Fast Vector Search in PostgreSQL: A Decoupled Approach

Jiayi Liu
Department of Computer Science
Purdue University
liu4127@purdue.edu

Yunan Zhang
Department of Computer Science
Purdue University
zhan4404@purdue.edu

Chenzhe Jin
Department of Computer Science
Purdue University
jin467@purdue.edu

Aditya Gupta
Department of Computer Science
Purdue University
gupta742@purdue.edu

Shige Liu
Department of Computer Science
Purdue University
liu3529@purdue.edu

Jianguo Wang
Department of Computer Science
Purdue University
csjgwang@purdue.edu

ABSTRACT

Vector databases have recently gained significant attention due to the emergence of LLMs. While developing specialized vector databases is interesting, there is a substantial customer base interested in integrated vector databases (that build vector search into existing relational databases like PostgreSQL) for various reasons. However, we observe a substantial performance gap between specialized and integrated vector databases, which raises an interesting research question: *Is it possible to bridge this performance gap*?

In this paper, we introduce PostgreSQL-V, a new system that enables fast vector search in PostgreSQL. Unlike prior work (e.g., pgvector) that inherits legacy overhead by reusing PostgreSQL's page-oriented structure, PostgreSQL-V adopts a novel architectural design that decouples vector indexes from PostgreSQL's core engine. This decoupling offers many benefits, such as directly leveraging native vector index libraries for high performance. However, it also introduces the challenge of index inconsistency, which we address with a lightweight consistency mechanism. Experiments show that PostgreSQL-V achieves performance on par with specialized vector databases and **outperforms pgvector by up to 8.9×** in vector search. To our knowledge, this is the first work to deliver specialized-level performance for vector search in PostgreSQL. We believe its insights can shed light on designing fast vector search in other relational databases, e.g., MySQL.

1 INTRODUCTION

Recently, there has been a tremendous surge of interest in vector databases [12, 17, 18, 23]. Examples include Faiss [3], Milvus [24], Pinecone [9], pgvector [7], SingleStore-V [13], and Oracle Vector Search [6]. Vector databases are designed to efficiently store and search high-dimensional vectors, with *vector similarity search* as the core operation for retrieving the top-k most similar vectors to a query.

The main driving force behind this wave of vector databases is LLMs, as they can address many inherent limitations of LLMs, such as hallucination, lack of domain knowledge, and the inability to incorporate up-to-date information [12, 17, 18, 23]. This is achieved through the paradigm of *Retrieval-Augmented Generation* (RAG), where vector databases store embeddings of domain-specific or real-time data, serving as an external knowledge base to provide relevant context to LLMs.

Current vector databases generally fall into two categories: Specialized and Integrated (or Extended) Vector Databases. Specialized systems are built from the ground up to manage vector data, such as Faiss [3][1], Milvus [24], and Pinecone [9]. They usually achieve excellent performance by treating vectors as first-class citizens.

In contrast, integrated vector databases extend existing database systems (mostly relational databases) to support vector search. Examples include pgvector [7], PASE [26], SingleStore-V [13], and Oracle Vector Search [6]. These systems offer several advantages, e.g., retaining the existing customer base, avoiding costly data migration, reducing silos and operational overhead with a unified platform, and leveraging SQL. However, these systems might compromise performance, as vector search is added as an afterthought.

Research Question. An interesting research question is: *Can we build integrated vector database systems that achieve performance comparable to specialized vector databases?*[2] This is the focus of this paper. In particular, we focus on vector search in PostgreSQL, as it is a widely used and representative relational database system.

State-of-the-Art and Limitations. To our knowledge, the two integrated vector databases that can match the performance of specialized vector databases are SingleStore-V [13] (based on SingleStore) and BlendHouse [16] (based on ByteHouse). However, both host databases (i.e., SingleStore and ByteHouse) use LSM-based storage engines, which present unique architectural benefits for supporting fast vector search that PostgreSQL-like systems do not have. Specifically, (1) The LSM-based storage engine facilitates directly calling native vector index libraries (e.g., Faiss [3]) without concern for transactional inconsistency because vector indexes are built only for on-disk immutable segments. For in-memory mutable segments, a full scan is performed. (2) Because vector indexes are built only on immutable segments, no in-place updates are required. This design is particularly friendly to vector indexes (e.g., HNSW and IVF_FLAT), as they are expensive to support in-place updates.

[1]Note that Faiss [3] is a popular vector index library (also used in Milvus [24]). In this paper, we also treat Faiss as a specialized vector database to simplify the presentation.
[2]Note that in this paper, when we talk about the potential performance gap between integrated and specialized vector databases, we assume that the comparison is conducted **under the same conditions** (e.g., the same vector index and same indexing parameters) to ensure fairness. They differ only in *how vector indexes are implemented* due to the potential architectural constraints of relational databases.

Regarding PostgreSQL, there are a few extensions that support vector search, e.g., pgvector [7], PASE [26], and pgvectorscale [8]. However, their performance is unsatisfactory (see Sections 4.2 and 4.3) for several reasons. First, these implementations follow a page-based structure to implement vector indexes in PostgreSQL. In contrast, modern vector databases primarily use memory-based vector indexes (e.g., HNSW, IVF_FLAT, and ScaNN [20]) to achieve high performance. This mismatch introduces significant performance overheads, such as page indirection and buffer memory overhead, whereas specialized vector databases like Faiss [3] can access data directly via memory pointers. Note that even for disk-based vector indexes like DiskANN [21], this approach also introduces substantial legacy overhead (Section 2). Second, existing implementations do not efficiently support vector index updates because PostgreSQL uses a heap-based storage engine that performs in-place updates, which are not friendly to vector indexes. Section 2 shows details.

Technical Overview. In this paper, we present PostgreSQL-V, an integrated vector database built within PostgreSQL that delivers performance comparable to specialized vector databases. At its core is a new design principle: *decoupling*. The key idea is to decouple vector indexes from PostgreSQL's storage engine, allowing them to be implemented flexibly and optimally without being constrained by PostgreSQL's architecture. The benefit is that, once decoupled, we can implement vector indexes using their native index libraries, which bypasses the legacy overhead of PostgreSQL's page-based structures. We can also support pluggable vector indexes to always incorporate the best implementations from the community. More benefits can be found in Section 3.1. However, decoupling introduces a major challenge, i.e., index inconsistency. To address this, we propose a new approach that separates the vector index metadata from its contents, leveraging PostgreSQL's transactional mechanism to manage metadata and a customized mechanism for the index contents (Section 3.3).

Contributions. The overall contribution is PostgreSQL-V, an integrated vector database system within PostgreSQL that, *for the first time, delivers performance on par with specialized vector databases.* Experiments show that PostgreSQL-V outperforms existing implementations, by up to **8.9**× better performance than pgvector [7] and up to **6.0**× better performance than PASE [26] in vector search. More importantly, PostgreSQL-V is fully compatible with pgvector and PASE (by implementing the same APIs in `IndexAmRoutine`), making it a viable drop-in replacement. Furthermore, the design principle of vector index decoupling developed in this paper is by no means restricted to PostgreSQL; it can be readily applied to enable fast vector search in other relational databases, e.g., MySQL (Section 5).

Open-Source. The source code of PostgreSQL-V is available at: https://github.com/purduedb/PostgreSQL-V.

2 MOTIVATION

Existing vector search implementations like pgvector [7] and PASE [26] follow the abstract index framework exposed in PostgreSQL to implement vector indexes.[3] However, this index framework requires indexes to follow a page-oriented structure to reuse PostgreSQL's heap-based storage engine. In this way, all pages (index pages and data pages) share the same storage engine and are accessed through the buffer manager. While this design enables vector indexes to inherit PostgreSQL's advantages, such as out-of-memory handling and transactional support, **it is ill-suited for vector indexing**.

First, mainstream vector indexes are memory-centric, e.g., IVF_FLAT, IVF_PQ, HNSW, and ScaNN. Converting them into page-based structures not only introduces complex engineering efforts (e.g., dealing with large vectors that span multiple pages [26]) but also incurs significant performance overhead. We conducted an experimental study in [28] to compare PASE [26] (an integrated vector database in PostgreSQL) and Faiss [3] (a specialized vector database) under the same conditions (e.g., the same index, same indexing parameters, all in memory). We found a huge performance gap in search, index construction, and index size. For example, when using HNSW, Faiss outperformed PASE by 2.2-7.3× in search time, 1.6-8.7× in index construction, and 2.9-13.3× in index size [28]. A key reason is that PASE (and PostgreSQL) is a disk-based system. Even if the entire dataset and index were stored in memory, the memory manager still needed to go through the buffer pool for page indirection, while Faiss natively stored vector indexes in memory and could directly locate the right tuple using a memory pointer without page-based indirection. More details can be found in [28].

Second, the existing design is also not suitable for disk-based vector indexes. One might think that an advantage of pgvector (or PASE) is its ability to support larger-than-memory vector indexes by reusing PostgreSQL's buffer manager. However, its performance is significantly slower than that of a native disk-based vector index, e.g., DiskANN [21].

Moreover, we argue that even if pgvector (or PASE) supports DiskANN in the future, its performance may still lag behind the native DiskANN library. This is because pgvector still accesses the in-memory component of the DiskANN index (i.e., quantized vectors) through PostgreSQL's buffer manager, whereas the native DiskANN library can access it directly through memory pointers. Also, the buffer manager may store non-vector data pages, which compete for buffer cache space with DiskANN. Section 4.3 shows more evaluation.

Third, the existing design is not suitable for vector index updates. pgvector (or PASE) stores vector indexes using PostgreSQL's heap-based storage engine, which performs in-place updates. However, vector indexes (e.g., HNSW and IVF_FLAT) are known to be inefficient with in-place updates, especially under concurrent access. Thus, many vector databases have adopted an LSM-based design. Examples include Milvus [24], Pinecone [15], SingleStore-V [13], AnalyticDB-V [25], BlendHouse [16], and Elasticsearch [14]. Section 3.2 shows more details.

3 SYSTEM DESIGN

In this section, we present the system design of PostgreSQL-V (Figure 1). Similar to pgvector and PASE, PostgreSQL-V stores vector data as a table column and builds a vector index (e.g., HNSW and IVF_FLAT) for that column. Then, the vector search is represented as an SQL query [26], which is further compiled into an optimized

[3]We exclude Google's AlloyDB (closed-source) though it supports vector search with ScaNN, as no details have been disclosed on *how ScaNN is integrated into AlloyDB.*

query plan that calls the pre-built vector index to find similar vectors. However, *the key innovation in PostgreSQL-V is decoupling vector indexes* as described next.

3.1 Decoupling the Vector Index

We propose a *new design principle that decouples vector indexes from PostgreSQL's core engine*, eliminating the need for a page-oriented structure that would otherwise impose significant performance overhead. In our design, vector indexes are implemented as an independent component that bypasses PostgreSQL's architectural constraints. This component supports the `IndexAmRoutine` interface [26] to remain compatible with PostgreSQL. The interface defines a set of functions for index updates, searches, and other operations, which are invoked by PostgreSQL's query engine. Figure 1 shows the overall system architecture of PostgreSQL-V, which consists of two parts: PostgreSQL's core engine and the decoupled vector index. PostgreSQL-V ensures transactional consistency via a new lightweight mechanism (described in Section 3.3).

Benefits of Decoupling. The overall benefit of the decoupled design is *flexibility*, which translates into high performance and cost-efficiency. **(1) High performance:** As vector indexes are decoupled, they can be implemented using any native vector index library (e.g., Faiss or hnswlib) or adopt any optimization developed in specialized vector databases, enabling PostgreSQL-V to achieve performance comparable to specialized vector databases. **(2) Pluggable index:** As the area of vector databases evolves rapidly with new vector indexes emerging, the decoupled design allows us to effortlessly plug new vector indexes and optimizations in PostgreSQL-V, ensuring it remains at the forefront of the field. **(3) Customized optimizations:** By decoupling vector indexes, PostgreSQL-V enables optimizations that were previously not possible in pgvector, such as using an LSM-based framework for efficient index updates (Section 3.2) or developing a fast crash recovery mechanism for vector indexes (Section 3.3). **(4) Optimized buffer manager:** Decoupling is also beneficial for disk-based vector indexes as it allows explicit control over the information retained in the buffer pool. For example, DiskANN requires all compressed PQ codes to reside in memory; however, if implemented using the pgvector approach, this information could be evicted by other non-vector pages since they share the same buffer pool. **(5) Leveraging modern hardware:** Once vector indexes are decoupled, they can be optimized for new hardware such as tiered storage, CXL memory, GPUs, and FPGAs, which would be difficult using the pgvector approach.

Challenges of Decoupling. The key challenge of decoupling vector indexes is index consistency. This is because updating the main table (e.g., inserting vectors) – handled by PostgreSQL's core engine – and updating the vector indexes – handled by the independent vector index component – are not atomic. Thus, if a crash occurs between the two operations, there will be index inconsistency. Section 3.3 presents a new approach to address this issue. Note that this is not an issue in pgvector [7] and PASE [26], as they follow PostgreSQL's page-based structure and can rely on PostgreSQL's transaction mechanism to guarantee consistency.

3.2 LSM-based Vector Index Framework

Decoupling vector indexes allows us to directly call native vector index libraries (e.g., Faiss) for fast vector search. However, these libraries often lack efficient support for vector index updates and concurrent read/write access. Thus, many specialized vector databases, such as Milvus [24] and Pinecone [15], as well as integrated vector databases like SingleStore-V [13] and BlendHouse [13], and recent research [29], wrap native vector index libraries within an LSM-based framework. PostgreSQL-V follows this design. This also shows the flexibility of the decoupled approach, enabling it to seamlessly incorporate best practices from specialized vector databases.

Specifically, the LSM-based framework consists of in-memory *mutable* segments (aka memtables) and on-disk *immutable* segments. Newly inserted vector data is first written to a memtable for real-time updates. When full, it is marked as immutable and flushed to disk. A vector index is then built asynchronously (via a native vector index library) for the immutable segments. Memtables may or may not be indexed. For example, SingleStore-V performs a full scan on memtables due to their small size [13]. A background process periodically merges smaller (immutable) segments into larger ones, during which their vector indexes are merged. Deletions are handled using a per-segment bitmap to mark soft deletes.

In PostgreSQL-V, we follow the above design. Specifically, we build vector indexes only for on-disk immutable segments and do not index vectors in memtables (following [13]), where each memtable contains just an array of vectors. To facilitate crash recovery (Section 3.3.3), we augment each vector with its VID (vector ID) and TID (tuple ID in the original relational table). Moreover, for every segment flushed to disk, we maintain a *flushed_VID* that indicates all vectors with VIDs up to that point have been flushed.

The LSM-based framework does not necessarily degrade search performance, as a background process continuously merges smaller segments into larger ones. Depending on the update rate, the number of segments is manageable. Moreover, the background process (e.g., index merging or rebuilding) in the LSM-based framework does not necessarily affect the foreground search, because there are many ways to minimize or even bypass the overhead, such as offloading these resource-intensive tasks to other nodes using the idea of compaction as a service [27]. Instead, the LSM-based design enables many benefits, e.g., fast updates (as new vector data is inserted into the memtable), efficient concurrent access (due to immutability), and is also friendly to crash recovery (due to immutability).

3.3 Index Consistency Guarantee

As mentioned in the challenges in Section 3.1, decoupling leads to vector index inconsistency. A straightforward solution is to allow the vector index to generate its own WALs and flush them before applying any changes. However, this approach does not work because our decoupled vector index must implement PostgreSQL's `IndexAmRoutine` interface to be recognized by PostgreSQL's query engine. But this interface *does not expose any transaction-related information*, such as transaction IDs and commit status, meaning the decoupled vector index's logging system is unaware of PostgreSQL's transaction status and cannot guarantee consistency.

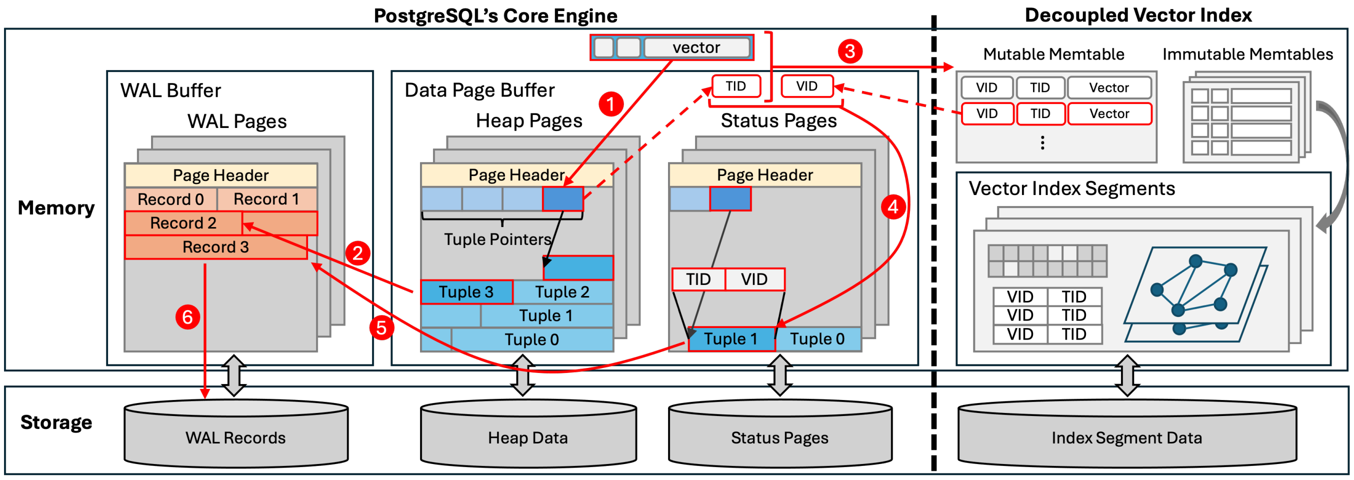

Figure 1: System Architecture of PostgreSQL-V

This introduces a design challenge: *ensuring index consistency requires integration with PostgreSQL's architecture, while high-performance vector search demands architectural decoupling.*

3.3.1 Lightweight Index Consistency Mechanism

To address the challenge, we propose a lightweight mechanism to guarantee crash consistency by observing two new opportunities: (1) PostgreSQL already provides a full-fledged crash recovery mechanism, and (2) the vector index is built on an LSM-based framework.

Specifically, we separate the vector index into two components: (1) the *metadata component*, which tracks the existence of index entries in the vector index. An entry is considered to "exist" if a vector is inserted and remains so until it is deleted. (2) the *functional component*, which performs vector search operations. Our main idea is that the metadata relies on PostgreSQL's transaction mechanism to maintain crash consistency with the heap table, while the functional component uses a customized mechanism (Section 3.3.3) to guarantee crash consistency.

The rationale is that, the metadata serves as the single source of truth, correcting any inconsistencies in the decoupled functional component. Since the entire vector search is performed within the decoupled component, PostgreSQL's architectural constraints do not impact the performance of the vector index.

The metadata component is implemented with *status pages*, which record <VID, TID> pairs for each entry in the decoupled index. VID (vector ID) is assigned monotonically by the decoupled index during each insertion, and TID (tuple ID) is the vector's position in the relational table. Status pages are integrated into PostgreSQL's architecture via slotted pages. Thus, their crash consistency is guaranteed by PostgreSQL's WAL mechanism. **In this way, the status pages (i.e., metadata) serve as a bridge between PostgreSQL's core engine and the decoupled vector index**.

3.3.2 Normal Processing

Next, we explain how the status pages and the decoupled vector index operate during normal transaction processing. As mentioned before, our decoupled vector index implements PostgreSQL's `IndexAmRoutine` interface. Functions defined in this interface are invoked by PostgreSQL's core engine. The `IndexAmRoutine` interface includes two types of index update operations:

Insertion. Figure 1 illustrates the entire process of inserting a vector within a transaction. Suppose the inserted tuple contains a vector and two other fields. The insertion process proceeds as follows: ❶ The tuple is first inserted into a heap page – using the slotted page layout – that resides in the data page buffer. ❷ A corresponding WAL record for this heap page is then generated and appended to the WAL buffer. ❸ PostgreSQL-V then invokes the vector index insertion. The vector from the inserted tuple, along with the TID – composed of the page number and tuple offset – is passed to the decoupled vector index. A VID is generated for the newly inserted vector. As the LSM-based framework is adopted, the vector data, TID, and VID are appended to the memtable. ❹ The <TID, VID> pair is written to a status page, which is cached in PostgreSQL's data page buffer. ❺ A corresponding WAL record for the status page is appended to the WAL buffer. ❻ Upon transaction commit, all WALs (of the heap and status pages) are flushed to disk, at which point all modifications are persisted.

Vacuum. The `IndexAmRoutine` interface does not directly support deletions and instead provides a vacuum operation, as PostgreSQL handles deletions lazily. Query correctness is ensured by performing visibility checks on result tuples after index scans. Periodically, PostgreSQL runs a vacuum process that physically removes dead tuples and invalidates their TIDs. To avoid dangling index entries, PostgreSQL first triggers an index vacuum to remove all index entries corresponding to these dead tuples before the tuples are removed from the heap table. During this process, if a vector resides in the memtable, it is removed directly. If it is located in a flushed segment, it is soft-deleted by marking it in the segment's bitmap. The status page then removes the associated <TID, VID> pair and appends a WAL record to log the deletion.

Benefits. This design eliminates the need for generating WAL records or triggering forced flushes in the decoupled vector index

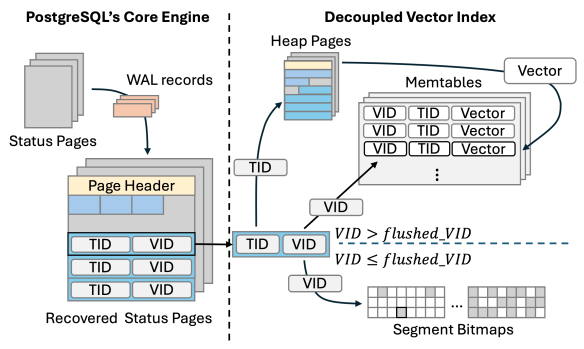

Figure 2: Recovery Process

during insertion and vacuum operations, as crash consistency is fully ensured by the WALs of the status pages (see Section 3.3.3). Thus, it introduces minimal overhead during normal transaction processing. Another important advantage is that this design is agnostic to the underlying vector indexes (whether HNSW or IVF_FLAT) because of the LSM-based index framework.

3.3.3 Vector Index Recovery

The recovery process is as follows (Figure 2). We first restore the status pages via PostgreSQL's recovery mechanism. Once these pages are consistent with the heap pages, they serve as the source of truth for determining which index entries should exist in the decoupled vector index. We then recover the LSM-related information, i.e., unflushed memtables and the bitmaps of flushed segments.

Step 1: Recovering Status Pages. Since status pages are integrated into PostgreSQL's architecture, they can be recovered by PostgreSQL's recovery mechanism, which sequentially replays their associated WAL records to ensure consistency with the corresponding heap pages (see the left side of Figure 2).

Step 2: Reconstructing Unflushed Memtables and Flushed Segments' Bitmaps. As we use the LSM-based framework, it is also friendly to crash recovery since flushed segments – the majority of the data – are immutable and do not require recovery. We only need to recover memtables and the bitmaps of flushed segments since they are in memory and may not be flushed before a crash. Recovering memtables is necessary as they may contain vectors from committed transactions. Recovering bitmaps is necessary as (1) the vector index is unaware of transaction states, so uncommitted vectors may be flushed to disk, and a crash before commit can leave dangling entries, requiring bitmaps to invalidate them; and (2) vector indexes generate no WALs, and modified bitmaps are not force-flushed during vacuum, so deletions may be lost after a crash.

Memtables and bitmaps are recovered in one pass (Figure 2). At the beginning of vector index recovery, an empty memtable and empty bitmaps for flushed segments are initialized. All bits in the bitmaps are set to 0. The recovery process proceeds by sequentially scanning the status pages and comparing the VID in each <TID, VID> pair with *flushed_VID*. The *flushed_VID*, maintained by the LSM structure, indicates that all vectors with $VID \leq flushed\_VID$ have been flushed to disk. If $VID \leq flushed\_VID$, the corresponding vector resides in a flushed segment, and its bit in the segment bitmap is set to 1. If $VID > flushed\_VID$, indicating that the vector was not persisted in the vector index before the crash, the vector must be retrieved from the heap table using its TID. The retrieved vector, along with its VID and TID, is then inserted into the memtable.

Recovery Correctness Analysis. The rationale behind our consistency mechanism is straightforward. Status pages can be regarded as a non-functional vector index whose consistency with the heap pages (table's vector data) is ensured in the same way as index pages in pgvector or PASE. This is because the status pages are fully integrated into PostgreSQL's architecture and thus can leverage its transactional mechanism. Thus, the status pages are guaranteed to be consistent with the restored heap pages after recovery. We then use the status pages to recover the decoupled vector index. This process ensures that the decoupled vector index is consistent with the status pages and, therefore, consistent with the heap pages.

3.4 Query Processing

The decoupling of the vector indexes does not affect query processing, as the implementation follows `IndexAmRoutine` – the same interface used by pgvector. Like pgvector, upon receiving an SQL query (with vector search), PostgreSQL-V first parses it and generates an execution plan. When it processes a vector index scan, it invokes the scan function in the `IndexAmRoutine`, which then calls the decoupled index. For filtered vector search, PostgreSQL-V uses the same approach as pgvector, which operates independently of the vector index scan. Therefore, our decoupled design does not compromise the vector search capabilities supported by pgvector.

4 PRELIMINARY EXPERIMENTS

4.1 Experimental Setup

We conduct all experiments on a Linux server with a 112-core Intel Xeon CPU (2.0 GHz), 251 GB of DRAM, and a 1.8 TB SSD.

Competitors. As PostgreSQL-V supports both memory-based indexes (currently HNSW and IVF_FLAT) and disk-based indexes (currently DiskANN), we evaluate it against multiple baselines.

For memory-based experiments, we compare it with **pgvector** [7] and **PASE** [26], two popular implementations of vector search in PostgreSQL. To show that PostgreSQL-V achieves performance comparable to specialized vector databases, we compare it with **Faiss** because PostgreSQL-V can implement its decoupled indexes using Faiss. **This ensures a fair comparison under the same conditions** (e.g., the same vector index and indexing parameters). Note that PostgreSQL-V is highly flexible and can use any vector index implementation (e.g., hnswlib, ScaNN, and Milvus). To show the flexibility, we also implement the HNSW index in PostgreSQL-V using the **hnswlib** [4], a native HNSW index library.

For disk-based experiments, PostgreSQL-V implements the decoupled vector index using the native DiskANN library [2]. As pgvector does not support any disk-based vector indexes (e.g., DiskANN), we compare it against **pgvectorscale** [8], which implements a DiskANN variant in PostgreSQL. We also compare PostgreSQL-V with the **native DiskANN library** [2]. We are aware that Microsoft is developing `pg_diskann` [10], which implements vector search inside PostgreSQL, but it is not open-sourced at the time of paper writing.

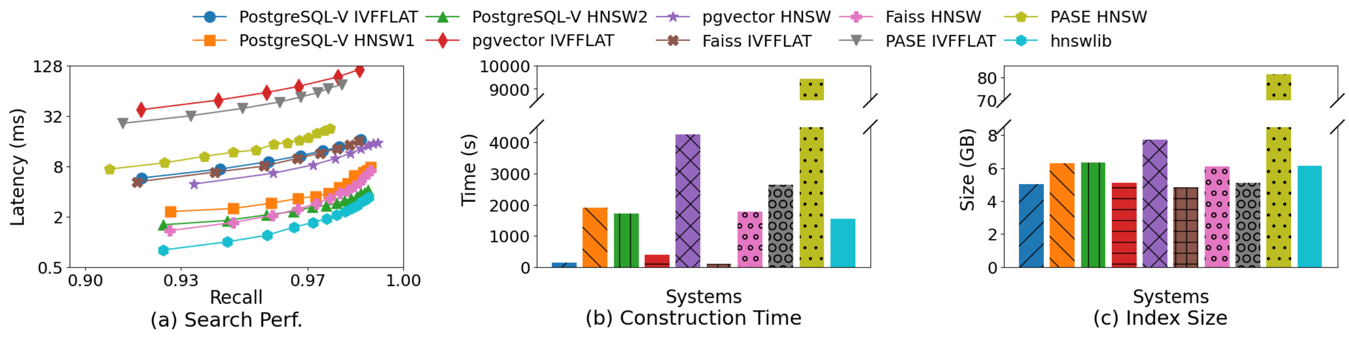

Figure 3: Memory-based Vector Indexes on SIFT10M

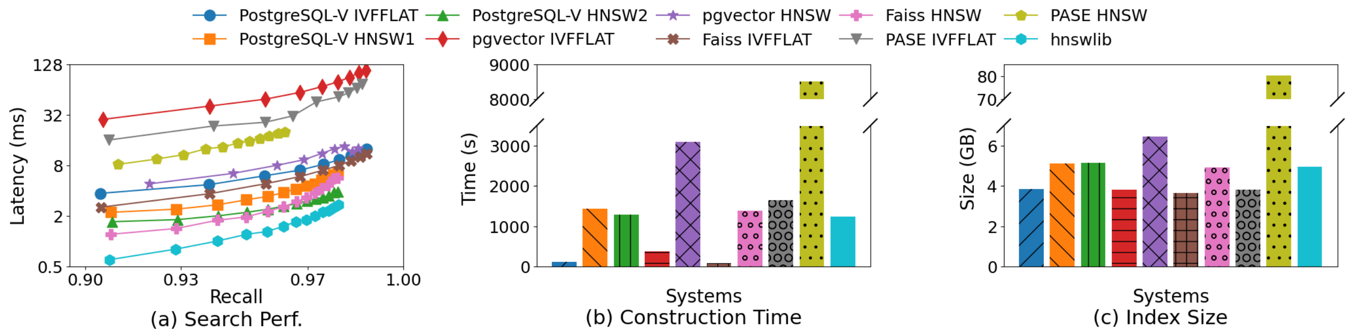

Figure 4: Memory-based Vector Indexes on Deep10M

4.2 Evaluating Memory-based Vector Indexes

To ensure a fair comparison, we configure the buffer size of pgvector and PASE to be large enough to cache the entire index. Figures 3 and 4 show the results on two standard datasets, SIFT10M and DEEP10M, where each dataset contains 10 million vectors and 10,000 queries.

As shown in Figures 3a and 4a, for vector search performance, PostgreSQL-V is **6.5×-8.9× faster** when using IVF_FLAT and **2.9×-3.5× faster** when using HNSW compared to pgvector. Note that PostgreSQL-V HNSW1 and HNSW2 refer to the HNSW implementations from Faiss and hnswlib.

Moreover, PostgreSQL-V is much faster in index construction time and takes less space than pgvector. PostgreSQL-V also significantly outperforms PASE in all aspects. The improvement is due to PostgreSQL-V's new design of vector index decoupling, which allows it to directly call the native vector index library and bypass PostgreSQL's architectural overhead.

PostgreSQL-V achieves performance on par with Faiss and hnswlib when using their implementations, showing the flexibility and high performance of our decoupled design. We expect PostgreSQL-V to match the performance of the native ScaNN library or Milvus if its indexes were implemented using ScaNN or Milvus. The minor gap is expected as PostgreSQL-V is a full database (not a library) and incurs the constant overhead (around 0.8ms) to parse SQL and retrieve rows from the heap table based on tuple IDs returned by the index library.

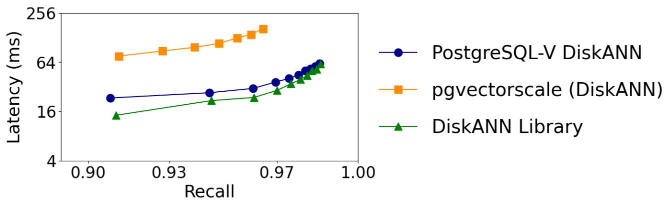

Figure 5: Disk-based Vector Indexes on Cohere10M

4.3 Evaluating Disk-based Vector Indexes

Following pgvectorscale's recommendation, we use the Cohere10M dataset [1], a higher-dimensional dataset with 768-dimensional vectors. We set the buffer size to 3GB for all systems (the total index size is 39GB). Figure 5 shows that PostgreSQL-V (DiskANN) outperforms pgvectorscale by **3.2×-4.0×** in search performance. The improvement is because of the better control of buffer memory enabled by decoupling (Section 2). Also, it nearly matches the performance of the native DiskANN library. This result is consistent with our observations from the memory-based evaluation.

4.4 Evaluating Crash Recovery

To validate the correctness of our recovery mechanism, we develop a test that continuously inserts and deletes vectors from a table. We simulate crashes by randomly killing PostgreSQL's process. After each recovery, we check consistency between the heap table and vector index to ensure no missing, dangling, or mismatched entries. This test is repeated over 1,000 times with no inconsistencies observed. The recovery overhead depends on the number of entries in the vector index and the number of unflushed vectors in the memtable. We evaluated a scenario with 10 million vectors (SIFT10M) in flushed segments and 100,000 unflushed vectors in the memtable. The recovery completed in 239 ms. Roughly 110 ms was spent scanning status pages and reconstructing the bitmaps. This step is lightweight as status pages *only store VIDs and TIDs* (*not the actual vector data*). The remaining 130 ms was spent fetching vector data from heap pages to reconstruct the memtable. This overhead can be reduced through future optimizations, such as smaller memtables, multiple background writers to speed up flushing, or exploring new LSM-aware recovery mechanisms.

5 LESSONS AND FUTURE WORK

In this work, we build a prototype, PostgreSQL-V, to support fast vector search within PostgreSQL and achieve performance comparable to specialized vector databases. The preliminary results are highly encouraging. We highlight two important lessons:

- **Lesson 1**: Decoupling vector indexes is crucial for significantly improving the performance of integrated vector databases.
- **Lesson 2**: To maintain index consistency after decoupling, we shall separate vector index metadata from its contents, leveraging the host database's transaction mechanism for metadata and a customized mechanism for the index contents (based on the metadata).

Future Work. This is a long-term project at Purdue. Looking ahead, while we will continue improving PostgreSQL-V (e.g., supporting ScaNN and Milvus) and conducting a comprehensive evaluation, we next highlight a few interesting directions inspired by this work.

Direction 1: Extension to the Cloud. Our decoupled design is highly suitable for cloud-native environments where resources are disaggregated. By deploying the vector index component as a service, it enables independent and elastic resource provisioning and scaling for vector indexes. Our design is advantageous for supporting vector search in cloud-native PostgreSQL, e.g., Amazon Aurora [19, 22] or Microsoft Socrates [11], whose vector search is not yet decoupled.

Direction 2: Extension to Other Relational Databases. We believe the lessons learned can be readily applied to enable efficient vector search in other relational databases. For instance, MySQL recently introduced vector search through MyVector [5], a UDF-based plugin. However, it does not guarantee transactional consistency between table data and the vector index, an issue that our proposed consistency mechanism can resolve. Looking ahead, we are highly interested in validating a bold conjecture that our techniques may be used to support fast vector search in **any relational database**, especially for those that are not based on LSM-based storage engines, as they can leverage Lesson 1 to achieve high performance while applying Lesson 2 to ensure transactional consistency.

Direction 3: Relational Databases for Grounding LLMs. This paper delivers a clear message: PostgreSQL (or relational databases in general) is well-suited to support efficient vector search. Given PostgreSQL's strong capability for handling complex queries that combine vector and non-vector data through powerful SQL, we argue that relational databases, rather than specialized vector databases, hold a unique position as RAG stores for grounding LLMs in the future. However, to support advanced RAGs, current relational databases must be reworked to efficiently store different types of data (e.g., graph, fulltext, image, video) and also return accurate results for arbitrary user queries (ideally, any query over any data), thereby evolving into true multi-model systems. We believe the core idea presented in this paper can inspire the design of such multi-model databases by *decoupling non-relational indexes from the relational engine while preserving transactional consistency.*

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under Grant Number 2337806 and by a gift funding from Google through the Google ML and Systems Junior Faculty Award.

REFERENCES

[1] [n. d.]. Cohere (https://huggingface.co/datasets/Cohere/wikipedia-22-12/).
[2] [n. d.]. DiskANN (https://github.com/microsoft/DiskANN).
[3] [n. d.]. Facebook Faiss (https://github.com/facebookresearch/faiss).
[4] [n. d.]. HNSWlib (https://github.com/nmslib/hnswlib).
[5] [n. d.]. MyVector (https://github.com/p3io/myvector-dev).
[6] [n. d.]. Oracle AI Vector Search (https://www.oracle.com/database/ai-vector-search/).
[7] [n. d.]. pgvector (https://github.com/pgvector/pgvector).
[8] [n. d.]. pgvectorscale (https://github.com/timescale/pgvectorscale).
[9] [n. d.]. Pinecone (https://www.pinecone.io/).
[10] 2025. Enable and Use DiskANN Extension (https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/how-to-use-pgdiskann).
[11] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD*. 1743–1756.
[12] Sebastian Bruch. 2024. *Foundations of Vector Retrieval*. Springer.
[13] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *PVLDB* 17, 12 (2024), 3772–3785.
[14] Adrien Grand. 2023. Vector Search in Elasticsearch: The Rationale Behind the Design (https://www.elastic.co/search-labs/blog/vector-search-elasticsearch-rationale).
[15] Amir Ingber and Edo Liberty. 2025. Accurate and Efficient Metadata Filtering in Pinecone's Serverless Vector Database. In *Proceedings of the 1st Workshop on Vector Databases at International Conference on Machine Learning*.
[16] Zhaojie Niu, Xinhui Tian, Xindong Peng, and Xing Chen. 2025. BlendHouse: A Cloud-Native Vector Database System in ByteHouse. In *ICDE*. 4332–4345.
[17] James Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *VLDB Journal (VLDBJ)* 33, 5 (2024), 1591–1615.
[18] James Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *SIGMOD*. 597–604.
[19] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. In *SIGMOD*. 180:1–180:26.
[20] Yannis Papakonstantinou, Alan Li, Ruiqi Guo, Sanjiv Kumar, and Phil Sun. 2024. ScaNN for AlloyDB (https://services.google.com/fh/files/misc/scann_for_alloydb_whitepaper.pdf).
[21] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS*. 13748–13758.
[22] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.
[23] Jianguo Wang, Shasank Chavan, Guoliang Li, Yannis Papakonstantinou, and Charles Xie. 2024. Vector Databases: What's Really New and What's Next? *PVLDB* 17, 12 (2024), 4505–4506.
[24] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua

Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. 2614–2627.

[25] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *PVLDB* 13, 12 (2020), 3152–3165.

[26] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *SIGMOD*. 2241–2253.

[27] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. In *SIGMOD*. 124:1–124:28.

[28] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. In *ICDE*. 2835–2849.

[29] Shurui Zhong, Dingheng Mo, and Siqiang Luo. 2025. LSM-VEC: A Large-Scale Disk-Based System for Dynamic Vector Search. *CoRR* abs/2505.17152 (2025).