QBase: A Query Aware Vector Search System

[†]Boyu Tan, [†]Tang Qian, [†]Ziquan Fang, [†]Lu Chen, [†]Mengzhao Wang, [†]Qilong Wang, [‡]Jingwen Zhao
[†]Zhejiang University [‡]Poisson Lab. of Huawei
[†]{*tanboyu, qt.tang.qian, zqfang, luchen, wmzssy, qlwang*}*@zju.edu.cn* [‡]jingwenzhao5@huawei.com

***Abstract*—Vector search has recently attracted massive attention in both academia and industry due to its widespread applications in various fields, such as multimedia search and large language models (LLMs). However, existing vector search systems often overlook the specific characteristics of similar query vector scenarios. That is, in real-world applications, many queries exhibit similarity, typically returning highly overlapping results. Inspired by that, to fully leverage the unique characteristics of similar query vectors, we propose QBase, a vector search system designed to enhance query accuracy and performance. QBase is optimized for query vectors and automatically identifies and processes similar queries. It incorporates the AV-tree index, which is specifically designed for query vectors, allowing it to exploit their similarities fully. Building on the AV-tree index, we further introduce the A3V-tree index, which includes numerous optimizations to support efficient multi-vector searches. QBase supports up to six types of vector queries and four query plans, utilizing a cost-estimation approach to select the most efficient query plan for optimal performance. Extensive evaluations across various vector datasets demonstrate that QBase outperforms the state-of-the-art vector systems, improving query accuracy by 3% to 30% and query efficiency up to 90%.**

***Index Terms*—Vector search, vector databases, hybrid search, approximate nearest neighbor search**

I. Introduction

With the proliferation of IoT devices, multimedia systems, and mobile applications, there is a significant growth in unstructured data such as high-definition videos, digital images, and text. Handling and analyzing unstructured data necessitates advanced processing techniques. With the success of deep learning models, it has become common practice to transform these unstructured data into high-dimensional vectors [1]–[3]. High-dimensional vector similarity search has been playing a crucial role in many fields, including large language models (LLMs) [4]–[7], scientific computing [8], [9], information retrieval [10], [11], databases [12], [13], recommendation systems [14], [15], etc.

Due to the curse of dimensionality [16], many traditional metric indexes such as M-Tree [17], SPB-Tree [18], and R-Tree [19] fail to function effectively and degrade to brute-force search for accurate vector search, which can be highly time-consuming and impractical for applications requiring millisecond-scale latency [20]–[22]. Therefore, a more efficient and practical alternative is to perform an approximate vector search, which is extensively implemented in industrial systems [23]–[26] and widely studied in the academic community [20], [27], [28]. And therefore, graph-based [13], [29]–[33] or partition-based [20], [29], [34], [35] indexes are primarily adopted as they achieve a state-of-the-art balance between search accuracy and efficiency in various application scenarios [21], [22], [36]–[39]. The basic idea is to strategically visit a subset of the complete vector dataset to obtain query results, significantly reducing redundant distance computations with a slight accuracy loss [23], [26], [29], [40]–[43]. Overall, existing vector search studies aim to balance query latency and recall rate of query results [40], [44].

From an application perspective, vector search involves two cases. **i) Multi-Vector Search.** It refers to scenarios where each object is associated with multiple vector representations. For example, in e-commerce applications, products are represented using both image and text vectors. Systems such as Milvus [23], VBase [27], and Qdrant [24](however, Qdrant just supports tensor) support this. **ii) "Vector + Filter" Search.** It involves not only vector data but also non-vector data (referred to as "Filter"), such as GPS coordinates. Milvus [23], VBase [27], Weaviate [25], and AnalyticDB-V [26] support such queries.

However, we notice that many issued queries usually return highly overlapped results [45], [46]. For example, in the knowledge graph field, the query points in a batch are relatively similar to each other [47]. This phenomenon widely exists in real-world applications. In recommendation systems [48], users' queries (e.g., vectors based on user past preferences) often cluster around similar items. This is because similar users may have similar interests in items. Similarly, during the occurrence of hot events such as major news or celebrity updates, a large number of similar queries tend to be triggered. Fig. 1 illustrates three real-world examples. In Fig. 1(a), a community of Taylor Swift enthusiasts is likely to search for content related to their idol. In Fig. 1(c), individuals interested in learning about machine learning are likely to seek information on the topic. In Fig. 1(b), photography enthusiasts search for sunset-related information, including both textual descriptions and images. Users can specify attributes of the sunset, such as its color or whether it occurs by the seaside. In these multimodal scenarios, users refine their semantic queries by incorporating multiple modalities, thereby obtaining more relevant results. Consequently, queries from these user groups tend to return highly similar search results.

When reviewing existing vector similarity search studies, we have identified three unaddressed challenges below.

Challenge I: how to jointly improve query efficiency and accuracy by leveraging users' queries? Existing studies tend to sacrifice either query accuracy or query efficiency. To improve efficiency, existing systems such as Milvus [23] leverage caching optimizations. When they find a hash value

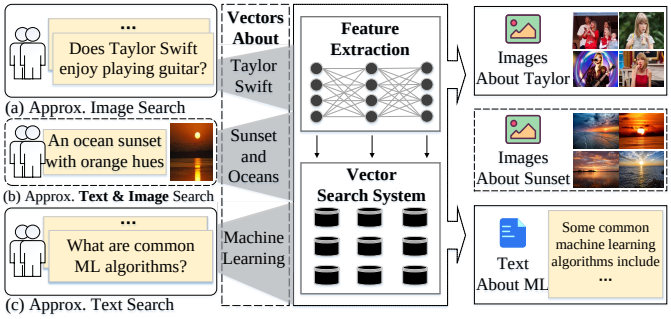


Fig. 1: Motivating Examples

in the result cache, they will select the corresponding *top*-1 result as the starting point for the vector index to improve efficiency. However, if the initial query result is inaccurate, this approach continuously propagates errors to subsequent queries, keeping them in a problematic state. For instance, in e-commerce searches, when users search for keywords of popular products, Milvus may lead to a disordered ranking of search results due to inaccurate cached results, thus affecting users' purchasing decisions. To improve accuracy, existing systems like VBase [27] use a more "brute force" way by leveraging underlying indexing mechanisms to execute queries. Although the indexing can guarantee high accuracy, it does not utilize the similarity characteristics of query vectors, resulting in decreased efficiency. For instance, when VBase processes complex queries, the search time may be excessively long, leading to an increase in user churn rate.

Overall, how to jointly improve the query accuracy and efficiency is challenging. To fill this gap, we leverage the characteristics of query vector similarity for optimization, i.e., we utilize query vectors for index construction. Inspired by AV-tree index [49] that use query vectors for index construction, we fully exploit the features of similar query vectors to improve its performance, including i) utilizing query vectors similar to each other for index construction, making the index more compact and effective; and ii) given multiple tree indexes built on different sets of query vectors, choosing the tree index (built on query vectors that are most similar to the current query vector) for search. Compared to naively storing prior query vectors, an index can both control the memory explosion due to query vectors and exploit its characteristics to further filter, enhancing query efficiency. Additionally, we implement *top*-k-based optimization method, by adaptively estimating the k-th nearest neighbor distance, irrelevant result vectors can be quickly filtered out to further improve the query efficiency while minimally sacrificing query accuracy.

Challenge II: how to correctly and appropriately determine whether two query vectors are "similar"? User queries are often continuous, and the similarity between vectors cannot be predetermined. Matching a new query vector solely with the current nearest query vector may result in a concentration of errors, as they may actually be distant in terms of distance value. If two query vectors are incorrectly identified as similar, it can lead to suboptimal query results. This essentially involves how to perform effective clustering. Using existing methods such as LSH [21], [50], [51] or k-means [52], [53] may pose challenges. For LSH, it's difficult to control the number of partitions to ensure proper differentiation of query vectors. Similarly, k-means requires determining the number of clusters in advance, which can lead to misclassification of query vectors that originally do not belong to the same group.

Therefore, we design a new method to address this challenge. We observe that in high-dimensional vector spaces, determining the similarity of query vectors cannot rely solely on the query vectors themselves. Due to the curse of dimensionality, traditional distance-based methods often fail to capture similarity accurately. In high-dimensional datasets, vectors are typically sparsely distributed, making the relations between them more complex. Therefore, we propose determining the similarity of query vectors based on the overall distribution of the vector dataset being searched. To achieve this, we exploit the property of uniform distribution of distances among high-dimensional vectors. By randomly sampling vectors from the dataset, we calculate the distances between these sampled vectors, and the average of these distances is defined as the "uniform distance". This uniform distance serves as a measure of the typical separation between vectors in the dataset. We then establish a threshold by reducing this uniform distance by a specific ratio, with the ratio being determined by the distribution characteristics of the dataset itself. For datasets with a more concentrated distribution, a smaller ratio is used to enforce a stricter similarity criterion. Conversely, for datasets with a more dispersed distribution, a larger ratio is adopted to capture a sufficient number of similar query vectors.

Challenge III: how to support a diverse range of vector search types to satisfy practical scenarios? Most existing well-known vector systems, including Vearch [43], Faiss [29], and AnalyticDB-V [26], do not effectively support both **Multi-Vector** and **Vector + Filter** query cases simultaneously. Although VBase and Milvus have already supported them, a more detailed examination reveals that these systems are still not comprehensive enough. In this paper, we further divide these two major categories into six specific query types. **S1: Single-Vector *Top-k*** involves simply retrieving the *top-k* results from a single-vector dataset. It is widely used in information retrieval [54] and advertising [55], [56]. **S2: Single-Vector *Top-k* plus scalar attribute filter** is based on Single-Vector *Top-k*. It further filters the results via additional attributes associated with the vectors. For instance, Milvus [23], AnalyticDB-V [26] and VBase [27] belong to this category. **S3: Multi-Vector *Top-k*** is similar to **S1**, but instead of retrieving single-vector, multi-vector is jointly searched. **S4: Multi-Vector *Top-k* plus scalar attribute filter** is similar to **S2**, but instead of retrieving single-vector, multi-vector is jointly searched. **S5: Vector Range Filter**. For a single-vector, it finds all results within a certain distance range from the query point. **S6: Multi-Vector Range Filter** further extends support for **Multi-Vector Search** based on **S5**. This is a typical scenario in the multimodal domain [57], [58]. Both Milvus and VBase only support **S3**, **S4**, and **S5** types. Existing systems, when handling **Multi-Vector Search**, essentially perform separate searches using multiple single-

vector indexes and then merge the results, which method fails to simultaneously consider other vector columns during the search process, leading to suboptimal efficiency. To address this issue, we implemented the A3V-tree index based on the AV-tree index. This index supports multi-vector search within a single-vector index, which significantly enhances both efficiency and accuracy. To ensure system robustness and enhance compatibility with general vector query scenarios, as well as to address the inefficiency of the A3V-tree index during startup, we design a hybrid indexing scheme, that deeply integrates the A3V-tree index with mainstream vector indexes.

To address the above challenges, we design the QBase system, i.e., a query-aware vector search system. It i) fully leverages the similarity characteristics of query vectors to enhance the accuracy and efficiency of vector search; ii) employs a hybrid index that combines existing state-of-the-art vector index HNSW and our optimized A3V-tree index to ensure high query efficiency and accuracy in all cases; iii) supports more comprehensive complex vector query search capabilities; and iv) designs a cost model to select the best query plan. Overall, this paper makes key contributions as follows.

- Inspired by real-world queries that typically return similar query results, we propose the first query-aware vector search system QBase, which significantly enhances query accuracy and efficiency. Besides, QBase supports various vector search types, offering a more comprehensive solution than state-of-the-art systems.
- We construct the index based on query vectors to address scenarios involving similar query vectors. Inspired by the AV-tree index, we extend it to support multi-vectors and leverage the similar characteristics of query vectors to enhance the vector search performance. Utilizing the uniform distance characteristics of high-dimensional vector datasets, we effectively derive a threshold to determine whether query vectors are similar to each other.
- QBase supports a comprehensive coverage of vector search types and provides various query plans. It implements a hybrid index consisting of state-of-the-art vector index and our designed index, and utilizes a cost model to select the best query plan with proper index structure to achieve high query accuracy and efficiency over all cases.
- Extensive experiments using real-life datasets show that QBase significantly outperforms competitors in terms of efficiency, accuracy, and scalability.

Our source code at https://github.com/ZJU-DAILY/QBase is available for further studies.

II. Preliminaries

A. Vector Space

A vector space is defined as a pair $\{\mathbb{V}^D, d_W\}$, where $\mathbb{V}^D$ denotes the vector domain, while d_W denotes the distance function. The form of a multi-vector is $\{v_1, v_2, ..., v_D\}$, which consists D single vector(s) v_i $(1 \leq i \leq D)$. Given two multi-vectors V_1 and V_2, $d_W(V_1, V_2) = \sum_{i=1}^{D} d(v_1[i], v_2[i]) * w_i$, where w_i denotes the corresponding weight. The commonly used distance function $d()$ is the L_2-norm (or any vector distance function that satisfies the following four characteristics). Generally, d_W satisfies the following four properties:

- **Identity**. The distance to itself is 0, $d_W(V_1, V_1) = 0$.
- **Non-negativity**. The distance between two distinct vectors is positive; if $V_1 \neq V_2$ then $d_W(V_1, V_2) > 0$.
- **Symmetry**. The distance from V_1 to V_2 is the same as that from V_2 to V_1, $d_W(V_1, V_2) = d_W(V_2, V_1)$.
- **Triangle Inequality**. For any three vectors V_1, V_2, and V_3, if $V_1 \neq V_2 \neq V_3$, then $d_W(V_1, V_3) \leq d_W(V_1, V_2) + d_W(V_2, V_3)$.

TABLE I: Symbols and Descriptions

Notation	Description
$d(\cdot,\cdot)$	Distance function for two single-vectors.
$d_W(\cdot,\cdot)$	Weighted distance function for two multi-vectors.
$d_{1.0}(\cdot,\cdot)$	Special case of d_W where each vector has weight 1.0.
δ	Threshold for constructing a new A3v-tree index.
D	Number of single-vectors in a multi-vector.
$\mathbb{R}^D$	Vector dataset with cardinality n.
$\mathbb{V}^D$	Multi-vectors, each with D single-vectors.
w_i	Weight of i-th single vector for multi-vector.
q	Query vector that can be weighted as a multi/single-vector.
ϵ	Query radius for range query.
k	Number of result vectors for k nearest neighbor search.

B. Similarity Queries

Definition 1. ***k Nearest Neighbor Search kNNS***(q, k). *Given a positive integer k $(k > 0)$, a query vector q and the vector dataset $\mathbb{R}^D$, kNNS(q, k) finds a result vector dataset composed of k vectors $\mathbf{V}_k = \{v_1, v_2, \ldots, v_k\}$ from $\mathbb{R}^D$ that are closest to q, i.e., for any $v_i \in \mathbf{V}_k$ and $v_j \in \mathbb{R}^D \setminus \mathbf{V}_k$, $d(v_i, q) \leq d(v_j, q)$.*

Definition 2. ***Range Search*** $RS(q, \epsilon)$. *Given a query radius ϵ, a query vector q and the vector dataset $\mathbb{R}^D$, $RS(q, \epsilon)$ finds a result vector dataset $\mathbf{V}_{RQ}$ from $\mathbb{R}^D$ that contains vectors v_i satisfying $d(v_i, q) \leq \epsilon$.*

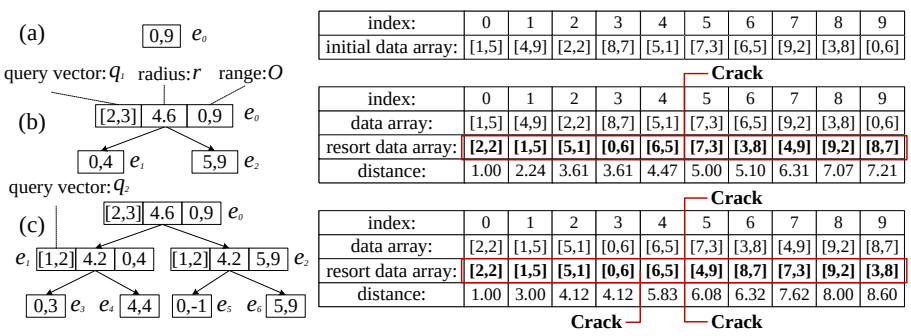
Definition 3. ***Multi-Vector Search***. *Specifically, for the above-mentioned RS and kNNS, when D is greater than 1, we collectively refer to it as Multi-Vector Search.*

C. AV-tree Index

To support efficient accurate queries on high-dimensional vectors, AV-tree index [49] is designed. It constructs the AV-tree index using the query data, and thus, the tree construction (or insert) based on query vectors and query processing are not distinct. Fig. 2 presents the running example of AV-tree index and corresponding similarity queries, while Algorithm 1 provides the pseudocode of AV-tree index construction and corresponding queries. Here, we use 10 2D vectors as data points (stored in the initial data array), and use L_2-norm distance as the metric for illustration.

(i) Initially, in Fig. 2(a), we start with an empty tree, which only contains an initial root node e_0 that stores an index range (0, 9). Note that, the index nodes do not store the detailed data (the detailed data is stored in the data array), but maintain index ranges.

(ii) Next, a range search with a radius $\epsilon = 4.6$ and a query vector q = [2, 3] is performed on the index in Fig. 2(a),



index:	0	1	2	3	4	5	6	7	8	9
initial data array:	[1,5]	[4,9]	[2,2]	[8,7]	[5,1]	[7,3]	[6,5]	[9,2]	[3,8]	[0,6]

index:	0	1	2	3	4	5	6	7	8	9
data array:	[1,5]	[4,9]	[2,2]	[8,7]	[5,1]	[7,3]	[6,5]	[9,2]	[3,8]	[0,6]
resort data array:	**[2,2]**	**[1,5]**	**[5,1]**	**[0,6]**	**[6,5]**	**[7,3]**	**[3,8]**	**[4,9]**	**[9,2]**	**[8,7]**
distance:	1.00	2.24	3.61	3.61	4.47	5.00	5.10	6.31	7.07	7.21

index:	0	1	2	3	4	5	6	7	8	9
data array:	[2,2]	[1,5]	[5,1]	[0,6]	[6,5]	[7,3]	[3,8]	[4,9]	[9,2]	[8,7]
resort data array:	**[2,2]**	**[1,5]**	**[5,1]**	**[0,6]**	**[6,5]**	**[4,9]**	**[8,7]**	**[7,3]**	**[9,2]**	**[3,8]**
distance:	1.00	3.00	4.12	4.12	5.83	6.08	6.32	7.62	8.00	8.60

Fig. 2: AV-tree Index Build and Search Example

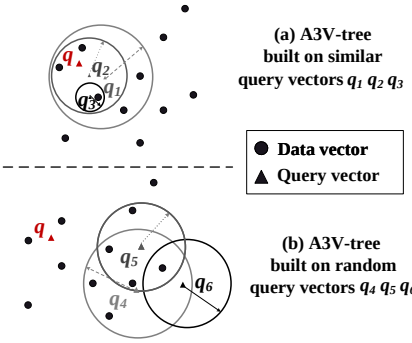


Fig. 3: A3V-tree Index

while the index is updated after the search as depicted in Fig. 2(b). Search-and-crack algorithm first checks the root node e_0 whether it contains any query vector (line 1). If it contains a query vector, this node is a non-leaf node; otherwise, it is a leaf node. In the initial case, the root node is a leaf node. Then it verifies whether each data vector p in this node is the final result (lines 4–9). Specifically, if $d(p,q) \leq \epsilon$, p is the range query result according to Definition 2 (line 8). In our example, a linear scanning is performed to find the result, i.e., we compute the distance between each data vector and the query vector and resort the data array in ascending order of the computed distances. Thus, data vectors with the index range (0,4) are the final results. Next, the algorithm calls the CrackInTwo function to update the index. As shown in Fig. 2(b), the range query information is inserted into the root node, while subnodes e_1 (to store data vectors with the index range (0,4) whose distances to q no larger than ϵ) and e_2 (to store data vectors with the index range (5, 9) whose distance to q larger than ϵ) are created.

(iii) Next, a range search with a radius $\epsilon = 4.2$ and a query vector q = [1, 2] is performed in the index in Fig. 2(b), while the index is updated after the search as depicted in Fig. 2(c). In this case, the root node e_0 contains a query vector, and two pruning rules and one validation rule are used. Specifically, 1) if $d(q, e.q) > \epsilon + e.r$, then the leaf sub-tree can be pruned (lines 11–13); 2) if $d(q, e.q) > e.r - \epsilon$, then the right sub-tree can be pruned (lines 20–23); and 3) if $d(q, e.q) \leq \epsilon - e.r$, then data vectors in the leaf sub-tree are the final result without any verification (lines 15–18). In our example, we cannot prune or validate any sub-tree, and thus, we further search the final result in e_1 and e_2 and also use the range query information to crack e_1 and e_2.

III. A3V-tree Index

A. A3V-tree Index Structure

As A3V-tree index extends AV-tree to support multi-vectors, the query vector (stored in non-leaf nodes) and data vector (stored in the data array) needs to be extended to store multi-vectors. The A3V-tree is very similar to AV-tree (as depicted in Fig. 2). Each non-leaf node $e = (q, r, O)$ stores the query vector q, a radius r, and a data index range O (i.e., the data vectors stored in the subtree of e). For each vector data $v \in e.left$ (i.e., each vector data in the left sub-tree of e), $d_{1.0}(q, v) \leq r$, while for each vector data $v \in e.right$ (i.e., each vector data in the right sub-tree of e), $d_{1.0}(q, v) > r$. To ensure the correctness

Algorithm 1: Distance-Range Search

Input: data array $\mathbb{R}^D$, query q, radius ϵ, node e, result R
Output: updated result set R

```
1  Function search-and-crack(ℝ^D, q, ϵ, e, R):
2      if e.q is null                                   // leaf node
3      then
4          foreach v in e                               // v is a data point
5          do
6              dist ← d(q, v)
7              if dist ≤ ϵ then
8                  result.push(v)
9          CrackInTwo(e.left, e.right, q, ϵ, ℝ^D)
10     else
11         if d(q, e.q) > ϵ + e.r                       // disjoint query ranges
12         then
13             search-and-crack(ℝ^D, q, ϵ, e.right, R)
14         else
15             if d(q, e.q) ≤ ϵ − e.r                   // e.q entirely in q
16             then
17                 Insert all data vectors of left sub-tree e.left in R
18                 search-and-crack(ℝ^D, q, ϵ, e.right, R)
19             else
20                 search-and-crack(ℝ^D, q, ϵ, e.left, R)
21                 if d(q,e.q) > e.r−ϵ                  // q not entirely in e.q
22                 then
23                     search-and-crack(ℝ^D, q, ϵ, e.right, R)
```

of similarity query results after extending to support multi-vectors, the radius of each non-leaf entry is calculated based on $d_{1.0}$, while the distance calculations between the query vector and the data vectors are based on d_W. Here, $d_{1.0}$ is a special case of d_W, i.e., $d_{1.0}(V_1, V_2) = \sum_{i=1}^{D} d(v_1[i], v_2[i]) * w_i$ with $w_i = 1 (1 \leq i \leq D)$. In addition, in our A3V-tree index, we use the median distance value of $d_{1.0}(e.q, V) (V \in e.O)$ as the radius of e to construct a balanced A3V-tree index.

1) Range Search: To ensure RS's correctness, apply the pruning and validation lemmas as follows.

Lemma 1. *Given a non-leaf node e, a query vector q, and a query radius ϵ, if $d_W(e.q,q) > e.r + \epsilon$, we can prune the left sub-tree of e.*

Proof. According to the construction phase of A3V-tree index, for all data vectors v in the left sub-tree of e (i.e., $\forall V \in O_l$), $d_{1.0}(V, e.q) \leq e.r$. According to the definition of d_W, we get $d_W(V, e.q) \leq d_{1.0}(V, e.q) \leq e.r$. Thus, if $d_W(e.q,q) > e.r + \epsilon$, then $d_W(e.q,q) > d_W(V, e.q) + \epsilon$. According to the triangle inequality, we can get $d_W(v,q) \geq d_W(e.q,q) - d_W(V, e.q) > \epsilon$. Thus, V cannot be the range search result via its definition, and the proof completes. □

Algorithm 2: kNNS

```
   Input: data array ℝ^D, query vector q, k
   Output: the k nearest neighbors
1  Procedure KNNSearch(ℝ^D, q, k):
2      searchPQ ← PriorityQueue⟨dist, node⟩ // small heap, guide search
3      resultPQ ← PriorityQueue⟨dist, pid⟩ // big heap
4      searchPQ.push([0, root])
5      while !searchPQ.empty() and (resultPQ.size() < k or
         searchPQ.top().dist < resultPQ.top().dist) do
6          e ← searchPQ.top().deheap()
7          if e is a leaf node then
8              foreach data vector V in e do
9                  if d_W(V, q) < resultPQ.top().dist then
10                     resultPQ.push([d_W(V, q), V]) // update result
11                     if resultPQ.size() ≥ k then
12                         resultPQ.pop()
13             compute median distance m_dist of d_1.0(V, q) (∀V ∈ e)
14             CrackInTwo(e.left, e.right, q, m_dist, ℝ^D)
15         else
16             leftMinDist ← max(0, d_W(e.q, q) − e.ϵ) // lemma4
17             if leftMinDist ≤ resultPQ.top().dist then
18                 searchPQ.push(leftMinDist, e.left)
19             rightMinDist ← max(0, min_w × e.r − d_W(e.q, q)) //
                 lemma5
20             if rightMinDist ≤ resultPQ.top().dist then
21                 searchPQ.push(rightMinDist, e.right)
```

Lemma 2. *Given a non-leaf node e, a query vector q, and a query radius ϵ, if $d_W(e.q,q) \leq \epsilon - e.r$, data vectors in the left subtree of e can be validated to the range search result.*

Proof. According to the construction phase of A3V-tree index, for all data vectors V in the left sub-tree of e (i.e., $\forall V \in e.left$), $d_{1.0}(V, e.q) \leq e.r$, and then we can get $d_W(V, e.q) \leq d_{1.0}(V, e.q) \leq r$. If $d_W(e.q,q) \leq \epsilon - e.r$, according to the triangle inequality, $d_W(V,q) \leq d_W(q, e.q) + d_W(V, e.q) \leq d_w(q, e.q) + e.r \leq \epsilon$. Thus, V is the query result by definition without further verification, and the proof is completed. □

Lemma 3. *Given a non-leaf node e, a query vector q, and a query radius ϵ, if $min_W * e.r - d_W(q, e.q) \geq \epsilon$, then the right sub-tree of e can be pruned, where $min_W = min_{i=1}^{D} w_i$.*

Proof. According to the construction phase of A3V-tree index, for all data vectors V in the right sub-tree of e (i.e., $\forall V \in e.right$), $d_{1.0}(V, e.q) > e.r$, and thus, $d_W(V, e.q) \geq min_W * d_{1.0}(V, e.q) > min_W * e.r$. According to the triangle inequality, $d_W(q, V) \geq d_W(V, e.q) - d_W(q, e.q) > min_W * e.r - d_W(q, e.q)$. If $min_W * e.r - d_W(q, e.q) \geq \epsilon$, then we can get $d_W(q, V) > \epsilon$, and thus, v cannot be the range search result. The proof completes. □

The range search algorithm on A3V-tree index is similar to that on AV-tree index (depicted in Algorithm 1), and thus, is omitted here. The only difference is that, we use Lemma 1 in line 11, Lemma 2 in line 15, and Lemma 3 in line 21.

2) k Nearest Neighbor Search: To ensure the correctness of kNNS, two pruning lemmas are present below.

Lemma 4. *Given a non-leaf node e and a query vector q, assume NND_k is the current k-th nearest neighbor distance to q, if $d_W(q, e.q) - e.r > NND_k$, then the left sub-tree of e can be pruned.*

Proof. According to the construction phase of A3V-tree index, for all data vectors V in the left sub-tree of e (i.e., $\forall V \in e.left$), $d_{1.0}(V, e.q) \leq e.r$, and thus, $d_W(V, e.q) \leq d_{1.0}(v, e.q) \leq e.r$. If $d_W(q, e.q) - e.r > NND_k$, then we can get $d_W(V, e.q) > NND_k$, and thus, V cannot be the kNNS result. The proof completes. □

Lemma 5. *Given a non-leaf node e and a query vector q, assume NND_k is the current k-th nearest neighbor distance to q and $min_w = min_{i=1}^{D} W_i$, if $min_W * e.r - d_W(q, e.q) > NND_k$, then the right sub-tree of e can be pruned.*

Proof. According to the construction phase of A3V-tree index, for all data vectors V in the right subtree of e (i.e., $\forall V \in e.right$), $d_{1.0}(V, e.q) > e.r$, and then we can get $d_W(V, e.q) \geq min_W * d_{1.0}(V, e.q) > min_W * e.r$. According to the triangle inequality, $d_W(q, V) \geq d_W(V, e.q) - d_W(q, e.q) > min_W * e.r - d_W(q, e.q)$. If $min_W * e.r - d_W(q, e.q) > NND_k$, then $d_W(q, V) > NND_k$, and thus, V cannot be the kNNS result. The proof completes. □

Algorithm 2 present the pseudo-code of kNNS and A3V-tree index construction. It initializes two priority queues searchPQ and resultPQ to store the A3V-tree nodes and the candidate results respectively (lines 2–3), and then inserts the root node into searchPQ (line 4). After that, a while loop is conducted (lines 5–20) until searchPQ is empty or kNN results are found. For each loop, the top node e in searchPQ is popped and visited. If e is a leaf node, we compute the distance between each data point V and the query vector, and update the kNN result set if $d_W(V, q) < resultPQ.top().dist$ (lines 7–11). In addition, the CrackInTwo function uses the query vector q to split e into two subnodes. If e is a non-leaf node, Lemmas 4 and 5 are used to prune the leaf and right sub-tree of e. If the sub-tree cannot be pruned, they are inserted into searchPQ (lines 16–21).

Complexity Analysis. We assume that a crack will not occur if the data size in any leaf node does not exceed θ, the sum of all dimensions for multi-vectors is M, and the total number of data vectors is n. Our improvements do not change the time and space complexity of the original AV-tree index [49]. Therefore, the worst time complexity is $O(n)$ for a query. However, after sufficient query request adjustments, the time complexity of a single query converges to that of using a fully built VP-tree [59](which partitions data around a selected point to efficiently perform nearest neighbor searches), i.e., $O(1+log(n/\theta))$. The space cost $O(M*n+M*n/\theta)$, where it takes $O(M*n)$ to store the data vectors and takes $O(M*n/\theta)$ to store the tree nodes.

Discussion. In the current design, we haven't detailed index modification of vector dataset (not query vectors). A feasible way is treating new data as index delta data. After sufficient data, do a regular rebuild. For deletions, use lazy marking and periodically rebuild. Merging during search. Updates can be seen as a combination of deletions and inserts.

B. Optimizations of A3V-tree Index

1) Threshold A3V-tree Index: A3V-tree index constructs the indexes according to the query vectors. However, it ignores the characteristics of similar query vector scenarios, and thus, its pruning ability degrades rapidly especially in high dimensional

space. To fully exploit the similar characteristics of query vectors, we consider two aspects: i) constructing the A3V-tree index based on similar query vectors and ii) performing similarity search on a A3V-tree constructed using similar query vectors. Fig. 3 provides A3V-tree examples to illustrate two scenarios. More specifically, only the ball regions (i.e., centered of $e.q$ with the radius $e.r$) of left sub-tree nodes e are illustrated for simplification. As shown in Fig. 3, if we construct the A3V-tree index using similar query vectors (i.e., q_1, q_2 and q_3 in Fig. 3(a)), we can get more compact ball regions for index nodes compared to the A3V-tree index built on random query vectors (i.e., q_1, q_4 and q_5 in Fig. 3(b)). For a new query vector q, if we use the A3V-tree built on query vectors similar to q (cf. Fig. 3(a)), the pruning ability becomes stronger than using the A3V-tree built on query vectors based on query vectors dissimilar to q (cf. Fig. 3(b)).

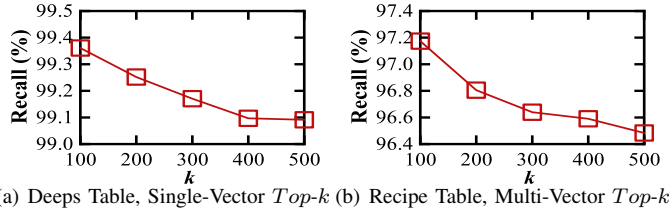
Motivated by these, we construct a threshold A3V-tree index. Given a threshold δ, when a new query vector q arrives, if the distance $d_W(q, e.q)$ between q and the current A3V-tree root node query vector $e.q$ exceeds the threshold, a new A3V-tree is constructed. Thus, in a threshold A3V-tree index, multiple A3V-trees exist. If $d_W(q, e.q)$ does not exceed the threshold, we find a nearest A3V-tree (i.e., the A3V-tree having the smallest distance between its root node query vector to the query vector), and use q to perform the search and crack this nearest A3V-tree. To accelerate the search of the nearest A3V-tree, a meta HNSW [32] index is built on query vectors of root nodes of all the A3V-trees.

δ Computation. It is crucial to determine the threshold δ. During the index construction, we randomly sample c (In fact, we recommend setting the value to 100. Based on our experiments across diverse datasets and scales, this choice consistently achieves good efficiency, suggesting its suitability for uniform distance calculation.) data vectors. Thus, the threshold δ is computed as the average distance between the sample data vectors and multiplied by a parameter σ:

$$\delta = \sigma * \frac{1}{\binom{c}{2}} \sum_{i=1}^{c-1} \sum_{j=i+1}^{c} d_W(V_i, V_j) \tag{1}$$

2) Top-k-based Lower Bound Optimization: Despite the effectiveness of the threshold A3V-tree index (improved by considering similar query vectors), its performance still degrades rapidly in high dimensional space, especially for kNN queries, due to the curse of dimensionality. To address it, we adopt a *top-k*-based adaptive optimization strategy. Consequently, when the dimensionality becomes excessively high, we can sacrifice the precision to enhance the performance. The A3V-tree index could be further optimized for kNNS. In fact, we found that 0 is not always the best choice for lower bound distance estimation and filtering in line 17 of Algorithm 2. Instead, we adopt a *top-k* based optimization method, i.e., the lower bound distance is calculated as:

$$dist_{adaptive} = \theta' * \sum_{i=1}^{N_q} distance_{k\text{-}th}(q_i)/N_q, \tag{2}$$



(a) Deeps Table, Single-Vector $Top\text{-}k$ (b) Recipe Table, Multi-Vector $Top\text{-}k$

Fig. 4: *Recall of kNNS vs. k*

Algorithm 3: HybridIndexAlgorithm

```
Input: data array ℝ^D,query vector q,k
Output: the k nearest neighbors
1  Procedure HybridIndexSearch(data array ℝ^D,q,k):
2      Find the nearest Av-Tree t to q
3      if t is none or distance is overhead then
4          create a new tree using q
5          AsyncHnsw(q,k)
6          return SyncHnsw(q,k)
7      else
8          if t's build points are not enough then
9              AsyncHnsw(q,k)
10             return SyncHnsw(q,k)
11         return KNNSearch(ℝ^D,q,k)
```

where N_q is the number of queries, $distance_{k\text{-}th}(q_i)$ denoting the k-th distance of the i-th kNN query, and θ' is the hyperparameter. Thus, line 18 of Algorithm 2 is replaced:

$$rightMinDist = max(dist_{adaptive}, n.\epsilon - d(e.q, q)). \tag{3}$$

However, this estimation leads to approximate kNNS. In our paper, we enable this optimization when the dimensionality reaches a certain upper limit (i.e., 1024).

We conduct 1000 queries on the Recipe table for multi-vector *top-k* and the Deeps table for single-vector *top-k* (the dataset descriptions can be found in §V-A2), and vary k value from 100 to 500. Fig. 4 reports the average recall of 1000 queries. As observed, the recall of kNNS is above 99% for single-vector *top-k* and above 96% for multi-vector *top-k*, and thus, the loss of precision of this optimization is minimal.

C. Hybrid Index Implementation

Although the two optimizations above can improve the A3V-tree index performance, the efficiency of A3V-tree index during the initial construction phase is nearly equivalent to linear scanning. Motivated by it, QBase implements a hybrid index combining HNSW [27] and A3V-tree index. For simplify, we adopt the strategies in VBase [27] when implementing HNSW for similarity queries on vectors. Fig. 5 presents the hybrid index architecture of our QBase system, which consists of ① index metadata and shared vector data, ② HNSW index, and ③ A3V-tree index. Finally, the hybrid index can be stored in persistent storage.

MetaIndex and Shared Vector Data. The **MetaIndex** is primarily used to maintain some mapping metadata, including the location information for index persistence and the storage location information for vector data. The **Shared Vector Data** stores data vectors, which are shared between the HNSW index and the A3V-tree index. We use columnar storage to store shared vector data $V = \{v_i | 1 \leq i \leq D\}$. Therefore, following

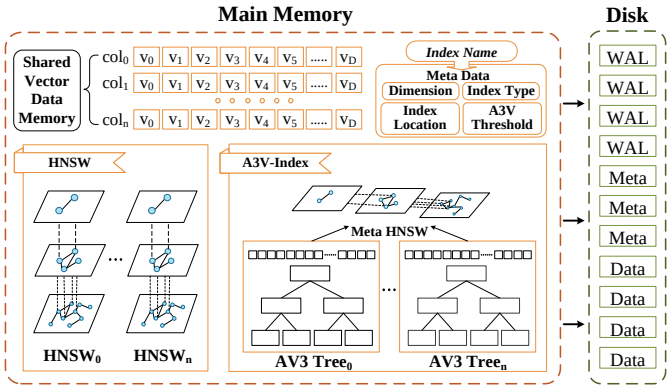


Fig. 5: The Hybrid Index Architecture in QBase

VBase [27], a HNSW index is constructed for single-vectors $\forall v_i \in V$. However, for the MetaHNSW index, when dealing with multi-vectors, we use VectorRecord (a combination composed of one or more vectors) in row-major format, i.e., D single vectors are stored in a VectorRecord. Existing systems such as Milvus [23], LanceDB, and Cassandra [60] support adopt the columnar storage [61], as they primarily support queries using indexes that only support single-vector. Note that, a columnar storage format is more suitable for single-vectors. However, considering that our index is designed for multi-vectors, we use row storage for query vectors in the non-leaf nodes, while using the column storage for shared data vectors when visiting leaf node data. Although we use a row-based approach to store multiple vectors, single vector queries require constructing an index specific to the single vector rather than using the multi-vector index to ensure query efficiency.

HNSW and A3V-tree Indexes. HNSW index is created on the vector dataset in advance, while the A3V-tree index is created during the query processing. Note that, the query processing on HNSW is same as that of VBase [27], and thus, is omitted. During the initial phase of A3V-tree, the similarity search degrades to the linear search, and thus, HNSW index is used to accelerate the search while constructing the A3V-tree index asynchronously. A cost model is designed (to be detailed in Section IV-B1) to select a proper index for each query vector. In addition, we also design a **IndexNode Allocator**, which is responsible for the recycling and allocation of nodes. For the continuous allocation of new nodes, we utilize atomic variables. During the index search, we employ index-level write locks to ensure the data consistency. When a large number of A3V-trees are generated, the IndexNode Allocator triggers the recycling mechanism for A3V-trees, which selects the A3V-tree with the least number of hits for recycling.

Although we use the HNSW index to accelerate the vector search for the low initialization efficiency of A3V-tree, the construction cost of A3V-tree initial crack is still high. In other words, when the search is finished via the HNSW index, the asynchronous A3V-tree crack cannot be finished at the same time. Thus, when a new search arrives, A3V-tree is unavailable for query processing, which degrades the search efficiency. To address it, in the initial phase, we asynchronously use the existing HNSW to perform separate searches. In asynchronous tasks, we increase the ef_search parameter of HNSW to obtain more search results. It reduces the query efficiency, but yields better result quality. Meanwhile, these results provide sufficient data to assist in the initialization of the A3V-tree index crack, i.e., the query result vectors are assigned to the left sub-tree, while the remaining vectors are assigned to the right sub-tree. However, this feature isn't always on. It is only selectively activated when the dataset scale is large, as the initialization of the A3V-tree is computationally time-consuming. This feature can be optionally enabled.

For Lemmas 1–4 on A3V-tree, we use $d_{1.0}$ to calculate the node radius $e.r$ for pruning and validation. Thus, when the weight ratios for multi-vectors are balanced, the pruning and validation lemmas work well. Motivated by this, when the weight ratios are unbalanced, we opt for HNSW by using the $weighted{-}round{-}robin$ search strategy. Assume that a multi-vector V contains two single vectors v_1 and v_2 with assigned weights w_1 and w_2 respectively. Unlike the traditional round-robin approach, where each HNSW index is iterated sequentially, the $weighted{-}round{-}robin$ method retrieves the first index for w_1 results before moving to the second index, and then retrieves the second index for w_2 results before switching back to the first index. This process continues iteratively until the search is complete. This method can be adapted to mutli-vectors with more than two single vectors.

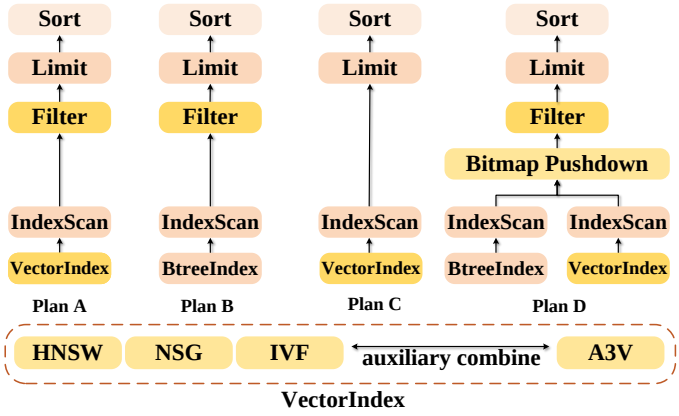
Disk Storage. We mainly store three types of files, i.e., *WAL, Meta, and Data* in disk. Since we primarily use in-memory index, the system persists the corresponding metadata and indexes when it shuts down, enabling normal operation upon the next startup. In the event of a system crash, we use WAL logs for recording. Specifically, for the A3V-tree index's peculiarity, it involves write operations during queries, thus requiring writing to WAL logs. Writing Meta data can be viewed as different Meta serving as snapshots. During system recovery, Meta information is used for recovery, and then WAL logs are replayed to restore the complete system. These processes adopt asynchronous batch execution to minimize the performance degradation. Additionally, we regularly clean up obsolete logs and Meta data. Note that, Data mainly stores index structure data and original vector data.

IV. QBase Implementation

A. Supported Queries

According to the two above search types, a general vector database system typically supports five categories of queries [27]:

- **S1: Single-Vector *Top-*k**: The query involves simply retrieving the *Top-*k results from a single-vector dataset.
- **S2: Single-Vector *Top-*k plus scalar attribute filter**: Based on Single-Vector *Top-*k, we need to further filter the results via additional attribute fields associated with the vectors, resulting in the final *Top-*k results.
- **S3: Multi-Vector *Top-*k**: Similar to **S1**, but instead of retrieving single-vectors, multi-vectors are jointly searched.



VectorIndex is hybrid index in fact. It can contains any main-stream vector index, and a3v index can be an auxiliary combination. But for now, **QBase** supports only HNSW.

Fig. 6: Query Plan

- **S4: Multi-Vector *Top-*k plus scalar attribute filter**: Similar to **S2**, but instead of retrieving single-vectors, multi-vectors are jointly searched.
- **S5: Vector Range Filter**: The query finds all vector results within a certain distance range from the query vector.

Furthermore, we elect to extend **S5** to accommodate **S6: Multi-Vector Range Filter**, which is useful in real-life analytics scenarios. For example, in multimodal search [62]–[65], a variety of data types (such as images, text, and audio) need to be searched based on their features within the specified range. Specifically, an e-commerce website might allow users to search for products by uploading images and entering text descriptions. These different modal features can be weightedly combined to find products that match all given features. To sum up, our system supports all the above six types of queries.

B. Query Optimizer

To ensure the high efficiency of QBase in query execution, we design a query optimizer that guarantees the selection of the most efficient execution method. We have developed four distinct query plans and provided a cost model to evaluate their overhead, enabling the dynamic selection of the optimal solution. Unlike traditional query schemes, our approach adopts a hybrid auxiliary index evaluation method. In our query plans, A3V-tree index combines existing vector index, considering factors of vector computation, query selectivity and candidate indexes during cost estimation and query plan selection.

1) Query Plan: QBase provides a cost-based query plan selection method for vector search queries, allowing us to select the most cost-effective query plan for retrieval during vector retrieval. In Fig. 6, we present four query plans for vector retrieval (The VectorIndex is, in fact, a hybrid index. It can incorporate any mainstream vector indexing method, with the A3V-tree index serving as an auxiliary component for enhanced combination and functionality). But for now, QBase supports only HNSW. Note that, the brute-force search scheme is not illustrated, which doesn't need any index.

PlanA (Vector Index Scan With Filter). When executing **S2 and S4**, PlanA serves as a candidate, which uses a hybrid index consisting of HNSW and A3V-tree index. When the hybrid index hits the HNSW index, the iterative optimization strategy of VBase [27] is used. When it hits the A3V-tree index, the value of k is expanded to $k/selectivity$ to ensure high result quality after the attribute filtering, where the attribute $selectivity$ can be easily obtained via the database optimizer. Since the extension of k will degrade the query efficiency, when $selectivity$ is relatively small, HNSW is a better solution. For PlanA, the result is not accurate.

TABLE II: Schema of Deeps Table

Column Name	Data Type	Example
id	integer	10
number	integer	300
l_comment	text	p furiously special ...
deep	vector(96) float32	[0.10090, ..., 0.04333]

PlanB (Vector Brute-force Scan With Filter). When executing **S2** and **S4**, PlanB is also a candidate option. When the attribute selectivity is very low, an efficient strategy is to first carry out attribute filtering with BtreeIndex to get the candidates and then acquire the final *top-*k result. PlanB can yield accurate results.

PlanC (Vector Index Scan). For **S1, S3, S5**, and **S6**, a hybrid vector index is used. If A3V-tree index is hit, return accurate results; otherwise, return approximate results.

PlanD (Vector&Btree Index BitmapPushdown Scan). For **S2 and S4**, a pre-filtering method uses BtreeIndex for attribute filtering to get bitmap, then pushes it down to vector index. Bitmap shows which data vector meets attribute filtering. PlanD is used only when hitting A3V-tree index possible. Unlike PlanA's k extension, PlanD combines bitmap and A3V-tree index for high query efficiency.

2) Cost Model: Regarding the cost estimation t_{hnsw} for HNSW, we adopt VBase's method [27]. Therefore, we only need to focus on the cost estimation for A3V-tree index.

Vector Distance Computation Cost. By considering the scalar filtering, the number of results that our index query should return is N_{res}/α (i.e., where N_{res} is the number of data points returned without scalar filtering and α is the scalar attribute filtering rate). As in PlanA, k needs to be expanded to $k/selectivity$ ($selectivity = \alpha$). Thus, the cost of distance calculations during the search is $N_{res}/\alpha * t_v$ (i.e., where t_v is the cost of calculating the distance for a single-vector).

IndexNode Access Cost. Apart from the cost of vector calculations, the main overhead during the index search process lies in accessing index nodes, the cost is $C * t_v$ (i.e., where C is the number of index nodes accessed during search).

Finally, considering the cost of selecting the A3V-tree index, we can obtain the scan cost of using the A3V-tree index as (here, t_{sel} is the cost of selecting the optimal A3V-tree index):

$$Cost1_{a3v\_index} = N_{res}/\alpha * t_v + C * t_v + t_{sel}. \quad (4)$$

The above estimation is for the cost of A3V-tree index after processing a certain number of queries. We observe that the quality of the index improves significantly when the A3V-tree index contains at least 2∼6 query points. However, in the early stage, the index performance is nearly equivalent to

linear scanning. Thus, during the initial phase, the query cost of the A3V tree index degrades to:

$$Cost2_{a3v\_index} = n * t_v + t_{sel}. \quad (5)$$

The actual query cost is:

$$\min(Cost1_{a3v\_index}, t_{hnsw}) \ or \ \min(Cost2_{a3v\_index}, t_{hnsw}). \quad (6)$$

V. Experiments

In this section, we first provide the experimental settings, and use research questions (RQs) to guide the experiments.

RQ1: How does QBase perform on query recall and efficiency vs. state-of-the-art vector search systems (§V-B)?

RQ2: How sensitive is QBase to parameters k, ϵ, σ, and the distance weight ratio w_i (§ V-C)?

RQ3: How scalable is QBase to the size of datasets (§V-D)?

RQ4: How do the Hybrid Index and the query selectivity impact QBase (§ V-E)?

A. Experimental Settings

We use three datasets for three corresponding tests: (1) ***Experiment 1***: 4 query types on a real single-vector dataset; (2) ***Experiment 2***: 4 query types on a real multi-vector dataset; (3) ***Experiment 3***: 8 query types on synthetic datasets. For each experiment, we design SQL query examples for the query types in Section §IV-A. We choose $L2$-norm for distance calculation. Although other triangle-inequality-satisfying metrics are supported, we only show $L2$-norm in experiments. Each table has a dataset size of 1M.

*1) **Experiment 1**:* To validate the effectiveness of QBase under single-vector queries, we use the "l_comment" field from the lineitem table of TPCH [66] as the string attribute; for numerical type field, we follow the VBase's approach, generating random values in the range [1, 10000]; and we synthesize the Deeps table by combining data from BigAnn-Benchmark [44]'s Deep dataset.

For single-vector, we set SQL queries for **S1**, **S2**, and **S5**.

Q1-1: Single-Vector *Top-k*

```
select id from Deeps order by ${p_deep} <->
    deep limit 50;
```

Q1-2(a): Single-Vector *Top-k* + Numeric Filter

```
select id from Deeps where popularity <= ${
    p_popularity} order by ${p_deep} <-> deep
    limit 50;
```

Q1-2(b): Single-Vector *Top-k* + String Filter

```
select id from Deeps where l_comment not like
    '%${p_l_comment}%' order by ${p_deep} <->
    deep limit 50;
```

Q1-3: Vector Range Filter

```
select id from Deeps where ${p_deep} <-> deep
    < ${D1};
```

Parameter Settings for Experiment 1. We set the parameter k of *Top-k* to 50 by following VBase. For the numerical filtering, we randomly set the selection rate for each query from 0 to 1; while for the string filtering, the selection rate is set to around 0.9. The attribute filtering fields are randomly sampled from the dataset. The radius for Q1-3 is set to 0.6.

TABLE III: Schema of Recipe Table

Column Name	Data Type	Example
recipe_id	identifier	1
images	list of strings	["images/0.jpg", ...]
description	text	[ingredients] + [...]
images_emb	vector(1025) float32	[0.0421, ..., 0.0273]
description_emb	vector(1025) float32	[0.0056, ..., 0.0034]
popularity	integer	300

*2) **Experiment 2**:* To validate the effectiveness of real multi-vector datasets, we utilize the **Recipe Table** from VBase [27], as depicted in Table III. We adopt a SQL query design approach similar to VBase but focus on multi-vectors.

Q2-1: Multi-Vector *Top-k*

```
select id from Recipe order by ${w1} * ${
    p_images_embedding} <-> images_embedding +
     ${w2} * ${p_description_embedding} <->
    description_embedding limit 50;
```

Q2-2(a): Multi-Vector *Top-k* + Numberic Filter

```
select id from Recipe where popularity <= ${
    p_popularity} order by ${w1} * ${
    p_images_embedding} <-> images_embedding +
     ${w2} * ${p_description_embedding} <->
    description_embedding limit 50;
```

Q2-2(b): Multi-Vector *Top-k* + String Filter

```
select id from Recipe where description <= ${
    p_description} order by ${w1} * ${
    p_images_embedding} <-> images_embedding +
     ${w2} * ${p_description_embedding} <->
    description_embedding limit 50;
```

Q2-3: Multi-Vector Range Filter

```
select id from Recipe where ${w1} * ${
    p_images_embedding} <-> images_embedding +
     ${w2} * ${p_description_embedding} <->
    description_embedding < ${D2}$;
```

Parameter Settings for Experiment 2. We directly use the attribute filter test values from VBase, ensuring the "popularity" column covers the entire range and the "description" column maintains a filtering rate of 0.9. *images_embedding* and *description_embedding* columns are generated via the AMQ [67] method that can use L_2-norm to measure the distance between two embeddings. For multi-vectors, each weight is randomly generated in the range [0.5, 0.7]. The search radius in Q2-3 (i.e., $D2$) is set to 1.1.

*3) **Experiment 3**:* . We synthesize a new experimental table called the Composite Table, as depicted in Table V. We obtain three 1M-scale vector datasets from BigAnn-Benchmark [44]. For the attribute fields, we use the same datasets as in **Experiment 1**. The purpose of **Experiment 3** is to conduct a comprehensive experiment on query types S1 to S6 and cover a more diverse range of datasets.

Q3-1: Single-Vector *Top-k*

```
select id from Composite order by ${p_deep}
    <-> deep limit 50;
```

TABLE IV: Schema of Composite Table

Column Name	Data Type	Example
id	integer	10
number	integer	300
l_comment	text	p furiously special ...
bigann	vector(128) uint8	[1, 2, 3, ..., 128]
ssnpp	vector(256) uint8	[1, 105, ..., 234]
deep	vector(96) float32	[0.097, ..., 0.097]

TABLE V: Schema of Food Table

Column Name	Data Type	Example
id	integer	10
salt	float	2600
main_category_string	text	Plant-based...
description_embedding	vector(100) float32	[-0.1837,0.1623,...]
main_category_embedding	vector(96) float32	[0.2006,0.0751,...]

Q3-2(a): Single-Vector *Top-k* + Numeric Filter

```
select id from Composite where popularity <= ${p_popularity} order by ${p_deep} <-> deep limit 50;
```

Q3-2(b): Single-Vector *Top-k* + String Filter

```
select id from Composite where l_commnet not like '%${p_l_comment}%' order by ${p_deep} <-> deep limit 50;
```

Q3-3: Vector Range Filter

```
select id from Composite where ${p_deep} <-> deep < ${D3};
```

Q3-4: Multi-Vector *Top-k*

```
select id from Composite order by w1 * ${p_bigann} <-> bigann + w2 * ${p_ssnpp} <-> ssnpp + w3 * ${p_deep} <-> deep limit 50;
```

Q3-5(a): Multi-Vector *Top-k* + Numeric Filter

```
select id from Composite where popularity <= ${p_popularity} order by w1 * ${p_bigann} <-> bigann + w2 * ${p_ssnpp} <-> ssnpp + w3 * ${p_deep} <-> deep limit 50;
```

Q3-5(b): Multi-Vector *Top-k* + String Filter

```
select id from Composite where l_comment not like '%${p_l_comment}%' order by w1 * ${p_bigann} <-> bigann + w2 * ${p_ssnpp} <-> ssnpp + w3 * ${p_deep} <-> deep limit 50;
```

Q3-6: Multi-Vector Range Filter

```
select id from Composite where w1 * ${p_bigann} <-> bigann + w2 * ${p_ssnpp} <-> ssnpp + w3 * ${p_deep} <-> deep < ${D4};
```

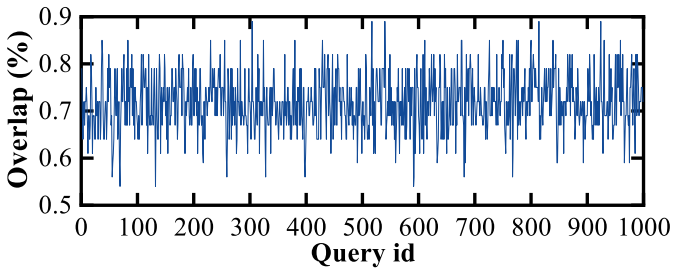


Fig. 7: Query Result Overlap

Parameter Settings for Experiment 3. The weights w_1 and w_2 are set using the same configuration as **Experiment 2**, while w_3 is set to 1.0. The attribute query fields were still generated using the same method as **Experiment 1**. For $top{-}k$ search, k is set to 50. For the range search radius ϵ, it is set to to 65000 in Q3-3 (i.e., $D3$ = 65000), while set to 20000 in Q3-6 (i.e., $D4$ = 200000).

Query Generation. For each type of query SQL, we execute 10,000 queries. First, we randomly select 10 data points, and then for each point, we randomly generate 1000 similar vector query points. Each batch of query vector points ensures that the result intersection is from 70% to 90%. Fig. 7 displays the distribution of overlap for adjacent query results in Q1-1. Here, $overlap = |s_1 \cap s_2| * 2/(|s_1| + |s_2|)$, where s_1 and s_2 denote the result set for two different queries.

Evaluation Metrics. To evaluate the efficiency, the average, medidan and p99 query latency of 10000 queries are used. Here, p99 is the 99th percentile latency from the execution results. To evaluate the quality of query results, recall is used. Here, $recall = |r_1 \cap r_2|/|r_2|$, where r_1 denoting the query result and r_2 denoting the ground-truth query result.

Baselines. To verify the performance of our proposed system QBase, we compare it against (i) two exsiting state-of-the-art vector database systems VBase [27] and Milvus [23], and (ii) Postgres [68] with the brute-force search solution.

All experiments were conducted on a machine equipped with one Intel(R) Xeon(R) CPU E5-2650v4 @ 2.20GHz processor, featuring 12 cores, and 128GB of RAM. The index used for experimental testing is HNSW [32] which is supported by both Milvus and VBase. We adopt the same HNSW parameter settings as VBase ($M = 16, efc = 200, efs = 64$).

B. Overall Performance

1) Experiment 1: Single-Vector Search.: Table VI presents the results in all performance metrics of QBase, PostgreSQL, VBase, and Milvus, while Fig. 8 presents the query latency of each query of 10000 queries. As observed, QBase outperforms all baseline systems in terms of both efficiency and accuracy in all cases. This is because, QBase uses similar query vectors to construct more compact index, and selects a A3V-tree index with similar root node query vector to achieve better pruning ability and higher query accuracy. Note that, Milvus struggles particularly in Q1-2(b) (i.e., *top-k* query with string filter), as it generates bitmaps for attribute values and then performs search, which consumes a significant amount of time for strings. However, there are performance fluctuations of QBase every 1000 queries. Because we generate 1000 similar queries for each sampled data vector, performance fluctuation occurs when a new dissimilar query arrives, further highlighting QBase's superior performance for similar query vectors. Note that, fluctuations also exist for Milvus. Milvus caches *top-1* result set from previous queries and uses cached hit data point as starting search point for vector index. However, this approach doesn't always yield benefits.

2) Experiment2: Multi-Vector Search.: Table VII presents the results in all performance metrics of QBase, PostgreSQL,

TABLE VI: Results of Q1-1 to Q1-3 on Experiment1 (Latency: ms)

System	Q1-1				Q1-2(a)				Q1-2(b)				Q1-3			
	Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)		
		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th
PostgreSQL	1	612.5	611.3	653.7	1	416.5	405.4	625.6	1	643.2	644.2	699.9	1	1541	1536	1610
Milvus	0.90	2.58	2.58	3.2	0.94	16.43	15.71	36.56	0.89	2502	2490	2721	0.93	4.6	2.55	25.24
VBase	0.95	0.94	0.85	1.53	0.94	1.45	0.59	21.06	0.95	1.01	0.93	1.69	0.975	1.01	0.48	6.77
QBase	**0.98**	**0.39**	**0.31**	**0.8**	**0.97**	**0.76**	**0.45**	**8.16**	**0.98**	**0.61**	**0.56**	**1.17**	**0.98**	**0.63**	**0.33**	**2.73**
Qdrant	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
pgvector	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**

TABLE VII: Results of Q2-1 to Q2-3 on Experiment2 (Latency: ms)

System	Q2-1				Q2-2(a)				Q2-2(b)				Q2-3			
	Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)		
		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th
PostgreSQL	1	6050	6086	7548	1	3179	3003	7497	1	6029	6091	7544	1	6260	6165	8042
Milvus	0.54	2027	1999	2485	0.09	2573	2533	3238	0.20	2573	2533	3241	-	-	-	-
VBase	0.63	150.0	143.6	275.9	0.71	164.5	173.2	387.1	0.63	169.2	162.5	309.4	-	-	-	-
QBase	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
Qdrant	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
pgvector	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**

TABLE VIII: Results of Q3-1 to Q3-6 on Experiment3 (Latency: ms)

System	Q3-1				Q3-2(a)				Q3-2(b)				Q3-3			
	Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)		
		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th
PostgreSQL	1	1101	945.5	2256	1	903.8	846.9	1957.4	1	1177	978	2348	1	1424	1410	1663
Milvus	0.94	2.47	2.46	3.27	0.94	16.99	16.5	37.28	0.89	1341	1336	1465	0.82	58.67	59.96	69.25
VBase	0.96	0.60	0.56	0.96	0.94	2.90	0.74	55.71	0.96	0.69	0.65	1.17	0.93	1.35	0.38	2.63
QBase	**0.97**	**0.58**	**0.39**	**0.85**	**0.95**	**1.66**	**0.69**	**5.04**	**0.98**	**0.61**	**0.53**	**1.08**	**0.96**	**0.83**	**0.35**	**2.07**
Qdrant	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
pgvector	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**

System	Q3-4				Q3-5(a)				Q3-5(b)				Q3-6			
	Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)		
		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th
PostgreSQL	1	2159	2158	2220	1	1837	1923	2455	1	1966	1988	2509	1	1063	1068	1121
Milvus	0.69	9245	92301	10336	0.02	8616	8606	9436	0.05	8613	8617	9389	-	-	-	-
VBase	0.87	47.92	47.78	85.02	0.88	99.59	51.90	759.4	0.92	28.94	26.37	63.22	-	-	-	-
QBase	**0.94**	**2.03**	**0.79**	**13.01**	**0.99**	**14.13**	**7.31**	**86.3**	**0.97**	**9.28**	**2.47**	**20.14**	**0.98**	**18.58**	**3.73**	**166.1**
Qdrant	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
pgvector	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**

TABLE IX: Results of Q4-1 to Q4-6 on Experiment4 (Latency: ms)

System	Q4-1				Q4-2(a)				Q4-2(b)				Q4-3			
	Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)		
		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th
PostgreSQL	1	1101	945.5	2256	1	903.8	846.9	1957.4	1	1177	978	2348	1	1424	1410	1663
Milvus	0.94	2.47	2.46	3.27	0.94	16.99	16.5	37.28	0.89	1341	1336	1465	0.82	58.67	59.96	69.25
VBase	0.96	0.60	0.56	0.96	0.94	2.90	0.74	55.71	0.96	0.69	0.65	1.17	0.93	1.35	0.38	2.63
QBase	**0.97**	**0.58**	**0.39**	**0.85**	**0.95**	**1.66**	**0.69**	**5.04**	**0.98**	**0.61**	**0.53**	**1.08**	**0.96**	**0.83**	**0.35**	**2.07**
Qdrant	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
pgvector	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**

System	Q4-4				Q4-5(a)				Q4-5(b)				Q4-6			
	Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)			Recall	Latency (ms)		
		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th		Avg.	Med.	99th
PostgreSQL	1	2159	2158	2220	1	1837	1923	2455	1	1966	1988	2509	1	1063	1068	1121
Milvus	0.69	9245	92301	10336	0.02	8616	8606	9436	0.05	8613	8617	9389	-	-	-	-
VBase	0.87	47.92	47.78	85.02	0.88	99.59	51.90	759.4	0.92	28.94	26.37	63.22	-	-	-	-
QBase	**0.94**	**2.03**	**0.79**	**13.01**	**0.99**	**14.13**	**7.31**	**86.3**	**0.97**	**9.28**	**2.47**	**20.14**	**0.98**	**18.58**	**3.73**	**166.1**
Qdrant	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**
pgvector	**0.91**	**7.62**	**2.21**	**82.42**	**0.92**	**80.18**	**57.91**	**190.2**	**0.91**	**46.11**	**44.20**	**85.23**	**0.97**	**31.56**	**25.41**	**127.29**

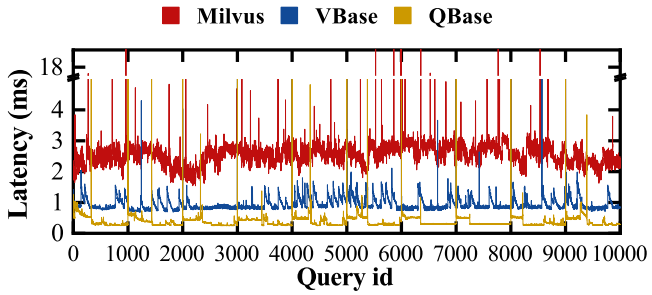


Fig. 8: Single *top-k* Query Latency Trend

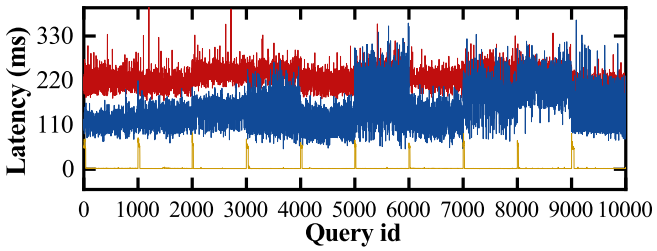


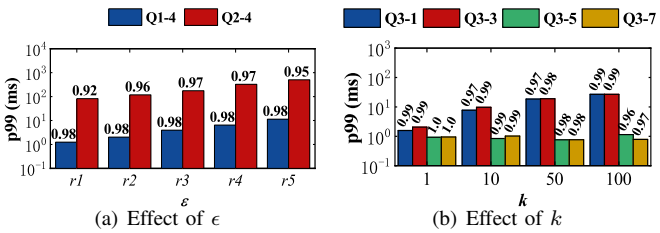
Fig. 9: Multi *top-k* Query Latency Trend

VBase, and Milvus. Additionally, Fig. 9 illustrates the query latency of each query of 10000 queries. Note that, results on Q2-3 of VBase and Milvus are omitted, as they only support 5 types of query while QBase supports 6 types (to be discussed in §VI). In addition, we observe that Milvus performs poorly in multi-vector searches. This is because Milvus uses columnar storage and searches each vector individually. For a multi-vector (i.e., a row with multi-columns), only when all columns are returned by each individual vector index, it will be considered as a candidate of search result [69]. For multi-vector search, Only QBase attains a recall rate above 90% and surpasses Milvus and VBase in efficiency, with an improvement of up to 72.45% against VBase, owing to QBase's multi-vector indexing advantage.

3) Experiment3: Comprehensive Vector Search.: Table IX provides the comprehensive results on all query types Q3-1 to Q3-6 As observed, QBase consistently demonstrates outstanding advantages in both accuracy and performance. Compared to VBase, for query types S1-S5 (i.e., Q3-1 to Q3-5(b)), our QBase's performance can be improved by up to 91.02%. For query type S6 (i.e., Q3-6), Qbase achieves 98% recall with 98.25% performance gain over PostgreSQL.

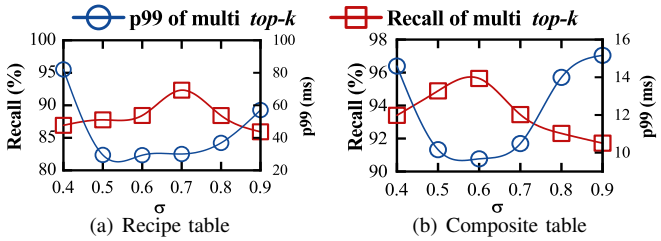
C. Parameter Study

Effect of ϵ. We use Q1-3 and Q2-3 to evaluate the impact of ϵ on the performance of QBase. Fig. 10(a) presents the p99 and recall by varying the search radius ϵ, where the number on th The search radius ranges of Q1-3 are [0.45, 0.5, 0.55, 0.6, 0.65] , while the search radius ranges of Q2-3 are [0.9,



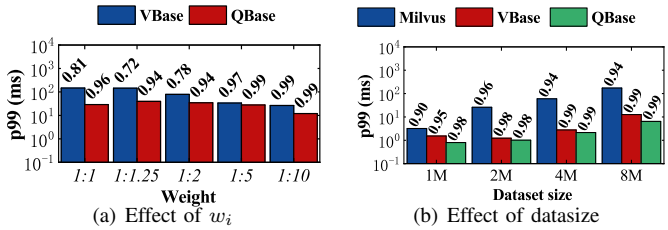
(a) Effect of ϵ (b) Effect of k

Fig. 10: Effect of ϵ & k



(a) Recipe table (b) Composite table

Fig. 11: σ Parameter Impact Difference



(a) Effect of w_i (b) Effect of datasize

Fig. 12: Effect of Weight and Datasize

1, 1.1, 1.2, 1.3]. As observed, the p99 (i.e., the query latency) increases while the recall slightly fluctuates with the growth of ϵ due to larger search space.

Effect of k. We use Q3-1, Q3-2(b), Q3-4, and Q3-6(b) to evaluate the impact of k on the performance of QBase, respectively. Fig. 10(b) presents the p99 and recall by varying k from 1 to 100. As observed, the p99 (i.e., the query latency) increases while the recall slightly drops with the growth of k due to larger search space.

Effect of σ. We use Q2-1 and Q3-4 to validate the impact of σ on the performance of QBase. Note that, σ is used to estimate of δ when constructing threshold A3V-tree index. Fig. 11 presents the recall and p99 by varying σ from 0.4 to 0.9. As observed, the best performance is achieved when σ values are in the range of 0.6 to 0.7. Thus, in our experiments, σ is set to the range of 0.6 to 0.7.

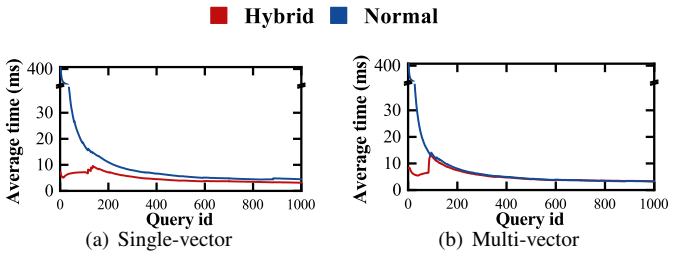
Effect of weight ratio w_i. We use Q2-1 to validate the impact of different weight ratios on the performance of QBase. Fig. 12(a) depicts p99 and recall of QBase by setting two weight ratios to 1:1, 1:1.25 (i.e., 0.8: 1), 1:2 (i.e., 0.5:1), 1:5 (i.e., 0.2:1), and 1:10 (i.e., 0.1:1). As observed, QBase outperforms VBase in all cases. In addition, the query latency decreases when the difference between two weights increases. This is because, when the difference between two weights increases, QBase utilizes weighted-round-robin search strategy as discussed in Section III-C to achieve better search performance.

D. Scalablity of Data Size

Due to the space limitation, we only use Q1-1 to evaluate of scalability of Qbase by varying the Deep [44] dataset size from 1M, 2M, 4M, 8M. Note that, the dataset size is default as 1M in our previous experiments. Fig. 12(b) depicts the p99 and recall results. As observed, our QBase outperforms VBase and Milvus by varying the dataset size, and the query latency linearly increases with the growth of dataset size.

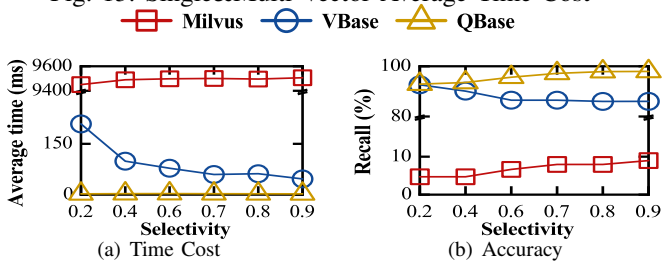
E. Hybrid Index and Selectivity Effect

For hybrid index effect, we chose Q3-1 and Q3-4 to test the advantages of the Hybrid Index in addressing the cold-start problem. We ensured that recall remained above 95%,



(a) Single-vector (b) Multi-vector

Fig. 13: Single&Multi-Vector Average Time Cost



(a) Time Cost (b) Accuracy

Fig. 14: Effect of Selectivity

and Fig. 13 demonstrates the advantage of the Hybrid Index in addressing the cold-start issue at the initial stage of querying when it is enabled. As to selectivity effect, we chose to use Q3-5(a) to demonstrate QBase's adaptability to query filtering rates by fixing the filtering rate of scalar column. We chose to fix the six filtering rates at 0.2, 0.4, 0.6, 0.7, 0.8, and 0.9. It can be seen from the Fig. 14 that QBase is generally superior to Milvus and VBase, indicating QBase has good adaptability to filtering rates and achieves excellent effect.

VI. Related Work

Vector Search Methods. To improve the efficiency of vector search, various indexes such as M-Tree [17], R-Tree [19], and KD-Tree [70] are developed. However, they are designed for exactly similar searches, which are unable to meet the efficiency requirements for extensive high-dimensional vector search. Consequently, many proprietary vector indexes for Approximate Nearest Neighbor Search are developed, which sacrifice accuracy to meet efficiency demands. These indexes typically fall into four major categories: partition-based vector indexes [20], [29], [34], [35], graph-based vector indexes [13], [29]–[33], hash-based vector indexes [21], [50], [51], [71], and tree-based vector indexes [72]. Recently, the AV-tree index constructed via the query vectors is proposed [49] to further improve the query efficiency. Based on similar query vectors, we construct a more compact index with better pruning and propose the optimized A3V-tree.

Vector Database Systems. Many vector database systems have emerged in academia and industry. Table X presents existing mainstream vector database systems and their supported query types, excluding linear scanning methods, **AnalyticDB-V** [26], **Pase** [42], **ElasticSearch** [73], **VBase** [27], **Milvus** [23], **pgvector** [74]. As seen, only QBase can support all six query types simultaneously in query vector similar scenarios, while VBase and Milvus can support five. For queries with a combination of vector and scalar data (**S2**), there are two main methods. The pre-filter method filters scalar data first and then does vector search (like in Weaviate [75]). The post-filter method does vector search first and then scalar filtering (such as in Vearch [41]). Milvus [23] and HQI [47] optimize these by partitioning scalar columns and using pre-filter for quick filtering before vector search. AnalyticDB-V [26] uses a cost-based approach to choose between the two. VBase [27] optimizes the post-filter method by dynamically increasing k for vector search and then doing scalar filtering. To achieve high query performance, QBase adopts a cost model to select the optimal query plan for all six query types.

TABLE X: Vector Database Systems

	S1	S2	S3	S4	S5	S6
AnalyticDB-V [26]	✓	✓	✗	✗	✗	✗
Pase [42]	✓	✓	✗	✗	✗	✗
ElasticSearch [73]	✓	✓	✗	✗	✗	✗
VBase [27]	✓	✓	✓	✓	✓	✗
Milvus [23]	✓	✓	✓	✓	✓	✗
pgvector [74]	✓	✗	✗	✗	✗	✗
QBase	✓	✓	✓	✓	✓	✓

VII. Conclusion

We develop QBase, a vector search database system on PostgreSQL. It's the first query-aware system using similar query vector traits. QBase covers many vector search types, has query plans with a cost model for the best choice and good performance. Comprehensive evaluations using various vector datasets verify that QBase achieves state-of-the-art performance across all datasets compared to the state-of-the-art vector search systems. We'll add more hybrid indexes later.

References

[1] A. Miech, D. Zhukov, J.-B. Alayrac, M. Tapaswi, I. Laptev, and J. Sivic, "Howto100m: Learning a text-video embedding by watching hundred million narrated video clips," in *ICCV*, 2019, pp. 2630–2640.

[2] M. Paulin, M. Douze, Z. Harchaoui, J. Mairal, F. Perronin, and C. Schmid, "Local convolutional features with unsupervised training for image retrieval," in *ICCV*, 2015, pp. 91–99.

[3] Z. A. Yilmaz, S. Wang, W. Yang, H. Zhang, and J. Lin, "Applying bert to document retrieval with birch," in *EMNLP-IJCNLP*, 2019, pp. 19–24.

[4] "*The ChatGPT retrieval plugin lets you easily find personal or work documents by asking questions in natural language.*" 2024. [Online]. Available: https://github.com/openai/chatgpt-retrieval-plugin

[5] Z. Jing, Y. Su, Y. Han, B. Yuan, H. Xu, C. Liu, K. Chen, and M. Zhang, "When large language models meet vector databases: A survey," 2024.

[6] A. Asai, S. Min, Z. Zhong, and D. Chen, "Acl 2023 tutorial: Retrieval-based lms and applications," 2023.

[7] N. Li, B. Kang, and T. D. Bie, "Skillgpt: a restful api service for skill extraction and standardization using a large language model," 2023.

[8] R. Nasr, D. S. Hirschberg, and P. Baldi, "Hashing algorithms and data structures for rapid searches of fingerprint vectors," *Journal of Chemical Information and Modeling*, vol. 50, no. 8, pp. 1358–1368, 2010.

[9] C. J. Zhu, M. Song, Q. Liu, C. Becquey, and J. Bi, "Benchmark on indexing algorithms for accelerating molecular similarity search," *Journal of Chemical Information and Modeling*, vol. 60, no. 12, pp. 6167–6184, 2020.

[10] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *SIGKDD*, 2018, pp. 311–320.

[11] J.-T. Huang, A. Sharma, S. Sun, L. Xia, D. Zhang, P. Pronin, J. Padmanabhan, G. Ottaviano, and L. Yang, "Embedding-based retrieval in facebook search," in *SIGKDD*, 2020, pp. 2553–2561.

[12] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *arXiv preprint arXiv:1707.00143*, 2017.

[13] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[14] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *RecSys*, 2016, pp. 191–198.

[15] S. Okura, Y. Tagami, S. Ono, and A. Tajima, "Embedding-based news recommendation for millions of users," in *SIGKDD*, 2017, pp. 1933–1942.

[16] K. L. Clarkson, "An algorithm for approximate closest-point queries," in *SCG*, 1994, p. 160–164.

[17] M. PaoloCiaccia, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997, pp. 357–368.

[18] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search and similarity joins," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 3, pp. 556–571, 2017.

[19] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, p. 47–57.

[20] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," *Advances in Neural Information Processing Systems*, vol. 34, pp. 5199–5212, 2021.

[21] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *ICCV*. IEEE, 2009, pp. 2130–2137.

[22] J. Li, H. Liu, C. Gui, J. Chen, Z. Ni, N. Wang, and Y. Chen, "The design and implementation of a real time visual search system on jd e-commerce platform," in *Middleware*, 2018, pp. 9–16.

[23] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A purpose-built vector data management system," in *SIGMOD*, 2021, p. 2614–2627.

[24] "*Qdrant*," 2024. [Online]. Available: https://qdrant.tech/blog/

[25] "*Weaviate*," 2024. [Online]. Available: https://weaviate.io/hybrid-search

[26] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: a hybrid analytical engine towards query fusion for structured and unstructured data," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3152–3165, 2020.

[27] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang *et al.*, "{VBASE}: Unifying online vector similarity search and relational queries via relaxed monotonicity," in *OSDI*, 2023, pp. 377–395.

[28] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri, "Diskann: fast accurate billion-point nearest neighbor search on a single node," in *NIPS*, 2019.

[29] "*A Library for Efficient Similarity Search and Clustering of Dense Vectors.*" 2018. [Online]. Available: https://github.com/facebookresearch/faiss

[30] C. Fu, C. Wang, and D. Cai, "High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, pp. 4139–4150, 2022.

[31] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proc. VLDB Endow.*, vol. 12, no. 5, p. 461–474, jan 2019.

[32] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, p. 824–836, 2020.

[33] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *SCG*, 2004, p. 253–262.

[34] A. Babenko and V. Lempitsky, "The inverted multi-index," in *CVPR*, 2012, pp. 3069–3076.

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

[36] N. Koenigstein, P. Ram, and Y. Shavitt, "Efficient retrieval of recommendations in a matrix factorization framework," in *CIKM*, 2012, pp. 535–544.

[37] H. Li, T. N. Chan, M. L. Yiu, and N. Mamoulis, "Fexipro: fast and exact inner product retrieval in recommender systems," in *SIGMOD*, 2017, pp. 835–850.

[38] D. Lian, H. Wang, Z. Liu, J. Lian, E. Chen, and X. Xie, "Lightrec: A memory and search-efficient recommender system," in *The Web Conference*, 2020, pp. 695–705.

[39] S. Xiao, Z. Liu, W. Han, J. Zhang, Y. Shao, D. Lian, C. Li, H. Sun, D. Deng, L. Zhang *et al.*, "Progressively optimized bi-granular document representation for scalable embedding based retrieval," in *The Web Conference*, 2022, pp. 286–296.

[40] "*ANN Bnechmark nearest. ,*" 2024. [Online]. Available: http://ann-benchmarks.com/

[41] J. Li, H. Liu, C. Gui, J. Chen, Z. Ni, N. Wang, and Y. Chen, "The design and implementation of a real time visual search system on jd e-commerce platform," in *Middleware*, 2018, p. 9–16.

[42] W. Yang, T. Li, G. Fang, and H. Wei, "Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension," in *SIGMOD*, 2020, p. 2241–2253.

[43] "*Distributed vector search for AI-native applications.*" 2024. [Online]. Available: https://github.com/vearch/vearch/

[44] "*Bigann Benchmarks*," 2021. [Online]. Available: https://big-ann-benchmarks.com/neurips21.html

[45] J. Sun, X. Zhang, and S. Lei, "The evolution of public opinion and its emotion analysis in public health emergency based on weibo data," in *LISS*, 2022, pp. 415–434.

[46] "*Analysis of the Capacity of Google Trends to Measure Interest in Conservation Topics and the Role of Online News*," 2024. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0152802

[47] J. Mohoney, A. Pacaci, S. R. Chowdhury, A. Mousavi, I. F. Ilyas, U. F. Minhas, J. Pound, and T. Rekatsinas, "High-throughput vector similarity search in knowledge graphs," *Proc. ACM Manag. Data*, vol. 1, no. 2, 2023.

[48] S. Dhelim, N. Aung, M. A. Bouras, H. Ning, and E. Cambria, "A survey on personality-aware recommendation systems," *Artificial Intelligence Review*, vol. 55, pp. 2409 – 2454, 2021.

[49] K. Lampropoulos, F. Zardbani, N. Mamoulis, and P. Karras, "Adaptive indexing in high-dimensional metric spaces," *Proc. VLDB Endow.*, vol. 16, no. 10, p. 2525–2537, 2023.

[50] C. Zuo, J. Qian, S. Feng, W. Yin, Y. Li, P. Fan, J. Han, K. Qian, and Q. Chen, "Deep learning in optical metrology: a review," *Light: Science & Applications*, vol. 11, no. 1, p. 39, 2022.

[51] O. Jafari, P. Maurya, P. Nagarkar, K. M. Islam, and C. Crushev, "A survey on locality sensitive hashing algorithms and their applications," *arXiv preprint arXiv:2102.08942*, 2021.

[52] K. Krishna and M. N. Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.

[53] M. Ahmed, R. Seraj, and S. M. S. Islam, "The k-means algorithm: A comprehensive survey and performance evaluation," *Electronics*, vol. 9, no. 8, p. 1295, 2020.

[54] T. Liu, C. Rosenberg, and H. A. Rowley, "Clustering billions of images with large scale nearest neighbor search," in *WACV*, 2007, pp. 28–28.

[55] R. Chen, B. Liu, H. Zhu, Y. Wang, Q. Li, B. Ma, Q. Hua, J. Jiang, Y. Xu, H. Deng, and B. Zheng, "Approximate nearest neighbor search under neural similarity metric for large-scale recommendation," in *CIKM*, 2022, p. 3013–3022.

[56] P. Li, W. Zhao, C. Wang, Q. Xia, A. Wu, and L. Peng, "Practice with graph-based ann algorithms on sparse data: Chi-square two-tower model, hnsw, sign cauchy projections," 2023.

[57] "*Multimodal Vector Search with Personalization*," 2024. [Online]. Available: https://www.marqo.ai/blog/context-is-all-you-need-multimodal-vector-search-with-personalization

[58] "*Open Search*," 2024. [Online]. Available: https://opensearch.org/docs/latest/search-plugins/multimodal-search/

[59] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, 1993, p. 311–321.

[60] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, p. 35–40, 2010.

[61] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," 2023.

[62] I. Tautkute, T. Trzciński, A. P. Skorupa, Ł. Brocki, and K. Marasek, "Deepstyle: Multimodal search engine for fashion and interior design," *IEEE Access*, vol. 7, pp. 84 613–84 628, 2019.

[63] J. Etzold, A. Brousseau, P. Grimm, and T. Steiner, "Context-aware querying for multimodal search engines," in *International Conference on Multimedia Modeling*, 2012, pp. 728–739.

[64] H. Wen, X. Song, X. Yang, Y. Zhan, and L. Nie, "Comprehensive linguistic-visual composition network for image retrieval," in *SIGIR*, 2021, pp. 1369–1378.

[65] S. Jandial, P. Badjatiya, P. Chawla, A. Chopra, M. Sarkar, and B. Krishnamurthy, "Sac: Semantic attention composition for text-conditioned image retrieval," in *WACV*, 2022, pp. 4021–4030.

[66] "*Tpch Benchmarks*," 2024. [Online]. Available: https://www.tpc.org/tpch/default5.asp

[67] W. Hong, X. Tang, J. Meng, and J. Yuan, "Asymmetric mapping quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 7, pp. 1783–1790, 2019.

[68] "*PostgreSQL: The World's Most Advanced Open Source Relational Database*," 2024. [Online]. Available: https://www.postgresql.org

[69] "*Milvus Multi-Vector Search Reduce*," 2024. [Online]. Available: https://github.com/milvus-io/milvus/blob/5452376e904a5978243603923d68e3f4e065efc4/internal/proxy/search_reduce_util.go#L443

[70] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, 1975.

[71] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, 1999, p. 518–529.

[72] K. Lu, H. Wang, W. Wang, and M. Kudo, "Vhp: approximate nearest neighbor search via virtual hypersphere partitioning," *Proc. VLDB Endow.*, vol. 13, no. 9, p. 1443–1455, may 2020. [Online]. Available: https://doi.org/10.14778/3397230.3397240

[73] "*Free and Open, Distributed, RESTful Search Engine.*" [Online]. Available: https://github.com/elastic/elasticsearch

[74] "*Open-source vector similarity search for Postgres*." [Online]. Available: https://github.com/pgvector/pgvector

[75] "*an open-source vector database that stores both objects and vectors, allowing for the combination of vector search with structured filtering with the fault tolerance and scalability of a cloud-native database.*" [Online]. Available: https://github.com/weaviate/weaviate