The VLDB Journal (2024) 33:1591–1615
https://doi.org/10.1007/s00778-024-00864-x

REGULAR PAPER

Survey of vector database management systems

James Jie Pan[1] · Jianguo Wang[2] · Guoliang Li[1]

Received: 12 October 2023 / Revised: 7 June 2024 / Accepted: 24 June 2024 / Published online: 15 July 2024

Abstract
There are now over 20 commercial vector database management systems (VDBMSs), all produced within the past five years. But embedding-based retrieval has been studied for over ten years, and similarity search a staggering half century and more. Driving this shift from algorithms to systems are new data intensive applications, notably large language models, that demand vast stores of unstructured data coupled with reliable, secure, fast, and scalable query processing capability. A variety of new data management techniques now exist for addressing these needs, however there is no comprehensive survey to thoroughly review these techniques and systems. We start by identifying five main obstacles to vector data management, namely the ambiguity of semantic similarity, large size of vectors, high cost of similarity comparison, lack of structural properties that can be used for indexing, and difficulty of efficiently answering "hybrid" queries that jointly search both attributes and vectors. Overcoming these obstacles has led to new approaches to query processing, storage and indexing, and query optimization and execution. For query processing, a variety of similarity scores and query types are now well understood; for storage and indexing, techniques include vector compression, namely quantization, and partitioning techniques based on randomization, learned partitioning, and "navigable" partitioning; for query optimization and execution, we describe new operators for hybrid queries, as well as techniques for plan enumeration, plan selection, distributed query processing, data manipulation queries, and hardware accelerated query execution. These techniques lead to a variety of VDBMSs across a spectrum of design and runtime characteristics, including "native" systems that are specialized for vectors and "extended" systems that incorporate vector capabilities into existing systems. We then discuss benchmarks, and finally outline research challenges and point the direction for future work.

Keywords Vector data management · Similarity search · *k* nearest neighbor · Approximate nearest neighbor · Nearest neighbor index

1 Introduction

The rise of large language models (LLMs) [68] for tasks like information retrieval [33], along with the growth of unstructured data for applications such as e-commerce and recommendation platforms [61, 122, 130], calls for new *vector database management systems* (VDBMSs) that can deliver traditional capabilities such as query optimization, transactions, scalability, fault tolerance, privacy, and security, but for unstructured data.

For LLMs specifically, commercial LLM-based chatbots are known to suffer from hallucinations, high usage costs, and forgetfulness, which can all be potentially addressed by VDBMSs. For example to address hallucinations and forgetfulness, one solution is retrieval-augmented generation (RAG), where prompts and answers are augmented with information retrieved from a VDBMS [33, 75].[1] To address high usage costs, commercial VDBMSs such as Zilliz[2] offer a semantic cache [36] where a user prompt is first checked against a VDBMS for similar prompts before a costly submission to the chatbot. Aside from LLMs, there are also many non-LLM applications, including image and video search,

✉ Guoliang Li
liguoliang@tsinghua.edu.cn

James Jie Pan
jamesjpan@tsinghua.edu.cn

Jianguo Wang
csjgwang@purdue.edu

1 Department of Computer Science and Technology, Tsinghua University, Beijing, China

2 Department of Computer Science, Purdue University, West Lafayette, Indiana, USA

[1] See also http://arxiv.org/abs/2402.01763.

[2] http://zilliz.com/.

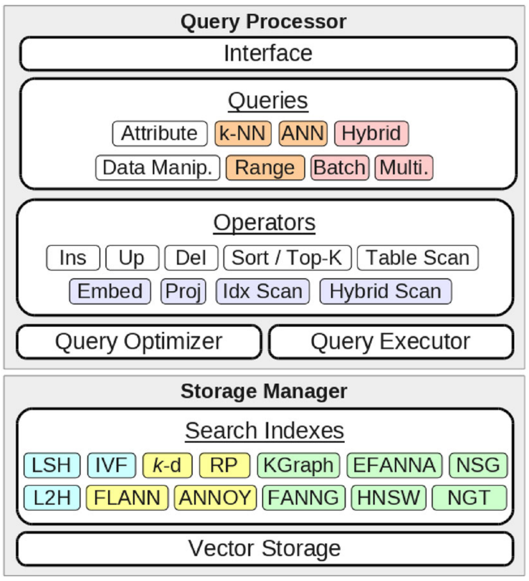

Fig. 1 Components of a VDBMS

drug discovery, facial and voice recognition, recommendation systems for e-commerce, and others [113].

As unstructured data are not represented by attributes from a fixed schema, they are retrieved not through structured queries but through similarity search [90]. To support this type of search, entities such as images and documents are first encoded into D-dimensional feature vectors via an embedding model before being stored inside a VDBMS. The dual-encoder or *dense retrieval* model [40, 70] describes this process.

Consequently, as shown in Fig. 1, the modules in a VDBMS consist of a *query processor*, which includes query definitions, logical operators, their physical implementations, and the query optimizer; and the *storage manager*, which maintains the search indexes and manages the physical storage of the vectors. The designs of these modules affect the characteristics and features of the VDBMS. Some applications, such as those based on LLMs, are read-heavy, requiring high query throughput and low latency. Others, such as e-commerce, are also write-heavy, requiring high write throughput, in addition to transaction isolation requirements and data consistency. There are also several different query types, such as predicated or non-predicated vector queries, that require different operators and query planning. Moreover, cloud-based VDBMSs like Zilliz, Weaviate [1], Pinecone [2], and others may aim to provide high scalability and elasticity along with availability and fault tolerance. Given an application, developing a suitable VDBMS therefore requires understanding the landscape of techniques and how they affect the system.

While there are mature techniques for processing structured data, this is not the case for vector data. We present five key obstacles. (1) *Ambiguous Search Criteria.* Structured queries use boolean predicates to precisely capture the intent of the user query, but vector queries rely on semantic similarity that is hard to unambiguously state. While many similarity scores exist, a chosen score may not align precisely with user intent. (2) *Expensive Comparisons.* Attribute predicates (e.g. $<$, $>$, $=$, and $\in$) can mostly be evaluated in $O(1)$ time, but a similarity comparison typically requires $O(D)$ time, where D is the vector dimensionality. (3) *Large Size.* A structured query usually only accesses a small number of attributes, making it possible to design read-efficient storage structures such as column stores. But vector search requires full feature vectors, moreover a single vector may even span multiple data pages, making disk retrievals more expensive while also straining memory. (4) *Lack of Structural Properties.* Structured attribute data possess certain properties, such as being sortable or ordinal, that lend themselves to traditional indexing techniques. But vectors have no equivalent properties, making it hard to adapt attribute index design principles to vectors.[3] (5) *Incompatibility with Attributes.* Structured queries over multiple attribute indexes can use simple set operations (e.g. $\cup$, $\cap$) to collect intermediate results into the final result set. But vector indexes typically stop after finding k most similar vectors, and combining these with the results from an attribute index scan can lead to fewer than expected results. On the other hand, modifying the scan to account for attribute predicates can degrade index performance. How to efficiently and effectively support these "hybrid" queries remains unclear.

There are now a variety of techniques that have been developed around these issues, aimed at achieving low query latency, high result quality, and high throughput while supporting large numbers of vectors. Some of these are results of decades of study on similarity search. Others, including hybrid query processing, indexes based on vector compression, techniques based on hardware acceleration, and distributed architectures are based on recent work. In this paper, we start by surveying these techniques from the perspective of a generic VDBMS, dividing them into those that apply to query processing and those that apply to storage and indexing. Query optimization and execution are treated separately from the core query processor. Following these discussions, we apply our understanding of these techniques to characterize existing VDBMSs.

Query Processing The query processor mainly deals with how to specify the query criteria and how to execute search queries. For the former, a variety of similarity scores, query types, and query interfaces are available. For the latter, a number of vector and search operators are available, in addi-

[3] Note that systems for certain types of vector retrieval, such as time series or spatial databases, can often exploit correlations and patterns in the series [53] as well as low-dimensional indexing techniques [103].

tion to search algorithms. We discuss the query processor in Sect. 2.

Storage and Indexing The storage manager mainly deals with how to organize and store the vector collection to support efficient and accurate search, predominantly through indexes. We classify indexes into *table-based* indexes such as E^{2}LSH [47], SPANN [43], and IVFADC [66], that are generally easy to update; *tree-based* indexes such as FLANN [92], RPTree [45, 46], and ANNOY [3] that aim to provide logarithmic search; and *graph-based* indexes such as KGraph [51], FANNG [64], and HNSW [85] that perform well empirically but are less understood theoretically.

To address the difficulty of indexing vector collections, existing indexes rely on randomization [28, 46, 47, 51, 65, 92, 111, 120], learned partitioning [66, 86, 92, 107, 124], and what we refer to as "navigable" partitioning [50, 84, 85]. To deal with large storage size, several techniques have been developed for indexes over compressed vectors, including quantization [66, 86, 108, 127, 130], as well as disk-resident indexes [43, 60]. We discuss indexing in Sect. 3.

Query Optimization The query optimizer mainly deals with plan enumeration, plan selection, and physical execution. To support hybrid queries, several hybrid operators have been developed based on what we refer to as "block-first" scan [60, 122, 130] and "visit-first" scan [133]. There are several techniques for enumeration and selection, including rule and cost-based selection [122, 130]. We discuss optimization in Sect. 4.

Query Execution For execution, many VDBMSs take advantage of distributed architectures to scale out vector search. There are also several techniques aimed at supporting high throughput updates, namely based on maintaining fast and slow writeable structures. To speed up local queries, storage locality of large vectors can be exploited to design hardware accelerated operators, taking advantage of capabilities such as processor caches [122], SIMD [31, 32, 122], and GPUs [67]. We discuss execution in Sect. 5.

Current Systems We classify existing VDBMSs into *native* systems which are designed specifically around vector management, including Vearch [77], Milvus [122], and Manu [61]; *extended* systems which add vector capabilities on top of an existing NoSQL or relational data management system, including AnalyticDB-V (ADBV) [130] and PASE [136]; and *search engines and libraries* which aim to provide search capability only, such as Apache Lucene [4], Elasticsearch [5], and Meta Faiss [6]. Native systems tend to favor high-performance techniques targeted at specific capabilities, while extended systems tend to favor techniques that are more adaptable to different workloads but are not necessarily the fastest. We survey current systems in Sect. 6.

Related Surveys A high-level survey is available that mostly focuses on fundamental VDBMS concepts and use cases [112]. Likewise, some tutorials are available that focus specifically on similarity search [100, 101]. We complement these works by focusing on specific problems and techniques related to vector data management as a whole. Surveys are also available covering data types that are related to vectors, such as time series and strings [52, 53], but not supported by VDBMSs.

For the remaining sections, we briefly discuss benchmarks in Sect. 7, followed by a summary of research challenges and open problems in Sect. 8. We conclude the survey in Sect. 9.

2 Query processing

Query processing in a VDBMS starts from a *query definition* describing the nature of the query. Once a query is received by the system, a number of operators are executed over the data collection to process the query. In particular, *search operators* find and retrieve relevant vectors from the collection, and the characteristics of the operator depends on the *search algorithm*.

2.1 Query definition

The main feature of vector search queries is a *similarity score* ranking vectors that the user intends to retrieve. There are also several *query types*, including data manipulation queries and other forms of search queries. Queries are conveyed through a *query interface*.

2.1.1 Similarity scores

A similarity score $f : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ maps two D-dimensional vectors, $\mathbf{a}$ and $\mathbf{b}$, onto a scalar, $f(\mathbf{a}, \mathbf{b})$. For some scores, larger values indicate greater similarity. For other scores that are based on distance functions,[4] values closer to 0 indicate greater similarity. Table 1 lists several common scores.

Definition 1 (*Hamming*) $d(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} \delta_{a_i b_i}$, where δ is the Kronecker delta.

Table 1 Common similarity scores

Type	Score	Metric	Complexity	Range
Sim	Inner Prod	✗	$O(D)$	$\mathbb{R}$
	Cosine	✗	$O(D)$	$[-1, 1]$
Dist	Minkowski	✓	$O(D)$	$\mathbb{R}^+$
	Hamming	✓	$O(D)$	$\mathbb{N}$

[4] A distance function obeys the metric axioms of identity ($d(\mathbf{a}, \mathbf{a}) = 0$), positivity ($d(\mathbf{a}, \mathbf{b}) > 0$ if $\mathbf{a} \neq \mathbf{b}$), symmetry ($d(\mathbf{a}, \mathbf{b}) = d(\mathbf{b}, \mathbf{a})$), and triangle inequality ($d(\mathbf{a}, \mathbf{c}) \leq d(\mathbf{a}, \mathbf{b}) + d(\mathbf{b}, \mathbf{c})$ for any three vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$).

The Hamming distance counts the number of differing dimensions between vectors $\mathbf{a}$ and $\mathbf{b}$. Other similarity scores start from the inner product, typically the dot product.

Definition 2 (*Inner Product*) $f(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} a_i b_i$,

or $f(\mathbf{a}, \mathbf{b}) = \langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a} \cdot \mathbf{b}$. Quantity $\sqrt{\langle \mathbf{a}, \mathbf{a} \rangle}$ defines the magnitude, $\|\mathbf{a}\|$. Euclidean distance between $\mathbf{a}$ and $\mathbf{b}$ is given by $\|\mathbf{a} - \mathbf{b}\|$. The dot product projects $\mathbf{a}$ onto $\mathbf{b}$, scaling the result by the magnitude of $\mathbf{b}$. If magnitude is unimportant, $\mathbf{a}$ and $\mathbf{b}$ can be normalized by $\hat{\mathbf{a}} = \mathbf{a}/\|\mathbf{a}\|$ and $\hat{\mathbf{b}} = \mathbf{b}/\|\mathbf{b}\|$. Then,

Definition 3 (*Cosine Similarity*) $f(\mathbf{a}, \mathbf{b}) = \langle \hat{\mathbf{a}}, \hat{\mathbf{b}} \rangle$

or $f(\mathbf{a}, \mathbf{b}) = \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{\|\mathbf{a}\|\|\mathbf{b}\|}$, equivalent to the angle between $\mathbf{a}$ and $\mathbf{b}$. Arbitrary p-norms, $\|\mathbf{x}\|_p = (|x_1|^p \ldots |x_D|^p)^{1/p}$, induce the Minkowski distance which generalizes Euclidean distance.

Definition 4 (*Minkowski*) The p-order Minkowski distance is $d(\mathbf{a}, \mathbf{b}) = (\sum_{i=1}^{n} |a_i - b_i|^p)^{1/p}$.

All positive and integer values of p yield metrics.

2.1.2 Query types

A VDBMS supports *data manipulation* queries that insert, update, and delete vectors to and from the collection as well as *vector search* queries that aim to return a subset of the collection satisfying the search criteria.
Data Manipulation Queries In traditional database systems, data is manipulated directly. But in a VDBMS, feature vectors represent actual entities, and they can be manipulated directly or indirectly.

An *embedding model* maps real-world entities (e.g. images) to feature vectors. Under direct manipulation, users freely manipulate the values of the vectors, and maintaining the model is the responsibility of the user. This is the case for systems such as PASE [136] and `pgvector` [7]. For indirect manipulation, vectors are hidden from users. The vector collection appears as a collection of entities, not vectors, and users manipulate the entities. The VDBMS is responsible for the model, which can be user-provided, for example through a user-defined function (UDF) as in Vald [8], or selected from a menu of pre-trained models. Pinecone [2], for instance, supports a large number of pre-trained `*2vec` models[5] by connecting to special providers[6] via REST API. There is extensive literature on designing embedding models, and we refer interested readers to [98].

To process data manipulation queries over vectors, different techniques exist depending on the storage structure.

[5] *E.g.* http://github.com/MaxwellRebo/awesome-2vec.

[6] *E.g.* http://huggingface.co.

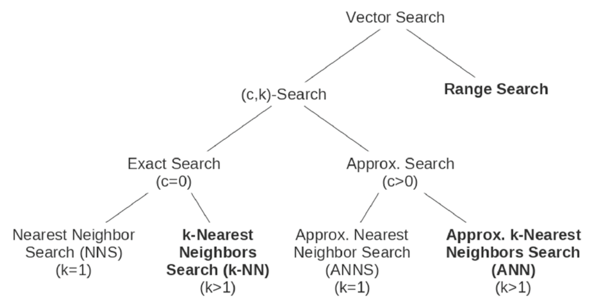
Vector Search
(c,k)-Search
Range Search
Exact Search (c=0)
Approx. Search (c>0)
Nearest Neighbor Search (NNS) (k=1)
k-Nearest Neighbors Search (k-NN) (k>1)
Approx. Nearest Neighbor Search (ANNS) (k=1)
Approx. k-Nearest Neighbors Search (ANN) (k>1)

Fig. 2 Basic search queries

Most VDBMSs heavily rely on vector indexes, and we discuss how to update these indexes in Sect. 3. At the same time, some VDBMSs adopt distributed architectures or utilize fast and slow storage structures in addition to indexes. We discuss data manipulation techniques for these systems in Sect. 5.
Basic Search Queries As shown in Fig. 2, there are several types of search queries, but not all VDBMSs support all types. Search queries can be viewed as either similarity maximization or equivalently as distance minimization with respect to a query vector, $\mathbf{q}$. We take the latter for the following definitions.

Most VDBMSs support "nearest neighbor" queries where the aim is to retrieve vectors from a collection S that are physical neighbors of $\mathbf{q}$ in the vector space. These queries may aim to return exact or approximate nearest neighbors, and may also specify the number of neighbors to return. We refer to these as (c, k)-*search* queries, where c indicates the approximation degree and k is the number of neighbors.

Out of these, most VDBMSs support the *approximate* k-*nearest neighbors* (ANN) query, which returns k vectors from S that are within a radius, centered over $\mathbf{q}$, of c times the distance between $\mathbf{q}$ and its closest neighbor.

Definition 5 (*ANN*) Find a k-size subset $S' \subseteq S$ such that $d(\mathbf{x}', \mathbf{q}) \le c(\min_{\mathbf{x} \in S} d(\mathbf{x}, \mathbf{q}))$ for all $\mathbf{x}' \in S'$.

Recent efforts have focused on $c > 1, k > 1$, as it supports a variety of modern applications[7].

On the other hand, a *range* query is parameterized by a radius, r, instead of the number of neighbors.

Definition 6 (*Range*) Find $\{\mathbf{x} \in S \mid d(\mathbf{x}, \mathbf{q}) \le r\}$.

[7] Other combinations of c and k have been historically important but are less prevalent in the modern literature. For the special case $k = 1$, this query has been called *approximate nearest neighbor search* which we abbreviate as ANNS and is covered extensively in [29]. When $c = 1$, this is known as an *exact* query. The case $c = 1, k = 1$ corresponds to *nearest neighbor search* (NNS) [65], and when $c = 1, k > 1$, the query is popularly called a k-*nearestneighbors* (k-NN) query. We note that there is also a large literature on the *maximum inner product search* (MIPS) problem, which is the NNS query but over inner products. We refer interested readers to [114] for an overview.

Range queries can be useful when the goal is to return *all* items within a similarity threshold, not just the top-k. For example, in a biological tissue database a range query can be used to return all images of cells with a similar cell type [41].

Some VDBMSs also support variations on these basic query types.

Predicated Search Queries In a predicated search query, or "hybrid" query, each vector is associated with a set of attribute values, and a boolean predicate over these values must evaluate to true for each record in the result set.[8] These queries are prevalent in item recommendation systems for e-commerce. For example:

Example 1 A hybrid k-NN query written in SQL is:

```
select * from items where price < 100
order by distance(query) limit k;
```

Here, `distance` is a distance function parameterized by the vectorized `query`, and every member of the result set must satisfy the conditions of being among the `k` nearest and of obeying the predicate, `price` < `100`.

Batched Queries For batched queries, a number of queries are revealed at once and the VDBMS can answer them in any order. These queries are especially suited to hardware-accelerated techniques [67, 122].

Multi-Vector Queries Some VDBMSs also support multi-vector search queries via aggregate scores. Sometimes, a single real-world entity is represented by multiple vectors. For example for facial recognition [122], a face may be represented by multiple images taken from different angles, leading to m feature vectors $\mathbf{x}_1 \ldots \mathbf{x}_m$[9]. One way of approaching this problem is to use an aggregate score that defines how to combine individual scores $f(\mathbf{x}_1, \mathbf{q}) \ldots f(\mathbf{x}_m, \mathbf{q})$ to yield a single value that can be compared. Examples are the mean aggregate $1/m \sum_{i=1}^{m} f(\mathbf{x}_i, \mathbf{q})$ and the weighted sum.

2.1.3 Query interfaces

Later on in Sect. 6, we classify existing VDBMSs into native systems that are designed specifically around vector data management and extended systems that add vector capabilities on top of an existing NoSQL or relational system.

For query interfaces, native and NoSQL VDBMSs tend to rely on simple APIs with a small number of permitted operations[10]. On the other hand, extended VDBMSs built over relational systems tend to take advantage of SQL extensions. In `pgvector` [7], a k-NN or ANN query is expressed as:

```
select * from items order by
embedding <-> [3,1,2] limit 5;
```

The syntax `R <-> s` returns the Euclidean distance between all the tuples of `R` and vector `s`, and other distance functions are supported via other symbols.

Similarly, range queries are expressed using `where`:

```
select * from items where
embedding <-> [3,1,2] < 5;
```

2.2 Operators

Queries are processed by executing a number of operators. A *vector operator* acts on individual vectors or the collection as a whole while a *search operator* finds and retrieves certain vectors from the collection based on the query definition.

Vector Operators Data manipulation queries are handled using insert, update, and delete vector operators. Typically vectors are stored in a single logical table, and a data manipulation operator modifies the table. Physically, the implementations of these operators depends on the underlying structure. For example, the physical table may be stored locally, in memory or on disk, or distributed across shards and replicas. The table can also be stored within an index. Each structure requires corresponding operators that can act on the structure. Some VDBMSs also support an embedding operator that takes an entity, such as a document or image, and produces a vector representation of the entity. These systems typically expose a set of record-level operators for indirect manipulation instead of exposing raw vector operators. Most VDBMSs also support linear algebraic operators and simple arithmetic operators for calculating similarity scores.

Search Operators To answer search queries, most VDBMSs implement a projection operator that projects each vector onto its similarity score against a query vector. For range queries, the operator may additionally push satisfying vectors into the result set upon calculating the score. For (c, k)-search queries, the operator can be combined with a sort operator in order to retrieve the highest scoring vectors, or be combined with a top-k data structure such as a priority queue.

[8] This type of query is called differently in different systems. In some VDBMSs such as [130], this is called a "hybrid" query. In others such as [9], this is called a "filtered" query.

[9] As another example, some VDBMSs such as [1] offer a "hybrid" search, not to be confused with the search in Note 8, that combines dense feature vectors with sparse term vectors. The sparse vector is scored separately, e.g. by weighted term frequency, and then combined with the feature vector similarity score to yield a final aggregate score.

[10] For example, Chroma [10] offers a Python API with nine administrative commands (e.g. `list_collections`) in addition to nine data commands: `count`, `add`, `get`, `peek`, `query`, `modify`, `update`, `upsert`, and `delete`.

Projection is sufficient for answering all range or (c, k)-search queries, and it can also be used in combination with a predicate checking operator to answer predicated queries [11]. But for an N-size collection, the complexity of answering a query using projection is dominated by the $O(DN)$ similarity calculations, potentially leading to long query latencies. To speed up the search, most VDBMSs rely on index-based search operators. In Sect. 3, we introduce the most common index structures along with respective scan operators. Then in Sect. 4, we describe how special "hybrid" operators can be used in combination with indexes to speed up predicated queries.

2.3 Search algorithms

A number of search algorithms exist with different accuracy and performance characteristics[11]. For example, a brute-force search via projection offers exact query answers but with $O(DN)$ complexity while index-based search offers approximate answers but usually sub-linear complexity. Some VDBMSs let the user choose the algorithm while others choose the algorithm automatically via an optimizer, discussed further in Sect. 4.

Similarity search has a long history and many theoretical results are known, especially for low-dimensional vectors[12]. For high-dimensional vectors, locality-sensitive hashing (LSH) [65] is perhaps the most well understood technique. But more recent efforts aimed at addressing certain limitations of LSH have led to other techniques, including tree and graph-based indexes that will be discussed in Sect. 3.

2.4 Discussion

The primary difference between a VDBMS and other database systems is the reliance on a similarity score. Given the same vector collection, different scores can lead to different rankings, and how closely the ranking matches the user intent[13] depends on the choice of score, presenting a challenge for *score selection* in addition to *score design*. Note that in most existing VDBMSs, vectors are stored in single collections and there is no support for join.

Score Selection If all the vectors have unit magnitude, then cosine angle is equal to dot product, and the Euclidean distance is proportional to both. In this case, the industry practice is to use cosine similarity as it is easier to calculate compared to Euclidean distance. But when vectors do not possess unit magnitude, it may not be obvious which score to use. The choice also depends on the nature of the embeddings themselves. For example, one use case recommends using dot product in order to capture item popularity, as the particular embedding model is known to yield long vectors for frequent items[14, 15].

User intent is also conveyed through an embedding vector (i.e. the query vector) that allows for similar ambiguity as the choice of score, leading to the idea of *query semantics* [112]. Hence, we imagine that future solutions will be more holistic, considering this problem from all aspects beyond score selection. For example, EuclidesDB [12] allows users to conduct the same search but over multiple embedding models and scores in order to identify the most semantically meaningful settings. As another example, [91] proposes interactively refining the query vector to achieve better semantic alignment.

Score Design The fact that different scores lead to different rankings opens the question of whether there is an "optimal" score that depends on the collection and workload. This question has led to the idea of *learned scores*. Applying a linear transformation over the vector space adjusts the relative proximities of the vectors without changing the vectors themselves. The distance of two vectors in the transformed space can be calculated using the Mahalanobis formula,[16] and finding a suitable transformation is one of the goals of *metric learning* [25, 88, 118, 139].

There is also a well-known fact that when D grows beyond around 10 dimensions, and when the dimensions are independent and identically distributed (IID), the cosine angles and Euclidean distances between the two farthest and two nearest vectors approach equality as the variance nears zero

[11] Accuracy is usually defined in terms of precision and recall. Precision is the ratio between the number of relevant results in the result set over the size of the result set, and recall is defined as the number of retrieved relevant results over all possible relevant results. For example in a k-NN query, the precision is $k'/|S'|$, where k' is the number of true nearest neighbors in S', and the recall is k'/k. Performance is usually defined in terms of query latency and throughput.

[12] Algorithms with $O(\log N)$ query times and $O(N)$ storage are known for $D = 1$ (e.g. binary search trees) and $D = 2$ [80]. For the latter case, k-d trees [37] are particularly well known, and the query complexity is $O(\sqrt{N})$. For $D \geq 3$, sub-linear search performance is much harder to obtain. In the general case, k-d trees offer $O(DN^{1-1/D})$ query complexity [73], which tends toward $O(DN)$ as D grows. On the other hand, [87] offers $O(D^{O(1)} \log N)$ query complexity but requires super-polynomial $O(N^{O(D)})$ storage. The modern belief is that even a fractional power of N query complexity cannot be obtained *unless* storage cost is worse than $N^{O(1)} D^{O(1)}$ [29, 104].

[13] As a simple example, consider items of different shapes and colors. A user may intend for items with similar shapes to be considered more similar than items with similar colors.

[14] http://developers.google.com/machine-learning/clustering/similarity/measuring-similarity.

[15] See also [113] where geometric distance yields different rankings depending on the meaning of the feature dimensions.

[16] For any positive semi-definite matrix M, $d(\mathbf{a}, \mathbf{b}) = \sqrt{(\mathbf{a} - \mathbf{b})^\top \mathsf{M} (\mathbf{a} - \mathbf{b})}$.

[39]. The effect of this *curse of dimensionality* is that vectors become indiscernible.[17]

Fortunately for many real-world datasets, the intrinsic dimensionality[18] is sufficiently low or the vectors are not IID, avoiding the curse. Even so, there have been attempts at attacking the curse which have led to using other Minkowski distances, such as the Manhattan distance ($p = 1$) or the Chebyshev distance ($p = \infty$), in an effort to recover discernability [26, 89].

3 Indexing

While all (c, k)-search and range queries can be answered by brute-force search, the $O(DN)$ complexity is prohibitive for large D and N. Instead, vector indexes speed up queries by minimizing the number of comparisons. This is achieved by partitioning S so that only a small subset is compared, and then arranging the partitions into structures that can be easily explored.

Unlike attribute indexes, vector indexes cannot take advantage of structural properties such as being sortable or ordinal. To achieve high accuracy, these indexes rely on novel techniques which we refer to as *randomization*, *learned partitioning*, and *navigable partitioning*. The large physical size of vectors also leads to use of compression, namely a technique called *quantization*, as well as *disk resident* designs. Additionally, the need to support predicated queries has led to special *hybrid operators* for indexes, which we discuss in Sect. 4.

Partitioning Techniques

- *Randomization* exploits probability amplification to discriminate similar vectors from dissimilar ones.
- *Learning-based* techniques partition S along hidden internal structure.
- *Navigable* indexes are designed so that different regions of S can be easily traversed.

Some indexes require regular maintenance to ensure efficient and accurate search. The maintenance characteristics of a vector index depend highly on the *data-dependency*[19] of the partitioning strategy. Some partitioning strategies are data-independent, where the partitioning rules are independent of S[20]. But the majority are data-dependent, meaning that they derive from the distribution of S. For example, some indexes partition S into k-means clusters that must be found beforehand. If subsequent updates to the index alter the distribution, then it may eventually become unbalanced, degrading efficiency and recall. In many cases, these indexes can only be maintained by peroidically rebuilding the index.

Storage Techniques

- *Quantization* involves a function, called a quantizer, which maps a vector onto a more space-efficient representation. Quantization is usually lossy, and the aim is to minimize information loss while simultaneously minimizing storage cost.
- *Disk resident* designs additionally aim to minimize the number of I/O retrievals in contrast to memory resident indexes which only minimize the number of comparisons.

In this section, we examine the main techniques for several common indexes. One particular index may use a combination of techniques, and so we classify indexes based on their structure and then point out which techniques are used in which index. There are three basic structures: *tables* divide S into buckets containing similar vectors; *trees* are a nesting of tables; and *graphs* connect similar vectors with virtual edges that can then be traversed. All of these structures are capable of achieving high query accuracy but with different construction, search, and maintenance characteristics. Following these technical details, in the closing discussion we offer recommendations to VDBMS users and conclude with open problems.

3.1 Tables

The main consideration for table-based indexes is the design of the bucketing hash function. The most popular table-based indexes for VDBMSs tend to use randomization and learned partitions, as shown in Table 2. For randomization, techniques based on LSH [27–29, 47, 65, 74, 83] are popular due to robust error bounds. For learned partitions, learning-to-hash (L2H) [124] directly learns the hash function, and

Table 2 Representative table-based indexes

Type	Index	Hash function
LSH	E^{2}LSH [47]	Rand. hyperplanes
	`IndexLSH`	Rand. bits
	FALCONN [28]	Rand. balls
L2H	SPANN [43]	Nearest centroid
Quant	`SQ`	Nearest discrete value
	`PQ`	Nearest centroid product
	IVFSQ	Nearest centroid
	IVFADC [66]	Nearest centroid

[17] A diagram is given in [93].

[18] A formal definition is given in [45].

[19] This term appears in [103] in the context of spatial databases.

[20] *E.g.* a spatial grid.

indexes based on quantization [66, 86, 108, 127, 130] typically uses k-means [38] to learn clusters of similar vectors.

Each of these indexes have similar construction and search characteristics. For construction, each vector is hashed into one of the buckets, and the complexity is $O(DN^{1+\epsilon})$, $0 \leq \epsilon \leq 1$. For LSH, each vector is hashed multiple times, leading to $\epsilon > 0$. For quantization-based approaches, k-means multiplies the complexity by a constant factor. For search, $\mathbf{q}$ is hashed onto a key and then the corresponding bucket is scanned. Hashing is generally on the order of $O(D)$. Usually only a small fraction of S is scanned, yielding a complexity of $O(DN^{\epsilon})$, including the cost of hashing and bucket scan.

For data manipulation queries, table-based indexes support vector insertion using the same hash functions used during construction, and vector deletion is handled by a search followed by physical deletion. But for data-dependent indexes, the hash functions themselves may need to be relearned following a series of out-of-distribution updates, requiring the index to be rebuilt. All the nearest-centroid methods listed in Table 2 are data-dependent as they all depend on k-means clustering. Once new centroids are discovered, all the vectors in S must be reinserted into the new buckets, otherwise query accuracy may suffer as near vectors may no longer occupy the same buckets.

3.1.1 Locality sensitive hashing

Locality sensitive hashing [65, 74] provides tunable performance with error guarantees, but it can require high redundancy in order to boost accuracy, increasing query and storage costs relative to other techniques.

In a "family" of hash functions $H = \{h : S \to U\}$, if $d(\mathbf{x}, \mathbf{q}) \leq r_1$, then $\Pr_H(h(\mathbf{x}) = h(\mathbf{q})) \geq p_1$, and if $d(\mathbf{x}, \mathbf{q}) \geq r_2$, then $\Pr_H(h(\mathbf{x}) = h(\mathbf{q})) \leq p_2$, for any $r_1, r_2, \mathbf{x} \in S$, and $\mathbf{q}$. A tunable family $G = \{g : S \to U^K\}$ is derived by letting $g(\mathbf{x})$ return the concatenation of $h_i(\mathbf{x})$ for i between 1 and some constant K.

The table is constructed by hashing each $\mathbf{x} \in S$ into each of the L hash tables using $g_1 \ldots g_L$. Typically, L is set to $L = O(1/p_1^K)$ with K set to $\lceil \log_{1/p_2} N \rceil$ [29].[21] Letting $\rho = \log(1/p_1)/\log(1/p_2)$ yields $L = O(N^{\rho}/p_1)$. The storage complexity is $O(LDN)$ which is $O(DN^{1+\rho})$ after substitution. In the practical case where $p_1 > p_2$, the value of ρ is between 0 and 1.

When a query appears, it is hashed using the L hash functions sampled from G, and collisions are kept as candidate neighbors. The candidates are then re-ranked or discarded based on true distances to $\mathbf{q}$. The query complexity is dominated by the L hash evaluations, which is $O(DN^{\rho})$.

[21] The exact value depends on accuracy and performance needs of the application [27].

When r_1 is set to $\min_{\mathbf{x} \in S} d(\mathbf{x}, \mathbf{q})$ and r_2 is set to cr_1, the c guarantee is relative to the minimum distance. This is useful when the query is static across the workload, but is hard to generalize over dynamic online queries. Hence for an index designed around some given hash family, not all queries may have similar candidate sets, making it hard to control precision and recall. Multi-probe LSH [83] is one attempt at addressing this issue by scanning multiple buckets at a time, thereby spreading out the search.

We mention a few popular LSH schemes. The first two are data-independent and require no rebalancing. In *E^2LSH*, each g is an $O(D)$ projection onto a random hyperplane. This achieves $\rho < 1/c$ [47]. The `IndexLSH` scheme is based on binary projections [6]. There have also been efforts at designing data-dependent hash families to yield lower ρ. *FALCONN* implements an LSH hash family based on spherical LSH [28]. The dataset is first projected onto a unit ball and then recursively partitioned into small overlapping spheres. The ρ value is $1/(2c^2 - 1)$. Other families are given in [29].

3.1.2 Learning to hash

Learning-based techniques aim to directly learn suitable mappings without resorting to hash families. For example, spectral hashing [131] designs hash functions based on the principal components of the similarity matrix of S. In [105], hash functions are modeled using neural networks. In SPANN [43], vectors are hashed to the nearest centroid following k-means. These techniques tend to require lengthy training and are sensitive to out-of-distribution updates, and they are not widely supported in VDBMSs. We point readers to [124] for a survey of techniques.

While many vector indexes are meant to reside in memory in order to take advantage of fast reads and writes, SPANN is designed to be disk-resident, and hence it employs several techniques for minimizing costly I/O. In order to support fast look-ups, hash keys are stored in memory while buckets are stored on disk. To control the amount of I/O per query, a hierarchical bucketing scheme modulates the final bucket sizes. Buckets overlap based on relative neighborhood rules [115] in order to reduce I/O for queries located near bucket boundaries.

3.1.3 Quantization

One of the main criticisms of LSH is that the storage cost can be large due to the use of multiple hash tables. For in-memory VDBMSs, large memory requirements can be impractical. While some efforts aim at reducing these storage costs [140], other efforts have targeted vector compression using quantization [66, 108].

Many of these techniques use k-means centroids as hash keys. A large K number of centroids can modulate the chance

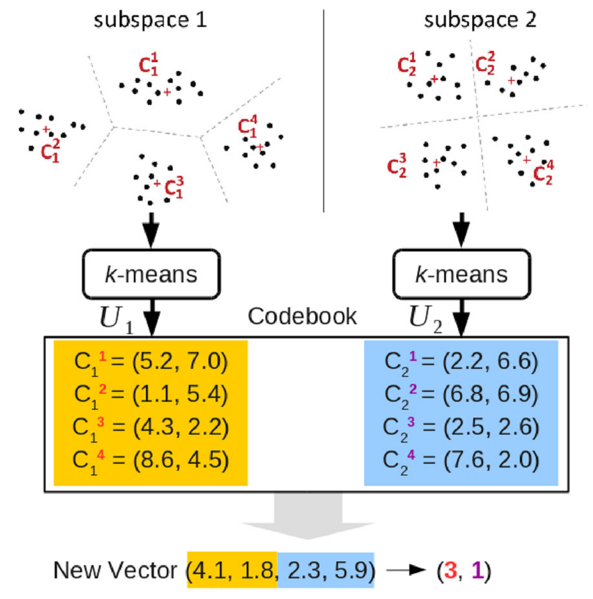

Fig. 3 In this example from [127], each 4-dimensional vector is divided into two 2-dimensional sub-spaces

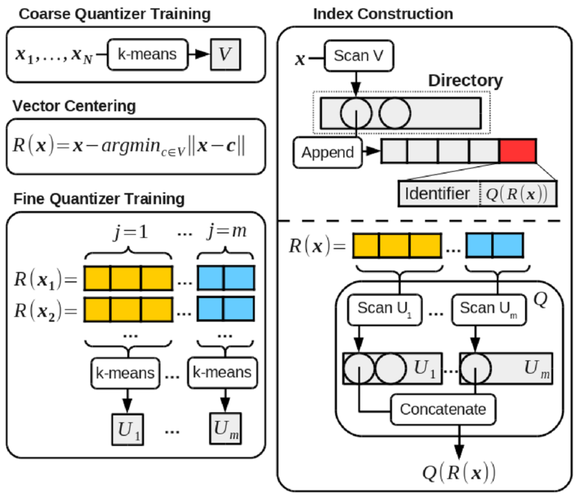

Fig. 4 Construction of an IVFADC index

of collisions, keeping buckets reasonably small and speeding up search. Normally, k-means terminates once a locally optimal set of centroids is found, with complexity $O(DN \cdot K)$ per iteration.

But large values of K make k-means expensive. *Product quantization* exploits the fact that the cross product of m number of (D/m)-dimensional spaces is a space of D dimensions, so that by setting $U = \prod_{j=1}^{m} U_i$, then $U \in \mathbb{R}^D$ when $U_j \in \mathbb{R}^{D/m}$. This means that to yield a count of K centroids, only $K^{1/m}$ centroids need to be found per U_j. Moreover as each U_j belongs to a lower dimensional space, the running time of k-means per U_j is reduced. The new complexity is $O(m)O(\frac{D}{m}NK^{1/m}i)$.

In practice, each U_j is constructed via k-means over the collection of sub-vectors $\{(x_i)_{i=(j-1)D/m+1}^{jD/m} \mid \mathbf{x} \in S\}$[22], and the set of all U_j is known as the "codebook". Vector $\mathbf{x}$ is then quantized by splitting it into m sub-vectors, $\mathbf{x}'_j$, finding the nearest centroid in U_j to $\mathbf{x}'_j$ for each $j \in 1\dots m$, and then concatenating these centroids. Each vector is thus stored using $m\log_2(D/m)$ bits, and the time complexity is $O(m)O(DK^{1/m})$. An example is shown in Fig. 3.

Various techniques such as Cartesian k-means [94], optimized PQ (OPQ) [58], hierarchical quantizers [135], and anisotropic quantizers [62] have been developed based on this idea, offering around 60% better recall in the best cases at the cost of additional processing. A survey of these techniques is given in [86]. The storage cost can also be reduced by constant factors [127]. There are also efforts at designing quantizers using techniques other than k-means. For example, [57] selects points along the surface of a hypersphere to serve as the codebook, deriving bounds on the quantization error.

We describe some quantization-based indexes. Faiss [6] supports a number of "flat" indexes where each vector is directly mapped onto its compressed version, without any bucketing. The standard quantizer index, `SQ`, performs a bit-level compression, for instance by mapping 64-bit doubles onto 32-bit floats. The `PQ` index directly maps each vector onto its PQ code. For *IVFSQ*, the vectors are compressed using `SQ` and bucketed to their nearest centroid. Even with product quantization, training a PQ quantizer over S can still be time consuming. To reduce this cost, *IVFADC* first buckets vectors using k-means over a small number of centroids, and then trains a PQ quantizer by sampling a few vectors from each of the buckets. To allow a single quantizer to apply to all the buckets, each vector $\mathbf{x}$ is subtracted from its bucket key, resulting in a "residual" vector $R(\mathbf{x})$ which is then used to train the quantizer. The full workflow is shown in Fig. 4. During search, query $\mathbf{q}$ is directly compared against the quantized vectors in the bucket that $\mathbf{q}$ maps onto. As $\mathbf{q}$ itself is not quantized, the comparison is referred to as an "asymmetric distance computation" (ADC).

For IVFADC, many distance calculations are likely to be repeated during bucket scan since many vectors may share the same PQ centroids. These calculations can be avoided by first computing $\|\overline{\mathbf{q}_j} - \mathbf{c}\|^2$ for all $\mathbf{c} \in U_j$ and for all $j \in 1\dots m$, where $\overline{\mathbf{q}_j}$ is the jth sub-vector of $\mathbf{q}$ [86]. This preprocessing step takes $O(m)O(\frac{D}{m}K')$, where K' is the number of centroids in U_j. But afterwards, ADC can be performed using just m look-ups, reducing bucket scan from $O(DN)$ to $O(mN)$.

Example 2 Below is the ADC look-up table when U is divided into m subsets, and where each subset contains K'

[22] The notation $(x_i)_{i=1}^{D}$ expands to $x_1x_2\dots x_D$.

Table 3 Representative tree-based indexes

Index	Splitting plane	Splitting point
k-d tree [37]	Axis parallel	Median
PKD-tree [107]	Principal dim	Median
FLANN [92]	Random principal dim	Median
RPTree [45, 46]	Random plane	Median + offset
ANNOY [3]	Random plane	Random median

centroids. Here, $\overline{\mathbf{q}_j}$ is the jth sub-vector of query $\mathbf{q}$, and $\mathbf{c}_i^j$ is the ith centroid in the jth subset of U.

$$\overbrace{\begin{matrix} d(\overline{\mathbf{q}_1}, \mathbf{c}_1^1) & \cdots & d(\overline{\mathbf{q}_1}, \mathbf{c}_{K'}^1) \\ \vdots & \ddots & \vdots \\ d(\overline{\mathbf{q}_m}, \mathbf{c}_1^1) & \cdots & d(\overline{\mathbf{q}_m}, \mathbf{c}_{K'}^1) \end{matrix}}^{U_1}, \ldots, \overbrace{\begin{matrix} d(\overline{\mathbf{q}_1}, \mathbf{c}_1^m) & \cdots & d(\overline{\mathbf{q}_1}, \mathbf{c}_{K'}^m) \\ \vdots & \ddots & \vdots \\ d(\overline{\mathbf{q}_m}, \mathbf{c}_1^m) & \cdots & d(\overline{\mathbf{q}_m}, \mathbf{c}_{K'}^m) \end{matrix}}^{U_m}$$

We mention one other technique. In ADBV [130], each bucket is further divided into finer sub-buckets in order to avoid accessing multiple full buckets. The resulting structure is called a "Voronoi Graph Product Quantization" (VGPQ) index.

3.2 Trees

For tree-based indexes, the main consideration is the design of the splitting strategy used to recursively split S into a search tree.

A natural approach is to split based on distance. The main techniques includes pivot-based trees [42, 143], such as VP-tree [137] and M-tree [44], k-means trees [92], and trees based on deep learning [76]. Other basic techniques are described in [106]. But while these trees are effective for low D, they suffer from the curse of dimensionality when applied to higher dimensions.

High-D tree-based indexes tend to rely on randomization for performing node splits. In particular, "Fast Library for ANN" (FLANN) [13, 92] combines randomization with learned partitioning via principal component analysis (PCA), extending the PKD-tree technique from [107], and "ANN Oh Yeah" (ANNOY) [3] is similar to the random projections tree (RPTree) from [45, 46]. These trees are summarized in Table 3.

The generic tree construction algorithm, from [45], is restated below:

```
procedure MakeTree($S$)
    if $|S| \le \tau$ then return Leaf
    end if
    Rule $\leftarrow$ ChooseRule($S$)
    LeftTree $\leftarrow$ MakeTree($\{\mathbf{x} \in S \mid \text{Rule}(\mathbf{x}) \text{ is true}\}$)
    RightTree $\leftarrow$ MakeTree($\{\mathbf{x} \in S \mid \text{Rule}(\mathbf{x}) \text{ is false}\}$)
    return (Rule, LeftTree, RightTree)
end procedure
```

The complexity is characteristically $O(DN \log N)$, ignoring any preprocessing costs. More precise bounds for several trees are given in [102].

Most trees are able to return exact query results by performing backtracking, where neighboring leaf nodes are also checked during the search. However, this is inefficient [129], leading to a technique called *defeatist search* [46]. In this procedure, the tree is traversed down to the leaf level, and all vectors within the leaf covering $\mathbf{q}$ are returned immediately as the nearest neighbors. While defeatist search does not guarantee exact results, there is no backtracking, and the complexity is $O(D \log N)$.

For data manipulation queries, insertions require $O(D \log N)$ on average and $O(DN)$ in the worst case. But all of the indexes in Table 3 are data-dependent. Splitting planes are derived directly from S and node splits are determined during construction, and so far there are no methods for rebalancing nodes after a number of out-of-distribution updates.

3.2.1 Non-random trees

Many high-D trees derive from k-d tree which splits along medians while rotating through dimensions:

```
procedure ChooseRule($S$)
    $i \leftarrow l \mod D$ where $l$ is the current depth
    Rule := $(x_i \le \text{median}(\{y_i \mid \mathbf{y} \in S\}))$
    return Rule
end procedure
```

This rule has the effect of fixing the splitting planes parallel to the dimensional axes [45, 102].

3.2.2 Random trees

If certain dimensions explain the variance more than others, then the intrinsic dimensionality is lower than D. But in this case, k-d tree is unable to partition along these dimensions, leaving it susceptible to the curse of dimensionality. This limitation has led to the discovery of more adaptive splitting strategies.

Principal Component Trees A principal component tree is a k-d tree that is constructed by first rotating S so that the axes are aligned with the principal components of S. The principal dimensions need to be found beforehand using principal component analysis (PCA). The complexity of this step is $O(D^2N + D^3)$. In *PKD-tree*, the splitting plane is selected by rotating through the principal dimensions [107]. To try to find better splits, *FLANN* [92] splits along random principal dimensions instead of strictly rotating through.

Random Projection Trees On the other hand, random splitting planes can be used to adapt to the intrinsic dimensionality without expensive PCA. *RPTree* [45, 46] extends the idea of randomly rotated trees explored in [117] by introducing random splits in addition to random splitting planes. The principle follows from spill trees [81], where partitions are allowed to overlap. In RPTree, perturbed median splits simulate the effects of overlapping splits but with less storage cost. The splitting rule is [46]:

```
procedure CHOOSERULE($S$)
    $\mathbf{u} \leftarrow$ A vector from the unit sphere
    $\beta \leftarrow$ A number from $[0.25, 0.75]$
    $v \leftarrow$ The $\beta$ fractile of $\pi_{\mathbf{u}}(S)$
    Rule := $(\mathbf{x} \cdot \mathbf{u} \leq v)$
    return Rule
end procedure
```

The values of $\mathbf{u}$, β, and $\mathbf{v}$ are uniformly chosen at random. The operation $\pi_{\mathbf{u}}(S)$ performs a projection of S onto $\mathbf{u}$. Variable v represents the perturbed median split, with $\beta = 0.5$ yielding the true median. Instead of splitting on the β fractile, ANNOY splits along the median of two random values sampled from $\pi_{\mathbf{u}}(S)$, which is simpler to compute. Several theoretical results[23][24] are known for RPTree, but it is not clear if these apply to ANNOY.

A forest of random trees can be used to improve recall. We mention that RPTree incurs a storage overhead of $O(DN)$ compared to k-d tree and FLANN due to storing the D-dimensional projection vectors, and this cost can be substantial for in-memory forests. In [69], several techniques are introduced to reduce this cost to $O(D \log N)$, for example by combining projections across the trees in a forest.

3.3 Graphs

A graph-based index is constructed by overlaying a graph on top of the vectors in $\mathbb{R}^D$ space so that each node v_i is positioned over the vector $\mathbf{x}_i$ within the space. This induces distances over the nodes, $d(v_i, v_j) = d(\mathbf{x}_i, \mathbf{x}_j)$, which are then used to guide vector search along the edges. An example is shown in Fig. 5.

[23] The rate at which RPTree fails to return the true nearest neighbor, $\mathbf{x}'$, to query $\mathbf{q}$ is bounded by the "potential" of the query over S, defined as $1/(N-1)\sum_{i=2}^{N}\|\mathbf{q}-\mathbf{x}'\|^2/\|\mathbf{q}-\mathbf{x}_i\|^2$. A detailed proof is available in [46].

[24] During search, query $\mathbf{q}$ must be projected onto each unit vector at every level during traversal, leading to a search complexity of $O(D \log N)$. In [102], this is improved to $O(D \log D + \log N)$ by using a circular rotation which can be applied in $O(D \log D)$ time and achieves a similar effect as random projections. The trinary projection (TP) tree introduced in [121] similarly targets expensive $O(D)$ projections. Instead of projecting onto random or principal vectors, the splitting strategy partitions onto principal trinary vectors, which are vectors consisting of only -1, 0, or 1. The principal trinary vectors can be approximated in $O(D)$ time. The search complexity remains $O(D \log N)$ but with smaller constant factors.

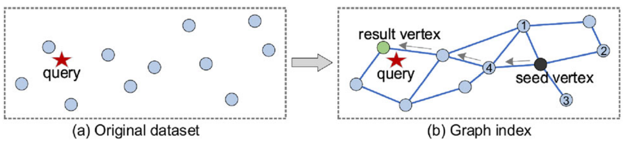

Fig. 5 In this example from [126], a graph is overlayed on top of the original dataset (**a**). Search begins from a "seed" vertex and is guided along the edges until it reaches the nearest node (vector) to the query (**b**)

Table 4 Representative graph-based indexes

Type	Index	Initialization	Construction
KNNG	KGraph [51]	Random KNNG	Iterative refine
	EFANNA	Random trees	Iterative refine
MSN	FANNG [64]	Empty graph	Random trial
	NSG [56]	Approx. KNNG	Fixed trial
	Vamana [111]	Random graph	Fixed trial
SW	NSW [84]	Empty graph	One-shot refine
	HNSW [85]	Empty graph	One-shot refine

The main consideration for these indexes is edge selection, in other words deciding which edges should be included during graph construction.

Graph-based indexes encapsulate all the partitioning techniques. Many graphs rely on random initialization or random sampling during construction. The k-nearest neighbor graph (KNNG) [55, 96] associates each vector with its k nearest neighbors through an iterative refinement process similar to k-means and which we consider to be a form of unsupervised learning. Other graphs, including monotonic search networks (MSNs) [50] and small-world (SW) graphs [84, 85], aim to be highly navigable, but differ in their construction. The former tend to rely on search trials that probe the quality of the graph [56, 64, 111] while the latter use a heuristic procedure which we refer to as "one-shot refine". Table 4 shows several graph indexes.

For data manipulation queries, we note that most of the indexes in Table 4 assume a static vector collection and so are not designed to support updates. Meanwhile, even as NSW [84] and HNSW [85] are constructed incrementally and are thus able to support dynamic insertions, and while recent implementations now also offer deletions via tombstoning,[25] they still show signs of data-dependency.[26] Unfortunately, there seem to be no methods for rebalancing these indexes, other than to rebuild them from scratch. There are also other efforts to design graph indexes that do support updates, such

[25] See http://github.com/nmslib/hnswlib.

[26] Deteriorating search accuracy has been observed for dynamic collections, even when the updates are mere re-insertions of previous deletions; see http://arxiv.org/abs/2105.09613

as [142] which uses restricted neighborhood updates, but these have yet to be incorporated into VDBMSs.

3.3.1 k-nearest neighbor graphs

In a KNNG, each node v_i is connected to k nodes representing the nearest neighbors to $\mathbf{x}_i$ [55]. For batched queries, $\mathbf{q}$ can be considered as a member of S, and a KNNG built over S allows exact k-NN search in $O(1)$ time through a simple look-up.

A KNNG can also be used to answer queries where $\mathbf{q} \notin S$. The basic idea is to recursively select node neighbors that are nearest to $\mathbf{q}$, starting from initial nodes, and add them into the top-k result set. The search complexity depends on the number of iterations before the result set converges. The search can start from multiple initial nodes, and if there are no more node neighbors to select, it can be restarted from new initial nodes [119].

A KNNG can be exact or approximated with a technique which we refer to as "iterative refine".
Exact An exact KNNG can be constructed by performing a brute force search N number of times, giving a total complexity of $O(DN^2)$. Unfortunately, there is little hope for improvement, as it is believed that the complexity is bounded by $N^{2-o(1)}$ [132]. An $O(N \log N)$ algorithm is given in [116] but with a constant factor that is $O(D^D)$. The algorithm in [97] achieves an empirical complexity of $O(N^{2-\epsilon})$, where $0 < \epsilon < 1$. This finding suggests the existence of efficient practical algorithms, despite the worst-case bounds.
Iterative Refine An approximate KNNG can be obtained by iteratively refining an initial graph. We give two examples. The *NN-Descent* (KGraph) method [51] begins with a random KNNG and iteratively refines it by examining the neighbors of the neighbors of each node v_i, replacing edges to v_i with edges to these second-order neighbors that are closer. When the dataset is growth restricted,[27] then each iteration is expected to halve the radius around each node and its farthest neighbor. This property leads to fast convergence, with empirical times on the order of $O(N^{2-\epsilon})$ for $0 < \epsilon < 1$. Instead of starting from a random KNNG, *EFANNA*[28] uses a forest of randomized k-d trees to build the initial KNNG. Doing so is shown to lead to higher recall and faster construction as it can quickly converge to better local optima. A similar approach is taken in [120] but where the tree is constructed via random hyperplanes.

[27] A growth restricted dataset is one where the number of neighbors of each node is bounded by a constant as the radius about the node expands.

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

3.3.2 Monotonic search networks

A KNNG is not guaranteed to be connected. Disconnected components complicates the search procedure for online queries by requiring restarts to achieve high accuracy [96, 119]. But by adding certain edges so that the graph is connected, it becomes possible to follow a single path beginning from any initial node and arriving at the nearest neighbor to $\mathbf{q}$.

A search path $v_1 \dots v_m$ is *monotonic* if $d(v_i, \mathbf{q}) > d(v_{i+1}, \mathbf{q})$ for all i from 1 to $m-1$. An MSN is a graph where the search path discovered by a "best-first" search, in which the neighbor of v_i that is nearest to $\mathbf{q}$ is greedily selected, is always monotonic. This property implies a monotonic path for every pair of nodes in the graph and that the graph is connected.

The search complexity depends on the sum of the out-degrees over the nodes in the search path. If there are few total edges in the graph, then the search complexity is likely to be small. The minimum-edge MSN that guarantees exact NNS is believed to be the Delaunay triangulation [93]. But constructing a triangulation requires at least $\Omega(N^{\lceil D/2 \rceil})$ time [54], impractical for large N and D. As a result, several approximate methods have been developed, but these necessarily sacrifice the search guarantee.

In the early work by [50], an MSN is constructed in polynomial time by refining a sub-graph of the Delaunay triangulation called the relative neighborhood graph (RNG) [115]. The RNG itself is not monotone, but it can be constructed in $O(DN^{2-o(1)} \log^{1-o(1)} N)$ under $\mathbb{R}^D$ Euclidean distance [109]. Even so, the $N^{2-o(1)}$ term makes this approach impractical for large N.

Instead of repeatedly scanning the nodes, *search trials* can be used to probe the quality of the graph. Depending on the path taken by best-first search, new edges are added so that a monotonic path exists between source and target. The algorithm is shown below.

```
procedure MakeMSN($S$)
    $G \leftarrow$ InitializeGraph(S)
    repeat
        $(s, t) \leftarrow$ ChooseSourceTargetPair($S$)
        $P \leftarrow$ GetSearchPath($G$, $(s, t)$)
        UpdateOutNeighbors($G, t, P$)
        for each $p \in P$ do
            UpdateOutNeighbors($G, p, t$)
        end for
    until Terminate()
    return $G$
end procedure
```

For InitializeGraph, some indexes begin with an empty graph [64], random graph [111], or approximate KNNG [56]. Simple graphs can be initialized quickly but more complex graphs may offer better quality. For ChooseSourceTargetPair, one way is to select random pairs [64], while another is to designate a node as the source for all search trials [56, 111]. We refer to these techniques as *random* and *fixed* trials, respectively.

Random Trial Indexes based on random trials are constructed over a large number of iterations, each one leading to closer approximations of an MSN. The construction time is thus adjustable with respect to the quality of the graph. In the *Fast ANN Graph* (FANNG) [64], graph construction terminates after a fixed number of trials, e.g. $50N$. The UpdateOutNeighbors routine adds an edge between t and the nearest node in the search path, $p^* \in P$, and then prunes out-neighbors of p^* based on "occlusion" rules derived from the triangle inequality so as to limit out-degrees. The empirical storage and search complexities are reported to be on the order of $O(DN^{1-\epsilon})$.

Fixed Trial In fixed trial construction, all trials are conducted from a special designated source node, sometimes called the "navigating" node. This node also serves as the source for all online queries. The index is constructed by conducting one trial to each node in a single pass over S. The construction complexity is generally about $O(DN^{1+\epsilon} \log N^{\epsilon})$ where the logarithmic term represents the cost of the search trials. The *Navigating Spreading-Out Graph* (NSG) index [56] starts from an approximate KNNG. For UpdateOutNeighbors, it uses an edge selection strategy based on lune membership, similar to [50]. To guarantee that all targets are reachable from the navigating node, it overlays a spanning tree to connect any unreachable targets. To speed up construction, *Vamana* [111] begins with a random graph instead of an approximate KNNG, and instead of checking lune membership, UpdateOutNeighbors uses a simple distance-based threshold similar to FANNG.

In [111], a disk-resident index called *DiskANN* is introduced. Like SPANN, it first partitions the vector collection into overlapping clusters using k-means. Then for each cluster, it constructs a Vamana index over the cluster. Each vector is compressed using PQ and stored in memory while each Vamana graph is stored on disk. The compressed vectors are used to guide the search while neighborhoods are fetched from disk. To reduce I/O overhead, multiple neighborhoods of the vectors comprising the search frontier are retrieved at once in a technique called "beam search". Full-precision vectors are stored alongside neighborhoods and are also retrieved in the same I/O operation, allowing them to be cached in memory and used for final re-ranking.

3.3.3 Small world graphs

A graph is *small-world* if the length of its characteristic path grows in $O(\log N)$ [128]. A *navigable* graph is one where the length of the search path found by the best-first search algorithm scales logarithmically with N [71]. A graph that is both navigable and small-world (NSW) thus possesses a search complexity that is likely to be logarithmic, even in the worst case.

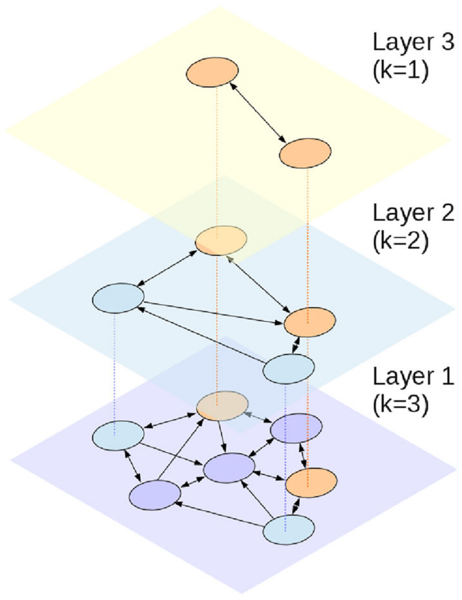

Fig. 6 Example HNSW index. Out-degrees in each layer are bounded by k to modulate search complexity. Vertical edges allow traversal down the layers while horizontal edges allow traversal within a layer

An NSW graph can be constructed using a procedure which we call *one-shot refine* and detailed in [84]. Nodes are sequentially inserted into the graph, and when a node is inserted, it is connected to its k nearest neighbors already in the graph. While NSW offers search paths that scale in $\log N$, the out-degrees also tend to scale in the logarithm of N, leading to polylogarithmic search complexity. In [85], a *hierarchical NSW* (HNSW) graph is given which uses randomization in order to restore logarithmic search. During node insertion, the node is assigned to all layers below a randomly selected maximum layer, chosen from an exponentially decaying distribution so that the size of each layer grows logarithmically from top to bottom. Within each layer, the node is connected to its neighbors following the NSW procedure, but where the out-degrees are bounded. Best-first search proceeds from the top-most layer. Figure 6 shows an example.

3.4 Discussion

Given the variety of indexes, we first give some recommendations for *index selection*. Many of these indexes were historically developed for single-use applications and not large-scale multi-user VDBMSs over dynamic data collections, leading to problems for *index design*.

Table 5 Representative search indexes

Structure	Index[a]	Partitioning	Residence	Complexity[b]			Update[c]	Error Bound
				Constr	Space	Query		
Table	E^2LSH [47]	Space	Mem	Med.	High	Med.	Y	✓
	FALCONN [30]	Space	Mem	Med.	High	Med.	R	✓
	*SQ	Discrete	Mem	Med.	Low	Med.	Y	✗
	*PQ	Clustering	Mem	Med.	Low	Med.	R	✗
	*IVFSQ	Clustering	Mem	Med.	Low	Med.	R	✗
	*IVFADC [66]	Clustering	Mem	Med.	Low	Med.	R	✗
	SPANN [43]	Clustering	Disk	Med.	Med.	Med.	R	✗
Tree	FLANN [92]	Space	Mem	High	High	Low	R	✗
	RPTree [45, 46]	Space	Mem	Low	High	Low	R	✓
	*ANNOY	Space	Mem	Low	High	Low	R	✗
Graph	NN-Descent (KGraph) [51]	Proximity	Mem	Med.	Med.	Med.	N	✗
	EFANNA	Proximity	Mem	High	Med.	Low	N	✗
	FANNG [64]	Proximity	Mem	High	Med.	Med.	N	✗
	NSG [56]	Proximity	Mem	High	Med.	Low	N	✗
	Vamana (DiskANN) [111]	Proximity	Disk	Med.	Med.	Low	N	✗
	*HNSW [85]	Proximity	Mem	Low	Med.	Low	Y	✗

[a] An asterisk (*) indicates supported by more than two commercial VDBMSs
[b] Based on theoretical results reported by authors, empirical results reported by authors, or our own analysis when no results are reported. Key for complexity columns, with $0 \leq \epsilon \leq 1$ and a natural constant, K: *Construction.* High=worse than $O(DN^{1+\epsilon})$, Med.=$O(DN^{1+\epsilon})$, Low=$O(DN \log N)$; *Size.* High=$O(DN \cdot K)$ or worse, Med.=$O((D+K) \cdot N)$, Low=$O(N \log D)$; *Query.* High=worse than $O(DN^{\epsilon})$, Med.=$O(DN^{\epsilon})$, Low=$O(D \log N)$
[c] Y=data-independent updates; R=updates with rebalancing; N=no updates

Index Selection As Table 5 shows, HNSW offers many appealing characteristics. It is easy to construct, has reasonable storage requirements, can be updated, and supports fast queries. It therefore comes as no surprise that it is supported by many commercial VDBMSs. The storage cost may still be a concern for very large vector collections, but there are ways to address this.[29]

Even so, there are cases where other indexes may be more appropriate. For batched queries or workloads where the queries belong to S, KNNGs may be preferred, as once they are constructed, they can answer these queries in $O(1)$ time. KGraph is easy to construct, but EFANNA is more adaptable to any online queries. For online workloads, the choice rests on several factors. If error guarantees are important, then an LSH-based index or RPTree can be considered. If memory is limited, then a disk-based index such as SPANN or DiskANN may be appropriate. If the workload is write-heavy, then table-based indexes may be preferred, as they generally can be efficiently updated. Out of these, E^2LSH is data-independent and requires no rebalancing. For read-heavy workloads, tree or graph indexes may be preferred, as they generally offer logarithmic search complexity.

[29] For example, Weaviate [1] allows constructing HNSW graphs over vectors that have been compressed with PQ.

Aside from these indexes, there have also been efforts at mixing structures in order to achieve better search performance. For example, the index in [35] and the *Navigating Graph and Tree* (NGT) index [14] use a tree to initially partition the vectors and then use a graph index over each of the leaf nodes.

Index Design A multi-user VDBMS may require some degree of transaction isolation. While isolation guarantees can be implemented at the system level as we will discuss in Sects. 5 and 6, it may also be possible to design concurrent indexes at the storage level. Unfortunately, designing such an index remains an open problem. For example, Faiss implements many vector indexes but none support concurrent updates. We are aware of one implementation of HNSW that supports concurrent reads and writes via local locks [15].

Similarly, many of these indexes were not designed for dynamic collections in the first place, and there remains a need for indexes that can be easily updated.

4 Query optimization

A query plan in a VDBMS is typically represented as a directed acyclic graph of operators used to answer a given query. There may be multiple possible plans per query, and the goal of the query optimizer is to select the optimal one, typically the latency minimizing plan. For now, query optimization in a VDBMS is mostly limited to predicated queries, as non-predicated queries can often be answered by a single index scan, leaving no room for optimization.

The first step is *plan enumeration* followed by *plan selection*. For predicated queries, vector indexes and attribute filters cannot be easily combined, resulting in the development of new *hybrid* operators.

4.1 Hybrid operators

Predicated queries can be executed by either applying the predicate filter before vector search, known as "pre-filtering"; after the search, known as "post-filtering"; or during search, known as "single-stage filtering". If the search is index-supported, then a mechanism is needed to inform the index that certain vectors are filtered out. For pre-filtering, *block-first scan* works by "blocking" out vectors in the index before the scan is conducted [60, 122, 130]. The scan itself proceeds as normal but over the non-blocked vectors. For single-stage filtering, *visit-first scan* works by scanning the index as normal, but meanwhile checking each visited vector against the predicate conditions [133].

Block-First Scan Blocking can be done online at the time of a query, or if predicates are known beforehand, it can be done offline. For online blocking, the aim is to perform the blocking as efficiently as possible. In ADBV [130] and Milvus [16, 122], a technique using bitmasks is given. A bitmask is constructed using traditional attribute filtering techniques. Then, during index scan, a vector is quickly checked against the bitmask to determine whether it is "blocked". Blocking can also be performed offline. In Milvus, S is pre-partitioned along attributes that are expected to be predicate targets. When a query arrives, it can then be executed on the relevant partition using a normal index scan.

For graph-based indexes, blocking can cause the graph to become disconnected. In Filtered-DiskANN [60] and elsewhere [9, 133], disconnections are prevented in the first place by strategically adding edges based on the attribute category of adjoining nodes. In [145], each edge of an HNSW index is labeled with a range window so that vectors which obey a one or two-sided range intersecting the window can be found by traversing the edge. These labels are determined offline by sorting the dataset beforehand and then inserting vectors into an HNSW in sorted order. The use of sorted insertion prevents disconnections.

Visit-First Scan For low-selectivity predicates, visit-first scan can be faster than online blocking because there is no need to block the vectors beforehand. But if the predicate is highly selective, then visit-first scan risks frequent backtracking as the scan struggles to fill the result set. One way to avoid backtracking is to use a traversal mechanism that incorporates attribute information. In [60], the filter condition is added to the best-first search operator. In [133], the distance function used for edge traversal is augmented with an attribute related component so that the scan favors nodes that are likely to pass the filter.[30]

4.2 Plan enumeration

As query plans tend to consist of a small number of operators, in many cases *predefining* the plans is not only feasible but also efficient, as it saves overhead of enumerating the plans online. But for systems that aim to support more complex queries, the plans cannot be predetermined. For extended VDBMSs based on relational systems, relational algebra can be used to express these queries, allowing *automatic* enumeration.

Predefined Some systems target specific workloads, thus focusing on *single plans* per query, while others predefine *multiple plans*. Single plans can be highly efficient as it removes the overhead of plan selection in addition to enumeration, but can be a disadvantage if the predefined plan is not suited to the particular workload. Non-predicated queries trivially have a single query plan when only one search method is available. For example in EuclidesDB [12], each database instance is configured with one search index which is used for every search query. This can also be true for predicated queries. For example in Weaviate [1], all predicated search queries are executed by pre-filtering. But multiple indexes lead to multiple plans. For example, ADBV supports brute force scan and table-based index scan over `PQ` or VGPQ, allowing a query to be executed using either of these methods.

Automatic Extended VDBMSs based on relational systems such as `pgvector` [7] and PASE [136] can take advantage of the underlying relational optimizer to perform plan enumeration as well as selection.

4.3 Plan selection

Existing VDBMSs perform plan selection either by using *handcrafted rules* or by using a *cost model*.

Rule Based If the number of plans is small, then selection rules can be used to decide which plan to execute. Figure 7 shows two examples, used by Qdrant [9] (Fig. 7a) and Yahoo Vespa [17] (Fig. 7b). Both depend on selectivity estimation thresholds.

Cost Based Plan selection can also be performed using a cost model. In ADBV and Milvus, a linear cost model sums the component costs of individual operators to yield the cost of each plan. Operator cost depends on the number of distance calculations as well as memory and disk retrievals. For predicated queries, these numbers are estimated from

[30] See also http://arxiv.org/abs/2203.13601.

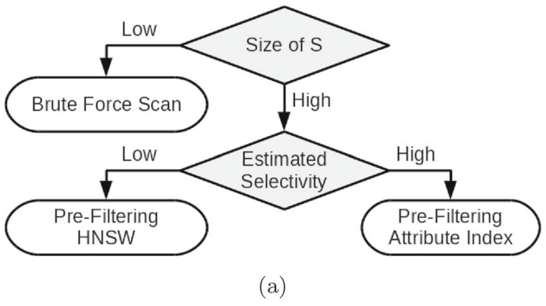

Fig. 7 Plan selection rules in **a** Qdrant and **b** Vespa

the selectivity of the predicate, but they also depend on the desired query accuracy, exposed to the user as an adjustable parameter. The effect of different accuracy levels on operator cost is determined offline.

4.4 Discussion

So far, query optimization in a VDBMS mainly involves predicated queries, and we point out several unaddressed challenges for these queries.

Cost Estimation For pre-filtering, the cost of the scan can be hard to estimate due to uncertainty around the amount of blocking. This applies more to tree or graph-based indexes, as for table-based indexes, the cost of a scan is bounded above by bucket size. Likewise for visit-first scan, the cost of the scan depends on the rate of predicate failures which is hard to know beforehand. On the other hand, post-filtering for a predicated k-NN query may lead to a result set that contains fewer than k items. In VDBMSs that use post-filtering, this is often mitigated by retrieving αk nearest vectors instead of just the k nearest. But higher α make search more expensive, and there is no clear way for deciding on the optimal value which minimizes search cost while guaranteeing k results in the final result set.

Operator Design In addition, designing efficient and effective hybrid operators remains challenging. For graph indexes, block-first scan can lead to disconnected components that either need to be repaired or that require new search algorithms for handling this situation. Existing offline blocking techniques are limited to small number of attribute categories. For visit-first scan, estimating the cost of the scan is challenging due to unpredictable backtracking, complicating plan selection.

5 Query execution

For data manipulation, several techniques exist to compensate for indexes that are hard to update. Additionally, many VDBMSs also take advantage of *distributed architectures* in order to scale to larger datasets or greater workloads. Some that are offered as a cloud service take advantage of disaggregated architectures[31] to offer high elasticity. There are also techniques for taking advantage of *hardware acceleration* to reduce query latency.

5.1 Data manipulation

Some VDBMSs handle insert, update, and delete operations *directly* at query time. For example, the latest implementation of HNSW supports insert and delete operations directly over the graph. Even so, deletes may disconnect the graph, requiring an expensive repair operation to reestablish connectivity. This situation can be avoided by “tombstoning” deleted nodes, in other words marking them non-returnable instead of physically deleting the node from the graph [15, 77].

Other VDBMSs use a combination of *fast and slow* writeable structures to delay the slowdown caused by updating an index. For example, Milvus stores vectors inside an LSM tree [82, 95]. New vectors are added to an in-memory table which gets flushed to disk as a new tree segment when either the size exceeds a threshold or at periodic intervals, and deletions are handled by inserting tombstone markers which are then reconciled when segments are merged. Users can choose to build indexes over the segments to speed up search. As another example, Manu arranges vectors inside a hierarchy of writeable structures. The first level consists of appendable “slices” that are indexed by quickly updatable table-based indexes. New vectors are added into an appropriate slice, and once the number of full slices reaches a threshold, all the slices are merged to form a new segment, indexed by a slow-writeable HNSW. At the same time, to avoid disruptions due to deletions, Manu stores all deletes inside a tombstone bitmap and reconciles them with the indexed structures once a certain number of deletes is reached. As another example, Vald uses the NGT index for searches which does not easily support inserts. To handle inserts, it stores new vectors inside an in-memory queue and then periodically rebuilds the index by

[31] See [123] for details about these architectures.

merging from the queue at fixed time intervals. In ADBV, in-memory HNSW is used as the fast writeable structure while a global disk-resident table-based index stored inside a distributed file system serves as the slow writable store. When the HNSW index on a node becomes too large, it is torn down and assimiliated into the global index.

5.2 Distributed query processing

For distributed VDBMSs, the main considerations are how to partition the vectors across the nodes, maintain data consistency, and support distributed search.

Partitioning If the collection contains structured attributes in addition to vectors, an attribute can be used as the partitioning key. For example, ADBV [130] accepts records that contain structured attributes in addition to vectors, and the default partitioning strategy is to partition along a user-specified attribute. For partitioning along vectors, it supports k-means partitioning. On the other hand, Vald [8] targets cases where the vectors are incrementally added, and it uses most-available memory as the partitioning strategy. Other VDBMSs use consistent or uniform hashing [1, 17, 61].

Consistency A distributed database must necessarily give up strong data consistency to achieve high availability [59]. Some VDBMSs such as Qdrant [9] offer no consistency guarantees at all, leaving transaction management up to the application, while others give up some degree of availability in order to provide strong consistency or strive for eventual consistency.

We give some examples below. In Weaviate, each shard is replicated to provide high availability. When a write occurs, it is processed on the corresponding shard and replicas. Each replica contains a number of searchable structures, including a WAL, an LSM tree for non-vector data, and an HNSW for vector data, and a write is not acknowledged until the write is fully incorporated into the correct structures. A quorum-based mechanism is used to maintain consistency [48]. Vespa offers strong data consistency by conducting searches over the latest-timestamp replica, but this means that it must wait for at least one replica to acknowledge all pending writes. Vearch [15, 77] uses HNSW as its search index and offers strong consistency by implementing a novel locking mechanism. Manu introduces tunable "delta" consistency based on time ticks. Each node consumes fixed-interval time ticks, and a search query can only be executed over a node if the duration between the last consumed time tick and the time of the query is within a user-defined threshold. Setting the threshold to zero leads to strong consistency while other values lead to eventual consistency.

Distributed Search Search queries are typically handled via scatter–gather. The search criteria are scattered across a set of searchable structures, including other shards, replicas, and various per-node structures, before the results are gathered and re-ranked to produce the final result set. For example, Manu and Vald conduct searches over all shards, and local search algorithms search the local structures on each shard. Weaviate conducts searches over replicas under a quorum-based mechanism while Vespa conducts searches over the latest-timestamp replica. In contrast, ADBV supports more targeted searches by allowing vectors to be partitioned along k-means clusters, similar to a nearest-centroid index. In this case, only the shards that are likely to contain relevant vectors need to be searched.

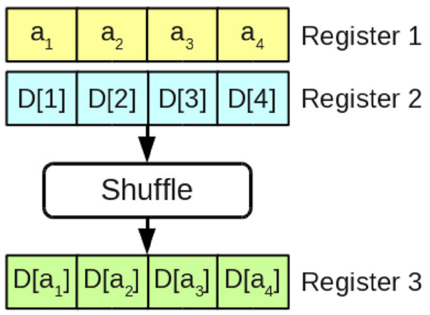

Fig. 8 Performing table look-up with SIMD

5.3 Hardware acceleration

Vector comparison requires reading full vectors into the processor, and this locality makes them amenable to hardware accelerated processing.

CPU Cache If data is not present in CPU cache, it must be retrieved from memory, stalling the processor. Milvus [16, 122] minimizes cache misses for batched queries by partitioning the queries into query blocks which are small enough to fit into CPU cache. The queries are answered a block at a time, and multiple threads can be used to process the queries. As each thread references the entire block when performing a search, the block is safe from eviction.

Single Instruction Multiple Data (SIMD) The original ADC algorithm performs a series of table look-ups and summations (see Example 2). While SIMD instructions can trivially parallelize the summations, look-ups require memory retrievals (in the case of cache misses) and are harder to speed up. In [6, 31, 32], the SIMD shuffle instruction is exploited to parallelize these look-ups within a single SIMD processor. As shown in Fig. 8, the look-up indices plus the entire look-up table are stored into the SIMD registers. The shuffle operator then rearranges the values of the table register so that the ith entry contains the value at the ith index, lining up the values for the subsequent additions.

The table is aggressively compressed in order to fit it into a register. In [32], some improvements are made to allow more values to be stored in the register, namely variable-bit centroids and splitting up large tables across multiple registers.

Graphical Processing Units (GPUs) A GPU consists of a large number of processing units in addition to a large device memory. The threads within a processing unit are grouped together in "warps", and each warp has access to a number of shared 32-bit registers [79]. In [6, 67], an ADC search algorithm for GPUs is given. Similar to the SIMD algorithm, the GPU algorithm likewise avoids memory retrievals, this time from GPU device memory, by performing table lookups within the registers via a shuffle operator called "warp shuffle". If the running k nearest neighbors are tracked in the registers, then k cannot be too large. Milvus works around this issue by conducting a k-NN over multiple rounds, flushing intermediate results store in the registers into host memory after each round.

5.4 Discussion

For distributed VDBMSs, how to effectively partition the vectors remains challenging. In a relational database, tables can be partitioned by attribute key, and in this way not all the partitions need to be searched during a query. But in a VDBMS, aside from the expensive clustering-based partitioning used in ADBV, there is no obvious strategy for quickly partitioning the vectors other than consistent or uniform hashing.

Designing GPU-accelerated indexes and algorithms is also an ongoing effort. For example, tree and graph-based indexes are less amenable to GPU-accelerated search due to issues such as task dependencies, non-contiguous memory allocations, and task diversity. To address these issues, [144] proposes a top-down algorithm for constructing a tree-based index that relies on global sorts in order to parallelize node partitioning at each level.

6 Current systems

The variety of data management techniques has led to an equally diverse landscape of commercial VDBMSs. In this section, we broadly categorize these systems into *native* systems, which are designed specifically for vector data management, and *extended* systems, which add vector capabilities on top of an existing system. Table 6 lists several systems. Following these discussions, we close by summarizing the existing systems and then give recommendations for users.

6.1 Native

Native systems aim to be highly specialized at providing high-performance vector data management. This is achieved by limiting the query capabilities, allowing them to simplify the components such as removing the query parser, using simple storage models such as single-table collections, and using a simple query optimizer over a small number of predefined plans, or in some cases offering no query optimization. We divide these systems into two subcategories, those that target *mostly vector* workloads [2, 8, 10, 12, 15], where the vast majority of queries access the vector collection, and those that target *mostly mixed* workloads [1, 9, 16, 18, 19, 61], where queries are also expected to access non-vector collections. Mostly mixed workloads may consist of traditional attribute queries or textual keyword queries along with predicated and non-predicated vector queries.

Mostly Vector Some mostly-vector systems focus exclusively on non-predicated queries while others offer predicated queries but supported by a single predefined plan. Mostly-vector systems often support only a single search index, typically graph-based, and hence have no need for a query parser, rewriter, or optimizer, omitting these components in order to reduce processing overhead. They also typically do not support exact search.

We give some examples. *Vald* [8] is a VDBMS aimed at providing scalable non-predicated vector search. Vald spans a Kubernetes[32] cluster consisting of multiple "agents". To increase the performance, the vector collection is sharded and replicated across the agents and search is conducted via scatter–gather. An individual Vald agent contains an NGT graph index built over its local shard. *Vearch* [15, 77] is targeted at high-performance image-based search for e-commerce. As the search does not need to be exact, Vearch only supports approximate search, and predicated queries are all executed using fast but potentially inaccurate post-filtering instead of going through a query optimizer.[33] Vearch adopts a disaggregated architecture with dedicated search nodes, allowing it to scale read and write capabilities independently. *Pinecone* [2] offers a scalable distributed system similar to Vald, but as it is offered as a managed cloud-based service, it can be a more user-friendly option. As shown in Fig. 9 *EuclidesDB* [12] is a system for managing embedding models, aimed at allowing users to experiment over various models and similarity scores. Users bring their own embedding models and interact via indirect manipulation, abstracting the underlying models. *Chroma* [10] is a centralized system similar to EuclidesDB. *Vexless* is an academic prototype that gives several techniques for addressing the limitations of a lambda-based serverless cloud architecture, namely low memory budgets, high communication overhead, and the problem of cold starts [110].

Mostly Mixed Mostly-mixed systems aim to support more sophisticated queries compared to mostly-vector systems. In general, they support a greater variety of basic queries,

[32] http://kubernetes.io.

[33] The latest version uses pre-filtering instead of post-filtering as this can be more efficient for strong filters.

Table 6 A list of current VDBMSs

Name	Type	Sub-type	Vector query			Query variant			Vector index			
			Ex	Ap	Rng	Pr	Mul	Bat	Tab	Tr	Gr	Opt
EuclidesDB (2018) [12]	Nat	Vec	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗
Vearch (2018) [15, 77]	Nat	Vec	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗
Pinecone (2019) [2]	Nat	Vec	✗	✓	✗	✓	✓	✗	Proprietary			U
Vald (2020) [8]	Nat	Vec	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗
Chroma (2022) [10]	Nat	Vec	✗	✓	✗	✓	✗	✗	✗	✗	✓	U
Weaviate (2019) [1]	Nat	Mix	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗
Milvus (2021) [16, 122]	Nat	Mix	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
NucliaDB (2021) [18]	Nat	Mix	✗	✓	✓	✓	U	✗	✗	✗	✓	✗
Qdrant (2021) [9]	Nat	Mix	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
Manu (2022) [61]	Nat	Mix	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Marqo (2022) [19]	Nat	Mix	✗	✓	✗	✓	✓	✓	✗	✗	✓	U
Vespa (2020) [17]	Ext	NoSQL	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓
Cosmos DB (2023)	Ext	NoSQL	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗
MongoDB (2023)	Ext	NoSQL	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗
Neo4j (2023)	Ext	NoSQL	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
Redis (2023)	Ext	NoSQL	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗
AnalyticDB-V (2020) [130]	Ext	Rel	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓
PASE+PG (2020) [136]	Ext	Rel	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓
pgvector+PG (2021) [7]	Ext	Rel	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓
SingleStoreDB (2022) [11, 99]	Ext	Rel	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓
ClickHouse (2023) [20]	Ext	Rel	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓
MyScale (2023) [21]	Ext	Rel	✓	✓	✓	✓	✗	✗	✓	✗	P	✓

Dates are estimated from paper publication, earliest Github release, blog post, or other indications. Dates indicate year when vector search capability first appeared in the product. Abbreviations: *Ex.* exact k-NN, *Ap.* ANN, *Rng.* range, *Pr.* predicated, *Mul.* multi-vector, *Bat.* batched, *Tab.* table, *Tr.* tree, *Gr.* graph, *Opt.* query optimizer, *Nat.* native, *Ext.* extended, *Vec.* mostly vector, *Rel.* relational, *U* unknown, *P* proprietary

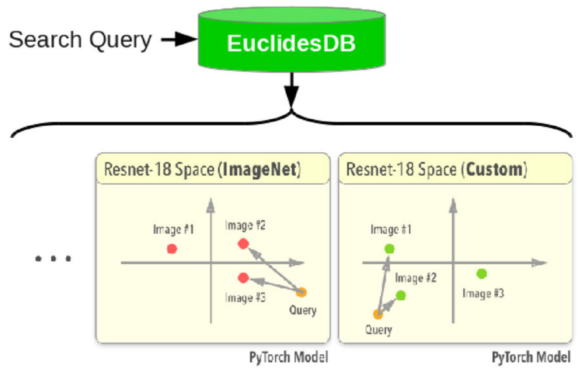

Fig. 9 EuclidesDB [12] supports querying multiple vector spaces at a time to assess the impact of different similarity scores and embedding models

including more support for exact and range queries, in addition to more query variants. These systems likewise have no need for query rewriters or parsers, but some make use of an optimizer. Some systems also have more sophisticated data and storage models in order to deal with predicated queries and attribute-only queries that go beyond simple retrieval.

In particular, *Milvus* [16, 122] is aimed at comprehensive support for vector search queries, and *Manu* [61] adds additional features on top of Milvus. All three basic query types are supported, in addition to the three query variants. To support these additional capabilities and support different workload characteristics, Milvus and Manu support different types of search indexes and handle predicated queries via a cost-based optimizer. *Qdrant* [9] likewise supports a large variety of search queries. For predicated queries, it uses a rule-based optimizer coupled and introduces a proprietary HNSW index in order to support efficient block-first scan. *NucliaDB* [18] and *Marqo* [19] are targeted at document search and use vector search to provide semantic retrieval. A key feature is the support for combining keywords and vectors through multi-vector search. Non-vector keyword queries are processed using text specific techniques, resulting in sparse term-frequency vectors which are then combined with dense feature vectors through an aggregate score in order to conduct multi-vector search. Finally, *Weaviate* targets document search over a graph model. This allows Weaviate to answer non-vector queries, such as retrieving all books written by a certain author, in addition to similarity queries via vector search. Weaviate adopts leaderless replicas to scale out reads and uses a quorum-based mechanism to support eventually consistent writes.

6.2 Extended

Extended systems inherit all the capabilities of an underlying data management system and are necessarily more complex compared to native systems. But these systems are also more capable. Nearly all the extended systems listed in Table 6 support all three basic query types and multiple indexes, and all of them support query optimization. These systems likewise divide into two subcategories, those where the underlying system is NoSQL and those where it is relational. The main challenge for these systems is how to integrate vector search into the system while still offering high performance.

NoSQL Many traits of a NoSQL system [49], such as schemaless storage, distributed architecture, and eventual consistency, are present in native systems, making a NoSQL extended VDBMS much like a native system.

For example, *Vespa* [17] is a scalable distributed NoSQL system designed for large-scale data processing workloads. Vespa aims at general data processing tasks, and it uses a flexible SQL-like query language. But like native systems, the storage model is simple, provided by a document store. The vector search extension adds a custom HNSW and a rule-based optimizer for predicated vector search queries. *Cassandra* [72] is a popular distributed NoSQL system based on a wide column store. Vector search capability will be available in version 5.0[34] by integrating HNSW into the storage layer, implementing scatter–gather across replicas, and extending the Cassandra query language with vector search operators. The Spark-based *Databricks*[35] platform is expected to support vector search, including predicated queries, in an upcoming version.

Aside from Vespa, several other document based NoSQL databases have now been extended to support vector search, including *MongoDB*,[36] *Cosmos DB*,[37] and *Redis*,[38] when coupled with the Redis Stack search extension. Vector search capability has also been extended to NoSQL systems other than document stores. For example, *Neo4j*[39] is a property graph database with experimental vector search capability. So far, only ANN queries are supported, using HNSW.

[34] See Cassandra Enhancement Proposal 30 (CEP-30).

[35] http://databricks.com.

[36] http://mongodb.com.

[37] http://cosmos.azure.com.

[38] http://redis.io.

[39] http://neo4j.com.

Relational For extended relational systems, their features mostly come from the inherent capabilities of relational systems. For example, SQL already is sufficient for expressing (c, k)-search queries, and these queries can already be answered by most relational engines upon adding a user-defined similarity function. Subsequently, extended relational systems focus more on tightly integrating vector search capability alongside existing components.

The approach taken by *SingleStore* [11, 99] is to offer only exact k-NN and range search through its native relational engine, without any vector search indexes, and to rely on its scalability via column store replicas and a distributed row store to support fast reads and writes. Vector search is handled by the native relational engine, extended with functions for calculating dot product and Euclidean distance.

On the other hand, *PASE* [136] and **`pgvector`** [7] take advantage of the extensibility of PostgreSQL to provide vector capabilities. PASE extends PostgreSQL with a flat quantization index and HNSW index in order to support vector search, and `pgvector` brings a vector data type, functions over this data type, and flat and HNSW index access methods into PostgreSQL. For both, vector queries are issued using SQL. If an index is created over a vector column, then queries are answered using the index, yielding approximate results. Otherwise, exact brute force scan is used. Plan selection is performed by the existing PostgreSQL query optimizer using the generic cost estimator, or by calling index-specific cost estimators to refine the estimate, if needed. Other features such as replication, fault tolerance, access controls, and concurrency control, are provided by PostgreSQL.

There are also VDBMSs built over other relational databases. *AnalyticDB-V* [130] adds vector search capability on top of AnalyticDB [138]. This is achieved primarily by introducing vector indexes, VGPQ and HNSW, and by augmenting the cost-based optimizer. *ClickHouse* [20] is a columnar database aimed at fast analytics coupled with an asynchronous merge mechanism for fast ingestion. It supports vector queries using ANNOY and HNSW. Likewise, *MyScale* [21] is a cloud service using ClickHouse as the backend. It adds table-based search indexes including flat indexes and IVFADC, and a proprietary search index called "multi-scale tree graph" (MSTG) which is shown to outperform both IVFADC and HNSW.

6.3 Libraries and other systems

Vector search engines and libraries are typically embedded into applications that require vector search, but they lack the capabilities of a full VDBMS.

Search Engines Apache Lucene [4] is a pluggable search engine for embedded applications. Latest versions offer vector search, supported by HNSW. While Lucene itself lacks features such as multi-tenancy, distributed search, and admin-

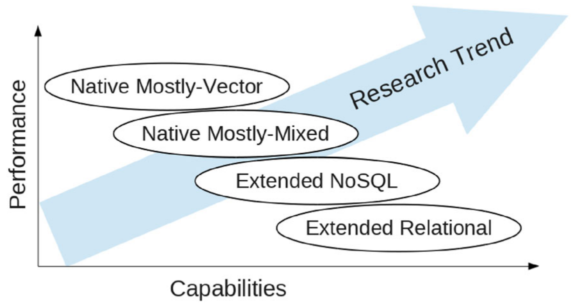

Fig. 10 High-level characteristics of VDBMSs

istrative features, many of these are provided by search platforms built on top of Lucene, including Elasticsearch [5], OpenSearch [22], and Solr [23]. These capabilities could make Lucene an attractive alternative to mostly-vector native VDBMSs.[40]

Libraries There are also libraries that implement specific indexes. For example, KGraph is an implementation of NN-Descent. Microsoft Space Partition Tree and Graph (SPTAG) [24] combines several techniques, including SPANN and NGT, into one index. Libraries are also available for LSH, including E2LSH[41] and FALCONN.[42] Likewise, Meta Faiss [6] offers a selection of indexes, including HNSW, an LSH family for Hamming distance, and quantization-based indexes.

Other Systems Other systems aim at other parts of the broader pipeline. Similar to relational ETL tools, Featureform[43] organizes the workflows that transform raw data into curated datasets used by downstream applications. Featureform exposes a vector search endpoint that executes a k-NN query over a configured provider, such as Pinecone. On the other hand, Activeloop[44] Deep Lake [63] offers tensor operations directly over a tensor warehouse, making it capable of performing vector search inside the warehouse.

6.4 Discussion

The designs of these databases cover a spectrum of characteristics involving query processing and vector storage, manifesting in a range of performance and capabilities, as shown[45] in Fig. 10. Based on this observation, we imagine that future work will target systems that can offer both high performance in addition to offering unified data management capabilities, represented by the arrow. These improvements may come from new features aimed at supporting new applications in addition to new techniques, such as more efficient disk-resident indexes, as discussed in Sect. 8.

For existing systems, we offer a few remarks. Native mostly-vector systems broadly offer high performance but are targeted at specific workloads, sometimes even specific queries, and thus have relatively limited capability. Meanwhile, native mostly-mixed systems offer more capabilities, notably predicated queries, and some such as Milvus [16, 122], Qdrant [9], and Manu [61] also perform query optimization. These, along with extended NoSQL systems, achieve a comfortable balance between high performance and search capabilities. On the other hand, extended relational systems offer the most capabilities but possibly less performance. But, as has been mentioned elsewhere,[46] relational systems are already major components of industrial data infrastructures, and being able to conduct vector search without introducing new systems into the infrastructure is a compelling advantage.

7 Benchmarks

Comprehensive cross-disciplinary comparisons of vector search algorithms and systems are surprisingly scarce, attributed to the wide range of fields from which search algorithms arise [78]. We point out two notable attempts at benchmarking these algorithms and systems.

In [78], a large number of ANN algorithms are uniformly implemented and evaluated across a range of experimental conditions. These algorithms include LSH, L2H, methods based on quantization, tree-based techniques, and graph-based techniques. The experiments are conducted over 18 datasets, ranging from a few thousand vectors to 10 million vectors, and with dimensions ranging from 100 to 4,096. The feature vectors are derived from real-world image, text, video, and audio collections, as well as synthetically generated. Algorithms are measured on query latency as well as the quality of the result sets based on precision, recall, and two other derivative measures.

In [34], the evaluation is extended to include full VDBMSs in addition to isolated algorithms. Whereas [78] aims to avoid the effects of different implementations, here these differences are kept in order to more accurately reflect real-world conditions. The standardized evaluation platform and the latest benchmark results are available online,[47] with results for several VDBMSs.

[40] For a discussion, see http://arxiv.org/abs/2308.14963.

[41] http://www.mit.edu/~andoni/E2LSH_gpl.tar.gz.

[42] http://github.com/falconn-lib/falconn.

[43] http://featureform.com.

[44] http://activeloop.ai.

[45] The ranking is consistent with empirical observations [34].

[46] https://arxiv.org/abs/2308.14963.

[47] http://ann-benchmarks.com.

8 Challenges and open problems

While much progress has been made on vector data management, some challenges remain unaddressed. First, we note that *score selection* and *score design* remain challenging. Second, *index design* also remains challenging, particularly around disk-based indexes, efficient updates, and concurrency. We note an encouraging recent framework for designing disk-resident graph-based indexes based on locality-preserving blocks of sub-graphs [125]. Third, *operator design* in terms of predicated queries remains an area where improvements may be possible. Fourth, designing efficient *distributed systems* remains challenging due to the difficulty of effectively partitioning vectors.

Moreover, there remain a number of new applications that have yet to be extensively studied. For example, e-commerce and recommender platforms make use of *incremental* k*-NN search*, where k is effectively very large but is retrieved in small increments so that the results appear to be seamlessly delivered to the user. So far, it is unclear how to support this search inside vector indexes. As another example, *multi-vector search* is important for applications such as face recognition. Existing techniques tend to use aggregate scores, but this can be inefficient as it multiplies the amount of distance calculations. Meanwhile, generic multi-attribute top-k techniques are hard to adapt to vector indexes [122]. Finally, as vector search becomes increasingly mission-critical, *data security and user privacy* become more important, especially for VDBMSs that offer managed cloud services. There is thus a need for new techniques that can support private and secure high-dimensional vector search [134], including federated search [141].

9 Conclusion

In this paper, we surveyed vector database management systems aimed at fast and accurate vector search, developed in response to recent popularity of dense retrieval for applications such as LLMs and e-commerce. We reviewed considerations for query processing, including similarity scores, query types, and basic operators. We also reviewed the design, search, and maintenance considerations regarding vector search indexes. We described several techniques for query optimization and execution, including plan enumeration, plan selection, operators for predicated or "hybrid" queries, and hardware acceleration. Finally, we discussed several commercial systems and the main benchmarks for supporting experimental comparisons.

Acknowledgements This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (61925205, 62232009, 62102215), Zhongguancun Lab, Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

References

1. http://weaviate.io
2. http://pinecone.io
3. http://github.com/spotify/annoy
4. http://lucene.apache.org
5. http://elastic.co
6. http://github.com/facebookresearch/faiss
7. http://github.com/pgvector
8. http://vald.vdaas.org
9. http://qdrant.tech
10. http://trychroma.com
11. http://singlestore.com
12. http://euclidesdb.readthedocs.io
13. http://github.com/flann-lib/flann
14. http://github.com/yahoojapan/NGT
15. http://github.com/vearch
16. http://milvus.io
17. http://vespa.ai
18. http://nuclia.com
19. http://marqo.ai
20. http://clickhouse.com
21. http://myscale.com
22. http://opensearch.org
23. http://solr.apache.org
24. http://github.com/microsoft/SPTAG
25. Abdelkader, A., Arya, S., da Fonseca, G.D., Mount, D.M.: Approximate nearest neighbor searching with non-Euclidean and weighted distances. In: SODA, pp. 355–372 (2019)
26. Aggarwal, C.C., Hinneburg, A., Keim, D.A.: On the surprising behavior of distance metrics in high dimensional space. In: ICDT (2001)
27. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM **51**(1), 117–122 (2008)
28. Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., Schmidt, L.: Practical and optimal LSH for angular distance. In: NeurIPS, pp. 1225–1233 (2015)
29. Andoni, A., Indyk, P., Razenshteyn, I.: Approximate nearest neighbor search in high dimensions. In: ICM, pp. 3287–3318 (2018)
30. Andoni, A., Razenshteyn, I.: Optimal data-dependent hashing for approximate near neighbors. In: STOC, pp. 793–801 (2015)
31. André, F., Kermarrec, A.M., Le Scouarnec, N.: Accelerated nearest neighbor search with Quick ADC. In: ICMR (2017)
32. André, F., Kermarrec, A.M., Le Scouarnec, N.: Quicker ADC: unlocking the hidden potential of product quantization with SIMD. IEEE Trans. Pattern Anal. Mach. Intell. **43**(5), 1666–1677 (2021)
33. Asai, A., Min, S., Zhong, Z., Chen, D.: Retrieval-based language models and applications. In: ACL (2023)
34. Aumüller, M., Bernhardsson, E., Faithfull, A.: ANN-benchmarks: a benchmarking tool for approximate nearest neighbor algorithms. Inform. Syst. **87**, 101374 (2020)
35. Azizi, I., Echihabi, K., Palpanas, T.: ELPIS: graph-based similarity search for scalable data science. Proc. VLDB Endow. **16**(6), 1548–1559 (2023)
36. Bang, F.: GPTCache: an open-source semantic cache for LLM applications enabling faster answers and cost savings. In: NLP-OSS, pp. 212–218 (2023)

37. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
38. Berg, M., Cheong, O., Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer-Verlag, Berlin (2008)
39. Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: ICDT (1999)
40. Chang, W.C., Yu, F.X., Chang, Y.W., Yang, Y., Kumar, S.: Pre-training tasks for embedding-based large-scale retrieval. In: ICLR (2020)
41. Chen, H., Ryu, J., Vinyard, M.E., Lerer, A., Pinello, L.: SIMBA: single-cell embedding along with features. Nat. Methods **21**, 1003–1013 (2024)
42. Chen, L., Gao, Y., Song, X., Li, Z., Zhu, Y., Miao, X., Jensen, C.S.: Indexing metric spaces for exact similarity search. ACM Comput. Surv. **55**(6), 1–39 (2022)
43. Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M., Wang, J., Yang, M., Wang, J.: SPANN: highly-efficient billion-scale approximate nearest neighbor search. In: NeurIPS (2021)
44. Ciaccia, P., Patella, M., Zezula, P.: M-Tree: an efficient access method for similarity search in metric spaces. In: Proc. VLDB Endow., pp. 426–435 (1997)
45. Dasgupta, S., Freund, Y.: Random projection trees and low dimensional manifolds. In: STOC, pp. 537–546 (2008)
46. Dasgupta, S., Sinha, K.: Randomized partition trees for exact nearest neighbor search. In: COLT, pp. 317–337 (2013)
47. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SCG, pp. 253–262 (2004)
48. Davidson, S.B., Garcia-Molina, H., Skeen, D.: Consistency in a partitioned network: a survey. ACM Comput. Surv. **17**(3), 341–370 (1985)
49. Davoudian, A., Chen, L., Liu, M.: A survey on NoSQL stores. ACM Comput. Surv. **51**(2), 1–43 (2018)
50. Dearholt, D., Gonzales, N., Kurup, G.: Monotonic search networks for computer vision databases. In: ACSSC, pp. 548–553 (1988)
51. Dong, W., Charikar, M., Li, K.: Efficient *k*-nearest neighbor graph construction for generic similarity measures. In: WWW (2011)
52. Echihabi, K., Zoumpatianos, K., Palpanas, T.: New trends in high-D vector similarity search: AI-driven, progressive, and distributed. Proc. VLDB Endow. **14**(12), 3198–3201 (2021)
53. Echihabi, K., Zoumpatianos, K., Palpanas, T., Benbrahim, H.: Return of the Lernaean Hydra: experimental evaluation of data series approximate similarity search. Proc. VLDB Endow. **13**(3), 403–420 (2019)
54. Edelsbrunner, H., Shah, N.R.: Incremental topological flipping works for regular triangulations. Algorithmica **15**, 223–241 (1996)
55. Eppstein, D., Paterson, M.S., Yao, F.F.: On nearest-neighbor graphs. Discrete Comput. Geom. **17**, 263–282 (1997)
56. Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. Proc. VLDB Endow. **12**(5), 461–474 (2019)
57. Gao, J., Long, C.: RaBitQ: quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. Proc. ACM Manag. Data **2**(3), 1–27 (2024)
58. Ge, T., He, K., Ke, Q., Sun, J.: Optimized product quantization for approximate nearest neighbor search. In: CVPR, pp. 2946–2953 (2013)
59. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002)
60. Gollapudi, S., Karia, N., Sivashankar, V., Krishnaswamy, R., Begwani, N., Raz, S., Lin, Y., Zhang, Y., Mahapatro, N., Srinivasan, P., Singh, A., Simhadri, H.V.: Filtered-DiskANN: graph algorithms for approximate nearest neighbor search with filters. In: WWW (2023)
61. Guo, R., Luan, X., Xiang, L., Yan, X., Yi, X., Luo, J., Cheng, Q., Xu, W., Luo, J., Liu, F., Cao, Z., Qiao, Y., Wang, T., Tang, B., Xie, C.: Manu: a cloud native vector database management system. Proc. VLDB Endow. **15**(12), 3548–3561 (2022)
62. Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., Kumar, S.: Accelerating large-scale inference with anisotropic vector quantization. In: ICML (2020)
63. Hambardzumyan, S., Tuli, A., Ghukasyan, L., Rahman, F., Topchyan, H., Isayan, D., McQuade, M., Harutyunyan, M., Hakobyan, T., Stranic, I., Buniatyan, D.: Deep Lake: a lakehouse for deep learning. In: CIDR (2023)
64. Harwood, B., Drummond, T.: FANNG: fast approximate nearest neighbour graphs. In: CVPR (2016)
65. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
66. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. **33**(1), 117–128 (2011)
67. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. IEEE Trans. Big Data **7**(3), 535–547 (2021)
68. Jurafsky, D., Martin, J.H.: Speech and Language Processing, 2nd edn. Prentice-Hall, Hoboken (2009)
69. Keivani, O., Sinha, K., Ram, P.: Improved maximum inner product search with better theoretical guarantee using randomized partition trees. Mach. Learn. **107**, 1069–1094 (2018)
70. Kim, Y.: Applications and future of dense retrieval in industry. In: SIGIR, pp. 3373–3374 (2022)
71. Kleinberg, J.M.: Navigation in a small world. Nature **406**, 845 (2000)
72. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
73. Lee, D., Wong, C.: Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta Inform. **9**, 23–29 (1977)
74. Leskovec, J., Rajaraman, A., Ullman, J.: Mining of Massive Datasets, 3rd edn. Cambridge University Press, Cambridge (2014)
75. Li, F.: Modernization of databases in the cloud era: building databases that run like Legos. Proc. VLDB Endow. **16**(12), 4140–4151 (2023)
76. Li, H., Ai, Q., Zhan, J., Mao, J., Liu, Y., Liu, Z., Cao, Z.: Constructing tree-based index for efficient and effective dense retrieval. In: SIGIR (2023)
77. Li, J., Liu, H., Gui, C., Chen, J., Ni, Z., Wang, N., Chen, Y.: The design and implementation of a real time visual search system on JD e-commerce platform. In: Middleware, pp. 9–16 (2018)
78. Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X.: Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. IEEE Trans. Knowl. Data Eng. **32**(8), 1475–1488 (2020)
79. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro **28**(2), 39–55 (2008)
80. Lipton, R.J., Tarjan, R.E.: Applications of a planar separator theorem. SIAM J. Comput. **9**(3), 615–627 (1980)
81. Liu, T., Moore, A.W., Gray, A., Yang, K.: An investigation of practical approximate nearest neighbor algorithms. In: NeurIPS, pp. 825–832 (2004)
82. Luo, C., Carey, M.J.: LSM-Based storage techniques: a survey. VLDB J. **29**(1), 393–418 (2019)
83. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi- probe LSH: efficient indexing for high-dimensional similarity search. In: Proc. VLDB Endow. pp. 950–961 (2007)

84. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Inform. Syst. **45**, 61–68 (2014)
85. Malkov, Y., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. Pattern Anal. Mach. Intell. **42**(4), 824–836 (2020)
86. Matsui, Y., Uchida, Y., Jégou, H., Satoh, S.: A survey of product quantization. ITE Trans. Media Technol. Appl. **6**(1), 2–10 (2018)
87. Meiser, S.: Point location in arrangements of hyperplanes. Inform. Comput. **106**(2), 286–303 (1993)
88. Meng, J., Wang, H., Xu, J., Ogihara, M.: ONe index for all kernels (ONIAK): a zero re-indexing LSH solution to ANNS-ALT (After Linear Transformation). Proc. VLDB Endow. **15**(13), 3937–3949 (2022)
89. Mirkes, E.M., Allohibi, J., Gorban, A.: Fractional norms and quasinorms do not help to overcome the curse of dimensionality. Entropy **22**(10), 1105 (2020)
90. Mitra, B., Craswell, N.: An introduction to neural information retrieval. Found. Trends Inf. Retr. **13**(1), 1–126 (2018)
91. Moll, O., Favela, M., Madden, S., Gadepally, V., Cafarella, M.: SeeSaw: interactive ad-hoc search over image databases. Proc. ACM Manag. Data **1**(4), 1–26 (2023)
92. Muja, M., Lowe., D.G.: FLANN: fast library for approximate nearest neighbors. In: VISAPP (2009)
93. Navarro, G.: Searching in metric spaces by spatial approximation. VLDB J. **11**(1), 28–46 (2002)
94. Norouzi, M., Fleet, D.J.: Cartesian *k*-means. In: CVPR (2013)
95. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Inform. **33**, 351–385 (1996)
96. Paredes, R., Chávez, E.: Using the *k*-nearest neighbor graph for proximity searching in metric spaces. In: SPIRE, pp. 127–138 (2005)
97. Paredes, R., Chávez, E., Figueroa, K., Navarro, G.: Practical construction of *k*-nearest neighbor graphs in metric spaces. In: WEA (2006)
98. Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M.P., Shyu, M.L., Chen, S.C., Iyengar, S.S.: A survey on deep learning: algorithms, techniques, and applications. ACM Comput. Surv. **51**(5), 1–36 (2018)
99. Prout, A., Wang, S.P., Victor, J., Sun, Z., Li, Y., Chen, J., Bergeron, E., Hanson, E., Walzer, R., Gomes, R., Shamgunov, N.: Cloud-native transactions and analytics in SingleStore. In: SIGMOD, pp. 2340–2352 (2022)
100. Qin, J., Wang, W., Xiao, C., Zhang, Y.: Similarity query processing for high-dimensional data. Proc. VLDB Endow. **13**(12), 3437–3440 (2020)
101. Qin, J., Wang, W., Xiao, C., Zhang, Y., Wang, Y.: High-dimensional similarity query processing for data science. In: KDD, pp. 4062–4063 (2021)
102. Ram, P., Sinha, K.: Revisiting *kd*-tree for nearest neighbor search. In: KDD, pp. 1378–1388 (2019)
103. Rigaux, P., Scholl, M., Voisard, A.: Spatial Databases: With Application to GIS. Morgan Kaufmann Publishers Inc., Burlington (2001)
104. Rubinstein, A.: Hardness of approximate nearest neighbor search. In: STOC, pp. 1260–1268 (2018)
105. Salakhutdinov, R.R., Hinton, G.E.: Learning a nonlinear embedding by preserving class neighbourhood structure. In: AISTATS (2007)
106. Sellis, T., Roussopoulos, N., Faloutsos, C.: Multidimensional access methods: trees have grown everywhere. Proc. VLDB Endow., pp. 13–14 (1997)
107. Silpa-Anan, C., Hartley, R.: Optimised KD-trees for fast image descriptor matching. In: CVPR (2008)
108. Sivic, Z.: Video Google: a text retrieval approach to object matching in videos. In: ICCV, pp. 1470–1477 (2003)
109. Su, T.H., Chang, R.C.: On constructing the relative neighborhood graphs in Euclidean *k*-dimensional spaces. Computing **46**, 121–130 (1991)
110. Su, Y., Sun, Y., Zhang, M., Wang, J.: Vexless: a serverless vector data management system using cloud functions. Proc. ACM Manag. Data **2**(3), 1–26 (2024)
111. Subramanya, S.J., Devvrit, Kadekodi, R., Krishnaswamy, R., Simhadri, H.: DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In: NeurIPS (2019)
112. Tagliabue, J., Greco, C.: (Vector) Space is not the final frontier: product search as program synthesis. In: SIGIR (2023)
113. Taipalus, T.: Vector database management systems: fundamental concepts, use-cases, and current challenges. Cognitive Syst. Res. **85**, 101216 (2024)
114. Teflioudi, C., Gemulla, R.: Exact and approximate maximum inner product search with LEMP. ACM Trans. Database Syst. **42**(1), 1–49 (2016)
115. Toussaint, G.T.: The relative neighbourhood graph of a finite planar set. Pattern Recognit. **12**(4), 261–268 (1980)
116. Vaidya, P.M.: An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. Discrete Comput. Geom. **4**, 101–115 (1989)
117. Vempala, S.S.: Randomly-oriented *k*-*d* trees adapt to intrinsic dimension. In: LIPIcs (2012)
118. Wang, F., Sun, J.: Survey on distance metric learning and dimensionality reduction in data mining. Data Min. Knowl. Disc. **29**, 534–564 (2015)
119. Wang, J., Li, S.: Query-driven iterated neighborhood graph search for large scale indexing. In: MM, pp. 179–188 (2012)
120. Wang, J., Wang, J., Zeng, G., Tu, Z., Gan, R., Li, S.: Scalable *k*-NN graph construction for visual descriptors. In: CVPR, pp. 1106–1113 (2012)
121. Wang, J., Wang, N., Jia, Y., Li, J., Zeng, G., Zha, H., Hua, X.S.: Trinary-projection trees for approximate nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. **36**(2), 388–403 (2014)
122. Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., Xie, C.: Milvus: A purpose-built vector data management system. In: SIGMOD, pp. 2614–2627 (2021)
123. Wang, J., Zhang, Q.: Disaggregated database systems. In: SIGMOD, pp. 37–44 (2023)
124. Wang, J., Zhang, T., Song, J., Sebe, N., Shen, H.T.: A survey on learning to hash. IEEE Trans. Pattern Anal. Mach. Intell. **40**(4), 769–790 (2018)
125. Wang, M., Xu, W., Yi, X., Wu, S., Peng, Z., Ke, X., Gao, Y., Xu, X., Guo, R., Xie, C.: Starling: an I/O-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. Proc. ACM Manag. Data **2**(1), 1–27 (2024)
126. Wang, M., Xu, X., Yue, Q., Wang, Y.: A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. Proc. VLDB Endow. **14**(11), 1964–1978 (2021)
127. Wang, R., Deng, D.: DeltaPQ: lossless product quantization code compression for high dimensional similarity search. Proc. VLDB Endow. **13**(13), 3603–3616 (2020)
128. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**, 440–442 (1998)
129. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. Proc. VLDB Endow. pp. 194–205 (1998)
130. Wei, C., Wu, B., Wang, S., Lou, R., Zhan, C., Li, F., Cai, Y.: AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. Proc. VLDB Endow. **13**(12), 3152–3165 (2020)

131. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: NeurIPS, pp. 1753–1760 (2008)
132. Williams, R.: On the difference between closest, furthest, and orthogonal pairs: Nearly-linear vs barely-subquadratic complexity. In: SODA, pp. 1207–1215 (2018)
133. Wu, W., He, J., Qiao, Y., Fu, G., Liu, L., Yu, J.: HQANN: Efficient and robust similarity search for hybrid queries with structured and unstructured constraints. In: CIKM (2022)
134. Xue, W., Li, H., Peng, Y., Cui, J., Shi, Y.: Secure *k* nearest neighbors query for high-dimensional vectors in outsourced environments. IEEE Trans. Big Data **4**(4), 586–599 (2018)
135. Yandex, A.B., Lempitsky, V.: Efficient indexing of billion-scale datasets of deep descriptors. In: CVPR, pp. 2055–2063 (2016)
136. Yang, W., Li, T., Fang, G., Wei, H.: PASE: PostgreSQL ultra-high-dimensional approximate nearest neighbor search extension. In: SIGMOD, pp. 2241–2253 (2020)
137. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA, pp. 311–321 (1993)
138. Zhan, C., Su, M., Wei, C., Peng, X., Lin, L., Wang, S., Chen, Z., Li, F., Pan, Y., Zheng, F., Chai, C.: AnalyticDB: real-time OLAP database system at Alibaba Cloud. Proc. VLDB Endow. **12**(12), 2059–2070 (2019)
139. Zhang, H., Cao, L., Yan, Y., Madden, S., Rundensteiner, E.A.: Continuously adaptive similarity search. In: SIGMOD, pp. 2601–2616 (2020)
140. Zhang, W., Ji, J., Zhu, J., Li, J., Xu, H., Zhang, B.: BitHash: an efficient bitwise locality sensitive hashing method with applications. Knowl. Based Syst. **97**, 40–47 (2016)
141. Zhang, X., Wang, Q., Xu, C., Peng, Y., Xu, J.: FedKNN: secure federated k-nearest neighbor search. Proc. ACM Manag. Data **2**(1), 1–26 (2024)
142. Zhao, W.L., Wang, H., Ngo, C.W.: Approximate k-NN graph construction: a generic online approach. IEEE Trans. Multimed. **24**, 1909–1921 (2022)
143. Zhu, Y., Chen, L., Gao, Y., Jensen, C.S.: Pivot selection algorithms in metric spaces: a survey and experimental study. VLDB J. **31**(1), 23–47 (2022)
144. Zhu, Y., Ma, R., Zheng, B., Ke, X., Chen, L., Gao, Y.: GTS: GPU-based tree index for fast similarity search. Proc. ACM Manag. Data **2**(3), 1–27 (2024)
145. Zuo, C., Qiao, M., Zhou, W., Li, F., Deng, D.: SeRF: segment graph for range-filtering approximate nearest neighbor search. Proc. ACM Manag. Data **2**(1), 1–26 (2024)