VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity

Qianxi Zhang[1] **Shuotao Xu**[1] **Qi Chen**[1, *] **Guoxin Sui**[1] **Jiadong Xie**[1, 2] **Zhizhen Cai**[1, 3]
Yaoqi Chen[1, 3] **Yinxuan He**[1, 4] **Yuqing Yang**[1] **Fan Yang**[1] **Mao Yang**[1] **Lidong Zhou**[1]
[1]Microsoft Research Asia [2]East China Normal University
[3]University of Science and Technology of China [4]Renmin University of China

Abstract

Approximate similarity queries on high-dimensional vector indices have become the cornerstone for many critical online services. An increasing need for more sophisticated vector queries requires integrating vector search systems with relational databases. However, high-dimensional vector indices do not exhibit monotonicity, a critical property of conventional indices. The lack of monotonicity forces existing vector systems to rely on monotonicity-preserving tentative indices, set up temporarily for a target vector's `TopK` nearest neighbors, to facilitate queries. This leads to suboptimal performance due to the difficulty to predict the optimal K.

This paper presents VBASE, a system that efficiently supports complex queries of both approximate similarity search and relational operators. VBASE identifies a common property, *relaxed monotonicity*, to unify two seemingly incompatible systems. This common property allows VBASE to circumvent the constraints of a `TopK`-only interface to achieve significantly higher efficiency, while provably preserving the semantics of `TopK`-based solutions. Evaluation results show VBASE offers up to three orders-of-magnitude higher performance than state-of-the-art vector systems on complex online vector queries. VBASE further enables analytical similarity queries that previous vector systems do not, and shows 7,000× speedup with 99.9% accuracy of exact queries.

1 Introduction

Recent advances in deep learning (embedding) models map almost all types of data (*e.g.*, images, videos, documents) into high-dimension *vectors* [60, 66, 88]. Queries on high-dimensional vectors enable complex semantic-analysis that was previously difficult if not impossible, thus they become the cornerstone for many important online services like search [25, 51], eCommerce [54], and recommendation systems [49, 53, 56, 84]. The "online" nature of these services requires vector search to complete in milliseconds [31, 36, 42, 64]. Such a strict latency conflicts with the inherently high cost of exact search algorithm [28], which forces end-users to settle on approximate query results on high-dimensional vectors. With emerging new vector search applications, queries on vectors become increasingly more complex, which often involve hybrid search on both scalar and vector data (§2.1). This naturally motivates an integration of vector search systems and relational databases.

Vector search and database systems differ in their ways of using the index, a critical structure to speed up queries. An important property of conventional indices like B-tree [22] is *monotonicity*. This property ensures that a query can traverse the data-set guided by an index monotonically along a certain direction. This often avoids total data scan, therefore enables efficient query execution. However, it is prohibitively expensive for high-dimensional vector indices [5, 25, 43, 55, 57, 85] to preserve monotonicity, because of the curse of dimensionality [28]. Instead, they are often organized as a graph or cluster-based irregular structure, which follows monotonicity *approximately*. Traversing such vector indices does not guarantee a strict monotonic order in terms of distances to a target vector, but it enables a system to efficiently determine when it is *unlikely* for new traversals to reach closer vectors to a target than the current K ones. Therefore, modern vector indices only support approximate `TopK`, i.e., to find K nearest neighbors approximately. A `TopK` query traverses a vector index for a sufficiently large number of steps, until it determines that a neighbor closer than the current K nearest ones is unlikely to be found.

To integrate vector search and database systems , existing vector database systems [76, 80, 86] choose to conform with *strict* monotonicity. To support similarity queries other than `TopK`, they first leverage `TopK` to collect K vectors, and sort the K vectors according to distances to a target vector, which sets up a *temporary index preserving monotonicity*. Complex relational operators can therefore execute on the temporary index in the traditional way. Consider the following vector search query, "find X number of products most similar to an

*Corresponding author.

image but under a certain price". A database planner would first run a vector search operator on the vector attribute of image embedding to find K nearest tuples, then apply a filter operator on the price attribute. But it is impossible to predict exactly how many tuples will pass the filter operator, which could be much less than K. Therefore this practice has the inherent difficulty of identifying the optimal K that produces exact X results. As a result, it resorts to either a setting of a conservatively large K or a trial-and-error of many Ks, which both lead to suboptimal query performance.

In this paper, we present VBASE, a new system capable of efficiently serving complex online queries that involve both approximate similarity search and relational operators on scalar and vector data-sets. VBASE identifies *Relaxed Monotonicity* as the common property abstracted from the two seemingly different systems: vector search systems and relational databases. *Relaxed monotonicity* requires index traversals to only follows monotonicity *approximately*. We observe that state-of-the-art vector indices all follow relaxed monotonicity property in a two-phase pattern: an index traversal first locates the nearest region to a target vector approximately, and then moves away from the target region progressively in an approximate way. Based on the observation, we formally define *Relaxed Monotonicity* property, which abstracts the core index traversal pattern that most existing vector indices already have (§3.1). *Relaxed monotonicity* can be viewed as a generalized form of monotonicity, thus it is also applicable to conventional scalar indices, such as B-trees. Therefore, *Relaxed Monotonicity* can serve as the common foundation of vector search and database systems.

With relaxed monotonicity, VBASE builds a unified query execution engine to support a wide range of queries both on scalar and vector data, including queries across multiple heterogeneous indices. VBASE's unified engine is based on a `Next` interface, instead of `TopK`, to support traversal in both vector and scalar indices. Meanwhile, the engine allows the derivation of a generalized termination condition from relaxed monotonicity to stop a query's execution timely.

A unique characteristic of VBASE is that its relaxed-monotonicity-based query execution engine can *provably* achieve *equivalent* query results to those produced by `TopK`-only solutions of the *optimal* $\widetilde{K}$ (§3.3) This powerful property allows VBASE to circumvent the constraints of a `TopK`-only interface to achieve significantly higher efficiency, while preserving the semantics of `TopK`-based queries. In particular, based on the derived generalized termination condition, VBASE is able to detect the $\widetilde{K}$ during index traversal without an prediction of $\widetilde{K}$. This allows VBASE to achieve similar performance to the well-optimized `TopK` vector search. For more complex queries than `TopK` searches, VBASE can achieve up to three order-of-magnitude lower average and tail query latency over state-of-the-art systems under similar result accuracy (i.e., same or even better recalls).

Moreover, with relaxed monotonicity, VBASE can even support approximate query types that previous systems do not, and show superior query performance and accuracy. For example, VBASE can finish a `join`-based vector query in 16 seconds with 99.9%+ recall rate, which is 7000× faster than a brute-force table scan.

Table 1: Online Similarity Query Support for Vectors

	S1	S2	S3	S4
ANN systems [25,43,46]	✓	✗	✗	✗
AnalyticDB-V [80]	✓	✓	✗*	✗*
PASE [86]	✓	✓	✗*	✗*
PostgreSQL [12]	✗*	✗*	✗*	✗*
Milvus [76]	✓	✓	✓	✗
Elasticsearch [4]	✓	✓	✗◇	✗

∗: Some systems can support these queries through exhaustive linear scan, but this cannot meet the requirements of online services.
◇: Only support one inverted index and one vector index.

In summary, we make the following contributions:

1. VBASE identifies and defines formally "*Relaxed Monotonicity*", a property that reveals, for the first time, the core of well-designed vector indices and why they work effectively in practice.
2. VBASE builds a *Unified Database Engine* based on *relaxed monotonicity*, which enables powerful complex queries leveraging both vector and scalar data indices.
3. We prove that VBASE's unified engine produces equivalent results to `TopK`-only methods using vector indices, with a much more efficient execution plan than that of `TopK`-only methods.
4. We implement VBASE based on PostgreSQL with 2000 lines of additional code, and show an end-to-end evaluation of eight complex SQL queries on a hybrid one million recipe data-sets [59] with both vector and scalar attributes.

We plan to make VBASE open-source to satisfy the emerging important vector analytic applications in the era of AI.

2 Background

2.1 Emerging Online Vector Queries

Vector has become a key form of data representation in the AI era. Deep learning has enabled a growing number of vector-centric online applications, including embedding-based retrieval [25,87], face recognition [69], code retrieval [37], question answering [52,63], Google Multisearch [7], Facebook near-exact duplicates detection [6], etc. More recently, AI applications have leveraged ChatGPT's retrieval plugin [10] to convert their proprietary knowledge, personal documents, and chat contexts into vectors. This enables the retrieval of relevant vectors with price, category, location, or time constraints to construct prompts in the chat.

Traditional applications also benefit from vectors empowered by AI. For example, search engine turns web documents into both bag-of-words sparse vectors and deep learning em-

bedding dense vectors to improve the relevance of search results. And recommendation systems turn images, videos, and descriptions of items into different vectors. Combined with scalar data like item price and category, these vectors are used to enhance recommendation experiences.

All these call for a general system to run sophisticated vector and scalar queries efficiently. In summary, these vector scenarios can be categorized into the following types.

S1: Single-vector `TopK`. embedding-based retrieval [25], recommendation [87], and question answering [52, 63] essentially search a vector data-set for the K closest vectors, given a query vector. Such queries can be naturally expressed by a `TopK` operator on a single-vector column.

S2: Single-vector `TopK` plus scalar attribute filtering. There are also requirements to find `TopK` results under certain scalar attribute constraints. Google Multisearch [7] belongs to this category. It allows users to provide additional text hints during a similarity image search.

S3: Multi-column `TopK`. Some vector analytics require intersecting results of multiple `TopK` searches over different vector attributes. For example, image-recipe retrieval [68] is a recipe search on multi-modal data attributes of both (vectorized) ingredient keywords and a sample dish image. Recent work [70, 83] shows that multi-column `TopK` search can boost result quality in applications such as question-answering.

S4: Vector similarity filter. Similarity filtering is a typical vector analytics scenario. For example, face recognition [69] and Facebook's near-exact duplicates detection [6] search for similar images (given an image) from a data-set with a similarity threshold. To support such applications, one could use vector filtering based on distance similarity between two images, i.e., distance-based range query.

All these vector query types have a strict latency requirement (e.g. milliseconds). Unfortunately, no existing systems can support all these online similarity queries comprehensively and efficiently (see Table 1).

2.2 The Division Between Databases and Vector Search Systems

Although databases can express the above queries through relational algebra, the division in the semantics between vector and conventional database indices makes it difficult to provide a unified system that efficiently runs various types of sophisticated online vector queries as shown in Table 1.

Relational database. A relational database is one of the most prominent tools to run sophisticated queries [16, 24, 29, 40]. In order to meet the low-latency "online" requirement, indices are widely adopted by databases to expedite query executions, such as B-tree [22], B+-tree [75] and more. These indices demonstrate *monotonicity*, a property that allows a query to traverse an index monotonically along a certain direction, e.g., in a descending or ascending order.

One of the most important types of online queries in the context of emerging vector scenarios is `TopK` query (§2.1). And a conventional database index can speed up `TopK` by traversing the index in the ascending or descending order and terminating the query as soon as it collects K results. This optimization applies to many `TopK` variants, such as `TopK` + filtering, and multiple-column `TopK` queries [33].

However, the effectiveness of such optimization relies on the assumption of monotonicity, which high-dimensional vector indices do not follow. We elaborate next.

Approximate vector search. The recent eruption of AI models has been generating a large and growing amount of high-dimensional vector data. For better learning representation, a vector can have hundreds of dimensions [60, 66, 88]. Due to the curse of dimensionality [28], no solutions can complete a high-dimensional vector query in sub-linear time. To address "online" scenarios, modern vector search systems resort to approximation to lower query latency dramatically (milliseconds) while maintaining a relatively high result accuracy (90%+ recall). These systems are often referred as *approximate nearest neighbor search* (ANNS) systems [5, 25, 43, 55, 57, 85].

Like relational databases, vector indices are adopted to facilitate approximate vector search. Representative vector indices are either organized as *partitions* (clustering-based [5, 17, 19, 25, 44, 45, 48, 90], hash-based [30, 41, 79, 81, 85]), high-dimensional tree-based [23, 57, 62, 78]), or *neighborhood graphs* [32, 39, 43, 55, 58, 77]. The difficulty of locating a vector in the high-dimensional space forces these vector indices to optimize for approximate `TopK`. In a `TopK` query, index traversal is guided by a query vector q approximately towards the nearest neighbors tortuously based on the distance between q and some anchor points (e.g. cluster centroids, or sampled graph vertices). During the traversal, the direction to q may change dramatically, thus the process does not guarantee to approach q in every traversal step and the vector index traversal is not monotonic.

The lack of monotonicity in vector indices bars database systems from directly using vector indices to expedite queries, which is the primary source of *the division between databases and vector search systems*.

`TopK`-based solutions to eliminate the division. Because vector indices are optimized for `TopK`, ANNS systems expose only a `TopK` interface. To close the monotonicity gap between databases and vector search systems, the current practice is to use ANNS `TopK` interface to create tentative indices based on K vectors sorted according to the distance to the target vector. Such tentative indices preserve monotonicity, which enables fast vector query processing in databases [76, 80, 86].

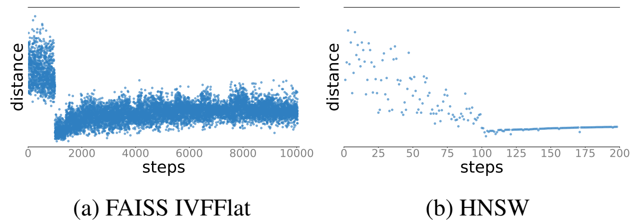
However, `TopK`-based tentative index solutions are unsatisfactory. It is difficult, if not impossible, to predictthe right size of $\widetilde{K}$ for the tentative index for queries, where a subsequent relational operator with a filter constraint can collect just the right number of results. This limitation universally applies

to TopK + filter queries, vector similarity filter queries, and more. Thus TopK-based tentative indices inevitably lead to choosing a conservatively very large K [80, 86] or performing trial-and-error with different sizes of K [76], both incurring excessive data accesses and computations.

3 VBASE Design

3.1 Relaxed Monotonicity

Unlike conventional scalar indices, high-dimensional vector indices are designed for approximate TopK and do not follow monotonicity. Figure 1 shows the TopK traversal patterns of two popular vector indices, FAISS IVVFlat [5] and HNSW [89] . As illustrated, vector index traversal does not comply with monotonicity, the distance towards the target vector oscillates *unpredictably* as the traversal progresses. A lack of monotonicity in these vector indices bars relational databases from directly using them to expedite queries (§2.2).



(a) FAISS IVFFlat (b) HNSW

Figure 1: Traversal patterns of two vector indices.

The Two-phase Vector Index Traversal Pattern. Nevertheless, Figure 1 reveals a two-phase index traversal pattern for both vector indices. In the first phase, the index traversal approaches the target vector region approximately in spite of large oscillations in vector distances. In the second phase, the index traversal stabilizes and steadily departs from the target vector region in an approximate way.

This two-phase traversal pattern is common in most vector indices we examine. We believe that the essence of well-designed vector indices is an effective data-structure that embodies this traversal pattern implicitly. Thus a TopK search query could terminate early when it enters the second phase as further traversals are unlikely to identify more similar vectors.

The Formal Definition of Relaxed Monotonicity. Based on the two-phase traversal pattern, we can formally define *Relaxed Monotonicity* which identifies if a vector index traversal has entered the second phase. The definition is built upon the intuition of how a vector TopK search is executed internally.

Figure 2 shows such an intuition by illustrating the process of a general vector search for a query vector q. The dashed arrow in the figure shows an index traversal path with respect to q. Following the two-phase pattern, the query first approaches the *neighborhood* of q gradually. In a high-dimensional space, the neighborhood of q is defined by a neighbor sphere centered around q, illustrated as a circle in Figure 2. Afterward,

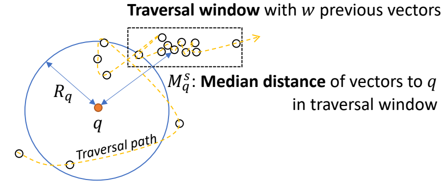


Figure 2: An Illustration of *Relaxed Monotonicity*'s intuition. A vector query q discovers q's neighborhood with E nearest vectors along the progression of a traversal path.

the index traversal leaves the sphere and enters phase two, where the query can terminate in this phase.

Figure 2 suggests that, to determine whether it enters phase two, a query needs to understand R_q, the radius of the neighbor sphere centered around q, and whether M_q^s, the distance between the query's current index traversal position (denoted as traversal step s) and q, is greater than R_q, i.e., it is traversing beyond the neighborhood of q.

Formally, R_q, the radius of q's neighborhood, is defined as:

$$R_q = Max(TopE(\{Distance(q, v_j) | j \in [1, s-1]\})), \quad (1)$$

where $TopE$ denotes the E nearest neighbors of q observed during the traversal, supposing that the traversal has reached step s so far. For a K nearest vector search query, it requires $E \geq K$ in order to produce sufficient final results. In Figure 2, the E vectors within the circle are the nearest neighbors of q of all the s vectors visited so far. During an index traversal, the sphere's radius R_q would gradually decrease during phase 1, and becomes stable during phase 2.

Given the definition of R_q, the system needs to define M_q^s, the distance measurement between the target vector q and the current index traversal position, denoted as traversal step s. M_q^s is then used to determine whether the traversal enters phase 2, i.e., leaving the neighbor sphere.

Mathematically, M_q^s is defined as the median distance to q of all vectors traversed in the most recent w steps, i.e., the traversal window.

$$M_q^s = Median(\{Distance(q, v_i) | i \in [s-w+1, s]\}), \quad (2)$$

where $Distance(q, v_i)$ denotes the distance between q and vector v_i, traversed in the past traversal window. Note that we use *median* instead of *mean* to disregard any outlier vectors in the traversal window, which has exceedingly large or small distances to q than others. For example, the two outlier vectors in the leftmost and rightmost positions in the traversal window shown in Figure 2.

Taking Eq.1 and Eq.2 together, we define *Relaxed Monotonicity* as:

Definition 1 Relaxed Monotonicity

$$\exists s, M_q^t \geq R_q, \forall t \geq s. \quad (3)$$

In other words, Def. 1 determines that a vector index follows *relaxed monotonicity* if there exists a certain index traversal step s, where all traversal steps t after step s transcends into a region (M_s^t as the region's distance to q) that is outside q's neighborhood sphere, defined by q's E nearest neighbors.

Importance of Relaxed Monotonicity. Relaxed monotonicity is the key for the database to circumvent the inefficient constraint of `TopK`-only interfaces, and to generate an efficient execution with on-the-fly early termination. When subsequent database operators following the vector index scan can determine that vector index traversal has entered the second phase, one would know that we are veering away from the target vector steadily. In such cases, we could early terminate the query if sufficient results have been collected, because new tuples with closer vector attributes are unlikely to be found.

Generality of Relaxed Monotonicity. All mainstream vector indices listed in ANN Benchmarks [2] perform vector search using four general components: 1) *index traversal* to navigate the vector data-set; 2) *termination check* to detect query termination signal; 3) *monotonicity check* to determine if a query enters Phase 2; and 4) *priority queue* for keeping K nearest vectors so far. Often in ANNS indices, *monotonicity check* is a necessary condition for the *termination check*.

Although vector indices implement these four components in different ways, their monotonicity check satisfies Def. 1. For example, Figure 1 shows that index traversal patterns of IVFFlat and HNSW follow *relaxed monotonicity* obviously. And Def. 1's parameters, i.e., the traversal window w and the neighborhood of q R_q, are able to capture the internal characteristics of index traversal patterns of these popular vector indices as well as conventional indices. Next, we elaborate on the setting of these parameters for representative indices.

- **Graph-based Vector Indices**, such as HNSW [89], follow the two-phase pattern using graph data-structures. In the first phase in Figure 1b, HNSW quickly navigates the traversal to the neighborhood of q through hierarchical coarse-grained to fine-grained navigating graphs. When it reaches the fine-grained graph, the traversal enters the second phase where it has found the neighborhood of q and departs away. Vector search using a graph-based index use best-first (BF) graph traversal from a fixed starting point. BF search maintains a *sorted candidate queue* with the size ef. This queue essentially represents the neighborhood sphere in Eq. 1 with ef vectors. Therefore E equals ef for HNSW. BF traversal explores the graph through the vectors in the candidate queue and expands the exploration by visiting their neighbors. If a neighbor is *unvisited* and its distance to the target vector q is smaller than the farthest vector in the sorted queue (i.e. R_q), the traversal replaces the farthest vector with the new one and resorts the queue. Because the traversal only compares the traversed vector itself with R_q, the traversal window w in Eq. 2 equals one.
- **Partition-based Vector Indices**, such as FAISS IVFFlat [5] and SPANN [25], divide vectors into multiple clusters where nearby vectors are conglomerated. During the traversal in the first phase in Figure 1a, IVFFlat traverses over the centroids to identify the m closest clusters, and then in the second phase it goes over the vectors in m clusters and terminates when all vectors in m clusters are traversed, which indicates Eq. 3 of *Relaxed Monotonicity* has been satisfied after the vector search visits m clusters. With this observation, E in Eq. 1 is set to K of the `TopK` query, and the traversal window w in Eq. 2 is set to the number of total vectors in m clusters.
- **Scalar Indices**, such as B-Tree, follow strict monotonicity. It is a special case of relaxed monotonicity, where w and E are both set to 1 and Eq.3 is always true.

Note that it is our observation that well-designed indices should satisfy *Relaxed Monotonicity*. VBASE abstracts and formalizes this property that most indices already preserve, and encapsulates it with mathematical terms such as E and w in Eq. 1 and 2. It is the responsibility of individual indices to guarantee relaxed monotonicity by tuning the corresponding hyperparameters like ef and m, which can be transformed to E and w, as discussed above.

3.2 Unified Query Execution Engine

With relaxed monotonicity, VBASE builds a unified query execution engine modeled after a traditional database engine with minimum changes. VBASE's unified engine is built on Volcano Model (i.e. Iterator Model) [35], where 1) a relational operator in a given query produces a stream of tuples iteratively that are consumed by downstream operators, and 2) the iterative execution stops if a termination condition is met.

Iterative Execution Model. VBASE fully complies with the Volcano Model. It reuses the traditional `Open`, `Next`, and `Close` interfaces to leverage index traversal so that no change is required for conventional indices. For vector indices that traditionally expose `TopK` interfaces only, VBASE performs a simple adaptation to expose their internal index traversal process, to conform to VBASE's `Next` interface. We will discuss the details of implementing `Next` for vector indices in §4.2.

Generalized Termination Condition Check. VBASE modifies the termination condition based on relaxed monotonicity. In particular, VBASE extends the original termination condition with *relaxed monotonicity check*. In addition to the original query termination condition, VBASE performs relaxed monotonicity check by inspecting Eq. 3. Because VBASE requires a vector index guarantees *Relaxed Monotonicity*, an inspection of Eq. 3 could be reduced to $M_q^s > R_q$, where *relaxed monotonicity* checks beyond step s are all assumed to be true.

The query execution would only stop if both the original termination condition and *relaxed monotonicity* check were passed. Please note for the traditional index, the *relaxed monotonicity* check is always true, which reduces to the termination check of the convention iterative model.

Next, we describe how VBASE's termination conditions work for TopK and distance-based ranger filter, two operators used by vector queries.

- **OrderBy with limit:** In traditional databases, TopK is usually expressed by an OrderBy operator with limit K. The traditional TopK query terminates immediately once K results have been collected, because all indexed tuples are ordered. For an approximate TopK query on vectors, VBASE need to check if the relaxed monotonicity check (i.e., reduced Eq.3) is passed, in addition to collecting K vectors. When *relaxed monotonicity* check is true, the index traversal has entered phase 2 (§3.1), which indicates it is veering away from the target vector steadily. And we can terminate the query after collecting K nearest vectors, because it is unlikely to find new vectors closer than the collected ones.
- **Range filter:** Distance-based range filter returns tuples whose values or distances to a target are within a range R. For a conventional query, the query stops when a tuple being traversed goes beyond range R, because monotonicity guarantees we have visited all tuples within R. For a vector query, the execution only stops if both a vector along the traversal path exceeds distance R and the relaxed monotonicity check is passed, which indicates we are in a stable phase (phase two) moving away from a target vector.

Advantages of a Unified Engine. The unified engine enables VBASE to preserve full compatibility with traditional databases and supports all the vector query types discussed in §2.1. It also creates new optimization opportunities for vector queries. For example, instead of filtering after TopK, VBASE can perform filtering during index traversal with flexible termination conditions (for TopK or range query). This optimization is one of the key reasons VBASE outperforms TopK-based solutions (§5.3). Moreover, the unified engine allows the incorporation of a refined NRA algorithm [33], which can significantly improve the performance of multi-column vector query (§4.4).

Interestingly, VBASE's unified engine also supports vector Join. Although not an online query, vector Join is useful in scenarios such as document auto-tagging [26, 72], where a small labeled (tagged) document set is used to identify a tag for each document in a large unlabeled document set by finding the document pair with the closest document embeddings (vectors). This can be thought of as running a Join operator on a document table and a label table with a distance-based match, which can be achieved in VBASE by an index join based on a range filter. Because such a Join can only be achieved by a full table scan in existing solutions, VBASE can outperform the baseline by over 7000× (§5.3).

3.3 Result Equivalence

In this section we demonstrate a powerful property of VBASE's unified query execution engine based on *Relaxed Monotonicity*, that it produces the equivalent results as a TopK method based on a tentative monotonicity-preserving index with the optimal $\widetilde{K}$. The optimal $\widetilde{K}$ is the minimal K' for the index traversal to satisfy K results in a TopK query. As K' is the minimal satisfactory value, the query latency is minimized. We rely on the quality of individual indices to ensure recalls.

Next, we formally prove the result equivalence for TopK+filter and range filer [20] queries, which are major types of similarity searches supported by existing TopK-based vector systems (Table 1). The proof also reveals the reason of VBASE's superior performance.

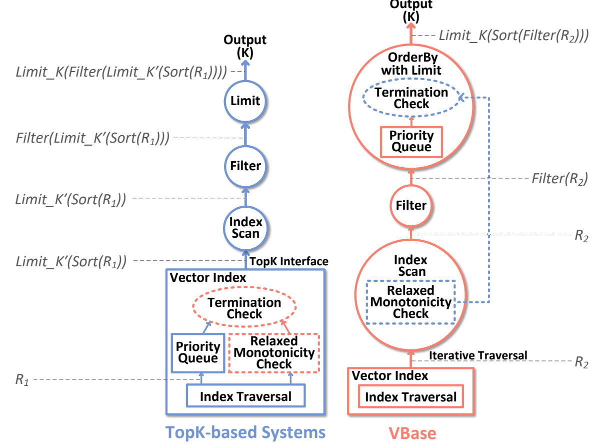


Figure 3: Result Equivalence

TopK +filter. To find K vectors matching the filter, a TopK-based system first collects K' vectors by calling TopK (K') and sorts the collected K' vectors. To achieve this, the system needs to traverse R_1 vectors through the underlying vector index. The query then runs on the sorted K' vectors by applying the filter and the K limit operators, producing the final results, denoted as r_1. Eq.4 formulates the above process, illustrated on the left in Figure 3.

$$r_1 = Limit_K(Filter(Limit_K'(Sort(R_1)))). \tag{4}$$

We define $filter_selectivity$ as the ratio of the output set on the input set of the filter operator. Eq. 4 can then be transformed to:

$$\begin{aligned} r_1 =& Limit_K''(Filter(Sort(R_1))), \\ & where\ K'' = min(K, K' \times filter_selectivity). \end{aligned} \tag{5}$$

Assuming the TopK-based system can predict the optimal $\widetilde{K}$, i.e., $K' = \widetilde{K} = K/filter_selectivity$, execution will get exactly K results, and Eq. 5 reduces to Eq. 6

$$r_1 = Limit_K(Filter(Sort(R_1))). \tag{6}$$

In comparison, VBASE traverses R_2 vectors via the same vector index as the TopK-based system, and gets results r_2. Eq. 7 formulates this process, shown on the right of Figure 3.

$$r_2 = Limit_K(Sort(Filter(R_2))). \tag{7}$$

We show in §3.2 and §4 that TopK-based systems and VBASE use the same vector index, follow the same index traversal algorithm, and are based on the same relaxed monotonicity check to terminate queries. Therefore both systems traverse the exact same set of tuples, i.e., $R_1 = R_2$. [1]. Because $R_1 = R_2$ and $Filter$ and $Sort$ are commutative, from Eq. 6 and 7 we conclude $r_1 = r_2$. **Q.E.D.**

It is difficult for a TopK-based solution to predict K' to get both correct results and high query efficiency. If $K' \times filter_selectivity < K$ in Eq.5, the system cannot get enough results, leading to poor accuracy. If $K' \times filter_selectivity > K$ in Eq. 5, the result is accurate at the expense of splurging extra index traversal. Some TopK-based system [76] performs trial-and-error with many values of K' until $K' \times filter_selectivity \geq K$, which results in excessive duplicated data access and processing. In contrast, VBASE determines $\widetilde{K} \times filter_selectivity = K$ on-the-fly, therefore achieving both high query accuracy and performance.

Range Filter Query. A range filter query based on TopK can be formulated as

$$r_1 = Filter(Limit_K'(Sort(R_1))), \tag{8}$$

Where R_1 stands for the traversed vectors by the underlying TopK primitive. Similar to Eq.4 vs. Eq.5, Eq. 8 can be transformed to

$$\begin{aligned} r_1 =& Limit_K''(Filter((Sort(R_1)))), \\ & where\ K'' = K' \times filter_selectivity. \end{aligned} \tag{9}$$

Under the assumption of optimal $\widetilde{K}$, assuming the query produces T vectors, we have $\widetilde{K} = K' = T/filter_selectivity$. Based on $\widetilde{K}$, Therefore,

$$r_1 = Limit_T(Filter(Sort(R_1))) = Filter(Sort(R_1)). \tag{10}$$

In comparison, VBASE traverses R_2 vectors via the same index and gets results r_2, which can be formulated as

$$r_2 = Filter(R_2). \tag{11}$$

Since the traversal algorithm and the termination condition are exactly the same as the TopK-based solution, both systems visit the same set of vectors, i.e., $R_1 = R_2$. As the filter conditions are not sensitive to order, $r_1 = r_2$. **Q.E.D.**

4 VBASE Implementation

4.1 Relaxed Monotonicity Check

We implement a common relaxed monotonicity check for all vector indices based on Definition 1 in §3.1. Specifically, we implement two queues to track the current traversal state: 1) a priority queue with size E called smallestQueue, to keep the visited nearest neighbors of a target vector q during the traversal; 2) recentQueue of size w to track the most recent traversal window. When a new vector v is visited via index traversal, smallestQueue and recentQueue are updated accordingly. And the relaxed monotonicity check is performed by calculating the current traversal state according to Eq.(3) based on vectors in smallestQueue and recentQueue.

Note that E and w are sensitive to data distribution and specific indexing algorithms. Increasing them tends to improve query accuracy at the expense of longer latency. In practice, they can be tuned to trade off query accuracy and latency.

4.2 Query Execution Engine

VBASE's unified query engine is implemented based on PostgreSQL, with minor extensions to modules regarding index traversal and termination conditions.

Vector index integration. Existing high-dimensional vector indices only expose TopK interface and keep the index traversal and relaxed monotonicity check internally to the system. VBASE re-architects the vector indices systems by exposing the internal index traversal algorithms with Open, Next, and Close interfaces, which can then be integrated into Volcano Model seamlessly.

VBASE has incorporated several state-of-the-art vector indices, including HNSW [89], IVFFlat [5] and SPANN [25], where SPANN is shown effective for billion-scale vector datasets. Next, we introduce the integration of HNSW and IVFFlat, a graph-based index and a partition-based index, respectively. Other algorithms can be integrated in a similar way.

HNSW [89] is a graph-based vector index consisting of hierarchical neighborhood graphs where the upper-layer graph keeps coarse-grained samples of the lower-layer graph. A query traverses the graphs from upper-layer to lower-layer following the best-first manner. The approximate nearest point found in the upper-layer graph is the entry point of the lower-layer graph. In VBASE, we remove the implementation regarding TopK, e.g., a priority queue to record the top k results, and only keep states necessary to carry on the index traversal algorithm. The relevant states include a bitmap to record previously visited vectors, the current vector being visited, and the candidate vectors to be visited next. These states will be kept during the query life cycle. To initiate a query on a vector index, VBASE calls Open to search on high-layer graphs. During query execution, each call to Next will return the current closest unvisited node, records it, and expands its neighbors into candidate vectors in the state. The state will be cleared in Close function. Overall, we modify less than 200 lines of code to integrate HNSW.

IVFFlat [5] is a partition-based index, which clusters vectors into lists and chooses the centroid as the representative of each list. In the Open interface used to initiate index traversal, VBASE sorts all the lists from near to far based on the dis-

[1] We assume vector index traversal is deterministic, true for most vector search systems.

tance between the target vector and the centroids. Upon calls to `Next`, the vectors in the corresponding nearest lists are read one by one. The query execution state in a partition-based index includes sorted lists and the current read position, which will be destroyed by a `Close`.

Index scan operator. We add a new "vector index" type using the index extension interface in PostgreSQL [12] to implement index scan. It forwards function calls to `Next` to the underlying vector index within the iterative interface. We use *array* to store high-dimensional vectors in the table and record their tuple addresses in the table as the vectors' metadata in the index. Once a vector is read from the underlying vector index, its metadata will also be returned so that VBASE can find the corresponding tuple in the main table.

Note that the relaxed monotonicity check described in §4.1 is implemented in the index scan operator so that multiple indices do not need to duplicate the implementation.

`OrderBy` with limit. VBASE implement `TopK` using `OrderBy` with limit plus an index scan operator. The system uses a priority queue to keep the candidate results. The `TopK` query terminates once the index traversal passes the relaxed monotonicity check in the upstream index scan operator and K vectors have been filled in the priority queue.

Note that vector indices are used for similarity queries, i.e. search closest vectors to the target vector. If a user would like to query the farthest `TopK` results from the target vector, the distance calculation method needs to be reversed before creating the indices.

Range filter and `Join`. VBASE implements an efficient range filter by concatenating it with an index scan operator. Only vectors passing the distance filter condition can be returned to the subsequent operators. The index traversal stops when the distance between the current vector to the target vector is larger than the filter constraint and the relaxed monotonicity check is passed in the index scan operator.

With the support of distance filter, VBASE can even support `Join` on high-dimensional vectors, which previous vector systems cannot support efficiently. Semantic-based join has been widely used in document auto-tagging [26, 72], which assigns one or multiple labels for each unseen document by finding the closest label embeddings to a document embedding. Previous systems can only support `Join` by brute-force table scan. VBASE executes a `Join` by nested-loop with index search, which outperforms existing systems by 7000× faster with 0.999 recall accuracy in our experiment.

More complex queries. The combination of the above operators can be used to support more complex queries efficiently.

4.3 Query Planning

Complex queries often require effective cost estimation on various query plans. In general, it includes vector algebra computation (e.g., distance calculation), selectivity estimation in case the query contains filters, and index scan cost.

Vector computation. Traditional databases estimate the cost of scalar data computation using a constant value t, e.g., $t = 0.0025$. But vector computation is more expensive, it involves the calculation of the distance between vectors, which is proportional to the number of dimensions. Thus VBASE models the cost of vector computation t_v as:

$$t_v(dim) = t \cdot c \cdot dim,$$

where t is a predefined value representing the cost of scalar operation, c is the coefficient related to SIMD optimizations for vector computation, and dim denotes vector dimension.

Selectivity estimation. If a query contains a filter, the query should estimate selectivity, the ratio of tuples that will pass the filter. VBASE relies on sampling-based methods to measure the distribution of high-dimensional vectors [67, 82]. Specifically, VBASE uniformly samples vector data at a ratio and stores the sampled vectors in the metadata of the database. Given a query q, it applies the filter on the sampled data to estimate the selectivity Sel on the full vector data-set.

$$Sel_{sample}(q) \approx Sel_{full_data}(q).$$

In our experiments, setting the sample rate to 0.001 can produce a good estimation with q-error < 1.1 in most cases while incurring only a tiny extra latency (<1ms). More details will be presented in §5.5.

Index scan cost estimation. This includes start-up cost and traversal cost. The start-up cost is the cost to locate the region nearby the target vector before returning vector data; Traversal cost represents the cost to iterate over the matched tuples through the index. For each index traversal step, the cost C_{step} includes t_{IO}, the IO cost to fetch the index data from disk, and $t_v(dim)$, the cost to calculate the distance: $C_{step} = t_v(dim) + t_{IO}$. For partition-based indices like IVFFlat and SPANN, the index scan cost C_p is:

$$C_p = N_c \times C_{step} + max(\lceil Sel(q)N/N_p \rceil, m) \times N_p \times C_{step},$$

where N is the table size, N_c is the number of centroids, N_p is the average number of data per partition, and m is the number of partitions the index traversal algorithm requires to traverse for *relaxed monotonicity* check.

The scan cost, C_g, of graph-based indices like HNSW is:

$$C_g = N_{start} \times C_{step} + max(Sel(q)N, N_E) \times R_{iter} \times C_{step},$$

where N_{start} is the number of steps to traverse upper-layer graphs in `Open` function of HNSW, N_E is the number of steps to satisfy *relaxed monotonicity* check, and R_{iter} is the average times of distance function called per step to reach next point. N_{start}, N_E and R_{iter} are dependent on the hyper-parameters of the index and the distribution of data, which can be automatically estimated by sampling, and this process can be embedded into databases' *Analyze* routine.

4.4 Multi-Column Scan Optimization

To support multi-column vector queries, `TopK`-based systems can only perform multi-column scan based on the multiple sets of sorted vectors collected by `TopK`. For example, Milvus performs NRA algorithm [33] for multi-column scan based on `TopK`. It doubles K and re-executes the query if the previous results are insufficient. Every attempt to a larger K is an independent traversal over the underlying vector index. This introduces excessive vector access and computation.

In contrast, VBASE implements NRA algorithm [33] natively based on the index scan operator, thus avoiding the repetitive execution of NRA. The NRA algorithm traverses each vector index in a round-robin manner. We observe that round-robin might not be an efficient choice. In the vector search scenario, different vector indices can return results of different quality, especially when the ranking function is summation with unequal weights from different indices. Figure 4 shows the results of such a case, where the round-robin method will unnecessarily traverse excessive low-quality vectors (i.e., dots in Figure 4).

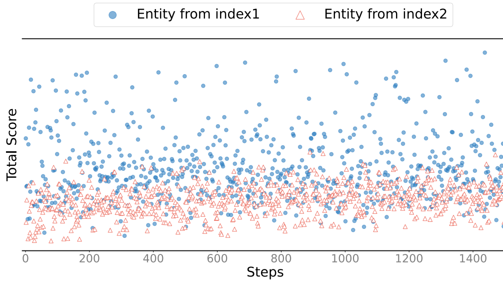


Figure 4: Index traversal pattern for a 2-index vector query on Recipe1M. The total score is a summation of two vectors' distance with a weight ratio of 1:2. Lower score means closer to the target vector. Blue dots and red triangles represent the total scores of entities using index1 and index2.

This observation leads us to a new index scan algorithm that scans through high-quality indices more frequently, i.e., triangles in Figure 4, so that the query can terminate earlier with even more accurate results. Such a non-uniform traversal manner may trap in local optima. In Figure 4, a greedy algorithm may prefer to visit index2 only. But the figure shows that index1 does have good quality vectors occasionally.

To balance exploration and exploitation, we use both local and global information to guide the index traversal (See Figure 5). Our approach divides the traversal process into several rounds and adds a traversal decision module. It maintains a local priority queue to store the last round's results. This local information helps us identify which index is more likely to return better results so we can visit it more in the next round. To avoid being trapped in local optima, the decision module also stores the average score of all traversed entities for each index (i.e., avg_i for $index_i$) and updates them each round. Based on this global information, We additionally traverse each index $W_i = \left\lceil n_2 \times \frac{1/avg_i}{\sum_{j=1}^{m} 1/avg_j} \right\rceil$ times in this round where n_2 is a hyper-parameter. Therefore, the high-quality index (with low avg_i) will be traversed more times while we can still ensure traversing the low-quality index (with high avg_i) at least once in each round. Table 7 in §5.3 highlights the benefit of this approach.

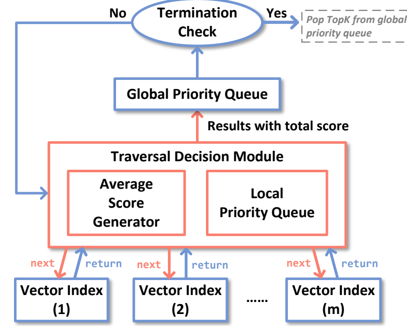


Figure 5: Overview of multi-column traversal optimization, assuming there are m indices.

5 VBASE Evaluation

In this section, we evaluate VBASE in comparison with other state-of-the-art vector search systems and vector-enabled databases based on `TopK`, and demonstrate VBASE has superior performance and accuracy on vector similarity queries.

5.1 Evaluation Benchmark

A lack of a comprehensive relational benchmark for complex vector applications necessitates us to create a vector benchmark to compare VBASE with various vector-similarity-enabled systems. The use of approximate vector processing also needs the benchmark to define new evaluation metrics.

Vector-scalar relational data-set. Because current vector search [2,3] and database benchmarks [13,14] have either vector or scalar data-sets *but not both*, we extend Recipe1M [68] to generate vector and scalar hybrid data-sets. Recipe1M data-set is a collection of more than 1 million recipes, each containing ingredients, cooking instructions, and a set of images of the finished dish.

Our evaluation data-set is organized as two tables: *Recipe Table* and *Tag Table*. Their schemas are shown in Table 2 and 3, respectively.

Table 2: Schema of Recipe Table

Column Name	Data Type	Example
recipe_id	identifier	1
images	list of strings	["data/images/1/0.jpg" , $\cdots$]
description	text	[ingredients] + [instruction]
images_embedding	vector	$[0.0421, 0.0296, \cdots, 0.0273]$
description_embedding	vector	$[0.0056, 0.0487, \cdots, 0.0034]$
popularity	integer	300

Table 3: Schema of Tag Table

Column Name	Data Type	Example
id	identifier	1
tag_name	text	"salad"
tag_vector	vector	$[0.0137, 0.0421, \cdots, 0.0183]$

Recipe Table stores 330,922 recipes from Recipe1M[2]. As shown in Table 2, *Recipe Table* inherits *recipe_id* and recipe *images* URIs from Recipe1M as two attributes. We merge Recipe1M's ingredients and instructions as a single string attribute called *description*.

In addition to the original data from *Recipe1M*, *Recipe Table* has two *vector* attributes, *images_embedding* and *description_embedding*. They are two 1,024-dimensional vector embeddings of recipe images and descriptions based on the cross-modal embedding model from [68]. We also extend the recipe item with an additional scalar attribute: *popularity*, a random integer in the range [0,10000].

Tag Table samples 10,000 recipes from Recipe1M, and assigns of *tags* for them. As shown in Table 3, *Tag Table* has scalar attributes of *id* and *tag_name*. Attribute *tag_name* is a set of strings of manually assigned tags (e.g. dessert, main course, salad, pizza, etc), and *id* is a unique integer assigned to the tag set. Each *Tag Table* row also has a *tag_vector*, which is a 1,024-dimensional *images_embedding* of a recipe using the same embedding model of the *Recipe Table*.

Vector similarity queries in SQL. We designed 7 SQL queries to emulate various vector online application scenarios (§2.1). In particular, we also designed Q8 which runs an analytic join query based on vector similarity match. These 8 relational queries cover most SQL operators of `Projection`, `Index Scan`, `Sort with Limit`, `Filter`, `Join`, which are important for online queries over vector and scalar data-set.

- *Q1: Single-Vector* `TopK`*.*

```
SELECT recipe_id FROM Recipe
ORDER BY INNER_PRODUCT(images_embedding,
      ${p_images_embedding}) LIMIT 50;
```

- *Q2: Single-Vector* `TopK` *+ Numeric Filter.*

```
SELECT recipe_id FROM Recipe
WHERE popularity <= ${p_popularity}
ORDER BY INNER_PRODUCT(images_embedding,
      ${p_images_embedding}) LIMIT 50;
```

- *Q3: Single-Vector* `TopK` *+ String Filter.*

```
SELECT recipe_id FROM Recipe
WHERE description NOT LIKE "%${p_ingredient}%"
ORDER BY INNER_PRODUCT(images_embedding,
      ${p_images_embedding}) LIMIT 50;
```

- *Q4: Multi-Column* `TopK`*.*

```
SELECT recipe_id FROM Recipe
ORDER BY INNER_PRODUCT(images_embedding,
      ${p_images_embedding}) + WEIGHT * INNER_PRODUCT(
      description_embedding, ${p_description_embedding})
LIMIT 50;
```

[2]We remove those in the 1 Million recipes that miss any of the ingredients, cooking instructions, and related images.

- *Q5: Multi-Column* `TopK` *+ Numeric Filter.*

```
SELECT recipe_id FROM Recipe
WHERE popularity <= ${p_popularity}
ORDER BY INNER_PRODUCT(images_embedding,
      ${p_images_embedding}) + WEIGHT * INNER_PRODUCT(
      description_embedding, ${p_description_embedding})
LIMIT 50;
```

- *Q6: Multi-Column* `TopK` *+ String Filter.*

```
SELECT recipe_id FROM Recipe
WHERE description NOT LIKE "%${p_ingredient}%"
ORDER BY INNER_PRODUCT(images_embedding,
      ${p_images_embedding}) + WEIGHT * INNER_PRODUCT(
      description_embedding, ${p_description_embedding})
LIMIT 50;
```

- *Q7: Vector Range Filter.*

```
SELECT recipe_id FROM Recipe
WHERE INNER_PRODUCT(images_embedding,${p_images_embedding})
      <= ${D};
```

- *Q8: Join.*

```
SELECT Recipe.recipe_id, Tag.tag_name
FROM Recipe JOIN Tag
ON INNER_PRODUCT(Recipe.images_embedding,Tag.tag_vector)
      <= ${D};
```

We ran these 8 queries on VBase and compared them with query processing in other systems. For each query, we generated 10,000 substitution parameters to cover various query conditions. In particular, we set K to 50 for `TopK` queries as in the experiment of Milvus [76]. We also designed the numeric filtering constraints to cover both high and low filtering selectivities. *p_popularity* is incremented from 1 to 10000. *p_ingredient* is sampled from ingredients keywords in Recipe1M. $WEIGHT$ is 1. D is 0.1 in Q7 and 0.01 in Q8.

Evaluation metrics. Vector query executions are approximate, hence we evaluate both query accuracy and performance in terms of *recall* and *latency*, respectively. Recall is a new metric to conventional database evaluations. It evaluates query accuracy against the ground truth. Recall has been widely used in approximate vector search systems [2,3,76,86]. They only evaluate recall because for `TopK` queries, recall and precision are the same as long as a system returns K results. For other queries with range filter constraints, precision will always be 1 if the system obeys the constraints. Therefore, we use recall to represent query accuracy. For each query, we calculate the average recall of the query results of all substitution parameters. For each query in Q1-Q7, we measure the *average*, *median*, *99th percentile* latency from the execution results of all substitution parameters. For Q8, we execute it 3 times and measure the *average* execution time.

5.2 Experiment Setup

Evaluation platform. All evaluations run on an Azure VM,*Standard_F64s_v2* [1], with 64 v-CPUs and 128 GiB memory running Linux Ubuntu 20.04 LTS. All queries run individually to avoid interference from other queries.

Baseline systems. We compare VBASE with the state-of-the-art vector search and database systems that support vector similarity queries.

Vector search baselines: We choose *Milvus* [76] and *Elasticsearch* [4] as our vector search baselines. Since they do not support SQL interface, we hand code our benchmark queries. We also implement Iterative Merging algorithm as claimed in Milvus paper [76] to enable multi-column TopK queries, although unavailable in its open-source code [8]. For Elasticsearch we use the version implemented by Open Distro (version 1.13) [9], which supports HNSW index.

Database baselines: For databases, we use the open-source *PASE* [11, 86] that implements the `TopK`-only solutions using tentative indices, and extended its maximum dimension support from 512 to 1024. We also run queries on *PostgreSQL* (version 13) [12] as a baseline to show the performance of traditional databases in performing vector similarity queries.

Common index settings. VBASE and baseline systems all use HNSW [89] with the same vector index settings ($M = 16, ef_construction = 200, ef_search = 64$). HNSW is the only vector index supported by PASE, Milvus, and Elasticsearch in common. Since HNSW index is kept in memory when it is used, in order to better compare the performance results of the index-based execution process without being affected by the caching strategy of the main table adopted by different systems, we also save the main table data in memory. All the database baselines and VBASE also created a B-tree index on *popularity* column to expedite numeric filtering.

5.3 Evaluation Results

Overview. Table 4 summarizes the overall evaluation results. We can see that each baseline system based on approximate vector indices (except PostgreSQL) can only process some of the 8 queries, while VBASE can process all of them.

Although PostgreSQL can process all queries and produce *exact* results, it uses a brutal force scan with a much higher query latency than the rest of the systems. The 1000× lower performance of PostgreSQL than other approximate systems makes it irrelevant to address the low-latency "online" scenarios. We run queries on PostgreSQL mostly to get ground truth to calculate the query result accuracy of other systems.

VBASE's query performance on `TopK` (Q1) is similar to or better than baseline systems because they essentially run the same algorithm in different implementations. For queries that are more complex than Q1, VBASE outperforms all baseline systems by $100\times - 1000\times$ because VBASE can determine the optimal $\widetilde{K}$ on-the-fly and others have to try different Ks to get sufficient results. While VBASE produces superior query performance, it can also achieve high recall similar to or even higher than approximate queries on baseline systems.

Next, we discuss each query's evaluation result in detail.

Q1 – Single vector TopK. Q1 is a `TopK` query without any filters. All approximate systems including VBASE in our evaluation run the same algorithm and produce identical results, therefore having the exact same recalls.

Nevertheless, we can see variations in Q1 latency from different systems. The reasons for performance variations are two-fold. The first reason is that these systems are implemented in different languages (Milvus/Elasticsearch vs. PASE/VBASE). For example, Milvus are implemented in GoLang and C++, and Elasticsearch is implemented in Java and C++. In comparison, PASE and VBASE are written in C. In general, we can see implementation in C outperforms other high-level language implementations by $2 - 10\times$.

The second reason for performance variation (PASE vs. VBASE) is that VBASE needs to fetch slightly more tuples than PASE. Although following the same algorithms and traversing the same amount of vectors in the index, VBASE follows an Iterator Model, which fetches every corresponding tuple from the main table during index traversal. PASE's implementation visits vectors in the index and only fetches the K tuples after getting `TopK` vectors in the index. As a result, VBASE performs slightly worse, 2.8% slower than PASE in terms of average and 99 percentile latency.

Q2-3 – Single vector TopK with scalar filter. Q2 and Q3 are vector similarity queries with filtering on scalar attributes. Q2 and Q3 differ in their filter predicates, where Q2 uses numeric filtering on the integer attribute *popularity*, and Q3 runs string filtering based on a regular expression. All baselines support Q2 and Q3, with an exception of Milvus for Q3 because it does not support string data type.

All approximate baselines run a *single shot* of K'. Based on different guesses of K's, baseline systems produce different results. Elasticsearch undershot K for Q2 and Q3, therefore cannot produce sufficient results and has low query accuracy. Even though Elasticsearch has fewer data traversals than the rest of the approximate systems with higher accuracy, its query latency is still the worst, possibly due to system inefficiency like in Q1. PASE and Milvus overshot K and produce high result accuracy like VBASE, but they have longer query latency because they traverse more data than VBASE.

Q3 has a fixed filter selectivity of around 0.9. For Q2 we have 10,000 queries of different parameters with uniform distribution of filter selectivity from 0 to 1. We compared the evaluation results of two best approximate systems VBASE and PASE for Q2, under three representative $filter_selectivity$ values (0.03, 0.3, 0.9) from low to high (Table 5). We found that different filter selectivities result in different optimal $\widetilde{K}$s: the lower the selectivity, the more data a system needs to examine, therefore larger $\widetilde{K}$. Because $\widetilde{K}$ is dynamic, PASE's static guess of K' cannot produce constantly high recalls under all filter selectivities. A conservatively large guess of $K' = 10,000$ can produce near-exact results by PASE, however, its query performance deteriorates dramatically. We also present the average $\widetilde{K}$ and standard deviation under different filter selectivities in Table 5. $\widetilde{K}$ varies for different filter se-

Table 4: 8 Queries Result Overview (Latency: ms)

System	Q1:Single-Vector TopK				Q2:Single-Vector TopK+Numeric Filter				Q3:Single-Vector TopK+String Filter				Q4:Multi-Column TopK			
	Recall	Latency			Recall	Latency			Recall	Latency			Recall	Latency		
		average	median	99th		average	median	99th		average	median	99th		average	median	99th
PostgreSQL	1	2,980.1	3,021.7	3,133.6	1	1,108.3	1,124.1	2,286.2	1	4,322.2	3529.3	9,953.0	1	5,610.0	5,604.7	5,769.8
PASE	0.9949	**4.8**	**3.5**	**5.1**	0.9987	29.3	28.7	61.7	0.9982	13.2	10.7	17.9	-	-	-	-
Milvus	0.9949	9.4	9	12.7	0.9919	33.7	23.9	121.4	-	-	-	-	0.9041	6,696.4	8,349.3	9,299.0
Elasticsearch	0.9949	43.1	41.8	48.9	0.5010	97.9	98.1	118.1	0.8378	79.9	90.0	100.9	-	-	-	-
VBase	*0.9949*	*4.9*	*3.9*	*5.3*	***0.9989***	***11.7***	***6.3***	***51.7***	***0.9983***	***7.9***	***6.7***	***10.4***	***0.9696***	***19.8***	***18.4***	***46.4***

System	Q5:Multi-Column TopK+Numeric Filter				Q6:Multi-Column TopK+String Filter				Q7:Vector Range Filter				Q8:Join			
	Recall	Latency			Recall	Latency			Recall	Latency			Recall	Latency		
		average	median	99th		average	median	99th		average	median	99th		average	median	99th
PostgreSQL	1	1,192.9	1,234.4	2,343.6	1	6,543.2	5,996.3	16,734.6	1	8,244.9	8,212.6	8,641.6	1	129,051,273.9	-	-
PASE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Milvus	0.9691	12,637.9	5,617.4	36,887.9	-	-	-	-	-	-	-	-	-	-	-	-
Elasticsearch	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
VBase	***0.9805***	***35.8***	***24.9***	***160.7***	***0.9626***	***21.6***	***18.3***	***64.8***	***0.9840***	***10.8***	***2.2***	***168.9***	***0.9992***	***16,335.9***	-[1]	-[1]

[1] We have only run one query parameter for Q8, so average, median and 99th percentile latency are the same.

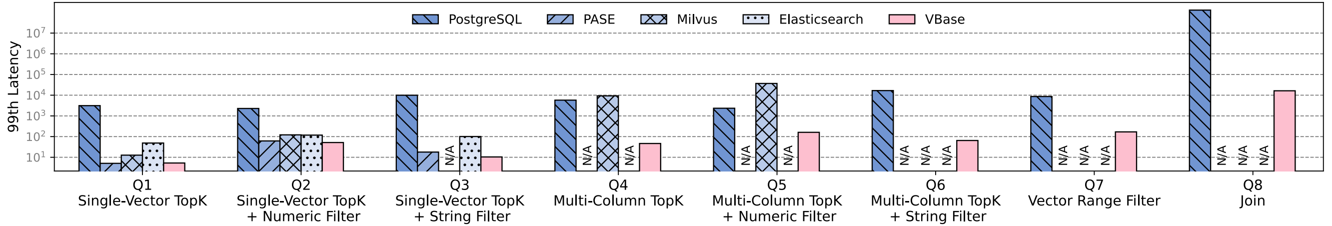


Figure 6: 99th Percentile Query Latency (ms)

lectivities. Even under the same selectivity, it also varies for different queries and the standard deviation is very large.

VBase can always produce the best query performance with high recalls since it can determine $\widetilde{K}$ on the fly.

Q4-6 – Multi-column TopK. Only Milvus and VBase support Q4-6, which are `TopK` queries over multiple vector indices. Q5,6 adds scalar filtering to Q4 just like in Q2,3, and Milvus cannot support string filter conditions in Q6. Milvus tries different K' to produce a sufficiently large intersection of multiple `TopK` results from different indices. Milvus's performance is worse than PostgreSQL which uses sequential table scan, because it cannot finish after several rounds of `TopK` guesses and accumulates a large number of random reads. In comparison, VBase determines the optimal $\widetilde{K}$ per each vector index based on relaxed monotonicity. Consequently, VBase outperforms Milvus by $200-300\times$ in terms of query latencies for Q4-6, and produces higher recalls (96%+).

We also experimented multi-column `TopK` queries with 4 kinds of weights in the ranking function, with different index-iteration algorithms as introduced in §4.4 (see Table 7). When the difference in weights is large (1:10), the greedy algorithm produces the best performance with high recalls. This is because the greedy approach identifies low-quality indices (i.e. ones with low-ranking weights) quickly, and avoids traversing them as much as possible. However, when weight differentiation decreases, the greedy algorithm can easily get trapped in local optima. This shortcoming of the greedy method is self-evident for a weight ratio of 1:1, where we can see greedy strategy extracts the highest number of entities while producing the lowest recall. In contrast, VBase shows higher recalls in all situations by dynamically determining a better strategy to switch among different indices while outperforming by 5% lower latency than the round-robin approach.

Q7 – Vector range filter. Table 4 shows that only VBase supports Q7. PASE does not support Q7 by default. We add a `Order By` distance clause with a hand-tuned "limit K" to force PASE to use its approximate vector index. This way it simulates the results of Q7. Like in Q2, it is difficult to set an appropriate K' for PASE ahead of time as shown in Table 6. We also present the average $\widetilde{K}$ and standard deviation, which also shows $\widetilde{K}$ changes dramatically for different queries. E.g., sometimes K is required to be 2300+ for optimality. VBase can achieve a great trade-off between query latency and recalls because its execution engine can determine $\widetilde{K}$ on-the-fly based on relaxed monotonicity.

On average Q7 only returns a small number of results that fit within the range. However, a small percentage of Q7s produce up to 10,000 results, which incurs high query costs in VBase. Therefore we can see that, in VBase, Q7's 99th percentile latency is much higher than the average (168.9ms vs 10.8ms) while the median is much smaller than the average.

Q8 – Join. PostgreSQL performs nested-loop join on table scan to get accurate results. In Q8, VBase is $7,900$x faster with recall=0.9992. Other systems cannot run this query due to the lack of a unified query engine.

5.4 VBase with SPANN

Table 8 shows the evaluation results for VBase with SPANN [25]. We run VBase with SPANN on Azure VM Standard_L16s_v3 with NVMe disks. SPANN is a partition-based ANNS index that uses external memory, i.e. disks. As

Table 5: Vector Search with Scalar Filter (Latency: ms)

System	Selectivity $=0.03$, $avg(\widetilde{K})=1,772, \sigma=224.16$				Selectivity $=0.3$, $avg(\widetilde{K})=291, \sigma=59.65$				Selectivity $=0.9$, $avg(\widetilde{K})=188, \sigma=50.33$			
	Recall	Latency			Recall	Latency			Recall	Latency		
		average	median	99th		average	median	99th		average	median	99th
PASE($K'=100$)	0.0567	5.1	5.1	6.3	0.5844	5.9	5.9	9.1	0.9947	5.5	5.7	10.4
PASE($K'=1,000$)	0.5885	21.5	21.2	30.8	0.9998	15.4	15.0	21.5	1	10.1	10.0	16.3
PASE($K'=10,000$)	1	62.8	62.5	78.7	1	48.9	49.3	61.1	1	41.8	41.7	53.1
VBase	0.9987	34.5	34.0	44.8	0.9966	7.6	7.0	8.5	0.9990	5.7	5.3	7.2

Table 6: Range Filter (Latency: ms)

System	$avg(\widetilde{K})=590, \sigma=1758.48$			
	Recall	Latency		
		average	median	99th
PASE($K'=100$)	0.7103	7.3	6.9	8.8
PASE($K'=1,000$)	0.9387	44.3	43.6	54.7
PASE($K'=10,000$)	0.9991	392.1	390.5	484.9
VBase	0.9840	10.8	2.2	168.9

Table 7: Multi-Column TopK Comparison (Latency: ms)

Weight[1]	Algorithm	NumOfScans[2]	Latency	Recall
1 : 1	Round-Robin	651.93	20.91	**0.9715**
	Greedy[3]	699.02	21.70	0.9313
	VBase	**638.56**	**20.56**	0.9705
1 : 2	Round-Robin	617.22	20.25	0.9802
	Greedy[3]	612.51	19.94	0.9655
	VBase	**593.99**	**19.78**	**0.9818**
1 : 5	Round-Robin	463.39	16.93	0.9946
	Greedy[3]	**372.96**	**14.90**	0.9949
	VBase	409.31	15.69	**0.9961**
1 : 10	Round-Robin	363.47	14.81	0.9981
	Greedy[3]	**274.86**	**12.69**	0.9985
	VBase	311.66	13.97	**0.9987**

[1] The weight ratio of two distances ($1:x$) in the ranking function.
[2] The average number of times we scan the two vector indices.
[3] In the greedy method, we first traverse each index 20 times and we find the index with the lowest average distance. Then, we extract candidates from this index only.

a result, query latencies on VBase with SPANN are generally higher than those for HNSW which are in-memory. This shows that VBase can support both partition-based vector indices as well as graph-based ones. In addition VBase can integrate indices stored both in memory and on disk seemly.

5.5 Cost Estimation

Selectivity estimation accuracy. We evaluate the accuracy of the selectivity estimation for vector range filter in terms of q-error [61]:

$$Q_{err} = \max(\frac{Sel_{esti}}{Sel_{real}}, \frac{Sel_{real}}{Sel_{esti}}).$$

As demonstrated in Figure 7, estimation based on sampling can provide a q-error less than 1.1 for most cases. When selectivity is lowest at 0.05, our samples cannot provide a high

Table 8: Queries on VBase with SPANN (Latency: ms)

Queries	Recall	Latency		
		average	median	99th
Q1	0.9911	9.4	9.2	11.6
Q2	0.9214	10.7	9.3	44.9
Q3	0.9847	9.7	9.4	11.8
Q4	0.9481	32.2	28.7	68.2
Q5	0.9757	87.4	55.9	519.7
Q6	0.9516	40.1	32.1	126.2
Q7	0.9923	17.8	9.3	283.5
Q8	0.9638	87,729.3	-[1]	-[1]

[1] We have only run one query parameter for Q8.

resolution, therefore its q-error increases up to 1.27. Increasing the sampling rate in cost estimation can reduce q-error further, but this increases selectivity estimation time for query planning, which is infeasible for online queries. In our experiments, such estimation accuracy is sufficient to support a good query plan strategy.

In comparison, systems like PASE [11] don't provide an estimation for selectivity and PASE sets the default value of selectivity estimation to 0.5.

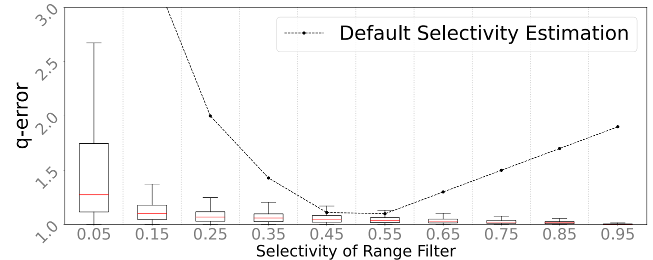


Figure 7: Q-error in different selectivity. Default estimated selectivity=0.5

Query Planning. We evaluate the efficacy of query planning using the following query.

```
SELECT recipe_id FROM Recipe WHERE
 ↪ INNER_PRODUCT(q, images_embedding) < ${r}
 ↪ AND popularity < ${p};
```

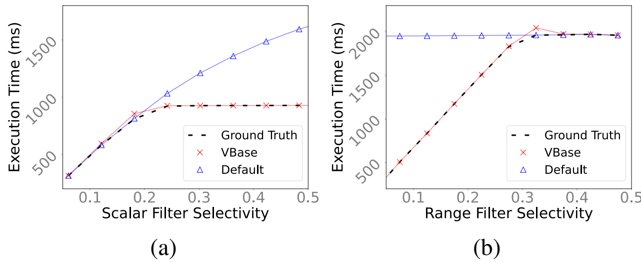
The vector range filter and the scalar filter can be accelerated via vector index or B-tree. Our experiments show that VBase can correctly choose the best execution plan under different selectivities, because our estimations of selectivity, vector computation, and index scan cost are accurate enough for

VBASE to construct good plans by reusing PostgreSQL's built-in mechanism.

Figure 8a shows execution times of different planning strategies for varying scalar selectivities and a fixed vector filter selectivity. The default strategy estimates selectivity as 0.5 as PASE does. The result demonstrates that VBASE can produce execution plans that closely match the ground truth of the best choice of index scan strategies. When scalar selectivity is less than 0.18, VBASE predicts accurately that the cost of execution via B-tree index is smaller than vector index traversal, and chooses to run it. Likewise, if scalar selectivity is over 0.18, VBASE correctly chooses vector index traversal.

On the other hand, experiments in Figure 8b vary vector filter selectivities and fix scalar filter selectivity. Like in Figure 8a, the result shows VBASE can create the high efficacy of query planning based on highly accurate cost estimation.

In comparison, systems like PASE do not provide an accurate estimation for selectivity. And they do not tune cost estimation for vector computation and vector index traversal either. The inaccurate estimation causes PASE to always execute queries via B-tree index in this experiment.



(a) (b)

Figure 8: Query execution time with different estimation. We fixed range filter selectivity=0.13 in (a), and scalar filter selectivity=0.90 in (b)

6 Related Works

Similarity Query in Databases. Several works [20, 21, 71, 73, 74] have explored extending database systems to support accurate similarity query of low-dimensional vector data, in which *K-NN(`TopK`)* and *range filter* query are well described. R-Tree [38], KD-Tree [34], M-Tree [27], Slim-Trees [47] can be used in these works as indices for low-dimensional data. [20] proposes to include similarity queries to SQL and run them on SIREN [21]. [71] presents similarity Join and similarity Group-by operators. The similarity Group-by operator for high-dimensional vector is actually equivalent to the clustering operation, which has been well-studied by [15, 18, 50, 65]. However, all of these works are about clustering data on the main table instead of the vector indices.

Vector Indices. Vector indices support approximate nearest neighbor search efficiently on high-dimensional vector by `TopK` interface. They can be divided into two categories: graph-based approach and partition-based approach. The partition-based approach divides the whole vector space into many sub-spaces, and uses some metric (e.g. a centroid, a hash value or a divisional plane) to represent all vectors that belong to a sub-space. During the traverse, it navigates a query to its approximate nearest sub-spaces step by step based on distances between the query and the representative metric of sub-spaces. Representative partition-based approaches include clustering-based solutions [5, 17, 19, 25, 44, 45, 48, 90], hash-based solutions [30, 41, 79, 81, 85], and tree-based solutions [23, 57, 62, 78]. The graph-based approach represents each vector as a vertex, each connected to its nearest vectors (i.e. neighbors) by edges in a graph. There are also some shortcut edges connecting to distant vector vertices, which can speed up the graph traversals. Using this neighborhood graph, index traversal can be guided by a query approximately towards its closest neighbors step by step from a fixed starting point [32, 39, 43, 55, 58, 77]. `TopK` interface in vector indices has limited query expressiveness.

Vector Databases based on `TopK`. AnalyticDB-V [80], PASE [86], Milvus [76], and Elasticsearch [4] support complex vector queries based on the original `TopK` interface in vector indices. AnalyticDB-V [80] and PASE [86] integrate vector indices into the database engine to support SQL interface for similarity queries. Elasticsearch [4] is a distributed full-text search engine, providing approximate nearest neighbor search based on HNSW [89]. AnalyticDB-V [80], PASE [86], and Elasticsearch [4] begin to support vector search plus scalar attribute filtering. Milvus [76] is a data management system to efficiently manage large-scale vector data, which can additionally support multi-column `TopK` queries by iteratively speculating the K with a growing value. In contrast, VBASE does not rely on a tentative index collected by `TopK`.

7 Conclusion

This paper presents VBASE, a vector database that integrates high-dimensional vector indices into PostgreSQL, a relational database to facilitate complex approximate similarity queries. Unlike conventional approaches that leverage `TopK` to collect the target vector's K nearest neighbors where a conventional index is constructed for query execution, VBASE builds on relaxed monotonicity, a common foundation between conventional and high-dimensional indices. This common foundation allows VBASE to build a unified query execution engine that produces query results equivalent to those produced by `TopK`-based solutions with the optimal $\widetilde{K}$. As a result, VBASE significantly outperforms state-of-the-art vector systems on complex vector queries.

8 Acknowledgement

We would like to thank our shepherd Marco Serafini and the anonymous reviewers for their insightful comments.

References

[1] Azure vm fsv2-series. https://learn.microsoft.com/en-us/azure/virtual-machines/fsv2-series.

[2] Benchmarking nearest neighbors. http://ann-benchmarks.com/.

[3] Billion-scale anns benchmarks. https://big-ann-benchmarks.com/.

[4] Elasticsearch. https://www.elastic.co/.

[5] Facebook faiss. https://github.com/facebookresearch/faiss.

[6] Facebook simsearchnet. https://ai.facebook.com/blog/using-ai-to-detect-covid-19-misinformation-and-exploitative-content/.

[7] Google multisearch. https://blog.google/products/search/multisearch/.

[8] Milvus. https://github.com/milvus-io/milvus.

[9] Open distro. https://github.com/opendistro-for-elasticsearch/.

[10] Openai chatgpt retrieval plugin. https://github.com/openai/chatgpt-retrieval-plugin.

[11] Pase. https://github.com/forrest-2007/PASE.

[12] postgresql. https://www.postgresql.org/.

[13] The TPC-C benchmark. http://www.tpc.org/tpcc/.

[14] The TPC-H benchmark. http://www.tpc.org/tpch/.

[15] Saurabh Arora and Inderveer Chana. A survey of clustering techniques for big data analysis. In *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pages 59–65, 2014.

[16] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, jun 1976.

[17] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence*, 37(6):1247–1260, 2014.

[18] B. Hari Babu, N. Subhash Chandra, and T. V. Gopal. Clustering algorithms for high dimensional data – a survey of issues and existing approaches. 2012.

[19] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 202–216, 2018.

[20] M. C. N. Barioni, H. L. Razente, A. J. M. Traina, and C. Traina. Seamlessly integrating similarity queries in sql. *Softw. Pract. Exper.*, 39(4):355–384, mar 2009.

[21] Maria Camila N. Barioni, Humberto Razente, Agma Traina, and Caetano Traina. Siren: A similarity retrieval engine for complex data. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1155–1158. VLDB Endowment, 2006.

[22] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[23] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[24] Donald D. Chamberlin. Early history of sql. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012.

[25] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 5199–5212. Curran Associates, Inc., 2021.

[26] Sheng Chen, Akshay Soni, Aasish Pappu, and Yashar Mehdad. Doctag2vec: An embedding based multi-label learning approach for document tagging. *arXiv preprint arXiv:1707.04596*, 2017.

[27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. *International conference on very large data bases (VLDB)*, 08 2001.

[28] Kenneth L Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 160–164, 1994.

[29] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[30] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme

based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, 2004.

[31] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[32] Wei Dong, Moses Charikar, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 577–586, 2011.

[33] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In Peter Buneman, editor, *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.

[34] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, sep 1977.

[35] G. Graefe. Volcano an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, feb 1994.

[36] Wayne D Gray and Deborah A Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of experimental psychology: applied*, 6(4):322, 2000.

[37] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, 2022.

[38] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 47–57, New York, NY, USA, 1984. Association for Computing Machinery.

[39] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1312–1317, 2011.

[40] G. D. Held, M. R. Stonebraker, and E. Wong. Ingres: A relational data base system. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, page 409–416, New York, NY, USA, 1975. Association for Computing Machinery.

[41] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[42] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4):219–230, 2013.

[43] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[44] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[45] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[46] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547, 2021.

[47] Caetano Jr, Agma Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. 03 2000.

[48] Yannis Kalantidis and Yannis Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2321–2328, 2014.

[49] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. Efficient retrieval of recommendations in a matrix factorization framework. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 535–544, 2012.

[50] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1), mar 2009.

[51] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th International Conference on Computer Vision*, pages 2130–2137, 2009.

[52] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

[53] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. Fexipro: fast and exact inner product retrieval in recommender systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 835–850, 2017.

[54] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. The design and implementation of a real time visual search system on JD e-commerce platform. In *Proceedings of the 19th International Middleware Conference, Middleware Industrial Track 2018, Rennes, France, December 10-14, 2018*, pages 9–16. ACM, 2018.

[55] Jie Ren. Minjia Zhang. Dong Li. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. In *Advances in Neural Information Processing Systems*, 2020.

[56] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. Lightrec: A memory and search-efficient recommender system. In *Proceedings of The Web Conference 2020*, pages 695–705, 2020.

[57] Ting Liu, Andrew W Moore, Alexander Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, {NIPS} 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, pages 825–832, 2004.

[58] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016.

[59] Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1m+: A dataset for learning cross-modal embeddings for cooking recipes and food images. *IEEE transactions on pattern analysis and machine intelligence*, 43(1):187–203, 2019.

[60] Antoine Miech, Dimitri Zhukov, Jean-Baptiste Alayrac, Makarand Tapaswi, Ivan Laptev, and Josef Sivic. Howto100m: Learning a text-video embedding by watching hundred million narrated video clips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2630–2640, 2019.

[61] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.*, 2(1):982–993, aug 2009.

[62] Marius Muja and David G. Lowe. Scalable Nearest Neighbour Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[63] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. Ms marco: A human generated machine reading comprehension dataset. In *CoCo@ NIPS*, 2016.

[64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[65] Divya Pandove, Shivan Goel, and Rinkl Rani. Systematic review of clustering high-dimensional and large datasets. *ACM Trans. Knowl. Discov. Data*, 12(2), jan 2018.

[66] Mattis Paulin, Matthijs Douze, Zaid Harchaoui, Julien Mairal, Florent Perronin, and Cordelia Schmid. Local convolutional features with unsupervised training for image retrieval. In *Proceedings of the IEEE international conference on computer vision*, pages 91–99, 2015.

[67] Jianbin Qin, Wei Wang, Chuan Xiao, and Ying Zhang. Similarity query processing for high-dimensional data. *Proc. VLDB Endow.*, 13(12):3437–3440, sep 2020.

[68] Amaia Salvador, Nicholas Hynes, Yusuf Aytar, Javier Marin, Ferda Ofli, Ingmar Weber, and Antonio Torralba. Learning cross-modal embeddings for cooking recipes and food images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3020–3028, 2017.

[69] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[70] Minjoon Seo, Jinhyuk Lee, Tom Kwiatkowski, Ankur P Parikh, Ali Farhadi, and Hannaneh Hajishirzi. Real-time open-domain question answering with dense-sparse phrase index. *arXiv preprint arXiv:1906.05807*, 2019.

[71] Yasin N. Silva, Walid G. Aref, Per-Ake Larson, Spencer S. Pearson, and Mohamed H. Ali. Similarity queries: Their conceptual evaluation, transformations, and processing. *The VLDB Journal*, 22(3):395–420, jun 2013.

[72] Yukihiro Tagami. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 455–464, 2017.

[73] Caetano Traina, Andre Moriyama, Guilherme Rocha, Robson Cordeiro, Cristina D. A. Ciferri, and Agma Traina. The similarql framework: Similarity queries in plain sql. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 468–471, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Caetano Traina, Agma J. M. Traina, Marcos R. Vieira, Adriano S. Arantes, and Christos Faloutsos. Efficient processing of complex similarity queries in rdbms through query rewriting. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, page 4–13, New York, NY, USA, 2006. Association for Computing Machinery.

[75] Robert E. Wagner. Indexing design considerations. *IBM Systems Journal*, 12(4):351–367, 1973.

[76] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[77] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1106–1113. IEEE, 2012.

[78] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian Sheng Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(2):388–403, 2014.

[79] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):769–790, 2018.

[80] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020.

[81] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In *Advances in neural information processing systems*, pages 1753–1760, 2009.

[82] Xian Wu, Moses Charikar, and Vishnu Natchu. Local density estimation in high dimensions. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5296–5305. PMLR, 10–15 Jul 2018.

[83] Xiang Wu, Ruiqi Guo, David Simcha, Dave Dopson, and Sanjiv Kumar. Efficient inner product approximation in hybrid spaces. *ArXiv*, abs/1903.08690, 2019.

[84] Shitao Xiao, Zheng Liu, Weihao Han, Jianjin Zhang, Yingxia Shao, Defu Lian, Chaozhuo Li, Hao Sun, Denvy Deng, Liangjie Zhang, et al. Progressively optimized bi-granular document representation for scalable embedding based retrieval. In *Proceedings of the ACM Web Conference 2022*, pages 286–296, 2022.

[85] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. Complementary hashing for approximate nearest neighbor search. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1631–1638. IEEE, 2011.

[86] Wen Yang, Tao Li, Gai Fang, and Hong Wei. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2241–2253, 2020.

[87] Ian En-Hsu Yen, Satyen Kale, Felix Yu, Daniel Holtmann-Rice, Sanjiv Kumar, and Pradeep Ravikumar. Loss decomposition for fast learning in large output spaces. In *International Conference on Machine Learning*, pages 5640–5649. PMLR, 2018.

[88] Zeynep Akkalyoncu Yilmaz, Shengjin Wang, Wei Yang, Haotian Zhang, and Jimmy Lin. Applying bert to document retrieval with birch. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*, pages 19–24, 2019.

[89] Malkov. D A Yashunin. Yu A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 824–836, 2018.

[90] Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. In *Proceedings of the 31th International Conference on Machine Learning (ICML)*, volume 32, pages 838–846, 2014.