Benchmarking Framework for Hybrid Relational-Vector Database Systems

Ayush Singh
Indian Institute of Technology Delhi
cs1200331@cse.iitd.ac.in

Kaustubh Beedkar
Indian Institute of Technology Delhi
kbeedkar@cse.iitd.ac.in

Srinivas Karthik
Microsoft
srv@microsoft.com

Harish Doraiswamy
Microsoft Research
harish.doraiswamy@microsoft.com

Srikanta Bedathur
Indian Institute of Technology Delhi
srikanta@cse.iitd.ac.in

Abstract

Modern database systems are increasingly expected to support hybrid queries that combine traditional relational operators with vector-based similarity search enabling applications such as semantic search, contextual recommendations, and multimodal analytics. However, existing benchmarks target either purely relational workloads or isolated similarity search, leaving a gap in evaluating database systems that aim to integrate both. In this paper, we present RvBench, a benchmarking framework for evaluating hybrid relational–vector workloads. RvBench comes with a relational data model that adapts the real-world MediaWiki schema to include vector columns and provides a suite of parameterized SQL templates that interleave similarity search with traditional SQL operations. The framework supports multiple similarity semantics (neighbors based on top-k, rank intervals, and sampled ranks) and allows generating hybrid workloads to evaluate database systems on both performance and retrieval quality. We demonstrate RvBench by benchmarking and comparing PostgreSQL and other commercial databases with a reference implementation of the workloads to highlight key trade-offs in query execution strategies and accuracy.

Keywords

Benchmarking, Relational-Vector Databases, Vector Database, Hybrid Queries

1 Introduction

Databases have long served as the cornerstone of data analytics, powering applications from transaction processing to large-scale reporting and business intelligence. Relational database systems in particular provide a rich set of operators (e.g., joins, filters, and aggregations) that allow users to express complex queries declaratively over structured data with strong consistency guarantees and automatic query optimization. Their ubiquity across industries attests to the robustness of the relational data model and the extensive ecosystem of query planners, storage engines, and indexing techniques that have been developed over decades.

1.1 Motivation

The widespread growth of machine learning (ML) and Artificial Intelligence (AI) technologies has resulted in the evolution of analytical workloads to include recommendations [16, 18, 22], semantic search [13, 25], and question answering [30]. As a consequence, it is increasingly common to have queries over the embeddings produced by ML models (e.g., word2vec [17, 28], large-scale transformer encoders [6, 10, 34, 39]) that map unstructured data (text, images, audio) into high-dimensional vector spaces [3, 9, 29]. These vector similarity-based queries, which use k-nearest neighbor (kNN) and/or approximate nearest neighbor (ANN) operations over these embeddings, enable "semantic" proximity searches that go beyond keyword matching. This has given rise to the new class of "vector databases" [33, 40, 42] that are designed to efficiently handle such kNN/ANN queries. Note that, these systems evaluate similarity search queries in isolation and do not support relational operations.

On the other hand, interleaving relational and similarity-search operations in a single *hybrid query* can yield significant advantages. First, it becomes easier to develop richer applications such as contextual recommendation systems, content-aware analytics pipelines, and real-time decision support, where structured and unstructured data must be processed jointly. As a concrete example, consider an e-commerce application where a user wants to view products that are similar to a given product with promotional offers or analyze which categories or brands have the most products in the least similar range to a given product. Table 1 lists several such use cases that illustrates the need for such interleaving queries. Second, from a database perspective, query optimizers can be leveraged to identify optimal plans to execute these queries. Thus, the developer does not have to individually handle/coordinate the relational and vector search operations. It is therefore not surprising that several academic (e.g.,[43, 45]), as well as industrial-strength database engines including PostgreSQL [8], SQL-server [36], and Oracle [31] have begun embedding vector support in SQL systems.

However, systematic exploration and evaluation of the design space of hybrid queries still remains nascent. Current benchmarks explore a only narrow slice of this space, as described next.

1.2 Related Work: Benchmarking Frameworks

Traditional Benchmarks. Traditional relational benchmarks include synthetic workloads such as TPC-H [38], TPC-DS [37], and JOB [21], as well as real-world workloads like STACK [27] from Stack Exchange. These benchmarks help test different aspects of the traditional relational query execution pipeline such as the query optimizer, indexes, cardinality estimations, etc. [5, 12].

Vector Benchmarks. To evaluate indexing structures and search algorithms for Approximate Nearest Neighbor (ANN) or vector search tasks, several benchmarks have been proposed. The most widely used is [24], featuring datasets like Glove1M, GIST1M, SIFT1M, and LAION. While popular, these datasets are limited in scale and lack diversity in embedding properties such as sparsity, dimensionality, and modality. To address this, the Big ANN Benchmarks [4] initiative introduced larger, more diverse datasets—e.g., Turing, SIFT, and OpenAI—and added support for hybrid queries

EDBT '27, Lille (France)

10.48786/edbt.2027.05

Table 1: Hyrbid query applications in E-commerce domain.

Visual Similarity Search: User uploads a product image to see the top-5 most visually similar products
Personalized Search with Filters: User searches for the top-10 similar products under 100 USD and from a specific brand
Category Insights: Top-20 similar products, grouped by category, showing only categories with more than 3 similar items.
Cross-sell Engine: Find top-3 similar products to a viewed item, and join with promotions table to show available discounts.
Mid-Tier Similarity Exploration: Iteratively browse beyond top-10 results by applying range-based similarity filters (e.g., ranks [11–20] to discover relevant but less obvious matches—supporting exploratory search
Market Segmentation Sampling: A business analyst samples products at ranks 2, 8, and 25 and their shipping region to view regions with at least 2 products.

combining vector similarity with boolean predicates. However, these benchmarks treat vectors as isolated entities–typically single-column tables or raw files–failing to reflect scenarios where vectors are embedded as first-class citizens in relational databases. This limits evaluation of semantic search over richly structured data.

Hybrid Relational-Vector Benchmarks. Currently available hybrid benchmarks focus only on evaluating the presence of filter predicates (range / categorical) together with ANN queries. These benchmarks can be classified into two categories. The first set of benchmarks [7, 14, 47] take existing data sources (i.e. vector embeddings) and augment them with scalar attributes using different data distributions. The second set of benchmarks [14, 41, 44] include data for which the scalar attributes naturally occur within the data. Some related but orthogonal works also include: (i) RAGO [19] that addresses end-to-end RAG system serving, proposing abstractions and optimizations for orchestrating LLM inference and vector search at the pipeline level, rather than within a database query optimizer; and (ii) Hybrid Querying over Relational Databases and LLMs ([46]) that focuses on beyond-database question answering through LLM integration and UDF-based reasoning.

Shortcomings. There are several shortcomings in these benchmarks. First, as mentioned above, their workloads only evaluate queries that interleave filters with ANN queries. Second, as a consequence of this, the datasets used in these benchmarks are single relation datasets (typically with a vector column and a attribute column on which the filter is defined). This therefore restricts the kind of queries that can be defined on them. Third, since the vector embeddings used are part of the original data source used in the evaluation, using these benchmarks would not be able to test the ANN index performance of alternative embeddings—having the ability to evaluate with diverse embedding models is crucial, as different real-world applications—ranging from semantic search and recommendation systems to multimodal data analytics rely on embeddings that capture distinct semantic, contextual, or domain-specific nuances. As mainstream database vendors continue to improve vector support on their offerings, the hybrid relational-vector workload space will only grow richer. Thus any relational-vector hybrid benchmark should satisfy the following properties to overcome the above shortcomings: (a) *expressive*–it provides a suite of templated queries capturing diverse interleavings of relational and similarity operators; (b) *extensible*–allows a diverse set of embedding models to be plugged in for evaluation; and (c) *scalable*–allows generating hybrid database with varying scale factors.

1.3 Contributions

In this paper, with the focus on analytical queries, we propose RvBench, an expressive and extensible benchmarking framework to evaluate hybrid relational-vector queries. The primary target audience for our benchmarking framework is SQL engines that support similarity constructs *natively*, in particular database and engine developers working on crucial components such as planner strategies, physical designs, index implementations, and execution methods. To this end, we design the benchmark database such that:

(1) the schema of the database follows a relational model which includes both traditional column types as well as vector columns (that store the embeddings). The database itself is based on real-world MediaWiki dataset [15] which was adapted to include vector columns on textual attributes.
(2) interchangeable embedding generators (e.g., openAI, BERT) can be used to generate the vector columns of varying dimensions.
(3) the database can be instantiated at different scale factors to support benchmarking under varying data/system loads.

RvBench comprises an expressive suite of queries that interleave relational constructs with similarity search operations[1]. Towards this end, we first define a taxonomy of the hybrid relational-vector query design space which is then used to create a suite of **39** query templates, each of which can be used to generate multiple queries with varying selectivity. The individual queries themselves were designed to reflect operational use cases such as quality control, metadata analysis and revision tracking of Wikipedia pages.

We demonstrate RvBench through an evaluation over popular open source (PostgreSQL with the *pgvector* [8] extension and Milvus [40]) and commercial engines. Since many of the queries proposed by RvBench are yet to be supported (through the use of vector indexes in the corresponding query plans) by these engines, we also provide an implementation of hand-crafted query plans for all the queries to act as a *reference*. This implementation was designed to highlight potential opportunities for improvement, also, that users can also plug-in a vector index of their choosing during the evaluation. Finally, we discuss the results of our experiments analyzing query performance and accuracy trade-offs of the above evaluated approaches, and highlight lessons for future hybrid query optimizer and index designs.

2 Background

We start with some background on similarity and approximate nearest-neighbor search queries as well as vector index structures that are commonly used to index vector embeddings.

2.1 Similarity Queries and Top-k Search

A similarity query operates over a collection of high-dimensional vectors by measuring a distance or similarity metric between a query vector q and each data vector v in the dataset. Common metrics include Euclidean (ℓ_2) distance and cosine similarity. In practice, users are typically interested in the top-k nearest neighbors to q, i.e., the k data vectors that minimize the chosen distance

[1]Other forms of hybrid queries may comprise operations (other than similarity search, eg: vector joins); we do not consider such queries in this paper.

or maximize similarity. Formally, given a set $V = \{v_1, v_2, \ldots, v_n\}$ and a distance function $d(\cdot, \cdot)$, a top-k vector search returns

$$\text{TopK}(q, k) = \underset{S \subseteq V,\ |S|=k}{\arg\min} \sum_{v \in S} d(q, v).$$

Exact top-k vector search requires computing $d(q, v)$ over all n vectors, which can become prohibitively expensive especially when the number of vectors n and/or the dimensionality d of the vectors are large. Therefore, approximate nearest-neighbor search is often employed in practice for these queries.

2.2 Approximate Nearest-Neighbor Search

To reduce query latency, many systems (e.g., [23]) employ approximate nearest-neighbor (ANN) algorithms that sacrifice accuracy for speed and lower resource consumption. Instead of exhaustively scanning the dataset, ANN approaches make use of an index structure (see below) to return a candidate set $\widetilde{S}$ whose elements are very likely to include the true top-k, with recall typically measured as

$$\text{Recall@}k = \frac{|\widetilde{S} \cap \text{TopK}(q, k)|}{k}.$$

By tuning ANN parameters (e.g., search depth, number of probes), users can trade off query accuracy against latency. ANN search has become integral to any production-scale workload, where millisecond-level responses are required over millions to billions of vectors.

2.3 Vector Index Structures

ANN approaches rely on specialized index structures that organize vectors to support sub-linear time search. We briefly discuss two commonly used indexes and refer readers to [32] for a more comprehensive discussion on different types of vector indexes.

Hierarchical Navigable Small World (HNSW) [26]. HNSW builds a multi-layer graph in which nodes correspond to data vectors that are connected to their nearest neighbors at each layer. Higher layers have sparser graph connectivity for long-range "jumps", while lower layers enable fine-grained navigation. Queries traverse from the top down, greedily moving to closer neighbors until convergence. HNSW offers high recall and low latency for a wide range of datasets [2]. At the same time, it also suffers from prolonged build times, increased memory consumption, and more complex maintenance of the index structure.

Inverted File (IVF) with Flat or PQ Encoding [20]. An IVF-based index partitions the vector space into m "cells" (e.g., via k-means clustering). Each cell holds a list of assigned vectors. At query time, only the n_{probe} closest centroids to q are scanned.

Two common IVF variants include: IVF-Flat that stores raw vectors in each cell; and IVF-PQ that applies product quantization (PQ) to compress vectors within cells. Here, PQ is the process of compressing high-dimensional vectors by splitting them into smaller sub-vectors that are then quantized independently using k-means clustering. Thus, IVF-PQ enables a fast and memory-efficient approximate nearest neighbor search by operating in the compressed domain at the cost of a slight accuracy loss.
Both these index types expose tunable knobs (graph connectivity, number of probes, quantization granularity) that affect the latency/accuracy trade-off. In the context of a relational system, embedding such indexes as first-class column indexes enables new hybrid query plans that interleave similarity search with filtering, joining, and aggregation operators.

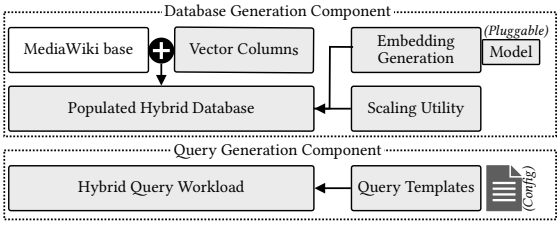


Figure 1: RvBench Architecture.

3 RvBench Architecture

In this section, we give an overview of the design of RvBench, an extensible and expressive benchmark to evaluate hybrid relational-vector workloads. As with any database benchmark, RvBench consists of two components: the database generator and the query workload generator. Our goal in the design of the benchmark database generator was twofold:

(1) *Extensibility:* Given the rapid proliferation of embedding models across domains, supporting a diverse set of models is crucial for flexibility and relevance as new embeddings continue to emerge, each capturing unique semantic and contextual properties. Allowing users to easily plug in and evaluate these models ensures that the benchmark remains adaptable to evolving trends reflecting real-world scenarios.
(2) *Scalability:* The user should be able to generate databases of different sizes. Unlike purely relational databases that scale with the number of rows and columns, hybrid relational-vector systems also scale with *vector dimensionality*. We therefore define the overall scale factor of a database generated by RvBench as a pair $SF = (S, D)$, where S represents the raw physical size of the database (similar to traditional benchmarks) and D denotes the embedding dimension. Varying these two parameters allows for a methodical evaluation of system performance as both relational and vector components vary. Furthermore, the ability to evaluate with varying D also helps test the efficacy of the underlying vector index.

The query workload consists of a diverse set of query templates designed to satisfy the following properties:

(1) *Hybrid Expressiveness*: The main aim of the query workload is to support a wide spectrum of queries that interleave relational operators with vector similarity search.
(2) *Parametric Flexibility*: The users should also be able to evaluate with different parameters (e.g., selectivity, distance metrics).

The architecture of the RvBench benchmark framework, designed to satisfy the above mentioned desiderata, is illustrated in Figure 1. It consists of two parts corresponding to database generation and query generation components.

Database Generation Component. RvBench builds upon the openly available real-world *MediaWiki* [15] database as a base to generate the hybrid relational-vector database. Specifically, we use the `Text`, `Revision`, `CategoryLinks`, and `Page` tables from the MediaWiki database and incorporate vector columns into the schema of the `Text` and `Page` tables as shown in Figure 2. Note that the primary goal of RvBench is to discern the effectiveness of the underlying system in handling practical queries involving relational and vector constructs, and not to stress test

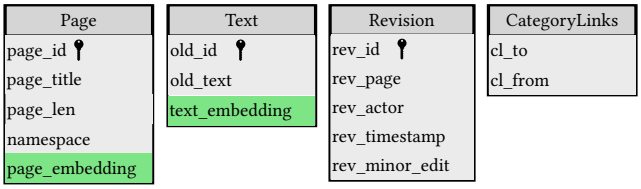


Figure 2: Benchmark schema comprising WikiMedia tables. Vector columns are highlighted in green color.

traditional aspects like "large" join orderings (like those present in benchmarks such as JOB). Thus, we surmise that these four tables suffice to exercise the different constructs (e.g., joins) of a query that interleaves relational with vector-based similarity operations.

RvBench, out of the box, supports all the models present in Python's transformer module providing users with the flexibility to use the state-of-the-art models for embedding generation. Additionally, the benchmark generation API also provides an interface that allows users to connect with other custom AI/ML models (e.g., private LLMs) to generate the embeddings for the vector columns.

The users can also specify the scale factor $SF = (S, D)$, of the database to be generated, where S is size in GB, and D is the vector dimension, which is determined by the embedding generation model. By default, the scale factor is set to $SF = (S = 100\text{ GB}, D = 384\text{ dimensions})$. This generates a database in which the `Text` and `Page` tables each have 10 million rows (and therefore 10 million vector embeddings). In our current implementation, databases of varying sizes are generated by appropriately sampling the rows of the above two tables from the full-sized WikiMedia database. Thus, at the default value of $D = 384$, the largest database RvBench generates has a size of 400 GB. When using an OpenAI embedding instead with $D = 1536$, this size increases to 1.4 TB.

Query Generation Component. This component comprises of a collection of parameterized SQL templates and parameter binder that, given a template and a configuration (e.g., query texts, k, thresholds, filter predicates), instantiates concrete hybrid SQL queries. Based on the motivating use cases (see Table 1), we devised a taxonomy of the relational-vector hybrid query design space (Section 4) which we then used to create the query templates (Section 5).

Discussion. RvBench is similar in spirit to the TPC benchmarking framework. Our goal is not to exhaustively explore configuration knobs or compare engines under all possible settings, but to provide a principled design space and representative query templates that highlight planner limitations and execution trade-offs for hybrid vector–relational workloads. Engine developers and researchers can extend this framework to incorporate additional configuration dimensions relevant to their systems.

4 Design Space for Hybrid Queries

Based on the motivating scenarios from the e-commerce example (Table 1), we systematically explore how various similarity constructs can interact with SQL constructs to propose a design space of hybrid relational-vector queries, illustrated in Figure 3.

Analogous to traditional relational queries, similarity search also necessitates filtering mechanisms tailored to user intent. In

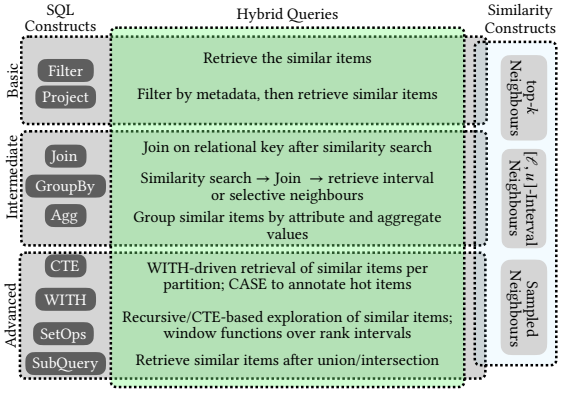


Figure 3: Design space for hybrid relational-vector queries.

this context, we identify three similarity constructs shown on the right side of Figure 3.

- **Top-k Nearest Neighbors:** This is the traditional form of vector search which retrieves the closest items most similar to a given query vector. The closest items can be specified either through a top-k constraint, or through a (maximum) distance threshold d.
- **Neighbors in an Interval$[\ell, u]$:** This is a generalization of the traditional neighbor query that retrieves items whose similarity ranks (or distances) fall between ℓ and u. This allows for more flexible retrieval beyond the most similar matches. For example, see the "mid-tier similarity exploration" use case in Table 1. In such exploratory search scenarios, users often seek to go beyond the top-k most similar results to examine the next tier of relevant items. Hybrid queries can enable this by applying rank-based filters on similarity scores (e.g., those ranked 10–20), allowing users to paginate through results in decreasing similarity bands. This supports discovery-oriented use cases where diversity or novelty is preferred over purely top-k results.
- **Sampled Neighbors:** This construct allows neighbors to be sampled, i.e., the queries are used to select items positioned at specific similarity ranks (or distances). For example, consider the market segmentation application in Table 1, where a business analyst samples neighboring products at ranks 2, 8, and 25.

As for the SQL constructs, we subdivide them into basic, intermediate, and advanced categories as shown in Figure 3:

- **Basic** queries involve attribute selection and projection, and those that have a `WHERE` clause for attribute-based filtering.
- **Intermediate** queries includes those with joins across related tables for richer context, `GROUP BY`, aggregation functions, and `HAVING` clauses for analytical workloads.
- **Advanced** incorporates `WITH` clauses, `CASE` expressions, set operations, and nested subqueries to support complex query logic.

This design space demonstrates that, by interleaving various similarity constructs with SQL constructs, one can support not just direct similarity search but also mid-tier exploration, *dissimilarity search*, diversity sampling, and novelty-driven discovery. Notably, sampled neighbor queries empower applications to sample products at arbitrarily chosen similarity levels, supporting

user-driven exploration and business analytics that are not possible with the traditional nearest neighbor queries.

In addition to the e-commerce application scenarios in Table 1, there are numerous additional use cases where such query patterns naturally arise. Domains such as content moderation, fraud detection, scientific research, and legal discovery all benefit from the ability to flexibly retrieve items by similarity, dissimilarity, diversity, or novelty. For example, in content moderation, multimodal and semantic similarity search enables the detection of harmful or plagiarized material across text, images, and video, even when content is obfuscated or rephrased. This approach is widely used to automate moderation workflows, improve accuracy, and reduce manual review requirements.

Furthermore, in recall-oriented settings, distance-based filtering (e.g., retrieving all items within a threshold distance d in vector space) is particularly valuable. This pattern, too, can be mapped onto the three query categories analogous to those shown in Figure 3: nearest within d (top-k), points within a distance band (continuous range), and points within multiple discrete distance bands. These versatile approaches enable a wide range of applications across industries, highlighting the broad relevance and impact of relational-vector query integration.

5 Query Workload

We now describe the query templates used in RvBench, inspired by real-world workloads from the WikiMedia database [1] and extend them with similarity search constructs in SQL. We derived these templates by examining over 50 MediaWiki query types to ensure that they are both realistic and semantically meaningful. We first describe the query templates for the traditional nearest neighbor-based queries with increasing relational complexity in Section 5.1. We then present analogous queries for the interval and sampled neighbor classes in Sections 5.2 and 5.3, respectively.

In all the queries, `s-op` represents the semantic similarity operation and `q` represent the input embedding (i.e., vector constraint of the query). Finally, for ease of presentation, we use bold font to differentiate between rank and distance based metrics.

Need for Hybrid Query Constructs. Although many of our benchmark queries can be expressed in standard SQL augmented with similarity functions (e.g., via pgvector), the resulting formulations are often verbose, unintuitive, and hard to compose. Beyond simple filter-plus-kNN patterns, expressing hybrid semantics typically requires complex rewrites with nested queries and manual ordering logic. This underscores the need for dedicated hybrid query constructs—language extensions that natively integrate relational and vector operations, improving expressivity while helping the optimizer better exploit vector indexes.

Discussion. We intentionally do not distribute templates uniformly across the design space since some regions correspond to application patterns that are more frequent in practice. Empirically, hybrid search workloads tend to concentrate in nearest-neighbor–centric patterns. Accordingly, our benchmarking framework includes more templates in these regions to reflect realistic usage rather than imposing an artificial uniformity.

5.1 Nearest Neighbor-based Queries

We present query templates for the 3 types of SQL constructs. For each type of query, there can be two variants: rank-based (i.e. top-k) and distance based. Due to space limitations, we provide SQL statements only for a representative set of queries. SQL statements for the other queries are simple variations of these and can be found in our repository [35].

5.1.1 Basic SQL Constructs. Queries that combine basic SQL constructs with vector constructs primarily test the quality of the underlying vector index. Depending on the relational filter constraint used, they could also asses the selectivity estimation quality of the optimizer (e.g., instead of using the vector index, it might be better to use a plan that filters the rows first, followed by computing the vector distances explicitly for each of the remaining rows when the filter is highly selective).

NQ 1: This is a traditional query that *lists the **top-k** Wikipedia page ids (with title) whose page title is most similar to* `q`. The SQL statement for this query can be written as follows:

```
SELECT page_id, page_title FROM page
ORDER BY page_embedding s-op q, page_id
LIMIT {k};
```

NQ 2: This is a distance-based equivalent of NQ 1 that *lists all Wikipedia page ids (with title) whose page title is semantically similar to* `q` *in the sense of having its **similarity distance** less than* `d`.

```
SELECT page_id,page_title FROM page
WHERE page_embedding s-op q < {d}
ORDER BY page_embedding s-op q, page_id;
```

NQ 3. This query is used to *list the **top-k** Wikipedia pages (sorted by similarity followed by page ID) that is most similar to* `q` *and whose content length is less than* `len`. The SQL statement for this query is similar to that of NQ 1 with the addition of a `WHERE` clause that filters based on the content length.

```
SELECT page_id,page_title FROM page
WHERE page_len < {len}
ORDER BY page_embedding s-op q, page_id LIMIT {k};
```

NQ 4. This distance-based equivalent of NQ 3 *lists Wikipedia pages (sorted by similarity distance, page ID) whose content length is less than* `len` *and whose **similarity distance** to* `q` *is less than* `d`.

5.1.2 Intermediate SQL Constructs. We now increase the complexity of the query through the addition of (relational) join and group-by aggregate clauses. In addition to evaluating vector indexing quality, these queries help also asses the query optimization strategies such as predicate ordering and push downs, as well as runtime execution strategies used for aggregation and index caching.

NQ 5. This query *retrieves the **top-k** IDs of Wikipedia revisions* (`rev_id`) *that are similar to* `q`, *and was revised between* `DATE_LOW` *and* `DATE_HIGH`. *The results are ordered by vector similarity followed by page ID.* As shown in the SQL statement below, this query requires the join of two tables `Text` and `Revision`.

```
SELECT rev_id FROM text JOIN revision ON old_id=rev_id
WHERE rev_timestamp>={date_low} AND rev_timestamp<={date_high}
ORDER BY text_embedding s-op q, old_id
LIMIT {k};
```

NQ 6. This is the distance-threshold equivalent of NQ 5 that *retrieves the IDs of Wikipedia revisions* (`rev_id`) *that were revised between* `DATE_LOW` *and* `DATE_HIGH`, *and whose **similarity distance** to* `q` *is less than* `d`.

NQ 7. This query *retrieves Wikipedia revision authors* (`rev_actor`) *ordered by the number of contributions to the **top-k** pages with length less than* `len`, *where the pages are similar to* `q`.

```
SELECT rev_actor,count(*) AS cou
FROM ( SELECT page_id FROM page WHERE page_len < {len}
       ORDER BY embedding s-op q, page_id
       LIMIT {k}
     ) as new_page join revision on page_id=rev_page
GROUP BY rev_actor
ORDER BY cou DESC;
```

NQ 8. The distance threshold equivalent of the NQ 7 that *retrieves Wikipedia revision authors* (rev_actor) *ordered by the number of contributions to pages with length less than* len, *where the* ***similarity distance*** *between the pages and* q *is less than* d.

5.1.3 Advanced SQL Constructs. These queries ups the ante through the inclusion of subqueries and advanced SQL constructs such as CASE statements, WITH clauses, etc. that also help test more complex optimization strategies.

NQ 9. This query *retrieves the* ***top-k*** *Wikipedia revisions for each year between* YEARL *and* YEARH *that are similar to* q. *For each selected revision, the query return its* year, *ID* (old_id), *and similarity score.* The SQL statement for this query, as shown below, makes use of a nested sub-query to accomplish the required task.

```
SELECT year, old_id, distance FROM (
    SELECT old_id, EXTRACT(YEAR FROM rev_timestamp) AS year,
        text_embedding s-op q AS distance,
        ROW_NUMBER() OVER (PARTITION BY EXTRACT(
            YEAR FROM rev_timestamp)
            ORDER BY text_embedding s-op q, old_id
        ) AS rank FROM text JOIN revision ON old_id = rev_id
    WHERE EXTRACT(YEAR FROM rev_timestamp) BETWEEN {year_low} AND {year_high}
) AS ranked_pages WHERE rank <= {k}
ORDER BY year DESC, distance ASC;
```

NQ 10. This is the distance threshold equivalent of the NQ 9 that *retrieves Wikipedia revisions for each year between* YEARL *and* YEARH *whose* ***similarity distance*** *to* q *is less than* d.

NQ 11. This query *retrieves the yearly distribution of Wikipedia revisions for the pages that are semantically similar to* q, *based on* ***top-k*** *similarity.*

```
SELECT EXTRACT(YEAR FROM rev_timestamp) AS year, COUNT(*)
FROM revision JOIN (SELECT page_id FROM page
    ORDER BY page_embedding s-op q, page_id
    LIMIT {k}
) AS filtered_pages ON revision.rev_page=filtered_pages.page_id
GROUP BY year;
```

NQ 12. This is the distance-threshold equivalent of NQ 11 that *retrieves the yearly distribution of Wikipedia revisions for the pages whose* ***similarity distance*** *to* q *is less than* d.

NQ 13. This query *retrieves Wikipedia revision authors* (rev_actor) *ranked by the total number of minor edits they made to the* ***top-k*** *pages that are similar to* q. As shown in the SQL statement below, this query performs a join between the Revision table and the result from a top-*k* subquery.

```
SELECT rev_actor, sum(rev_minor_edit) AS total_minor_edits
FROM (SELECT old_id FROM text
    ORDER BY text_embedding s-op q, old_id
    LIMIT {k}
) AS new_text JOIN revision ON old_id=rev_id
GROUP BY rev_actor
ORDER BY total_minor_edits DESC;
```

NQ 14. This is the distance-threshold equivalent of NQ 13 that *retrieves Wikipedia revision authors* (rev_actor) *ranked by the total number of minor edits they made to pages that are at a* ***similarity distance*** *less than* d *to* q.

NQ 15. This query *retrieves, for each Wikipedia category, the page that is most semantically similar to the average embedding (centroid) of all pages in that category.*

```
WITH category_centroids AS (
    SELECT cl.cl_to, AVG(p.page_embedding) AS centroid
    FROM page p JOIN categorylinks cl ON p.page_id=cl.cl_from
    GROUP BY cl.cl_to)
SELECT c.cl_to, p.page_id, p.page_title
FROM category_centroids c JOIN LATERAL (
    SELECT p.page_id, p.page_title
    FROM page p JOIN categorylinks cl ON p.page_id=cl.cl_from
    WHERE cl.cl_to=c.cl_to
    ORDER BY p.page_embedding s-op c.centroid, p.page_id
    LIMIT 1
) p ON TRUE;
```

NQ 16. This query retrieves the **top-***k* distinct results closest to either of the two query vectors q1 and q2.

```
SELECT * FROM (
    Select DISTINCT old_id, old_text FROM (
        SELECT old_id, old_text, text_embedding s-op q1 AS dist
        FROM text
        UNION
        SELECT old_id, old_text, text_embedding s-op q2 AS dist
        FROM text
    )
    ORDER BY dist
) LIMIT {k};
```

NQ 17. This query retrieves the **top-***k* distinct results closest to both of the two query vectors q1 and q2.

5.2 Interval Neighbor-based Queries

The queries in this category replace the top-*k* (or the *distance* < d) constraints with a two-sided interval. This set includes a total of 10 queries, IQ 1 to IQ 10, that are analogous to NQ 1 to NQ 10 respectively. To illustrate how these queries can be portrayed using SQL, we list two queries, IQ 1 and IQ 2, below. We refer the reader to our repository [35] for details on the other 8 queries.

IQ 1: This is the interval-neighbor equivalent of NQ 1 that *lists all the Wikipedia pages whose rank defined by its similarity to* q *is the* ***range*** $[\ell, u]$. The SQL statement for this query is shown below:

```
SELECT old_id,old_text FROM text
ORDER BY text_embedding s-op q, old_id
LIMIT {r-l+1} OFFSET {l-1};
```

IQ 2: This is the distance-threshold equivalent of the above query, where the rank is replaced by a **similarity distance** interval as shown in the SQL statement below.

```
SELECT old_id,old_text FROM text
WHERE text_embedding s-op q BETWEEN {d} and {d*}
ORDER BY text_embedding s-op q, old_id;
```

5.3 Sampled Neighbor-based Queries

Queries in this category, as the name indicates, are used to sample a discrete set of ranks (or intervals for the distance version). Similar to interval neighbor-based queries, we design 10 queries, SQ 1–SQ 10, in this category that are analogous to NQ 1 to NQ 10, respectively. We use two of the queries, SQ 1 and SQ 2 to illustrate how these queries are realized using SQL.

SQ 1: This query *lists Wikipedia pages whose content similarity* ***ranks*** *at specific positions* $r_1, r_2, \ldots, r_n$ *in terms of similarity to* q. This query is expressed in SQL as follows:

```
SELECT * FROM (
  SELECT old_id, old_text, ROW_NUMBER() OVER () AS rank
    FROM (SELECT old_id, old_text FROM text
          ORDER BY text_embedding s-op q) AS ordered_limited
) AS ranked WHERE rank IN ({r1}, {r2}, ..., {rn});
```

SQ 2: The distance-threshold equivalent of SQ 1, this query *lists Wikipedia pages whose content is between multiple* ***similarity distance*** *ranges–specifically, at least* d_1 *and at most* d_1^*, *or at least* d_2 *and at most* d_2^*, *or at least* d_n *and at most* d_n^* *— in terms of similarity to* q. *Rank the results by similarity and then by page ID.*

```
SELECT old_id, old_text FROM text
WHERE (text_embedding s-op q BETWEEN {d1} AND {d1*})
   OR (text_embedding s-op q BETWEEN {d2} AND {d2*})
   ...
   OR (text_embedding s-op q BETWEEN {dn} AND {dn*})
ORDER BY text_embedding s-op q, old_id;
```

6 Reference Implementation

While the current extensions in both relational and vector-native systems enable basic forms of hybrid query processing, their expressive power remains limited. As can be seen in Section 7, only 6 out of the 39 query templates make use of the vector index on a commercial engine. Even when the index is used for these queries (such as in PostgreSQL), they often use suboptimal execution strategies. Therefore, to better illustrate the potential opportunities for improvement in query optimization and execution within such systems, we include a *reference implementation* as part of RvBench that implements *hand-crafted query plans* for these queries. Though not optimized, it provides an initial target for these database engines.

The reference implementation is built on two core utility functions - namely, `Search` and `Reconstruct`. These two together enable vector-based retrieval and subsequent data reconstruction.

- `Search-Rank`(q, k) : Given a query embedding $\mathbf{q}$ and an index over stored embeddings, this function returns the top-k row identifiers corresponding to the nearest neighbors in the embedding space. While various indexes can be employed, in this work we use HNSW and IVFFlat as representative index types.
- `Reconstruct(table_name, columns, row_ids)`: This function retrieves data from the specified columns for a set of rows identified by `row_ids`, which may correspond to primary keys or auxiliary row-level identifiers maintained during indexing.

The reference implementation further provides users with an option to choose either a HSNW or an IVFFlat index for the `Search` functions. We use the Faiss library [11] to support these indexes.

Next, we describe the implementation for rank-based similarity queries, followed by distance-based queries. Finally, we briefly discuss some of the index-specific optimizations that were done.

6.1 Rank-based Queries

Traditional top-k (e.g., NQ 1). This is implemented simply invoking the `Search-Rank`$(q, *)$ followed by `Reconstruct`.

Filters (e.g., NQ 3). A filter is implemented as a set of iterative calls to `Search-Rank`$(\mathbf{q}, *)$ as follows. Let s denote the estimated selectivity of the predicate. Assuming uniform selectivity, we initially invoke `Search-Rank`$(\mathbf{q}, \lceil \frac{k}{s} \rceil)$ to account for potential filtering. If fewer than k qualifying rows are returned, we repeat the process by doubling the retrieval budget each time until the search returns k rows satisfying the filter predicates.

Joins (e.g., NQ 5). The reference implementation follows the same iterative refinement strategy used for filters. Specifically, we invoke `Search-Rank`$(\mathbf{q}, *)$ multiple times, progressively increasing the retrieval budget to compensate for tuples that may be filtered out due to the join and relational predicates. Let s_1 and s_2 denote the estimated selectivities of the relational filter and the join condition, respectively. The initial invocation is `Search-Rank`$(\mathbf{q}, \left\lceil \frac{k}{s_1 \cdot s_2} \right\rceil)$. This process continues, doubling the size of the retrieval if necessary, until k results that satisfy all predicates and joins are obtained.

Group by-Aggregates (e.g., NQ 7). Once the results from an invocation of `Search-Rank` / filter / join is obtained, we use `Reconstruct` to retrieve the group-by key and aggregation columns, which is then used to perform a hash-based group by and aggregation.

Interval & Sampled Neighbor-based Queries. In this class of queries, filters are based on interval or sampled neighbor semantics. To implement interval-based neighbor queries, where the goal is to return only those ranked between positions ℓ and u, we invoke `Search-Rank`$(\mathbf{q}, u)$ to obtain the top-u nearest neighbors to the query embedding $\mathbf{q}$. The final result is obtained by filtering out the top-$(\ell - 1)$ entries, retaining only the neighbors ranked from ℓ to u. Similarly, for sampled neighbor queries that retrieve specific ranks $r_1, \ldots, r_n$, we invoke `Search-Rank`$(\mathbf{q}, r_n)$ to obtain the top-r_n nearest neighbors to the query embedding $\mathbf{q}$, and then filter to retain only the desired ranks.

We are aware that this approach need not be the best way to tackle such ranges. However, we would like to reiterate that our goal in this context is to simply provide a reasonable baseline. As support for such hybrid queries are added to various database engines, we expect more performant and innovative solutions to be proposed.

6.2 Distance-based Queries

All queries in this category follow a similar implementation template akin to the rank-based similarity ones above. The queries differ in the sense that given a query embedding $\mathbf{q}$ and a distance threshold d, it retrieves vectors that lie within distance d of $\mathbf{q}$. This is implemented via a new utility function, `Search-Distance`$(\mathbf{q}, d)$, which returns all such qualifying entries.

We again use an iterative refinement strategy: we repeatedly invoke `Search-Rank`$(\mathbf{q}, *)$, beginning with a small retrieval size (e.g., 1), and progressively double the number of results requested. After each iteration, the maximum distant candidate is checked against d, and the process continues until there exists at least one neighbor in the returned result whose distance to $\mathbf{q}$ is greater than d.

6.3 Index Conscious Optimizations

For distance-based queries using HNSW, typical top-k search explores a portion of the graph whose size is roughly proportional to k. This makes incremental search strategies feasible—such as doubling k across iterations—especially when the stopping criterion (e.g., the top result's distance still below a constraint) isn't met. Importantly, in HNSW, query latency doesn't increase significantly with multiple such iterations.

However, the same strategy doesn't work well for IVF-Flat. Each search involves scanning full clusters, making repeated iterations expensive. Instead, it is more efficient to apply the distance constraint directly within a single search. Therefore, for IVF-Flat, we use Faiss's native distance-based filtering in a single pass, which achieves similar recall as multiple iterations but with lower latency.

7 Evaluation

We benchmark our reference implementation, pgvector (PostgreSQL's native vector extension), as well as publicly available commercial offerings using RvBench. Our evaluation is designed to quantify the trade-offs between accuracy and efficiency across

Table 2: Benchmark queries and their parameters used in the experiments.

	Queries and Parameters													
	NQ 2/NQ 14	NQ 3/NQ 7	NQ 4/NQ 8	NQ 5	NQ 6	NQ 11/NQ 16	NQ 12	NQ 9	NQ 10	IQ 1	IQ 2	IQ 3/IQ 7	IQ 4/IQ 8	IQ 5
Small	d=1.81	page_len=50 k=10	page_len=50 d=1.88	d1=2010 d2=2013 k = 10	d=1.81 d1=2010 d2=2013	k = 10	d=1.88	start_year=2010 end_year=2013 k = 10	start_year=2010 end_year=2013 d=1.81	limit=10 offset=40	d=1.81 d*=1.88	page_len=50 limit=10 offset=40	page_len=50 d=1.88 d*=1.94	d1=2010 d2=2013 limit=10 offset=40
Medium	d=1.88	page_len=100 k=50	page_len=100 d=1.94	d1=2012 d2=2018 k = 50	d=1.88 d1=2012 d2=2018	k = 50	d=1.94	start_year=2012 end_year=2018 k = 50	start_year=2012 end_year=2018 d=1.88	limit=50 offset=50	d=1.88 d*=1.95	page_len=100 limit=50 offset=50	page_len=100 d=1.94 d*=1.99	d1=2012 d2=2018 limit=50 offset=50
Large	d=1.95	page_len=300 k=100	page_len=300 d=1.99	d1=2015 d2=2024 k = 100	d=1.95 d1=2015 d2=2024	k = 100	d=1.99	start_year=2015 end_year=2024 k = 100	start_year=2015 end_year=2024 d=1.95	limit=100 offset=100	d=1.95 d*=2.03	page_len=300 limit=100 offset=100	page_len=300 d=1.99 d*=2.07	d1=2015 d2=2024 limit=100 offset=100

	IQ 6	IQ 9	SQ 1	SQ 2	SQ 3/SQ 7	SQ 4/SQ 8	SQ 5	SQ 6	SQ 9
Small	d=1.81 d*=1.88 d1=2010 d2=2013	start_year=2010 end_year=2013 l=41 r=50	ranks_list= [1,3,5,7]	distance_ranges= (1.81,1.88),(1.92,1.95)	page_len=50 ranks_list=[1,3,5,7]	page_len=50 distance_ranges= (1.88,1.94),(1.97,1.99)	d1=2010-10-07 d2=2013-12-29 ranks_list=[1,3,5,7]	d1=2010-10-07 d2=2013-12-29 distance_ranges= (1.81,1.88),(1.92,1.95)	start_year=2010 end_year=2013 ranks_list=[1,3,5,7]
Medium	d=1.88 d*=1.95 d1=2012 d2=2018	start_year=2012 end_year=2018 l=51 r=100	ranks_list= [5,10,15,20,25, 30,35,40,45,50]	distance_ranges= (1.88,1.95),(1.99,2.03)	page_len=100 ranks_list= [5,10,15,20,25, 30,35,40,45,50]	page_len=100 distance_ranges= (1.94,1.99),(2.04,2.07)	d1=2012-05-21 d2=2018-07-14 ranks_list= [5,10,15,20,25, 30,35,40,45,50]	d1=2012-05-21 d2=2018-07-14 distance_ranges= (1.88,1.95),(1.99,2.03)	start_year=2012 end_year=2018 ranks_list= [5,10,15,20,25, 30,35,40,45,50]
Large	d=1.95 d*=2.03 d1=2015 d2=2024	start_year=2015 end_year=2024 l=101 r=200	ranks_list= [1,11,21,31,41, 51,61,71,81,91]	distance_ranges= (1.88,1.92),(1.95,1.97), (1.99,2.01),(2.03,2.04)	page_len=300 ranks_list= [1,11,21,31,41, 51,61,71,81,91]	page_len=300 distance_ranges= (1.94,1.97),(1.99,2.01), (2.04,2.05),(2.07,2.08)	d1=2015-09-11 d2=2024-03-16 ranks_list= [1,11,21,31,41, 51,61,71,81,91]	d1=2015-09-11 d2=2024-03-16 distance_ranges= (1.88,1.92),(1.95,1.97), (1.99,2.01),(2.03,2.04)	start_year=2015 end_year=2024 ranks_list= [1,11,21,31,41, 51,61,71,81,91]

different indexing schemes and query parameters, and demonstrates interesting insights into the current state of hybrid workload support.

7.1 Setup

Implementation. We implemented RvBench in C++17 and compiled it with GCC 9.4.0. All query execution, accuracy calculation, and query file generation scripts are written in Python 3.8.10. We ran PostgreSQL 17.2 (Ubuntu 17.2-1.pgdg20.04+1) with pgvector 0.8.0 for the database experiments. All experiments were conducted on a server equipped with an Intel Xeon E5-2695 v3 (2.30 GHz, 8 cores) and 32GB of RAM. PostgreSQL configuration changes included disabling JIT, `work_mem = 24GB`, and `max_parallel_workers_per_gather = 0` (single threaded). `enable_seqscan` was set to `off` when evaluating pgvector with its index.

Dataset and Workload. We evaluated all hybrid queries (Section 5) with different parameter configurations. Due to space constraints, we limit our discussion to the ones listed in Table 2. For each query, we considered three variants: *small, medium, and large*, based on the query parameters. To generate the vector constraint `q` for different queries, we used ChatGPT to generate Wikipedia page titles and the corresponding page content. Unless otherwise mentioned, all queries were evaluated on a SF=(100,384) scale factor database. We generated embeddings using the Hugging Face transformers library with MiniLM-L12-H384-uncased (384-D) and all-mpnet-base-v2 (768-D). A 128-D variant was derived from the MiniLM vectors via Linear Component Analysis (LCA).

Metrics. We used *L2* and *cosine* distance to evaluate vector similarity. Due to space constraints, we only report results using the L2 distance and refer readers to [35] for experiments using the cosine distance. For each query, we measure query latency and in addition, precision@k and RMS error. Precision is defined for queries that return a non-aggregated result. For queries that compute an aggregate, we use RMS Error to measure the quality of the results.

Approaches. We benchmark our Reference implementation with two alternative vector indexing backends (HNSW and IVF-Flat), and examine the impact of varying query parameters such as the number of nearest neighbors k, distance thresholds, and filter selectivity. To contextualize the results, we benchmark pgvector, PostgreSQL's native vector extension, using its built-in similarity functions. pgvector is also evaluated using both HNSW and IVFFlat indexes. In the interest of space, we report the results of more popular HNSW indexes in some cases. Not all hybrid queries are currently supported by pgvector–such queries are executed by PostgreSQL without the use of a vector index;

Note that the Reference implementation isolates index-aware execution strategies (search-rank and reconstruct; in Section 6), and allows us to plug different ANN backends (HNSW, IVFFlat) so we can evaluate index-level tradeoffs independently of a full DBMS stack. This allows us to study opportunities an optimizer or planner that current engines could exploit. We would like to further clarify that our setup disables concurrency and logging given our focus on OLAP workloads. Additionally, we made sure that query parsing and planning time is excluded for PostgreSQL. Furthermore, our Reference implementation also uses disk—the relational tables are stored on files, and read on demand.

Results Organization. In what follows, we report our findings by categorizing the results into those that highlight: (a) effective use of the kNN index (Section 7.2); (b) planner limitations in terms of leveraging the index (Section 7.3); and (c) limitations of current kNN index abstractions underscoring the need for richer index semantics to support these advanced hybrid queries (Section 7.4). We then discuss the impact of query parameters and choice of indexes (Section 7.5), and data sizes across different

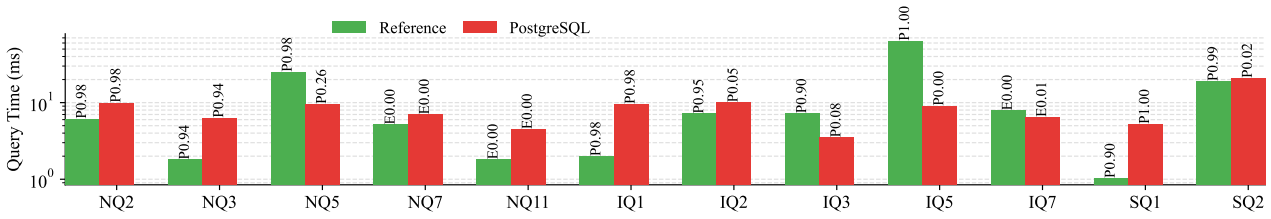


Figure 4: Query latency for hybrid queries where both the Reference implementation and pgvector use a vector index. Labels atop each bar report quality: P = Precision (non-aggregate queries) and E = RMSE (aggregate queries).

scale factor (Section 7.6). Finally, we document the current state of commercial systems in handling hybrid workloads (Section 7.7).

7.2 Effective *k*NN Index Usage

We first focus on hybrid queries where both Reference and PostgreSQL leverage the vector index. Figure 4 reports results using the HNSW index with the *medium* parameter setting (cf. Table 2).

Across neighbor-based queries (NQ 2, NQ 3, NQ 5, NQ 7, and NQ 11), Reference generally achieves accuracy comparable to or better than PostgreSQL while often delivering lower latency, except when adaptive probing incurs extra work. For simple retrieval queries such as NQ 2 and NQ 3, both systems repeatedly invoke `Search-Rank`$(q, *)$ with increasing (k); their quality is similar because the initial (k) estimates are adequate, and latency differences mainly reflect implementation details in the underlying HNSW libraries. In contrast, queries with joins or aggregates (NQ 5, NQ 7, NQ 11) reveal clear tradeoffs between accuracy, latency, and index-aware planning.

In NQ 5, which combines ANN retrieval with a join and filter, Reference achieves substantially higher recall than PostgreSQL by adaptively increasing (k) until enough join-qualified neighbors are found. By contrast, PostgreSQL 's fixed seed (k) often leads to premature termination of the ANN search. This highlights the sensitivity of HNSW accuracy to seed selection and stopping criteria: a sufficiently large initial (k), chosen with awareness of downstream relational operators, can significantly improve quality.

For aggregate-heavy variants such as NQ 7 and NQ 11, both systems can index-scan vectors and stream results into a GROUP BY, leading to comparable execution times. We report accuracy using RMSE for these queries, since correctness depends not only on retrieving the right tuples but also on computing correct aggregate values per group. As expected, RMSE is higher (i.e. lower accuracy) than in non-aggregate queries like NQ 3, reflecting the compounded uncertainty introduced by joins and aggregation. Nevertheless, when planning is index-aware, the Reference matches or slightly improves upon PostgreSQL 's accuracy without incurring prohibitive overhead.

Interval-based queries (IQ 1, IQ 2, IQ 3, IQ 5, and IQ 7) further stress the interaction between ANN search and relational semantics. In IQ 1, where the task is to return ranks in $[\ell, u]$, both systems behave similarly because the query maps cleanly to a top-u search followed by slicing. In contrast, IQ 2 and IQ 3 expose weaknesses in PostgreSQL's stopping logic: when rank is specified via distance thresholds or combined with filters, a poorly chosen seed k leads to lower precision. The Reference's strategy of probing until distance bounds are satisfied—or until enough filter-qualifying tuples are found—yields higher accuracy at modest additional cost. In join-heavy cases such as IQ 5 and IQ 7, this "retry" logic again trades slightly higher latency for substantially better result quality, underscoring the value of adaptive, index-aware probing.

Finally, the sampled queries (SQ 1 and SQ 2) demonstrate the clearest efficiency gap. In SQ 1, where only specific non-contiguous ranks are requested, the Reference retrieves only the top-7 neighbors and filters them, whereas PostgreSQL over-queries (e.g., $k = 51$ in HNSW) before trimming results, incurring unnecessary latency without any accuracy gain. This inefficiency arises from a lack of correlation between the ORDER BY clause and the rank selection in PostgreSQL's planning. In SQ 2, which replaces explicit ranks with a union of distance intervals, PostgreSQL again suffers from a poor initial k, while the Reference performs a single scan up to the global maximum threshold $d^*_{\max}$ and post-filters into subranges.

Takeaway. These results highlight the need for selectivity-aware planning of ANN parameters: poor estimation can lead to suboptimal choices of the seed k, while a cost model that jointly reasons about relational operators and vector indexes is critical for choosing both the index and its search-time parameters.

7.3 Planner Unsupported Queries

We now consider the set of queries which can be implemented using a vector index in Reference but not in PostgreSQL. The results are shown in Figure 5. For all of these queries, PostgreSQL resorts to a "brute-force" search thus having accurate results but with very high latency, often orders-of-magnitude greater than the Reference.

A common theme among these set of queries are that they mostly use a distance-based similarity in them. In case of NQ 4, the Reference iteratively queries the index and applies both the distance filter and the predicate filter to obtain the result. PostgreSQL, on the other hand, falls back to a `bitmap scan` on the table, and so pays the full cost of reading all rows before filtering. Similarly, when executing NQ 6, PostgreSQL resorts to a nested loop join with full scans due to a mix of vector and scalar predicates that are present along with join. Consequently, PostgreSQL's latency is very high, while our Reference implementation remains on par with NQ 4.

Figure 6 shows the query plans corresponding to the Reference implementation of NQ 8, and PostgreSQL's approach. This query extends NQ 4 's pattern with an aggregation, but without an `ORDER BY` clause. PostgreSQL fails to recognize the embedded *k*NN call and executes a full table scan with filtering, resulting in multi-order-of-magnitude slower performance. The Reference inlines the index lookup prior to join and thus achieves millisecond-scale response across all query sizes. The same argument applies to NQ 12 with `ORDER BY/LIMIT` clause because it's

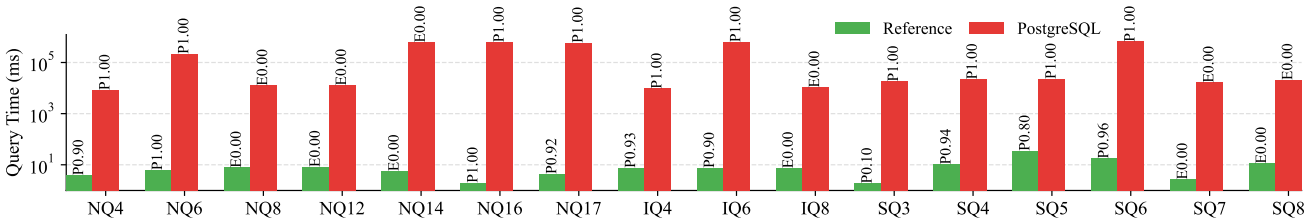
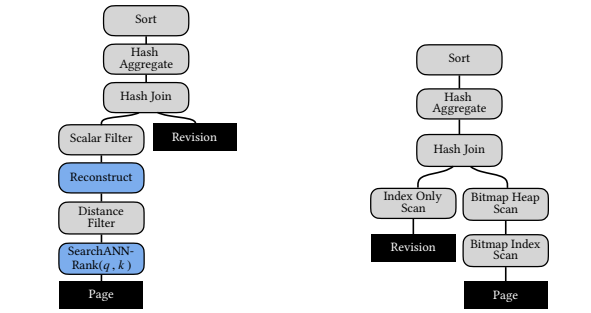


Figure 5: Query latency for hybrid queries for which pgvector does not leverage a vector index even though these queries are amenable to them. Labels atop each bar report quality: P = Precision (non-aggregate queries) and E = RMSE (aggregate queries).



(a) Reference Implementation **(b) Postgres+pgvector Plan**

Figure 6: Query Plans for NQ 8

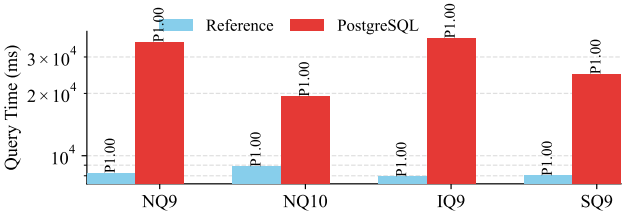


Figure 7: Results for hybrid queries where plans cannot leverage vector indexes. Labels atop each bar report quality: P = Precision (non-aggregate queries) and E = RMSE (aggregate queries).

query planner falls back to kNN index, and NQ 14 which also has a distance predicate that is followed by a join and an aggregation.

Query NQ 16 ranks items by the minimum of two distances—each computed against a different embedding—and is unsupported by PostgreSQL because its planner cannot compose multiple distance functions. Our approach exploits two key observations. First, to find the top-k by $\min(d_1, d_2)$, we independently retrieve the top-k neighbors sorted by distance d_1 and by distance d_2. Second, we merge these two ranked lists: for any item appearing in both, we record $\min(d_1, d_2)$; for items appearing in only one list, we conservatively approximate its score by the distance from that list (relying on the assumption that the two distance orderings are similarly distributed). Finally, we sort the merged candidate set by these combined scores and output the top-k. This two-phase strategy bounds the work done for the two index scans and produces exact results under realistic data distributions. PostgreSQL fails to leverage the index for the same reasons even for NQ 17 which ranks items by the maximum of two distances—again, one per embedding. Unlike the case of NQ 16, the Reference employs an *exponential-probe heuristic* until the intersection of the two result sets contains at least k items. It then computes $\max(d_1, d_2)$ for each item in this intersection and sorts them to finally return the top-k.

We also observe similar patterns where PostgreSQL's planner resorts to full scans in interval-neighbor queries. Queries IQ 5, IQ 6, and IQ 8, being interval variants of NQ 5, NQ 6, and NQ 8 respectively, follow the same trend as the top-k queries. However, they result in a lower overall accuracy, primarily due to the challenges introduced by interval-based filtering.

Similar to the interval-based queries, the presence of sampled ranks as part of a intermediate or complex query does not equate to the use of the index by PostgreSQL's planner. It thus fails to leverage the index for most of the sampled neighbor-based queries as well. While the Reference implementation can handle such queries, it suffers from a reduced result quality in some cases. For example, SQ 4, which mirrors the structure of IQ 4, the Reference aggregates query results over a range of sampled distance intervals, progressing from minimum to maximum, and applies the distance filter at the end. This approach allows the Reference to avoid scanning the entire table while still preserving accuracy. Similarly, the Reference adapts its search iteration to account for the join cardinality in SQ 5 yielding correct results at the expense of a small number of extra index calls. We see similar trends for SQ 6, SQ 7 and SQ 8.

Takeaway. PostgreSQL frequently falls back to full scans when distance predicates are combined with scalar filters, joins, or nested subqueries, and also fails to exploit the KNN index for distance filters without LIMIT or ORDER BY. While Reference can push vector predicates through the planner to achieve millisecond latencies for pure or filtered top-k queries, accuracy can still degrade for range predicates, underscoring the need for neighborhood-aware vector index design.

7.4 Index Unsupported Queries

There are seven queries in RvBench's workload that do not work well with the current vector index abstractions. The performance of four of these queries is shown in Figure 7. NQ 9 and NQ 10 both require one kNN (or distance) search to be performed per partition (year). Using current vector index abstractions would require the index construction on each of the partitions, which is not a practically viable approach. Thus, neither the Reference implementation nor PostgreSQL can directly support "partitioned kNN": both resort to sequential scans per partition. Likewise, IQ 9 issues an interval search within each partition, while SQ 9 requests discrete ranks within each partition, which are not supported either.

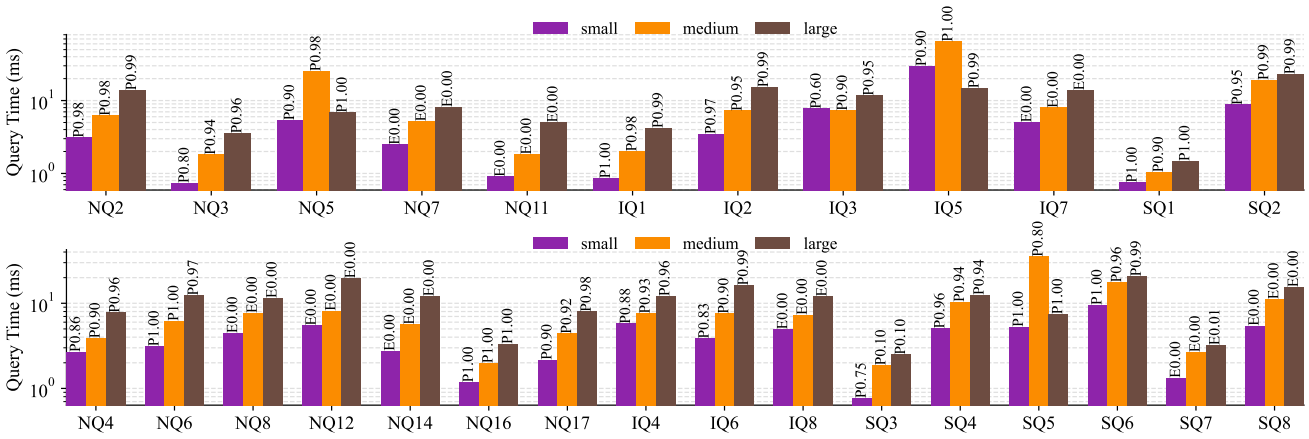


Figure 8: Impact of query parameters. Labels on bars report quality: P = Precision (non-aggregate queries) & E= RMSE (aggregate queries).

Takeaway. Overall, these queries highlight current limitations in kNN index abstractions when they are combined with grouping operations. Unlike simple filter-kNN, performing kNN search within group-by partitions is neither supported directly, nor is it straightforward to adapt existing index structures to enable such queries. This underscores the need for new index abstractions that natively support partition-aware similarity queries.

7.5 Impact of Query Parameters & Index Choice

In these set of experiments, we study the impact of query parameters, i.e. *small, medium and large* (cf Table 2). Figure 8 shows the results for workloads discussed in Sections 7.2 and 7.3. While here we briefly discuss some key results for the Reference implementation with HNSW index; we point readers to our repository for experiments with other index types and systems.

In terms of query latency, we observe the expected overall trend that execution time increases with query size, although the relationship is not uniform across all query classes. For top-k-based queries, larger values of k naturally require deeper index exploration, leading to higher search cost. For interval-based queries, increased query size—captured by larger values of r—similarly expands the search region and thus raises latency. Likewise, for sampled queries, higher maximum requested ranks translate into larger candidate sets and longer retrieval times. Notably, however, queries such as NQ 5, IQ 5, and SQ 5 deviate from this pattern: the `large` setting executes significantly faster than `medium`. This counterintuitive behavior arises from differences in filter selectivity (e.g., 52% vs. 10% for NQ 5), which dominate search cost in these timestamp-based workloads. More broadly, these results illustrate that a wider temporal range does not necessarily imply higher search time when relational selectivity changes substantially.

With respect to result quality, accuracy is generally stable or improves slightly as query size grows. This reflects the fact that larger result sets are more tolerant to small retrieval errors. Nevertheless, we identify several exceptions—specifically NQ 3, IQ 3, NQ 4, and IQ 4 —where accuracy improves more markedly from the `small` to larger settings. When k is small (as in low-top-k queries), even a few incorrect tuples can materially reduce precision after filtering. In contrast, for larger k, the same number of errors has a negligible effect on overall quality. An additional outlier is SQ 3, which exhibits particularly low accuracy in the `medium` and `large` regimes. Here, the interaction between the filter predicate and rank sampling amplifies sensitivity to minor rank misalignment: a shift of even one position can substantially degrade accuracy.

We also evaluate the impact of index choice (HNSW and IVFFlat) on query performance. We omit the results due to space constraints; these results are included in our repository [35] for completeness. Overall, we observe that the retrieval quality of HNSW is consistently superior to that of IVFFlat, aligning with the widely observed advantage of graph-based indexes in achieving higher recall [24].

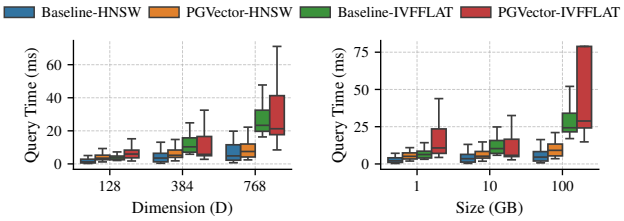
Takeaway. Overall, our experiments show that query sizes do impact the selectivity of the filter which in turn highlights the need of selectivity-conscious vector index design. Further choice of index also impacts the query performance both in terms of latency and accuracy.

7.6 Scalability Experiments

We now evaluate the Reference implementation and PostgreSQL by varying the scale factor (SF). For these experiments, we fix the query search parameters (i.e., # clusters searched for in IVFFlat and queue size for HNSW), while the index creation parameters are tuned according to the dataset size.

First, we fix the dataset size at S = 10GB and vary the embedding dimension D. The results are presented in Figure 9a. As expected, for all techniques query times increase with dimensions. However, the effect of varying dimension is more prominent for IVFFlat when compared to HNSW. This is due to the search strategy that differs across the index types: the number of neighbors searched using HNSW are small, due to which the distance computation time (which depends on the dimension of the vector) remains relatively low. On the other hand, since cluster sizes of the IVFFlat index are significantly larger than the degree of the HNSW graph, distances to more number of points needs to be computed, which results in the addition of a significant overhead as dimensions increase.

Next, we fix the embedding dimension at D = 384 and vary the dataset size S (Figure 9b). The Reference with HNSW consistently outperforms PostgreSQL across all sizes. Moreover, HNSW query times remain stable since the search space—governed by the graph degree (which remains constant across sizes) as well as



(a) Scalability (Size=10GB, D=*) (b) Scalability (Size=*, D=384)

Figure 9: Scalability experiments with varying SF.

the queue size used during querying (which is also constant). In contrast, IVFFLAT is more scale-sensitive: the recommended number of clusters is `nlist` $= n/1000$ for $n \leq 10^6$ and `nlist` $= \sqrt{n}$ beyond that, resulting in larger clusters and hence higher query times at larger scales.

7.7 Evaluating Commercial Systems

We also evaluated several commercial systems, denoted as **A1**, **A2**, and **A3**[2]. A1 represents a traditional enterprise-grade relational database with recent vector support extensions, A2 corresponds to a cloud-based analytical engine, and A3 is a distributed database optimized for mixed workloads. Our benchmarking revealed that current commercial systems remain in their early stages and do not yet fully support RvBench 's query workload. For instance, A1 used the index for only 6 of the 39 queries, while A2 does not support index construction beyond 200K embeddings. We also found A3 to be significantly slower than the others with it's default configuration.

Figure 10 additionally compares the Reference system against three offerings—PostgreSQL, Milvus, and A1 on the common subset of queries they all support. Due to space limitations, we only show results for 4 queries; more details can be found in [35]. We used a smaller dataset (500K rows in page table due to constraints on dataset size in free version of A1). Overall, PostgreSQL exhibits trends closely aligned with those observed in our earlier large-scale experiments. As before, PostgreSQL fails to support NQ 4 and IQ 4, but for the remaining queries its behavior is predictable and consistent with prior results.

Among other systems, Milvus supports a broader subset of queries (NQ 1–NQ 4 and IQ 1 –IQ 4) and generally achieves precision comparable to Reference, albeit with slightly higher latency in most cases. A notable exception is IQ 3, where precision drops visibly: this query combines a rank range with a relational filter and requires retrieving neighbors up to a much higher rank (200 versus 100 in the analogous NQ 3). The likely explanation is premature termination of the ANN search when too few filter-qualifying neighbors are found, leading to incomplete candidate sets — further indicate that residual gaps stem from underlying index quality rather than query planning.

A1 supports a different subset of queries (NQ 1, NQ 3, NQ 7, NQ 11, NQ 13, and NQ 19) and delivers perfect precision (100%) and zero RMSE across all of them, but at substantially higher latency than both the Reference and Milvus. This suggests that A1 internally considers a much larger candidate set before final selection. Unlike other systems, A1 does not expose low-level

[2]System names are anonymized in accordance with the "Dewitt Clause" which prohibit public benchmarking or performance comparisons of commercial systems without explicit vendor permission

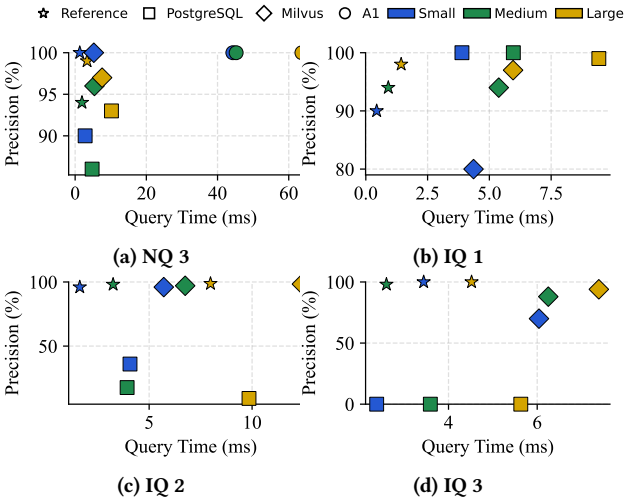


Figure 10: Results for a representative common subset of queries supported by all systems.

tuning knobs such as `ef_search`; instead, it accepts a target accuracy parameter per query, which it maps internally to search-time behavior. While this abstraction simplifies usage, it obscures control over performance and not well suited for workloads where accuracy depends heavily on filter selectivity and join structure.

Takeaway. These results indicate that current commercial systems are actively maturing in their support for complex relational-vector queries with clear tradeoffs between usability, latency, and accuracy.

8 Conclusion

We proposed RvBench, a benchmarking framework to evaluate analytical database systems for hybrid queries that interleave relational operators with vector similarity search. It comprises of two components: The first generates a relational-vector database based on the MediaWiki tables with configurable parameters such as scale factor and embedding model. The second generates the hybrid relational-vector query workloads–this was accomplished by first studying the design space of hybrid queries, which was then used to create query templates comprising parameterized queries that interleave SQL constructs with similarity search. We also provided a Reference implementation of the hybrid query workload. An evaluation with PostgreSQL and commercial engines reveals that tight planner–index integration and adaptive index-probing strategies reduce latency by up to an order of magnitude. Additionally, these strategies maintain higher precision (lower RMSE), especially for complex queries involving filters, joins, nesting, and partitioned retrieval

In future, we plan to extend RvBench to leverage large language models to generate more realistic textual data and embeddings for very large scale factors. We also intend to explore new index and planner abstractions, such as partition-aware kNN, interval/sampled neighbor primitives, and adaptive parameter tuning, to better support the full spectrum of hybrid relational–vector queries in next-generation analytical engines. Our focus was on OLAP-style vector–relational workloads, while extending to OLTP workloads with index maintenance, inserts, and transactional updates is an interesting future work.

Artifacts

The code developed for this work and used for all the experiments presented is made publicly available at https://github.com/IITD-data-systems/RVBench

References

[1] [n. d.]. MediaWiki SQL queries. https://quarry.wmcloud.org/query/runs/all?search_term=revision.

[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374. doi:10.1016/j.is.2019.02.006

[3] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2055–2063.

[4] Big ANN Benchmarks 2025. https://big-ann-benchmarks.com

[5] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 61–76.

[6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 1877–1901.

[7] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6, Article 246 (Dec. 2024), 27 pages. doi:10.1145/3698822

[8] contributors. 2025. Open-source vector similarity search for Postgres. https://github.com/pgvector/pgvector Accessed: 2025-07-01.

[9] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys)*. ACM, 191–198.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. doi:10.18653/V1/N19-1423

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[12] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. *Proc. VLDB Endow.* 13, 8 (April 2020), 1206–1220. doi:10.14778/3389133.3389138

[13] Ming Du, Arnau Ramisa, Amit Kumar K. C, Sampath Chanda, Mengjiao Wang, Neelakandan Rajesh, Shasha Li, Yingchuan Hu, Tao Zhou, Nagashri Lakshminarayana, Son Tran, and Doug Gray. 2022. Amazon Shop the Look: A Visual Search System for Fashion and Home. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 2822–2830. doi:10.1145/3534678.3539071

[14] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=8t8zBaGFar

[15] Wikimedia Foundation. 2025. Wikipedia Database Dumps. https://dumps.wikimedia.org.

[16] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 311–320. doi:10.1145/3219819.3219885

[17] Martin Grohe. 2020. Word2vec, Node2vec, Graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *Proceedings of the 39th ACM Symposium on Principles of Database Systems (PODS)*. ACM, 1–16.

[18] Yunzhong He, Yuxin Tian, Mengjiao Wang, Feier Chen, Licheng Yu, Maolong Tang, Congcong Chen, Ning Zhang, Bin Kuang, and Arul Prakash. 2023. Que2Engage: Embedding-based Retrieval for Relevant and Engaging Products at Facebook Marketplace. In *Companion Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*, Ying Ding, Jie Tang, Juan F. Sequeda, Lora Aroyo, Carlos Castillo, and Geert-Jan Houben (Eds.). ACM, 386–390. doi:10.1145/3543873.3584633

[19] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. 2025. RAGO: Systematic Performance Optimization for Retrieval-Augmented Generation Serving. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[20] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594 42nd International Conference on Very Large Data Bases (VLDB) 2016 — New Delhi.

[22] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-based Product Retrieval in Taobao Search. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, Feida Zhu, Beng Chin Ooi, and Chunyan Miao (Eds.). ACM, 3181–3189. doi:10.1145/3447548.3467101

[23] Yanjun Lin, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2025. Survey of Filtered Approximate Nearest Neighbor Search over the Vector-Scalar Hybrid Data. *arXiv preprint* arXiv:2505.06501 (2025). doi:10.48550/arXiv.2505.06501 Preprint posted May 10, 2025; provides a taxonomy and evaluation framework for hybrid vector-scalar ANN search.

[24] M. Aumüller and E. Bernhardsson and A. Faithfull. 2019. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Information Systems* (2019). doi:10.1016/j.is.2019.02.006

[25] Alessandro Magnani, Feng Liu, Suthee Chaidaroon, Sachin Yadav, Praveen Reddy Suram, Ajit Puthenputhussery, Sijie Chen, Min Xie, Anirudh Kashi, Tony Lee, and Ciya Liao. 2022. Semantic Retrieval at Walmart. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 3495–3503. doi:10.1145/3534678.3539164

[26] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (April 2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[27] Ryan Marcus. 2021. Stack dataset. rmarcus.info/stack.html

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL] https://arxiv.org/abs/1301.3781

[29] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[30] 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* 1, 2 (2023), 197:1–197:25. doi:10.1145/3589777

[31] Oracle. 2025. Oracle AI Vector Search. https://www.oracle.com/in/database/ai-vector-search/. Accessed: 2025-07-18.

[32] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *VLDB J.* 33, 5 (2024), 1591–1615. doi:10.1007/S00778-024-00864-X

[33] Pinecone 2025. https://www.pinecone.io/

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[35] RvBench 2025. https://github.com/IITD-data-systems/RVBench

[36] SqlServer 2025. https://www.microsoft.com/en-in/sql-server

[37] Transaction Processing Performance Council. 2008. TPC Benchmark™ DS (Decision Support). http://www.tpc.org/tpcds/. Standard benchmark for decision support systems including complex queries and concurrent data maintenance.

[38] Transaction Processing Performance Council. 2014. TPC Benchmark™ H (Decision Support). http://www.tpc.org/tpch/. Accessed: 2025-07-18.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5998–6008.

[40] 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 '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2614–2627. doi:10.1145/3448016.3457550

[41] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/32e41d6b0a51a63a9a90697da19d235d-Abstract-Conference.html

[42] Weaviate 2025. https://weaviate.io/

[43] 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.* 13,

12 (2020), 3152–3165. doi:10.14778/3415478.3415541

[44] 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. In *Proceedings of the VLDB Endowment*, Vol. 13. VLDB Endowment, 3152–3165. doi:10.14778/3415478.3415541

[45] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 377–395. https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi

[46] Fuheng Zhao, Divyakant Agrawal, and Amr El Abbadi. 2025. Hybrid Querying Over Relational Databases and Large Language Models. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[47] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), 69:1–69:26. doi:10.1145/3639324