Survey of Vector Database Management Systems

Sandesh katta
Department of Computer Science
Purdue University
West Lafayette, USA
katta5@purdue.edu

Pruthvi Belide
Department of Computer Science
Purdue University
West Lafayette, USA
pbelide@purdue.edu

Lakshmi P Maddipati
Department of Industrial Engineering
Purdue University
West Lafayette, USA
lmaddipa@purdue.edu

***Abstract*—**The surge in unstructured data, coupled with the prevalent use of large language models (LLMs), has exposed the difficulties of handling large-scale high-dimensional vector data. Existing systems that deal with this kind of vector data can be categorized into two types: 1) traditional database systems extending functionalities to vector data and 2) purpose-built vector database management systems. The goodness of the above systems revolves around a multitude of factors, some of which include Indexing and Search Optimization, Fast Vector Similarity Search, Efficient Vector Storage, and Scalability etc., The paper systematically explores the hurdles in conducting similarity searches in vectors, highlighting issues such as the ambiguity of semantic similarity, the computational overhead in comparing similarities, the absence of a natural partitioning method for efficient indexing, and the challenges of handling hybrid queries that necessitate both attribute and vector-based information. These challenges underscore the shortcomings of existing systems and algorithms, prompting the exploration of innovative approaches to address these obstacles in query processing, storage/indexing, and optimization, paving the way for the development of more effective vector database management systems (VDBMSs) to serve modern-day applications.

***Index Terms*—** hybrid queries, vector storage, semantic similarity

I. Introduction

Vector database is a type of database that store and effectively retrieve vector data, which are mathematical representations of features or attributes. Each vector has a certain number of dimensions, which can range from tens to thousands, depending on the complexity and granularity of the data.

Vector databases have several advantages over traditional databases.

Fast and accurate similarity search and retrieval: Vector database can find the most similar or relevant data based on their vector distance or similarity, which is a core functionality for many applications that involve natural language processing, computer vision, recommendation systems, etc. Traditional database can only query data based on exact matches or predefined criteria, which may not capture the semantic or contextual meaning of the data.

Support for complex and unstructured data: Vector database can store and search data that have high complexity and granularity, such as text, images, audio, video, etc. These types of data are usually unstructured and do not fit well into the rigid schema of traditional database. Vector database can transform these data into high-dimensional vectors that can capture their features or attribute.

Scalability and performance: Vector database can handle large-scale and real-time data analysis and processing, which are essential for modern data science and AI applications. Vector database can use techniques such as sharding, partitioning, caching, replication, etc. to distribute the workload and optimize the resource utilization across multiple machines or clusters. Traditional database may face challenges such as scalability bottlenecks, latency

This paper aims to provide an easily accessible description of fundamental concepts behind VDBMSs without focusing on the intricacies of a single product. The structure of the paper is outlined as follows: Section 2 addresses vector indexes, data structures designed for the efficient search and retrieval of vector data. These are built upon similarity search algorithms research while taking into account scalability and performance considerations associated with non-volatile storage mediums such as disks and SSDs. Moving on to Section 3, the focus shifts to the strageries to process complex queries encompassing multiple vector data types (multi-vector queries) or queries involving both attribute and vector data (hybrid queries).

II. Indexing

K-Nearest Neighbor Search (K-NNS): Find a vector (x) from the dataset given a query vector (q) such that the distance function between q and x is minimum. A brute-force similarity search gives us a time complexity O(nd) where n is the size of the dataset (n) and d is the dimensionality of the vector [4], [9]. This is not an effective search strategy since it is exhaustive and causes an increased latency. This is where vector indices come to our rescue. These indices efficiently traverse the dataset to minimize the number of comparisons and speed up the query processing.

Exact solutions for K-Nearest Neighbor Search (K-NNS) are effective mainly in scenarios with relatively low-dimensional data due to the challenges associated with the ”curse of dimensionality.” To tackle this issue, the concept of Approximate Nearest Neighbors Search (K-ANNS) was introduced, relaxing the requirement for exact searches [7]. Well-known K-ANNS solutions rely on approximated versions of tree algorithms, locality-sensitive hashing (LSH), product quantization (PQ), and proximity graphs (PG) . A vector index is an efficient organization of vector data using the above solutions. Some indices may even employ a combination of techniques, and as a result, we categorize indexes based on

their structure.

A. Hash-based approach: Hash-based indexing involves the use of hash functions to map high-dimensional vectors to specific locations within the database, facilitating expedited query processing.

Locality-Sensitive Hashing: LSH is a hash-based technique that helps to efficiently search for similar items in a large dataset. However, using LSH comes with some trade-offs.

In LSH [2], [5], there's a "family" of hash functions. If two items are close (determined by a distance measure), the probability that they end up with the same hash value is high. On the other hand, if they are far apart, the probability of sharing the same hash value is low. This behavior is controlled by parameters r_1, r_2 and p_1, p_2.

In a family of hash functions (H):

$$if \quad d(x,q) \leq r_1 \quad \text{then} \quad Pr_H(h(x) = h(q)) \geq p_1$$

$$if \quad d(x,q) \geq r_2 \quad \text{then} \quad Pr_H(h(x) = h(q)) \leq p_2$$

Here d(x,q) is is the distance measure between items *x* and *q* and r_1, r_2 and p_1, p_2 are tuning parameters.

We create a flexible group of hash functions by combining several individual hash functions. This group helps in constructing hash tables, which are like organized storage spaces. The number of these hash tables is set based on the required accuracy and performance of the application. When a search query is made, the item is hashed using these functions, and potential matches are found. The complexity of this process depends on factors like the number of hash tables and the dimensionality of the data (D).

B. Quantization-based approach: A quantizer transforms a vector into a more compact representation. This process is typically lossy, to minimize both information loss and storage requirements.

Product Quantization: PQ is a compression technique designed for condensing high-dimensional vectors into more efficient representations [5].

The process involves dividing a vector into multiple subvectors and applying a clustering algorithm, like k-means, to each subvector.

A vector x is split into m subvectors as shown below:

$$x = (x_1, x_2, x_3, ...x_m)$$

where x_i is ith sub-vector of x, dimensionality of x_i is d/m (d - dimension of x)

This assigns each subvector to a finite number of possible values, referred to as centroids, resulting in a concise code represented by centroid indices for each subvector.

Centrioid is calculated as:

$$c_i = \frac{1}{|S_i|} \sum\nolimits_{x \in S_i} x \tag{1}$$

The algorithm leverages vector quantization, where each subvector is mapped to its nearest centroid in a predetermined codebook. Initial steps include dividing vectors into equal-sized subvectors, with the length controlled by the parameter 'm'. For each subvector, 'k' centroids are found using the k-means algorithm, determining the size of the codebook. The final code is formed by concatenating centroid indices.

Product quantization stands out for its simplicity, relying on standard clustering algorithms and basic distance approximation methods. Performance is influenced by factors such as data dimensionality, the number of subvectors, centroids per subvector, and the chosen distance approximation method.

C. Tree-based approach: These trees are designed to group similar data points in the same subtree, expediting the identification of approximate nearest neighbors.

Annoy (Approximate Nearest Neighbors Oh Yeah): Annoy utilizes a recursive process to build a forest of binary trees [5]. In the initial step, two random vectors, a and b, are chosen from the dataset. The entire vector space is split along a hyperplane equidistant from both a and b. Vectors in the left half of this hyperspace are assigned to the left subtree, while those in the right half go to the right subtree.

The second iteration repeats this process for both the left and right subtrees, creating a tree with a depth of two and four leaf nodes. This recursive subdivision continues until a leaf node contains fewer than a predefined number, K, of elements. In the original Annoy implementation, users can set the value of K.

Once the index is fully constructed, the query processing begins. Given a query vector q, the search involves traversing the tree. At each intermediate node, a hyperplane splits the space. The query vector's position relative to the hyperplane is determined by calculating its distance to the left and right vectors. This process iterates until a leaf node that contains an array of, at most, K vectors is reached. These vectors are then ranked and returned as the top results to the user.

The median hyperplane between two points p and q is given by:

$$w.x + b = 0$$

Where w is p - q, x is any point on the hyperplane and b is the bias term. b = - 1/2 (w.p+w.q)

The nearest neighbor of a query point q in the forest is given by the formula:

$$min_{x \in C(q)} d(q, x)$$

where C(q) is the set of candidate points obtained by traversing each tree in the forest and retrieving all the points in the leaf node that q belongs to, and d is a distance function, such as Euclidean distance or cosine distance.

D. Graph-based approach: These types of indices operate on the idea that data points in a vector space can be modeled as a graph, with nodes representing the data values and edges denoting the similarity between data points. The construction of the graph ensures that nodes representing similar data points are connected by edges.

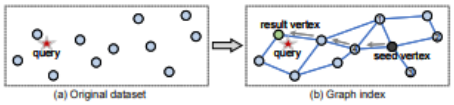

Fig. 1. Proximity Graph

Hierachical navigable small world (HNSW): The HNSW algorithm introduces a hierarchical structure to the graph by allocating each point to various layers based on different probabilities. In this structure, higher layers consist of fewer points connected by longer edges, while lower layers comprise more points connected by shorter edges. The uppermost layer consists of a single point, serving as the entry point for the search process. The algorithm incorporates a parameter denoted as M, which regulates the maximum number of neighbors assigned to each point in each layer.

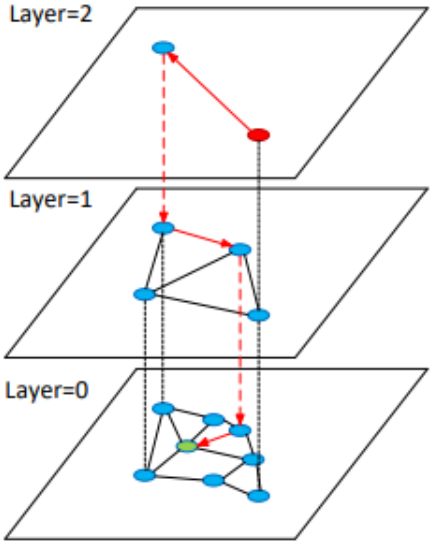

Fig. 2. Hierarchical Navigable Small World (HNSW)

HNSW offers better performance compared to alternative methods for ANNs, including tree-based and hash-based techniques [6]. Notably, HNSW excels in accommodating arbitrary distance metrics, adapting effectively to dynamic datasets, and delivering high levels of accuracy and recall while keeping memory consumption low. The algorithm strategically begins the search from the highest layer and progressively moves down to the lowest layer, selecting the closest node at each layer as the next hop. However, the above indexing techniques have been in place for a while and they don't perform very well when data nature becomes dynamic. With the rise of modern-day applications that generate humungous amounts of data it is necessary for the ANNS search to process the queries without high latencies. Along with this due to the challenge of dealing with high-dimensional data, it can be expensive to determine the correct neighbors for a new vector, which is crucial for updating the index. To mitigate these update costs, current

Structure	Index	Partitioning	Residence
Hash	LSH	Space	Memory
Quantization	PQ	Clustering	Memory
Tree	Annoy	Space	Memory
Graph	HNSW	Proximity	Memory
	NSW	Proximity	Memory
Combination	SPFresh	Proximity and Partitioning	Disk

TABLE I
LIST OF INDEXES AND WHERE THEY RESIDE.

systems employ a secondary index to gather updates, and these updates are later integrated into the main index through a global rebuilding process conducted at regular intervals. Nevertheless, this approach leads to significant fluctuations in search latency and accuracy. Additionally, it demands substantial resources and is exceptionally time-consuming to execute a complete index rebuild.

To efficiently handle extensive vector datasets with reduced expenses, an index like SPFresh [3], which is a disk-based vector index can be extremely useful. SPFresh facilitates lightweight incremental in-place local updates, eliminating the necessity for a global rebuild.

SPFresh: This disk-based vector index relies on LIRE, a Lightweight Incremental REbalancing protocol at its core. This protocol gathers minor updates to vectors within local partitions and efficiently re-establishes a balanced distribution of local data at a minimal cost. In contrast to the resource-intensive global rebuilds, LIRE excels at preserving index quality by addressing data distribution irregularities locally in real time.

LIRE Protocol: LIRE proceeds on a balanced SPANN kind of index. The idea behind this is that a well-balanced index may experience localized changes with just a single vector update. This ensures that the entire rebalancing process remains light and cost-effective [1], [12].

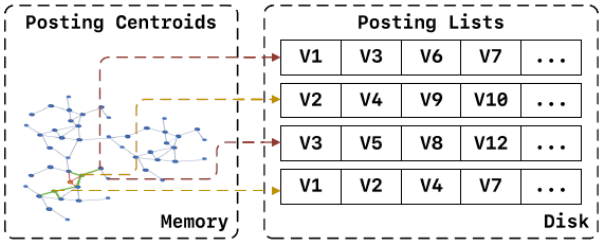

Fig. 3. A Balanced SPANN Index

An essential characteristic of an effectively partitioned vector index is the Nearest Partition Assignment (NPA). This means that each vector should be assigned to the nearest posting to ensure it is accurately represented by the posting centroid. Since continuous updates to a posting can negatively impact query recalls and latency, SPFresh addresses this by

splitting a posting once it reaches a predetermined maximum length. However, a straightforward splitting approach may compromise the NPA property of the index.

When a posting reaches a maximum allowed length it is then split. However, the creation of new centroids via a spit makes previous NPA compliance obsolete for vectors in the nearby postings. To fix this LIRE reassigns nearby postings of a split.

Re-assignment of vectors can be costly since it involves expensive modifications to on-disk postings for each reassigned vector. Therefore, it is crucial to accurately identify the appropriate set of neighborhoods (neighboring postings) to prevent unnecessary reassignment. This way SPFresh provides support for in-place lightweight updates.

III. Query Processing

A. Multi vector query processing

In many cases like Facial Recognization, Each entity contains more than one vector attribute for example one vector for eyes, one for nose, one for mouth etc. While queriying we need to match all these vectors at same time. Formally Each entity e contains α vectors $v_0, v_1, v_2, ..., v_\alpha$ then given a query q, aggregated scoring function g over the similarity function f is defined as

$$g(f(q.v_0, e.v_0), f(q.v_0, e.v_0), ..., f(q.v_\alpha, e.v_\alpha))$$

There are multiple strageries to deal with multivector queries

1) Vector Fusion: Idea behind vector fusion is to combine all vectors, perform normal vector search, but inorder to this work we need our aggerate scoring function and similarity function to be compatable. Formally consider $g(x_1, .., x_\alpha) = w1 \times x_1 + ... + w_\alpha \times x_\alpha$, where f is dot product. we can re write aggergate scoring function as

$$\begin{aligned} &dot([e.v_1, ..., e.v_\alpha], [w_1 \times q.v_1, ..., w_\alpha \times q.v_\alpha]) \\ &\quad = w_0 \times dot(e.v_0, q.v_0) + ... + w_\alpha \times dot(e.v_\alpha, q.v_\alpha) \\ &\quad = g(f(q, e)) \end{aligned}$$

If aggerate function and scoring function are not compiatable, we can use other strategies like Iterative Merging,

2) Iterative Merging: Idea behind iterative merging is simple, get top k on each of vector attribute, combine the scores using Non Random Access algorithm [3] if we cannot find sufficient top k, we go back and increase k to 2k.

Algorithm 1 Iterative merging

```
K' ← k;
while K' > threshold do
    for all i do
        R_i ← VectorQuery(q.v_i, D_i, K_i);
        if results are fully determined with NRA [3] on all R_i then
            return top-k results from U_i, R_i;
        else
            K' ← K' × 2;
        end if
    end for
    return top-k results from U_i, R_i;
end while
```

B. Attribute Filtering

Attribute filtering, a crucial aspect in various applications involving hybrid query types, combines vector and non-vector data to retrieve relevant information based on attribute constraints and vector similarities. For example, a shopping application like American Eagle or Amazon allows us to search for a particular product by image search and several other filters like price, brand, etc. This section explores and evaluates different strategies employed in attribute filtering, analyzing their strengths, weaknesses, and performance implications in query processing. This section assumes that each entity is associated with a single vector and attribute. Strategies discussed encompass attribute-first and vector-first approaches, culminating in a novel partition-based methodology that significantly enhances query performance. Each query comprises two conditions: C_A, representing the attribute constraint, and C_V, denoting the normal vector query constraint responsible for returning the top-k Similar vectors.

1) Strategy A: attribute-first-vector-full-scan: [10] The method uses the attribute constraint C_A to find relevant entities through index search, primarily using binary search or a B-tree index. If data exceeds memory limits, skip pointers expedite search operations. All entities in the result set are thoroughly scanned against the query vector, generating the final top-k results. This method suits highly selective C_A conditions and ensures precise results.

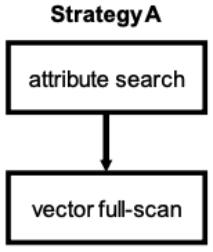

Fig. 4. Strategy A

2) Strategy B: attribute-first-vector-search: [10] This method identifies entities based on attribute constraint C_A and creates a bitmap of resulting entity IDs. The approach checks this bitmap during standard vector query processing, only including vectors that pass bitmap testing. This strategy suits moderately selective scenarios where C_A or C_V is selective.

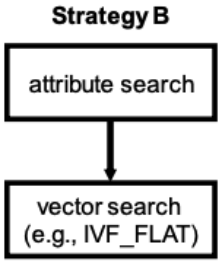

Fig. 5. Strategy B

3) Strategy C: vector-first-attribute-full-scan: [10] This method uses vector indexing techniques to acquire relevant entities based on the vector constraint C_V. The entities undergo a scan to confirm satisfaction of attribute constraint C_A. It searches for $\theta * k$ outcomes during vector query processing to ensure final results. This strategy is suitable for highly selective vector constraints with few candidates.

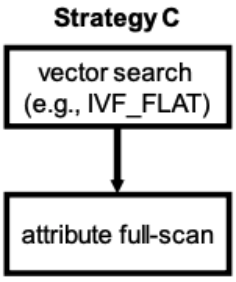

Fig. 6. Strategy C

4) Strategy D: cost-based: [10] This Strategy is a cost-based method proposed in AnalyticDB-V [11] that calculates the costs of strategies A, B, and C and selects the one with the lowest cost. According to Milvus, the cost-based approach is appropriate in nearly all situations.

5) Strategy E: partition-based: [10] Milvus implements this methodology by partitioning the dataset based on frequently searched attributes, employing a cost-based strategy for each partition. The approach maintains attribute frequency

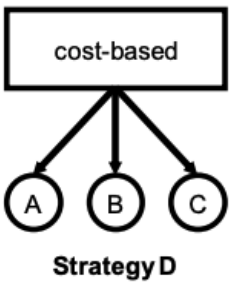

Fig. 7. Strategy D

in a hash table and increments counters whenever a query references a specific attribute. If a query range covers a particular partition's range, the strategy bypasses attribute constraint checks and focuses solely on vector query processing. For example, when price is the frequently queried attribute, the strategy divides the dataset into N partitions. When queried, the strategy may utilize only P_0 and P_1 for searching, enhancing query performance significantly.

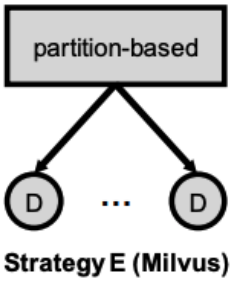

Fig. 8. Strategy E

C. VBase - Relaxed Monotonicity

1) Monotonicity: B-trees or B+ tree indices are maintained in conventional database systems. These indices enable query traversal in ascending or descending order, showcasing a property known as Monotonicity. This characteristic allows seamless navigation along the index in a consistent and ordered direction. In the figure 9, if we want to search for the number 11, the query traversal navigates monotonically, starting from the root.

Maintaining Monotonicity in vector databases poses significant challenges due to their high dimensionality. Hence, these databases adopt irregular structures such as graphs or cluster-based arrangements. For instance, in TopK queries, the traversal direction is guided by a query vector 'q,' moving towards approximate nearest neighbors based on distance metrics from anchor points (e.g., cluster centroids or sampled graph vertices). Unlike in traditional databases, this traversal can significantly alter the direction to 'q.' The process involves exploring the vector index for numerous steps until

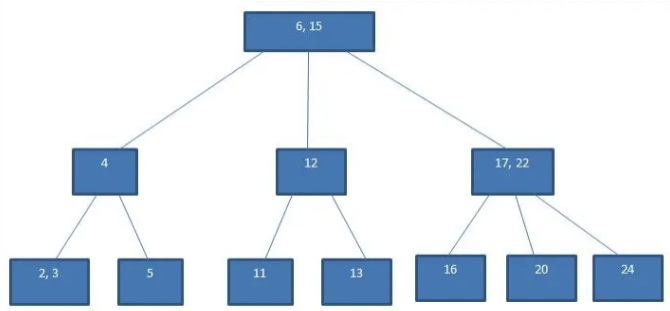

Fig. 9. An example of B Tree [8]

determining the unlikelihood of finding a closer neighbor than the current K neighbors. Finally, tentative indices are created based on K vectors sorted by their proximity to the target vector, establishing a temporary index that upholds Monotonicity.

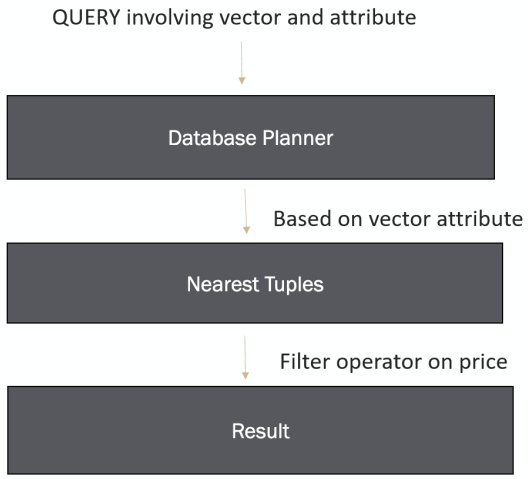

Fig. 10. Strategy C execution Flow

2) Database planner for Strategy C: In database planning for Strategy C, consider a shopping scenario from above. In this context, assume the query is to find the Top K products most similar to an image but within a specified price range. The process involves an initial approximate search for the closest vectors, followed by applying filters based on attributes like price to refine and present the final output as indicated in figure 10. The primary challenge with this approach lies in the inability to accurately predict the number of tuples that will satisfy the filter operator. Specifically, determining the exact Top-N required to ensure that subsequent filtering yields an outcome of at least k tuples becomes an uncertain task. Two common approaches emerge to address this issue. The first involves conservatively setting a significantly large K or employing a trial-and-error method by experimenting with various K values. However, these methods tend to result in suboptimal query performance.

3) Relaxed Monotonicity: [13] VBASE identifies Relaxed Monotonicity as a shared attribute between vector search systems and relational databases. This characteristic is a pivotal factor in enabling databases to overcome the limitations of TopK-only interfaces and facilitates efficient execution by allowing on-the-fly early termination. Moreover, it upholds the semantics of TopK-based solutions. VBASE adopts a Two-phase pattern, delineated in Figure 11. In the initial phase, index traversal approximates the target vector region despite significant oscillations in vector distances. Subsequently, in the second phase, the traversal stabilizes, steadily moving away from the target vector region in an approximate manner.

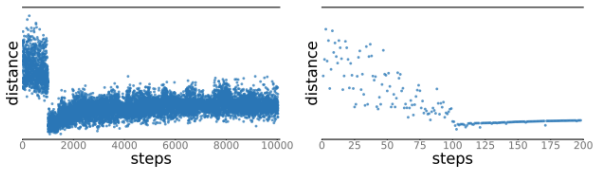

Fig. 11. Travesal Patterns [13]

The Formal Definition of Relaxed Monotonicity is established based on a two-phase traversal pattern, outlining how a vector index traversal progresses through a query vector q. Figure 12 illustrates this process, showcasing the traversal path concerning q. Initially, the query gradually approaches the neighborhood of q, depicted as a neighbor sphere centered around q. Following this, the traversal exits the sphere and enters phase two, where potential termination of the query occurs.
R_q, the radius of q's neighborhood, is defined as

$$R_q = Max(TopE(Distance(q, v_j)|j \in [1, s1]))1001[13]$$

The term "TopE" represents the E nearest neighbors of query vector q observed during the ongoing traversal at step s. For a query aiming to retrieve the K closest vectors, it necessitates that the marked count E is equal to or greater than K to ensure the generation of an adequate number of final results.

M_s^q is subsequently employed to ascertain if the traversal progresses into phase 2, signifying its departure from the neighborhood sphere surrounding the query vector.
M_s^q is defined as,

$$M_s^q = Median(Distance(q, v_i)|i \in [sw + 1, s])1001[13]$$

"Distance(q, v_i)" represents the distance between query vector q and vector vi, traversed within the previous traversal window. The median is utilized instead of the mean to mitigate the impact of outlier vectors within the traversal window that may have exceptionally large or small distances from q compared to other vectors. For instance, this helps account for outlier

vectors at the extreme leftmost and rightmost positions in the traversal window depicted in Figure 12. Finally, Relaxed Monotonicity can be defined as

$$\exists s, M_t^q \geq R_q, \forall t \geq s1001[13]$$

Overall, The TopK search query can terminate early upon entering the second phase as further traversal is improbable to reveal more similar vectors. Graph-based Vector Indices, such as Hierarchical Navigable Small World, employ a strategy where, in the initial phase, they navigate through hierarchical coarse-grained to fine-grained graphs to approach the neighborhood of query vector q. Upon reaching the fine-grained graph, the traversal transitions into the second phase, signifying the discovery of the neighborhood before departing. VBASE also implements filtering during index traversal. This optimization is a fundamental factor contributing to VBASE's superior performance over TopK-based solutions.

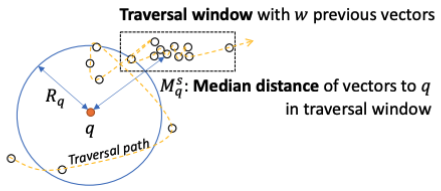

Fig. 12. An Illustration of Relaxed Monotonicity [13]

4) VBase - NRA Improvement: [13] TopK-based systems perform multi-column scans using multiple sets of sorted vectors obtained through TopK in multi-column vector queries. For instance, Milvus applies the NRA [3] algorithm for multi-column scans, doubling the value of K and re-executing the query if the previous results are inadequate. Each attempt with a larger K constitutes an independent traversal over the underlying vector index, resulting in excessive vector access and computational overhead. In contrast, VBASE natively incorporates the NRA algorithm, eliminating the need for repetitive NRA executions. The NRA algorithm traverses each vector index in a round-robin manner. However, there may be better approaches than this approach, especially in vector search scenarios.

The VBase approach utilizes local and global information to balance exploration and exploitation to guide index traversal (refer to Figure 13). The traversal process is divided into rounds, incorporating a traversal decision module. This module maintains a local priority queue, storing results from the previous round. Leveraging this local information helps identify indices that are more likely to yield better results, prompting more frequent visits in the subsequent round. To prevent local optima, the decision module also stores and updates the average score of all traversed entities for each index in each round. Additionally, guided by this global information, we traverse each index Wi times in a round. Wi is calculated based on a hyper-parameter n2 and a weighted ratio of average scores. This strategy ensures that high-quality indices are frequently traversed while guaranteeing at least one traversal for low-quality indices in each round.

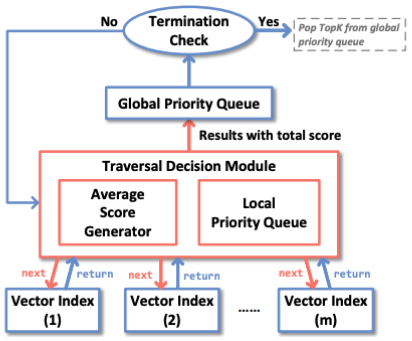

Fig. 13. NRA Improvement [13]

IV. Current Challenges

A. Similarity Measures

As we go to higher dimension or dealing with sparse vector data, current similarity measure like elucdian distance, cosine similarity or dot product become irrelevant. Still there is no single similarity measure that perfectly captures notion of similarity under all conditions. This means, databases need to analyze the data and select the measure, which is not done by any vector database.

B. How to pick right index?

There is always a tradeoff between latency and recall while dealing with vector data due to absence of total ordering. Exact accurate search is very costly in vector domain due to absence of total ordering. Almost all use cases, don't require Exact accurate search. Finding the right balance between query latency and recall of the results is crucial for given use case and could have large implications. To find the right balance, end user requires comprehensive understanding the intricacies of algorithms and their tradeoffs.

C. Dealing with heterogeneous vector data types

Even though Milvus and Vbase has done good job of dealing different vector data types of vector data, such as dense vectors, sparse vectors, binary vectors, etc and each type of vector data may have different characteristics and requirements, such as dimensionality, sparsity, distribution, similarity metrics, There is no effeictive way for querying vector and non vector data effectively. This area is still unexplored.

D. Data Freshness Problem

Currently most indexes are built for read heavy in other words updating indexes sometimes require complete rebuilding. In most use cases data is static or there is no need for online updates, but there are use cases where online updates are important Hence require indexes which are designed to accomdate online updates. We feel SPFresh is the right step in this direction.

REFERENCES

[1] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighbor search. *arXiv preprint arXiv:2111.08566*, 2021.

[2] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1073–1081, 2011.

[3] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *CoRR*, cs.DB/0204046, 2002.

[4] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. Manu: a cloud native vector database management system. *arXiv preprint arXiv:2206.13843*, 2022.

[5] Yikun Han, Chunjiang Liu, and Pengfei Wang. A comprehensive survey on vector database: Storage and retrieval technique, challenge. *arXiv preprint arXiv:2310.11703*, 2023.

[6] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[7] James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021*, 2023.

[8] Sudiksha. B tree and B+ tree - BeUNI - Medium. 12 2021.

[9] 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.

[10] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery.

[11] 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. *Proc. VLDB Endow.*, 13(12):3152–3165, aug 2020.

[12] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. Spfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 545–561, 2023.

[13] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 377–395, Boston, MA, July 2023. USENIX Association.