Nova: A Multi-Purpose Vector Engine for Low-Latency, Multi-Tenant, and Cross-Table Hybrid Retrieval

Dechuang Chen*
The Chinese University of Hong Kong
Hong Kong, China
dcchen@se.cuhk.edu.hk

Bing Chen*
Alibaba Cloud Computing
Hangzhou, China
huanjin.cb@alibaba-inc.com

Jianwei Lu
Zhiwei Wu
Alibaba Cloud Computing
Hangzhou, China
{lujianwei.ljw,linzhi.wzw}@alibaba-inc.com

Zeyu Yang
Qiang Gu
Alibaba Cloud Computing
Hangzhou, China
{chengyue.yzy,guqiang.gu}@alibaba-inc.com

Fangyuan Zhang
Harbin Institute of Technology
Shenzhen, China
zhangfangyuan@hit.edu.cn

Sibo Wang
The Chinese University of Hong Kong
Hong Kong, China
swang@se.cuhk.edu.hk

Zeyuan Yu
Caihua Yin
Alibaba Cloud Computing
Hangzhou, China
{zeyuan.yzy,caihua.ych}@alibaba-inc.com

Wenchao Zhou
Feifei Li
Alibaba Cloud Computing
Hangzhou, China
{zwc231487,lifeifei}@alibaba-inc.com

ABSTRACT

Vector retrieval underpins AI applications like recommendation, image search, and RAG. While similarity search is mature, production deployments struggle with high-concurrency ingestion, dense multi-tenancy, and complex cross-table hybrid queries bridging structured and unstructured data. Existing systems often fail to reconcile these demands, leading to resource contention and data staleness.

We propose *Nova*, a cloud-native, multi-purpose vector engine integrated into Alibaba Cloud's AnalyticDB-PG (ADB-PG). Nova introduces a *dual-layer delta-base architecture* that decouples real-time ingestion from indexing, ensuring low-latency ingestion, immediate data visibility, and lock-free queries even under bursty workloads. To support diverse storage hierarchies, Nova implements *Nova Memory* for ultra-low latency and *Nova Disk* for cost-effective, large-scale storage. Crucially, Nova treats vector search as a first-class citizen within the relational optimizer, enabling efficient cross-table hybrid retrieval via partial/covering indexes and cost-based adaptive joins. We further enhance multi-tenant isolation and stability through lightweight concurrent indexing and anytime query execution.

Evaluations on standard benchmarks and production workloads show Nova significantly outperforms Milvus and pgvector, achieving lower latency, higher throughput, and stronger read/write stability.

PVLDB Reference Format:
Dechuang Chen, Bing Chen, Jianwei Lu, Zhiwei Wu, Zeyu Yang, Qiang Gu, Fangyuan Zhang, Sibo Wang, Zeyuan Yu, Caihua Yin, Wenchao Zhou, and Feifei Li. Nova: A Multi-Purpose Vector Engine for Low-Latency, Multi-Tenant, and Cross-Table Hybrid Retrieval. PVLDB, 19(12): 4453 - 4465, 2026.
doi:10.14778/3827998.3828045

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828045

1 INTRODUCTION

As artificial intelligence is increasingly adopted across applications, vector retrieval has become a key technological pillar for modern applications such as face recognition [32], recommendation systems [24], and image retrieval [19]. This trend has driven advances in vector similarity search [13, 15, 16, 20, 21, 27] and specialized systems such as Milvus [37], pgvector [4], Elasticsearch [3], and ADB-V [38], which aim to provide low-latency, high-throughput, and highly available vector retrieval and management. Recently, Retrieval-Augmented Generation (RAG) [14] has further reinforced the importance of vector retrieval. By incorporating external knowledge bases, RAG enables large language models to acquire domain-specific knowledge and integrate real-time information, mitigating hallucination. Its implementation heavily relies on efficient vector retrieval mechanisms [10, 23, 26]. As application scenarios continue to expand, vector retrieval workloads and requirements become increasingly diverse, bringing the following challenges.

Low Latency and Robustness under Resource Contention. Increased latency in queries and data ingestion directly affects user experience and may even cause downstream service timeouts. Meanwhile, vector workloads often face bursty writes and load fluctuations. In a front-end RAG setting, for example, a user upload can trigger batch ingestion: a large document may be split into tens of thousands of chunks, embedded, and inserted in a short period. Background processes such as `vacuum` periodically contend for computational and I/O resources. A database must effectively handle such disturbances to sustain stable query and write operations.

Multi-Tenancy with High Density. Consider the Alibaba Cloud Model Studio [2] platform, which offers RAG services that allow users to upload documents and build private knowledge bases. In this context, maintaining a dedicated database instance per user is impractical in terms of resource utilization and operational cost. Hence, a multi-tenant instance architecture is commonly adopted, where multiple users share the same database instance to manage their knowledge bases. This shared model demands lightweight storage structures and computational costs for vector indexing, as well as the ability to dynamically schedule resources across workloads.

Cross-Table Hybrid Query. While single-table hybrid queries, i.e., vector search with scalar filters, have been widely studied [22, 28, 37, 38, 40, 43], structured and unstructured data often reside in separate tables in practice. In Alibaba Cloud Model Studio's RAG scenario, documents are split into text chunks with vectors stored in a dedicated vector table, while metadata is stored separately. Queries may need to retrieve relevant vector chunks while filtering or joining on metadata. Merging them into one wide table would cause redundant metadata storage and costly updates. This is especially pronounced under MVCC, where metadata updates cause a large number of vectors to be copied and re-inserted into the vector index, causing severe write amplification. Therefore, database systems need efficient support for cross-table joins together with hybrid retrieval.

One Size Does Not Fit All. Different applications impose varying requirements on database capabilities. In biochemical applications, for example, enterprises may represent molecular structures and chemical reactions as vectors and build vector databases with tens of billions of records. Such systems are often for internal use, with low concurrency but high cost sensitivity. In contrast, social-network applications use vectors to characterize user behavior and support real-time matching for new social connections, demanding extremely low query latency and high system stability. No single solution can accommodate all these scenarios today; database systems must offer multiple trade-offs and adapt to evolving requirements.

To address these challenges, we propose and implement **Nova**, a multi-purpose vector engine inside the OLAP system AnalyticDB-PG (ADB-PG) [42] at Alibaba Cloud. Nova prioritizes low latency and stability, maintaining stable read/write P99 latency under workload fluctuations while avoiding frequent lock contention and performance jitter. To this end, Nova generalizes a delta-base two-layer architecture to handle transient writes, ensuring low latency and real-time data ingestion. It further designs specialized storage and management mechanisms, together with a lightweight lock-free retrieval process, to sustain query performance under heavy writes. Nova incorporates various optimizations for hybrid queries, such as partial indexes and covering indexes, and extends the cost model to support cross-table hybrid retrieval. For multi-tenant scenarios, Nova introduces pre-warming and anytime queries to improve first-time query stability under cold starts. Leveraging its dual-mode architecture, Nova also redesigns lightweight concurrent index building to avoid "DDL storms" and cross-tenant interference during index construction. On this framework, Nova provides two index variants: **Nova Disk**, a hybrid index for large-scale vector management, and **Nova Memory**, a graph index for high-performance in-memory search, covering diverse business scenarios. Nova has been deployed on Alibaba Cloud, serving as a core infrastructure that delivers a scalable, versatile, low-latency, highly available, and robust vector retrieval and processing service for multiple platforms and services, including Alibaba Cloud Model Studio. It currently supports over 400,000 users and manages vector data exceeding 100 billion entries.

Our contributions are summarized as follows:

- We propose a dual-layer delta-base architecture that decouples real-time writes from base indexes for low-latency ingestion, immediate visibility, and lock-free queries. It further provides two-level WAL durability and multi-tenancy optimizations (Sec. 3).
- We design and implement two optimized indexes: Nova Memory, a graph-based index with duplicate merging and space reuse, and Nova Disk, a hybrid index with adaptive scanning and dynamic expansion for large-scale management (Sec. 4).
- We provide the first systematic treatment of cross-table hybrid queries, integrating partial/covering indexes and a cost-based adaptive join strategy into an extended optimizer to efficiently bridge vector scans and relational joins (Sec. 5).
- Experiments on public and private datasets (GIST, Cohere, Tongyi) show that Nova outperforms widely used systems such as Milvus and pgvector, achieving lower latency, higher throughput, and stronger read/write stability (Sec. 6).

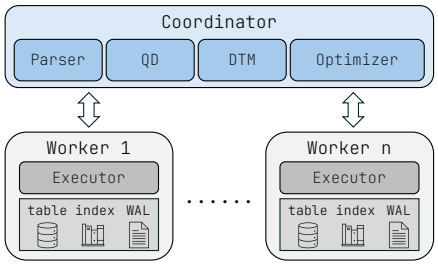


Figure 1: The overview of AnalyticDB-PG

2 BACKGROUND

AnalyticDB-PG. Nova is derived from ADB-V [38] and built on AnalyticDB-PG (ADB-PG for short) [42], a cloud-native real-time data warehouse [41, 42] based on the classic *Massively Parallel Processing (MPP)* architecture [9] to distributes data and computation across a cluster for high parallelism. As shown in Fig. 1, ADB-PG consists of one coordinator node for query orchestration and multiple worker nodes for independent data processing. The coordinator includes a Parser for client requests, an Optimizer for distributed execution plans, a Query Dispatcher (QD) for task scheduling and result aggregation, and a Distributed Transaction Manager (DTM) for ACID guarantees. Tables are typically row-partitioned and distributed across worker nodes. Each worker, similar to a standalone database instance, maintains local indexes and write-ahead logs (WAL) for local data to accelerate query processing and ensure high availability, while executing assigned query plans.

Alibaba Cloud Model Studio. Alibaba Cloud Model Studio is a comprehensive LLM operations (LLMOps) platform supporting the full lifecycle of AI applications, including model evaluation, fine-tuning, deployment, and inference. It enables users to build diverse applications on the Tongyi model family. As illustrated in Fig. 2, its RAG service allows users to construct proprietary knowledge bases to enhance model performance. Powered by the Nova engine, AnalyticDB-PG serves as the backend for knowledge base management. To date, it has helped over 400,000 users create more than 600,000 databases and manage a massive corpus of over 100 billion vectors. To optimize resource utilization, Model Studio operates under a **high-density multi-tenant architecture**, where a single instance serves 10,000 to 30,000 users with an average of 300 concurrent requests. This scale imposes stringent requirements on **low latency and robustness under resource contention**, necessitating strict isolation to prevent "noisy neighbor" effects. Furthermore, the platform must address the challenge that **one size does not fit all**: workloads vary significantly from latency-sensitive online services

to cost-sensitive large-scale storage, requiring the engine to provide flexible trade-offs between performance and cost. In Model Studio, document parsing and embedding generation run on a separate, horizontally scalable tier and asynchronously feed vectors to the database via a message queue. Because this tier is provisioned with compute capacity far exceeding that of the database, the sustained write bottleneck falls on the vector database, motivating the throughput-oriented index design that follows.

The platform's data model further complicates retrieval through cross-table hybrid queries. Each knowledge base is normalized into two tables: `t_embedding` for vectorized document chunks and `t_metadata` for attributes like filenames and tags, with the two tables linked by the `doc_id` field. Merging them into a single wide table is inefficient due to metadata storage redundancy, and storing multiple vectors per row likewise makes updates expensive. This is especially pronounced under MVCC: even a metadata-only change forces entire vector entries to be copied and re-inserted into the vector index, causing severe write amplification. Consequently, vector searches frequently require joining unstructured vector data with structured metadata based on custom user-defined filters. This places stringent demands on the engine to efficiently execute cross-table joins combined with vector similarity search, bridging the gap between relational operations and vector retrieval to ensure high-performance, context-aware RAG applications.

3 SYSTEM DESIGN

In this section, we introduce Nova's delta-base dual architecture. To maximize efficiency, Nova moves beyond conventional page-based storage and adopts a file-based manager with `mmap`, better matching access patterns of vector workloads. On this basis, Nova refines a lock-free retrieval workflow over the base index to provide low-latency and stable query performance under write-intensive workloads. On top of this framework, we implement two scenario-specific index variants, detailed in Sec. 4: **Nova Memory**, a graph index optimized for in-memory queries, and **Nova Disk**, a hybrid index for large-scale vector management. This section first presents Nova's core architecture and operational logic in Secs. 3.1-3.2, then shows the high-availability mechanisms in Sec. 3.3, and finally discusses framework-level optimizations for multi-tenancy in Sec. 3.4.

3.1 Nova Dual-layer Framework

Framework Architecture. Fig. 3 shows the delta-base dual architecture adopted by Nova, consisting of three primary components: delta files, which receive real-time write traffic and temporarily store incremental data; the base index, representing the underlying index structure (the specific form of which varies by index type); and the Write-Ahead Log (WAL), used for failure recovery to ensure high availability of the index. The base index is composed of a metadata file and several index-related structural files (e.g., posting lists in an IVF-based [44] index). Furthermore, a background process continuously monitors the state of the delta files. A `flush` operation is triggered when the number of accumulated vectors in a delta file exceeds a predefined threshold or when a specific time interval has elapsed, thereby integrating the incremental data into the base index.

Delta Files. Delta files are crucial to the low-latency read/write operations and system stability of Nova, serving multiple roles within the system. In each delta file, vectors and their corresponding context identifiers (`ctids`) are stored contiguously but separately, facilitating hardware acceleration through instruction-level parallelism (e.g., SIMD). First, data in each delta file is organized as a ring queue, making insertions lightweight and ensuring low write latency. This design also handles sudden bursts of write traffic, contributing to system stability. Through two-path recall (probing both delta files and the base index), incremental data becomes visible immediately after being written to a delta file, ensuring real-time data freshness. Second, delta files confine write-write conflicts—both between concurrent inserts and between `insert` and `vacuum` operations (Ref. to Sec. 3.2)—within themselves. This enables lock-free scans on the base index, as it only needs to manage read-write concurrency between `flush`/`vacuum` operations and scans.

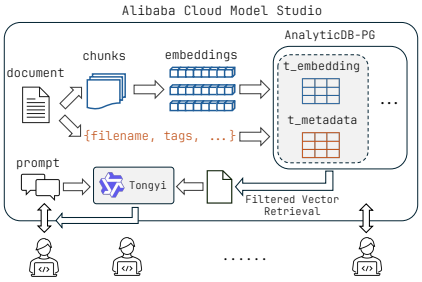


Figure 2: The overview of Alibaba Cloud Model Studio

Delta files rely on read-write locks for concurrency control. Given that their operations are lightweight, the duration for which write locks are held is relatively short. Additionally, Nova employs multiple delta files (three by default) to further enhance concurrency. While retrieving data from delta files introduces some performance overhead, this cost is negligible compared to the performance gains achieved through lock-free execution on the base index.

File-Based Storage. We observed that traditional page-based storage remains ill-suited for vector data. First, page-based layouts often suffer from *internal fragmentation* when page sizes do not align with vector dimensions. For example, pgvector uses 8KB pages, while a 1024-dimensional vector already requires over 4KB as 4-byte floats; thus, each page can store only one vector plus its `ctid` and metadata, resulting in substantial space wastage. Second, page-level read-write locks for concurrency control can degrade performance for index scans during operations like `vacuum`. To address these issues, Nova employs a file-based storage management scheme, where all data is written directly to files in its native in-memory layout. When an index is opened, these files are mapped directly into the memory address space via `mmap`, allowing data to be manipulated as if it were resident in memory. The `mmap` mechanism automatically caches hot data in memory and evicts cold data when memory is scarce, a process that is entirely transparent to the upper-layer implementation. Nova's file-based storage assumes a POSIX-compatible file system and is currently deployed over local NVMe disks and Alibaba Cloud Enterprise SSD volumes. Extending this substrate to object stores such as Alibaba Cloud OSS is left as future work.

Quantization Support. Nova supports many quantization schemes, such as PQ [21] and RabitQ [16]. When quantization is enabled, the base index stores only quantized vectors, while the delta file retains the original vectors. Nova also maintains separate auxiliary files,

such as PQ codebooks and RabitQ rotation matrices, to accelerate distance computation over quantized vectors.

Lazy Index Construction. Some vector indexes require substantial training data; for instance, cluster-based methods require a sufficient number of vectors for reliable clustering. To simplify integration, Nova allows pre-creating such indexes on empty tables, persisting construction parameters in metadata. The base index remains empty and the delta-file is deferred from indexing until a sampling threshold is met. When the threshold is reached, index construction is triggered and the buffered data is ingested into the base index.

3.2 Operation

Scan. The `scan` operation retrieves the top-k approximate nearest neighbors for a query vector, supporting optional filtering via bitmaps or predicates. It performs dual-path recall by querying both the delta file and the base index, merging and deduplicating the results. During execution, the scanner acquires a read lock on the delta file while the base index is accessed lock-free. A user-configurable recall amplification factor can be applied to further boost the recall.

Insert. The `insert` operation appends a new vector to the system. An insertion transaction first writes to the WAL; it then acquires a write lock to append the vector and its `ctid` to an available delta file. It can commit once both writes are complete.

Flush. The `flush` operation synchronizes entries from the delta file into the base index. The executor first acquires a read lock on the delta file to transfer entries safely, then upgrades to a write lock only to truncate the file. Base-index insertions are batched for higher throughput and may optionally be lock-free. The write lock is held only for the brief truncation step. During the read-locked transfer phase, concurrent inserts can route to other delta files, since Nova maintains multiple delta files. Scans acquire only read locks, so they proceed during transfer and, at most, wait for the short truncation window on the delta file being flushed.

Vacuum. The `vacuum` operation removes logically deleted vectors. If a base index exists, Nova first flushes the delta file, followed by a vacuum of the base index (which may also be performed lock-free). If the base index has not been built, Nova acquires a write lock on the delta file to filter dead tuples and compact the remaining tuples in place. It ensures that `flush` and `vacuum` never run concurrently, so at most one writer mutates the base index at any time.

3.3 High Availability

Nova ensures high availability through Write-Ahead Logging (WAL) and checkpointing. In general, WAL records must be persisted before the corresponding data is written to disk; otherwise, crash recovery may observe data inconsistencies. Yet, enforcing this order under Nova's `mmap`-based storage is challenging because dirty-page flushes are kernel-controlled and are inherently non-deterministic. Direct writes to `mmap`-allocated memory may persist data before its WAL record, while forcing a WAL flush before every write would create many small I/Os and severely degrade performance.

Logically, Nova's WAL records are divided into a *meta* class (e.g., records for the metadata file in Fig. 3) and a *data* class (e.g., records for the structure file(s) in Fig. 3), each with its own flush policy over the shared PostgreSQL WAL stream. The meta records are small and determine data visibility via boundaries such as maximum offsets or slots. Data beyond boundaries is ignored during scans and reclaimed by vacuum. Nova always flushes data WAL before meta WAL, and a transaction becomes durable only after the meta WAL is persisted.

Given the strong consistency requirements of metadata, we enforce strict ordering control over its persistence. To ensure that the meta WAL is always written first, Nova copies the contents of the metadata file into a separate memory position when opening the index. All subsequent writes and vacuum operations are performed on this in-memory copy, rather than the mmapped file. At transaction commit time, the meta WAL is flushed first, after which the metadata changes are written back to the metadata file. In contrast, data modifications are applied directly to the mmapped file and may be persisted asynchronously or out of order by the kernel. This design confines costly synchronous I/O to the typically small metadata footprint, avoiding the penalty of synchronous writes on large volumes of data, thereby achieving a favorable tradeoff between consistency and high-volume data performance.

For checkpointing, which requires that in-memory state be synchronized with disk, Nova flushes dirty pages from the page cache by invoking `fsync` on the index files.

Implementation. Nova logs all index-state mutations on PostgreSQL's WAL stream using a custom resource manager (`RM_NOVA_ID`) via the Resource Manager extension interface; each mutation is emitted as a standard WAL record via `XLogInsert`. A small set of record types covers directory and meta-page creation, file extension, batch base-index build, byte-level updates (single- and multi-region), and delta-base swap, encoding every mutation in an idempotent, replay-safe form (file path, offset, payload). Meta-class and data-class records share the same WAL stream, transaction boundaries, and flush-on-commit guarantee. The meta/data split only determines when a record's data may be written to its mmap-backed page; both classes remain in the same log. During crash recovery, PostgreSQL's startup process invokes Nova's redo handler for `RM_NOVA_ID` records, replaying them via `seek` and `write` on the index files. The entire recovery path runs inside PostgreSQL's standard recovery loop, avoiding any cross-system flush coordination.

Redo Optimization. During crash recovery or mirror synchronization, Nova performs a redo operation by replaying the logs from the most recent checkpoint. To minimize the impact on concurrent workloads, each WAL entry includes a type field. Large logs are written using direct I/O to bypass the page cache and avoid evicting other workloads' cached data, while smaller logs continue to use buffered I/O to coalesce I/O and improve redo efficiency.

3.4 Optimization

Concurrent Index Construction. To meet the high concurrency DDL demands of multi-tenant environments, we introduce a lightweight concurrent index construction protocol tailored for Nova. Unlike the standard `CREATE INDEX CONCURRENTLY` found in PostgreSQL, which typically involves four phases, two full table scans, and multiple long-running transaction waits that often lead to lock contention and "DDL storms", our approach streamlines the process into three distinct phases: initialization, base index construction, and activation. This efficiency is achieved by offloading write operations to delta files during index creation.

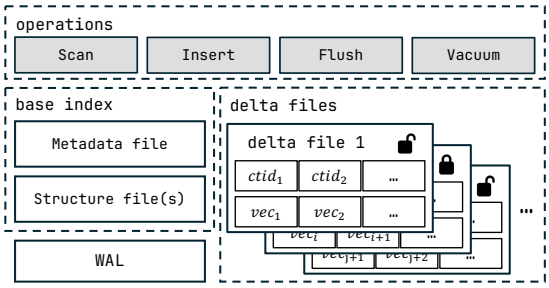


Figure 3: Nova Index Framework Overview

During initialization, a short transaction updates the system catalogs to mark the index as "writable", allowing subsequent DML operations to proceed by redirecting writes to delta files without blocking. After all pre-existing transactions complete, base index construction starts a new transaction to scan the table and build the main index without blocking concurrent writes. Finally, a brief activation transaction marks the index as "readable", making it available to queries. This protocol requires only a single full table scan and significantly reduces lock contention. Although it may introduce minor data duplication—resolved safely at query time—the resulting throughput gains for online DDL operations offer a critical advantage in high-concurrency environments.

A similar approach can be used to implement `REINDEX INDEX CONCURRENTLY` (RIC) on Nova indexes, offering a path to handle centroid drift in NovaD. As data drifts from the original IVF centroids, recall on the disk-resident base index degrades. The proper remedy is to re-cluster and rebuild. Nova exposes drift indicators such as per-cluster size skew and observed recall regression, which `autovacuum` or an upstream workload manager can sample. RIC is fired when a configured threshold is crossed, and foreground reads and writes remain non-blocking throughout.

Workload-Aware Index Pre-Warming. To eliminate cold-start latencies caused by disk I/O, we propose a proactive framework that pre-warms indexes by loading them into memory ahead of anticipated access. At the core of this framework lies a priority model that integrates automated prediction with manual overrides. The priority score P_i for a given index i is defined as:

$$P_i = \alpha \cdot S_{\text{pred}} + \beta \cdot S_{\text{man}} - \gamma \cdot S_{\text{cost}}$$

where each term represents a normalized score. The prediction score S_{pred} is derived from the confidence of an ARIMA time-series model, which analyzes an index's historical queries per second (QPS) to forecast future access peaks. The manual score S_{man} reflects manually submitted requests and is weighted by the historical accuracy, or reputation, of the request source to ensure its reliability. The cost score S_{cost} introduces a penalty proportional to the index's on-disk size, moderating the priority of large indexes to promote fair resource allocation. The coefficients α, β, and γ are tunable hyperparameters used to balance the influence of predictive trends, manual input, and resource cost, respectively. Based on this priority queue, the scheduler loads the memory-mapped regions of the Nova index into physical memory, subject to system resource constraints. This data-driven, cost-aware approach effectively transforms index pre-warming into an automated performance guarantee.

Anytime Query. To mitigate P99 latency jitter caused by long-tail requests in interactive queries or cold starts, we introduce an anytime query mechanism designed to deliver meaningful partial results within a predictable time bound.

The core idea is to push timeout control from the query executor down to the physical scan operators—such as ANN index scans or sequential scans. Upon reaching a timeout, a scan operator gracefully terminates and returns the partial results accumulated so far, rather than causing the entire query to abort. This soft timeout strategy ensures a timely response, improving user experience compared to hard failure. By confining timeout logic to the stateless scan layer, the system preserves robustness and avoids the complexity of interrupting stateful operators such as joins.

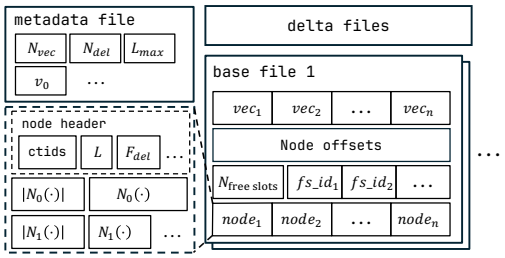


Figure 4: Nova Memory Structure

4 INDEX DESIGN

4.1 Nova Memory

Nova Memory (denoted as NovaM) adopts graph-based HNSW [27] as its core index, delivering strong performance when data fully resides in memory. HNSW is a popular vector index built on a multi-layer graph: sparse upper layers connect clusters for long-range jumps, while dense lower layers support accurate nearest neighbor search. Inspired by skip lists, HNSW searches top-down layer by layer, balancing global and local exploration through greedy traversal. At the implementation level, NovaM is engineered with a focus on performance optimization, supporting efficient space management and lock-free `scan`, `insert`, and `vacuum`.

Index Layout. Fig. 4 shows the storage structure of NovaM. The metadata includes the total number of vectors N_{vec}, the number of vectors marked as deleted N_{del}, the maximum HNSW layer L_{max}, the entry node v_0, and other parameters. Vectors are partitioned into multiple base files, each with a fixed capacity for storing vectors; the number of base files can be dynamically increased to scale the index capacity. Nova adopts a decoupled storage design for vectors and graphs: vector data is stored centrally, while graph structure information (neighbor nodes) resides in a separate node data area. Node offsets record the storage location of each node. Nova efficiently manages and reuses space from deleted nodes via a free node array. In the figure, $N_{\text{free slots}}$ denotes the number of free node slots, and fs_id_i indicates the position of a free slot. NovaM maintains per-node metadata, including the layer count L, a deletion flag F_{del}, a list of `ctids`, and the node's neighbors $N_i(\cdot)$ at each layer.

Insert. In HNSW, inserting a vector first requires randomly generating an integer L that specifies the new node's maximum layer. The algorithm then finds and connects to approximate nearest neighbors from layer 0 to L, followed by edge pruning. NovaM largely follows this framework, but faces two engineering challenges in production. The first is *index bloat and degradation* caused by duplicate vectors. Besides intentional insertion of identical vectors at the application

level, semantically duplicate vectors may arise from flush transaction retries after rollback, or from updates to non-vector columns in rows containing vectors. The original HNSW implementation creates a separate node for each duplicate vector, leading to index bloat and degraded recall. The second challenge is efficient reuse of obsolete space. As data is updated and deleted, the HNSW graph accumulates space left by logically deleted nodes. NovaM must reclaim this space to optimize storage, avoid index bloat, and maintain robustness.

Duplicate Vector Merging. To handle duplicate vectors, NovaM allows a single node to be associated with multiple `ctid`s. Duplicates mainly come from two sources: explicit duplicate insertions and PostgreSQL's MVCC mechanism. When a non-vector column is updated without a HOT (Heap Only Tuple) update, PostgreSQL creates a new heap tuple with a new `ctid` and propagates the write to all indexes, including the ANN index. This creates a new index entry carrying an identical vector with a different `ctid`, making duplicate merging important for index compactness and search efficiency. During insertion, NovaM checks whether any candidate neighbor at layer 0 is within a predefined distance threshold. If so, and its `ctid` list is not full, NovaM appends the new `ctid` to the list in a lock-free manner and completes the write. Otherwise, it creates a new physical node in the base file.

Space Reuse. In NovaM, the vacuum process maintains a free-slot array to reclaim obsolete nodes whose `ctid` lists are empty or whose writes have been aborted. During insertion, Nova first tries to obtain a free slot ID from the tail of this array. If successful, it reuses the corresponding node's layer count instead of generating a new random one. If the write is completed by appending to an existing `ctid` list, i.e., the acquired node is unused due to duplicate vector merging, the free node is returned to the array. Operations on the free-slot array are thread-safe because index scans never access it, and Nova enforces mutual exclusion between flush and vacuum.

Vacuum. Due to NovaM's graph structure and embedded `ctid` lists, `vacuum` must perform fine-grained maintenance on both the graph and storage, rather than simply deleting index entries. It traverses nodes to remove dead `ctid`s and reclaim nodes whose `ctid` lists become empty, mitigating index bloat. The process is designed to reclaim obsolete nodes safely while maintaining query performance, concurrency correctness, and high availability.

Removing a node from a graph index is expensive, as its incoming edges may be scattered across the graph. NovaM therefore batch-reclaims deleted nodes using three node states: active, deleted, and free. Active nodes are normal graph nodes. Deleted nodes have empty `ctid` lists after a vacuum pass; they are kept temporarily to preserve graph connectivity and can be traversed during query routing, but are not returned as results, used as neighbors for new nodes, or reused. Free nodes have been removed from the graph and are ready for reuse. By the end of `vacuum`, deleted nodes are removed from the graph and added to the free-slot array.

The `vacuum` operation has three phases: `remove_heap_ctids`, `repair_graph`, and `mark_free`. First, `remove_heap_ctids` removes dead `ctid`s from active nodes and marks a node as deleted if its list becomes empty. It also records the highest active node, excluding the entry point, for possible entry-point updates. If the number of deleted nodes exceeds a threshold, `repair_graph` scans active nodes, removes edges to deleted neighbors, and invokes `repair_node` to rebuild affected neighbor lists. Finally, `mark_free` globally marks reclaimed nodes as free, preventing concurrent queries from accessing them after reuse.

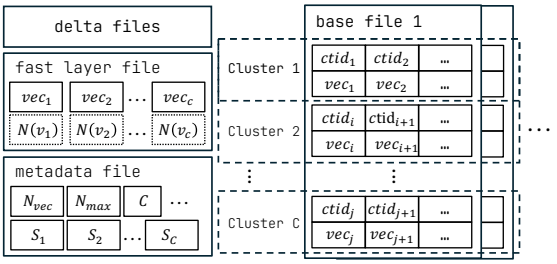


Figure 5: Nova Disk Structure

Lock-free Design. To ensure correctness between `vacuum` and `scan` without locking, we must handle three key concurrency scenarios. First, to manage concurrency between `remove_heap_ctids` and `scan`, NovaM employs an in-place two-pointer technique with atomic operations to filter out dead `ctid`s within a node: the left pointer seeks dead `ctid`s, while the right pointer seeks live `ctid`s; the live element pointed to by the right pointer is atomically copied to the position held by the left pointer. Finally, the size of the `ctid` list is atomically updated. This process may temporarily expose duplicate `ctid`s to a concurrent query, but these duplicates are safely eliminated when merging delta and base results, ensuring that the `ctid`s finally returned to the upper layer remain duplicate-free. Second, to handle concurrency between `repair_node` and `query`, `repair_node` prevents queries from accessing invalid edges by generating a new, complete neighbor list and atomically replacing the old neighbors one by one. During this process, a query may simultaneously see both new and old neighbors. However, all neighbor IDs read point to valid nodes that are indeed neighbors of the original node, preserving graph connectivity without significantly degrading recall quality. Additionally, during the `mark_free` phase, a query might concurrently access a node being processed. Therefore, `mark_free` only adds the slot of the free node to the free node array, without clearing the node's content. We have placed memory barriers at appropriate locations to implement release-acquire semantics, ensuring temporal correctness.

4.2 Nova Disk

While NovaM provides fast in-memory HNSW graph search, Nova Disk (denoted as NovaD) targets the disk-resident, large-scale, multi-tenant regime where index size exceeds available memory and workloads mix scans, inserts, and vacuums at high concurrency. We choose an IVF-based base layer over a graph-based on-disk index (e.g., DiskANN [20]) because its cluster-local organization yields bounded per-query I/O and mostly sequential access while naturally accommodating per-tenant memory budgets through on-demand cluster loading. This avoids the heavy build cost, in-memory PQ/hot-node residency, merge-time write amplification, and random-IO-driven P99 tail of a graph-on-disk index, which collide with the per-tenant memory and stability budgets NovaD must honor.

NovaD adopts a hybrid two-layer index structure, consisting of a fast layer for centroid navigation and a base layer for vector storage, as illustrated in Fig. 5. The base layer is inspired by inverted file (IVF) indexes [44], clustering vectors based on similarity, while the fast layer optionally constructs a graph index over centroids to

expedite subspace localization. This design maintains high retrieval performance under memory-resident constraints, effectively balancing cost and efficiency. The storage layout is optimized to minimize fragmentation and random I/O. Furthermore, lock-free concurrency control mechanisms for `scan`, `insert`, and `vacuum` operations ensure low-latency and stable nearest-neighbor search.

Index Layout. As shown in Fig. 5, the metadata includes the total number of vectors N_{vec}, the number of clusters C, the maximum per-base file capacity $N_{\max}$, and the size of each cluster $S_1, S_2, \ldots, S_C$, among other parameters. The fast layer stores the C cluster centroids $\text{vec}_1, \text{vec}_2, \ldots, \text{vec}_C$ in a compact, cache-conscious layout. Depending on the scale of centroids, this layer may optionally be augmented with a graph index; when enabled, the centroid list is immediately followed by the corresponding adjacency lists $N(v_1), N(v_2), \ldots, N(v_C)$, facilitating efficient navigation to proximate subspaces.

For the base layer, NovaD avoids the naive "one file per cluster" design, which creates many small files under large C and hurts file-system performance. Instead, all vectors are stored in a shared base file, where each cluster receives at most $N_{\max}/C$ slots. When a cluster exhausts its quota in the last base file, `expand` creates a new base file exclusively for its overflow; slots for other clusters remain logically allocated but physically unwritten, avoiding storage waste. Vector data and associated `ctid`s are stored in separate contiguous regions to improve sequential access and cache locality.

Scan. A `scan` operation begins by traversing the fast layer to identify candidate clusters whose centroids are proximate to the query vector, followed by a directed traversal of the vectors resident in those clusters. Each cluster's starting offset within a base file is statically determined; combined with its current size S_i, the exact set of file extents to be read is unambiguously derived. Consequently, scanning a single cluster incurs only a bounded number of random I/Os, primarily during transitions between base files. Reads of S_i are performed atomically, and visibility validation of the vector in the base file is conducted, thereby preserving consistency under concurrent `insert` and `vacuum` operations.

Adaptive Scan. Users can balance latency and recall via configurable parameters (e.g., `nprobes` indicating the number of clusters examined). More notably, NovaD incorporates an adaptive scan mechanism that dynamically calibrates the traversal scope based on the distance distribution between the query vector and the candidate centroids. This adaptive behavior navigates the trade-off between latency and recall. For instance, if the distances to multiple centroids lie within 20% of one another, the query vector is inferred to reside near cluster boundaries; in such cases, all proximate clusters are exhaustively scanned to safeguard recall. Conversely, when a single centroid dominates in proximity, the query is presumed to lie inside that cluster, enabling early termination to curtail I/O overhead.

Insert. The fast layer first resolves the target cluster for the incoming vector. Leveraging the current cluster size S_i, the system computes the precise slot location and writes the vector, followed by an atomic increment of S_i, entirely lock-free. Should the insertion cause the cluster to reach its slot capacity within the current base file, the aforementioned `expand` routine is seamlessly invoked, provisioning a new base file to accommodate future insertions for that cluster.

Vacuum. The `vacuum` operation reclaims storage by scanning base file entries and identifying logically deleted vectors. A dual-pointer technique is employed to record source entries and their designated destinations, which may reside in different base files. During relocation, the destination slot is first marked invisible; the source entry is then copied and made visible; finally, the source slot is invalidated. Upon compaction completion, S_i is updated, and any base file rendered entirely obsolete is promptly removed. It is worth noting that transient duplicate entries may become observable during certain phases of `vacuum`. However, correctness is ensured through deduplication performed during the `scan` operation.

5 ADVANCED QUERY PROCESSING

5.1 Hybrid Query

This section introduces the implementation and optimization strategies for hybrid retrieval involving single-table and cross-table queries in Nova. In general, Nova adopts a cost model-based approach similar to ADB-V, selecting the optimal plan from multiple candidate query plans. As shown in Fig. 6, building on the categories proposed by ADB-V, Nova supports the following plan types: Type 1 (Plan A, Brute-force): scans all rows that satisfy scalar predicates, computes their distances to the query vector, and returns the top-k results; Type 2 (Plan B, Plan C, Pre-filtering with Bitmaps): constructs a bitmap of row IDs that satisfy scalar predicates, and then uses this bitmap to skip ineligible vectors during the vector index scan; Type 3 (Plan D, Post-filtering): retrieves a candidate set of size $\sigma \times k$ via the vector index (where σ is an amplification factor) and subsequently verifies scalar predicates on these candidates. Nova inherits these strategies and introduces Type 4 (On-the-fly Filtering). Type 4 keeps Type 2's pre-filter semantics: a vector is admitted into the index scan only if it satisfies the scalar predicate. The only difference from Type 2 is how the predicate is materialized. Type 2 builds an explicit bitmap of qualifying row IDs in advance, whereas Type 4 represents the predicate as a callback expression evaluator invoked per candidate `ctid` during the vector index traversal, avoiding bitmap construction. Type 4 is therefore distinct from Type 3, which is a post-filter plan that retrieves $\sigma \times k$ vector candidates from the index first and then evaluates scalar predicates on them.

To further enhance performance and reduce random I/O from heap accesses, Nova incorporates multiple optimization techniques such as **partial index** and **covering index**. NovaM additionally introduces an iterative query mechanism that refines Type 3: rather than fixing the amplification factor σ at plan time, the mechanism incrementally expands the vector-search frontier until k post-filtered results are obtained. The iterative mechanism is a refinement of Type 3's post-filter pipeline rather than a new plan type, and is independent of Type 4's callback-based pre-filter.

Partial Index. As often observed in production workloads, queries frequently target hot-spot predicates. Nova supports partial indexes on these predicate-defined subsets. If a query is fully contained within a partial index, Nova executes a direct vector search on the restricted index, thereby narrowing the search scope. If multiple partial indexes are applicable, Nova selects the most specific one (i.e., the smallest subset) to execute the query.

Covering Index. In real-world applications, high-frequency queries often require filtering on specific structured fields. For instance, in the RAG scenario of Model Studio, almost all queries require filtering based on document IDs. Inspired by relational databases, Nova supports covering indexes. By materializing frequently queried

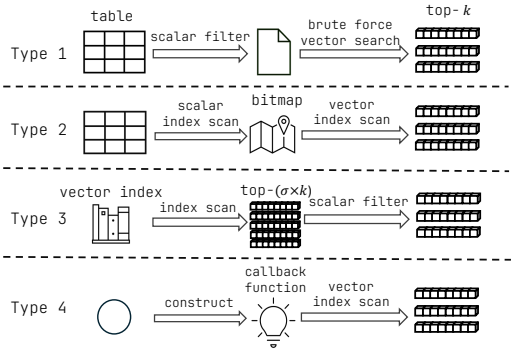


Figure 6: Nova Hybrid Query Framework

scalar attributes directly within the ANN index structure (e.g., within the graph nodes of NovaM), filtering can be performed during the index scan. This eliminates the need for auxiliary heap accesses and significantly reduces I/O overhead.

Iterative Query. To overcome the limitations of a static amplification factor in Type 3, which often fails to adapt to runtime data distributions, NovaM introduces an iterative retrieval mechanism leveraging the incremental traversal nature of graph indexes, inspired by [43]. The algorithm initially generates a small candidate batch; if scalar filtering yields fewer than k results, the search frontier is incrementally expanded by exploring adjacent nodes until the requirement is met or a stopping condition is met.

Plan Selection. Nova employs a hybrid selection strategy combining rule-based heuristics with a refined cost model. The system first prioritizes partial index rewriting when applicable: if a partial index covers the query predicates, the query is rewritten as a pure vector or hybrid retrieval targeting this reduced index. For pure vector retrieval, the choice between brute-force and index-based scans depends on whether the expected retrieval volume ($k \times \sigma$) represents a significant fraction of the total cardinality. For hybrid retrieval, Nova extends the ADB-V cost model by incorporating factors such as scalar selectivity, ANN scan cost, distance computation cost, heap access cost and bitmap construction overhead. For iterative queries, the model further accounts for the per-iteration cost and the expected number of iterations required to satisfy k.

5.2 Cross-Table Hybrid Query

As discussed in Sec. 2, cross-table hybrid retrieval is increasingly important in practice because of the natural one-to-many relationship between documents and chunks in RAG. A typical scenario queries the vector table `t_embedding` while applying filters based on metadata stored in the `t_metadata` table, with the two tables joined on the `doc_id` field. However, traditional optimizers often generate unstable execution plans when filter selectivities vary. To address this, we propose two candidate strategies: an ANN-driven Hash Join and a metadata-driven Nested Loop Join. We further introduce a cost-based adaptive strategy that dynamically chooses between them by comparing their estimated costs.

ANN-driven Hash Join. This plan prioritizes the vector search. It first performs a large-scale Approximate Nearest Neighbor (ANN) scan on the embedding table and uses the results to build a hash table. This table is then probed by the rows from the filtered metadata table.

The performance of this plan is dominated by the scale of the initial ANN scan. We model its cost, $Cost_{\text{HJ}}$, as:

$$Cost_{\text{HJ}} \propto \min\left(k \cdot \sigma \cdot N/N_f, S_{\max}\right).$$

Here, k is the query limit, σ is an ANN scan amplification factor, N is the total number of rows in the embedding table, and N_f is the estimated number of rows satisfying the metadata filter. The ratio N/N_f represents the inverse filter selectivity. The cost is capped by $S_{\max}$, a system-defined limit on the number of scannable points.

Metadata-driven Join. This plan applies the metadata filter first. It joins the filtered metadata table with the embedding table on `doc_id` to retrieve the chunks of all qualifying documents, then exactly computes the vector distances between the query and these chunks to produce the top-k. For the relational join step, ADBPG supports several physical operators—including Nested Loop, Hash Join, and Merge Join—and selects among them using standard relational cost estimation. We analyze the Nested Loop variant below, because it dominates the overall cost when the metadata filter is highly selective (i.e., N_f is small). This selective regime is precisely where the strategy is preferred over an ANN-driven Hash Join.

Conversely, this cost is determined by the total number of inner table accesses. Its cost, $Cost_{\text{NL}}$, is modeled as: $Cost_{\text{NL}} \propto N_f \times R_{\text{ppi}}$. Here, N_f (the number of qualifying documents in the outer loop) is multiplied by R_{ppi}, the estimated average number of chunks per document, representing the amount of work per lookup.

Adaptive Selection Strategy. During query optimization, the system compares the estimated costs, $Cost_{\text{HJ}}$ and $Cost_{\text{NL}}$. It selects the Hash Join when the metadata filter is less selective (i.e., N_f is large), because a single large ANN scan is cheaper than many small lookups. It favors the Metadata-driven Join when the filter is highly selective (i.e., N_f is small), because the small number of outer-loop iterations makes inner scans more economical. This adaptive strategy ensures the generation of stable and efficient query plans, mitigating performance volatility caused by fluctuating selectivities and the difficulty of estimating ANN costs accurately.

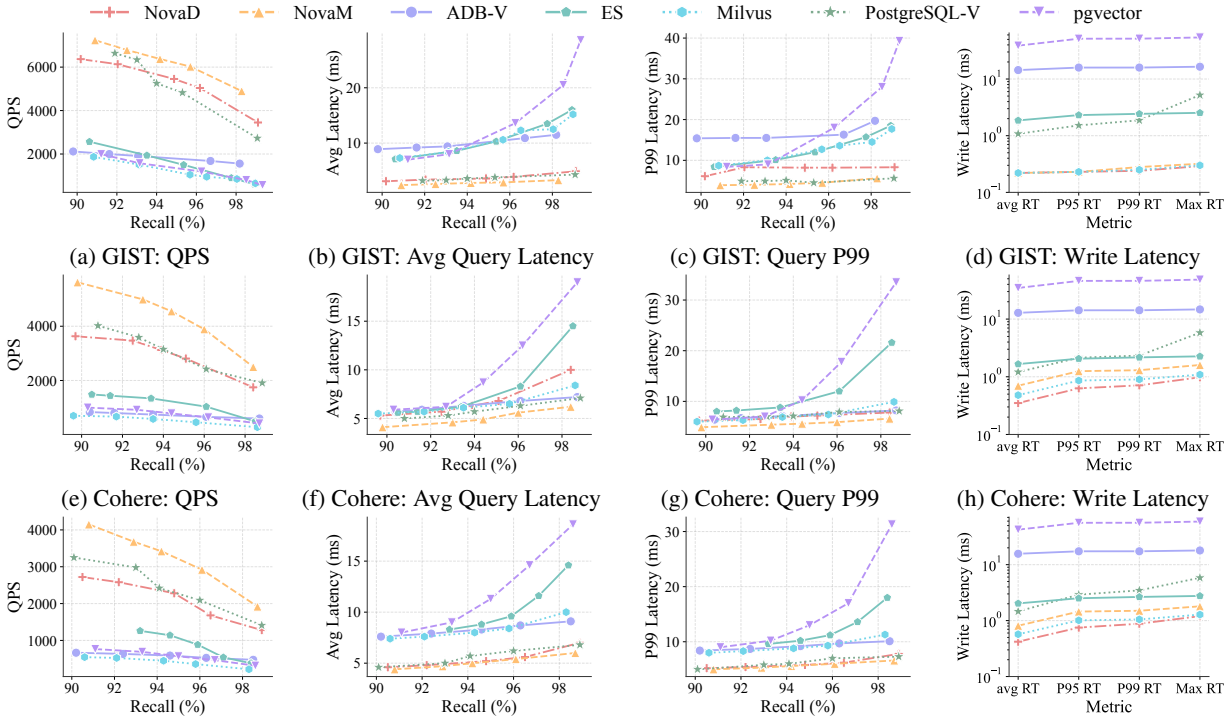
6 EVALUATIONS

This section evaluates Nova on public and in-house datasets. We compare it with its predecessor ADB-V [38], as well as Elasticsearch [3], Milvus [37], pgvector [4], and PostgreSQL-V [25].

6.1 Evaluation Setup

Environment. All experiments used Alibaba Cloud ECS g8i general-purpose instances with a 1:4 vCPU-to-memory ratio and Intel Xeon Emerald Rapids processors. Overall Performance (Sec. 6.2), Ablation Study (Sec. 6.4), and RAG Production Case Study (Sec. 6.5) ran on one `ecs.g8i.4xlarge` instance with 16 vCPUs and 64 GiB memory, matching the VectorDBBench [6] configuration. Multi-Tenant Large-Scale Testing (Sec. 6.3) ran on a 4-node cluster of `ecs.g8i.2xlarge` instances, each with 8 vCPUs and 32 GiB memory, totaling 32 vCPUs and 128 GiB memory, to simulate a production multi-tenant RAG deployment.

Datasets. We use two public datasets, GIST [30] and Cohere [1], and one internal dataset, Tongyi. GIST contains one million 960-dimensional image GIST feature vectors. Cohere contains ten million 768-dimensional text embeddings generated by the Cohere embedding model. Tongyi-10M and Tongyi-100M are in-house RAG



(a) GIST: QPS (b) GIST: Avg Query Latency (c) GIST: Query P99 (d) GIST: Write Latency

(e) Cohere: QPS (f) Cohere: Avg Query Latency (g) Cohere: Query P99 (h) Cohere: Write Latency

(i) Tongyi: QPS (j) Tongyi: Avg Query Latency (k) Tongyi: Query P99 (l) Tongyi: Write Latency

Figure 7: Performance on GIST 1M, Cohere 10M, and Tongyi 10M datasets, measuring QPS, average latency, P99, and Insert Latency.

datasets with 10 million and 100 million 1024-dimensional vectors, respectively, derived from chunked documents. Tongyi consists of two tables, `t_embedding` and `t_metadata`, storing chunk vectors and document metadata linked by `doc_id`, mirroring the Model Studio knowledge-base structure (see Sec. 2).

Baselines. We compare Nova with its predecessor ADB-V [38] and four widely used vector database systems: Elasticsearch [3] (ES), Milvus [37], pgvector [4], and PostgreSQL-V [25]. ES is a distributed search and analytics engine supporting hybrid search with vector fields and keyword retrieval. Milvus is a vector-native system using an LSM-tree-like storage structure and per-segment indexes. ADB-V and pgvector extend AnalyticDB (ADB) and PostgreSQL, respectively, with vector storage and similarity search; ADB-V is known for its hybrid retrieval design. PostgreSQL-V is an open-source PostgreSQL extension that decouples vector indexes from PostgreSQL's core engine and organizes vectors in an LSM-tree-like layout with mutable and immutable segments. We configure ADB-V, pgvector, ES, and PostgreSQL-V with HNSW indexes, and Milvus with IVF-RBQ, its recommended setting. For Nova, NovaD uses RaBitQ quantization, while NovaM runs without quantization because its quantization integration was still in progress at submission time.

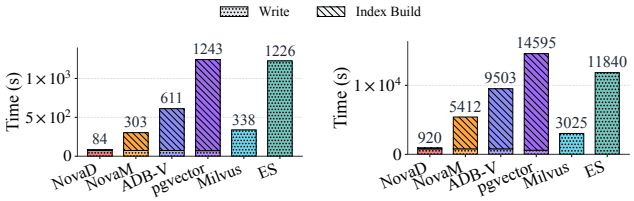
Metrics. For top-k similarity search, we measure accuracy by recall, defined as $|S \cap S'|/|S'|$, where S is the query result set and S' is the ground truth. We fix $k = 100$ in all experiments. Throughput is measured as queries per second (QPS) for reads and transactions per second (TPS) for writes. We benchmark throughput under concurrency levels 1, 8, 16, 32, 64, and 96 using the same search parameters, and report the peak QPS. Latency is reported by mean response time (mRT), P95 RT, P99 RT, and maximum response time (MaxRT).

6.2 Overall Performance

This section evaluates Nova under VectorDBBench's default single-table configuration [6], covering both NovaD and NovaM.

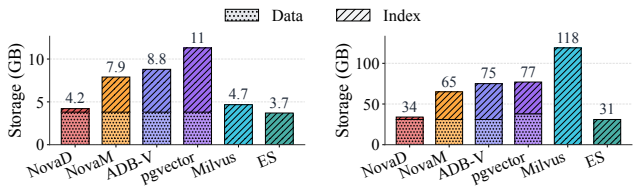
Vector Retrieval and Ingestion Performance. Fig. 7 presents the vector retrieval performance on the GIST-1M, Cohere-10M, and Tongyi-10M datasets. Across all three datasets and varying recall targets, NovaD and NovaM consistently achieve the highest throughput and lowest P99 latency, benefiting from their cache-friendly storage design and lightweight lock-free retrieval process. For instance, on GIST-1M at 95–96% recall, NovaD achieves over 5,000 QPS with a P99 latency of 8.2 ms, delivering approximately 4.2× the throughput and 1/2.2× the latency of pgvector, and 4.8× the throughput and 1/1.5× the latency of Milvus. Similarly, NovaM attains over 6,000 QPS at 95.7% recall with a P99 latency of 4.5 ms, substantially outperforming all other baselines. PostgreSQL-V achieves higher recall than pgvector at comparable QPS but still falls below NovaM in query throughput across the three datasets, since its per-segment HNSW results must be aggregated across multiple segments at query time. Although NovaD's index is disk-resident while NovaM's resides in memory, NovaD's quantized distance computation and smaller cache footprint compensate for much of the I/O cost, keeping NovaD's per-query latency and QPS close to NovaM's.

Figure 7 also illustrates the write performance across systems. Note that the write experiment uses a single ingestion thread per system, so write throughput is the reciprocal of the per-insert latency reported in the subplots, and we therefore omit a separate write-throughput subplot. On all three datasets, NovaD, NovaM, and Milvus exhibit orders-of-magnitude lower latency than other



(a) GIST 1M. **(b) Cohere 10M.**

Figure 8: Index construction time (in seconds).



(a) GIST 1M. **(b) Cohere 10M.**

Figure 9: Storage footprint comparison.

systems due to their asynchronous ingestion strategies. On the Cohere and Tongyi datasets, NovaD achieves even lower latency than NovaM and Milvus. This is because Milvus and NovaM trigger backpressure—throttling frontend throughput once growing segments or deltas exceed a threshold—when background flushes cannot keep up. NovaD's lighter-weight insertion design enables it to sustain high-speed ingestion with minimal performance degradation. PostgreSQL-V's write latency is markedly higher than the other systems because its LSM-tree-like layout commits inserts into a mutable segment but can later trigger compaction and re-indexing across immutable segments, producing significant write amplification. Since NovaM and ADB-V share the same HNSW base index and differ mainly in whether a delta layer is used, the gap between their write-latency curves quantifies the cost of bypassing the delta layer and inserting directly into the base index.

Index Construction Performance. Fig. 8 presents the index construction time of each system on the GIST-1M and Cohere-10M datasets. For Nova, ADB-V, and pgvector, we first insert data into a heap table and then build a vector index on that table. Following the methodology of VectorDBBench [6], we report the durations of these two phases separately. In contrast, Milvus and ES require the index structure to be defined before data ingestion; consequently, the reported write time already includes index building, and there is no separate index build phase. The write phases of NovaM, NovaD, ADB-V, and pgvector exhibit similar durations due to their comparable heap-table storage architectures. For graph-based indexing methods, index construction dominates the overall time.

NovaD reduces its index build time to only 12%–14% of the total, achieving the lowest overall construction time among all systems. This efficiency stems from its hybrid indexing approach, which builds a graph index only on a small subset of cluster centroids. As a result, NovaD outperforms the slowest system, pgvector, by factors of 14.8× and 15.8× on GIST-1M and Cohere-10M, respectively, and is 4.0× and 3.3× faster than Milvus with an IVF-RBQ index. Thanks to its file-based storage management that avoids the overhead of heavyweight page management, NovaM also delivers strong performance. It surpasses all other graph-based methods, achieving speedups of 2.0×, 4.1×, and 4.0× over ADB-V, pgvector, and ES on GIST-1M, and 1.8×, 2.7×, and 2.2× on Cohere-10M, respectively.

Storage Footprint. Fig. 9 presents the storage footprint comparison. NovaD exhibits significant index compression. On GIST 1M, NovaD's index is only 408 MB, 1/12 of ADB-V (5.0 GB) and 1/18 of pgvector (7.5 GB). On Cohere 10M, NovaD's index is 3.0 GB, while ADB-V, pgvector, and Milvus require 44 GB, 39 GB, and 118.95 GB, respectively. Hierarchical quantization encoding substantially reduces storage overhead while maintaining retrieval quality.

6.3 Multi-Tenant Large-Scale Testing

Distinct from the configurations in Sec. 6.2, this experiment simulates a production-grade, high-density multi-tenant RAG scenario where a single instance serves hundreds of isolated vector tables under stringent memory constraints. We generate 300 tables by randomly sampling from the Tongyi-100M dataset, such that the total index size is 6–8 times the available memory—matching the ratio observed in real-world production environments. This configuration imposes stringent requirements on memory efficiency, I/O management, and concurrent scheduling. Therefore, we primarily evaluate NovaD, which is designed for large-scale data scenarios. For Milvus and Elasticsearch, we use their officially recommended distributed deployments with equivalent compute resources.

```
SELECT e.id, e.doc_id, m.org_id, m.doc_name,
       (1.0 - (e.vector <=> $query_vector)) AS score
FROM t_embedding_{$tenant_id} e
JOIN t_metadata_{$tenant_id} m ON e.doc_id = m.doc_id
WHERE m.org_id = $org_id
ORDER BY score LIMIT $limit;
```

Query Type. We evaluate the above query type, which is a typical cross-table join hybrid retrieval scenario: identifying the k-nearest neighbors in the `t_embedding` whose associated metadata satisfy a given filter. Tenant isolation is achieved via table-level separation. The filter selectivity α is controlled via the `org_id` field, where $\alpha \in [0.01, 0.99]$. Notably, certain systems like Milvus do not natively support such join-based hybrid queries. For these systems, we utilize a pre-joined dataset of `t_embedding` and `t_metadata`. We emphasize that this workaround is a suboptimal compromise that introduces additional storage and update overhead.

Methodology. We adopt an open-loop load injection model by randomly selecting a subset of active tables from the total 300 tables to simulate concurrent queries from multiple tenants. Query requests are sent at a fixed rate, with the maximum concurrency set to 1,000 QPS, and each experiment runs for 120 seconds. We record end-to-end response times and compute mRT, P99 RT, MaxRT, and achieved QPS. All systems are tuned to maintain a recall of at least 95%.

Result. Fig. 10 presents the impact of the number of active tenants on system performance. Overall, as the number of active tenants increases, system QPS degrades and latency rises due to more frequent data swapping in and out of memory. Benefiting from its hybrid index design, NovaD consistently outperforms the other four systems across all tenant configurations. Similarly, as shown in Fig. 11, NovaD achieves the highest query throughput and lowest latency across all filter selectivity settings, indicating its robustness under varying selectivity in cross-table hybrid retrieval.

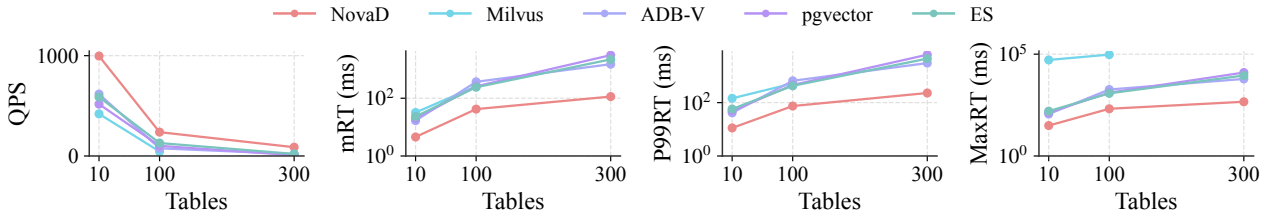


Figure 10: Impact of active tenant count on performance in multi-tenant scenarios (recall ≥ 95%).

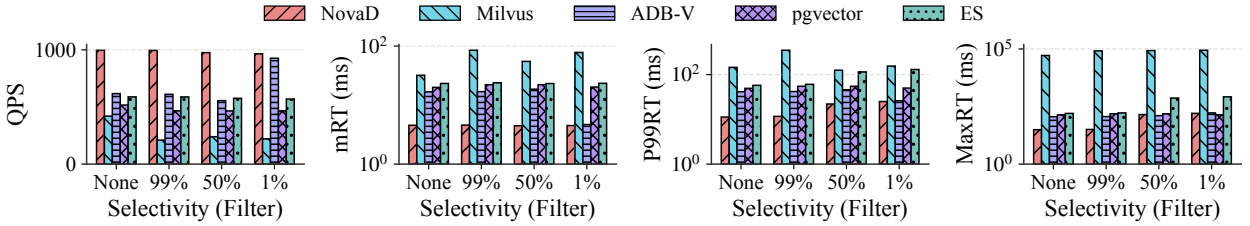
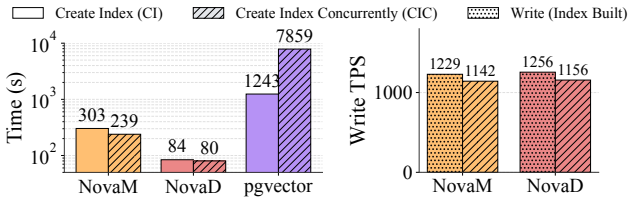


Figure 11: Impact of filter selectivity on cross-table hybrid retrieval performance (recall ≥ 95%).

Table 1: CIC impact on Top-100 recall on GIST 1M.

System	Create Index	Create Index Concurrently
NovaM	99.14%	99.17%
NovaD	99.19%	99.19%



(a) Build time: CI vs. CIC. **(b) Write TPS: Index Built vs. CIC.**

Figure 12: Create Index Concurrently ablation on GIST 1M [30]: (a) build time and (b) write TPS.

6.4 Ablation Study

In this section, we evaluate the independent contributions of Nova's design choices, specifically the Create Index Concurrently protocol and the impact of delta file size on query performance.

Create Index Concurrently (CIC). In online scenarios, index construction must minimize disruption to concurrent writes. PostgreSQL's native CIC, used by pgvector, follows a four-phase transaction protocol that waits for all existing transactions to complete, often causing "DDL storms" and build latency dominated by the longest-running transaction (T_{long}). As shown in Fig. 12(a), pgvector's build time reaches 7,859 s due to this bottleneck. In contrast, Nova uses its Delta-Base architecture to decouple the main build from concurrent DML by redirecting incoming writes to delta files, eliminating the wait for T_{long}. As a result, NovaD completes the build in only 80 s, matching a full-data build, while maintaining high write TPS (Fig. 12(b)) and 99.1% recall (Table 1). This confirms that Nova's three-phase protocol resolves lock contention in high-concurrency environments.

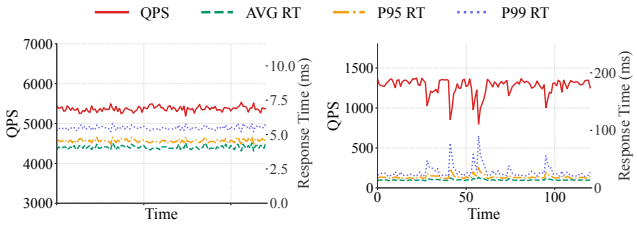
Impact of Delta File Size. Since Nova employs a layered storage architecture, data residing in delta files requires unindexed scanning, which can impact retrieval performance. Fig. 14 demonstrates that as the delta size increases from 1,000 to 10,000 vectors, QPS decreases from approximately 6,400 to 3,700, and P99 RT rises significantly. This observation validates the necessity of Nova's auto-flush mechanism, which is driven by both write heat and a size threshold (defaulting to 5,000 vectors).

Delta-Layer and Lock-Free Base Index. ADB-V uses the same HNSW base index as NovaM but inserts directly into the graph with page-level locks, while NovaM adds a delta layer and lock-free scan path, isolating their effects. Fig. 13 shows that under 1,000 TPS background inserts, NovaM maintains stable QPS and P95/P99 latencies even during flushes, thanks to its delta layer, lightweight `flush`, and lock-free scans. In contrast, ADB-V suffers collapsing QPS and spiking tail latencies due to page-level lock contention on the HNSW graph. Fig. 15 reports write throughput with 16 ingestion threads on a 10M-vector dataset: NovaM without an index gives the upper bound, and NovaM with the index stays close because delta files absorb writes and base-index maintenance runs asynchronously. ADB-V falls far behind, as every insert contends on the graph under fine-grained locks. Together, these results quantify the stability and throughput gains from NovaM's delta layer and lock-free scan path on top of the same HNSW algorithm.

6.5 Case Study in RAG Production

To validate Nova's stability under real-world production stress, we conduct a 24-hour comparative study on the multi-tenant Alibaba Cloud Model Studio [2] platform. The experiment simulates a high-stress maintenance window, where a large-batch ingestion at 3:00 AM triggers a resource-intensive `vacuum` to reclaim storage. We compare Nova against its predecessor, ADB-V [38].

As shown in Fig. 16(a), ADB-V degrades during `vacuum`, with query and upsert P99 latencies spiking sharply. This exposes limitations of its shared-buffer, page-based architecture. Large ingestion pollutes the shared buffer pool, evicts cached index pages, and forces later queries to incur frequent disk I/O. Meanwhile, `vacuum` scans the heap and locks pages to clean dead tuples, contending with concurrent queries on the same pages. This blocks user-facing operations and causes significant latency jitter. In contrast, Fig. 16(b)



(a) NovaM concurrency (b) ADB-V concurrency

Figure 13: Read/write concurrency of NovaM and ADB-V.

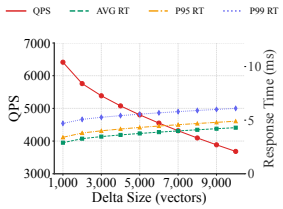


Figure 14: Delta size impact

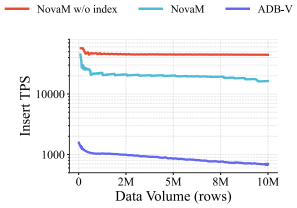
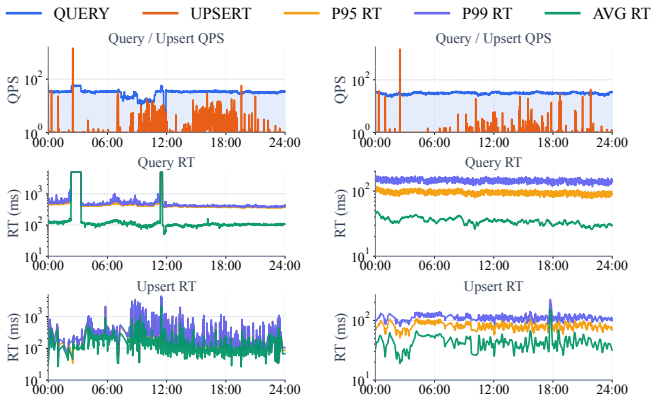


Figure 15: Insert TPS.



(a) ADB-V (b) Nova

Figure 16: 24-hour Performance Comparison of ADB-V and Nova in Alibaba Cloud Model Studio Scenario.

shows that Nova remains stable throughout the cycle. This robustness comes from its Delta-Base dual-layer architecture and file-based storage. With `mmap`, Nova uses the OS page cache, which is less vulnerable to sudden pollution than database-managed buffer pools. More importantly, the Delta-Base framework confines write conflicts to delta files, enabling lock-free queries on the base index. As a result, `vacuum` does not block foreground queries, providing the predictable low latency required by production RAG systems.

7 RELATED WORK

Vector Similarity Search. Vector similarity search has been extensively studied, with approximate indexes mainly falling into graph-based, clustering-based, and hashing-based methods. Graph-based methods [15, 20, 27] navigate the search space via structured topologies; clustering-based methods [11, 13, 39] partition vectors into local regions for pruning; and hashing-based methods use Locality-Sensitive Hashing [12, 18] for similarity-preserving dimensionality reduction. Quantization [16, 17, 21] further reduces distance-computation cost, and recent work has explored GPU to improve throughput [31, 35, 36]. These works mainly focus on algorithms, while real deployment still requires substantial system engineering.

Dynamic Vector Index. Several systems use buffer-and-merge update paths for streaming vector indexes. FreshDiskANN [33] maintains an in-memory delta graph merged into SSD-resident DiskANN; SPFresh [39] uses a block buffer locally rebalanced into an IVF base via LIRE; Quake [29] studies adaptive partitioning under workload drift; and LVQ [8] proposes locally adaptive quantization. These systems optimize base-index algorithms, such as cluster geometry, parameter tuning, and distance-computation cost. Nova adopts a similar dual-layer pattern, but targets system-level goals, including mmap-based file management, concurrent index construction, and lock-free query scans. Algorithmic refinements such as LIRE-style rebalancing and LVQ-style streaming quantization are orthogonal and can be integrated into Nova's framework.

Filtered Vector Search. Several works study vector search with range or label filters. VBASE [43] interleaves ANN search with scalar-predicate evaluation under relaxed monotonicity. Mohoney et al. [28] jointly evaluate vector neighborhoods and graph patterns for knowledge-graph queries. Digra [22] combines a vector index with a B-tree for dynamic range-filtered ANNS, while RangePQ [40] embeds range filtering into PQ-based ANN indexes. ADB-V [38] and Milvus [37] further explore plan enumeration and specialized indexes. These systems generally assume vectors and predicates reside in the same table or logical record. Yet in production multi-tenant RAG, metadata and chunk embeddings are often stored separately (Sec. 2). Nova addresses this cross-table setting with a cost-based adaptive join strategy that switches between ANN-driven Hash Join and Metadata-driven Join inside a production relational optimizer.

VDBMS. Vector Database Management Systems (VDBMS) generally follow two paradigms. Purpose-built systems, such as Milvus [37], Weaviate [7], and Pinecone [5], are designed for vector workloads, but often provide limited support for complex structured queries compared with mature systems such as PostgreSQL [34]. The other paradigm extends existing databases with vector indexing and retrieval, including pgvector [4], AnalyticDB-V [38], SingleStore-V [10], GaussDB-Vector [23], TigerVector [26], and PostgreSQL-V [25]. These systems leverage mature database infrastructures, but are constrained by legacy frameworks. Neither paradigm sufficiently addresses high-density multi-tenancy or cross-table hybrid search.

8 CONCLUSION

To meet the evolving demands of vector management in high-density multi-tenant scenarios, we introduce Nova, an analytical engine for low-latency, stable, real-time, and highly available vector retrieval and ingestion. Nova adopts a delta-base dual-layer architecture with file-based storage management and lock-free retrieval. We further develop two indexes, Nova Disk and Nova Memory, for efficient hybrid retrieval over single-table and cross-table queries. Experiments on public and internal datasets show Nova's advantages over existing baselines. Nova has been deployed on Alibaba Cloud as core vector retrieval infrastructure for platforms including Alibaba Cloud Model Studio. Ongoing work efforts focus on extending Nova to object storage and GPU acceleration to reduce storage cost and improve performance to meet diverse requirements.

REFERENCES

[1] 2024. Cohere Datasets and Embedding. https://docs.cohere.com/docs/datasets.

[2] 2026. Alibaba Cloud Model Studio. https://www.alibabacloud.com/en/product/modelstudio.

[3] 2026. Elastic. https://www.elastic.co.

[4] 2026. pgvector. https://github.com/pgvector/pgvector.

[5] 2026. Pinecone. https://www.pinecone.io/.

[6] 2026. VectorDBBench. https://github.com/zilliztech/VectorDBBench.

[7] 2026. Weaviate. https://github.com/weaviate/weaviate.

[8] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore L. Willke. 2023. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.* 16, 11 (2023), 3433–3446.

[9] K. E. Batcher. 1980. Design of a Massively Parallel Processor. (1980), 836–840.

[10] 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. *Proc. VLDB Endow.* (2024), 3772–3785.

[11] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *NeurIPS 2021*. 5199–5212.

[12] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Symposium on Computational Geometry*. 253–262.

[13] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. THE FAISS LIBRARY. *IEEE Transactions on Big Data* (2025), 1–17.

[14] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *SIGKDD*. 6491–6501.

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* (2019), 461–474.

[16] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* (2024), 167.

[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2014), 744–755.

[18] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*. 518–529.

[19] Martin Grohe. 2020. word2vec, node2vec, graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *PODS*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). 1–16.

[20] 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 *Advances in Neural Information Processing Systems*.

[21] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* (2011), 117–128.

[22] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. *Proc. ACM Manag. Data* (2025), 148:1–148:26.

[23] Guoliang Li, Ji Sun, James Pan, Jiang Wang, Yongqing Xie, Ruicheng Liu, and Wen Nie. 2025. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. *Proc. VLDB Endow.* (2025), 4951–4963.

[24] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *SIGMOD*. 835–850.

[25] Jiayi Liu, Yunan Zhang, Chenzhe Jin, Aditya Gupta, Shige Liu, and Jianguo Wang. 2026. Fast Vector Search in PostgreSQL: A Decoupled Approach. In *CIDR 2026*.

[26] Shige Liu, Zhifang Zeng, Li Chen, Adil Ainihaer, Arun Ramasami, Songting Chen, Yu Xu, Mingxi Wu, and Jianguo Wang. 2025. TigerVector: Supporting Vector Search in Graph Databases for Advanced RAGs. In *SIGMOD/PODS*. 553–565.

[27] Yury A. Malkov and Dmitry A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* (2016).

[28] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proc. ACM Manag. Data* (2023), 197:1–197:25.

[29] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2025. Quake: Adaptive Indexing for Vector Search. In *OSDI*. 153–169.

[30] Aude Oliva and Antonio Torralba. 2001. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. *Int. J. Comput. Vis.* (2001), 145–175.

[31] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *ICDE*. 4236–4247.

[32] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *CVPR*. 815–823.

[33] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR* (2021).

[34] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.

[35] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via CPU/GPU Collaborative Filtering and Re-ranking. In *FAST 2025*. 171–185.

[36] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. Fast k-NN Graph Construction by GPU based NN-Descent. In *CIKM '21*. 1929–1938.

[37] 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.

[38] 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. *Proc. VLDB Endow.* (2020), 3152–3165.

[39] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *SOSP*. 545–561.

[40] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* (2025), 152:1–152:26.

[41] Fangyuan Zhang, Mengqi Wu, Chunlei Xu, Yunong Bao, Jiyu Qiao, Yingli Zhou, Hua Fan, Caihua Yin, Wenchao Zhou, and Feifei Li. 2025. Streaming View: An Efficient Data Processing Engine for Modern Real-time Data Warehouse of Alibaba Cloud. *Proc. VLDB Endow.* 18, 12 (2025), 5153–5165.

[42] Fangyuan Zhang, Caihua Yin, Hua Fan, Fenghua Fang, Yineng Chen, Xuqi Wang, Mengqi Wu, Bing Chen, Tianbo Jin, Sibo Wang, Wenchao Zhou, and Feifei Li. 2025. AnalyticDB-PG: A Cloud-native High-performance Data Warehouse in Alibaba Cloud. *Proc. VLDB Endow.* (2025), 5139–5152.

[43] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *OSDI*. 377–395.

[44] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* (2006), 6.