Vector Database: Systems, Algorithms and Future

1st Mocheng Li
School of Data Science
Chinese University of Hongkong, Shenzhen
Shenzhen, China
224040369@link.cuhk.edu.cn

***Abstract*—Vector databases serve as the fundamental technology for similarity search, enabling efficient retrieval of high-dimensional data. They are widely utilized in applications such as recommendation systems, search engines, and Retrieval-Augmented Generation (RAG) in AI.**

This paper provides an overview of three popular vector databases, Faiss, ADBV and Milvus, which are designed to efficiently handle high-dimensional vector search tasks. Faiss is an open-source library that focuses on efficient similarity search and vector indexing. ADBV enhances the efficiency of attribute filtering and vector search by introducing a cost-based approach to select the best strategy for different query scenarios, while Milvus, an open-source project, supports distributed, high-performance retrieval in various environments. The paper will delve into their architectures, key features, and performance characteristics, introducing their capabilities in terms of scalability, and indexing techniques. The aim is to guide researchers and practitioners in selecting the most suitable vector database for their specific use cases in data retrieval and machine learning applications.

***Index Terms*—Vector databases, Nearest Neighbor Search, AI-driven Analytics, Graph-based Structures, Product Quantization**

I. Introduction

Approximate Nearest Neighbor (ANN) search has become a cornerstone for solving high-dimensional data retrieval problems, with applications spanning recommendation systems [2], [8], [39], medical diagnosis [18], Retrieval-Augmented Generation (RAG) [12], [21], and more. As data grows in scale and complexity, developing efficient and scalable ANN methods has become increasingly important.

ANN search has become a critical component in numerous applications, such as recommendation systems, search engines, and vector-based AI solutions. Over the years, several methods have demonstrated exceptional performance in solving high-dimensional similarity search problems. Techniques like HNSW (Hierarchical Navigable Small World) [34], DiskANN [19], [25], [42], and IVF_PQ (Inverted File Index with Product Quantization) [11] have achieved significant success due to their efficiency, scalability, and adaptability to large datasets.

In recent years, the growing demand for efficient similarity search in high-dimensional spaces has led to the development of specialized vector databases. Among the most widely used vector databases are Faiss [11] and Milvus [44], both of which offer robust solutions for vector search tasks, albeit with distinct design philosophies and features.

The combination of traditional databases with vector databases offers a compelling solution for a variety of use case [36], [46]. Traditional databases continue to manage structured metadata, transactional data, and complex queries, while vector databases handle the fast retrieval of unstructured data representations, like embeddings. The integration of both types of databases enables a holistic approach where structured data can be joined seamlessly with unstructured vector data. This combination is particularly powerful in applications where both types of information—e.g., user profiles (structured data) and content embeddings (vector data)—are needed to make more personalized and accurate predictions.

Beyond algorithmic advancements, several modern database systems now natively support vector storage and similarity search, enabling seamless integration with large-scale applications. Systems like AnalyticDB-V [46] and PostgreSQL [10] (with extensions like pgvector) provide built-in functionality for managing and querying vector data. These systems bridge the gap between ANN methods and production environments, making vector search accessible and scalable for real-world applications.

Despite the high availability and utility of existing methods like HNSW, DiskANN, IVF_PQ and those vector search algorithms, there are notable limitations in daily usage. ANN search often needs to work in tandem with specific attributes and restrictions. For example, in recommending commodities, filters such as price limitations or categorical constraints must be applied alongside similarity search, requiring a combination of ANN algorithms with filtering mechanisms.

The challenge, however, lies in the integration of these two systems. Since traditional databases are not designed to handle high-dimensional vectors, retrieving vectors from a traditional database for similarity search can result in significant delays and inefficiencies. However, by combining both systems—using traditional databases for structured data and vector databases for vector-based search—organizations can achieve a more efficient and scalable solution. For instance, the traditional database can store metadata related to the vectors (e.g., identifiers, categories, attributes), while the vector database handles the high-performance retrieval of similar vectors, allowing applications to benefit from both structured and unstructured data in a seamless manner.

Several databases are specifically designed to handle vector similarity search, offering efficient and scalable solutions for high-dimensional data retrieval. These databases play a crucial role in applications like recommendation systems, semantic search, and AI-powered analytics. Some notable examples

include Milvus [44], Pinecone [47], Qdrant, and FAISS [11]. These systems are purpose-built for efficient vector storage and retrieval, offering features like approximate nearest neighbor (ANN) search, scalability, and hybrid query support. Their designs address the unique challenges of high-dimensional data, ensuring performance and accuracy in demanding real-world applications.

This paper aims to provide an in-depth introduction of Faiss [11], ADBV [46] and Milvus [44], highlighting their respective strengths and use cases. We will examine their architectural designs, indexing strategies, query performance, and ease of integration with existing machine learning frameworks. By understanding the core principles of each database, this paper will offer valuable insights for researchers and engineers seeking to implement efficient vector search solutions in various applications. By analyzing these two approaches, this paper aims to highlight their unique techniques, challenges, and trade-offs, providing insights into their applications and potential future advancements in ANN search.

II. Related Work

In this section, I introduce the background of vector database, focusing on three notable techniques: ANN search algorithm [11], [25], [34], OLAP system, and the development of searching on unstructured data. These tpoics form the foundation for efficient and scalable vector batabase system.

A. Approximate Nearest Neighbor Search

The primary objective of ANN search is to efficiently identify the closest data points to a given query point in high-dimensional spaces. This problem is challenging due to the curse of dimensionality, which makes exact nearest neighbor search computationally expensive and infeasible for large-scale datasets. To overcome this, ANN search techniques aim to provide approximate results with reduced computational complexity while maintaining a high level of accuracy.

Several tree-based algorithms have been proposed with theoretical guarantees [23], [40] in an attempt to reduce the running time of the brute force solution. However, because of the curse of dimensionality [24], when it comes to high dimensional features, these algorithms do not perform any better than an exhaustive search [45].

Now the well applied ANN search methods are typically classified into two broad categories: graph-based algorithms and quantization-based methods. Each of these approaches has its own strengths and trade-offs, making them suitable for different applications and use cases.

1) Graph-based Algorithms: Graph-based methods represent the dataset as a graph, where each data point is a node, and edges connect points that are similar to each other. These algorithms aim to create efficient data structures that allow for fast search in high-dimensional spaces by exploiting local and global graph structures. Some of the most widely used graph-based ANN algorithms include:

- **NSG (Navigable Small World Graph) [35]** The NSG algorithm is designed for efficient nearest neighbor search in unstructured data. The graph is structured to facilitate fast querying by maintaining a "small world" property, meaning that most nodes are connected to a few others that can efficiently lead to any other node in the graph.
- **HNSW (Hierarchical Navigable Small World)**: HNSW [34] is a highly efficient algorithm that leverages a hierarchical graph structure. It builds a multi-layer graph where the top layers contain fewer nodes and represent the global structure, while the lower layers contain more nodes and focus on local connectivity. Its structure is shown in figure 1
- **Vamana Graph**: Vamana graph [25] constructs one layer graph in a fast way by connecting and pruning form a random connected graph. It is designed to optimize the trade-off between search accuracy and computational efficiency, making it suitable for large-scale data retrieval tasks.

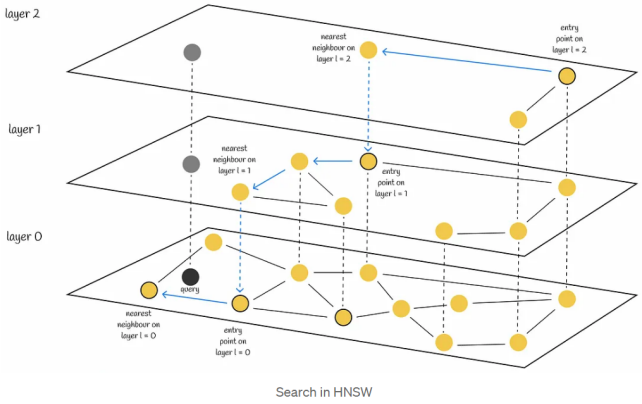


Fig. 1. Example of HNSW

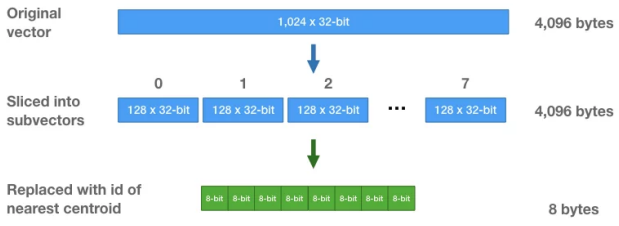


Fig. 2. Example of PQ

2) Quantization-based Methods: Quantization-based methods work by approximating the data points using a smaller set of representative values, or centroids, which reduces the complexity of the search task. These methods are particularly effective for very large datasets, as they reduce the amount of data that needs to be stored and compared during the search. Common quantization-based ANN algorithms include:

- **PQ (Product Quantization)**: PQ is a technique that splits the data space into multiple subspaces, each of which is quantized separately, shown in figure 2. This

allows the algorithm to represent each data point as a product of smaller quantized components, significantly reducing storage requirements and speeding up distance computations. PQ is widely used in applications such as image and video retrieval.

- **SQ (Scalar Quantization)**: SQ [51] is a simpler form of quantization where each data vector is approximated by a single scalar value (usually the centroid of a cluster). This method is less computationally expensive than PQ, but it may not be as accurate when dealing with high-dimensional data, as it does not capture the full structure of the data.
- **OPQ (Optimized Product Quantization)**: OPQ [16] extends PQ by jointly optimizing the quantization process across multiple subspaces. This optimization improves the overall approximation of the original data, leading to better search performance and higher accuracy compared to standard PQ.
- **RaBitQ**: RaBitQ [15] is a recent development in quantization methods that aims to improve upon the traditional PQ and OPQ by introducing more advanced optimization techniques and exploring non-linear quantization approaches. It is designed to handle high-dimensional data more efficiently, particularly in cases where other quantization methods may struggle with accuracy or scalability.

3) Inverted File Index (IVF): IVF method are often combined with quantization methods. IVF divides the dataset into coarse partitions, each represented by a centroid. During indexing, The dataset is clustered into K partitions using algorithms like k-means. Each vector is assigned to its closest centroid. During querying, The query vector is matched to its closest centroids, limiting the search to only the associated partitions.

B. OLAP system.

Most OLAP systems like OLAP databases (Vertica [29], Greenplum [1], etc.), batch processing systems (Spark-SQL [3]), analytical cloud services (Amazon Redshift [22], Google BigQuery [41] and Alibaba AnalyticDB [49]) have been used in a wide range of fields and provided excellent analytic ability for users in practice. However, these systems only work on traditional structured datasets and do not support hybrid queries containing unstructured data.

C. Solutions for searching unstructured data.

Jegou et al. implement the extension on Elasticsearch [13] to support ANNS over vector data. Microsoft develops GRIP [50], a vector search solution that adopts HNSW to reduce the expensive centroid searching cost of encodingbased algorithms and supports different kinds of vector indexes [14], [26]. Recently, unstructured data processing arise a lot of interests [30], [37], [44].

III. PRELIMINARIES

In this section, we will present the notations and terminology used in this paper.

Notation	Meaning
d	vector dimension
N	number of vectors
$q \in \mathbb{R}^d$	query vector
$x_i \in \mathbb{R}^d$	i^{th} database vector
k	number of requested results
$\varepsilon \in \mathbb{R}^+$	radius for range search
K	number of centroids for quantization
M	number of sub-quantizers

TABLE I
NOTATIONS

Table I shows the common notations used throughout the paper. Let $x \in \mathbb{R}^d$ be a query point, where d is the dimension of a vector. $d(a, b)$ denotes the distance between vector a and vector b. The L2 distance, also known as Euclidean distance $d(a, b) = \|a - b\|$. Inner product $\langle a, b\rangle = \sum_{i=1}^{n} a_i b_i$.

Let vector x_i in dataset D with its size $n = |D|$, and a set of queries $q \in Q$, the goal of nearest neighbor search is to get the top k nearest neighbors in a certain metrics. It can be written as:

$NN_k(q_i, d) = \{\{x_m\}_1^k | d(q_i, x_m) \leq d(q_i, x_l)\}$, where $\{x_m\}_1^k \cap \{x_l\}_1^{n-k} = \emptyset$

IV. FAISS

FAISS (Facebook AI Similarity Search) [11] is an open-source library developed by Facebook AI Research for efficient similarity search and clustering of high-dimensional vectors. It is designed to handle large-scale datasets, making it ideal for applications like image retrieval, recommendation systems, and natural language processing, where vectors (embeddings) represent data points. FAISS provides optimized algorithms for nearest neighbor search, supporting both exact and approximate methods.

The library supports a variety of indexing structures, such as Flat (exact search) and IVF (Inverted File Index for approximate search), allowing users to balance between search speed and accuracy. FAISS is highly optimized for performance, with GPU support to accelerate computation and reduce search times. It can be easily integrated into machine learning pipelines and handles datasets ranging from small to extremely large, offering efficient memory management and scalability. Its flexibility and speed make FAISS a popular tool for similarity search tasks in AI applications.

A. Structure

Faiss originated in a research environment. Consequently, it grew organically as indexing research advanced. Below, we'll briefly cover the guiding principles that maintained the library's coherence, how optimization is carried out, and an example demonstrating how Faiss internals are exposed for it to be embedded in a vector database.

1) Code Structure: The core of Faiss is implemented in C++. Its guiding principles are as follows:

- The code should be as open as possible to enable users to access all the implementation details of the indexes.
- Faiss should be easy to embed from external libraries.
- The core library concentrates on vector search only.

Faiss's basic data types are concrete (not templates). Vectors are consistently represented as 32-bit floats, which are portable and strike a good balance between size and accuracy. Likewise, all vector ids are represented using 64-bit integers. Although this is often larger than needed for sequential numbering, it is widely employed for database identifiers.

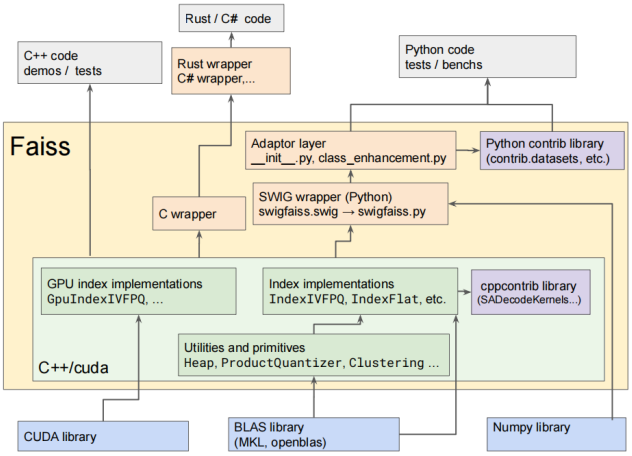


Fig. 3. Architecture of the Faiss library

2) High-level Interface: Figure 3 illustrates the structure of the library. The C++ core library and the GPU add-on have minimal dependencies, requiring only a BLAS implementation and CUDA itself.

To enable experimentation, the entire library is wrapped for scripting languages like Python with numpy. For this purpose, SWIG5 exhaustively generates wrappers for all C++ classes, methods, and variables. The associated Python layer also incorporates benchmarking code, dataset definitions, and driver code. Moreover, an increasing amount of functionality is being embedded in the contrib package of Faiss.

Faiss also offers a pure C API, which is beneficial for creating bindings with programming languages such as Rust or Java. The Index is presented to the end user as a monolithic object, even when it incorporates other indexes like quantizers, refinement indexes, or sharded subindexes. As a result, an index can be duplicated using clone index and serialized into a single byte stream via the write index function. It also includes the necessary headers so that it can be read by the generic read index function.

The index factory. Index objects can be instantiated explicitly in C++ or Python, yet it's more common to build them using the index factory function. This function accepts a string that details the index structure and its main parameters. For instance, the string $PCA160, IVF20000\_HNSW, PQ20x10, RFlat$ creates an IVF index where $K_{IVF} = 20000$. In this setup, the coarse quantizer is a HNSW index, the vectors are represented by a PQ20x10 product quantizer. The data undergoes preprocessing with a PCA to reduce it to 160 dimensions, and the search results are re-ranked using a refinement index that conducts exact distance computations. All index parameters are set to reasonable defaults; for example, the PQ encodes the residual of the vectors with respect to the coarse quantization centroids. Faiss indexes can also be utilized as vector codecs with functions like sa_encode, sa_decode, and sa_code_size.

3) Optimization: **Approach to optimization:** Faiss initially focuses on achieving feature completeness. All indexes are first implemented in a non-optimal form. Optimization of the code is carried out only when it becomes clear that the runtime performance matters significantly for a specific index. The non-optimized implementation is used to ensure the accuracy of the optimized version.

Typically, only a portion of data sizes are subjected to optimization. For instance, with PQ indexes, only when K is equal to 2^8 or 2^4 and $d/M \in \{2, 4, 8, 16, 20\}$ are they fully optimized. Regarding IndexLSH search, only code sizes 4, 8, 16, and 20 are optimized. By specifying these sizes, it becomes possible to create "kernels" – sequences of instructions without explicit loops or conditional tests – which aim to maximize arithmetic throughput.

Once the generic scalar CPU optimizations have been fully utilized, Faiss proceeds to conduct specific optimizations tailored to certain hardware platforms.

CPU vectorization. Modern CPUs are capable of Single Instruction, Multiple Data (SIMD) operations, such as AVX/AVX2 on x86 architectures and NEON on ARM architectures. Faiss takes advantage of these at three distinct levels. At the first level, when operations are relatively straightforward (for example, an element-wise vector sum), the code is structured in a manner that enables the compiler to vectorize it independently. This usually involves adding "restrict" keywords to indicate that arrays do not overlap.

The second level makes use of SIMD variables and instructions via C++ compiler extensions. Faiss incorporates $simdlib$, which is a collection of classes designed to serve as a layer above the AVX and NEON instruction sets. However, a significant portion of the SIMD work is tailored specifically for one instruction set, most commonly AVX, as it tends to be more efficient.

The third level of optimization focuses on adapting the data layout and algorithms to accelerate their SIMD implementation. The 4-bit product and additive quantizer implementations follow this approach, inspired by the SCANN library [20]. Specifically, the layout of the PQ codes for several consecutive vectors is interleaved in memory. This allows for a vector permutation to be employed, enabling parallel LUT lookups. This particular implementation is carried out in the FastScan variants of PQ and AQ indexes ($IndexPQFastScan, IndexIVFResidualQuantizerFastScan$, etc.).

GPU Faiss. Porting Faiss to the GPU is a complex task due to significant architectural differences. The details of the GPU Faiss implementation are provided in [28], and here we summarize the challenges faced during its GPU implementation. Modern multi-core CPUs are highly optimized for latency. They utilize an elaborate cache hierarchy, branch prediction, speculative execution, and out-of-order code execution to enhance the execution of serial programs. In contrast, GPUs have a more limited cache hierarchy and omit many of these latency optimizations. Instead, they possess a large number of concurrent threads of execution (for instance, Nvidia's A100 GPU can support up to 6,912 warps, with each warp roughly equivalent to a 32-wide vector SIMD CPU thread of execution). They also have a substantial number of floating-point and integer arithmetic functional units (the A100 has up to 19.5 teraflops per second of fp32 fused-multiply add throughput) and a massive register set that enables a high number of long-latency pending instructions to be in progress simultaneously (the A100 has 27 MiB of register memory). Hence, GPUs are largely throughput-optimized machines. The algorithmic techniques employed in vector search can be categorized into three broad groups: Distance Computation: This involves calculating the distance of floating-point or binary vectors (which might have been obtained through dequantization from a compressed form). GPUs handle distance computation with ease and can readily outperform CPUs in this regard, as they are optimized for matrix-matrix multiplication like that seen in IndexFlat or IVFFlat. Table Lookups and Scanning: Such as those in PQ distance computations or when traversing IVF lists. These operations can also be made efficient on GPUs. It's possible to stage small tables (as in product quantization) in shared memory (which is roughly a user-controlled L1 cache) or register memory and conduct lookups in parallel across all warps. For sequential table scanning in IVF indices, where data needs to be loaded from main (global) memory, despite the high access latencies, since we know beforehand what data to access, the data movement can be pipelined or use double buffering to achieve close to peak performance.

Irregular, Sequential Computations: Examples include linked-list traversal (used in graph-based indices) or ranking the k closest vectors. On the CPU, ranking distances to select the k closest vectors is typically implemented using a min- or max-heap. However, on the GPU, the sequential nature of heap operations would push it into a latency-bound regime. This is the biggest challenge for the GPU implementation of vector search, as the time required for the heap implementation is an order of magnitude greater than that for all other arithmetic operations. To address this, an efficient GPU k-selection algorithm [Johnson et al., 2019] was developed. It enables ranking candidate vectors in a single pass and operates at a significant fraction of the peak possible performance within memory bandwidth limits. It relies heavily on the high-speed, large register memory on GPUs and uses small-set bitonic sorting via warp shuffles with buffering techniques.

For irregular computations like traversing graph structures in graph-based indices such as HNSW, they tend to remain in the latency-bound regime (due to sequential traversal) rather than being bound by arithmetic throughput or memory bandwidth. In such cases, GPUs are at a disadvantage compared to CPUs, and emerging techniques like CAGRA [Ootomo et al., 2023] are needed to parallelize otherwise sequential graph traversal operations.

B. *Index*

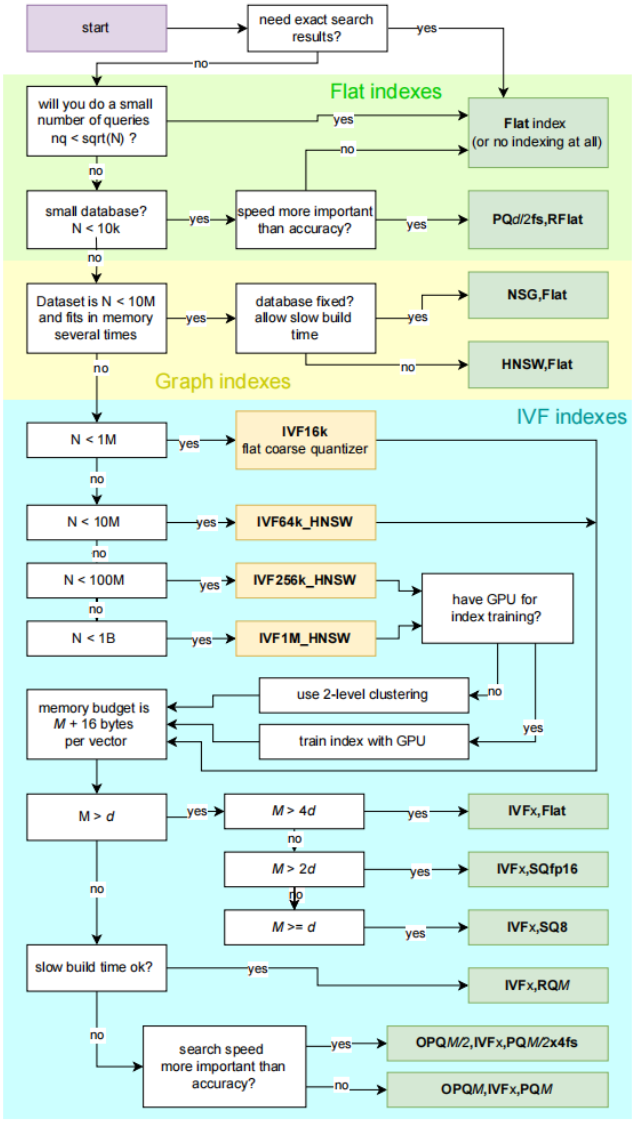


Fig. 4. Decision tree to choose a Faiss index

For most cases, choosing an appropriate index can be done by following the process in Figure 4. First, determine if indexing is needed. In some cases, a direct brute force search is the best option. Otherwise, choose between IVF and graph-based indexes.

An IVF index can be considered a special case of a graph-based index, especially when a small graph-based index is used as a coarse quantizer. Graph-based indices are suitable

for indexes with no memory usage constraint, typically for those below 1M vectors. For indexes beyond 10M vectors, construction time usually becomes the limiting factor. For larger indexes that require compression to fit database vectors in memory, IVF indexes are the only option. The decision tree in Figure 4 offers initial directions. The Faiss wiki[1] has comprehensive benchmarks for various database sizes and memory budgets. To refine index parameters, benchmarking should be used.

C. Experiments

encoding time. Figure 5 illustrates the tradeoff between encoding time and Mean Squared Error (MSE). When considering a specific code size, it turns out to be more accurate to utilize a smaller number of sub-quantizers (M) along with a higher K value.

Regarding GPU encoding for Learned Step Quantization (LSQ), it doesn't consistently provide an advantage. The Look-Up Table (LUT)-based encoding of Residual Quantization (RQ) is quite interesting in the context of RQ/Product Residual Quantization (PRQ) quantization, especially when the beam size is larger.

In the 64-byte regime, it can be observed that LSQ isn't as competitive as RQ. Progressive Learned Step Quantization (PLSQ) and PRQ, on the other hand, gradually become more competitive when dealing with larger memory budgets. Moreover, they are also faster because they operate on smaller vectors.

Vector compression benchmark. Figure 6 presents the tradeoff between code size and accuracy for numerous variants of the codecs.

Additive quantizers prove to be the optimal choice when it comes to small code sizes. For larger code sizes, it's advantageous to independently encode several sub-vectors by employing product-additive quantizers.

Learned Step Quantization (LSQ) is more accurate than Residual Quantization (RQ) for small codes, yet it doesn't scale effectively to longer codes. It's worth noting that product quantizers are somewhat less accurate than additive quantizers; however, given their short encoding time, they remain an appealing option.

Scalar quantizers perform well for very long codes and are even faster. The 2-level Product Quantization (PQ) options are what an IVFPQ index utilizes for encoding: there's a first-level coarse quantizer and a second-level refinement of the residual.

$\mathbf{K_{IVF}}$ settings. Figure 7 displays the optimal settings of K_{IVF} for different database sizes. When K_{IVF} is set to a small value like 4096, the coarse quantization runtime becomes negligible, and the search time increases linearly in relation to the database size. For larger datasets, it's advantageous to raise the value of K_{IVF}. As indicated in (18), the ratio of K_{IVF} to the square root of N is approximately 15 to 20. It should be noted that this ratio depends on the data distribution as well as the target accuracy.

[1]https://github.com/facebookresearch/faiss/wiki/Indexing-1G-vectors

PQ settings. Figure 8 demonstrates that encoding residuals is advantageous when dealing with shorter codes. However, for larger codes, the contribution made by the residual becomes less significant. In fact, given that the original data has 96 dimensions, it can be compressed to 64 bytes with relatively good accuracy.

V. AnalyticDB-V

AnalyticDB-V is a high-performance, distributed database designed for real-time analytical processing on massive volumes of vector data. Developed by Alibaba Cloud, AnalyticDB-V specializes in handling vector-based data, commonly used in machine learning, artificial intelligence, and recommendation systems, where high-dimensional vectors such as embeddings are integral. The database offers seamless integration with vector search, enabling fast retrieval of similar items in datasets with millions or billions of vectors.

Key features of AnalyticDB-V include high scalability, distributed processing, and low-latency querying, allowing users to efficiently store and process large amounts of unstructured data. It leverages advanced indexing techniques, such as HNSW and PQ, to optimize nearest neighbor search and support real-time data analytics. Additionally, AnalyticDB-V provides an intuitive interface for data scientists and developers, enabling integration with AI models and machine learning pipelines. This makes it ideal for applications in recommendation engines, image search, and natural language processing tasks requiring fast, accurate vector retrieval at scale.

The system uses a columnar storage model, which reduces disk I/O and improves compression rates, making it ideal for OLAP (Online Analytical Processing) scenarios. It also supports seamless scaling, providing elasticity to handle fluctuating workloads and large data volumes. AnalyticDB-V is compatible with popular SQL interfaces and integrates with various big data processing tools, facilitating easy adoption in data-driven applications such as business intelligence, machine learning, and real-time data analytics. Its architecture ensures fault tolerance and high availability, ensuring continuous operation without data loss.

A. Architecture Overview

The architecture of ADBV is presented in Figure 9 and is mainly composed of three types of nodes: Coordinator, Write Node, and Read Node. Coordinators accept, parse, optimize SQL statements, and dispatch them to read/write nodes. ADBV adopts a typical read/write decoupling approach [49], which trades consistency for low query latency and high write throughput. As a result, write nodes are solely responsible for write requests (i.e., INSERT, DELETE, and UPDATE), while read nodes handle SELECT queries. Newly ingested data is flushed into Pangu upon commit.

In the storage layer, ADBV adopts the lambda framework to manage vectors efficiently. The streaming layer handles real-time data insertion and modification, while the batching layer periodically compresses newly inserted vectors and rebuilds

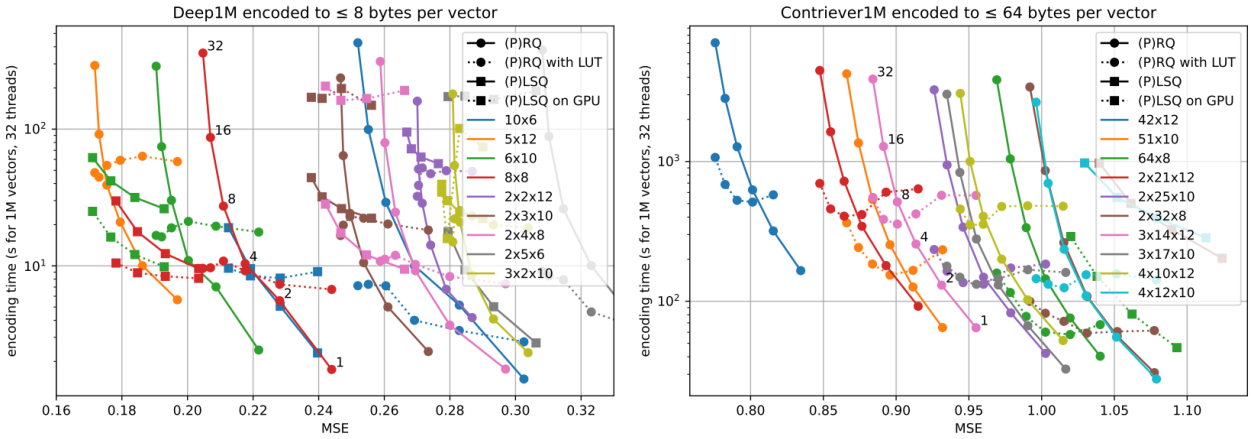


Fig. 5. Comparison of additive quantizers in terms of Encoding time vs. accuracy (MSE)

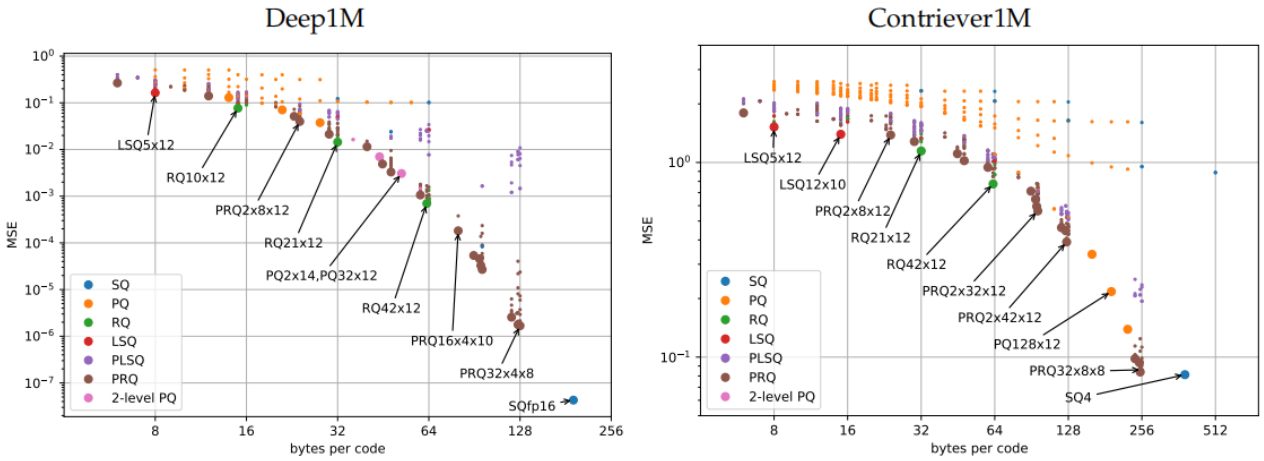


Fig. 6. Tradeoff of accuracy vs. code size for different codecs on the Deep1M and Contriever1M datasets

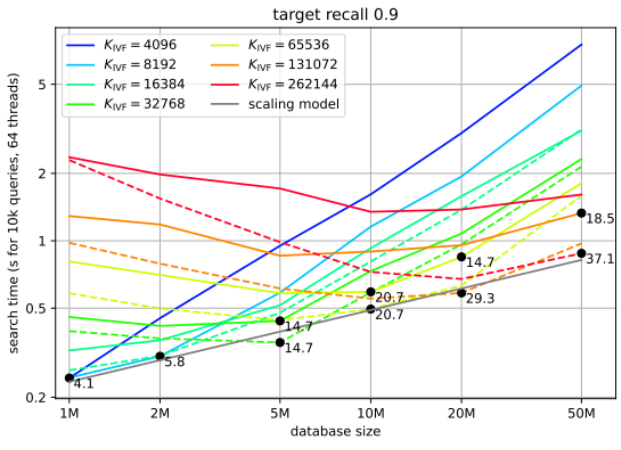


Fig. 7. Search time as a function of the database size N for BigANN1M with different K_{IVF} settings

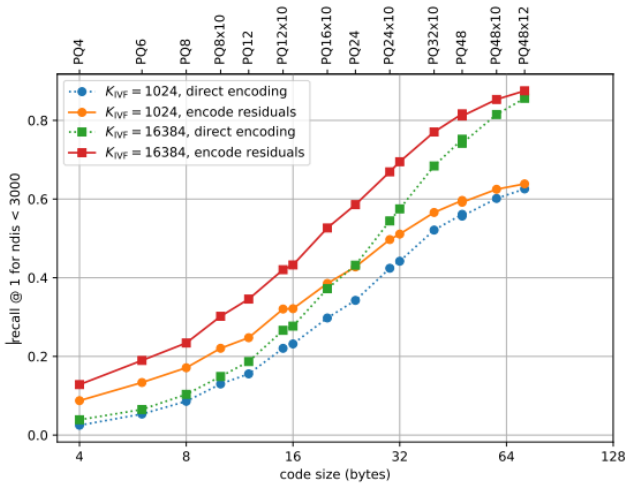


Fig. 8. Comparing IVF indexes with and without residual encoding for $K_{IVF} \in \{2^{10}, 2^{14}\}$ on the Deep1M dataset (d=96 dimensions), with different product quantization settings.

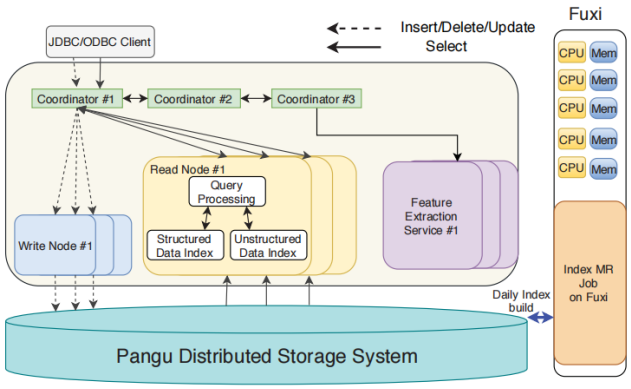


Fig. 9. Analytic DB-V architecture

ANNS indexes. Additionally, ADBV pushes down several expensive predicates to the storage layer to fully utilize the computational capability of the nodes.

B. Lambda Framework

The complexity of searching over the entire vector dataset is unacceptable, so an index must be built to mitigate the cost. However, traditional index techniques like KD-tree [6] and ball-tree [38], which are widely used in low dimensions, do not perform well for high-dimensional vectors generated by deep learning models. It has been empirically proven that such solutions exhibit linear complexity for high-dimensional vectors [45].

To address this, algorithms like HNSW (Hierarchical Navigable Small World) [34] and LSH (Locality-Sensitive Hashing) [17] have been proposed for real-time, approximate index building on vectors.

To address the challenge of supporting real-time inserts, we adopt the lambda framework. Under this framework, ADBV uses HNSW to build an index for newly-inserted vectors (i.e., incremental data) in real time. Periodically, ADBV merges the incremental data with baseline data into a global index using the proposed VGPQ algorithm and discards the HNSW index.

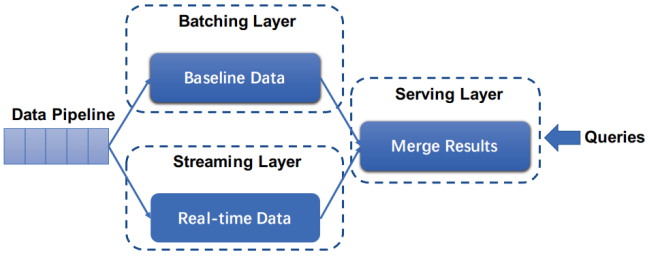


Fig. 10. Analytic DB-V Lambda Framework

Figure 10 illustrates the lambda framework, which consists of three layers: the batching layer, the streaming layer, and the serving layer. These layers work together to process each incoming query. The batching layer returns search results based on baseline data. The streaming layer performs two tasks: processing data modifications (i.e., INSERT, DELETE, and UPDATE) and producing search results over incremental data.

ADBV contains two types of data: baseline data and incremental data. The incremental data includes newly-written WALs (stored in Pangu), along with vector data and its indexes on read nodes. It also contains a data-status bitset to track which vector data has been deleted. Incremental raw data and indexes are much smaller in size compared to baseline data and can be entirely cached in memory. HNSW is used to build the index for incremental data, allowing index building and search to be conducted simultaneously.

C. Clustering-based Partitioning

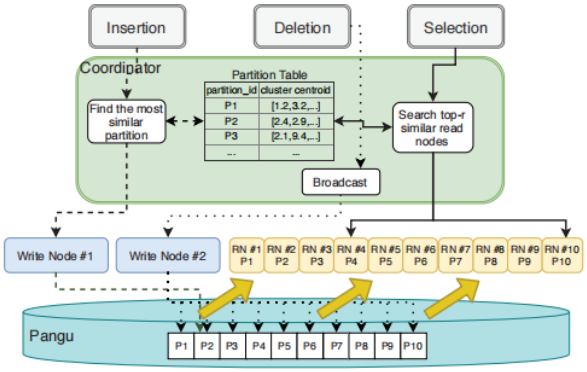


Fig. 11. Clustering-based partition pruning

As shown in Figure 11, ADBV provides the ability to partition vector data across multiple nodes to achieve high scalability. However, partitioning techniques used for structured data, such as hash and list partitioning, are unsuitable for vector data. These techniques rely on equivalence and range assessment, while analytics on vector data is based on similarity (e.g., Euclidean distance). Directly adopting these strategies would require queries to execute on all partitions indiscriminately, without any pruning effect.

To solve this problem, we propose a clustering-based partitioning approach. For the partitioned column, we apply k-means [17] to calculate centroids based on the number of partitions. For example, if we define 256 partitions, 256 centroids are calculated. Each vector is then clustered to the centroid with the largest similarity, forming a partition for each cluster. Index building and data manipulation are subsequently conducted on each individual partition.

For partition pruning, ADBV dispatches the query to N partitions, which are the most similar to the queried vector. N is a query hint defined by users, reflecting the trade-off between query performance and accuracy.

D. Vector Processing Algorithms

Hybrid query processing in ADBV relies heavily on dedicated approximate nearest neighbor search (ANNS) algorithms. These efficient ANNS algorithms are implemented

as physical operators to handle top-k retrieval tasks. In this subsection, we introduce how ANNS algorithms are used to process vector queries and propose a novel ANNS algorithm, VGPQ (Voronoi Graph Product Quantization), to further improve query efficiency, particularly in the batching layer.

1) Vector Query Processing: To enable fast retrieval on high-dimensional datasets, a trade-off between accuracy and running time is needed. ADBV uses neighborhood-based algorithms for nearest neighbor search in the streaming layer and quantization-based algorithms in the batching layer to improve efficiency.

At the streaming layer, ADBV implements HNSW. While it supports dynamic insertion, it cannot scale to large datasets, making it suitable for querying newly-inserted data. At the batching layer, ADBV uses PQ to encode vectors with compact, low-dimensional, and lossy representations, reducing pairwise distance calculation costs. However, PQ requires offline training of its codebook. To handle large-scale data, IVFPQ [26] clusters PQ codes with k-means and scans relevant groups during queries. ADBV improves efficiency further with VGPQ.

2) Voronoi Graph Product Quantization: VGPQ (Voronoi Graph Product Quantization) improves query efficiency in IVFPQ by partitioning Voronoi cells into subcells based on centroid midpoints. In IVFPQ, vectors are clustered using k-means, and each vector's PQ code is linked to the nearest centroid. VGPQ refines this by dividing each centroid's Voronoi cell into subcells, focusing on the relevant subcells during query processing. The preprocessing steps involve k-means clustering, neighbor selection, subcell construction using midpoints, and inserting PQ codes into inverted files. VGPQ reduces construction costs by selecting the top-b nearest centroids for subcell creation.

3) Storage Design For VGPQ: In VGPQ's in-memory storage structure, three components are designed to optimize execution: Indexing data, PQ data, and Direct map. PQ data stores PQ codes in fixed-size pages, organized by subcell. Indexing data contains feature vectors and anchors, linking each centroid to subregions. When a page is full, its pageid is added to the corresponding linked list in Indexing data. The Direct map establishes a bi-directional relationship between rowid and PQ code location, enabling efficient access. This design ensures a single copy of each PQ code and supports hybrid queries, enhancing data access and query performance.

4) hybrid Query Optimization: The optimizer of ADBV analyzes the Abstract Syntax Tree (AST) from the query parser to identify patterns for retrieving top-k tuples based on unstructured columns (e.g., "$orderbyDISTANCE()LIMITk$"). When this pattern is detected, the optimizer converts the logical plan into multiple physical execution plans, incorporating ANNS algorithms wrapped into anns scan nodes for approximate nearest neighbor searches. These scans reduce the computational cost of retrieving top-k neighbors from large datasets. Four effective physical plans are proposed, considering different ANNS algorithms, and are discussed in the context of the query example shown in Figure 12.

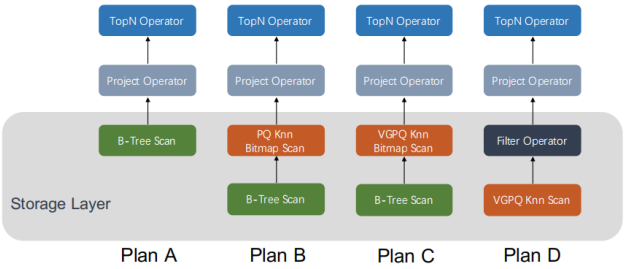


Fig. 12. Physical plans

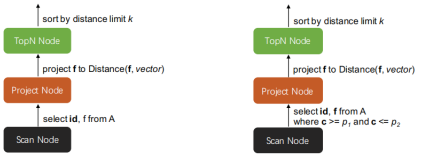
E. EXPERIMENTS

In this subsection, ADBV is evaluated using both public and in-house datasets to assess the effectiveness of the proposed designs. The evaluation focuses on demonstrating the performance improvements of ADBV compared to existing solutions.

1) Experimental setup: **Environment:** We conduct the experiments with a 16-node cluster on Alibaba Cloud; each node has 32 logical cores, 150GB DRAM and 1TB SSD. The host machines are all equipped with one Intel Xeon Platinum 8163 CPU (2.50GHz), and hyperthreading is triggered. Machines are connected via a 10Gbps Ethernet network.

Dataset: We use two public datesets and one in-house dataset to evaluate our system, as listed below:

- Deep1B [4] is a public dataset consists of image vectors extracted from a deep neural network. It contains one billion 96-dimensional vectors.
- SIFT1B [27] is a public dataset consists of handcrafted SIFT features. It contains one billion 128-dimensional vectors.
- AliCommodity consists of 830 million 512-dimensional vectors extracted from commodity images used at Alibaba. It also contains 21 structured columns including color, sleeve type, style, create time, etc.



(a) $Q1$ Logical Plan (b) $Q2$ Logical Plan

Fig. 13. Logical plans

Query types: We reuse the two query templates provided in figure 13 for evaluation. According to our observations in production environments, most of queries processed by ADBV follow these patterns.

Metrics: We use the recall to measure the accuracy of a result set returned by an ANNS algorithm (or a system).

Suppose the exact result set is S, the recall is defined as $recall = |S \cap S'|/|S|$, where S is the result set returned by a ANNS algorithm (or a system) and $||$ computes the carnality of a set. We also use the recall in Top-k results (recall@Top$k$) to evaluate the system performance, which means $|S| = |S'| = k$.

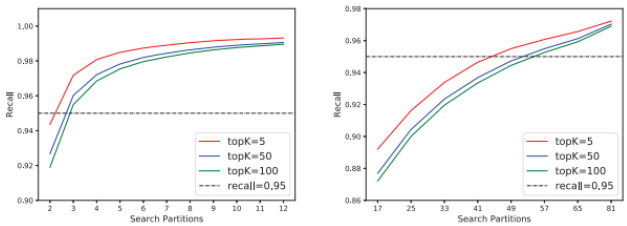
2) VGPQ: We compare VGPQ and IVFPQ across three aspects: accuracy, index construction time, and index file size.

method	time (min)	size (GB)
IVFPQ(4096)	155	112
IVFPQ(8192)	199	112
VGPQ(4096,64)	144	112
VGPQ(8192,64)	178	112
VGPQ(8192,128)	182	112

TABLE II
THE CONSTRUCTION TIME AND INDEX SIZE COMPARISON BETWEEN VGPQ AND IVFPQ ON ALICOMMODITY

Table II lists the construction time and index file size for different parameter settings on AliCommodity. It shows that the index file sizes generated by VGPQ and IVFPQ are very similar. The construction time for both methods primarily depends on the number of centroids in k-means (referred to as 'n clusters' in Algorithm 1). The performance of VGPQ is also influenced by the number of subcells. However, we observe that adjusting the number of subcells does not significantly affect the construction time of VGPQ.

3) lustering-based partition pruning: We demonstrate the effect of clustering-based partition pruning on query throughput in ADBV. Two distributed data tables, each with 512 clustering-based partitions, are created for SIFT1B and Deep1B.



(a) SIFT1B_512p (b) Deep1B_512p

Fig. 14. Performance analysis for clustering-based partition pruning

As shown in Figure 14, recall improves with respect to different top-k settings as the number of partitions searched increases. However, searching more partitions also leads to lower query throughput. We observe that partition pruning is more effective for queries with relatively small k. Based on our empirical experience, clustering-based partition pruning enables ADBV to achieve significant throughput improvements (ranging from 10× to 100×) on large-scale datasets (with 1000+ storage nodes), particularly for queries with small k, which is common in real-world applications.

4) Hybrid query optimization: We evaluate the accuracy-aware, cost-based optimization for hybrid query execution. We demonstrate that the proposed approach ensures the accuracy of query results while identifying the optimal physical plan across various scenarios.

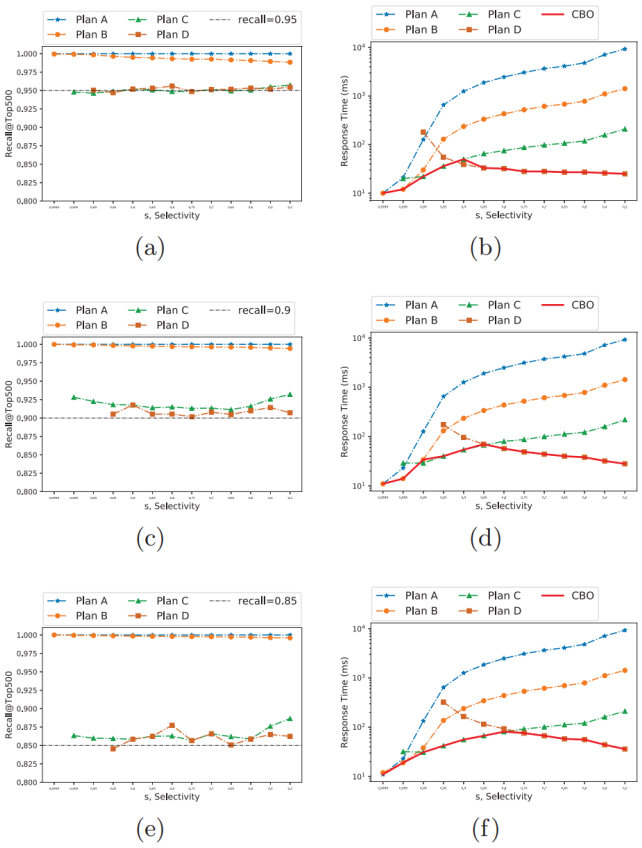


Fig. 15. Performance analysis for clustering-based partition pruning

As illustrated in the left column of Figure 15, most of these plans can provide results that meet the required recall under different selectivities. This suggests that our accuracy-aware cost-based optimizer (CBO) is effective in selecting appropriate hyperparameters for each physical plan. Additionally, in all three representative cases, the optimizer consistently selects the optimal plan from the four options, as shown in the right column of Figure 12. This confirms that the proposed cost models are valuable in most scenarios.

VI. MILVUS

This section introduces Milvus, a purpose-built data management system designed for storing and searching large-scale vector data in data science and AI applications. Unlike generalized relational databases that adapt to support vectors, Milvus follows the design philosophy of 'one-size-does-not-fit-all' [43], making it a specialized system for high-dimensional vectors. Milvus offers a wide range of application interfaces, including SDKs in Python, Java, Go, and C++, as well as RESTful APIs, enabling easy integration with applications. It is highly optimized for heterogeneous computing architectures,

leveraging modern CPUs and GPUs (including multiple GPU devices) to maximize efficiency. Additionally, Milvus supports various query types, including vector similarity search with multiple similarity functions, attribute filtering, and multi-vector query processing

Milvus supports various types of indexes, including quantization-based indexes [26], [28] and graph-based indexes [14], [34], and provides an extensible interface for easily incorporating new indexes into the system. To manage dynamic vector data, such as insertions and deletions, Milvus uses an LSM-based structure while ensuring consistent real-time searches through snapshot isolation. As a distributed data management system, Milvus is deployed across multiple nodes to provide scalability and availability.

A. System Design

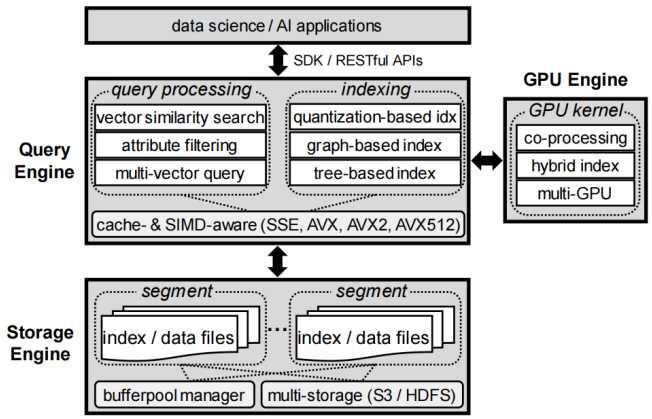


Fig. 16. Milvus System

Figure 16 illustrates the architecture of Milvus, which consists of three major components: the query engine, the GPU engine, and the storage engine.

Query Engine: It handles client query processing over vector data, optimized for modern CPUs. The query engine minimizes cache misses and utilizes SIMD instructions to enhance performance.

GPU Engine: This co-processing engine accelerates performance through vast parallelism and supports multiple GPU devices to further improve efficiency.

Storage Engine: Responsible for data durability, the storage engine uses an LSM-based structure for dynamic data management. It operates on various file systems (e.g., local file systems, Amazon S3, and HDFS) and includes a buffer pool in memory.

This structure emphasizes the functionality and role of each component in Milvus' architecture.

1) Query Processing: We first present the concept of entity used in Milvus and then explain query types, similarity functions, and application interfaces.

Entity. To effectively address a wide range of data science and AI applications, Milvus supports query processing for both vector and non-vector data. The term entity is used to describe data within the system. Each entity in Milvus is defined as one or more vectors, along with optionally some numerical attributes (non-vector data). For instance, in an image search application, the numerical attributes could represent characteristics like age and height of a person, in addition to multiple machine-learned feature vectors derived from their photos (e.g., describing different angles like front-face, side-face, or posture). Currently, Milvus supports numerical attributes, which have been commonly observed in various applications.

Query types. Milvus supports three primitive query types:

- Vector Query: This query type performs traditional vector similarity search [8], [26], [33], [34], where each entity is represented by a single vector. The system returns the k most similar vectors, where k is a user-defined parameter.
- Attribute Filtering: Each entity is defined by a single vector and associated attributes [46]. The system returns the k most similar vectors while adhering to the attribute constraints. For example, in recommend systems, users might want to find clothes similar to a given query image, while ensuring the price is below 100.
- Multi-vector Query: Each entity is represented by multiple vectors [5]. The query returns the top-k most similar entities, determined by an aggregation function (e.g., weighted sum) applied to the multiple vectors of each entity.

Similarity functions. Milvus offers commonly used similarity metrics, including Euclidean distance, inner product, cosine similarity, Hamming distance, and Jaccard distance, allowing applications to explore vector similarity in the most effective approach.

Application interfaces. Milvus provides easy-to-use SDK (software development kit) interfaces that can be directly called in applications written in various languages including Python, Java, Go, and C++. Milvus also supports RESTful APIs for web applications.

2) Indexing: In Milvus, we support two primary types of indexes to optimize vector search for different applications: quantization-based indexes, such as IVF_FLAT [11], [26], [28], IVF_SQ8 [11], [28], and IVF_PQ [11], [16], [26], [28], as well as graph-based indexes, including HNSW [34] and RNSG [14]. The design decisions behind these indexes are influenced by several factors, such as the latest literature reviews [31], industrial-strength systems (e.g., Alibaba PASE [48], Alibaba AnalyticDB-V [46], Jingdong Vearch [30]), open-source libraries (e.g., Facebook Faiss [11], [28]), and input from customers. LSH-based approaches are excluded due to their lower accuracy compared to quantization-based methods, particularly for billion-scale datasets [46], [48].

B. Dynamic Data Management

Milvus supports efficient insertions and deletions by adopting the concept of an LSM-tree [32]. Newly inserted entities are initially stored in memory as a MemTable. Once the MemTable reaches a predefined threshold or after a set interval (e.g., one second), it becomes immutable and is flushed to disk

as a new segment. Smaller segments are merged into larger ones to facilitate fast sequential access. Milvus implements a tiered merge policy, similar to Apache Lucene, which merges segments of roughly equal sizes until a configurable size limit (e.g., 1GB) is reached. Deletions are handled using the same out-of-place approach, with obsolete vectors removed during the segment merge process. Updates are achieved through deletions and insertions. By default, Milvus builds indexes only for large segments (e.g., $>$ 1GB), though users can manually trigger index creation for smaller segments if needed. Both data and indexes are stored in the same segment, making the segment the fundamental unit for searching, scheduling, and buffering.

C. Storage Management

Vector storage. For single-vector entities, Milvus stores all vectors continuously, without explicitly storing row IDs. The vectors are sorted by their row IDs, so given a row ID, Milvus can directly access the corresponding vector, as all vectors are of uniform length. For multi-vector entities, Milvus stores vectors from different entities in a columnar format. For example, assuming that there are three entities (A, B, and C) in the database and each entity has two vectors v_1 and v_2, then all the v_1 of diFFerent entities are stored together and all the v_2 are stored together. That is, the storage format is $A.v_1, B.v_1, C.v_1, A.v_2, B.v_2, C.v_2$.

Attribute storage. The attributes are stored column by column. In particular, each attribute column is stored as an array of $< key, value >$ pairs where the key is the attribute value and $value$ is the row ID, sorted by the key. Additionally, Milvus builds skip pointers (i.e., minimum and maximum values) following the Snowflake indexing scheme [9] for data pages stored on disk. This indexing mechanism enables efficient point and range queries on the attribute columns. For example, it allows fast queries such as "price is less than $100" by quickly narrowing down the relevant data pages that meet the specified condition.

Bufferpool. Milvus assumes that the majority, if not all, of the data and indexes are kept in memory to ensure high performance. When this is not the case, it uses an LRU (Least Recently Used)-based buffer manager to manage memory effectively and ensure data is efficiently loaded and accessed from disk when needed.

Multi-storage. For flexibility and reliability, Milvus supports multiple file systems, including local file systems, Amazon S3, and HDFS, for underlying data storage. This versatility enhances the system's adaptability and makes it easier to deploy Milvus in cloud environments, enabling scalable and efficient data management.

D. HETEROGENEOUS COMPUTING

In this subsection, we present the optimizations implemented in Milvus to efficiently leverage heterogeneous computing platforms that involve both CPUs and GPUs.

1) CPU-oriented Optimizations: The fundamental problem for query processing over quantization based indexes is that, given a collection of queries $\{q_1, q_2, ..., q_m\}$ and a collection of = data vectors $\{v_1, v_2, ..., v_n\}$, how to quickly find for each query q_i its top-k similar vectors? In practice, users can submit batch queries so that $m \geq 1$.

Milvus addresses these issues with two key strategies. First, it reuses data vectors as much as possible across multiple queries to minimize CPU cache misses. This optimization focuses on reducing L3 cache misses, as accessing memory incurs a significant penalty, and the L3 cache is much larger than L1/L2 caches, offering more room for optimization. Second, Milvus employs fine-grained parallelism by assigning threads to data vectors rather than query vectors, which helps maximize multi-core parallelism, especially since the data size is typically much larger than the query size in practice.

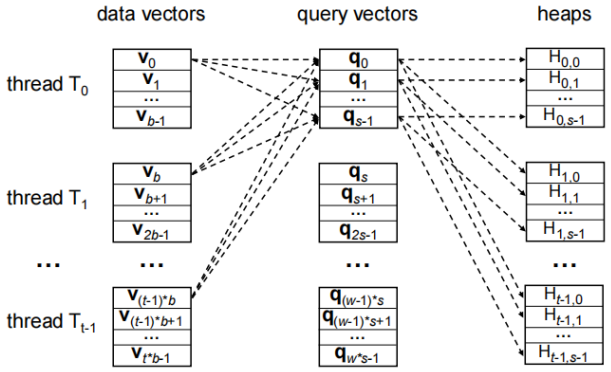


Fig. 17. Milvus Cache Aware Design

E. Experiments

1) Experimental Setup: All experiments are conducted on Alibaba Cloud, utilizing various types of computing instances (up to 12 nodes) to optimize for cost efficiency. By default, we use the CPU instance ecs.g6e.4xlarge, which is equipped with a Xeon Platinum 8269 Cascade 2.5GHz processor, 16 vCPUs, 35.75MB L3 cache, AVX512, 64GB of memory, and NAS elastic storage. For GPU-intensive tasks, we utilize the ecs.gn6ic16g1.4xlarge instance, featuring an NVIDIA Tesla T4 GPU with 64KB private memory, 512KB local memory, 16GB global memory, and a PCIe 3.0 16x interface.

Datasets. To ensure reproducibility, we use two publicly available datasets to evaluate Milvus: SIFT1B [27] and Deep1B [4]. SIFT1B consists of 1 billion 128-dimensional SIFT vectors (512GB), while Deep1B contains 1 billion 96-dimensional image vectors (384GB) extracted from a deep neural network. Both datasets are widely used in previous studies on vector similarity search and approximate nearest neighbor search [35, 41, 65, 68].

Competitors. We compare Milvus against two open-source systems: Jingdong Vearch (v3.2.0) [30] and Microsoft SPTAG [7]. We also compare Milvus with three industrial-strength commercial systems (with latest version as of July 2020)

anonymized as System A, B, and C for commercial reasons. Since Milvus is implemented on top of Faiss, we also present the performance comparison by evaluating the algorithmic optimizations in Milvus.

Evaluation metrics. We use the recall to evaluate the accuracy of the top-k results returned by a system where k is 50 by default.

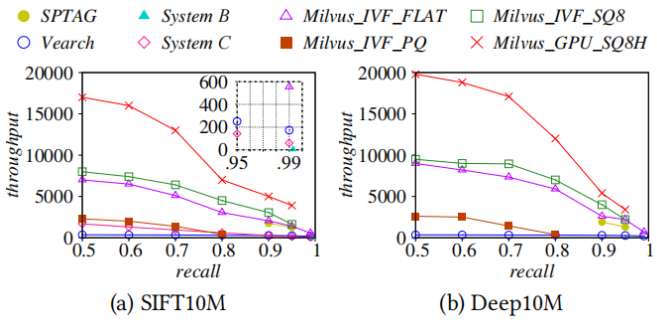


Fig. 18. System evaluation on IVF indexes

2) Comparing with Prior Systems: In this experiment, we compare Milvus with prior systems in terms of recall and throughput using the first 10 million vectors from the SIFT1B and Deep1B datasets (SIFT10M and Deep10M).

All systems are run on a single node, except for Systems A, B, and C, which require multiple nodes. Milvus and the other systems use two types of indexes: IVF_FLAT and HNSW, both of which are supported by most systems.

The results shown in Figure 18 for IVF indexes reveal that Milvus (even the CPU version) significantly outperforms other systems, with speed improvements of up to two orders of magnitude while maintaining similar recall. Milvus is 6.4 to 27.0 times faster than Vearch, 153.7 times faster than System B (even when System B uses four nodes), and 1.3 to 2.1 times faster than SPTAG. However, SPTAG requires 14 times more memory than Milvus and cannot achieve the same level of recall. The GPU version of Milvus performs even better as it can fully utilize the GPU memory.

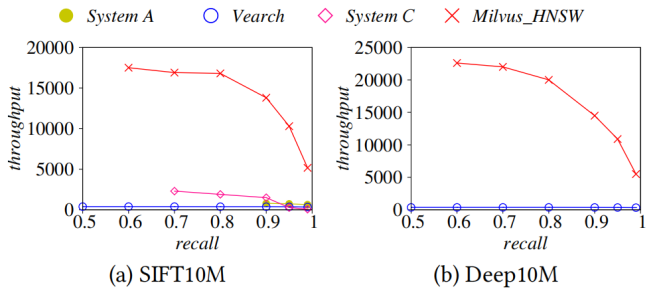


Fig. 19. System evaluation on HNSW indexes

Figure 19 presents the results for the HNSW index across different systems. Milvus significantly outperforms the existing systems, achieving performance improvements ranging from 15.1 to 60.4 times faster than Vearch, 8.0 to 17.1 times faster than System A, and 7.3 to 73.9 times faster than System C. We exclude the results for System A on Deep10M as it does not support the inner product metric. Additionally, the results for System C on Deep10M are omitted because the index building process fails to complete even after more than 100 hours.

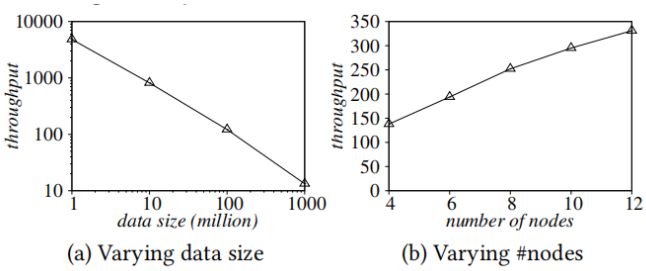


Fig. 20. Scalability

3) Scalability: Figure 20(a) displays the results for a single node of ecs.re6.26xlarge (104 vCPUs and 1.5TB memory), where the entire data cannot fit into memory. As the dataset size increases, the throughput decreases proportionally. Figure 20(b) illustrates the scalability of distributed Milvus, with data sharded across multiple nodes, each of which is an ecs.g6e.13xlarge instance (52 vCPUs and 192GB memory). As the number of nodes increases, the throughput scales linearly. Notably, Milvus achieves higher throughput on the ecs.g6e.13xlarge instances compared to the ecs.re6.26xlarge instance, which can be attributed to reduced competition for shared CPU caches and memory bandwidth across more cores.

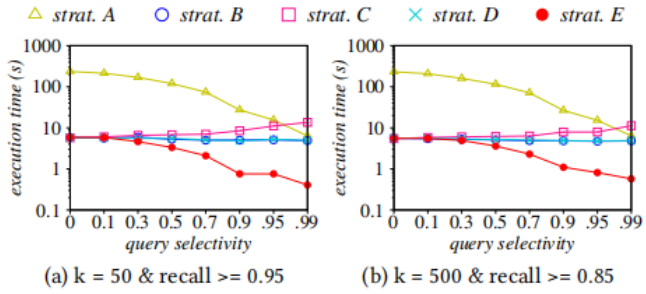


Fig. 21. Attribute filtering in Milvus

4) Evaluating Attribute Filtering: We define query selectivity as the percentage of entities that fail the attribute constraint C_A, following the method in [46].

Figure 21 shows the performance results with varying query selectivity. Strategy A improves as selectivity increases because fewer vectors need to be examined. Strategy B, however, is insensitive to selectivity, as its bottleneck lies in vector similarity search. Strategy C is slower than Strategy B, as it requires checking 1.1 times the number of vectors. Strategy D, which uses a cost-based approach to select the best between A, B, and C, outperforms all other strategies. Our new approach,

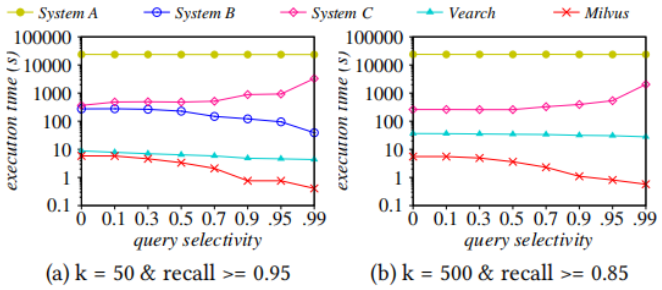


Fig. 22. Attribute filtering in Milvus

Strategy E, significantly outperforms Strategy D by up to 13.7× due to partitioning.

Figure 22 compares Milvus against Systems A, B, C, and Vearch in terms of attribute filtering, showing that Milvus outperforms these systems by a factor of 48.5× to 41,299.5×. Note that the results for System B are omitted in Figure 15b because its parameters are fixed by the system and cannot be changed by the user.

VII. Conclusion

Vector database systems are becoming increasingly important in the realm of artificial intelligence and data science due to their ability to efficiently manage and search high-dimensional vector data. This paper introduces ADBV (Attribute-Driven Vector Search) and Milvus, two key contributions to the field of vector similarity search.

Faiss, crafted by Facebook AI Research, is a powerful open-source library. It specializes in efficient similarity search and vector indexing, streamlining nearest neighbor searches crucial in ML and AI tasks.

With ADBV, practitioners can manage massive high-dimensional vectors and structured attributes within a single system. The proposed VGPQ algorithm further enhances hybrid query processing performance on large volumes of baseline data. Additionally, hybrid queries are natively optimized through the design of accuracy-aware cost-based optimization. ADBV has been successfully deployed in Alibaba Group and on Alibaba Cloud, supporting various complex business scenarios.

Milvus is a high-performance, scalable vector database designed to handle billion-scale datasets with both CPU and GPU optimizations. Through extensive experiments, we demonstrated that Milvus significantly outperforms existing systems in terms of recall, throughput, and scalability, thanks to its advanced engineering optimizations such as fine-grained parallelism, cache-aware strategies, and hybrid execution between GPUs and CPUs.

We anticipate that the insights and advancements they bring will fuel further research and development, spurring the creation of more intelligent and efficient data management systems.

Acknowledgment

I'm sincerely grateful to the instructor and teaching assistant of this course. Their dedication, patient guidance, and timely feedback have been invaluable. Whenever I hit a snag, they helped clarify concepts and inspire me, making this course rewarding.

References

[1] Greenplum. https://greenplum.org/.

[2] Yassine Afoudi, Mohamed Lazaar, and Mohammed Al Achhab. Hybrid recommendation system combined content-based filtering and collaborative prediction using artificial neural network. *Simulation Modelling Practice and Theory*, 113:102375, 2021.

[3] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[4] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

[5] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018.

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

[7] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. Sptag: A library for fast approximate nearest neighbor search, 2018.

[8] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, et al. Approximate nearest neighbor search under neural similarity metric for large-scale recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 3013–3022, 2022.

[9] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

[10] Korry Douglas and Susan Douglas. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing, 2003.

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[12] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

[13] BV Elasticsearch. Elasticsearch. *software], version*, 6(1), 2018.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[15] Jianyang Gao and Cheng Long. Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

[16] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):744–755, 2013.

[17] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[18] Rishabh Goel. Using text embedding models and vector databases as text classifiers with the example of medical data. *arXiv preprint arXiv:2402.16886*, 2024.

[19] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, pages 3406–3416, 2023.

[20] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pages 3887–3896. PMLR, 2020.

[21] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. Lightrag: Simple and fast retrieval-augmented generation. *arXiv preprint arXiv:2410.05779*, 2024.

[22] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.

[23] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[24] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

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

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

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

[28] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[29] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173*, 2012.

[30] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. The design and implementation of a real time visual search system on jd e-commerce platform. In *Proceedings of the 19th International Middleware Conference Industry*, pages 9–16, 2018.

[31] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[32] Chen Luo and Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[33] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Intelligent probing for locality sensitive hashing: Multi-probe lsh and beyond. *Proceedings of the VLDB Endowment*, 2017.

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

[35] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[36] DB Mongo. Mongodb, 2015.

[37] Cun Mu, Jun Zhao, Guang Yang, Jing Zhang, and Zheng Yan. Towards practical visual search engine within elasticsearch. *arXiv preprint arXiv:1806.08896*, 2018.

[38] Stephen M Omohundro. Five balltree construction algorithms. 1989.

[39] James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. *The VLDB Journal*, 33(5):1591–1615, 2024.

[40] John T Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, 1981.

[41] Kazunori Sato. An inside look at google bigquery. *White paper, URL: https://cloud. google. com/files/BigQueryTechnicalWP. pdf*, 2012.

[42] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[43] Michael Stonebraker and Uĝur Çetintemel. " one size fits all" an idea whose time has come and gone. In *Making databases work: the pragmatic wisdom of Michael Stonebraker*, pages 441–462. 2018.

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

[45] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

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

[47] Hang Yang, Jing Guo, Jianchuan Qi, Jinliang Xie, Si Zhang, Siqi Yang, Nan Li, and Ming Xu. A method for parsing and vectorization of semi-structured data used in retrieval augmented generation. *arXiv preprint arXiv:2405.03989*, 2024.

[48] Wen Yang, Tao Li, Gai Fang, and Hong Wei. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 2241–2253, 2020.

[49] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. Analyticdb: real-time olap database system at alibaba cloud. *Proceedings of the VLDB Endowment*, 12(12):2059–2070, 2019.

[50] Minjia Zhang and Yuxiong He. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1673–1682, 2019.

[51] Wengang Zhou, Yijuan Lu, Houqiang Li, and Qi Tian. Scalar quantization for large scale image search. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 169–178, 2012.