Vector Database Management Techniques and Systems

James Jie Pan
jamesjpan@tsinghua.edu.cn
Tsinghua University
Beijing, China

Jianguo Wang
csjgwang@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Guoliang Li
liguoliang@tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

Feature vectors are now mission-critical for many applications, including retrieval-based large language models (LLMs). Traditional database management systems are not equipped to deal with the unique characteristics of feature vectors, such as the vague notion of semantic similarity, large size of vectors, expensive similarity comparisons, lack of indexable structure, and difficulty of answering "hybrid" queries that combine structured attributes with feature vectors. A number of vector database management systems (VDBMSs) have been developed to address these challenges, combining novel techniques for query processing, storage and indexing, and query optimization and execution and culminating in a spectrum of performance and accuracy characteristics and capabilities. In this tutorial, we review the existing vector database management techniques and systems. For query processing, we review similarity score design and selection, vector query types, and vector query interfaces. For storage and indexing, we review various indexes and discuss compression as well as disk-resident indexes. For query optimization and execution, we review hybrid query processing, hardware acceleration, and distibuted search. We then review existing systems, search engines and libraries, and benchmarks. Finally, we present research challenges and open problems.

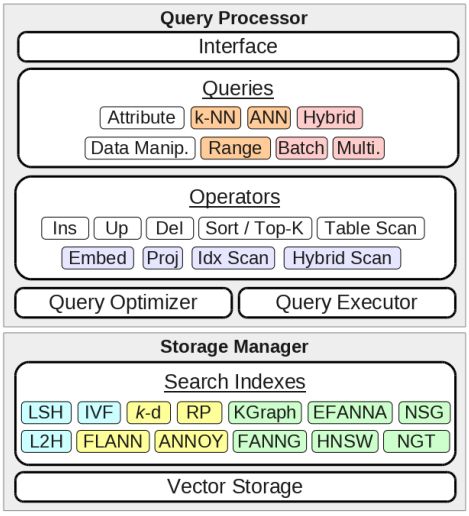

Figure 1: Overview of a VDBMS (Vector Database Management System).

CCS CONCEPTS

• **Information systems** → **Data management systems**.

KEYWORDS

Vector Database, Vector Similarity Search, Dense Retrieval, k-NN

ACM Reference Format:
James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3626246.3654691

1 INTRODUCTION

High-dimensional feature vectors are now used in a variety of dense retrieval search applications, including retrieval-based large language models (LLMs) [28, 31, 51], e-commerce [54], recommendation [82], document retrieval [76], and so on [45, 51, 79, 84]. These applications may involve billions of vectors and require millisecond query latencies, all while needing to scale to increasing workloads without sacrificing performance or response quality.

But existing traditional database management systems, including NoSQL and relational databases, are not designed for these datasets and workloads. First, vector queries rely on the concept of *similarity* which can be vague for different applications, requiring a different query specification. Second, similarity computation is more expensive than other types of comparisons seen in relational predicates, requiring efficient techniques. Third, processing a vector query often requires retrieving full vectors from the collection. But each vector may be large, possibly spanning multiple disk pages, and the cost of retrieval is more expensive compared to simple attributes while also straining memory. Fourth, vector collections lack obvious properties that can be used for indexing, such as being sortable or ordinal, preventing the use of traditional techniques. Finally, "hybrid" queries require accessing both attributes and vectors together, but it remains unclear how to do this efficiently.

These challenges have led to the rise of *vector database management systems* (VDBMSs) specially adapted for these applications, and there are now over 20 commercial VDBMSs developed within the past five years. A typical VDBMS is composed of a query processor and a storage manager (Figure 1). For the query processor, VDBMSs introduce new approaches to query interfaces, query types such as k-nearest-neighbor search, hybrid, and multi-vector queries, and data operators such as similarity projection and hybrid index scan. New techniques have also been proposed for query

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06.
https://doi.org/10.1145/3626246.3654691

optimization and execution, including plan enumeration and selection for hybrid query plans and hardware accelerated search. For the storage manager, several techniques for large-scale vector indexing and vector storage are now available, including hashing and quantization-based approaches like PQ [1] and IVFADC [49] that tend to be easy to update; tree-based indexes like FLANN [62] and ANNOY [2] that tend to support logarithmic search complexity; and graph-based indexes like HNSW [58] and others that are efficient in practice but with less theoretical understanding. Together, these techniques culminate into a variety of native systems, extended systems, and search engines and libraries over a spectrum of performance and accuracy characteristics and capabilities.

In this tutorial, we review techniques and systems for vector data management along with benchmarks, followed by remaining challenges and open problems.

Tutorial Overview. This tutorial contains three parts and the intended length is 1.5 hours. The first part discusses specific techniques and will last 50 minutes.

(1) Query Processing (10 min.). Query processing begins with a search specification that defines the search parameters. These include considerations for similarity score design, score selection and the curse of dimensionality [22, 30, 61], the query type such as basic, multi-vector, and hybrid queries [79, 84], and the query interface.

(2) Storage and Indexing (30 min.). Vector indexes tend to rely on novel partitioning techniques such as randomization, learned partitioning, and navigable partitioning. The performance, accuracy, and storage characteristics of an index depend on the techniques used and the index structure. We classify indexes into table, tree, or graph-based, and then describe the techniques for each index type. To deal with large size, we also discuss vector compression using quantization [49, 59] and disk-resident indexes [32, 74].

(3) Optimization and Execution (10 min.). For processing hybrid queries, several plan enumeration and plan selection techniques have been proposed, including rule-based [3, 4] and cost-based selection [79, 84], and which also introduce new hybrid operators including block-first scan [43, 79, 84, 87] and visit-first scan [43, 87]. Several techniques also speed up vector search via hardware acceleration using SIMD [26, 27], GPUs [50], and distributed search. To address slow index updates, some VDBMSs also rely on out-of-place updates.

The second part discusses vector database management systems and will last 30 minutes.

(1) Native Systems (10 min). Native systems such as Pinecone [5], Miluvs [6, 79], and Manu [45] aim at high performance vector search applications by offering a narrower range of capabilities.

(2) Extended Systems (10 min). For applications that require more sophisticated capabilities, several extended systems such as AnalyticDB-V [84], PASE [90], `pgvector` [7], and Vespa [4] have been developed based on NoSQL or relational systems.

(3) Search Engines and Libraries (5 min). Several search engines such as Apache Lucene [8] and Elasticsearch [9] now also incorporate vector search capability via integrated vector indexes. Several libraries are also available such as Meta Faiss [1] that provide vector search functionality.

(4) Benchmarks (5 min). We describe two notable benchmarks that evaluate a wide variety of search algorithms and systems over a range of workloads [29, 91].

The final part discusses challenges and open problems (10 min.). We describe several fundamental challenges, including how to perform similarity score selection, design more efficient hybrid operators and indexes, and estimate the cost of hybrid plans. We also describe future applications, including index-supported incremental search, multi-vector search, and enhancing security and privacy.

Target Audience. This tutorial is intended for database researchers interested in understanding and advancing the state-of-art techniques for large-scale vector database management and modern applications beyond similarity search. This tutorial may also benefit industry practitioners interested in learning about the latest commercial systems. There are no prerequisites beyond a basic understanding of database concepts.

Related Tutorials. A recent tutorial [28] discusses how vector search can be used for retrieval-based LLMs. There are also separate tutorials on similarity search techniques [37, 67, 68].

Our tutorial aims to complement these tutorials by focusing on vector database management systems as a whole, and most of this tutorial has not been covered elsewhere. Specifically, most of the overlap is confined to Section 2.2, and the extent is not large. In [37], a broad taxonomy of search techniques is given, along with representative examples. Similarly in [67, 68], an overview of various exact and approximate search techniques is given. Some of the material in Section 2.2, mainly locality-sensitive hashing, learning-to-hash, ANNOY, and HNSW, overlaps with these past tutorials. But Section 2.2 also discusses key indexing trends in VDBMSs that have not been discussed in past tutorials, including disk-based indexes, quantization-based compression approaches for handling large vector collections, and the diversity of graph-based indexes. Aside from this section, all other sections in this tutorial have not been covered elsewhere as they pertain to the VDBMS as a whole, including query processing, hybrid operators, plan enumeration, and plan selection, and a survey of existing VDBMSs.

2 TUTORIAL

2.1 Query Processing

Similarity Scores. Dense retrieval works based on similarity. A similarity score can be used to quantify the degree of similarity between two feature vectors. While many scores have been proposed, different scores may lead to different query results, and so how to perform *score selection* is an important problem for VDBMSs, in addition to *score design*.

(1) Score Design. A similarity score is designed to accurately capture similarity relationships between feature vectors. Existing similarity scores can be classified as *basic scores*, *aggregate scores*, and *learned scores*. Basic scores are derived directly from the vector space and include Hamming distance, inner product, cosine angle, Minkowski distance, and Mahalanobis distance. For certain workloads involving multiple query or feature vectors per entity, aggregate scores such as mean, weighted sum, and others [79] combine multiple scores into a single scalar score that can be more easily compared.

It may also be possible to improve query results by learning a suitable score directly over the vector space. This is the goal of *metric learning*, and several techniques have been proposed [21, 60, 91].

(2) Score Selection. Score selection aims to select the most appropriate score for a particular application. While many scores are known, *automatic score selection* remains challenging. We mention one attempt to dynamically adjust the score based on the query [82]. Score selection is also related to *query semantics*, as certain query entities such as text strings may still be ambiguous and need to be resolved before a suitable score can be selected [75]. Finally, the *curse of dimensionality* limits the usefulness of certain distance-based scores, requiring other scores to compensate [22, 30, 61].

Query Types and Basic Operators. *Data manipulation* queries aim to alter the vector collection. As each vector is derived from an embedding model, it is possible to integrate the model within the VDBMS. A VDBMS also must handle vector search queries. *Basic search queries* include k-nearest neighbor (k-NN) and approximate nearest neighbor (ANN) queries, and *query variants* include predicated, batched, and multi-vector queries. To answer these queries, a VDBMS compares the similarity between the query vector and a number of candidate vectors using *similarity projection*. The quality of a result set is measured using precision and recall.

(1) Data Manipulation. The embedding model can live inside or outside the VDBMS. Under *direct* manipulation, users directly modify the feature vectors, and the user is responsible for the embedding model [7, 90]. Under *indirect* manipulation, the collection appears as a collection of entities, not vectors, and users manipulate the entities [5, 10]. The VDBMS is responsible for the embedding model.

(2) Basic Search Queries. In a (c, k)*-search query*, the goal is to retrieve k vectors that are most similar to the query vector, and where no retrieved vector has a similarity score that is a factor of c worse than the best non-zero score. In a *range query*, a similarity threshold is given, and the goal is to retrieve all vectors with similarity scores within the threshold. Some particular cases of (c, k)-search queries have been studied, notably $c = 0, k > 1$ corresponding to the k-NN query and $c > 0, k > 1$ corresponding to the ANN query. These queries have been individually studied for many decades, leading to a variety of techniques and theoretical results [24, 48, 70].

(3) Query Variants. Most VDBMSs support predicated or "hybrid" queries, and some also support batched and multi-vector queries for applications such as e-commerce, facial recognition, and text retrieval [11, 79, 84]. In a hybrid query, vectors in the collection are associated to structured attributes regarding the represented entity, and each vector in the search result set must also satisfy boolean predicates over the corresponding attributes [84]. For batched queries, a number of search queries are given at once, and the VDBMS must answer all the queries in the batch. Several techniques have been proposed to exploit commonalities between the queries in order to speed up processing the batch [50, 79]. Finally in a multi-vector query, multiple feature vectors are used to represent either the query, each entity, or both, and these can be supported via aggregate scores [79]. Multi-vector queries support several additional sub-variants.

(4) Basic Operators. Similarity projection can be used to answer vector search queries by projecting each vector in the collection onto its similarity score.

Query Interfaces. Some VDBMSs aim to support only a small number of query types and simple APIs are sufficient. Other VDBMSs aim to support a wide range of query types and may rely on SQL extensions.

2.2 Indexing

An index can be used to speed up search queries, but vectors cannot be indexed like structured attributes as they lack a natural sort order and categories that are used in typical attribute indexes such as B-tree. As a result, vector indexes rely on techniques including *randomization*, *learned partitioning*, and *navigable partitioning* in order to partition the collection so that it can be more easily explored. To address the large size of vectors, *disk-resident* indexes have also been proposed, in addition to techniques based on a compression technique called *quantization*. A single index may combine several of these techniques, but the performance of an index also depends on its structure that can be *table-based*, *tree-based*, or *graph-based*. In this section, we describe several vector indexes that are used in existing VDBMSs, starting from table-based indexes.

Table-Based Indexes. A table-based index partitions the vector collection into buckets, and each bucket can be retrieved by looking up a key like the rows in a hash table. In general, table-based indexes are easy to maintain but search performance may be worse than other indexes at same recall if buckets are very large. Large buckets can improve recall but are harder to scan while small buckets may suffer from low recall but are easier to scan. Existing techniques, including *locality sensitive hashing* (LSH) and *learning to hash* (L2H), tend to rely on randomization and learned partitioning to improve performance at high recall. Furthermore, *quantization* is a compression technique that relies mostly on learning compression codes in order to reduce storage costs.

(1) Locality Sensitive Hashing (LSH). Locality sensitive hashing relies on random hash functions to bucket vectors. The basic idea is to hash each vector into each of L tables, with each hash function a concatenation of K number of hash functions that belong to a "hash family". To answer a query, the query vector is hashed to each of the tables, and collisions are kept as candidates. The hash family, along with the tunable parameters L and K, can be designed to give error guarantees for ANN. Many hash families are known with varying performance and accuracy characteristics, including random hyperplanes in E^2LSH [35], binary projections in `IndexLSH` [1], and overlapping spheres in FALCONN [23, 25].

(2) Learning to Hash (L2H). Learning to hash aims to use machine learning techniques to directly learn a hash function that can bucket similar vectors together. One technique is k-means clustering in SPANN, where each vector is bucketed along with other members of the same cluster [32]. The SPANN index also introduces several techniques for disk-resident collections such as overlapping buckets to reduce I/O retrievals. Other techniques include spectral hashing [85] and techniques based on neural networks [71]. While these techniques may produce high quality partitionings, they are data dependent and cannot easily handle out-of-distribution updates. A survey of techniques can be found in [81].

(3) Quantization. Quantization aims to map each vector onto a smaller discrete subset of compression codes [44]. For example, the SQ index works by mapping each vector onto a bit-compressed representation, where every 64-bit dimension is reduced to 32 bits [1], and the IVFSQ index first buckets the vectors into a few k-means clusters, and then stores the bit-compressed vectors in each bucket. The k-means centroids can also be directly used as compression codes [42, 56], but this may require large k for large collections. *Product quantization* splits the original space into multiple subspaces, each of which can support more efficient k-means clustering compared to the original space [49]. The PQ index directly maps each vector onto its product quantization code [1]. The IVFADC index first buckets the vectors into a few k-means clusters, and then stores the product quantized codes in each of the buckets. Other techniques aim to reduce the compression error and include Cartesian k-means [64], optimized PQ (OPQ) [41], hierarchical quantizers [89], and the score-aware ScaNN anisotropic quantizer [1] [46]. A survey of quantization techniques is available at [59].

Tree-based indexes. A tree-based index recursively partitions the vectors to yield a search tree, in general offering logarithmic search. One of the fundamental indexes is k-d tree, which performs partitioning splits deterministically and has well-understood properties [33, 69]. More recent techinques rely on randomization to determine splits or for other purposes, as deterministic trees cannot easily adapt to the intrinsic dataset dimensionality. For example, a *principal component tree* first finds the principal components of the dataset, and then splits along the principal axes. The PKD-tree splits by rotating through the principal axes [72], while FLANN splits along random principal dimensions [62]. To avoid the expensive pre-processing step for finding the principal components incurred by these indexes, *random projection trees* adopt random splits. The RPTree uses random splitting planes along with random splitting thresholds [33, 34]. To improve recall, a forest of trees can be used, similar to the multiple hash tables used in LSH. The ANNOY index is similar to RPTree but selects the splitting threshold based on random medians [2].

Graph-based indexes. A graph can be overlayed on top of the vectors so that a node resides at each vector location in the vector space, inducing distances between the nodes. The nodes corresponding to similar vectors are then connected with edges that can be weighted by similarity. These edges, along with the distance metric, guide vector search through the graph. The edge selection problem that determines which edges to include in the graph has been extensively studied, leading to distinct graph categories. Typically edges are selected in order to yield high performance search, but as graphs are highly data dependent, they tend to be hard to update. In a *k-nearest neighbor graph* (KNNG), each vector is connected to its k most similar vectors. This allows k-NN queries to be answered exactly in $O(1)$ time if the query vector is a member of the vector collection. In a *monotonic search network* (MSN) and *small world graph* (SWG), the aim is to select edges that make the graph easily navigable. All graph types may make use of randomized, learned, and navigable partitioning techniques during graph construction.

(1) k-Nearest Neighbor Graphs (KNNGs). The brute-force approach for constructing a KNNG requires $O(N^2)$ time, and unfortunately this appears to be a fundamental limit [86] despite some empirical results [65, 77]. Other techniques aim to approximate a KNNG via iterative refinement. For example, KGraph (NN-Descent) begins with a random KNNG and refines it by examining the second-order neighbors of each node in the graph [36]. To improve recall, EFANNA[2] begins with a KNNG constructed using a forest of randomized k-d trees. In [78], a similar approach is taken but using trees that split on random hyerplanes.

(2) Monotonic Search Networks (MSNs). An MSN guarantees that a monotonic search path exists for every pair of nodes in the graph, allowing for a simple best-first vector search procedure. But similar to KNNGs, MSNs are also difficult to construct [38, 63]. Recent techniques use *search trials* that repeatedly probe the quality of the graph as edges are added. During a search trial, a best-first search is conducted on the given source and target nodes. If no monotonic search path can be found, then edges are added until such a path exists. An initial graph such as a random graph [74] or approximate KNNG [40] may be used to initialize the MSN to reduce the number of trials. Some indexes such as FANNG [47] perform a large number of search trials over random node pairs, while others such as NSG [40] and Vamana [74] designates a "navigating node" as the source for all trials and selects random targets instead in order to speed up construction. In [74], a disk-resident version of Vamana (DiskANN) is also introduced.

(3) Small World Graphs (SWGs). A "small-world" graph is one where the characteristic path length scales logarithmically with the number of nodes [83], and a "navigable" graph is one where the number of nodes visited by best-first search grows in $O(\log N)$ [52]. These properties combine to support efficient search. In [57], a navigable SWG (NSW) is constructed by inserting nodes one at a time into the graph and connecting each one to its k nearest neighbors that are already in the graph. In [58], a hierarchical NSW (HNSW) graph is constructed by assigning each node to a random maximum "layer", chosen from an exponentially decaying distribution, and then adding it to each of the underlying layers while connecting it to its k nearest neighbors in each layer to avoid the degree explosion problem of a flat graph.

2.3 Query Optimization and Execution

For predicated queries, plan enumeration and selection may depend on *hybrid operators*. We introduce these first before discussing enumeration, selection, and query execution.

Hybrid Operators. A hybrid operator works by combining vector index scan with attribute search, and there are generally two approaches. In *block-first scan*, parts of the vector index are prevented from exploration ("blocked") based on their associated attribute values, and then the index scan proceeds as normal over the blocked index. The main consideration is how to efficiently perform the blocking. In *visit-first scan*, the scan operator itself is modified so that it takes into consideration the attribute values on the visited vectors during index traversal. The main consideration is how to design the operator so that search is efficient.

[1] http://github.com/google-research/google-research/tree/master/scann

[2] http://arxiv.org/abs/1609.07228

(1) Block-First Scan. For block-first scan, some techniques perform *online blocking*, where the index is blocked at query time. This allows it to be more flexible to different queries but adds query latency overhead. To reduce the overhead, [6, 79, 84] first construct a bitmask using traditional attribute filtering techniques. This bitmask is then used during index scan to quickly determine if a vector is blocked. Other techniques perform *offline blocking*, where the index is blocked beforehand. In [6, 79], the vector collection is pre-partitioned along attributes so that at query time, only the relevant partition needs to be searched. In [3, 43, 87], online blocking can cause a graph-based index to become disconnected, complicating the search procedure. Thus, these techniques construct the graph in a way that can prevent disconnections from occurring by considering attribute values during edge selection.

(2) Visit-First Scan. For visit-first scan, if the predicate is highly selective, then the scan may backtrack in order to fill the result set. To avoid backtracking, several techniques infuse the best-first search operator for a graph-based index with attribute information so that the scan prefers nodes that satisfy the predicate [43, 87].

Plan Enumeration. There may be multiple query plans for any given query. For example for non-predicated queries, a VDBMS may support multiple search indexes, and each can potentially be used to answer the query. For predicated queries, there are three broad approaches. The predicate can be applied first, for example via block-first scan, known as "pre-filtering"; applied onto the result set after the search, known as "post-filtering"; or applied while the search is being conducted, for example via visit-first scan, known as "single-stage filtering". To enumerate all the possible plans, a VDBMS may *predefine* all the plans for every supported query type, or it may *automatically* enumerate the plans.

(1) Predefined. Some VDBMSs predefine a *single plan* for each query type, removing the overhead of plan selection. This is useful for workloads with specific needs. For example for e-commerce, Vearch [12, 54] executes all predicated search queries using post-filtering. Post-filtering risks returning fewer than k results for a (c, k)-search query, but for e-commerce, this is acceptable. Other VDBMSs such as Weaviate [13] execute all predicated search queries using pre-filtering. Still others such as Euclid [14] only support a single search index at a time, and all search queries are executed using the index. There are also some VDBMSs that predefine *multiple plans* for different queries. For example, AnalyticDB-V [84] supports four different plans for predicated queries based on index availability, and then a cost-based optimizer is used to select the plan.

(2) Automatic. Some VDBMSs that are based on relational systems, such as `pgvector` [7] and PASE [90], take advantage of the underyling relational optimizer to automatically perform plan enumeration. This is achieved by extending the relational language with vector search operators, including index scan operators.

Plan Selection. Plan selection aims to select the optimal query plan, usually the minimum latency plan. This is achieved using *rule-based* or *cost-based* selection.

(1) Rule Based. For VDBMSs that only support a small number of plans, simple rules can be sufficient for plan selection. For example, Qdrant [3] and Vespa [4] use predicate selectivity estimates as a heuristic for deciding whether to perform pre-filtering, post-filtering, or single-stage brute-force scan.

(2) Cost Based. Other VDBMSs use a cost model to select the plan with minimum cost. For example, AnalyticDB-V [84] and Milvus [6, 79] devise costs for several vector operators in order to use a linear cost model that aggregates the I/O and computation cost of each plan operator in order to yield a total plan cost.

Query Execution. Several techniques aim to exploit *hardware acceleration* in order to speed up search queries. Additionally, many VDBMSs adopt *distributed* architecture, allowing them to scale to larger datasets and workloads. Finally, as some vector indexes are hard to update, some VDBMSs adopt mechanisms for *out-of-place* updates that enable high write throughput without sacrificing search throughput.

(1) Hardware Acceleration. Modern machines equipped with SIMD and GPUs support high amounts of data parallelism. In [26, 27], in addition to parallelizing similarity projection, memory retrieval is identified as a key bottleneck during IVFADC index scan, and a technique is introduced to reduce the amount of retrievals by exploiting the storage capacity of SIMD registers along with SIMD shuffle. In [50], a similar technique is introduced for GPU registers. In Milvus [6, 79], the small register size is identified as a limiting factor, and a technique based on multiple rounds is introduced in order to support (c, k)-search queries with large k.

(2) Distributed Search. Several VDBMSs adopt a distributed architecture where the vector collection is sharded and replicated and scatter-gather is used to answer vector search. To partition the collection into shards, the vectors can be equally partitioned or the partitioning can be index guided, such as placing all vectors in the same bucket into the same partition for a table-based index. These can also take advantage of disaggregated architecture [80] or cloud functions [73] to increase elasticity.

(3) Out-of-Place Updates. Many vector indexes are hard to update due to their data dependent nature, leading to long query latencies. To address this issue, some VDBMs perform updates out-of-place, either by applying updates asynchronously over replicas [10, 13, 84], storing updates in a temporary structure and then applying them in bulk at a more appropriate time [10, 84], or using dedicated structures such as a log-structured merge (LSM) tree [6, 45, 79].

2.4 Existing Systems

Native. Native systems aim at providing dedicated vector capabilities. Some native systems target *mostly vector* workloads, whereas others target *mostly mixed* workloads that consist of queries over both vectors and attributes.

(1) Mostly Vector. Mostly-vector systems aim to support efficient vector queries with limited support for attribute-related capabilities such as predicated search. As a result, these systems tend to be streamlined for vectors. The interface tends to be a simple API, and there is usually no query parser or rewriter, reducing the query processing overhead. Usually there is also no optimizer as all queries are handled in the same way by a single search index. Some systems like EuclidesDB [14] and Vald [10] do not support predicated search at all, whereas others like Vearch [12], Pinecone [5], and Chroma [15] support it with a single predefined plan.

(2) Mostly Mixed. Mostly-mixed systems aim to support a wider variety of queries and query plans, including attribute-only queries in some cases. This makes them more complex, featuring more sophisticated data and storage models as well as query optimization. For example, Milvus [6, 79], Qdrant [3], and Manu [45] all include query optimizers. Systems like Weaviate [13], NucliaDB [16], and Marqo [11] support other queries in addition to vector queries. Weaviate uses a graph data model over data entities and supports graph-based queries, while NucliaDB and Marqo support non-vector keyword queries in addition to combined similarity and keyword queries via dense and sparse multi-vectors.

Extended. When deploying a native system is difficult[3] or when it simply lacks capabilities, an extended system may be preferred. Extended systems aim to offer best of both worlds by integrating vector capabilities into a non-vector NoSQL or relational system.

(1) NoSQL. Many NoSQL systems have or are planned to be extended to support vector capabilities, including Vespa [4], Cassandra [53], Spark-based Databricks, MongoDB, CosmosDB, and Redis. This is typically achieved by adding vector search indexes into the storage engine. For example Cassandra plans to integrate an HNSW index into the storage layer, implement scatter-gather to support distributed vector search, and extend the Cassandra query language with vector-related operators.

(2) Relational. If a similarity score is available, a relational system can already answer vector queries via brute-force scan, as demonstrated by SingleStore [17, 66]. Moreover, after the built-in query optimizer is made aware of vector operators, it can be used for plan enumeration and selection as in PASE [90] and `pgvector` [7] which also introduce search indexes into PostgreSQL. Other relational systems with similar vector support include AnalyticDB-V [84], Clickhouse, and MyScale. A recent work [92] analyzed the potential limitations of using relational databases to support vector search and explored ways to overcome these limitations.

Search Engines and Libraries. Some applications may not require a fully managed system. In this case, a few popular search engines, including Apache Lucene [8], Elasticsearch [9], OpenSearch [18], and Solr [19] now support vector search and may be preferred. There are also several libraries that offer low-level search capability, including Microsoft SPTAG [20] and Meta Faiss [1].

2.5 Benchmarks

We discuss two comprehensive benchmarks. In [55], a wide variety of ANN techniques and search indexes are reimplemented and evaluated across a range of workloads. The datasets go up to thousands of dimensions and are collected from real-world image, text, video, and audio collections. In [29], existing implementations that are available for use are compared, leaving intact implementation-specific techniques. It also includes several commercial VDBMSs in the evaluation, and the results are available online[4].

2.6 Challenges and Open Problems

Despite this progress, several challenges remain.

(1) Similarity Score Selection. Approaches for similarity score selection remain lacking. EuclidesDB [14] offers one approach where many scores and embedding models can be queried at once, but the ultimate decision is left to the user. In [82], the score is dynamically adjusted based on the query, but the technique is limited to social media recommendation.

(2) Operator and Index Design. Existing hybrid operators are limited to only small number of attribute categories, and there is a need for more powerful operators that can support more complex hybrid queries. Additionally, aside from DiskANN [74] and SPANN [32], there remains a lack of disk-based indexes.

(3) Cost Estimation. The cost of block-first scan and visit-first scan is difficult to estimate due to the uncertain effect from blocking. For tree and graph-based indexes, blocking and predicate failures can lead to excessive backtracking, increasing query latency. On the other hand, post-filtering leads to uncertainty in the size of the result set as fewer than k results may be returned for a (c, k)-search query due to the filter. One way to avoid this is to retrieve αk results before the filter so that there are enough results after the filter is applied [79, 84], but how to tune α remains unclear.

(4) Security and Privacy. Many VDBMSs are offered as managed cloud services that may be vulnerable to attack. For multi-tenant systems, there is a need for techniques that can support private and secure vector operations, such as secure k-NN search [88, 93].

(5) Incremental Search. Applications such as e-commerce rely on incremental search, where the result set is seamlessly fetched in parts. Techniques such as [39] exist for incremental k-NN search but it is unclear how to support this search within vector indexes.

(6) Multi-Vector Search. Applications such as facial recognition or contextual text retrieval make use of multi-vector search. While aggregate scores can be used to support these applications, they require significant computations and increase query latency. Generic multi-attribute top-k techniques also cannot easily be applied to vector indexes [79], and new techniques are needed.

3 PRESENTERS

James Jie Pan is a postdoctoral researcher at Tsinghua University. His research interest is vector data management.

Jianguo Wang is a tenure-track assistant professor in the Department of Computer Science at Purdue University. His research interests include disaggregated databases and vector databases. He is a recipient of the NSF CAREER Award.

Guoliang Li is a professor in the Department of Computer Science at Tsinghua University in Beijing, China. His research interests include machine learning for databases, database systems, and data cleaning and integration.

ACKNOWLEDGEMENT

This paper was sponsored by National Key Research and Development Program of China (2023YFB4503600), and NSF of China (61925205, 62232009, 62102215). Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806.

[3] http://arxiv.org/abs/2308.14963
[4] http://ann-benchmarks.com