	Cohere1M	OpenAI	Agnews	ArxivForFanns	Cohere6M
Raw embeddings (F32)		2.38 GiB	2.86 GiB	2.86 GiB	2.93 GiB	4.58 GiB	17.23 GiB
VSS (HNSW)	<i>Overhead</i>	2.64 GiB	1.38 GiB	1.26 GiB	1.31 GiB	3.67 GiB	16.81 GiB
	<i>Total</i>	5.03 GiB	4.25 GiB	4.12 GiB	4.24 GiB	8.25 GiB	34.03 GiB
	<i>Overhead %</i>	52.6 %	32.6 %	30.5 %	30.8 %	44.5 %	49.4 %
pgvector (HNSW)	<i>Overhead</i>	1.49 GiB	976.26 MiB	976.57 MiB	2.89 GiB	4.57 GiB	–
	<i>Total</i>	3.87 GiB	3.81 GiB	3.81 GiB	5.83 GiB	9.15 GiB	–
	<i>Overhead %</i>	38.5 %	25.0 %	25.0 %	49.6 %	50.0 %	–
pgvector (IVFFlat)	<i>Overhead</i>	172.09 MiB	982.40 MiB	980.48 MiB	2.94 GiB	4.59 GiB	–
	<i>Total</i>	2.55 GiB	3.82 GiB	3.82 GiB	5.88 GiB	9.16 GiB	–
	<i>Overhead %</i>	6.6 %	25.1 %	25.1 %	50.1 %	50.0 %	–
PDXearch (Global, F32)	<i>Overhead</i>	58.35 MiB	14.40 MiB	8.66 MiB	11.83 MiB	18.04 MiB	76.15 MiB
	<i>Total</i>	2.44 GiB	2.88 GiB	2.87 GiB	2.95 GiB	4.60 GiB	17.30 GiB
	<i>Overhead %</i>	2.3 %	0.5 %	0.3 %	0.4 %	0.4 %	0.4 %
PDXearch (Global, U8)	<i>Overhead</i>	58.35 MiB	14.40 MiB	8.66 MiB	11.83 MiB	18.04 MiB	76.15 MiB
	<i>Total</i>	668.58 MiB	746.82 MiB	741.09 MiB	763.18 MiB	1.16 GiB	4.38 GiB
	<i>Overhead %</i>	8.7 %	1.9 %	1.2 %	1.6 %	1.5 %	1.7 %
PDXearch (Row Group, F32)	<i>Overhead</i>	67.57 MiB	25.01 MiB	20.71 MiB	22.70 MiB	32.81 MiB	138.36 MiB
	<i>Total</i>	2.45 GiB	2.89 GiB	2.88 GiB	2.96 GiB	4.61 GiB	17.36 GiB
	<i>Overhead %</i>	2.7 %	0.8 %	0.7 %	0.7 %	0.7 %	0.8 %
PDXearch (Row Group, U8)	<i>Overhead</i>	67.57 MiB	25.01 MiB	20.71 MiB	22.70 MiB	32.81 MiB	138.36 MiB
	<i>Total</i>	677.80 MiB	757.43 MiB	753.13 MiB	774.05 MiB	1.18 GiB	4.44 GiB
	<i>Overhead %</i>	10.0 %	3.3 %	2.8 %	2.9 %	2.7 %	3.0 %

The DuckDB extensions currently do not include persistence/fragmentation overhead. DuckDB VSS has persistence but reports the size reported by the USearch library inside, which is based on one contiguous memory allocation. Currently, DuckDB PDXearch does not support persistence (Section 7.3). Hence, PDXearch simply reports the size of its in-memory data structures. Future work will investigate how persistence affects the index size and performance of the DuckDB VSS and DuckDB PDXearch extensions.

¹There are at least two ways to address the observed fragmentation. Firstly, pgvector supports lower precision (quantized) embedding storage (< 4 bytes) [38]. Secondly, it might be possible to change PostgreSQL’s page size (known as BLCKSZ) from 8 KiB up to 32 KiB, thus reducing the internal fragmentation percentage. It is unclear whether pgvector supports this (<https://github.com/pgvector/pgvector/issues/248>). We consider both these approaches out of scope for this thesis.

Next, we focus on the DuckDB solutions. As expected, in the majority of cases, VSS' HNSW index takes up significantly more memory than the PDXearch IVF variants. This is because of the graph structure. In Table 6.4, we can see VSS' HNSW data structure overhead range from 30.5 % to 52.6 %. By tweaking the HNSW construction parameters, this overhead could be reduced (at the loss of recall), but it is clear that by nature the IVF approach is more suitable for memory-constrained systems. The PDXearch row group (F32) approach is 30.0 % to 51.3 % smaller than the DuckDB VSS index.

Given that PDXearch's internal IVF index stores only centroids and metadata, it is not a surprise that the overhead is low. As shown in Table 6.4, the PDXearch overhead scales with the number of embeddings (and thus the number of centroids) in the dataset. The PDXearch overhead ranges from 8.66 MiB (Global, OpenAI) to 138.36 MiB (Row Group, Cohere6M). For these datasets, the row group variant has a larger total number of clusters and centroids than the global variant, which slightly increases the row group approach's overhead. The PDXearch global and row group approaches differ only slightly (~10 MiB) in in-memory use. The row group approach uses a larger total number of lists and thus stores more centroids.

PDXearch's quantized (U8) row group index clearly demonstrates the value of quantization. This approach uses 82.1 % to 86.9 % less memory than the DuckDB VSS index. However, as discussed in Section 2.3, this comes at a recall cost. Note that quantization (F32 vs. U8) does not affect the index overhead, as we always store the centroids at full precision (F32).

Finally, we argue that the lower index size of the IVF indexes (especially quantized U8) has two upsides. First, they make it possible to process larger datasets on a memory-constrained device (e.g., laptop) without running out of memory. Second, they allow a larger percentage of the index to stay in RAM instead of swap, thus likely improving index construction and index search throughput.

Our main findings from this in-memory index size experiment are:

Main Finding 1: pgvector's size is increased significantly due to the internal fragmentation caused by its index persistence. The DuckDB solutions do not include such an overhead.

Main Finding 2: PDXearch's IVF index size is significantly smaller than the DuckDB VSS and pgvector HNSW indexes. When storing full-sized (F32) embeddings, the PDXearch row group index is 30.0 % to 51.3 % smaller than the DuckDB VSS index. Enabling PDXearch's scalar quantization (U8) shrinks the embeddings inside the index. This incurs a recall cost but allows it to use 82.1 % to 86.9 % less memory than DuckDB VSS (F32). DuckDB VSS' HNSW could support this quantization, but it currently does not. We argue that the lower index size of the IVF indexes (especially quantized U8) makes it possible to process larger datasets and to process those more efficiently by staying in RAM.

6. EVALUATION

6.2.2 Exp. 2 – Construction time

Setup This experiment investigates the index construction time across all systems. In other words, we measure how long it takes to execute `CREATE INDEX . . .`. We constructed each index three times. The indexes’ construction parameters match those used in the upcoming (filtered) search experiments. To increase fairness between DuckDB and pgvector, the DuckDB and pgvector IVF indexes use the same index construction parameters (i.e., the number of lists/partitions/clusters). The same applies to the respective HNSW indexes (i.e., `ef_construction = 128`, `m = 16`).

The Cohere6M dataset is large. We measured its construction only for DuckDB PDXearch and (multi-threaded) DuckDB VSS. Only DuckDB PDXearch uses it in subsequent experiments. We excluded pgvector for the sake of time.

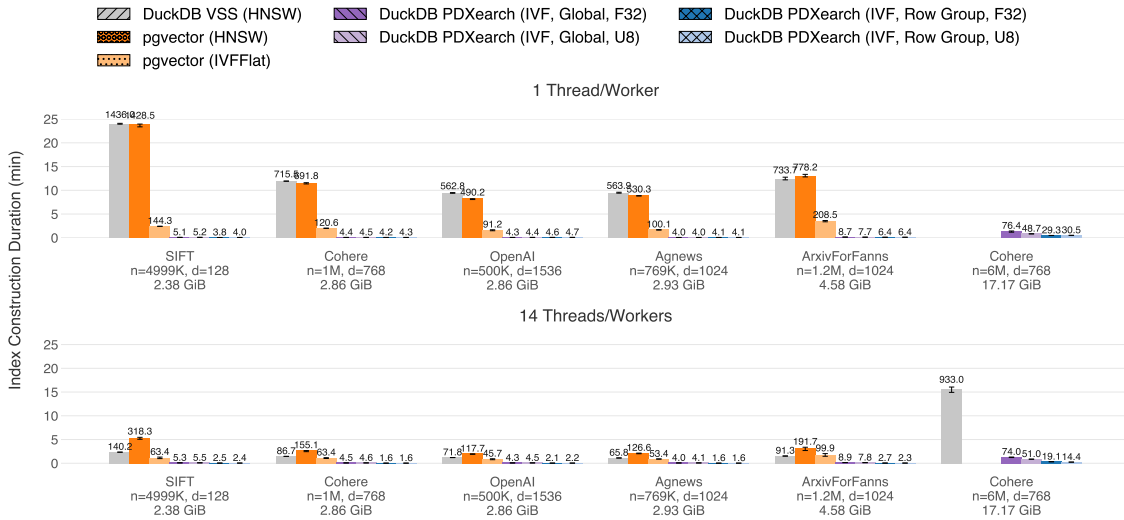


Figure 6.3: Exp. 2 – Construction time of each system’s index for the datasets used in the other experiments. Both single-threaded and multi-threaded construction are shown. Each index was constructed three times per dataset. We report the median construction time in seconds on top of each bar and depict the minimum and maximum values (i.e., range) using error bars. Hence, all three measurements are shown. The y-axis ticks are in minutes.

Analysis Figure 6.3 shows the results. Comparing single- and multi-threaded construction, DuckDB VSS shows significant speed-up, which is due to its parallelized index construction. DuckDB VSS is also better able to utilize parallelism than pgvector’s HNSW. PDXearch global shows no speed-up, and the PDXearch row group approach a less significant one. It was not possible to control the Apple M4 Pro’s Apple Matrix Coprocessor (AMX) unit during benchmarking. This means that for the single-threaded experiments, the machine’s AMX was unbounded, resulting in fast single-threaded PDXearch construction times. By monitoring CPU utilization, we confirm that, as expected, all solutions (including pgvector) use all 14

CPUs during index construction in the multi-threaded case. We use the multi-threaded times to compare across systems fairly, such that not only PDXearch uses the whole machine.

While pgvector fully saturates the 14 CPUs during index construction, both the IVF and HNSW indexes take longer to construct than the competitors. We argue that this is because pgvector deals with the internal fragmentation discussed in the previous experiment.

We observe that the HNSW approaches take significantly longer to build than the IVF ones. As discussed in Chapter 2, by nature, a partition-based IVF index is faster to build than a graph-based HNSW index.

As mentioned, pgvector (IVFFlat) and PDXearch's global variant use the same number of partitions. However, it is clear that the PDXearch implementation is significantly faster, even in the multi-threaded scenario. The aforementioned "pgvector is slower" arguments apply. Furthermore, we argue that the K-means clustering library used in PDXearch (Super K-means) can more effectively use the Apple M4 Pro hardware.

The PDXearch row group variant clearly achieves the fastest index construction times. Comparing DuckDB VSS and DuckDB PDXearch row group construction time with 14 threads directly, PDXearch row group F32 is 33.4x to 56.9x faster to construct. The quantized row group U8 approach has to quantize the embeddings yet achieves a slightly higher speed up, 33.4x to 64.6x, when comparing to DuckDB VSS. The construction time is attractive (e.g., for interactive data scientist workloads). Even the large 17.17 GiB Cohere6M dataset can be indexed in less than 20 seconds.

We expected the row group approach to be slightly faster to construct than the single global IVF (Table 4.1). Looking at the data, only for the multi-threaded Cohere6M workload is the difference between those two significant. It appears that the smaller total number of lists (2454 vs. 23520; Table 6.3) and Super K-Means' parallelized clustering keep the global approach's index construction competitive for these datasets. Future work should explore this in more detail.

Our main findings from this index construction time experiment are:

Main Finding 3: As expected, IVF indexes are significantly faster to construct than the HNSW indexes. In particular, the PDXearch row group index (IVF) is 33.4x to 56.9x faster to construct than DuckDB VSS' HNSW index. With PDXearch's U8 scalar quantization, this changes to a 33.4x to 64.6x speedup. The fast index construction we expected from the IVF has held up, staying below 20 seconds for datasets of up to 17.17 GiB.

Main Finding 4: For most of the datasets, the global approach achieves a construction time similar to the row group approach. We expected the row group approach to be faster. We argue that the difference in the total number of lists and the parallelized construction of the global IVF ensure the global approach keeps up for datasets of this scale.

6. EVALUATION

6.3 Non-Filtered Search

In this section, to establish the indexes' baseline performance, we evaluate non-filtered search.

6.3.1 Exp. 3 – Normal interface

Setup This experiment investigates the non-filtered search performance across four datasets. The systems can use 14 workers/threads. We aim to gain insight into the DuckDB overhead (present when using the normal interface) discussed in Section 5.8.3. That section also introduced a workaround: the new blob interface.

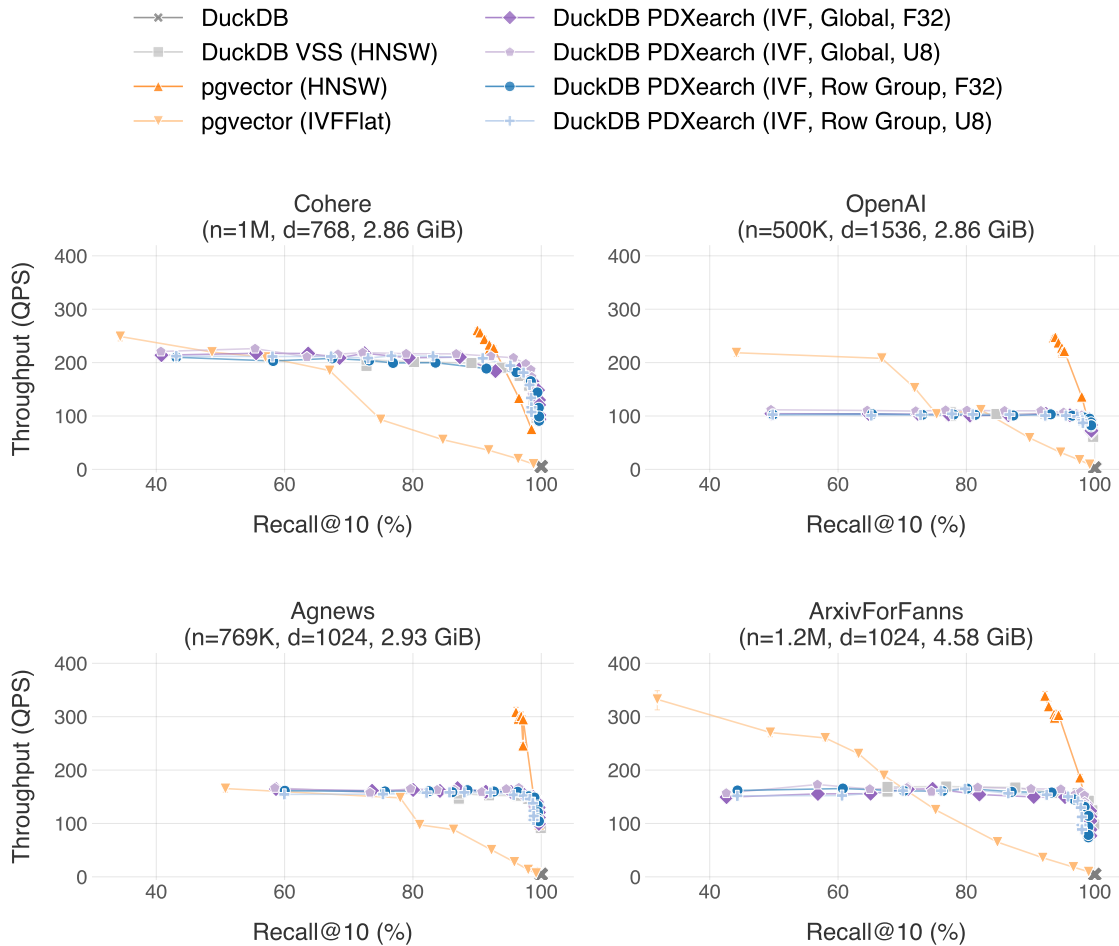


Figure 6.4: Exp. 3 – Non-filtered search comparison of all systems. Normal interface with DuckDB overhead. Each point represents the median QPS of 1000 search queries. The vertical error bars indicate the bootstrapped 95 % confidence interval. The recall is the mean of the 1000 queries. Multi-threading is allowed; that is, the systems can use at most 14 workers/threads.

6.3 Non-Filtered Search

Analysis The DuckDB systems' results are shown in Figure 6.4. The QPS around 80 % recall ranges roughly from 100 to 200 QPS. For interactive data scientist workloads on datasets such as these, this performance might suffice. However, we know the underlying libraries are capable of significantly higher QPS for that workload on this hardware. Furthermore, as mentioned in Section 5.8.3, the DuckDB extensions suffer from overhead caused by DuckDB's planner and optimizers.

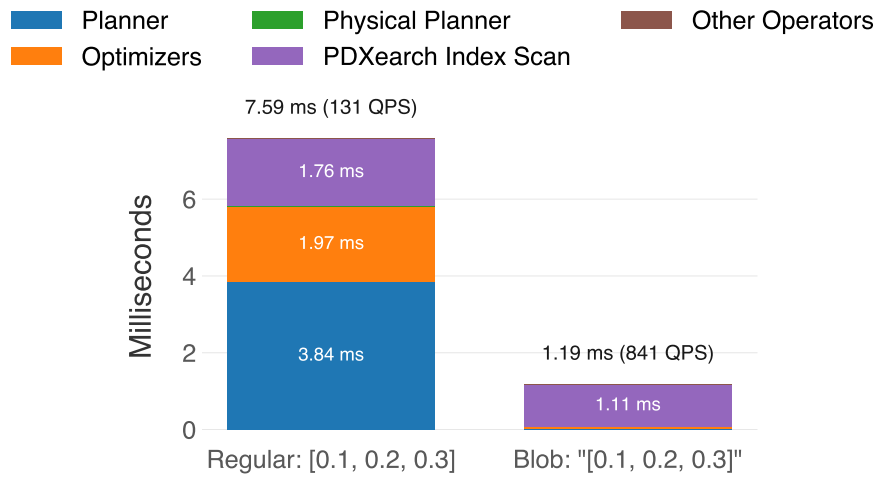


Figure 6.5: Exp. 3 – Non-filtered search time decompositions of two setups. One setup with the standard DuckDB interface and one setup with a new blob interface that significantly decreases query preprocessing overhead. The timings are the mean across 1000 single-threaded search queries on the ArxivForFanns ($1.2M \times 1024$) dataset. With $k = 10$ and $n_probe = 5$ we achieve a recall of at least 80 %.

To gain insight into this overhead, we collected fine-grained timings for a specific PDXearch row group configuration on the ArxivForFanns dataset. Figure 6.5 shows the decomposed search latency. We observe that the time spent in the optimizers and planner far exceeds the time spent in our index search operator. As shown on the right, with the new blob interface approach, the overhead becomes negligible. Going forward, the experiments will all use the new blob interface. In the next experiment, we will dive deeper into the systems' index search performance.

Our main finding from this normal interface non-filtered search experiment is:

Main Finding 5: As expected, the standard VSS interface in DuckDB suffers from significant overhead. This affects both DuckDB VSS and DuckDB PDXearch. It results in both of those solutions reaching a similar QPS, between 100 and 200, which one could argue is sufficient for some workloads (e.g., exploring a dataset). Regardless, in subsequent experiments we use the new blob interface, which practically eliminates the overhead.

6. EVALUATION

6.3.2 Exp. 4 – Blob interface

Setup This experiment investigates the non-filtered search performance across five datasets. It is similar to the previous experiment, except that the DuckDB solutions use the blob interface, and the Cohere6M dataset is now also included. We include both single-threaded and multi-threaded performance.

As mentioned, the construction parameters (e.g., the number of lists) used were set based on heuristics. To be more specific, the PDXearch global and pgvector IVFFlat follow advice from the pgvector authors. The PDXearch row group approach aims for a ratio of 256 embeddings per cluster. The focus of this experiment is on the performance of the indexes when standard heuristics are used to set the number of lists. As a data scientist, it is reasonable to use such recommendations instead of running a lengthy and expensive parameter search. As mentioned in the experimental setup, the performance is highly interdependent on the dataset and the system itself.

We exclude the SIFT dataset from the (filtered) search experiments. A bug in the new blob interface’s encoding (Section 5.8.3) causes an erroneous mapping. Hence, it currently cannot be used to handle the large numbers that make up SIFT’s embeddings.

Analysis Figure 6.6 and Figure 6.7 show the results. First we focus on the “×” depicted in the bottom right of each plot. This represents DuckDB’s exact search performance, that is, the throughput achieved without an index. DuckDB computes the distance between the query embedding and each one of the embeddings in the dataset. Using 14 threads, DuckDB’s exact search achieves 5.2 to 2.5 queries per second across these datasets (Cohere6M excluded), with 100% recall. It is clear that indexes provide a significant speed-up.

Secondly, looking at the DuckDB systems’ results, we clearly see the recall curve trade-off that was introduced in Section 6.1.7. As we sweep across the runtime parameter space (n_probe , ef_search), we increase the runtime search effort, thus decreasing the QPS and increasing the recall. For the leftmost point of each line, a low runtime effort was used ($n_probe = 1$, $ef_search = 4$). Note that for the PDXearch row group variant, in total we will probe “ $n_probe \times \text{number of row groups}$ ” clusters. It is interesting to see that probing a single or few clusters in the PDXearch IVFs yields a $> 40\%$ mean recall across the 1000 search queries.

As expected, the scalar quantization (U8) used by the PDXearch variants increases their QPS at a slight recall cost. If the recall loss is a concern, then in many cases increasing n_probe can make up for it while still achieving higher QPS due to the quantization. The only downside is that U8 does not reach as high a recall as F32. Whether this matters depends on the use case.

6.3 Non-Filtered Search

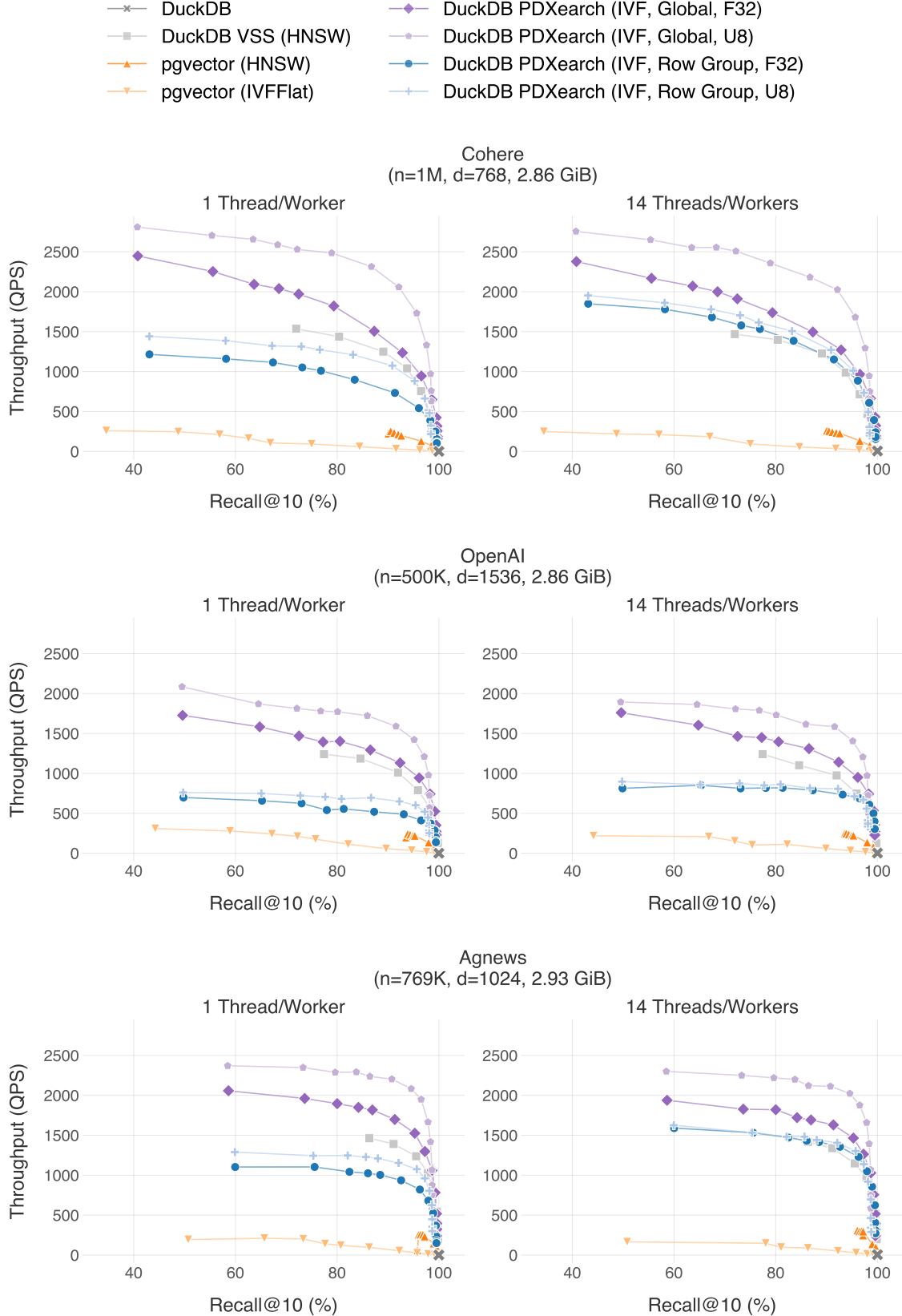


Figure 6.6: Exp. 4 – Non-filtered search comparison of all systems. DuckDB systems use the blob interface. Each point represents the median QPS of 1000 search queries. The vertical error bars indicate the bootstrapped 95% confidence interval. The recall is the mean of the 1000 queries. Continued in Figure 6.7.

6. EVALUATION

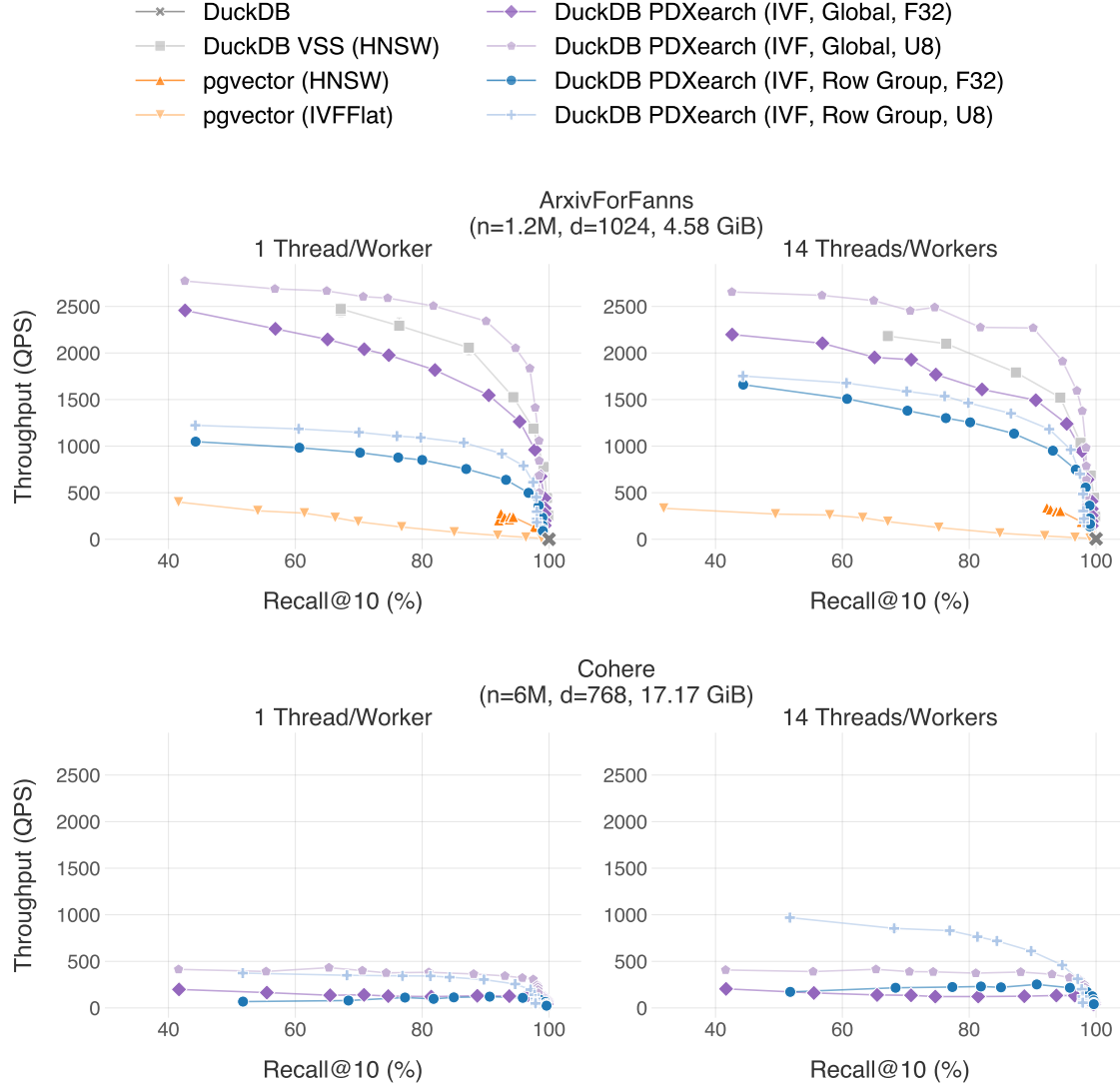


Figure 6.7: Exp. 4 – Non-filtered search comparison of all systems. Continues Figure 6.6. DuckDB systems use the blob interface. Each point represents the median QPS of 1000 search queries. The vertical error bars indicate the bootstrapped 95 % confidence interval. The recall is the mean of the 1000 queries.

PDXearch's row group approach was designed for parallel search, and it clearly benefits from it. However, the parallel efficiency is not high; that is, the 14 threads do not result in a 14x speed-up. Future work should explore the row group approach's parallelization overhead.

Since the PDXearch global approach is single-threaded, it does not benefit from multi-threading. As shown by the results, often the multi-threaded case actually has slightly lower performance. The slightly reduced QPS is due to unused thread overhead.

When comparing the PDXearch row group and global variants, global clearly achieves higher QPS. From the design chapter, we know that the row group approach has a different total number of lists. In global, nearby embeddings are part of the same partition, whereas in the row group approach, they are forced to be spread across partitions due to the row groups. This leads to more overhead for the row group approach's search. Across the datasets, at approximately 90 % recall and single-threaded, the row group approach incurs a QPS decrease ranging from 4.3 % (Cohere6M) to at the worst 58.7 % (ArxivForFanns) compared to the global approach. Multi-threaded, the QPS gap slims down and ranges from -89.3 % (Cohere6M; an increase) to at worst a 36.3 % (ArxivForFanns) decrease. As shown, in the largest dataset, Cohere6M, when using multiple threads, the row group approach does come out ahead. Hereby, future work should investigate how the two approaches scale.

We expect DuckDB VSS' single-threaded HNSW to achieve higher throughput than the PDXearch IVFs most of the time. Across the datasets, as expected, the row group approach is slower than or meets the performance of DuckDB VSS. This is still a solid result, given the IVF's construction time, memory use, and maintenance benefits. The quantized U8 global approach achieves surprising throughput, beating DuckDB VSS' HNSW on all datasets. Again, we should investigate how the approaches scale as part of future work.

Finally, compared to the DuckDB systems, pgvector's throughput is significantly lower. Its HNSW index is faster than its IVF index, which is correct. The single-threaded HNSW cannot keep up with DuckDB VSS' single-threaded HNSW search, despite using the same construction parameters. The same applies to the single-threaded DuckDB PDXearch and pgvector IVF. As mentioned in previous experiments, we think this is partly due to internal fragmentation. Furthermore, note that we measure serial throughput in these experiments. pgvector achieves significantly higher concurrent throughput when we use its inter-query parallelism to handle queries from multiple database connections in parallel (Section 6.1.5).

6. EVALUATION

Our main findings from this blob interface non-filtered search experiment are:

Main Finding 6: Unless the use case requires reaching $\geq 97\%$ recall, PDSearch's U8 scalar quantization should be enabled. Non-filtered search throughput is increased at a negligible recall loss.

Main Finding 7: When using the default/recommended construction parameters, the PDSearch row group IVF provides non-filtered search performance competitive to DuckDB's HNSW index. The PDSearch global IVF performs even better, exceeding the HNSW index in 3 out of 4 datasets. The PDSearch library demonstrates it can contend with DuckDB VSS' USearch. IVF can be fast, in addition to IVF's other upsides.

Main Finding 8: pgvector has significantly lower serial throughput than the DuckDB solutions. It is more suitable for inter-query parallelism (processing queries from multiple connections in parallel). Furthermore, its internal fragmentation, robustness compared to DuckDB's extensions, and a potentially suboptimal pgvector benchmarking setup might be to blame.

6.3.3 Exp. 5 – Varying the number of clusters

Setup This experiment takes a closer look at the Cohere6M dataset, focusing on the effect the number of clusters has on performance. We depict the same results obtained earlier, yet this time we show only the DuckDB approaches. Furthermore, we collect measurements with a global approach that uses the same number of lists as the row group approach. Namely, both the row group approach and this *global 256/cl approach* use 1 cluster for every 256 embeddings, which leads to 480 clusters per row group and a total of 23 520 clusters for this dataset. We do this to ensure both the global approach and the row group approach compute the same number of distance calculations to the centroids. This is a significantly larger number of clusters than what we and the pgvector authors would recommend for a global IVF index. Regardless, this provides another perspective on the systems' performance. The regular global approach uses the recommended 2454 clusters. This experiment also serves as the non-filtered performance baseline for the final filtered search experiment (Section 6.4.2).

Analysis From Figure 6.8 we observe that the two global approaches differ insignificantly when using F32. When using SQ (U8) the gap becomes larger, with the 2454 clusters version pulling ahead. This seems to indicate that the new global index is weighed down by overhead; the 23 520 clusters are too much. Finally, as seen earlier, the global index's search implementation uses only a single thread and thus does not benefit from the 14 threads.

6.3 Non-Filtered Search

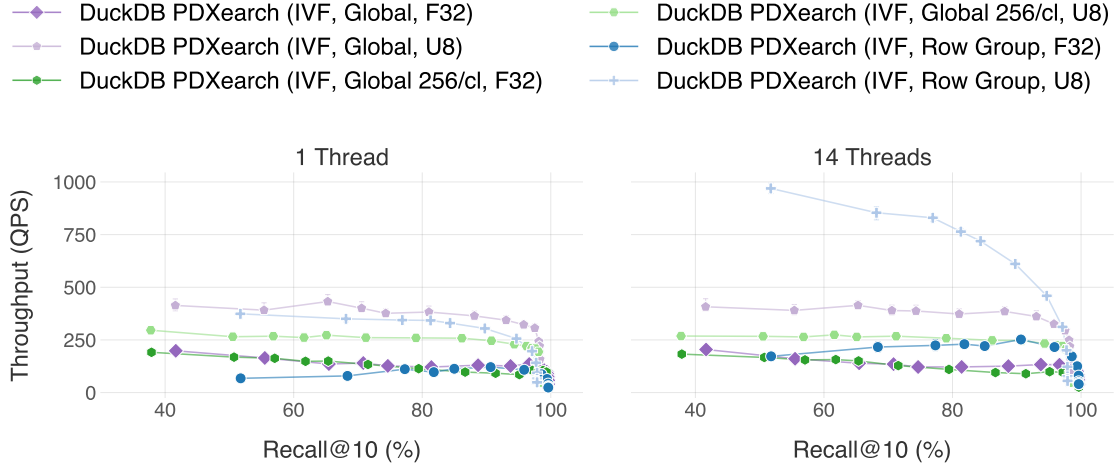


Figure 6.8: Exp. 5 – Non-filtered search comparison of DuckDB PDXearch’s row group and global approach. The normal global approach uses 2454 clusters, the “1 cluster for every 256 embeddings” global approach uses 23 520 clusters, matching the total number of clusters used by the row group approach. This is the standard shuffled Cohere6M (17.17 GiB) dataset. Each point represents the median QPS of 1000 search queries. The vertical error bars indicate the bootstrapped 95 % confidence interval. The recall is the mean of the 1000 queries.

The F32 row group approach shows lower QPS between 50 %–80 % recall, which remains unexplained. Single-threaded, the F32 row group approach performs similarly to the global approaches. As expected, multi-threading greatly benefits the parallelized index search in the row group approach. Around 90 % recall the PDXearch row group approach achieves approximately 200 QPS on this 17.17 GiB dataset.

Our main findings from this Cohere6M non-filtered search experiment are:

Main Finding 9: When using U8 quantization, the global 256/cl approach (23 520 clusters) achieves lower QPS compared to the regular global approach (2454 clusters). ~10 times the number of clusters increases overhead, resulting in lower throughput. Under F32, there is an insignificant difference between the two approaches.

Main Finding 10: As shown in the previous experiment, the row group approach achieves single-threaded performance similar to global approaches. Multiple threads allow the row group approach to exceed the global approaches’ performance. As discussed in the previous experiment, on smaller datasets this is not the case.

6. EVALUATION

6.4 Filtered Search

In this section, we evaluate the performance of the main topic of the thesis, filtered search.

6.4.1 Exp. 6 – Filtering spectrum

Setup This experiment investigates how the systems deal with varying filtering percentages. We include three filter rates: 99 %, 50 %, and 1 %. This means the predicates allow that specific percentage of rows to pass. The filter is a simple $\geq$ comparison on the randomly shuffled *id* column. This shuffling hinders DuckDB’s ability to use zone maps for efficient filtering. The systems are allowed to use 14 threads/workers. The two datasets differ in the number of embeddings and dimensions but are the same total size.

Analysis In filtered search, the systems now have to evaluate the filter *and* compute the top-*k*. As shown in Figure 6.9, DuckDB’s exact search (ENNS; the “ $\times$ ”) is fastest in the Cohere 1 % of rows pass case, which clearly shows that filtering out embeddings cheapens its top-*k* calculation.

DuckDB VSS’ post-filtered HNSW excels when most rows pass the filter. Unfortunately, as discussed in Section 2.4, DuckDB VSS’ post-filtering implementation does not account for filters that eliminate half to most of the rows. After retrieving a top-*k* through the HNSW index search, it filters them, likely returning $< K$ items, as most were filtered out. As shown in Figure 6.9, this makes DuckDB VSS unusable when 1 % or 50 % of rows pass, reaching a mean recall of ~ 50 % and ~ 1 % respectively.

pgvector’s post-filtered HNSW and IVF indexes achieve textbook results. It performs best when most tuples pass and surprisingly well when only half of the tuples pass the filter. Its iterative index scan mechanism allows it to achieve high recall even when only 1 % of tuples pass the filter, but that does come at a cost to throughput. As expected, the HNSW index outperforms its IVF index. Again, as in previous experiments, in general its performance seems limited. In addition to the aforementioned issues, PostgreSQL being row-oriented will not aid its predicate evaluation performance.

DuckDB PDXearch’s global and row group approaches achieve similar throughput. This is unlike the non-filtered search performance (Section 6.3.2), where the global approach pulled ahead. We assume that by adding costly predicate evaluation to the mix, the difference between the approaches is muted. Note that for both approaches the predicate evaluation is multi-threaded. PDXearch’s pre-filtering allows us to skip embeddings during the index search. As expected, this allows PDXearch to achieve ~ 900 QPS at 90 % recall when few tuples pass the filter. When most tuples pass, pre-filtering is suboptimal, yet we still manage ~ 500 QPS at 90 % recall. The difference between post-filtering and pre-filtering allows DuckDB VSS

6.4 Filtered Search

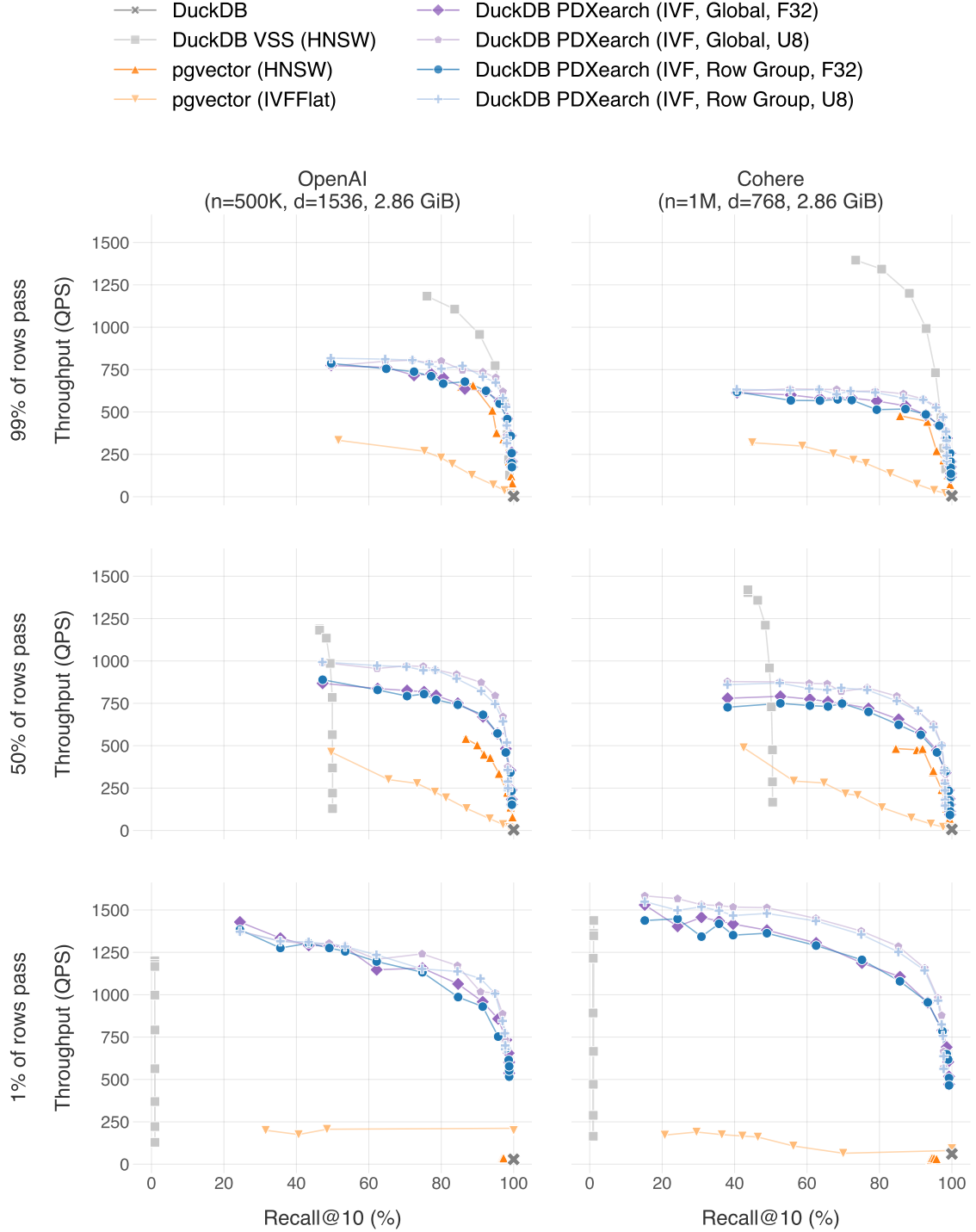


Figure 6.9: Exp. 6 – Filtered search comparison of all systems using a synthetic predicate attribute (*id*). The filter is a simple $\geq$ comparison on the *id* column. Dataset rows are shuffled, so the *id* column values are not in order. The systems are allowed to use 14 threads/workers. Each point represents the median QPS of 1000 search queries. The vertical error bars indicate the bootstrapped 95 % confidence interval. The recall is the mean of the 1000 queries.

6. EVALUATION

to exceed DuckDB PDXearch’s performance when 99 % of rows pass. Similar to previous experiments, PDXearch’s U8 SQ improves throughput at only a slight recall cost.

Our main findings from this filter spectrum filtered search experiment are:

Main Finding 11: PDXearch global and row group achieve similar filtered search throughput. Yet in the previous non-filtered search experiment, global was faster. We argue the predicate evaluation overhead mutes the difference. Scalar quantization (SQ) again improves throughput at only a slight recall cost. As expected, our pre-filtering provides its highest throughput when handling queries that eliminate most rows. On these two 2.86 GiB datasets and using recommended construction parameters, we achieve ~900 QPS at 90 % recall when 1 % of rows pass and ~500 QPS at 90 % recall when 99 % of rows pass. As expected, when most rows pass, the throughput is lower than DuckDB VSS’ post-filtered HNSW.

Main Finding 12: DuckDB VSS’ post-filtering approach reaches ~1000 QPS at 90 % recall when most rows pass but currently cannot handle queries that eliminate most of the rows. It will return $< K$ results in that case, resulting in low recall and making it practically unusable for filtered search.

Main Finding 13: The post-filtered pgvector’s HNSW and IVF produce high recall across the board, but as in previous experiments, they cannot keep up in terms of throughput.

6.4.2 Exp. 7 – Row group pruning

Setup This experiment investigates whether the row group approach’s row group pruning works. We see how the filtering percentage of the query and the order of the rows affect performance. As mentioned in Section 4.3, a potential benefit of the row group approach is row group pruning. That is, when no rows in a row group pass the predicate, we can completely skip searching the index of that *pruned* row group.

Similarly to the non-filtered experiment in Section 6.3.3, we compare the DuckDB PDXearch global and PDXearch row group approaches using the large ~17 GiB Cohere dataset. This dataset has 6 021 120 embeddings stored across 49 row groups. This number of row groups makes it more suitable for row group pruning than the smaller datasets. Both approaches use one partition for every 256 embeddings, leading to 480 partitions per row group and 23 520 partitions for the dataset in total. Because the approaches use the same number of partitions, the number of distance calculations to the centroids is equal, unless the row group pruning mechanism skips one or more row groups.

For the test queries’ filter conditions, we use range predicates. Each test query’s predicate targets a specific filtering percentage (e.g., 1 % of rows pass, 10 % of rows pass, etc.). The

range predicate's bounds (i.e., start and end) are random, but the length of the range ensures the filtering percentage is met. There are 10 500 test queries, covering 21 distinct filtering percentages, and thus 500 queries per filtering percentage.

We have two versions of this dataset: the original dataset, in which the rows are shuffled, and a version in which the rows are ordered by the predicate column. We expect that DuckDB can use its zone maps to evaluate the range filters on the ordered dataset more efficiently.

One might notice that some points are missing from the graph. 4 data points in this experiment caused the macOS out of memory (OOM) killer to terminate the process. 1 data point caused a kernel panic. The lack of these points does not affect our conclusions. However, they do stress that this 17.17 GiB dataset is on the edge of what is possible with the current implementation (particularly for the F32 global index) on a 48 GiB RAM machine. Ad hoc monitoring showed a high use of swap. Optimizing peak memory usage during index creation and adding out-of-core support is part of future work.

Analysis As a reminder, in the non-filtered experiment in Section 6.3.3, we concluded that when both the row group and global approaches use F32 and 1 thread, there is no discernible performance difference. Using quantization (U8) and 14 threads gives the row group approach a performance boost.

In Figure 6.10, we start with a single-threaded comparison of the global 256/cl and row group approach. Both the F32 and U8 results are shown. Beware that, as the plot legend explains, unlike previous experiments, the low-opacity dotted lines present the dataset with the *randomized* attribute column. The full-opacity lines present the *sorted* dataset.

A range filter on the *randomized* dataset causes the passing rows to be scattered across multiple row groups. This, for both approaches, reduces the efficiency of DuckDB's filter evaluation (no clean zone map boundaries). It is also the worst-case scenario for the row group approach, as it has to search multiple row groups' IVF index with few passing rows each. As shown by Figure 6.10, regardless of the filtering percentage, the approaches differ little in throughput.

The best-case scenario for the row group approach is the *sorted* dataset. This is where queries that eliminate most tuples will require only a few consecutive row groups to be searched. DuckDB can use its zone maps to more efficiently find these row groups (row group pruning). Furthermore, in the row group approach, we can further exploit this by row group pruning the micro IVF indexes as well. Compared to the randomized dataset, we can see both approaches benefit from DuckDB's efficient range filter evaluation. Furthermore, the row group pruning seems to work, as the single-threaded row group approach significantly exceeds the performance of the global approach for the $\leq 1\%$ filtering percentage cases. Comparing F32 and U8, it is again clear that SQ improves throughput at a slight recall cost.

6. EVALUATION

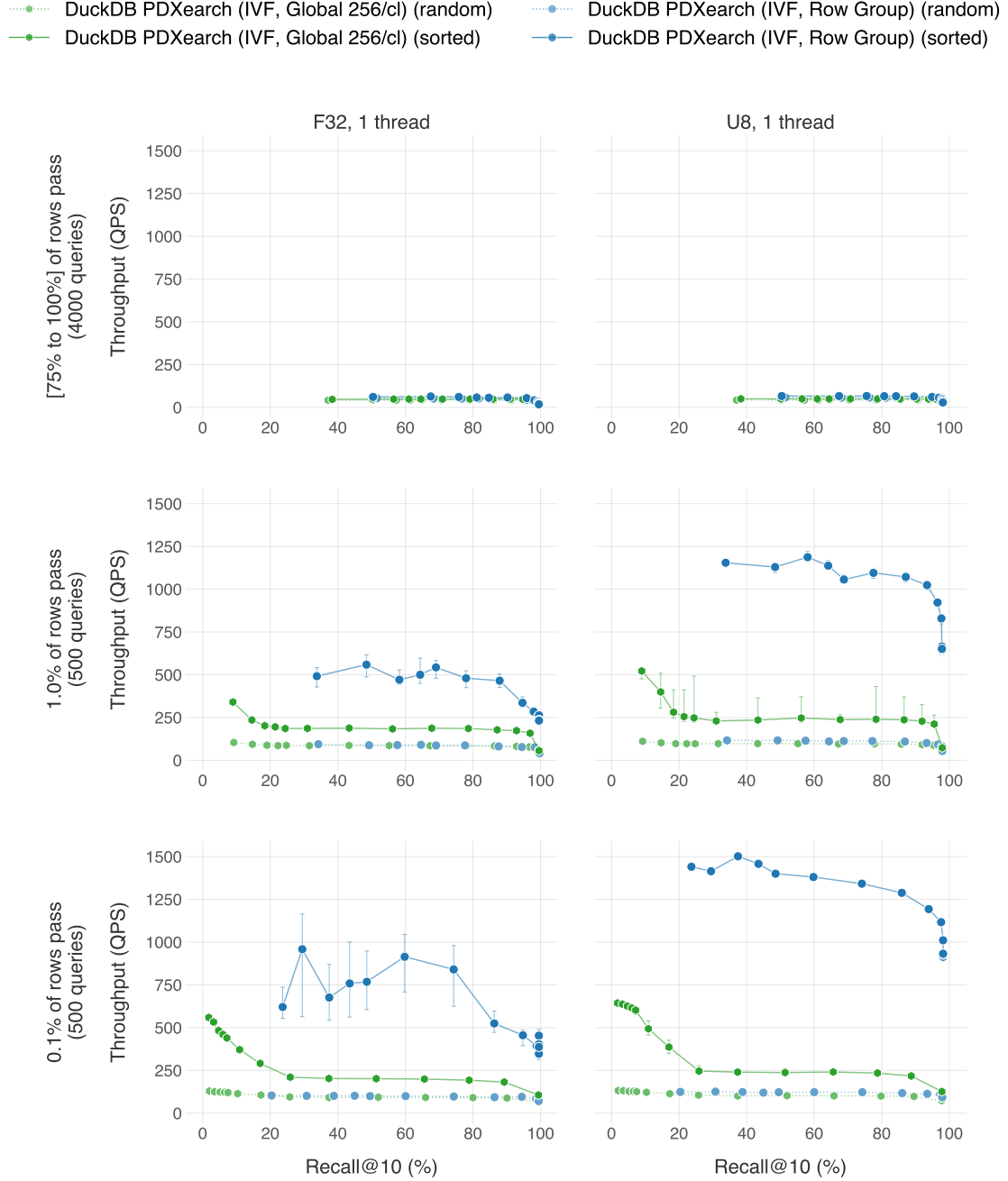


Figure 6.10: Exp. 7 – Filtered search comparison of DuckDB PDSearch's global and row group approach on Cohere6M. *Single-threaded F32 vs. U8*. Both approaches use a total of 23 520 clusters; that is, “1 cluster per every 256 embeddings”. This figure depicts single-threaded results only. Each point represents the median QPS of the search queries for that filtering percentage bucket (e.g., 500 queries when 1 % of rows pass). The vertical error bars indicate the bootstrapped 95 % confidence interval. The recall is the mean of the search queries for that filtering percentage bucket.

6.4 Filtered Search

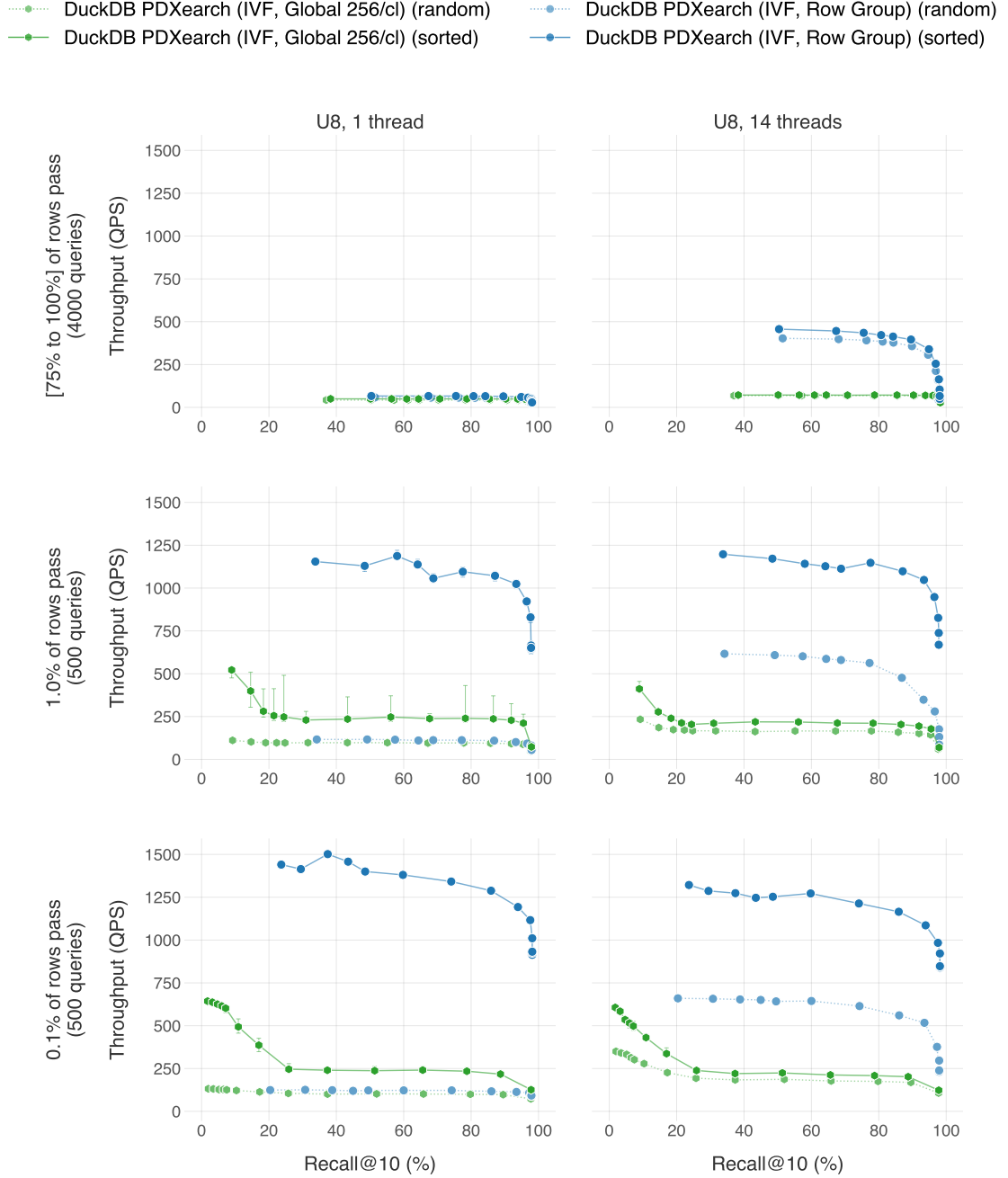


Figure 6.11: Exp. 7 – Filtered search comparison of DuckDB PDSearch's global and row group approach on Cohere6M. *Scalar quantized (U8) 1 thread vs. 14 threads.* Both approaches use a total of 23 520 clusters; that is, “1 cluster per every 256 embeddings”. This figure depicts quantized (U8) results only. Each point represents the median QPS of the search queries for that filtering percentage bucket (e.g., 500 queries when 1% of rows pass). The vertical error bars indicate the bootstrapped 95% confidence interval. The recall is the mean of the search queries for that filtering percentage bucket.

6. EVALUATION

Figure 6.11 shows the effect of the number of threads. We examine the U8 results (instead of F32) as the quantized index achieves the highest throughput. The global implementation has single-threaded search, so the vector search using multiple threads will only cause overhead, as concluded in Section 6.3.2. However, it does speed up DuckDB's filter evaluation, which improves the global approach's performance on the random dataset. The sorted dataset does not see an increase, presumably because the zone maps are so efficient that multi-threading does not help. Across the filtering percentage spectrum, the global implementation ranges between ~ 70 and ~ 200 QPS at 90 % recall.

For the row group approach, we have two steps that are parallelized, both the filtering and the micro IVF index search. As indicated by earlier non-filtered experiments (Section 6.3.3), when most (or all) rows pass, the row group approach benefits from multi-threading, reaching ~ 400 QPS at 90 % recall. When few rows pass in the sorted dataset, the passing rows are concentrated in a few row groups. In that scenario, there is too little work; hence, multi-threading does not increase the throughput. We exceed 1000 QPS at 90 % recall. It is different when few rows pass in the randomized dataset. Since the passing rows are scattered across more row groups, the filter evaluation is more costly, and fewer row groups are pruned, requiring more to be searched. Multi-threading improves the row group approach's throughput on a worst-case randomized dataset, reaching 400 QPS at 90 % recall.

Our main finding from this row group pruning filtered search experiment is:

Main Finding 14: The row group approach meets or significantly exceeds the performance of the global approach in all scenarios on this ~ 17 GiB dataset. This is due to the parallelized row group search and, when applicable, row group pruning. On queries with a filtering percentage between 75 % – 100 %, multi-threading and SQ enable it to reach ~ 400 QPS at 90 % recall. For queries where only 0.1 % – 1 % of tuples pass on a randomized dataset, the throughput increases slightly. The same queries on a sorted dataset permit row group pruning, reaching ≥ 1000 QPS at 90 % recall with only a single thread.

Conclusion and Future Work

In this thesis, we claim that vector-extended relational analytical DBMSs 1) should not only adjust the system to integrate the index but also design the index to fit the system and to integrate it tightly, and 2) that they should consider integrating a lightweight predicate-agnostic IVF index through pre-filtering. To support our claim, we design and implement a DuckDB extension for VSS that uses a lightweight and fast IVF index. The index tightly integrates with DuckDB through an integrated filtering approach, which is an intertwined version of pre-filtering. We propose two designs: a baseline approach with a single global IVF index and an IVF-per-row group approach. The latter leverages DuckDB's morsel-driven parallelized execution model for efficient processing of filtered VSS queries. We discuss each design's tradeoffs and implementations and compare them to plain DuckDB (no index), DuckDB's current VSS extension (HNSW), and PostgreSQL's pgvector extension (IVF and HNSW). Our experiments show that our IVF-based extension is competitive with DuckDB VSS' HNSW in terms of QPS, and can achieve high recall on filtered VSS queries, even when few rows pass the filter. Furthermore, the row group approach yields different performance characteristics due to its tight integration with DuckDB's execution model. Despite it being slower than the naive global implementation for single-threaded non-filtered search on small datasets, the real gains appear on datasets of increased sizes, queries where $\leq 1\%$ of rows pass, and predicates that align with the data's insertion order. In this scenario, the micro IVF approach can leverage intra-operator parallelism, row group pruning, and fast predicate evaluation. In the next sections, we answer the proposed research questions, discuss the limitations of our evaluation, and consider future work.

7. CONCLUSION AND FUTURE WORK

7.1 Answers to the Research Questions

We use the design tradeoffs (Table 4.1, Table 6.1), implementation discussion, and the evaluation’s main findings to answer the research questions we introduced in Section 1.2.

(Q1): How can we design a tightly integrated vector search architecture that provides efficient filtered search within a relational analytical vector-extended DBMS?

In the introduction, we target analytical vector-extended DBMSs where efficient maintenance (i.e., handling updates) and efficient filtered search are priorities. We argued that, in that context, a SOTA IVF tightly integrated through an approach similar to pre-filtering is a suitable approach. In the thesis, we used the following sub-questions to determine the remainder of the design.

(Q1a): How can we leverage the parallelization model of the DBMS to improve vector search performance? DuckDB uses morsel-driven parallelism, which affects its execution model and storage layout. Our redesigned row group approach uses a separate micro IVF per storage block (i.e., per row group). This allows not only DuckDB’s predicate evaluation to be executed in parallel but the micro IVF index search as well. This yields intra-operator parallelism, speeding up the vector search query. Furthermore, for filtered queries, this aligned approach allows us to extend DuckDB’s row group pruning to the index search. This makes the execution of filtered search queries more efficient by skipping the search of filtered-out micro IVFs.

(Q1b): How should we handle queries with predicates that eliminate almost all tuples? The pre-filtering integration and row group pruning make filtered search queries where few tuples pass efficient. As discussed in Section 5.6, to make these queries user-friendly, we implemented an iteration mechanism. This mechanism ensures that the user receives the top- k items without the need to manually tweak `n_probe` and rerun the query to compensate for the percentage of rows that pass in each partition.

We evaluated the characteristics of our specific analytical vector-extended DBMS, DuckDB. We landed on a parallelism and storage model-aligned approach with the row group pruning and iteration mechanisms to improve the query execution efficiency and user experience. We encourage other systems to evaluate their architecture to see if adapting their vector search index to their specific system can yield performance breakthroughs.

(Q2): What does the design offer?

We conclude what the design offers through the following sub-questions.

(Q2a): How do the features and performance of the baseline global approach compare to the other systems? In other words, is a pre-filtered global SOTA IVF a suitable approach for efficient filtered search on an analytical vector-extended DBMS, where not just throughput but data management is also important? Our results indicate that it is.

7.1 Answers to the Research Questions

The global IVF uses roughly 30 % to 50 % less memory than DuckDB VSS' HNSW, is approximately 10 to 26 times faster to construct, and can easily support efficient incremental maintenance. The non-filtered search performance is competitive, exceeding DuckDB VSS' throughput in 3 out of 4 datasets. This confirms that, on these benchmarking datasets, the PDXearch IVF library [27] can match the throughput of an HNSW index. For filtered search, we exploit DuckDB's optimized predicate evaluation. As expected, when most rows pass global's filtered search performance lags behind DuckDB VSS' post-filtering integration. However, for queries where most tuples are eliminated we argue¹ that our pre-filtered IVF reaches significantly higher throughput than the post-filtered HNSW approaches.

(Q2b): How do the features and performance of the tightly integrated row group approach compare to the other systems? In other words, what does designing the index around the system and integrating it tightly offer?

As discussed in the previous question, a pre-filtered SOTA IVF index fits an analytical vector-extended DBMS well. The row group approach's use of a micro IVF per row group changes its performance characteristics.

The row group approach uses a similar amount of memory, roughly 30 % to 50 % less than DuckDB VSS' HNSW. It is even faster to construct than the global IVF, being approximately 33 to 56 times faster to construct than DuckDB VSS' HNSW. Furthermore, index maintenance is likely spread across a few small IVF indexes instead of a global one, making maintenance more efficient and mitigating the impact of data distribution drift. In terms of non-filtered search, the row group approach incurs an overhead because it has multiple IVFs, leading to lower throughput than the global approach. It can compensate partially by using multiple threads, matching DuckDB VSS' throughput on 2 out of 4 datasets. For filtered search on the smaller datasets, it matches the global IVF's performance. On the large 17.17 GiB dataset, the row group approach pulls ahead. First, its multi-threaded search increases throughput regardless of the filtering percentage and dataset. Second, for queries that filter out most rows on datasets with sorted predicate columns, it can use its row group pruning to increase the throughput significantly.

(Q2c): How does PDXearch's scalar quantization affect the performance?

Across all experiments, PDXearch's scalar quantization (SQ) increases throughput at a slight recall loss and negligible computational overhead (i.e., quantizing the embeddings). Furthermore, it reduced the in-memory index size to $\sim \frac{1}{4}$ of the original, and accelerated index construction on the largest two datasets. As concluded in Section 6.3.2, the U8 scalar quantization (SQ) should always be enabled, unless the use case requires a recall of 97 % or above.

¹As explained in Section 2.4, pre-filtering is preferred when most tuples are filtered out. Unfortunately, pgvector's relatively low throughput and DuckDB VSS' lack of an iteration mechanism do not allow us to reach this conclusion from the evidence, as we do not have other systems to compare to.

7. CONCLUSION AND FUTURE WORK

7.2 Limitations of the Evaluation

This section acknowledges threats to the validity of the evaluation and suggests future benchmarking opportunities. Section 7.3 focuses on extension features.

7.2.1 Workloads

Maintenance The experiments in this thesis tell only part of the story. The index’s ability to handle inserts and deletes (maintenance) efficiently is a design priority, but the current implementation does not yet support it, and a maintenance workload benchmark is missing. Once supported, we should benchmark a maintenance workload (e.g., run VectorDBBench’s streaming workload or develop a more comprehensive workload).

More systems The breadth of the systems that are part of the evaluation (Table 6.1) is limited. Although PostgreSQL’s pgvector is mature and in wide use, it is a row-oriented client-server DBMS. DuckDB VSS is suitable but limited in its maturity (a lacking filtered-search implementation). We should incorporate more systems, including DuckDB’s Faiss¹ and Lance extensions, and other vector-extended OLAP VDBMSs such as ClickHouse [111].

Larger datasets The datasets cover common benchmark characteristics (number of dimensions and dataset size). However, to evaluate the system’s scalability, which is particularly interesting given the row group approach’s search overhead, we need larger datasets (similar to Cohere6M and larger). This might necessitate hardware with more main memory. Additionally, once implemented, we can benchmark out-of-core (larger than memory) processing as well.

Quantization in other systems In the thesis, we consider only PDXearch’s 8-bit SQ. We should include other systems’ quantization offerings in the benchmarks as well.

Concurrent throughput The thesis benchmarks focus on serial throughput, which relates to a system’s intra-operator parallelism. This single query at a time fits a data scientist’s exploratory query pattern. Another interesting performance aspect is concurrent throughput, which systems such as pgvector optimize (i.e., inter-operator parallelism). Once concurrency is supported by PDXearch, we can benchmark concurrent search performance across the systems.

7.2.2 Data gathering

Blob interface encoding bug The blob interface introduced in Section 5.8.3, has a bug in its encoding. As discussed in Section 6.3.2, this prevents us from using the SIFT dataset in our experiments. We should resolve the issue or use a different approach to address the DuckDB overhead on vector search queries.

¹The Faiss extension uses pre-filtering, similar to PDXearch, and incorporates the Faiss library [15, 50]. Many of the system’s characteristics will overlap, arguably increasing fairness.

7.2 Limitations of the Evaluation

Diverse hardware We ran our experiments on the Apple M4 Pro machine. An interesting follow-up would be to repeat our experiments on different hardware (e.g., CPU architecture, amount of RAM).

Construction index parameter sweep In Section 6.1 we argued that sweeping only the run-time parameters and relying on recommendations for the construction parameters is realistic for data scientist workloads and common in the literature. However, using a full parameter sweep (such as [12]) will show the implementation’s performance limits. This additional information could help guide the design and implementation of the extension and improve our construction parameter recommendations.

Always measure one level deeper The measurements gathered for the thesis mostly consisted of the end-to-end latency (serial throughput) and slightly deeper measurements as necessary (e.g., latency decomposition for the blob interface). Additionally, during development, we have used the DuckDB profiler, statistical profiling, and custom metadata output, but adding this ad hoc is impractical. In the spirit of Ousterhout’s “Always Measure One Level Deeper” article [112], we could capture many low-level details (e.g., number of distance calculations, number of partitions probed (per row group), number of pruned row groups, number of iterations, number of filter checks, bytes read, page accesses, peak memory usage, etc.). We suggest building emission of these metrics into the extension permanently, guarded with compile-time features. This simplifies making evidence-based decisions.

7.2.3 PDXearch

Multi-threaded global variant The current PDXearch global variant has single-threaded search. As discussed in Section 2.2.2, PDXearch’s global IVF could be searched in parallel. Comparing the parallelized row group approach to this could be insightful.

Maturity Lastly, the lack of features and stability could give PDXearch an unfair advantage. Unlike pgvector, the PDXearch extension currently does not support persistence¹, maintenance, concurrency, etc. This lack of features also swings the other way; PDXearch, in its experimental state, is likely less optimized than other approaches. Overall, it makes it harder to generalize and apply findings about design decisions to external systems. The following section, Section 7.3, lists the future work planned for the extension.

¹As found in Figure 6.2, pgvector’s persistence led to significant internal fragmentation, negatively impacting performance.

7. CONCLUSION AND FUTURE WORK

7.3 Future Work

In this section, we briefly discuss feature enhancements for the PDSearch extension. As mentioned earlier, we assume that we are on DuckDB v1.4.4 with PDSearch extension commit cbed8a5¹.

Maintenance The current implementation supports only static collections. We aim to add support for inserts and deletes to the index. The SOTA IVF (and row group) design choices should allow for smooth incremental updates.

Persistence Although on small to medium datasets, index construction takes only seconds, we should allow the index to be persisted to disk and loaded from disk when necessary. DuckDB uses 256 KiB storage blocks by default. It will be interesting to see how we can fit our (quantized) embeddings onto those blocks while keeping in mind PDX's transposed format, IVF partitions, and efficient access patterns. This persistence can be enhanced with support for block eviction and lazily loading only parts of the index at a later date.

Out of core The benchmarking workloads fit in RAM (or spilled to swap). To handle big datasets efficiently, we should modify the implementation and verify that it is ready for out-of-core processing. In addition, we should investigate how to integrate best with DuckDB memory management.

Concurrency Currently the index supports only a single index scan at a time. We should at least broaden that to *concurrent index scans and serialized maintenance*. The end goal is efficient support for concurrent transactional access to the index, integrating properly with DuckDB's MVCC, and allowing *concurrent index scans and concurrent maintenance*.

Query coverage As discussed in Section 5.3 and Section 5.8.1, the extension optimizes only simple pushed-down filters at this point and does not integrate well with DuckDB's late materialization optimization. We should rewrite the PDSearch optimizer and operators, ensuring that we are predicate agnostic and making the plan efficient (e.g., use late materialization where it makes sense, no redundant operators).

Row group tracking As mentioned in Section 5.8.2, the current implementation is not flexible; it requires all but the last row groups to be full. If exposed, we should fetch the necessary row group metadata from DuckDB directly to support arbitrarily sized row groups. Or alternatively, let go of DuckDB's row groups and explore other grouping approaches. We should consider support for multiple data sources (tables, Parquet, etc.) and deal correctly with DuckDB's background compaction of row groups.

DuckDB overhead Section 5.8.3 discussed DuckDB's planner and optimizer overhead for VSS queries and the blob interface workaround. We should tackle the root problem or, if that is infeasible, work around the issue transparently.

¹The PDSearch extension's thesis version can be found at: <https://github.com/Noorts/PDSearch/tree/cbed8a5b35dd0ee8e6697ade0c92679b66ee8d40>

7.3 Future Work

Additional features We could add support for features that the DuckDB VSS or pgvector extensions support, e.g., similarity joins (i.e., batch VSS queries), the inner product distance metric, and partial indexing.

DuckLake We can explore what role catalog database indices should play in a DuckLake setting. One direction would be to combine DuckLake with the PDXearch extension to offer lakehouse-scale vector similarity search.

References

- [1] TONI TAIPALUS. **Vector Database Management Systems: Fundamental Concepts, Use-Cases, and Current Challenges**. *Cognitive Systems Research*, **85**:101216, June 2024. 1, 2, 7, 8
- [2] JAMES JIE PAN, JIANGUO WANG, AND GUOLIANG LI. **Survey of Vector Database Management Systems**, October 2023. 1, 2, 3, 7, 8, 9, 15, 16, 47
- [3] LE MA, RAN ZHANG, YIKUN HAN, SHIRUI YU, ZAITIAN WANG, ZHIYUAN NING, JINGHAN ZHANG, PING XU, PENGJIANG LI, WEI JU, CHONG CHEN, DONGJIE WANG, KUNPENG LIU, PENGYANG WANG, PENGFEI WANG, YANJIE FU, CHUNJIANG LIU, YUANCHUN ZHOU, AND CHANG-TIEN LU. **A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge**, June 2025. 1, 2, 3, 7, 15, 54
- [4] ILIAS AZIZI, KARIMA ECHIHABI, AND THEMIS PALPANAS. **Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art**. *Proc. ACM Manag. Data*, **3**(1):43:1–43:31, February 2025. 1, 2, 3, 4, 7, 9, 14
- [5] MICHAEL GÜNTHER, MAIK THIELE, AND WOLFGANG LEHNER. **RETRO: Relation Retrofitting For In-Database Machine Learning on Textual Data**, January 2020. 1, 7
- [6] WEN LI, YING ZHANG, YIFANG SUN, WEI WANG, WENJIE ZHANG, AND XUEMIN LIN. **Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement (v1.0)**, October 2016. 1, 2, 3, 7, 8
- [7] LEONARDO KUFFO. **Juggling High-Dimensional Vectors in the AI Era**, 2025. 1, 14
- [8] PATRICK IFF, PAUL BRUEGGER, MARCIN CHRAPEK, MACIEJ BESTA, AND TORSTEN HOEFLE. **Benchmarking Filtered Approximate Nearest Neighbor Search Algorithms on Transformer-based Embedding Vectors**, July 2025. 1, 3, 8, 9, 16, 17, 18, 49, 52, 54, 55, 93
- [9] LEONARDO KUFFO, ELENA KRIPPNER, AND PETER BONCZ. **PDX: A Data Layout for Vector Similarity Search**. *Proc. ACM Manag. Data*, **3**(3):196:1–196:26, June 2025. 1, 2, 4, 9, 11, 12, 13, 14, 44, 45, 52
- [10] MARTIN AUMUELLER, ERIK BERNHARDSSON, AND ALEC FAITFULL. **ANN-Benchmarks**. 1, 9, 49
- [11] JIAXU ZHU, JIAYU YUAN, KAIWEN YANG, XIAOBAO CHEN, SHIHUAN YU, HONGCHANG LV, YAN LI, AND BOLONG ZHENG. **An Experimental Evaluation of Hybrid Querying on Vectors**. *Proceedings of the VLDB Endowment*, **19**(2):183–195, October 2025. 1, 2, 3, 9, 14, 16, 52

-
- [12] MARTIN AUMÜLLER, ERIK BERNHARDSSON, AND ALEXANDER FAITHFULL. **ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms**, July 2018. 2, 14, 52, 81
 - [13] YU A. MALKOV AND D. A. YASHUNIN. **Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **42**(4):824–836, April 2020. 2, 9, 10, 11
 - [14] ASHOT VARDANIAN, ISHKHAN NAZARYAN, SEMANTIC RELEASE BOT, CRAIG SPENCE, AND EVANDRO LEOPOLDINO GONÇALVES. **Unum-Cloud/Usearch: V0.5.0**. Zenodo, May 2023. 2, 11, 23
 - [15] MATTHIJS DOUZE, ALEXANDR GUZHVA, CHENGQI DENG, JEFF JOHNSON, GERGELY SZILVASY, PIERRE-EMMANUEL MAZARÉ, MARIA LOMELI, LUCAS HOSSEINI, AND HERVÉ JÉGOU. **The Faiss Library**, February 2025. 2, 4, 11, 12, 14, 15, 22, 53, 80
 - [16] JIMMY LIN, RONAK PRADEEP, TOMMASO TEOFILI, AND JASPER XIAN. **Vector Search with OpenAI Embeddings: Lucene Is All You Need**, August 2023. 2
 - [17] CE GAO. **Do We Really Need a Specialized Vector Database? (2023)**, February 2025. 2
 - [18] ERIC HANSON AND ARNAUD COMET. **Why Your Vector Database Should Not Be a Vector Database | SingleStoreDB**, April 2023. 2
 - [19] MARK RAASVELDT, HANNES MÜHLEISEN, AND DUCKDB LABS. **DuckDB GitHub**, October 2025. 2, 8, 21, 51
 - [20] MAX GABRIELSSON AND DUCKDB LABS. **DuckDB VSS GitHub**. DuckDB, October 2025. 2, 22, 23, 46, 51
 - [21] JIAYANG SHI, YUZHENG CAI, AND WEIGUO ZHENG. **Filtered Approximate Nearest Neighbor Search: A Unified Benchmark and Systematic Experimental Study [Experiment, Analysis & Benchmark]**, September 2025. 2, 15, 16, 52, 55
 - [22] DUO LU, HELENA CAMINAL, MANOS CHATZAKIS, YANNIS PAPAKONSTANTINOU, YANNIS CHRONIS, VAIBHAV JAIN, AND FATMA ÖZCAN. **An In-Depth Study of Filter-Agnostic Vector Search on a PostgreSQL Database System: [Experiments and Analysis]**, March 2026. 3, 24
 - [23] YUMING XU, HENGYU LIANG, JIN LI, SHUOTAO XU, QI CHEN, QIANXI ZHANG, CHENG LI, ZIYUE YANG, FAN YANG, YUQING YANG, PENG CHENG, AND MAO

REFERENCES

- YANG. **SPFresh: Incremental In-Place Update for Billion-Scale Vector Search**. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, pages 545–561, New York, NY, USA, October 2023. Association for Computing Machinery. 3, 11, 13, 14
- [24] NITISH UPRETI, HARSHA VARDHAN SIMHADRI, HARI SUDAN SUNDAR, KRISHNAN SUNDARAM, SAMER BOSHRA, BALACHANDAR PERUMALSWAMY, SHIVAM ATRI, MARTIN CHISHOLM, REVTI RAMAN SINGH, GREG YANG, TAMARA HASS, NITESH DUDHEY, SUBRAMANYAM PATTIPAKA, MARK HILDEBRAND, MAGDALEN MANOHAR, JACK MOFFITT, HAIYANG XU, NAREN DATHA, SURYANSH GUPTA, RAVISHANKAR KRISHNASWAMY, PRASHANT GUPTA, ABHISHEK SAHU, HEMESWARI VARADA, SUDHANSHU BARTHWAL, RITIKA MOR, JAMES CODELLA, SHAUN COOPER, KEVIN PILCH, SIMON MORENO, AAYUSH KATARIA, SANTOSH KULKARNI, NEIL DESHPANDE, AMAR SAGARE, DINESH BILLA, ZISHAN FU, AND VIPUL VISHAL. **Cost-Effective, Low Latency Vector Search with Azure Cosmos DB**, 2025. 4, 14, 52
- [25] HAIKE XU, MAGDALEN DOBSON MANOHAR, PHILIP A. BERNSTEIN, BADRISH CHANDRAMOULI, RICHARD WEN, AND HARSHA VARDHAN SIMHADRI. **In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search**, February 2025. 4, 14
- [26] YANNIS PAPAKONSTANTINOU, ALAN LI, RUIQI GUO, SANJIV KUMAR, PHIL SUN, AND GOOGLE CLOUD. **ScaNN for AlloyDB**, April 2024. 4, 14
- [27] LEONARDO KUFFO. **Cwida/PDX**. CWI Database Architectures Group, April 2026. 4, 12, 13, 16, 44, 79
- [28] LEONARDO KUFFO. **Cwida/SuperKMeans**. CWI Database Architectures Group, April 2026. 4, 13, 53
- [29] LEONARDO KUFFO, SVEN HEPKEMA, AND PETER BONCZ. **A Super Fast K-means for Indexing Vector Embeddings**, March 2026. 4, 13, 30
- [30] PETER BONCZ, M. ZUKOWSKI, AND NIELS NES. **MonetDB/X100: Hyper-Pipelining Query Execution**. *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005*, January 2005. 8, 21
- [31] TOMAS MIKOLOV, ILYA SUTSKEVER, KAI CHEN, GREG CORRADO, AND JEFFREY DEAN. **Distributed Representations of Words and Phrases and Their Compositionality**, October 2013. 8

REFERENCES

- [32] ALEC RADFORD, JONG WOOK KIM, CHRIS HALLACY, ADITYA RAMESH, GABRIEL GOH, SANDHINI AGARWAL, GIRISH SASTRY, AMANDA ASKELL, PAMELA MISHKIN, JACK CLARK, GRETCHEN KRUEGER, AND ILYA SUTSKEVER. **Learning Transferable Visual Models From Natural Language Supervision**, February 2021. 8
- [33] ELIAS JÄÄSAARI, VILLE HYVÖNEN, MATTEO CECCARELLO, TEEMU ROOS, AND MARTIN AUMÜLLER. **VIBE: Vector Index Benchmark for Embeddings**, May 2025. 9, 14, 52
- [34] ALEXANDER PONOMARENKO, YU MALKOV, KOV, ANDREY LOGVINOV, AND VLADIMIR KRYLOV. **Approximate Nearest Neighbor Search Small World Approach**. In *International Conference on Information and Communication Technologies and Applications*, January 2011. 9
- [35] YURY MALKOV, ALEXANDER PONOMARENKO, ANDREY LOGVINOV, AND VLADIMIR KRYLOV. **Approximate Nearest Neighbor Algorithm Based on Navigable Small World Graphs**. *Information Systems*, **45**:61–68, September 2014. 9
- [36] WILLIAM PUGH. **Skip Lists: A Probabilistic Alternative to Balanced Trees**. *Communications of the ACM*, **33**(6):668–676, June 1990. 9
- [37] DUCKDB LABS. **DuckDB Docs - Vector Similarity Search (VSS) Extension**. 10, 11, 22, 23, 50
- [38] ANDREW KANE. **Pgvector/Pgvector**. pgvector, March 2026. 11, 12, 23, 51, 53, 58, 93
- [39] VASILIS MAGEIRAKOS, BOWEN WU, AND GUSTAVO ALONSO. **Cracking Vector Search Indexes**. *Proceedings of the VLDB Endowment*, **18**(11):3951–3964, July 2025. 11, 14
- [40] QI CHEN, BING ZHAO, HAIDONG WANG, MINGQIN LI, CHUANJIE LIU, ZENGZHONG LI, MAO YANG, AND JINGDONG WANG. **SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search**, November 2021. 11
- [41] JASON MOHONEY, DEVESH SARDA, MENGZE TANG, SHIHABUR RAHMAN CHOWDHURY, ANIL PACACI, IHAB F. ILYAS, THEODOROS REKATSINAS, AND SHIVARAM VENKATARAMAN. **Quake: Adaptive Indexing for Vector Search**, June 2025. 11, 13, 14
- [42] S. LLOYD. **Least Squares Quantization in PCM**. *IEEE Transactions on Information Theory*, **28**(2):129–137, March 1982. 11, 26

REFERENCES

- [43] J. MAX. **Quantizing for Minimum Distortion.** *IRE Transactions on Information Theory*, **6**(1):7–12, March 1960. 11
- [44] JASON MOHONEY, ANIL PACACI, SHIHABUR RAHMAN CHOWDHURY, UMAR FAROOQ MINHAS, JEFFERY POUND, CEDRIC RENGGLI, NIMA REYHANI, IHAB F. ILYAS, THEODOROS REKATSINAS, AND SHIVARAM VENKATARAMAN. **Incremental IVF Index Maintenance for Streaming Vector Search**, November 2024. 13, 14
- [45] SIMON VAN NOORT. **Noorts/PDXearch**, May 2026. 13, 51, 55
- [46] LEONARDO KUFFO. **Indexing 10M Vector Embeddings in 1 Minute on a Single CPU**, February 2026. 14
- [47] JEFFERY POUND, FLORIS CHABERT, ARJUN BHUSHAN, ANKUR GOSWAMI, ANIL PACACI, AND SHIHABUR RAHMAN CHOWDHURY. **MicroNN: An On-device Disk-resident Updatable Vector Database.** In *Companion of the 2025 International Conference on Management of Data*, pages 608–621, Berlin Germany, June 2025. ACM. 14
- [48] H JÉGOU, M DOUZE, AND C SCHMID. **Product Quantization for Nearest Neighbor Search.** *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **33**(1):117–128, January 2011. 15
- [49] ZILLIZ. **Scalar Quantization and Product Quantization - Zilliz Learn**, March 2023. 15
- [50] JAAP AARTS AND ARJEN P. DE VRIES. **DuckDB FAISS GitHub.** duckdb-faiss-ext, September 2025. 16, 22, 30, 80
- [51] LIANA PATEL, PETER KRAFT, CARLOS GUESTRIN, AND MATEI ZAHARIA. **ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data**, March 2024. 16
- [52] WEIJIE ZHAO, SHULONG TAN, AND PING LI. **Constrained Approximate Similarity Search on Proximity Graph**, November 2022. 16
- [53] XIAOLIANG XU, CHANG LI, YUXIANG WANG, AND YIXING XIA. **Multiattribute Approximate Nearest Neighbor Search Based on Navigable Small World Graph.** *Concurrency and Computation: Practice and Experience*, **32**(24):e5970, 2020. 17
- [54] MENGZHAO WANG, LINGWEI LV, XIAOLIANG XU, YUXIANG WANG, QIANG YUE, AND JIONGKANG NI. **Navigable Proximity Graph-Driven Native Hybrid Queries with Structured and Unstructured Constraints**, March 2022. 17

-
- [55] CHAOJI ZUO, MIAO QIAO, WENCHAO ZHOU, FEIFEI LI, AND DONG DENG. **SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search**. *Proceedings of the ACM on Management of Data*, 2(1):1–26, March 2024. 17
 - [56] YUEXUAN XU, JIANYANG GAO, YUTONG GOU, CHENG LONG, AND CHRISTIAN S. JENSEN. **iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search**, September 2024. 17
 - [57] MILVUS. **Filtered Search | Milvus Documentation**. 18
 - [58] CHUANGXIAN WEI, BIN WU, SHENG WANG, RENJIE LOU, CHAOQUN ZHAN, FEIFEI LI, AND YUANZHE CAI. **AnalyticDB-V: A Hybrid Analytical Engine towards Query Fusion for Structured and Unstructured Data**. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, August 2020. 18
 - [59] MARK RAASVELDT AND HANNES MÜHLEISEN. **DuckDB: An Embeddable Analytical Database**. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, Amsterdam Netherlands, June 2019. ACM. 19, 51
 - [60] DUCKDB LABS. **Adoption Metrics and Benchmark Results for DuckDB v1.4 LTS**, October 2025. 19
 - [61] HANNES MÜHLEISEN AND DUCKDB LABS. **DuckDB-Quack Announcement at AI Council**, May 2026. 19, 22, 23
 - [62] DUCKDB LABS. **Frequently Asked Questions**. 19
 - [63] TIGERBEETLE. **Tigerbeetle/Tigerbeetle**. TigerBeetle, May 2026. 19
 - [64] SURAJIT CHAUDHURI AND UMESHWAR DAYAL. **An Overview of Data Warehousing and OLAP Technology**. *SIGMOD Rec.*, 26(1):65–74, March 1997. 19
 - [65] TIGERBEETLE. **Online Transaction Processing (OLTP)**, April 2025. 19
 - [66] ALEX MONAHAN AND DUCKDB LABS. **Sorting on Insert for Fast Selective Queries**, May 2025. 21
 - [67] DANIEL J. ABADI, SAMUEL R. MADDEN, AND NABIL HACHEM. **Column-Stores vs. Row-Stores: How Different Are They Really?** In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 967–980, Vancouver Canada, June 2008. ACM. 21
 - [68] MIKE STONEBRAKER, DANIEL J. ABADI, ADAM BATKIN, XUEDONG CHEN, MITCH CHERNIACK, MIGUEL FERREIRA, EDMOND LAU, AMERSON LIN, SAM

REFERENCES

- MADDEN, ELIZABETH O'NEIL, PAT O'NEIL, ALEX RASIN, NGA TRAN, AND STAN ZDONIK. **C-Store: A Column-Oriented DBMS**. In MASSACHUSETTS INSTITUTE OF TECHNOLOGY AND MICHAEL L. BRODIE, editors, *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. Association for Computing Machinery, 1 edition, December 2018. 21
- [69] TORSTEN GRUST. **Design and Implementation of DuckDB Internals: Query Execution Plans and Pipelining**, April 2026. 21
- [70] VIKTOR LEIS, PETER BONCZ, ALFONS KEMPER, AND THOMAS NEUMANN. **Morsel-Driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-Core Age**. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 743–754, Snowbird Utah USA, June 2014. ACM. 21
- [71] DUCKDB LABS. **Data Chunks**. 22
- [72] DUCKDB LABS. **Vectors**. 22
- [73] DUCKDB LABS. **DuckDB Docs - Array Functions**. 22
- [74] LANCE. **DuckDB - Lance**. 22
- [75] DUCKDB LABS. **Community Extensions Download Metrics**. 23
- [76] MICHAEL STONEBRAKER AND LAWRENCE A. ROWE. **The Design of POSTGRES**. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, June 1986. Association for Computing Machinery. 23
- [77] POSTGRESQL. **PostgreSQL: About**. 23
- [78] POSTGRESQL. **61.1. Query Handling as a Complex Optimization Problem**, February 2026. 23
- [79] POSTGRESQL. **15.1. How Parallel Query Works**, February 2026. 23
- [80] DUCKDB LABS. **Why DuckDB**. 24
- [81] CHRISTOPHER D. MANNING, PRABHAKAR RAGHAVAN, AND HINRICH SCHÜTZE. *Introduction to Information Retrieval*. Cambridge University Press, 2008. 26
- [82] J. A. HARTIGAN AND M. A. WONG. **Algorithm AS 136: A K-Means Clustering Algorithm**. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **28**(1):100–108, 1979. 26

REFERENCES

- [83] EDWARD W. FORGY. **Cluster Analysis of Multivariate Data: Efficiency versus Interpretability of Classifications.** *Biometrics*, 21(3):768–769, 1965. 26
- [84] DUCKDB LABS. **Indexes.** 28
- [85] ALEX MONAHAN AND DUCKDB LABS. **Using Zone Maps to Skip Reading Data,** May 2025. 28
- [86] TORSTEN GRUST. **Design and Implementation of DuckDB Internals: Query Rewriting and Optimization,** April 2026. 31
- [87] LEONARDO KUFFO. **What If We Store Vector Embeddings Vertically?,** March 2025. 44
- [88] MAX GABRIELSSON. **What’s New in the Vector Similarity Search Extension?,** October 2024. 47
- [89] APPLE. **MacBook Pro (16-Inch, 2024) - Tech Specs.** 49
- [90] PASSMARK SOFTWARE. **Apple M4 Pro 14 Core Benchmark.** 49
- [91] GAURAV GUPTA, JONAH YI, BENJAMIN COLEMAN, CHEN LUO, VIHAN LAKSHMAN, AND ANSHUMALI SHRIVASTAVA. **CAPS: A Practical Partition Index for Filtered Similarity Search,** August 2023. 49
- [92] MARK RAASVELDT, PEDRO HOLANDA, TIM GUBNER, AND HANNES MÜHLEISEN. **Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing.** In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6, Houston TX USA, June 2018. ACM. 49
- [93] SIMON VAN NOORT. **Noorts/VectorDBBench,** April 2026. 52, 55
- [94] HERVE JEGOU, ROMAIN TAVENARD, MATTHIJS DOUZE, AND LAURENT AMSALEG. **Searching in One Billion Vectors: Re-rank with Source Coding.** In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864, Prague, Czech Republic, May 2011. IEEE. 52
- [95] LAURENT AMSALEG AND HERVÉ JÉGOU. **Evaluation of Approximate Nearest Neighbors: Large Datasets.** 52
- [96] ZILLIZ. **Zilliztech/VectorDBBench.** Zilliz, February 2026. 52, 54
- [97] COHERE. **Cohere / Wikipedia-22-12 · Datasets at Hugging Face,** 2022. 52
- [98] ELIAS JÄÄSAARI, VILLE HYVÖNEN, MATTEO CECCARELLO, TEEMU ROOS, AND MARTIN AUMÜLLER. **Vector-Index-Bench/Vibe · Datasets at Hugging Face.** 52

REFERENCES

- [99] MIXEDBREAD. **Mixedbread-Ai/Wikipedia-Embed-En-2023-11 · Datasets at Hugging Face**. 52
- [100] LEONARDO KUFFO. **PDX: Public Data**. 52
- [101] AI2. **Allenai/C4 · Datasets at Hugging Face**, August 2024. 52
- [102] SCPL. **SPCL/Arxiv-for-Fanns-Large · Datasets at Hugging Face**, July 2025. 52, 93
- [103] SIDDHARTH GOLLAPUDI, NEEL KARIA, VARUN SIVASHANKAR, RAVISHANKAR KRISHNASWAMY, NIKIT BEGWANI, SWAPNIL RAZ, YIYONG LIN, YIN ZHANG, NEE-LAM MAHAPATRO, PREMKUMAR SRINIVASAN, AMIT SINGH, AND HARSHA VARDHAN SIMHADRI. **Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters**. In *Proceedings of the ACM Web Conference 2023*, pages 3406–3416, Austin TX USA, April 2023. ACM. 52
- [104] ZILLIZ. **Zilliz: Vector Database Built for Enterprise-Grade AI Applications**. 54
- [105] PATRICK IFF, PAUL BRUEGGER, MARCIN CHRAPEK, DAVID KOCHERGIN, MACIEJ BESTA, AND TORSTEN HOEFLE. **Spcl/Fanns-Benchmark**. SPCL, February 2026. 54
- [106] JIAYANG SHI. **Sjyouuuuug/Filterbenchmark**, January 2026. 54
- [107] PINECONE. **Pinecone-Io/VSB**. Pinecone, February 2026. 54
- [108] QDRANT. **Qdrant/Vector-Db-Benchmark**. Qdrant, March 2026. 54
- [109] MYSCALE. **Myscale/Vector-Db-Benchmark**. MyScale, July 2025. 54
- [110] PAUL GROSS. **Cwida/Benchmark**. CWI Database Architectures Group, November 2025. 54
- [111] ROBERT SCHULZE, TOM SCHREIBER, ILYA YATSISHIN, RYADH DAHIMENE, AND ALEXEY MILOVIDOV. **ClickHouse - Lightning Fast Analytics for Everyone**, 2024. 80
- [112] JOHN OUSTERHOUT. **Always Measure One Level Deeper – Communications of the ACM**, July 2018. 81

A

Dataset Origins

This appendix continues Section 6.1.3’s discussion of the datasets by explaining how the custom datasets were constructed. Refer to Table 6.2 for an overview of all datasets’ properties. The `utils` directory in the thesis’ `vss-benchmarks` repository¹ contains the utility scripts used to create the datasets. We used DuckDB’s exact search (ENNS) to create the top- k ground truth for all these datasets’ test queries. Afterward, we add the required VectorDBBench classes to represent these new datasets.

SIFT

The SIFT dataset is available in VectorDBBench as a capacity workload but not yet for non-filtered search. We removed 1000 rows from the original dataset’s 5 000 000 rows, using them as test queries. Hence, the dataset has 4 999 000 rows.

agnews

The `agnews mxbai` dataset was originally used to benchmark the `PDXearch` library. The library’s `BENCHMARKING.md` refers to a Google Drive folder² from which the dataset was downloaded. We used a script (which is available in the benchmarking repository) to transform the `hdf5` dataset to VectorDBBench’s format and added it as a dataset and search workload.

arxiv-for-fanns

The `arxiv-for-fanns` dataset used in this thesis is a modified version of the original `arxiv-for-fanns-large` dataset [8, 102]. Unfortunately, `pgvector` limits full precision embeddings to a maximum of 2000 dimensions³. Thus, the original `arxiv-for-fanns` dataset’s 4096 dimensions do not fit without using lower precision `pgvector` features. To limit the scope of the thesis, we

¹The repository that archives the experimental setup, scripts, and measurements can be found at: <https://github.com/Noorts/vss-benchmarks/tree/ca07ec1437d1d8c0ed6885a81007f4ddf616703a>.

²The `PDXearch` library’s dataset folder can be found at: <https://drive.google.com/drive/folders/1f76UCrU52N2wToGMFg9ir1MY8ZocrN34?usp=sharing>.

³For more information about `pgvector`’s dimensionality limitations, refer to [38] and <https://github.com/pgvector/pgvector/issues/461>.

A. DATASET ORIGINS

opted not to use those. Instead, we used a dimensionality reduction technique called principal component analysis (PCA) to reduce the dimensions of the dataset and query vectors from 4096 down to 1024. An *explained variance* of 99.99 % indicates that we did not lose too much of the similarity signal stored in the embeddings. In addition to the dimensionality reduction, we also sampled 1 200 000 embeddings from the original 2 735 264 embeddings to make the dataset more realistic to process on a modern laptop. Furthermore, we used a random permutation of the dimensions' order to ensure that the dimensions with the highest explained variance do not come first. This would otherwise give the PDXearch pruning algorithm an unfair advantage.

Cohere6M

Out of the box, VectorDBBench provides a 10 000 000 embeddings by 768 dimensions dataset called Cohere10M. This is ~28.6 GiB of raw embeddings, making it too big to process on the 48 GiB machine. The suboptimal index construction implementation causes memory spikes by keeping the dataset in memory multiple times (i.e., once in the global IVF's embedding buffer, once (partially) in Super K-Means' sampling buffer, etc.). Even with swap, these spikes cause the macOS OOM killer to terminate the process. Optimizing peak memory usage during index creation and adding out-of-core support is part of future work.

We focused on keeping the dataset as large as possible while supporting our own PDXearch implementations.¹ We sampled a new shuffled dataset from the original shuffled 10 000 000 embeddings dataset. At 6 021 120 embeddings (17.17 GiB at full precision (F32); precisely 49 full row groups), index construction was relatively stable. Beside this new shuffled dataset, we also created a dataset with the same 6 021 120 rows, but sorted by its *id* column.

This dataset was set up specifically for filtered search evaluation (Section 6.4.2). We use range predicates on the *id* column. The test queries were created by sampling the embeddings from the original dataset's 1000 test queries, and by creating synthetic range predicates that target a specific filtering percentage. The range predicate's bounds (i.e., start and end) are random, but the length of the range ensures the filtering percentage is met. We have 21 distinct filtering percentage targets (e.g., 1 %, 10 %) and we created 500 test queries for each target.

¹For the large Cohere6M dataset, we ignored DuckDB VSS because the memory overhead of the HNSW index is likely too high. pgvector was also excluded because based on previous experiments we observed its throughput was too low.