Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL

Yunan Zhang   Shige Liu   Jianguo Wang
Purdue University
{*zhan4404, liu3529, csjgwang*}*@purdue.edu*

***Abstract*—High-dimensional vector data is gaining increasing importance in data science applications. Consequently, various database systems have recently been developed to manage vector data. These systems can be broadly categorized into two types: specialized and generalized vector databases. Specialized vector databases are explicitly designed and optimized for storing and querying vector data, while generalized vector databases support vector data management within a relational database like PostgreSQL. It is expected (and confirmed by our experiments) that generalized vector databases exhibit slower performance. However, *it is not clear whether there are fundamental limitations (or just implementation issues) for relational databases to support vector data management*.**

This paper aims to answer this question. We chose PostgreSQL as a representative relational database due to its popularity. We focused on PASE, as it is a high-performance and open-sourced PostgreSQL-based vector database. We analyzed the source code of PASE and compared its performance with Faiss, a high-performance and open-sourced specialized vector database, to identify the *underlying root causes* of the performance gap and analyze how to bridge the gap. Based on our results, we provide insights and directions for building a future generalized vector database that can achieve comparable performance to a high-performance specialized vector database.

***Index Terms*—Vector Databases; Vector Similarity Search; Specialized Vector Databases; Generalized Vector Databases**

I. Introduction

High-dimensional vector data (10s to 1000s of dimensions) and query processing over vector data (a.k.a vector similarity search) have traditionally played a fundamental role in various areas, including data management, data science, data mining, information retrieval, and machine learning. Recently, there has been a new wave of interest in managing high-dimensional vector data. This is largely due to the prevalence of vector embedding, which converts unstructured data into learned vector representations, e.g., item2vec [1], word2vec [2], doc2vec [3], and graph2vec [4]. Many advanced data analytics, such as image/video search, product recommendation, and chemical structure analysis, can be performed directly over these vectors using vector similarity search [5]–[10].

To efficiently manage vector data, a new breed of databases termed Vector Databases has been developed recently in both big tech companies and startups. For instance, Alibaba developed PASE [11] and AnalyticDB-V [12] to process and query vector data. Facebook open-sourced Faiss [13], [14] to support efficient vector similarity search. Database startups such as Zilliz (Milvus) [15] and Pinecone [16] also built their own vector databases to compete for market shares, especially when vector databases can be used in ChatGPT [17], [18].

Those vector databases can be broadly classified into two categories: *Specialized Vector Databases* and *Generalized Vector Databases*. Specialized vector databases are designed from scratch, particularly to manage vector data, following the design principle of “one-size-does-not-fit-all” [19]. Examples include Faiss [13], [14], Milvus [20], and Pinecone [16]. They can usually achieve excellent performance through customized optimizations because they treat vector data as a first-class citizen within the systems. On the other hand, Generalized Vector Databases are built to support vector data management inside a relational database (e.g., PostgreSQL), following the design rationale of “one-size-fits-all” [21]. Examples include PASE [11], AnalyticDB-V [12], and pgvector [22]. These systems integrate vector data management into the existing relational database ecosystem to provide better usability (e.g., using SQL language to query vector data), reuse certain functionalities, eliminate data silos, and reduce operational costs with a single unified system. However, the downside is that they might sacrifice some performance and efficiency in querying and storing vector data to some extent to accommodate the relational model, which is expected and confirmed in our experiments (Sec. V and Sec. VII).

Motivation. So, a natural question is, *are there fundamental limitations (or just implementation issues) for relational databases to support vector data management?* Answering the above question is important to understand whether it is possible for a generalized vector database to achieve high performance and generality at the same time (even in the future). Unfortunately, existing studies have not answered this question.

Overview. This paper aims to answer the above question for the first time. We have chosen PostgreSQL as our representative relational database due to its widespread popularity. Indeed, many generalized vector databases, including PASE [11], AnalyticDB-V [12], and pgvector [22], are built

The first two authors contributed equally. Jianguo Wang is the corresponding author.

on the PostgreSQL foundation. In particular, we focus on PASE [11], because it is open-sourced and achieves high performance (see Figure 2).

To analyze PASE, we choose Faiss [13], a high-performance open-sourced specialized vector database, as a reference for comparison. We delve into the source code of each system (as they are open-sourced) and profile the performance metrics when running different benchmarks on real datasets. Then, we identify the *root causes* that contribute to the slow performance in PASE and analyze how to bridge the gap. Through this investigation, we are able to determine whether there are any fundamental limitations in supporting vector search within a relational database.

Contributions. We make the following contributions:

1) We present the first study that identifies **seven non-trivial root causes** (summarized in Sec. IX) for the performance issues in a high-performance generalized vector database (i.e., PASE [11]). We believe that these findings can be applied to other generalized vector databases. Overall, this study significantly advances the understanding of generalized vector databases.

2) By analyzing the root causes, we contribute an interesting message to the database community: **It is feasible to bridge the performance gap by building a new generalized vector database in the future that achieves comparable performance to a highly optimized specialized vector database, which means that there are no fundamental limitations in supporting vector search in relational databases.** We further present actionable guidelines for building such a system.

Open-source. The source code of the work is available at https://github.com/YunanZzz/VecDB-ICDE24.

II. Background

A. Vector Similarity Search

Vector similarity search (a.k.a high-dimensional nearest neighbor search) has been studied extensively in the past [23]–[35] due to the fundamental importance in many real-world applications. Let $v = [v_0, v_1, ..., v_{d-1}]$ be a d-dimensional vector and n be the number of vectors in a dataset $\mathcal{D}$, given a query vector q, the problem of vector similarity search returns similar vectors (e.g., top-k) to q from the dataset $\mathcal{D}$ based on a similarity (or distance) function, e.g., Euclidean distance or cosine distance.

B. High-dimensional Indexes

Vector similarity search is challenging because d and n are usually large, e.g., d can be 10s to 1000s of dimensions, and n can be millions to even billions of vectors, depending on different applications. Thus, it is computationally-intensive to find exact answers due to the known "dimensionality curse" issue [36]. Therefore, most works (including all the specialized and generalized vector databases that we are aware of) resort to approximate solutions for performance reasons by designing optimized high-dimensional indexes as we describe next.

In the literature, there are mainly four types of high-dimensional indexes: quantization-based indexes (e.g., IVF_FLAT [23], [24], IVF_PQ [24], IVF_SQ8 [23], and [25], [26]), graph-based indexes (e.g., HNSW [27], NSG [28], Rand-NSG [29]), LSH-based indexes [30]–[32], and tree-based indexes [33], [34]. However, LSH-based indexes tend to have lower accuracy than quantization-based approaches for large-scale data [11], [12]. Tree-based indexes are not scalable to high dimensions [12], [24]. A recent benchmark [37] shows that IVF_FLAT, IVF_PQ, and HNSW are competitive among all the indexes and are also implemented in most vector databases (e.g., Faiss [13], [14], Milvus [20], Pinecone [16], PASE [11] and AnalyticDB-V [12]). Thus, we focus on these three indexes in this paper, and we explain them below.

IVF_FLAT [23], [24]. Both IVF_FLAT and IVF_PQ belong to the category of quantization-based indexes. The main idea is to map a data vector $\mathbf{v}$ to a codeword $z(\mathbf{v})$ according to a quantizer z. In practice, we use the K-means clustering algorithm to construct the codebook (of size c) by treating each centroid as a codeword and assigning each data vector to its closest centroid. Thus, there are two phases for index construction. (1) **Training phase**: train the centroid vectors (via K-means) using a collection of sample vectors with a sample ratio *sr*; (2) **Adding phase**: assign each data vector to the nearest centroid and insert it to the corresponding bucket (a.k.a inverted list or cluster) [24].

Answering a vector similarity search of a query $\mathbf{q}$ takes two steps. First, it determines the closest n_{probe} (user-input parameter) buckets based on the similarity between the query $\mathbf{q}$ and the centroid of each bucket. Second, it only scans each relevant bucket to find the promising vectors.

IVF_PQ [24]. IVF_PQ is similar to IVF_FLAT, with the only difference of applying another layer of quantization (called product quantization (PQ) [24], [38]) inside each bucket to reduce space overhead. The main idea of product quantization is to partition each vector into m disjoint sub-vectors and apply the K-means clustering within each sub-space to reduce space overhead. A codebook with c_{pq} codewords (PQ-refined clusters) is constructed in each sub-space so that each vector can be encoded using $m \cdot \log c_{pq}$ bits [24], [38], which can significantly reduce space with the downside of lower recall rate when compared with IVF_FLAT.

HNSW [27]. HNSW is a graph-based index that constructs a proximity graph. It treats each vector as a *vertex* and pre-computes certain nearest neighbors of each vertex as *edges* [27], [28]. The underlying intuition is that a neighbor's neighbor is likely to be a neighbor as well.

HNSW [27] is a hierarchical graph with multiple levels where higher levels are shortcuts over lower levels (similar to a skip list) and each level is a graph structure. The lowest level contains all the vectors, while other levels contain a subset of vectors, determined by a probability function. Graphs at different levels are interconnected through common vectors. Let us consider the graph construction for the lowest level, as the process for other levels is similar. Let *bnn* be a base neighbor count, then every vertex (vector) in the lowest-level graph usually has $2 * bnn$ neighbors [27] (while it has *bnn* neighbors for graphs in other levels). Whenever a new vertex **v** is inserted, a priority queue of length *efb* is created to compute the nearest neighbors between **v** and the current vertices inserted so far to determine the edges to **v**.

Similar to skip lists, HNSW [27] begins the search process from the top and proceeds downward. At the bottom level, the process starts from an entry point, which is added to a priority queue with a size of *efs*. This queue prioritizes vertices based on their distance to the query. The system extracts a vertex from the priority queue, updates the top-k candidates, and inserts the neighbors of the extracted vertex into the queue for future exploration. The algorithm stops when the vertex extracted from the queue is worse than the top-k candidates that have been found.

Compared to quantization-based indexes, graph-based indexes usually provide better performance and accuracy, albeit at the cost of higher memory overhead and longer index construction time.

C. Specialized Vector Databases

Specialized vector databases are designed to efficiently implement those high-dimensional indexes and address other system-related issues (e.g., query interface and memory management) into a full-fledged system that can be easily used by customers. Examples include Faiss [13], Milvus [20], Pinecone [16], Vespa [39], Qdrant [40], and Weaviate [41]. Since those systems are purposely built for vector data, they can usually achieve high performance by implementing indexes in the most efficient way.

D. Generalized Vector Databases

In contrast to specialized vector databases that are built from scratch purely for optimizing vector data management, generalized vector databases implement vector similarity search inside a relational database (e.g., PostgreSQL), following the design principle of one-size-fits-all [21].

Main Idea. The main idea of generalized vector databases is to store vector data as a column within a relational table and build a high-dimensional index (e.g., IVF_FLAT [23], [24], IVF_PQ [24], and HNSW [27]) for that column. Then, the vector similarity search is represented as a SQL query, which will be further compiled into an optimized query plan that will call the pre-built high-dimensional index to find similar vectors. Thus, vector similarity search is seamlessly integrated into the SQL database ecosystem in this generalized approach.

Challenges. It is non-trivial to build a generalized vector database because there are many challenges to be addressed in order to be compatible with the existing relational database. For example, how to represent the vector data? How to extend the SQL syntax to represent vector similarity search? How to build a new high-dimensional index that follows the relational database's structure (e.g., page structure and memory layout)? How to make the newly-built index recognizable by the SQL query optimizer? How to configure the internal parameters of a high-dimensional index (e.g., number of clusters and probes in IVF_FLAT [23], [24]) using SQL? How to define and specify the similarity function using SQL?

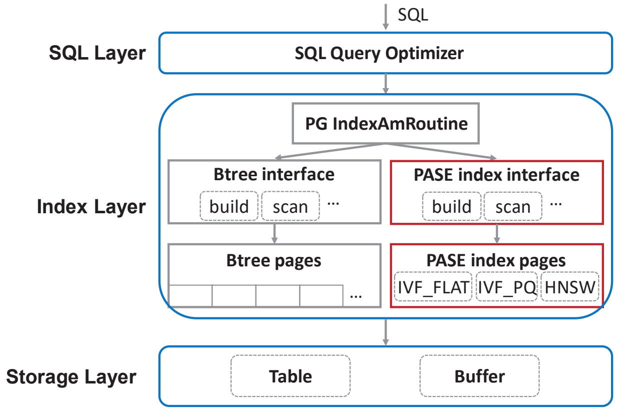


Fig. 1: System Architecture for PASE [11]

Systems. There are a few generalized vector databases that we are aware of, e.g., pgvector [22], PASE [11], AnalyticDB-V [12], and ClickHouse [42]. Except for ClickHouse, all the other three vector databases are implemented based on PostgreSQL, though they have different implementations.

E. PASE

We provide the background of PASE [11] (an open-sourced PostgreSQL-based vector database) as it is the main focus of this paper due to its high performance (see Sec. III). Figure 1 shows the system architecture of PASE with three major components: SQL layer, index layer, and storage layer.

SQL Layer. It represents vector similarity search using a SQL query by extending the syntax of the traditional SQL language. In PASE [11], vector data is represented using the array data type (e.g., `float[]`) provided by PostgreSQL.[1]

Assuming there is a table `T` with two columns *id* (integer type) and *vec* (vector type), we can use the following SQL to create it:

```sql
CREATE TABLE T (id int, vec float[]);
```

Assuming the query vector is $[0.1, 0.2, 0.3]$, then we can find the top-10 similar vectors in PASE [11] using:

```sql
SELECT   id
FROM     T
ORDER BY vec <op> '0.1,0.2,0.3'::PASE ASC
LIMIT    10;
```

The operator `<op>` is a special operator defined in PASE [11] to compute the similarity between two vectors. The `ORDER BY` statement will sort results based on the similarity to the query vector. The `LIMIT 10` returns the top 10 results.

Index Layer. To speed up query processing, PASE [11] relies on indexes. For example, we can use the following SQL to create the IVF_FLAT index on the column *vec*:

```sql
CREATE INDEX ivfflat_idx ON T
       USING ivfflat_fun(vec)
```

[1] Vector data can also be implemented as a user-defined data type for encryption reasons as mentioned in [11].

```
      WITH  (distance_type = 0,
             dimension = 128,
             clustering_params = "10,256");
```

We can also specify the similarity function and the internal parameters of IVF_FLAT. For example, the above SQL shows that the similarity function is Euclidean distance (type 0). The parameter 10 means that the sampling ratio is 10/1000 (i.e., 0.01). The parameter 256 means the number of clusters in IVF_FLAT.

However, in order for a newly created index to be compatible with the existing SQL query plan, the index implementation has to follow certain rules. First, it needs to implement the interfaces, e.g., `build()`, `insert()`, `delete()`, `scan()`, via PostgreSQL's IndexAmRoutine [43]. Second, the index needs to follow PostgreSQL's index page structure in order to be accessed via the buffer manager and storage engine. PASE [11] presents the detailed page structure for each index.

Storage Layer. PASE [11] stores vector data in a table in the same way as other attributes. In PASE, tables and indexes are stored on disk, but frequently accessed pages are cached in memory via the buffer manager.

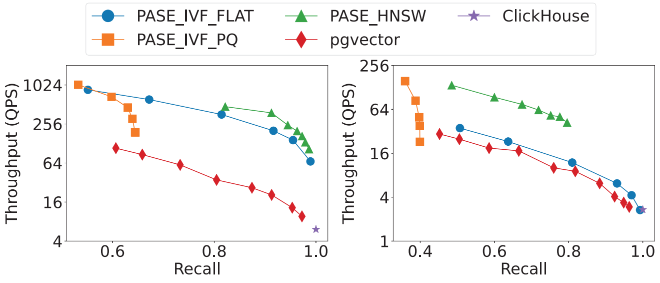
III. Experimental Methodology and Scope

Experimental Methodology. The goal of this work is to understand whether there are fundamental limitations in supporting vector search in relational databases. To do so, we choose PostgreSQL as a representative relational database for two reasons: (1) PostgreSQL is a classic and widely used relational database; (2) Many generalized vector databases (e.g., PASE [11], AnalyticDB-V [12], pgvector [22]) are based on PostgreSQL.

In particular, this paper focuses on PASE [11], a vector database based on PostgreSQL, because (1) PASE is open-sourced so that we can analyze the codebase and profile the execution time to understand the root causes; (2) PASE exhibits the highest performance among all open-sourced generalized vector databases (see Figure 2). Thus, this paper investigates whether there are any limitations in PASE to support vector search and shows how to overcome the limitations (if any).[2] To achieve this, it is crucial to select a representative specialized vector database as a reference point and compare PASE with it to understand the potential limitations in PASE. In this paper, we use Faiss [14] as the reference specialized vector database, because Faiss achieves high performance compared to other specialized vector databases and also it is open-sourced.

We conduct extensive experiments to compare PASE [11] and Faiss [14], using the same index type and parameters, to show the performance difference. Then we delve into the source code of each system and profile the performance metrics to *identify the root causes of the performance gap and analyze how to bridge the gap*. Through this analysis, we can determine if there are fundamental limitations.

Scope of This Work. Note that the purpose of this paper is not to benchmark various specialized vector databases or generalized vector databases. Instead, the objective is to identify any potential limitations in supporting vector search within a relational database. It requires identifying the root causes of the performance gap of a representative generalized vector database (i.e., PASE) and a representative specialized vector database (i.e., Faiss) as explained in the methodology.



(a) SIFT1M (b) GIST1M

Fig. 2: Comparing Open-Sourced Generalized Vector Databases. Sec. IV-A shows the details of the experimental setup, e.g., datasets, hardware configurations, and index parameters. Note that we do not show AnalyticDB-V [12] as it is a closed-sourced system.

We limit our discussions to a single-node setting because PASE [11] (as well as Faiss [13]) is a single-node database. However, if the entire vector dataset cannot fit into a single node, we can adopt the sharding approach used in distributed databases [44] to partition the vector data among different nodes, each running a vector database. This is not expected to change our conclusions. Also, this work does not consider GPU and GPU-based vector similarity search [14], [45], as PASE does not support GPU. Lastly, we assume that all the vector data and indexes can fit into main memory because: (1) Faiss, like other specialized vector databases, is a main-memory system that does not support out-of-memory data management; (2) Main-memory databases are becoming increasingly popular today and there is an emerging wave of main-memory databases [46]–[49], as memory capacity has grown large enough for many applications and the price per GB has significantly decreased.

IV. Experimental Setup

A. Experimental Platform

We conduct all the experiments on a Linux server with a 152-core Intel Xeon CPU (2.40GHz), 192GB DRAM, and 2TB SATA SSD, similar to the server configuration in PASE [11].

Our server has enough memory to keep the entire vector data and index in main memory when comparing PASE [11] and Faiss [13], as Faiss is an in-memory vector database. Before measuring the execution time, we run a warm-up experiment so that the data and index are stored in main memory and we run three more times to obtain the average time.

By default, we follow PASE [11] to use a single thread to run the experiments. But we also conduct multi-thread experiments in Sec. V-D and Sec. VII-D.

[2] Note that our findings are also applicable to many other databases as explained in Sec. IX.

TABLE I: Statistics of Real-world Datasets

Dataset	# Dimensions	# Vectors	# Queries
SIFT1M [51]	128	1,000,000	10,000
GIST1M [51]	960	1,000,000	1,000
Deep1M [8]	256	1,000,000	1,000
SIFT10M [51]	128	10,000,000	10,000
Deep10M [8]	96	10,000,000	10,000
TURING10M [52]	100	10,000,000	10,000

TABLE II: Parameters

	Meaning and Default Value
k	top-k vector similarity search
	Default Value: 100
c	number of clusters in IVF_FLAT and IVF_PQ
	Default Value: 1000 in SIFT1M/GIST1M/DEEP1M; 3162 in SIFT10M/DEEP10M/TURING10M (square root of the data size)
n_{probe}	number of selected candidate clusters for searching in IVF_FLAT and IVF_PQ (Sec. II-B)
	Default Value: 20
sr	sampling ratio of clustering in IVF_FLAT and IVF_PQ
	Default Value: 0.01
m	number of partitioned sub-vectors in IVF_PQ (Sec. II-B)
	Default Value: 16 in SIFT1M, SIFT10M and DEEP1M; 60 in GIST1M; 12 in DEEP10M; 10 in TURING10M
c_{pq}	number of PQ-refined clusters of a sub-vector space in IVF_PQ (Sec. II-B)
	Default Value: 256
bnn	base neighbor count for a vector node in HNSW (Sec. II-B)
	Default Value: 16
efb	priority queue length in HNSW index construction (Sec. II-B)
	Default Value: 40
efs	priority queue length in HNSW search (Sec. II-B)
	Default Value: 200

B. Datasets

Table I shows the statistics of the six datasets used in this work. They are also widely used in prior works, e.g., [11], [12], [14], [20], [50].

C. Parameters

We largely follow the terminology in [11], [24], [27] to introduce the parameters and set default values, shown in Table II. Unless otherwise stated, we use the default parameters for experiments.

D. Evaluation Metrics

We adopt commonly used metrics for measuring vector similarity search. Specifically, we use index construction time, index size, query time, and recall rate. Since this work focuses on understanding the potential limitations of PASE by identifying the root causes of the performance difference between PASE and Faiss, we will use the same index with the same parameters in the experiments to compare these two databases. As a result, the recall rate will be the same in PASE and Faiss and we omit the experimental results on recall rate due to space constraints.

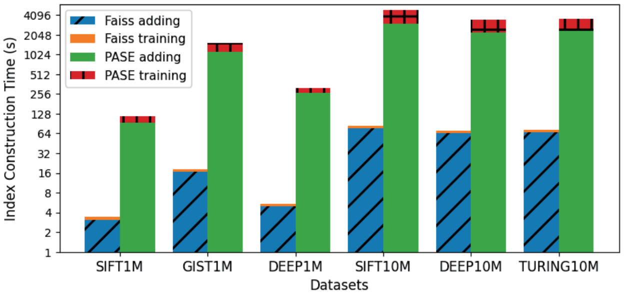


Fig. 3: Index Construction Time for IVF_FLAT

V. Evaluating Index Construction

In this section, we compare PASE [11] and Faiss [13] in terms of index construction time on IVF_FLAT (Sec. V-A), IVF_PQ (Sec. V-B), and HNSW (Sec. V-C).

A. IVF_FLAT

1) Overall Results

Figure 3 compares the IVF_FLAT index construction time of PASE and Faiss on the six datasets with the same parameters (in Table II). Figure 3 shows the overall time as well as training/adding time. To our surprise, PASE is **35.0**× $\sim$ **84.8**× **slower** than Faiss on different datasets even if they build the same IVF_FLAT index with exactly the same parameters. We will analyze it next in Sec. V-A2. In particular, the adding phase takes the majority of the time in both systems.

2) Investigation

Next, we investigate the underlying reasons that cause the huge performance gap.

We first exclude the impact of disk I/O because even if we use tmpfs, an in-memory file system in Linux that treats main memory as disk, the performance does not change much. We then use Perf [53] and Flame Graphs [54] to profile the execution time when building IVF_FLAT in PASE. The results show that `fvec_L2sqr_ref()` is the performance bottleneck in PASE. The function computes the Euclidean distance between two vectors.

This motivates us to carefully check the distance computing code in Faiss, especially in the adding phase. We find that Faiss uses the SGEMM (Single Precision General Matrix Multiplication) function in the BLAS library [55], which is a highly optimized low-level library for linear algebra operations.

Specifically, let $\mathbf{c}_0, \mathbf{c}_1, ..., \mathbf{c}_{K-1}$ be the K centroids trained by the K-means clustering algorithm, the adding phase assigns each base vector $\mathbf{x}_i$ to the nearest centroid $\mathbf{c}_i$. A straightforward solution, also used in PASE [11], is to compute the distance between $\mathbf{x}_i$ and all the centroids to find the closest centroid. However, Faiss [13] converts the problem to a matrix-matrix multiplication problem and uses SGEMM to speed up the process. By observing that $d^2(\mathbf{c}_i, \mathbf{x}_i) = \|\mathbf{c}_i\|^2 + \|\mathbf{x}_i\|^2 - 2\mathbf{c}_i \cdot \mathbf{x}_i$, Faiss computes all $\|\mathbf{x}_i\|^2$ and $\|\mathbf{c}_i\|^2$ (for all i) and then

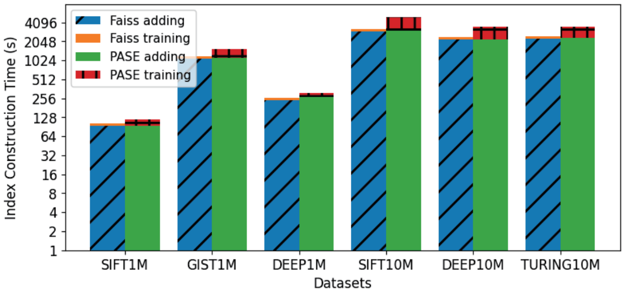


Fig. 4: Index Construction Time for IVF_FLAT Without SGEMM

uses matrix-matrix multiplication in SGEMM to compute all $\mathbf{c}_i \cdot \mathbf{x}_i$. Then Faiss stores those items (i.e., $\|\mathbf{x}_i\|^2$, $\|\mathbf{c}_i\|^2$, and $\mathbf{c}_i \cdot \mathbf{x}_i$) in a table and refers to the table whenever necessary to avoid redundant computing.

To verify our conjecture, we disable the SGEMM code in Faiss and use the same code as in PASE. Figure 4 shows the updated results. We can see that without the acceleration of SGEMM, the adding phase of Faiss IVF_FLAT consumes a similar time to PASE IVF_FLAT, which confirms the effectiveness of SGEMM. Note that there is a minor performance gap for the training phase in PASE IVF_FLAT and Faiss IVF_FLAT in Figure 4, that is because PASE and Faiss use a slightly different implementation of K-means to train the centroids.

3) *Insight*

This experiment delivers an interesting message that SGEMM, implemented in Faiss [13] but not in PASE [11], plays a critical role in the IVF_FLAT index construction. We denote this root cause as **RC#1**. It can improve performance **by an order of magnitude or more**. However, this is an implementation issue. This performance gap can be bridged because we can also implement the same optimization inside PASE, although it takes some engineering efforts.

B. *IVF_PQ*

1) *Overall Results*

Figure 5 evaluates the IVF_PQ index construction time of PASE and Faiss using the same parameters (described in Table II). It demonstrates that **Faiss outperforms PASE by 6.5**$\times$ $\sim$ **20.2**$\times$, showing a similar performance trend with the IVF_FLAT index in Figure 3. This is expected because both IVF_FLAT and IVF_PQ are quantization-based indexes.

2) *Investigation*

Following the experimental analysis in Sec. V-A, we investigate the impact of SGEMM. By disabling the SGEMM code of IVF_PQ in Faiss, we compare the IVF_PQ index construction time of PASE and Faiss, see Figure 6. It shows that the gap is negligible. Same as IVF_FLAT, the minor gap is due to the implementation difference in the K-means clustering and product quantization algorithm.

3) *Insight*

The gap is due to the same **RC#1** described in Sec. V-A3.

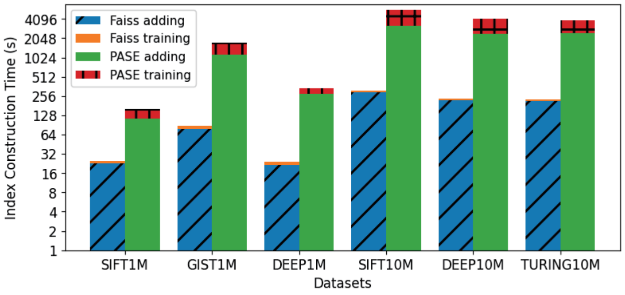


Fig. 5: Index Construction Time for IVF_PQ

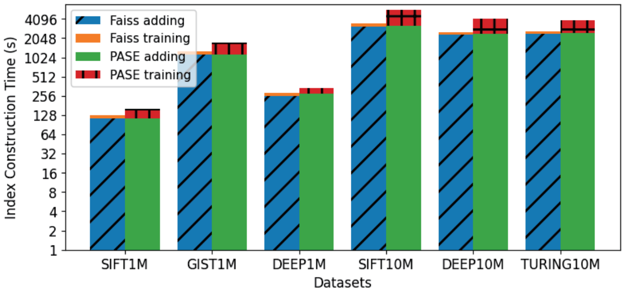


Fig. 6: Index Construction Time for IVF_PQ Without SGEMM

C. *HNSW*

1) *Overall Results*

Figure 7 compares the HNSW index construction time in PASE and Faiss using the same parameters (in Table II). As we can see from the figure, there is still a significant performance gap of **1.6**$\times$ $\sim$ **8.7**$\times$ on the six datasets. But it turns out that the root cause is different from that in IVF_FLAT and IVF_PQ as we analyze below in Sec. V-C2.

2) *Investigation*

Our initial trial is to investigate the SGEMM code following the analysis in Sec. V-A and Sec. V-B. However, it turns out that HNSW does not use SGEMM for acceleration even in Faiss, because HNSW is a graph-based index, which is different from IVF_FLAT and IVF_PQ (two quantization-based indexes).

We resort to Perf [53] to analyze the performance bottleneck. Table III illustrates the time breakdown for the

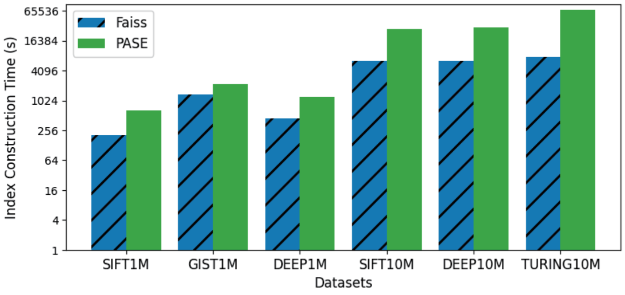
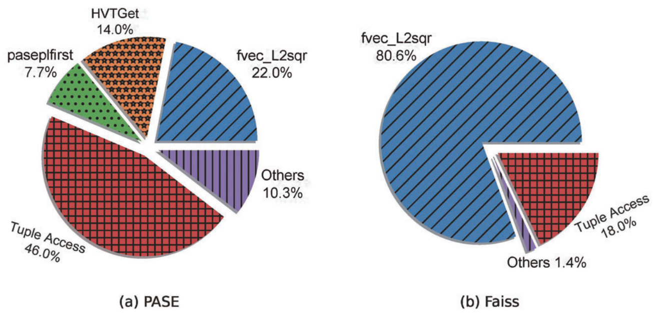


Fig. 7: Index Construction Time for HNSW

TABLE III: Time Breakdown of HNSW Building on SIFT1M

	SearchNb-ToAdd	AddLink	GreedyUp-dateNearest	Shrink-NbList	Others
PASE	75.55% 487.30sec	6.99% 45.09sec	6.69% 43.15sec	6.52% 42.05sec	3.82% 24.64sec
Faiss	70.37% 142.01sec	5.91% 11.93sec	10.86% 21.92sec	8.73% 17.62sec	4.13% 8.33sec



(a) PASE (b) Faiss

Fig. 8: Time Breakdown in `SearchNbToAdd()`

HNSW index construction time on SIFT1M in PASE and Faiss. It includes both the relative and absolute time consumption. Table III shows that `SearchNbToAdd()` is the most time-consuming part. That function searches neighbors for a newly inserted vector to construct HNSW. We can see that PASE `SearchNbToAdd()` is much slower than Faiss `SearchNbToAdd()` in Table III.

To investigate why, we further show the time breakdown of `SearchNbToAdd()` in Figure 8. It shows that Faiss spends most of the time (80.6%) on distance calculation (`fvec_L2sqr`) while PASE only spends 22% of its time on it. But in terms of the absolute time, they are quite similar (114 sec vs. 107 sec). However, PASE wastes a lot of time on other functions that are related to the architecture of relational databases. For example, PASE spends 46% of its time on `Tuple Access` and 14% on `HVTGet()` that are negligible in Faiss, where `Tuple Access` includes reading a page buffer from buffer pool in PASE and finding the right tuple based on page ID and tuple ID, and `HVTGet()` checks if a vector has been visited when searching neighbors. The overhead is introduced because PASE is based on PostgreSQL, which is a disk-optimized database system. However, Faiss is an in-memory vector database system that can directly locate the right tuple using a memory pointer without page-based indirection. Besides that, PASE spends 7.7% of its time on `paseplfirst()` to traverse the neighbors via indirection and functions calls, while Faiss can directly access the neighbors stored in an array.

3) Insight

This experiment shows another interesting root cause that the memory management in PASE (inherited from PostgreSQL) incurs significant overheads that make the construction time of HNSW in PASE much slower than that in Faiss. We denote this root cause as **RC#2**. That is because PASE (as well as PostgreSQL) is a disk-based database system. Even if the entire dataset and index are stored in memory, the memory manager still needs to go through the buffer pool for page indirection. That will make many simple functions slow, e.g., `HVTGet()`, `paseplfirst()`, and tuple accesses in Figure 8. But Faiss can directly access vectors using memory pointers for fast performance.

This gap is bridgeable as it requires a memory-optimized table design to bypass the buffer manager in relational databases.

Note that IVF_FLAT and IVF_PQ do not suffer from the memory management inefficiency because they are quantization-based indexes involving mostly sequential accesses (instead of random accesses as in HNSW). IVF_FLAT and IVF_PQ build indexes by keeping track of buffer IDs that will be inserted along with tuples, so they can directly access the buffer without indirection.

D. Impact of Parallelism

In this experiment, we study the impact of parallelism using multiple threads for index construction. Since PASE does not support parallelism for index construction, we focus on Faiss only to see whether parallelism can indeed improve performance, which can indicate the limitations of PASE. Due to space constraints, we only use SIFT1M to study the parallel construction of IVF_FLAT and IVF_PQ in Faiss.

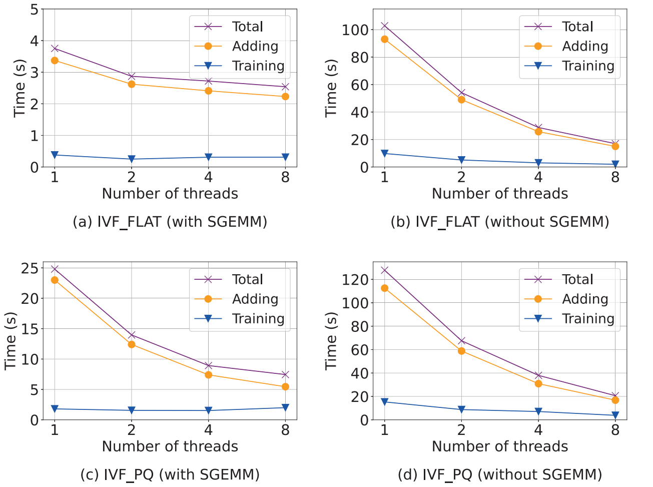
From the analysis in Sec. V-A and Sec. V-B, SGEMM can significantly affect the index construction process in IVF_FLAT and IVF_PQ. Thus, in this experiment, we show the results of disabling and enabling SGEMM for parallel index construction. Figure 9 shows the results by setting the number of threads as 1, 2, 4, and 8 and other parameters are default values in Table II. It shows that except for IVF_FLAT with SGEMM (Figure 9a), all the other figures scale very well with the number of threads. That is because when SGEMM is enabled in IVF_FLAT, the adding part is significantly reduced using matrix-matrix multiplication. Thus, even with multiple threads, the execution time will not reduce that much on that part. However, there is a clear performance improvement when SGEMM is disabled in IVF_FLAT (Figure 9b) because the adding part consumes a lot of time without SGEMM.

This experiment shows that parallelism is an important factor contributing to the performance gap between PASE and Faiss. We denote this root cause as **RC#3**. However, this gap can be bridged, because we can implement the same technique inside PASE, and we expect a similar performance speedup.

E. Impact of Parameters

In this experiment, we study the impact of key parameters on the performance gap of PASE and Faiss when building indexes. For IVF_FLAT and IVF_PQ, we vary the number of clusters (c) and set it as 100, 500, 1000 following previous work [11]. For HNSW, we vary the base neighbor count (bnn) and set it as 16, 32, and 64 following [27].

Figure 10 shows that the performance gap between PASE and Faiss increases as c and bnn increases on SIFT1M. For IVF_FLAT and IVF_PQ, when the number of clusters c



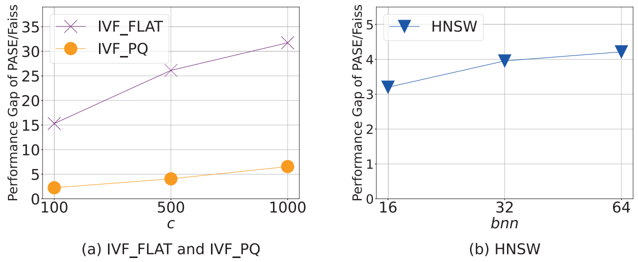
(a) IVF_FLAT (with SGEMM)

(b) IVF_FLAT (without SGEMM)

(c) IVF_PQ (with SGEMM)

(d) IVF_PQ (without SGEMM)

Fig. 9: Impact of Parallelism for Index Construction in Faiss



(a) IVF_FLAT and IVF_PQ

(b) HNSW

Fig. 10: Varying Parameters for Index Construction

increases, the computations required on K-means clustering will increase as well. Thus, the index build time in PASE and Faiss will increase. However, Faiss uses SGEMM for speedup as we analyze in Sec. V-A, the execution time will only increase mildly. Thus, the performance gap between the two is enlarged.

For HNSW, when the base neighbor count *bnn* increases, PASE spends more time on accessing neighbors and tuples because they have to go through the buffer manager, which incurs much more overhead than Faiss. Thus, the gap increases as well.

VI. Evaluating Index Size

In this section, we evaluate PASE [11] and Faiss [13] in terms of index size on IVF_FLAT (Sec. VI-A), IVF_PQ (Sec. VI-B), and HNSW (Sec. VI-C).

A. IVF_FLAT

Figure 11 shows the index size of IVF_FLAT in PASE and Faiss using the same parameters (explained in Table II). It shows that the IVF_FLAT index size is **almost the same** on the two systems.

That is because the page structure of IVF_FLAT in PASE can be well aligned with the memory representation. In particular, IVF_FLAT is stored in centroid pages and data pages where the centroid pages store centroid vectors and data pages store base vectors in the buckets of each centroid. Thus, there is nearly no difference between the size of PASE IVF_FLAT and Faiss IVF_FLAT.

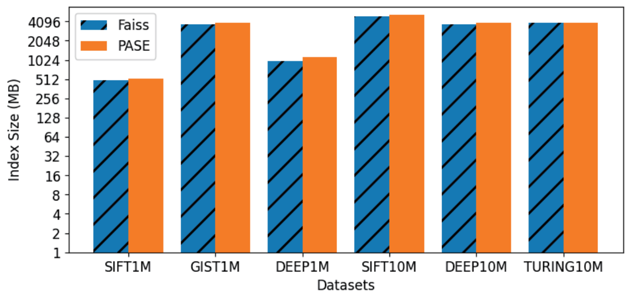


Fig. 11: Index Size for IVF_FLAT

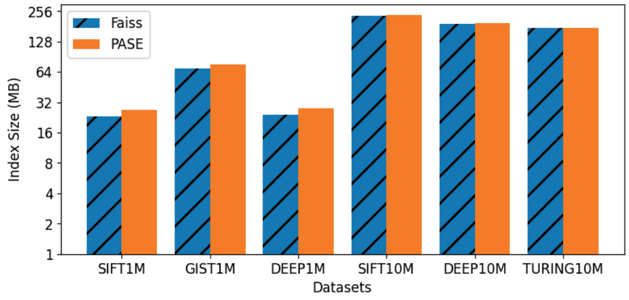


Fig. 12: Index Size for IVF_PQ

B. IVF_PQ

Figure 12 shows the index size of IVF_PQ in PASE and Faiss. Again, there is no obvious difference in the index size for the same reasons explained in Sec. VI-A.

C. HNSW

1) Overall Results

Figure 13 compares the HNSW index size in PASE and Faiss using the same parameters (in Table II). To our surprise, PASE consumes **2.9× ∼ 13.3× more space** than Faiss for the HNSW index.

2) Investigation

We find that there are two reasons that make PASE HNSW consume more space.

The first reason is that PASE HNSW allocates a 24-byte struct (`HNSWNeighborTuple`) for each vertex ID in

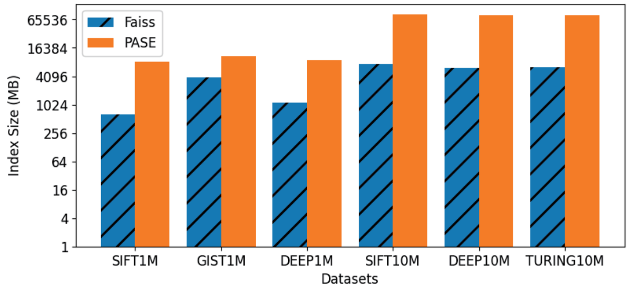


Fig. 13: Index Size for HNSW

TABLE IV: HNSW Index Size With 8KB/4KB Page Size in PASE

	SIFT1M	GIST1M	DEEP1M
8KB Page Size	8333 MB	11000 MB	8929 MB
4KB Page Size	4464 MB	7813 MB	5213 MB

the HNSW graph while Faiss HNSW uses only 4 bytes as expected. Specifically, `HNSWNeighborTuple` includes two structs, i.e., `PaseTuple` and `HNSWGlobalId`, where `PaseTuple` contains an 8-byte char pointer to form a virtual link and `HNSWGlobalId` stores 12-byte information (i.e., `nblkid`, `dblkid` and `doffset`) to locate a vertex.[3]

The second reason is that the PASE HNSW always starts from a new page to store a new adjacent list (i.e., the neighbors of a vertex). This method can result in significant space waste if the number of neighbors is small. For instance, the default value of *bnn* in HNSW is 16, a common choice as suggested in [11], [27]. Thus, most vectors, residing in the lowest two levels of the HNSW graph, will only have 32 or 48 edges. This arrangement would consume 768 bytes or 1152 bytes respectively, which are significantly less than a page size, which is 8KB by default in PostgreSQL. To verify our conjecture, we reduce the page size from 8KB to 4KB and find that the PASE HNSW index size is reduced by (almost) half as shown in Table IV.

3) *Insight*

This experiment shows an interesting root cause that the disk-based page structure in PASE (based on PostgreSQL) can make the HNSW index size significantly higher than that in Faiss. We denote this root cause as **RC#4**. This gap is related to disk-based relational database systems. However, it is possible to minimize or even bridge the gap using a memory-optimized table design.

VII. Evaluating Search Performance

In this section, we compare PASE [11] and Faiss [13] in terms of search performance on IVF_FLAT (Sec. VII-A), IVF_PQ (Sec. VII-B), and HNSW (Sec. VII-C).

A. *IVF_FLAT*

1) *Overall Results*

Figure 14 shows the average query time using IVF_FLAT in PASE and Faiss on the six datasets. Table I shows the number of queries for each dataset. We use the same index parameters (described in Table II) to evaluate the performance difference between PASE and Faiss. Figure 14 shows that PASE is **2.0**× ∼ **3.4**× **slower** than Faiss on different datasets.

2) *Investigation*

Next, we analyze the root cause of the performance gap in Figure 14. Following the analysis in Sec. V-A, one might guess that SGEMM could be an important factor contributing to the performance gap. However, we find that SGEMM does not improve much performance in the search process because the bottleneck is no longer finding the relevant buckets. Instead, it is the search process within each bucket.

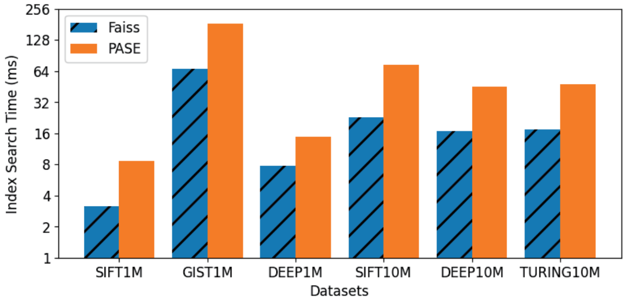


Fig. 14: Search Time for IVF_FLAT

TABLE V: Time Breakdown of IVF_FLAT Search on SIFT1M

	fvec_L2sqr	Tuple Access	Min-heap	Others
PASE	54.80% 4.69 ms	23.50% 2.01 ms	13.42% 1.15 ms	8.28% 0.71 ms
Faiss	94.96% 2.98 ms	1.80% 0.06 ms	0.29% 0.01 ms	2.95% 0.09 ms

We use Perf [53] to show the time breakdown on SIFT1M in Table V for the search process. Both relative and absolute time are recorded in the table. We can see that the majority of time is spent on distance calculation in both PASE and Faiss. However, it is interesting that Faiss spends 94.96% of its time on distance computing while PASE spends 54.80% on that. But in terms of the absolute time, PASE takes 1.6× more time than Faiss. After a careful look at the distance computing code in PASE and Faiss, we find that IVF_FLAT produces different centroids that lead to different clustering results in the two systems. To verify our conjecture, we use the same centroids and buckets produced in PASE and apply them to Faiss (termed Faiss∗) and re-run the experiments, see Figure 15. It shows that the gap becomes smaller between Faiss∗ and PASE.

Table V also reveals that PASE spends much more time on

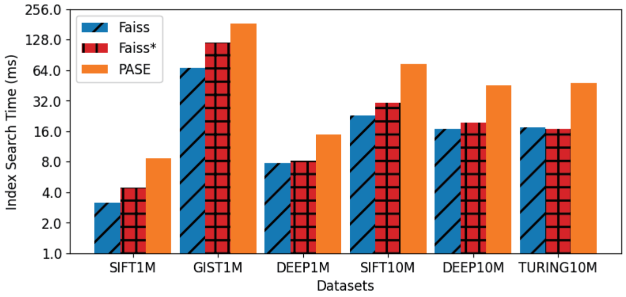


Fig. 15: Search Time for IVF_FLAT With Replaced Centroids (Faiss∗)

[3] Note that although sizeof(`PaseTuple`) = 8 and sizeof(`HNSWGlobalId`) = 12, the combined struct `HNSWNeighborTuple` takes 24 bytes due to memory alignment.

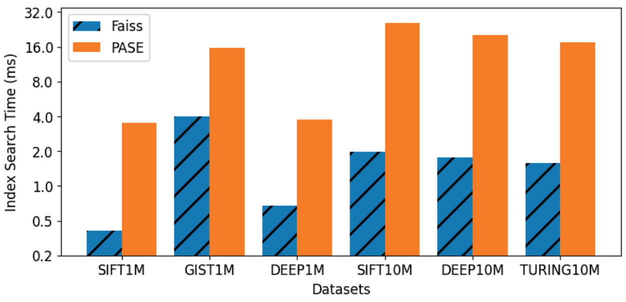


Fig. 16: Search Time for IVF_PQ

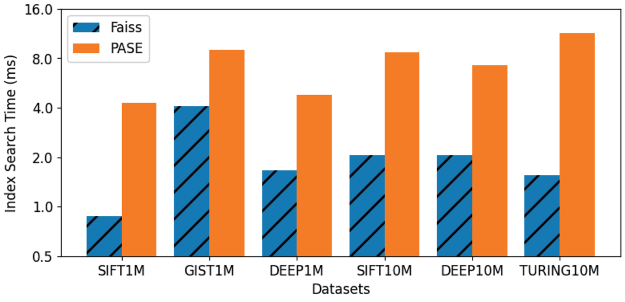


Fig. 17: Search time for HNSW

tuple accesses than Faiss does. This is related to the overhead of accessing a tuple as mentioned in Sec. V-C3.

Besides that, Table V shows another interesting factor on min-heap, which is used to quickly find out top-k smallest values once the distances are computed. PASE spends much more time than Faiss on that. We find that it is because of the heap size. In Faiss, the computed distances will be inserted into a heap of size k to find top-k similar vectors. However, PASE uses a heap of size n where n is the total number of vectors in the n_{probe} buckets.

3) Insight

This experiment finds a few factors that contribute to the performance gap between the search process of PASE and Faiss using IVF_FLAT. The first one is K-means implementation, which will affect the centroids and clusters in IVF_FLAT. We denote this root cause as **RC#5**. The second reason is about tuple access (memory management), which is the **RC#2** that we have mentioned in Sec. V-C3. The third reason is the top-k query optimization that uses a bigger heap size in relational databases. We denote this root cause as **RC#6**, which is related to the heap size in top-k computation. However, this is an implementation issue that can be fixed in PASE.

B. IVF_PQ

1) Overall Results

Figure 16 shows the average query time using IVF_PQ in PASE and Faiss on the six datasets. We use the same parameters (Table I and Table II) to evaluate the performance difference between PASE and Faiss for searching. We can see from the figure that PASE is **3.9**$\times$ $\sim$ **11.2**$\times$ **slower** than Faiss when searching IVF_PQ.

2) Investigation

Next, we analyze the root cause in Figure 16. We resort to Perf [53] to plot the time breakdown on SIFT1M following the approach in Table V. There are a few root causes that we have seen in Sec. VII-A, e.g., min-heap, distance computing, and tuple access.[4]

A new unique factor in IVF_PQ is its internal implementation on the precomputed table, which avoids redundant computing by storing the distances between partitioned sub-vectors and PQ-refined centroids. PASE IVF_PQ uses a straightforward implementation to compute the precomputed table while Faiss IVF_PQ uses an optimized solution that divides the task into computing L2 norms and inner product.

3) Insight

This experiment demonstrates a new factor – precomputed table – that can affect the performance of PASE IVF_PQ. We denote this root cause as **RC#7**. Additionally, there are other factors we have mentioned earlier that can affect the performance, e.g., min-heap (**RC#6**), distance computing (**RC#5**), and tuple access (**RC#2**). However, for the new factor (**RC#7**), we can bridge the gap by implementing the same optimizations inside PASE.

C. HNSW

Figure 17 shows the average query time using HNSW in PASE and Faiss on the six datasets. We use the same parameters (in Table II) to evaluate the performance difference between PASE and Faiss. Figure 17 shows that PASE is **2.2**$\times$ $\sim$ **7.3**$\times$ **slower** than Faiss on different datasets when searching HNSW.

We use Perf [53] to analyze the time breakdown of searching HNSW and it shows that the actual distance computing time of PASE and Faiss is almost the same. The performance gap is mainly due to the tuple access (**RC#2**) analyzed in Sec. V-C3.

D. Impact of Parallelism

In this experiment, we study the impact of parallelism on the query processing in PASE and Faiss using different indexes. We focus on intra-query parallelism that uses multiple threads to answer a single query.[5] Since both PASE and Faiss do not support parallel query processing on HNSW, we only show the results on IVF_FLAT and IVF_PQ (with the parameters in Table II), see Figure 18.

Figure 18 shows some interesting results. Faiss IVF_FLAT and IVF_PQ scale well with the number of threads but PASE IVF_FLAT and IVF_PQ do not, although they use the same idea to allocate multiple threads for searching different buckets. We find that, Faiss IVF_FLAT and IVF_PQ use a local heap to store the local top-k results when searching a bucket

[4] Note that tuple access does not play an important role in IVF_PQ because the index size of IVF_PQ is small and hence the time spent on tuple access is small as well.

[5] Note that inter-query parallelism is straightforward to achieve in both PASE and Faiss by partitioning queries to different threads.

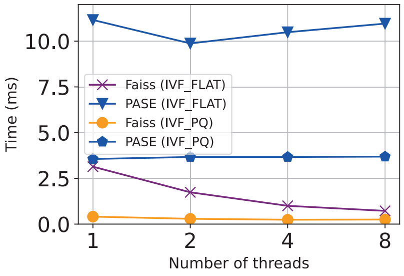
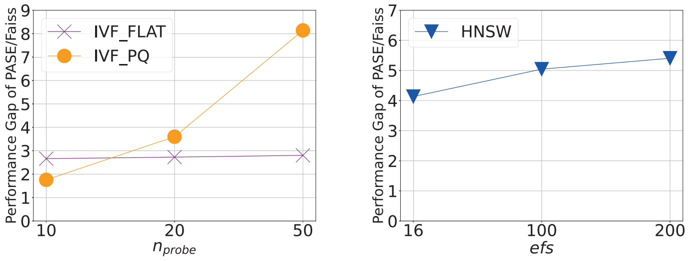


Fig. 18: Impact of Parallelism for Search Time



(a) IVF_FLAT and IVF_PQ

(b) HNSW

Fig. 19: Varying Parameters to Search Time Gap

and then combines those local ones into a global heap of top-k results (without locking). However, PASE IVF_FLAT and IVF_PQ directly use a global heap with locks to support concurrent insertions, which will lead to significant performance overhead. However, this gap is bridgeable by implementing the same parallelism inside PASE.

Since this root cause is related to parallelism, we categorize it into **RC#3** mentioned in Sec. V-D.

E. *Impact of Parameters*

In this experiment, we study the impact of key parameters on the performance gap of PASE and Faiss for the search process on SIFT1M. For IVF_FLAT and IVF_PQ, we vary the number of searched buckets (n_{probe}) as 10, 20, and 50. For HNSW, we vary the search queue length (*efs* defined in Table II) as 16, 100, and 200. Other parameters are set as default in Table II.

Figure 19 shows the performance gap between PASE and Faiss. For IVF_FLAT, the performance gap does not have a noticeable change as n_{probe} increases, because the search performance gap on IVF_FLAT is mostly determined by the root causes **RC#5**, **RC#2**, and **RC#6** as analyzed in Sec. VII-A, which do not change much as n_{probe} increases. For IVF_PQ, the performance gap increases along with n_{probe} because Faiss has an optimized implementation of the precomputed table that has already calculated the squared norms of the PQ-centroids in the training phase, thus, the precomputed table computation in the search phase will take less time. For HNSW, as analyzed in Sec. VII-C, distance computation takes only 12.56% of the time in PASE and 75.81% in Faiss. PASE spends most of its time in tuple access and other parts. As *efs* increases, the number of explored vectors also increases, so the time that PASE spends on non-distance computation will increase much more than that in Faiss. Thus, the performance gap is enlarged.

VIII. Related Work

Vector data and vector similarity search have been extensively studied in many areas such as data management, information retrieval, and machine learning, see [10] for a recent survey. Existing works can be roughly classified into two lines of research: *Algorithms* and *Systems*. Algorithms research focuses on the theoretical foundations (e.g., [56]–[58]) and efficient indexes (e.g., [24], [27]–[29], [31]–[35] as described in Sec. II-B) for vector similarity search. However, those works did not answer the question of whether there are fundamental limitations in supporting vector search in relational databases. Actually, those works provide a foundation and building blocks for the vector databases. Instead, systems research focuses on developing full-fledged vector data management systems for fast vector similarity search. Examples include Faiss [13], [14], Milvus [20], Pinecone [16], PASE [11] and AnalyticDB-V [12]. Our work falls into the category of systems research.

As described in the introduction (Sec. I), those vector database systems can be roughly classified into two categories: specialized vector databases (e.g., Faiss [13], [14], Milvus [20], and Pinecone [16]) and generalized vector databases (e.g., PASE [11] and AnalyticDB-V [12]). However, existing studies have not answered the question of whether there are fundamental limitations in supporting vector search in relational databases.

This work is relevant to the bigger picture of building specialized and generalized data systems. Although there are debates in designing specialized or generalized data systems, e.g., in graph systems [59], [60], in time series databases [61], [62], and in scientific databases [63], it is not clear about the situation in vector databases, and this paper contributes a data point in this arena.

IX. Conclusion and Future Work

A. *Summary*

The overall conclusion of this study is that **there is no fundamental limitation in using a relational database (e.g., PostgreSQL) to support efficient vector data management**.

Although PASE (a highly optimized generalized vector database based on PostgreSQL) is still slower than Faiss (a highly optimized specialized vector database), the performance gap is attributed to **implementation issues**. With careful implementation, it is feasible to bridge the gap, though it may require some engineering efforts. We show a list of actionable guidelines in Sec. IX-C to build such a system. In this way, it becomes possible to utilize a single relational database to support efficient vector search, achieving performance comparable to that of a highly optimized specialized vector database.

B. *Lessons*

This work identifies a collection of underlying reasons that cause the performance difference between PASE (a representative generalized vector database) and Faiss (a representative specialized vector database). We summarize the **root causes (RC)** of the performance gap as follows and discuss how to overcome the root causes and hence bridge the gap in Sec. IX-C.

- **RC#1: SGEMM Optimization**. SGEMM plays an important role in the index construction of vector databases. For example, Faiss leverages the SGEMM library [55] to improve performance while PASE does not. SGEMM is important to improve the index construction process of IVF_FLAT and IVF_PQ, see Sec. V-A and Sec. V-B.
- **RC#2: Memory Management**. Directly accessing data (including vectors and indexes) in memory without going through the buffer manager and page-based indirection is critical when the entire vector data and indexes are stored in memory. For example, even if PASE stores everything in memory, it still incurs significant overhead for memory management because PostgreSQL is a disk-oriented system. This can affect both index construction (e.g., HNSW in Sec. V-C) and search process (Sec. VII). Note that although this insight was known in main-memory relational databases [46], it is new in the setting of vector databases.
- **RC#3: Parallel Execution**. Supporting multiple threads to build indexes and search vector data can improve performance. For example, PASE does not support intra-query parallelism well due to the overhead of operating a shared data structure (e.g., heap). This can affect both index construction (Sec. V-D) and search (Sec. VII-D).
- **RC#4: Memory-centric Page Structure**. Use a memory-based layout (instead of page-based layout) when the vector data and indexes are stored in memory.[6] This can save a significant amount of space for HNSW (Sec. VI-C) but it does not have much effect on the index size of IVF_FLAT and IVF_PQ.
- **RC#5: K-means Implementation**. Different clustering implementations can also affect performance. For example, PASE and Faiss use a slightly different implementation for K-means, which can affect IVF_FLAT and IVF_PQ on search performance, see Sec. VII-A and Sec. VII-B.
- **RC#6: Heap Size in Top-k Computation**. PASE uses a heap size of n instead of k to compute k smallest values among n values for top-k similarity search. This will mostly affect the search performance, see Sec. VII.
- **RC#7: Precomputed Table Implementation**. Leveraging an optimized implementation for the precomputed table in IVF_PQ (e.g., used in Faiss but not in PASE) can affect the search performance of IVF_PQ, see Sec. VII-B.

Applicability and Transferability of the Root Causes. (1) Although the root causes are derived from analyzing PASE as it is a high-performance generalized vector database, the root causes are not specific to PASE or PostgreSQL alone. They are applicable to other relational databases aiming to support efficient vector search. For example, SGEMM, memory management, parallel execution, memory-centric page structure, k-means, heap size, and precomputed tables are all relevant to databases like Oracle and MySQL. Thus, we recommend that practitioners carefully examine these root causes when implementing vector search inside any relational database, as PostgreSQL is representative enough. (2) Although the root causes are derived by comparing PASE and Faiss, with Faiss serving as the reference specialized database due to its high performance in specialized vector databases, the lessons learned from this study remain valuable even if faster-than-Faiss databases or another specialized database is chosen. That is because this paper at least shows that it is feasible to build a generalized vector database to match Faiss's performance, which is a significant and non-trivial contribution that has not been explored in existing studies.

C. Future Direction: How to Bridge the Gap?

A follow-up of the work is how to overcome the root causes? In other words, how to build a new generalized vector database in the future that achieves comparable performance to a highly optimized specialized vector database? We show a few actionable guidelines and we are currently working on it.

Step#1: Start from an in-memory database. To overcome **RC#2** and achieve high performance, we may use an in-memory database, e.g., MonetDB [64], Hyrise [65], or SingleStore [66]. Note that if a disk-based relational database must be chosen, we recommend either using a memory-optimized table design, as in GaussDB [49], or implementing a standalone vector index in the memory region of the disk-based relational database. This can help overcome many performance overheads associated with disk-based relational databases.

Step#2: Enable SGEMM. The system shall enable SGEMM to bypass the overhead of **RC#1** and significantly improve the performance of index construction.

Step#3: Optimized top-k computation. The system shall use the proper heap size (i.e., k) for top-k computation to overcome the overhead introduced by **RC#6**.

Step#4: Parallelism. The system shall efficiently support both index construction and index search with multiple threads. This requires the implementation of the operator-level (e.g., vector search) parallelism in relational databases, which can bridge the performance gap due to **RC#3**.

Step#5: More optimized implementations. The system needs to reduce space amplification, support optimized K-means, and precomputed table as mentioned in **RC#4**, **RC#5**, and **RC#7**.

ACKNOWLEDGMENT

We sincerely thank Wen Yang, the first author of PASE [11], for helping us understand PASE in many ways. We also sincerely thank Zhehao Peng for his early participation in the project.

[6] If a disk-based page structure must be used, we recommend allowing multiple adjacent lists in HNSW to be stored on the same page instead of starting a new page for each new adjacent list.

References

[1] O. Barkan and N. Koenigstein, “Item2Vec: Neural Item Embedding for Collaborative Filtering,” in *International Workshop on Machine Learning for Signal Processing (MLSP)*, 2016, pp. 1–6.

[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” in *International Conference on Learning Representations (ICLR)*, 2013.

[3] Q. V. Le and T. Mikolov, “Distributed Representations of Sentences and Documents,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2014, pp. 1188–1196.

[4] M. Grohe, “Word2vec, Node2vec, Graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data,” in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2020, pp. 1–16.

[5] P. Covington, J. Adams, and E. Sargin, “Deep Neural Networks for YouTube Recommendations,” in *ACM Conference on Recommender Systems (RecSys)*, 2016, pp. 191–198.

[6] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, “The Rise of Deep Learning in Drug Discovery,” *Drug Discovery Today*, vol. 23, no. 6, pp. 1241–1250, 2018.

[7] A. C. Mater and M. L. Coote, “Deep Learning in Chemistry,” *Journal of Chemical Information and Modeling*, vol. 59, no. 6, pp. 2545–2559, 2019.

[8] A. Babenko and V. S. Lempitsky, “Efficient Indexing of Billion-Scale Datasets of Deep Descriptors,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2055–2063.

[9] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed Representations of Words and Phrases and their Compositionality,” in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2013, pp. 3111–3119.

[10] J. J. Pan, J. Wang, and G. Li, “Survey of Vector Database Management Systems,” *CoRR*, vol. abs/2305.01087, 2023.

[11] W. Yang, T. Li, G. Fang, and H. Wei, “PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension,” in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2020, pp. 2241–2253. [Online]. Available: https://github.com/alipay/PASE

[12] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, “AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 12, pp. 3152–3165, 2020.

[13] “Facebook Faiss.” [Online]. Available: https://github.com/facebookresearch/faiss

[14] J. Johnson, M. Douze, and H. Jégou, “Billion-Scale Similarity Search with GPUs,” *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[15] “Zilliz.” [Online]. Available: https://zilliz.com/

[16] “Pinecone.” [Online]. Available: https://www.pinecone.io/

[17] [Online]. Available: https://github.com/openai/chatgpt-retrieval-plugin/tree/main/datastore/providers

[18] S. Blanchard, “Supporting ChatGPT with Vector Databases for Optimized Efficiency and Accuracy,” 2023. [Online]. Available: https://www.dbta.com/Editorial/News-Flashes/Supporting-ChatGPT-with-Vector-Databases-for-Optimized-Efficiency-and-Accuracy-158503.aspx

[19] M. Stonebraker and U. Cetintemel, “”One Size Fits All”: An Idea Whose Time Has Come and Gone,” in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2005, pp. 2–11.

[20] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, “Milvus: A Purpose-Built Vector Data Management System,” in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2021, pp. 2614–2627.

[21] J. Dittrich and A. Jindal, “Towards a One Size Fits All Database Architecture,” in *Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 195–198.

[22] “pgvector.” [Online]. Available: https://github.com/pgvector/pgvector

[23] “Faiss Indexes.” [Online]. Available: https://github.com/facebookresearch/faiss/wiki/Faiss-indexes

[24] H. Jégou, M. Douze, and C. Schmid, “Product quantization for nearest neighbor search,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 33, no. 1, pp. 117–128, 2011.

[25] F. André, A. Kermarrec, and N. L. Scouarnec, “Cache Locality is not Enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 4, pp. 288–299, 2015.

[26] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, “Accelerating Large-Scale Inference with Anisotropic Vector Quantization,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2020, pp. 3887–3896.

[27] Y. 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 (PAMI)*, vol. 42, no. 4, pp. 824–836, 2020.

[28] C. Fu, C. Xiang, C. Wang, and D. Cai, “Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 12, no. 5, pp. 461–474, 2019.

[29] S. J. Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, “Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node,” in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2019, pp. 13 748–13 758.

[30] A. Gionis, P. Indyk, and R. Motwani, “Similarity Search in High Dimensions via Hashing,” in *International Conference on Very Large Data Bases (VLDB)*, 1999, pp. 518–529.

[31] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, “Intelligent Probing for Locality Sensitive Hashing: Multi-Probe LSH and Beyond,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, no. 12, pp. 2021–2024, 2017.

[32] B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, and C. S. Jensen, “PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 5, pp. 643–655, 2020.

[33] K. Lu, H. Wang, W. Wang, and M. Kudo, “VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 9, pp. 1443–1455, 2020.

[34] C. Silpa-Anan and R. I. Hartley, “Optimised KD-trees for Fast Image Descriptor Matching,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008, pp. 1–8.

[35] Y. Li, J. Wang, B. S. Pullman, N. Bandeira, and Y. Papakonstantinou, “Index-Based, High-Dimensional, Cosine Threshold Querying with Optimality Guarantees,” in *International Conference on Database Theory (ICDT)*, vol. 127, 2019, pp. 11:1–11:20.

[36] R. Weber, H. Schek, and S. Blott, “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces,” in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 194–205.

[37] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, “Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement,” *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 32, no. 8, pp. 1475–1488, 2020.

[38] R. Wang and D. Deng, “DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 13, pp. 3603–3616, 2020.

[39] “Vespa: The Open Big Data Serving Engine.” [Online]. Available: https://github.com/vespa-engine/vespa

[40] “Qdrant: Vector Search Engine for the next generation of AI applications.” [Online]. Available: https://github.com/qdrant/qdrant

[41] “Weaviate: An Open Source Vector Database.” [Online]. Available: https://github.com/weaviate/weaviate

[42] “Vector Search with ClickHouse.” [Online]. Available: https://clickhouse.com/blog/vector-search-clickhouse-p2

[43] “PostgreSQL Index Access Method Interface Definition.” [Online]. Available: https://www.postgresql.org/docs/11/indexam.html

[44] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[45] W. Zhao, S. Tan, and P. Li, “SONG: Approximate Nearest Neighbor Search on GPU,” in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2020, pp. 1033–1044.

[46] F. Faerber, A. Kemper, P. Larson, J. J. Levandoski, T. Neumann, and A. Pavlo, “Main Memory Database Systems,” *Foundations and Trends in Databases*, vol. 8, no. 1-2, pp. 1–130, 2017.

[47] M. Stonebraker and A. Weisberg, “The VoltDB Main Memory DBMS,” *IEEE Data Engineering Bulletin*, vol. 36, no. 2, pp. 21–27, 2013.

[48] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, “Hekaton: SQL Server’s Memory-optimized OLTP Engine,” in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2013, pp. 1243–1254.

[49] H. Avni, A. Aliev, O. Amor, A. Avitzur, I. Bronshtein, E. Ginot, S. Goikhman, E. Levy, I. Levy, F. Lu, L. Mishali, Y. Mo, N. Pachter, D. Sivov, V. Veeraraghavan, V. Vexler, L. Wang, and P. Wang, "Industrial Strength OLTP Using Main Memory and Many Cores," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 12, pp. 3099–3111, 2020.

[50] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 32, no. 8, pp. 1475–1488, 2020.

[51] "Datasets for Approximate Nearest Neighbor Search." [Online]. Available: http://corpus-texmex.irisa.fr/

[52] "Billion-Scale Approximate Nearest Neighbor Search Challenge." [Online]. Available: https://big-ann-benchmarks.com/

[53] "Perf: Linux Profiling with Performance Counters." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[54] "Flame Graphs." [Online]. Available: https://www.brendangregg.com/flamegraphs.html

[55] "BLAS (Basic Linear Algebra Subprograms)." [Online]. Available: https://netlib.org/blas/

[56] A. Rubinstein, "Hardness of Approximate Nearest Neighbor Search," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2018, pp. 1260–1268.

[57] M. Goswami, R. Jacob, and R. Pagh, "On the I/O Complexity of the k-Nearest Neighbors Problem," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2020, pp. 205–212.

[58] K. G. Larsen, T. Malkin, O. Weinstein, and K. Yeo, "Lower Bounds for Oblivious Near-Neighbor Search," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020, pp. 1116–1134.

[59] J. Fan, A. G. S. Raj, and J. M. Patel, "The Case Against Specialized Graph Analytics Engines," in *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[60] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Aggregation Support for Modern Graph Analytics in TigerGraph," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2020, pp. 377–392.

[61] "TimescaleDB: Postgres for Time-Series." [Online]. Available: https://www.timescale.com/

[62] "InfluxDB." [Online]. Available: https://www.influxdata.com/

[63] M. Stonebraker, A. Ailamaki, J. Kepner, and A. S. Szalay, "The Future of Scientific Data Bases," in *International Conference on Data Engineering (ICDE)*, 2012, pp. 7–8.

[64] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 40–45, 2012.

[65] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner, "Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2019, pp. 313–324.

[66] A. Prout, S. Wang, J. Victor, Z. Sun, Y. Li, J. Chen, E. Bergeron, E. N. Hanson, R. Walzer, R. Gomes, and N. Shamgunov, "Cloud-Native Transactions and Analytics in SingleStore," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2022, pp. 2340–2352.