Milvus: A Purpose-Built Vector Data Management System

Jianguo Wang*, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, Charles Xie

* Zilliz & Purdue University
* *csjgwang@{zilliz.com; purdue.edu}*

Zilliz
{firstname.lastname}@zilliz.com

ABSTRACT

Recently, there has been a pressing need to manage high-dimensional vector data in data science and AI applications. This trend is fueled by the proliferation of unstructured data and machine learning (ML), where ML models usually transform unstructured data into feature vectors for data analytics, e.g., product recommendation. Existing systems and algorithms for managing vector data have two limitations: (1) They incur serious performance issue when handling large-scale and dynamic vector data; and (2) They provide limited functionalities that cannot meet the requirements of versatile applications.

This paper presents Milvus, a purpose-built data management system to efficiently manage large-scale vector data. Milvus supports easy-to-use application interfaces (including SDKs and RESTful APIs); optimizes for the heterogeneous computing platform with modern CPUs and GPUs; enables advanced query processing beyond simple vector similarity search; handles dynamic data for fast updates while ensuring efficient query processing; and distributes data across multiple nodes to achieve scalability and availability. We first describe the design and implementation of Milvus. Then we demonstrate the real-world use cases supported by Milvus. In particular, we build a series of **10** applications (e.g., image/video search, chemical structure analysis, COVID-19 dataset search, personalized recommendation, biological multi-factor authentication, intelligent question answering) on top of Milvus. Finally, we experimentally evaluate Milvus with a wide range of systems including two open source systems (Vearch and Microsoft SPTAG) and three commercial systems. Experiments show that Milvus is up to two orders of magnitude faster than the competitors while providing more functionalities. Now Milvus is deployed by hundreds of organizations worldwide and it is also recognized as an incubation-stage project of the LF AI & Data Foundation. Milvus is open-sourced at *https://github.com/milvus-io/milvus*.

CCS CONCEPTS

• **Information systems** → **Database management system engines**; **Data access methods**;

KEYWORDS

Vector database; High-dimensional similarity search; Heterogeneous computing; Data science; Machine learning

ACM Reference Format:
Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457550

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457550

1 INTRODUCTION

At Zilliz, we have experienced a growing need from various customers to manage large-scale high-dimensional vector data (ranging from 10s to 1000s of dimensions) in many data science and AI applications. This is largely due to two trends. The first one is an explosive growth of unstructured data such as images, videos, texts, medical data, and housing data due to the prevalence of smartphones, IoT devices, and social media apps. According to IDC, 80% of data will be unstructured by 2025 [36]. The second trend is the rapid development of machine learning that can effectively transform unstructured data into learned feature vectors for data analytics. In particular, a recent popular approach in recommender systems is called *vector embedding* that converts an item to a feature vector (such as item2vec [11], word2vec [52], doc2vec [37], graph2vec [26]) and provides recommendations via finding similar vectors [13, 15, 25, 51]. For example, YouTube embeds videos to vectors [15]; Airbnb models houses with vectors [25]; Bioscientists describe the molecular structural information of drug compounds using vectors [13, 51]. Besides that, images and texts are also naturally represented by vectors [8, 53].

Those applications present unique requirements and challenges for designing a scalable vector data management system. These include: (1) The need to support not only fast query processing on large-scale vector data but also the efficient handling of dynamic vector data (such as insertions and deletions). As an example, Youtube uploads 500 hours of user-generated videos per minute and meanwhile offers real-time recommendations [67]. (2) The need to provide advanced query processing such as attribute filtering [65] and multi-vector query processing [10] beyond simple vector similarity search. Here attribute filtering is to only search vectors that satisfy a given filtering condition, which is useful in e-commerce applications [65], e.g., finding the T-shirts similar to a given image vector that also cost less than $100. And multi-vector query processing targets for the scenario where each object is described by multiple vectors, e.g., profiling a person using a face vector and a posture vector in many computer vision applications [10, 56].

Table 1: System comparison

	Billion-Scale Data	Dynamic Data	GPU	Attribute Filtering	Multi-Vector Query	Distributed System
Facebook Faiss [3, 35]	✓	✗	✓	✗	✗	✗
Microsoft SPTAG [14]	✓	✗	✗	✗	✗	✗
ElasticSearch [2]	✗	✓	✗	✓	✗	✓
Jingdong Vearch [4, 39]	✗	✓	✓	✓	✗	✓
Alibaba AnalyticDB-V [65]	✓	✓	✗	✓	✗	✓
Alibaba PASE (PostgreSQL) [68]	✗	✓	✗	✓	✗	✗
Milvus (this paper)	✓	✓	✓	✓	✓	✓

Existing works on vector data management mainly focus on vector similarity search [14, 20, 22, 33, 35, 39, 45, 46, 48, 49, 57, 65, 68], but they cannot meet the above requirements due to poor performance (on large-scale and dynamic vector data) and limited functionalities (e.g., not being capable of supporting attributefiltering and multi-vector queries) to support versatile data science and AI applications.

More specifically, we classify existing works into two categories: *algorithms* and *systems*. For the algorithmic works on vector similarity search, e.g., [20, 22, 33, 45, 46, 48, 49, 57], together with their open-source implementation libraries (exemplified by Facebook Faiss [35] and Microsoft SPTAG [14]), there are several limitations. (1) They are algorithms and libraries, not a full-fledged system that manages vector data. They cannot handle large amount of data very well since they assume that all the data and index are stored in main memory and cannot span multiple machines. (2) Those works usually assume data to be static once ingested into the system and cannot easily handle dynamic data while ensuring fast real-time searches. (3) They do not support advanced query processing. (4) Those works are not optimized for the heterogeneous computing architecture with CPUs and GPUs.

For the system works on vector similarity search, e.g., Alibaba AnalyticDB-V [65] and Alibaba PASE (PostgreSQL) [68], they follow the one-size-fits-all approach to extend relational databases for supporting vector data by adding a table column called "vector column" to store vectors. However, those systems are not specialized for managing vector data and they do not treat vectors asfirst-class citizens. (1) Legacy database components such as optimizer and storage engine preventfi ne-tuned optimizations for vectors, e.g., the query optimizer misses significant opportunity to best leverage CPU and GPU for processing vector data. (2) They do not support advanced query processing such as multi-vector queries.

Another relevant system is Vearch [4, 39], which is designed for vector search. But Vearch is not efficient on large-scale data. Experiments (Figure 8 and Figure 15) show that Milvus, the system introduced in this paper, is 6.4× ∼ 47.0× faster than Vearch. Also, Vearch does not support multi-vector query processing.

This paper presents Milvus, a purpose-built data management system to efficiently store and search large-scale vector data for data science and AI applications. It is a specialized system for high-dimensional vectors following the design practice of one-size-not-fits-all [60] in contrast to generalizing relational databases to support vectors. Milvus provides many application interfaces (including SDKs in Python/Java/Go/C++ and RESTful APIs) that can be easily used by applications. Milvus is highly tuned for the heterogeneous computing architecture with modern CPUs and GPUs (multiple GPU devices) for the best efficiency. It supports versatile query types such as vector similarity search with various similarity functions, attributefi ltering, and multi-vector query processing. It provides different types of indexes (e.g., quantization-based indexes [33, 35] and graph-based indexes [20, 49]) and develops an extensible interface to easily incorporate new indexes into the system. Milvus manages dynamic vector data (e.g., insertions and deletions) via an LSM-based structure while providing consistent real-time searches with snapshot isolation. Milvus is also a distributed data management system deployed across multiple nodes to achieve scalability and availability. Table 1 highlights the main differences between Milvus and other systems.

In terms of implementation, Milvus is built on top of Facebook Faiss [3, 35], an open-source C++ library for vector similarity search. But Milvus significantly enhances Faiss with improved performance (e.g., optimizing for the heterogeneous computing platform in Sec. 3, supporting dynamic data management efficiently in Sec. 2.3 and distributed query processing in Sec. 5.3), enhanced functionalities (e.g., attributefi ltering and multi-vector query processing in Sec. 4), and better usability (e.g., application interfaces in Sec. 2.1) to be a full-fledged easy-to-use vector data management system.

Product impact. Milvus is adopted by hundreds of organizations and institutions worldwide in variousfi elds such as image processing, computer vision, natural language processing, voice recognition, recommender systems, and drug discovery. More importantly, Milvus was accepted as an incubation-stage project of the LF AI & Data Foundation in January 2020.[1]

Contributions. This paper makes the following contributions:

- **System design and implementation** (Sec. 2 and Sec. 5): The overall contribution is the design and implementation of Milvus, a purpose-built vector data management system for managing large-scale and dynamic vector data to enable data science and AI applications. Milvus is open-sourced at *https://github.com/milvus-io/milvus*.
- **Heterogeneous computing** (Sec. 3): We optimize Milvus for the heterogeneous hardware platform with modern CPUs and GPUs for fast query processing. For CPU-oriented design, we propose both cache-aware and SIMD-aware (e.g., SSE, AVX, AVX2, AVX512) optimizations. For GPU-oriented design, we design a new hybrid index that takes advantages of the best of CPU and GPU, and we also develop a new scheduling strategy to support multiple GPU devices.
- **Advanced query processing** (Sec. 4): We support attribute filtering and multi-vector query processing beyond simple vector similarity search in Milvus. In particular, we design a new partition-based algorithm for attributefi ltering and two algorithms (vector fusion and iterative merging) for multi-vector query processing.

[1] https://lfaidata.foundation/projects/milvus

- **Novel applications** (Sec. 6): We describe novel applications powered by Milvus. In particular, we build a series of **10** applications[2] on top of Milvus to demonstrate its broad applicability including image search, video search, chemical structure analysis, COVID-19 dataset search, personalized recommendation, biological multi-factor authentication, intelligent question answering, image-text retrieval, cross-modal pedestrian search, and recipe-food search.

2 SYSTEM DESIGN

In this section, we present an overview of Milvus. Figure 1 shows the architecture of Milvus with three major components: query engine, GPU engine, and storage engine. The query engine supports efficient query processing over vector data and it is optimized for modern CPUs by reducing cache misses and leveraging SIMD instructions. The GPU engine is a co-processing engine that accelerates performance with vast parallelism. It also supports multiple GPU devices for efficiency. The storage engine enables data durability and incorporates an LSM-based structure for dynamic data management. It runs on variousfi le systems (including localfile systems, Amazon S3, and HDFS) with a bufferpool in memory.

2.1 Query Processing

Wefi rst present the concept of entity used in Milvus and then explain query types, similarity functions, and application interfaces.

Entity. To best capture versatile data science and AI applications, Milvus supports query processing over both vector data and non-vector data. We define the term *entity* as follows to incorporate the two. Each entity in Milvus is described as one or more vectors and optionally some numerical attributes (non-vector data). For example, in the image search application, the numerical attributes can represent the age and height of a person in addition to possibly multiple machine-learned feature vectors of his/her photos (e.g., describing front-face, side-face, or posture [10]). In the current version of Milvus, we only support numerical attributes as observed from many applications. But in the future, we plan to support categorical attributes with indexes like inverted lists or bitmaps [64].

Query types. Milvus supports three primitive query types:

- Vector query: This query type is the traditional vector similarity search [33, 41, 48, 49], where each entity is described as a single vector. The system returns k most similar vectors where k is a user-input parameter.
- Attributefi ltering: Each entity is specified by a single vector and some attributes [65]. The system returns k most similar vectors while adhering to the attributes constraints. As an example in recommender systems, users want tofi nd similar clothes to a given query image while the price is below $100.
- Multi-vector query: Each entity is stored as multiple vectors [10]. The query returns top-k similar entities according to an aggregation function (e.g., weighted sum) between multiple vectors.

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.

[2]https://github.com/milvus-io/bootcamp/tree/master/EN_solutions

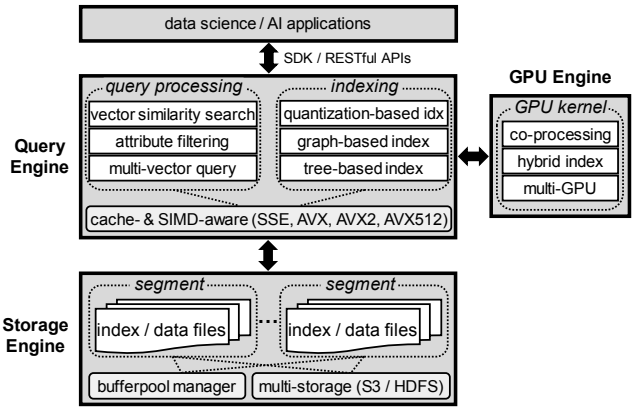


Figure 1: System architecture of Milvus

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.

2.2 Indexing

Indexing is of tremendous importance to query processing in Milvus. But a challenging issue we face is to decide which indexes to support in Milvus, because there are numerous indexes developed for vector similarity search. The latest benchmark [41] shows that there is no winner in all scenarios and each index comes with tradeoffs in performance, accuracy, and space overhead.

In Milvus, we mainly support two types of indexes:[3] quantization-based indexes (including IVF_FLAT [3, 33, 35], IVF_SQ8 [3, 35], and IVF_PQ [3, 22, 33, 35]) and graph-based indexes (including HNSW [49] and RNSG [20]) to serve different applications. The design decision is based on factors including the latest literature review [41], industrial-strength systems (e.g., Alibaba PASE [68], Alibaba AnalyticDB-V [65], Jingdong Vearch [39]), open-source libraries (e.g., Facebook Faiss [3, 35]), and inputs from customers. We exclude LSH-based approaches because they have lower accuracy than quantization-based approaches on billion-scale data [65, 68].

Considering there are many new indexes coming out every year, Milvus is designed to easily incorporate the new indexes with a high-level abstraction. Developers only need to implement a few pre-defined interfaces for adding a new index. Our hope is that Milvus can eventually become a standard platform for vector data management with versatile indexes.

2.3 Dynamic Data Management

Milvus supports efficient insertions and deletions by adopting the idea of LSM-tree [47]. Newly inserted entities are stored in memory first as MemTable. Once the accumulated size reaches a threshold, or once every second, the MemTable becomes immutable and then getsfl ushed to disk as a new segment. Smaller segments are merged into larger ones for fast sequential access. Milvus implements a tiered merge policy (also used in Apache Lucene) that aims to merge segments of approximately equal sizes until a configurable size limit (e.g., 1GB) is reached. Deletions are supported in the same out-of-place approach except that the obsoleted vectors are

[3]Milvus also supports tree-based indexes, e.g., ANNOY [1].

removed during segment merge. Updates are supported by deletions and insertions. By default, Milvus builds indexes only for large segments (e.g., > 1GB) but users are allowed to manually build indexes for segments of any size if necessary. Both index and data are stored in the same segment. Thus, the segment is the basic unit of searching, scheduling, and buffering.

Milvus offers snapshot isolation to make sure reads and writes share a consistent view and do not interfere with each other. We present the details of snapshot isolation in Sec. 5.2.

2.4 Storage Management

As mentioned in Sec. 2.1, each entity is expressed as one or more vectors and optionally some attributes. Thus, each entity can be regarded as a row in an entity table. To facilitate query processing, Milvus physically stores the entity table in a columnar fashion.

Vector storage. For single-vector entities, Milvus stores all the vectors continuously without explicitly storing the row IDs. In this way, all the vectors are sorted by row IDs. Given a row ID, Milvus can directly access the corresponding vector since each vector is of the same length. For multi-vector entities, Milvus stores the vectors of different entities in a columnar fashion. For example, assuming that there are three entities (A, B, and C) in the database and each entity has two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, then all the $\mathbf{v}_1$ of different entities are stored together and all the $\mathbf{v}_2$ are stored together. That is, the storage format is $\{A.\mathbf{v}_1, B.\mathbf{v}_1, C.\mathbf{v}_1, A.\mathbf{v}_2, B.\mathbf{v}_2, C.\mathbf{v}_2\}$.

Attribute storage. The attributes are stored column by column. In particular, each attribute column is stored as an array of $\langle key, value \rangle$ pairs where the *key* is the attribute value and *value* is the row ID, sorted by the *key*. Besides that, we build skip pointers (i.e., min/max values) following Snowflake [16] as indexing for the data pages on disk. This allows efficient point query and range query in that column, e.g., price is less than $100.

Bufferpool. Milvus assumes that most (if not all) data and index are resident in memory for high performance. If not, it relies on an LRU-based buffer manager. In particular, the caching unit is a segment, which is the basic searching unit as explained in Sec. 2.3.

Multi-storage. Forfl exibility and reliability, Milvus supports multiplefi le systems including localfi le systems, Amazon S3, and HDFS for the underlying data storage. This also facilitates the deployment of Milvus in the cloud.

2.5 Heterogeneous Computing

Milvus is highly optimized for the heterogeneous computing platform that includes CPUs and GPUs. Sec. 3 presents the details.

2.6 Distributed System

Milvus can function as a distributed system deployed across multiple nodes. It adopts modern design practices in distributed systems and cloud systems such as storage/compute separation, shared storage, read/write separation, and single-writer-multi-reader. Sec. 5.3 explains more.

3 HETEROGENEOUS COMPUTING

In this section, we present the optimizations for Milvus to best leverage the heterogeneous computing platform involving both CPUs and GPUs to achieve high performance.

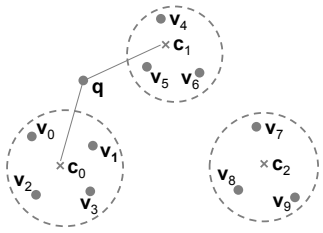


Figure 2: An example of quantization

As explained in Sec. 2.2, Milvus mainly supports quantization-based indexes (including IVF_FLAT [3, 33, 35], IVF_SQ8 [3, 35], and IVF_PQ [3, 22, 33, 35]) and graph-based indexes (including HNSW [49] and RNSG [20]). In this section, we use quantization-based indexes to illustrate our optimizations because they consume much less memory and are much faster to build index while achieving decent query performance when compared to graph-based indexes [65, 68]. Note that many optimizations (such as SIMD and GPU optimizations) can be applied to graph-based indexes.

3.1 Background

Before diving into optimizations, we explain vector quantization and quantization-based indexes. The main idea of vector quantization is to apply a quantizer z to map a vector $\mathbf{v}$ to a codeword $z(\mathbf{v})$ chosen from a codebook C [33]. The K-means clustering algorithm is commonly used to construct the codebook C where each codeword is the centroid and $z(\mathbf{v})$ is the closest centroid to $\mathbf{v}$. Figure 2 shows an example of 10 vectors ($\mathbf{v}_0$ to $\mathbf{v}_9$) of three clusters with centroids being $\mathbf{c}_0$ to $\mathbf{c}_2$, then $z(\mathbf{v}_0)$, $z(\mathbf{v}_1)$, $z(\mathbf{v}_2)$, or $z(\mathbf{v}_3)$ is $\mathbf{c}_0$.

Quantization-based indexes (such as IVF_FLAT [3, 33, 35], IVF_SQ8 [3, 35], and IVF_PQ [3, 22, 33, 35]) use two quantizers: coarse quantizer andfi ne quantizer. The coarse quantizer applies the K-means algorithm (e.g., K is 16384 in Milvus and Faiss [3]) to cluster vectors into K buckets. And thefi ne quantizer encodes the vectors within each bucket. Different indexes may use differentfi ne quantizers. IVF_FLAT uses the original vector representation; IVF_SQ8 uses a compressed representation for the vectors by adopting one-dimensional quantizer (called "scalar quantizer") to compress a 4-bytefl oat value to a 1-byte integer; and IVF_PQ uses product quantization that splits each vector into multiple sub-vectors and applies K-means for each sub-space.

Query processing (of a query $\mathbf{q}$) over quantization-based indexes takes two steps: (1) Find the closest n_{probe} buckets (or clusters) based on the distance between $\mathbf{q}$ and the centroid of each bucket. For example, assuming n_{probe} is 2 in Figure 2, then the closest two buckets of $\mathbf{q}$ are centered at $\mathbf{c}_0$ and $\mathbf{c}_1$. The parameter n_{probe} controls the tradeoff between accuracy and performance. Higher n_{probe} produces better accuracy but worse performance. (2) Search within each of the n_{probe} relevant buckets based on differentfine quantizers. For example, if the index in Figure 2 is IVF_FLAT, then it needs to scan the vectors $\mathbf{v}_0$ to $\mathbf{v}_6$ in the two buckets.

3.2 CPU-oriented Optimizations

3.2.1 Cache-aware Optimizations in Milvus

The fundamental problem for query processing over quantization-based indexes is that, given a collection of m queries $\{\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_m\}$ and a collection of n data vectors $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n\}$, how to quickly

find for each query $\mathbf{q}_i$ its top-k similar vectors? In practice, users can submit batch queries so that $m \geq 1$.

This operation happens infi nding the relevant buckets as well as searching within each relevant bucket. The original implementation in Facebook Faiss [3], which Milvus is built on top of, is inefficient because it incurs many CPU cache misses as explained below. Thus, Milvus develops an optimized approach to significantly reduce data movement between main memory and CPU caches.

Original implementation in Facebook Faiss [3]. Faiss uses the OpenMP multi-threading to process queries in parallel. Each thread is assigned to work on a single query at a time. The thread is released (for next query) once the current task isfi nished. Each task compares $\mathbf{q}_i$ with all the n data vectors and maintains a k-sized heap to store the results.

The above solution in Faiss has two performance issues: (1) It incurs many CPU cache misses, because for each query the entire data needs to be streamed through CPU caches and cannot be reused for the next query. Thus, each thread accesses m/t times of the entire data where t is the total number of threads. (2) It cannot fully leverage multi-core parallelism when the batch size m is small.

Optimizations in Milvus. Milvus develops two ideas to tackle the issues. First, it reuses the accessed data vectors as much as possible for multiple queries to minimize CPU cache misses. Specifically, it optimizes for reducing L3 cache misses because the penalty to access memory is high and also L3 cache size (typically 10s MB) is much bigger than L1/L2 cache, leaving more room for optimizations. Second, it usesfi ne-grained parallelism that assigns threads to data vectors instead of query vectors to best leverage multi-core parallelism, because the data size n is usually much bigger than the query size m in practice.

Figure 3 shows the overall design. Specifically, let t be the number of threads, then each thread T_i is assigned $b = n/t$ data vectors:[4] $\{\mathbf{v}_{(i-1)*b}, \mathbf{v}_{(i-1)*b+1}, ..., \mathbf{v}_{i*b-1}\}$. Milvus then partitions the m queries into query blocks of size s such that each query block (together with its associated heaps) can alwaysfi t in the L3 CPU cache. We decide s later on in Equation (1). Here we assume that m is divisible by s. Milvus computes the top-k results of each query block at a time with multiple threads. Whenever each thread loads its assigned data vectors to L3 cache, they will be compared against the entire query block (with s queries) in the cache. To minimize the synchronization overhead, Milvus assigns a heap per query per thread. In particular, assuming the i-th query block $\{\mathbf{q}_{(i-1)*s}, \mathbf{q}_{(i-1)*s+1}, ..., \mathbf{q}_{i*s-1}\}$ is in cache, Milvus dedicates the heap $H_{r-1,j-1}$ for the j-th query $\mathbf{q}_{(i-1)*s+j-1}$ on the r-th thread T_{r-1}. Thus, the results of a query $\mathbf{q}_i$ are spread over t threads of heaps. Thus, it needs to merge the heaps of each thread to obtain thefi nal top-k results.

Next, we discuss how to determine the query block size s such that s queries and their associated heaps can alwaysfi t in L3 cache. Let d be the dimensionality, then the size of each query is $d \times$ sizeof(float). Since each heap entry contains a pair of vector ID and similarity, then the total size of the heaps (per query) is $t \times k \times$ (sizeof(int64) + sizeof(float)) where t is the number of threads. Thus, s is computed as follows:

$$s = \frac{\text{L3's cache size}}{d \times \text{sizeof(float)} + t \times k \times (\text{sizeof(int64)} + \text{sizeof(float)})}. \quad (1)$$

[4]We assume that n is divisible by t.

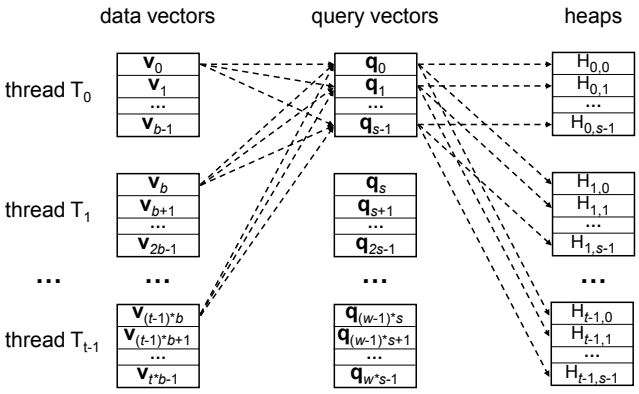


Figure 3: Cache-aware design in Milvus

In this way, each thread only accesses $m/(s*t)$ times of the entire data, which is s times smaller than the original implementation in Facebook Faiss [3]. Experiments (Sec. 7.4) show that this improves performance by a factor of 1.5× to 2.7×.

3.2.2 SIMD-aware Optimizations in Milvus

Modern CPUs support increasingly wider SIMD instructions. Thus, it is not surprising that Facebook Faiss [3] implements SIMD-aware algorithms to accelerate vector similarity search. We make two engineering optimizations in Milvus: (1) Supporting AVX512; and (2) Automatic SIMD-instruction selection.

Supporting AVX512. Faiss [3] does not support AVX512, which is now available in mainstream CPUs. Thus, we extend the similarity computing function with AVX512 instructions, such as `_mm512_add_ps`, `_mm512_mul_ps`, and `_mm512_extractf32x8_ps`. Now Milvus supports SIMD SSE, AVX, AVX2, and AVX512.

Automatic SIMD-instruction selection. Milvus is designed to work well on a wide spectrum of CPU processors (both on-premises and cloud platforms) with different SIMD instructions (e.g., SIMD SSE, AVX, AVX2, and AVX512). Thus the challenge is, given a single piece of software binary (i.e., Milvus), how to make it automatically invoke the suitable SIMD instructions on any CPU processor? Faiss [3] does not support it and users need to manually specify the SIMDfl ag (e.g., "-msse4") during compilation time. In Milvus, we take a considerable amount of engineering effort to refactor the codebase of Faiss. We factor out the common functions (e.g., similarity computing) that rely on SIMD accelerations. Then for each function, we implement four versions (i.e., SSE, AVX, AVX2, AVX512) and put each one into a separated sourcefi le, which is further compiled individually with the corresponding SIMDflag. During runtime, Milvus can automatically choose the suitable SIMD instructions based on the current CPUfl ags and then link the right function pointers using hooking.

3.3 GPU-oriented Optimizations

GPU is known for vast parallelism and Faiss [3] supports GPU for query processing over vector data. Milvus enhances Faiss in two aspects: (1) Supporting bigger k in the GPU kernel; (2) Supporting multi-GPU devices.

Supporting bigger k in GPU kernel. The original implementation in Faiss [3] does not support top-k query processing where k is greater than 1024 due to the limit of shared memory. But many

application such as video surveillance and recommender systems may need bigger k for further verification or re-ranking [69, 71].

Milvus overcomes this limitation and supports k up to 16384 although technically Milvus can support any k.[5] When k is larger than 1024, Milvus executes the query in multiple rounds to cumulatively produce thefi nal results. In thefi rst round, Milvus behaves the same as Faiss and gets the top 1024 results. For the second and later rounds, Milvusfi rst checks the distance of the last result (denoted as d_l) in the previous round. Apparently, d_l is so far the largest distance in the partial results. To handle vectors with equivalent distance to the query, Milvus also records vector IDs in the result whose distances are equal to d_l. Then Milvusfi lters out vectors whose distances are smaller than d_l or IDs are recorded. From the remaining data, Milvus gets the next 1024 results. By doing so, Milvus ensures that results in previous rounds will not appear in the current round. After that, the new results are merged with the partial results obtained in earlier rounds. Milvus processes the query in a round-by-round fashion until a sufficient number of results are collected.

Supporting multi-GPU devices. Faiss [3] supports multiple GPU devices since they are usually found in modern servers. But Faiss needs to declare all the GPU devices in advance during compilation time. That means if the Faiss codebase is compiled using a server with c GPUs, then the software binary can only be running in a server that has at least c GPUs.

Milvus overcomes this limitation by allowing users to select any number of GPU devices during *runtime* (instead of compilation time). As a result, once the Milvus codebase is compiled into a software binary, it can run at any server. Under the hood, Milvus introduces a segment-based scheduling that assigns segment-based search tasks to the available GPU devices. Each segment can only be served by a single GPU device. This is particularly a goodfi t for the cloud environment with dynamic resource management where GPU devices can be elastically added or removed. For example, if there is a new GPU device installed, Milvus can immediately discover it and assign the next available search task to it.

3.4 GPU and CPU Co-design

In this mode, the GPU memory is not large enough to store the entire data. Facebook Faiss [3] alleviates the problem by using a low-footprint compressed index (called IVF_SQ8 [3])[6] and moving data from CPU memory to GPU memory (via PCIe bus) on demand. However, wefi nd that there are two limitations: (1) The PCIe bandwidth is not fully utilized, e.g., our experiments show that the measured I/O bandwidth is only 1~2GB/s while PCIe 3.0 (16x) supports up to 15.75GB/s. (2) It is not always beneficial to execute queries on GPU (than CPU) considering the data transfer.

Milvus develops a new index called SQ8H (where 'H' stands for hybrid) to address the above limitations (Algorithm 1).

Addressing thefi rst limitation. We investigate the codebase of Faiss andfi gure out that Faiss copies data (from CPU to GPU) bucket by bucket, which underutilizes the PCIe bandwidth since each bucket can be small. So the natural idea is to copy multiple buckets simultaneously. But the downside of such multi-bucket-copying is the handling of deletions where Faiss uses a simple in-place update approach because each bucket is copied (and stored) individually. Fortunately, deletions (and updates) are easily handled in Milvus since Milvus adopts an efficient LSM-based out-of-place approach (Sec. 2.3). As a result, Milvus improves the I/O utilization by copying multiple buckets if possible (line 3 of Algorithm 1).

Algorithm 1: SQ8H

```
1 let n_q be the batch size;
2 if n_q ≥ threshold then
3     run all the queries entirely in GPU (load multiple buckets
      to GPU memory on thefl y);
4 else
5     execute the step 1 of SQ8 in GPU:finding n_probe buckets;
6     execute the step 2 of SQ8 in CPU: scanning every relevant
      bucket;
```

Addressing the second limitation. We observe that GPU outperforms CPU only if the query batch size is large enough considering the expensive data movement. That is because more queries make the workload more computation-intensive since they search the same data. Thus, if the batch size is bigger than a threshold (e.g., 1000), Milvus executes all the queries in GPU and loads necessary buckets if GPU memory is insufficient (line 2 of Algorithm 1). Otherwise, Milvus executes the query in a hybrid manner as follows. As mentioned in Sec. 3.1, there are two steps for searching quantization-based indexes:finding n_{probe} relevant (closest) buckets and scanning each relevant bucket. Milvus executes step 1 in GPU and step 2 in CPU because we observe that step 1 has a much higher computation-to-I/O ratio than step 2 (line 5 and 6 in Algorithm 1). That is because in step 1, all the queries compare against the *same* K centroids tofind n_{probe} nearest buckets, and also the K centroids are small enough to be resident in the GPU memory. By contrast, data accesses in step 2 are more scattered since different queries do not necessarily access the same buckets.

4 ADVANCED QUERY PROCESSING

4.1 Attribute Filtering

As mentioned in Sec. 2.1, attributefi ltering is a hybrid query type that involves both vector data and non-vector data [65]. It only searches vectors that satisfy the attributes constraints. It is crucial to many applications [65], e.g.,fi nding similar houses (vector data) whose sizes are within a specific range (non-vector data). For presentation purpose, we assume that each entity is associated with a single vector and a single attribute since it is straightforward to extend the algorithms to multiple attributes. We defer multi-vector query processing to Sec. 4.2.

Formally, each such query involves two conditions C_A and C_V where C_A specifies the attribute constraint and C_V is the normal vector query constraint that returns top-k similar vectors. Without loss of generality, C_A is represented in the form of $a >= p_1$ && $a <= p_2$ where a is the attribute (e.g., size, price) and p_1 and p_2 are two boundaries of a range condition (e.g., $p_1 = 100$ and $p_2 = 500$).

There are several approaches to solve attributefi ltering as recently studied in AnalyticDB-V [65]. In Milvus, we implement those

[5] In Milvus, we purposely limit k to 16384 to prevent large data movement over networks. Also, that number is sufficient for the applications we have seen so far.

[6] Note that IVF_SQ8 takes 1/4 the space of IVF_FLAT while losing only 1% recall. But the design principle and optimizations in Sec. 3.4 can be applicable to other quantization-based indexes such as IVF_FLAT and IVF_PQ.

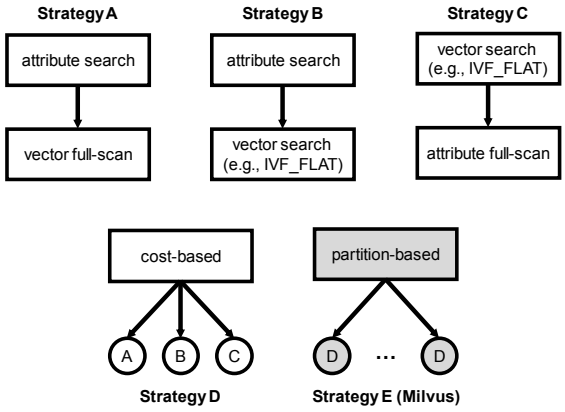


Figure 4: Different strategies for attributefiltering

approaches (i.e., strategies A, B, C, D as explained below). We then propose a partition-based approach (i.e., strategy E), which is up to 13.7× faster than the strategy D (i.e., state-of-the-art solution) according to the experiments in Sec. 7.5. Figure 4 shows a summary and we present the details next.

Strategy A: attribute-first-vector-full-scan. It only uses the attribute constraint C_A to obtain relevant entities via index search. Since the data is stored mostly in memory, we use binary search, but a B-tree index is also possible. When data cannotfi t in memory, we use skip pointers for fast search. After that, all the entities in the result set are fully scanned to compare against the query vector to produce thefi nal top-k results. Although simple, this approach is suitable when C_A is highly selective such that only a small number of candidates are required for further verification. Another interesting property of this strategy is that it produces the exact results.

Strategy B: attribute-first-vector-search. The difference with the strategy A is that after it obtains the relevant entities according to attribute constraint C_A, it produces a bitmap of the resultant entity IDs. Then it conducts the normal vector query processing based on C_V and checks the bitmap whenever a vector is encountered. Only vectors that pass bitmap testing are included in thefi nal top-k results. This strategy is suitable in many cases when C_A or C_V is moderately selective.

Strategy C: vector-first-attribute-full-scan. In contrast to the strategy A, this approach only uses the vector constraint C_V to obtain the relevant entities via vector indexing like IVF_FLAT. Then the resultant entities are fully scanned to verify if they satisfy the attribute constraint C_A. To make sure there are k final results, it searches for $\theta \cdot k$ ($\theta > 1$) results during the vector query processing. This strategy is suitable when the vector constraint C_V is highly selective that the number of candidates is relatively small.

Strategy D: cost-based. It is a cost-based approach that estimates the cost of the strategy A, B, C, and picks up the one with the least cost as proposed in AnalyticDB-V [65]. From [65] and our experiments, the cost-based strategy is suitable in almost all cases.

Strategy E: partition-based. This is a partition-based approach that we develop in Milvus. The main idea is that it partitions the dataset based on the frequently searched attribute and applies the cost-based approach (i.e., the strategy D) for each partition. In particular, we maintain the frequency of each searched attribute in a hash table and increase the counter whenever a query refers to that attribute. Given a query of attributefi ltering, it only searches the partitions whose attribute-ranges overlap with the query range. More importantly, if the range of a specific partition is covered by the query range, then this strategy does not need to check the attribute constraint (C_A) anymore and only focuses on vector query processing (C_V) in that partition, because all the vectors in that partition satisfy the attribute constraint.

As an example, suppose that there are many queries involving the attribute 'price' and the strategy E splits the dataset intofi ve partitions: $\mathcal{P}_0$[1∼100], $\mathcal{P}_1$[101∼200], $\mathcal{P}_2$[201∼300], $\mathcal{P}_3$[301∼400], $\mathcal{P}_4$[401∼500]. Then if the attribute constraint (C_A) of the query is [50∼250], then only $\mathcal{P}_0$, $\mathcal{P}_1$, and $\mathcal{P}_2$ are necessary for searching because their ranges overlap with the query range. And when searching $\mathcal{P}_1$, there is no need to check the attribute constraint since its range is completely covered by the query range. This can significantly improve the query performance.

In the current version of Milvus, we create the partitions offline based on historical data and serve query processing online. The number of partitions (denoted as ρ) is a parameter configured by users. Choosing a proper ρ is subtle: If ρ is too small, then each partition contains too many vectors and it becomes hard to prune irrelevant partitions for this strategy; If ρ is too big, then the number of vectors in each partition is so small that the vector indexing deteriorates towards linear search. Based on our experience, we recommend ρ to be chosen such that each partition contains roughly 1 million vectors. For example, on a billion-scale dataset, there are around 1000 partitions. However, it is an interesting future work to investigate the use of machine learning and statistics to dynamically partition the data and decide the right number of partitions.

4.2 Multi-vector Queries

In many applications, each entity is specified by multiple vectors for accuracy. For example, intelligent video surveillance applications use different vectors to describe the front face, side face, and posture for each person captured on camera [10]. Recipe search applications use multiple vectors to represent text description and associated images for each recipe [56]. Another source of multi-vector is that many applications use more than one machine learning model even for the same object to best describe that object [30, 69].

Formally, each entity contains μ vectors $\mathbf{v}_0$, $\mathbf{v}_1$, ..., $\mathbf{v}_{\mu-1}$. Then a multi-vector queryfi nds top-k entities according to an aggregated scoring function g over the similarity function f (e.g., inner product) of each individual vector $\mathbf{v}_i$. Specifically, the similarity of two entities X and Y is computed as $g(f(X.\mathbf{v}_0, Y.\mathbf{v}_0), ..., f(X.\mathbf{v}_{\mu-1}, Y.\mathbf{v}_{\mu-1}))$ where $X.\mathbf{v}_i$ means the vector $\mathbf{v}_i$ of the entity X. To capture a wide range of applications, we assume the aggregation function g to be monotonic in the sense that g is non-decreasing with respect to every $f(X.\mathbf{v}_i, Y.\mathbf{v}_i)$ [19]. In practice, many commonly used aggregation functions are monotonic, e.g., weighted sum, average/median, and min/max.

Naive solution. Let $\mathcal{D}$ be the dataset and $\mathcal{D}_i$ is a collection of $\mathbf{v}_i$ of all the entities, i.e., $\mathcal{D}_i = \{e.\mathbf{v}_i | e \in \mathcal{D}\}$. Given a query q, the naive solution is to issue an individual top-k query for each vector $q.\mathbf{v}_i$ on $\mathcal{D}_i$ to produce a set of candidates, which are further computed to obtain thefi nal top-k results. Although simple, it can miss many true results leading to extremely low recall (e.g., 0.1). This approach

was widely used in the area of AI and machine learning to support effective recommendations, e.g., [29, 70].

In Milvus, we develop two new approaches, namely vector fusion and interactive merging that target for different scenarios.

Vector fusion. We illustrate the vector fusion approach assuming that the similarity function is inner product and we will explain how to extend to other similarity functions afterwards. Let e be an arbitrary entity in the dataset and $\mathbf{v}_0$, $\mathbf{v}_1$, ..., $\mathbf{v}_{\mu-1}$ be the μ vectors that each entity contains, this approach stores for each entity e its μ vectors as a concatenated vector $\mathbf{v} = [e.\mathbf{v}_0, e.\mathbf{v}_1, ..., e.\mathbf{v}_{\mu-1}]$. Let q be a query entity, during query processing, this approach applies the aggregation function g to the μ vectors of q, producing an aggregated query vector. For example, if the aggregation function is weighted sum with w_i for each weight, then the aggregated query vector is: $[w_0 \times q.\mathbf{v}_0, w_1 \times q.\mathbf{v}_1, ..., w_{\mu-1} \times q.\mathbf{v}_{\mu-1}]$. Then it searches the aggregated query vector against the concatenated vectors in the dataset to obtain thefi nal results. It is straightforward to prove the correctness of vector fusion because the similarity function of inner product is decomposable.

The vector fusion approach is simple and efficient because it only needs to invoke the vector query processing once. But it requires a decomposable similarity function such as inner product. This sounds restrictive but when the underlying data is normalized, many similarity functions such as cosine similarity and Euclidean distance can be converted to inner product equivalently.

Iterative merging. If the underlying data is not normalized *and* the similarity function is not decomposable (e.g., Euclidean distance), then the above vector fusion approach is not applicable. Then we develop another algorithm called iterative merging (see Algorithm 2) that is built on top of Fagin's well-known NRA algorithm [19], a general technique for top-k query processing.[7]

Our initial try is actually to use the NRA algorithm [19] by treating the results of each $q.\mathbf{v}_i$ on $\mathcal{D}_i$ as a stream provided by Milvus. However, we quicklyfi nd that it is inefficient because NRA frequently calls `getNext()` to obtain the next result of $q.\mathbf{v}_i$ interactively. However, existing vector indexing techniques such as quantization-based indexes and graph-based indexes do not support `getNext()` efficiently. A full search is required to get the next result. Another drawback of NRA is that, it incurs significant overhead to maintain the heap since every access in NRA needs to update the scores of the current objects in the heap.

Thus, iterative merging makes two optimizations over NRA: (1) It does not rely on `getNext()` and instead calls VectorQuery($q.\mathbf{v}_i$, $\mathcal{D}_i$, k') with adaptive k' to get the top-k' query results of $q.\mathbf{v}_i$. As a result, it does not need to invoke the vector query processing for every access as NRA does. It can also eliminate the expensive overhead of the heap maintenance as in NRA. (2) It introduces an upper bound of the maximum number of steps to access since the query results in Milvus are approximate.

Algorithm 2 shows iterative merging. The main idea is that it iteratively issues a top-k' query processing for each $q.\mathbf{v}_i$ on $\mathcal{D}_i$ and puts the results to $\mathcal{R}_i$, where $\mathcal{D}_i$ is a collection of $\mathbf{v}_i$ of all the entities in the dataset $\mathcal{D}$, i.e., $\mathcal{D}_i = \{e.\mathbf{v}_i | e \in \mathcal{D}\}$, see line 3 and 4 in Algorithm 2. Then it executes the NRA algorithm over all the $\mathcal{R}_i$. If at least k results can be fully determined (line 5), i.e., NRA can safely stop, then the algorithm can terminate since top-k results

[7]Note that the TA algorithm in [19] cannot be applied in this setting because TA requires random access that is not available here.

Algorithm 2: Iterative merging

```
1 k' ← k;
2 while k' < threshold do
      // run top-k' processing for each q.v_i on D_i
3     foreach i do
4         R_i ← VectorQuery(q.v_i, D_i, k');
5     if k results are fully determined with NRA [19] on all R_i
      then
6         return top-k results;
7     else
8         k' ← k' × 2;
9 return top-k results from ∪_i R_i;
```

can be produced. Otherwise, it doubles k' and iterates the process until k' reaches to a pre-defined threshold (line 2).

In contrast to the vector fusion approach, the iterative merging approach makes no assumption on the data and similarity functions, thus it can be used in a wide spectrum of scenarios. But the performance will be worse than vector fusion when the similarity function is decomposable.

Note that in the databasefi eld, there are many top-k algorithms proposed, e.g., [5, 12, 31, 42, 62]. However, those algorithms cannot be directly used to solve the multi-vector query processing, because the underlying vector indexes cannot support `getNext()` efficiently as mentioned earlier. The proposed iterative merging approach (Algorithm 2) is a generic framework so that it is possible to incorporate other top-k algorithms (e.g., [42]) by replacing line 5. But it remains an open question in terms of optimality and it is also interesting to optimize multi-vector query processing in the future.

5 SYSTEM IMPLEMENTATION

In this section, we present the implementation details of asynchronous processing, snapshot isolation, and distributed computing.

5.1 Asynchronous Processing

Milvus is designed to minimize the foreground processing via asynchronous processing to improve throughput. When Milvus receives heavy write requests, itfi rst materializes the operations (similar to database logs) to disk and then acknowledges to users. There is a background thread that consumes the operations. As a result, users may not immediately see the inserted data. To prevent this, Milvus provides an API flush() that blocks all the incoming requests until the systemfi nishes processing all the pending operations. Besides that, Milvus builds indexes asynchronously.

5.2 Snapshot Isolation

Milvus provides snapshot isolation to make sure reads and writes see a consistent view since Milvus supports dynamic data management. Every query only works on the snapshot when the query starts. Subsequent updates to the system will create new snapshots and do not interfere with the on-going queries.

Milvus manages dynamic data following the LSM-style. All the new data are inserted to memoryfi rst and thenfl ushed to disk as immutable segments. Each segment has multiple versions and a new

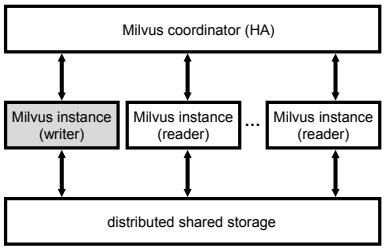


Figure 5: Milvus distributed system

version is generated whenever the data or index in that segment is changed (e.g., uponfl ushing, merging, or building index). All the latest segments at any time form a snapshot. Each segment can be referenced by one or more snapshots. When the system starts, there are no segments. Assuming that there are some inserts flushed to disk at t_1, which forms segment 1. Later on at t_2, segment 2 is generated. Now there are two snapshots in the system where snapshot 1 points to segment 1 and snapshot 2 points to both segment 1 and segment 2. So the segment 1 is referenced by two snapshots. All the queries before t_2 work on snapshot 1 and all the queries after t_2 work on snapshot 2. There is a background thread to garbage collect the obsolete segments if they are not referenced.

Note that the snapshot isolation is applied to the internal data reorganizations in the LSM structure. In this way, all the (internal) reads are not blocked by writes.

5.3 Distributed System

For scalability and availability, Milvus is a distributed system that supports data management across multiple nodes. From the high level, Milvus is a shared-storage distributed system that separates computing from storage to achieve the best elasticity. The shared-storage architecture is widely used in modern cloud systems such as Snowflake [16] and Aurora [63].

Figure 5 shows the overall architecture consisting of three layers. The storage layer is based on Amazon S3 (also used in Snowflake [16]) because S3 is highly available. The computing layer processes user requests such as data insertions and queries. It also has local memory and SSDs for caching data to minimize frequent accesses to S3. Besides that, there is a coordinator layer to maintain the metadata of the system such as sharding and load balancing information. The coordinator layer is highly available with three instances managed by Zookeeper.

Next, we elaborate more on the computing layer, which is stateless to achieve elasticity. It includes a single writer instance and multiple reader instances since Milvus is read-heavy and currently a single writer is sufficient to meet the customer needs. The writer instance handles data insertions, deletions, and updates. The reader instances process user queries. Data is sharded among the reader instances with consistent hashing. The sharding information is stored in the coordinator layer. There are no cross-shard transactions since there are no mixed reads and writes in the same request. The design achieves near-linear scalability as shown in the experiments (Figure 10). All the computing instances are managed by Kubernetes (K8s). When an instance is crashed, K8s will automatically restart a new instance to replace the old one. If the writer instance crashes, Milvus relies on WAL (write-ahead logging) to guarantee atomicity. Since the instances are stateless, crashing will not affect data consistency. Besides that, K8s can also elastically add more reader instances if existing ones are overloaded.

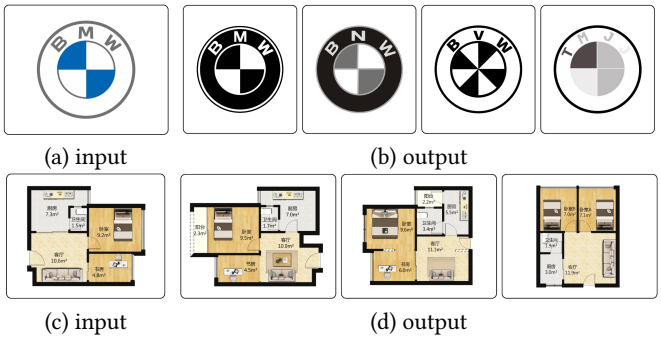


Figure 6: Milvus for image search

To minimize the network overhead between computing and storage, Milvus employs two optimizations: (1) The computing layer only sends logs (rather than the actual data) to the storage layer, similar to Aurora [63]. As mentioned in Sec. 5.1, Milvus asynchronously processes the logs with a background thread to improve performance. In the current implementation, the background thread comes from the writer instance since the writer's load is not too high. Otherwise, the log processing can be managed by a dedicated instance. (2) Another optimization is that each computing instance has a significant amount of buffer memory and SSDs to reduce accesses to the shared storage.

6 APPLICATIONS

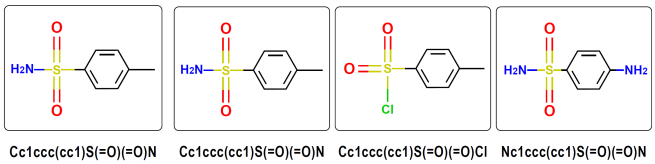
In this section, we present applications that are powered by Milvus. We have built **10 applications** on top of Milvus that includes image search, video search, chemical structure analysis, COVID-19 dataset search, personalized recommendation, biological multi-factor authentication, intelligent question answering, image-text retrieval, cross-modal pedestrian search, and recipe-food search. This section presents two of them due to space limit and more can be found in *https://github.com/milvus-io/bootcamp/tree/master/EN_solutions*.

6.1 Image Search

Image search is a well known application of vector search where each image is naturally converted to a vector using deep learning models such as VGG [58] and ResNet [28].

Two tech companies, Qichacha[8] and Beike Zhaofang,[9] currently use Milvus for large-scale image searches. Qichacha is a leading Chinese website for storing and searching business information (of over 100 million companies), e.g., the names of officers/shareholders and credit information. Milvus supports Qichacha infi nding similar trademarks for customers to check if their trademarks have been registered. Beike Zhaofang is one of the biggest online real estate transaction platform in China. Milvus supports Beike Zhaofang in finding similar houses and apartments (e.g.,fl oor plans). Figure 6 shows an example of searching business trademarks and houses in Qichacha and Beike Zhaofang using Milvus.

[8]https://www.qcc.com/
[9]https://www.ke.com/



(a) input (b) output

Figure 7: Milvus for chemical structure analysis

6.2 Chemical Structure Analysis

Chemical structure analysis is an emerging application that depends on vector search. Recent studies have demonstrated that a new efficient paradigm of understanding the structure of a chemical substance is to encode it into a high-dimensional vector and use vector similarity search (e.g., with Tanimoto distance [9]) tofind similar structures [9, 66].

Milvus is now adopted by Apptech,[10] a major pharmaceutical company developing new medicines and medical devices. Milvus significantly reduces the time of chemical structure analysis from hours to less than a minute. Figure 7 shows an example of searching similar chemical structures using Milvus.

7 EXPERIMENTS

7.1 Experimental Setup

Experimental platform. We conduct all the experiments on Alibaba Cloud and use different types of computing instances (up to 12 nodes) for different experiments to save monetary cost. By default, we use the CPU instance of ecs.g6e.4xlarge (Xeon Platinum 8269 Cascade 2.5GHz, 16 vCPUs, 35.75MB L3 cache, AVX512, 64GB memory, and NAS elastic storage). The GPU instance is ecs.gn6i-c16g1.4xlarge (NVIDIA Tesla T4, 64KB private memory, 512KB local memory, 16GB global memory, and PCIe 3.0 16x interface).

Datasets. To be reproducible, we use the following two public datasets to evaluate Milvus: SIFT1B [34] and Deep1B [8]. SIFT1B contains 1 billion 128-dimensional SIFT vectors (512GB) and Deep1B contains 1 billion 96-dimensional image vectors (384GB) extracted from a deep neural network. Both are standard datasets used in many previous works on vector similarity search and approximate nearest neighbor search [35, 41, 65, 68].

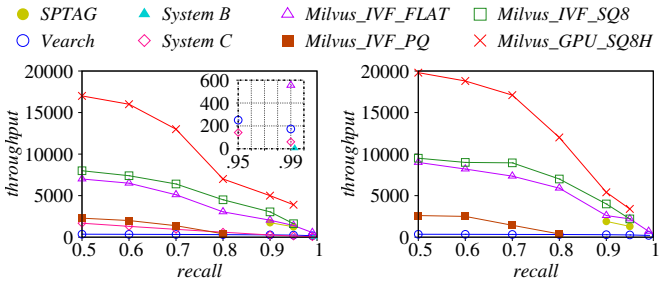
Competitors. We compare Milvus against two open-source systems: Jingdong Vearch (v3.2.0) [4, 39] and Microsoft SPTAG [14]. We also compare Milvus with three industrial-strength commercial systems (with latest version as of July 2020) anonymized as System A, B, and C for commercial reasons. Since Milvus is implemented on top of Faiss [3, 35], we also present the performance comparison by evaluating the algorithmic optimizations in Milvus (Sec. 7.4).

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. Specifically, let S be the ground-truth top-k result set and S' be the top-k results from a system, then the recall is defined as $|S \cap S'|/|S|$. Besides that, we also measure the throughput of a system by issuing 10,000 random queries to the datasets.

7.2 Comparing with Prior Systems

In this experiment, we compare Milvus against prior systems in terms of recall and throughput. We use thefi rst 10 million vectors

[10]https://www.wuxiapptec.com/



(a) SIFT10M (b) Deep10M

Figure 8: System evaluation on IVF indexes

from each dataset (referred to as SIFT10M and Deep10M) because prior systems are slow in building indexes and executing queries on billion-scale datasets. Note that we also evaluate Milvus on the full billion-scale vectors in Sec. 7.3 to demonstrate the system scalability. Except for the three commercial systems (A, B, and C) that the minimum configuration requires multiple nodes, we run all other systems (including Milvus) in a single node. Specifically, we run System A and C on two nodes (with 64GB memory per node); System B on four nodes (with 128GB memory per node).

In this experiment, we use two indexes IVF_FLAT and HNSW whenever possible since both are supported by most systems, although Milvus supports more indexes.

Figure 8 shows the results on IVF indexes (i.e., quantization-based indexes). Overall, Milvus (even CPU version) significantly outperforms existing systems by up to two orders of magnitude while keeping the similar recall. In particular, Milvus is 6.4× ~ 27.0× faster than Vearch; 153.7× faster than System B even if System B runs on four nodes;[11] 4.7× ~ 11.5× faster than System C even if System C runs on two nodes; 1.3× ~ 2.1× faster than SPTAG (tree-based index). But SPTAG cannot achieve very high recall (e.g., 0.99) as Milvus does and also SPTAG takes 14× more memory than Milvus (17.88GB vs. 1.27GB).[12] The GPU version of Milvus is even faster since data canfi t in the GPU memory in this setting. We omit the results of System B on Deep10M since it only supports the Euclidean distance metric. We also omit the results of Vearch on GPU because there are multiple bugs in building indexes that their engineers were stillfi xing by the time of paper writing.[13] We defer the results of System A to Figure 9 since it only supports the HNSW index.

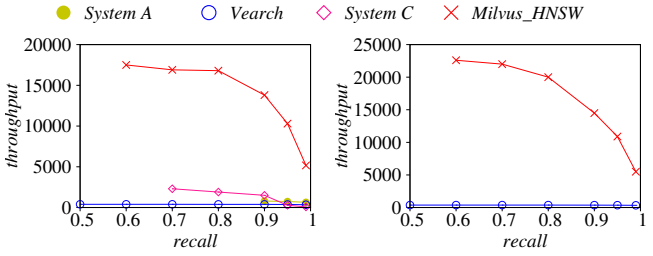
The performance advantage of Milvus comes from a few factors in addition to engineering optimizations. (1) Milvus introducesfine-grained parallelism that supports both inter-query and intra-query parallelism to best leverage multi-core CPUs. (2) Milvus develops cache-aware and SIMD-aware optimizations to reduce CPU cache misses and leverage wide SIMD instructions. (3) Milvus optimizes for the hybrid execution between GPU and CPU.

Figure 9 shows the results on the HNSW index of each system. Milvus outperforms existing systems by a large margin. Specifically, it is 15.1× ~ 60.4× faster than Vearch; 8.0× ~ 17.1× faster than

[11]Note that System B has a single data point in Figure 8 and relatively low performance because it used brute-force search as it disabled the parameter tuning (e.g., n_{probe} and n_{list}) when we tested in 08/2020. But we expect a better performance in System B once the parameter tuning is enabled (to use index) in the future.

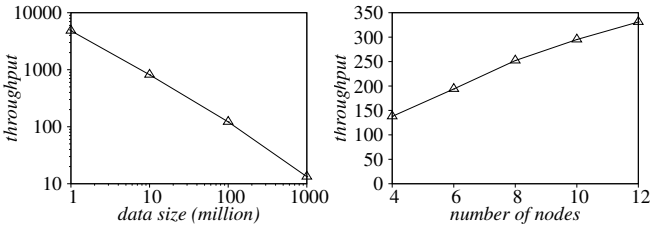
[12]Besides that, SPTAG does not support dynamic data management, GPU, attribute filtering, multi-vector query, and distributed systems that Milvus provides, see Table 1.

[13]We submitted a bug report in 09/2020: *https://github.com/vearch/vearch/issues/252.*



(a) SIFT10M (b) Deep10M

Figure 9: System evaluation on HNSW indexes



(a) Varying data size (b) Varying #nodes

Figure 10: Scalability

System A; 7.3× ~ 73.9× faster than System C. We omit System A on Deep10M because System A does not support inner product. We omit also System C on Deep10M because the index building fails to complete after more than 100 hours.

7.3 Scalability

In this experiment, we evaluate the scalability of Milvus in terms of data size and the number of servers. We use the IVF_FLAT indexing on the SIFT1B dataset that includes 1 billion vectors.

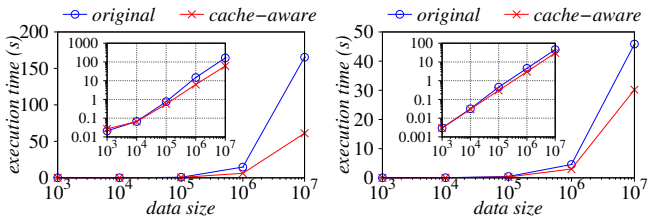
Figure 10a shows the results on a single node of ecs.re6.26xlarge (104 vCPUs and 1.5TB memory) that canfi t the entire data in memory. As the data increases, the throughput gracefully drops proportionally. Figure 10b shows the scalability of distributed Milvus. The data is sharded among the nodes where each node is of ecs.g6e.13xlarge (52 vCPUs and 192GB memory). As the number of nodes increases, the throughput increases linearly. Note that we observe that Milvus achieves higher throughput on the ecs.g6e.13xlarge instance than the ecs.re6.26xlarge instance due to the higher competition on the shared CPU caches and memory bandwidth among more cores.

7.4 Evaluation of Optimizations

Figure 11 shows the impact of cache-aware design on two CPUs with different L3 cache sizes: 12MB (Intel Core i7-8700 3.2GHz) and 35.75MB (Xeon Platinum 8269 Cascade 2.5GHz). We set the query batch size as 1000 and vary the data size (i.e., the number of vectors) from 1000 to 10 million. It shows that the cache-aware design can achieve a performance improvement up to 2.7× and 1.5× when the cache size is 12MB and 35.75MB, respectively.

Figure 12 shows the impact of SIMD-aware optimizations following the experimental setup in Figure 11. It compares the performance of AVX2 and AVX512 on the Xeon CPU. Figure 12 demonstrates that AVX512 is roughly 1.5× faster than AVX2.

Figure 13 evaluates the efficiency of the hybrid algorithm SQ8H (Algorithm 1) in Milvus on SIFT1B where data cannotfi t into GPU



(a) 12MB L3 cache (b) 35.75MB L3 cache

Figure 11: Evaluating the cache-aware design

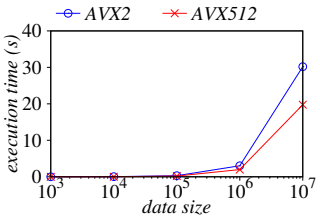


Figure 12: SIMD optimizations

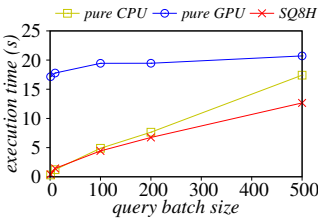


Figure 13: GPU indexing

memory. We compare SQ8H with SQ8 on pure CPU and pure GPU. It shows that GPU SQ8 is slower than CPU SQ8 due to the data transfer. As the query batch size increases, the performance gap between GPU and CPU becomes smaller since more computations are pushed to the GPU. In all cases, SQ8H is faster than running SQ8 on pure CPU and pure GPU. That is because SQ8H only stores the centroids in GPU memory to execute thefi rst step and allows the CPU to execute the second step so that there is no any data segment transferred to GPU memory on thefl y.

7.5 Evaluating Attribute Filtering

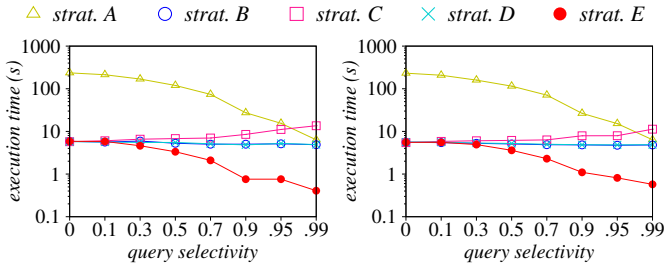
We define the query selectivity as the percentage of entities that fails the attribute constraint C_A following [65]. Thus, a higher selectivity means that a smaller number of entities can pass C_A. Regarding the dataset, we extract thefi rst 100 million vectors from SIFT1B and augment each vector with an attribute of a random value ranging from 0 to 10000. We follow [65] to generate two scenarios of different k (50 and 500) and recall (0.95 and 0.85).

Figure 14 shows the results with varying query selectivity. For the strategy A, its performance increases as the selectivity increases because the number of examined vectors decreases. The strategy B is insensitive to the selectivity since the bottleneck is vector similarity search. The strategy C is slower than the strategy B since it requires to check θ times of the vectors where θ is 1.1 in this experiment. The strategy D outperforms A, B, and C since it uses a cost-based approach to choose the best between the three. Our new approach, i.e., the strategy E, significantly outperforms the strategy D by up to 13.7× due to the partitioning.

Figure 15 compares Milvus against System A, B, C, and Vearch in terms of attributefi ltering. It shows that Milvus outperforms those systems by 48.5× to 41299.5×. Note that we omit the results of System B in Figure 15b because its parameters arefi xed by the system that users are not allowed to change.

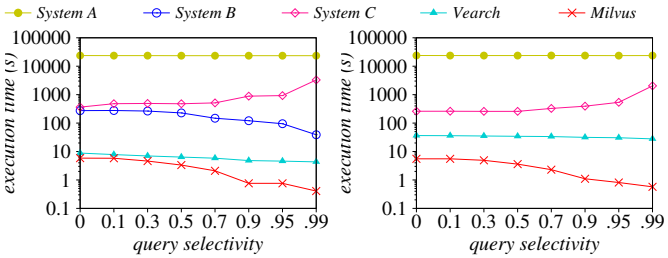
7.6 Evaluating Multi-vector Query Processing

In this experiment, we evaluate the algorithms for multi-vector query processing. Since SIFT1B and Deep1B only contain one vector



(a) k = 50 & recall >= 0.95 (b) k = 500 & recall >= 0.85

Figure 14: Attributefi ltering in Milvus



(a) k = 50 & recall >= 0.95 (b) k = 500 & recall >= 0.85

Figure 15: Attributefi ltering comparison

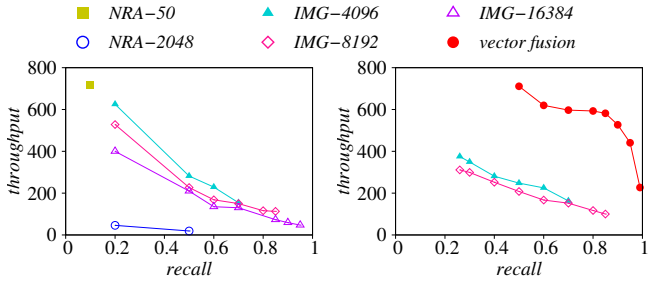
per entity, we then use another dataset called Recipe1M [50, 56] that includes more than one million cooking recipes and food images. Thus, each entity is described by two vectors: text vector (i.e., recipe description) and image vector (i.e., food image). We randomly pick up 10000 queries from the dataset and set k as 50 in this experiment. We use the IVF_FLAT indexing in this experiment. Besides that, we use weighted sum as the aggregation function.

Figure 16a shows the results where the similarity metric is Euclidean distance. We compare the standard NRA algorithm of different k (50 and 2048) and our iterative merging ("IMG" for short) of different k' (4096, 8192, and 16384). It shows that the standard NRA approach is either slow or produces low recall. In particular, the NRA-50 approach is fast but the recall is only 0.1. The NRA-2048 increases the recall a bit (up to 0.5), but the performance is low while our iterative merging algorithm (with k' being 4096) is 15× faster than NRA-2048 with a similar recall. That is because IMG does not need to invoke the vector query processing every time and also it has lower maintenance cost of heaps.

Figure 16b shows the results on the inner product metric. We compare the iterative merging (IMG-4096 and IMG-8192) with vector fusion. It shows that vector fusion is 3.4× ~ 5.8× faster since it only needs to issue a single top-k vector similarity search.

8 RELATED WORK

Vector similarity search (a.k.a high-dimensional nearest neighbor search) is an extensively studied topic both for approximate search (e.g., [7, 41]) and exact search (e.g., [38, 42]). This work focuses on approximate search in order to achieve high performance. Prior works on approximate search can be roughly classified in four categories: LSH-based [23, 23, 24, 32, 40, 44, 45, 48, 73], tree-based [17, 46, 54, 57], graph-based [20, 43, 49, 61, 72], and quantization-based [3, 6, 22, 27, 33, 35]. However, those works are



(a) Euclidean distance (b) Inner product

Figure 16: Multi-vector processing in Milvus

all about indexes while Milvus is a full-fledged vector data management system including indexes, query engine, GPU engine, storage engine, and distributed system. Moreover, Milvus's extensible index framework can easily incorporate those indexes as well as any new index if necessary. There are also open-source libraries for vector similarity search, e.g., Faiss [35] and SPTAG [14]. But they are libraries not systems. We summarize the differences in Table 1.

Recent industrial-strength vector data management systems such as Alibaba PASE [68] and Alibaba AnalyticDB-V [65] are not particularly optimized for vectors. Their approach is to extend the relational database to support vectors. As a result, the performance suffers severely as demonstrated in experiments. Specialized vector systems like Vearch [39] are not suitable for billion-scale data and Vearch is significantly slower than Milvus.

There are also GPU-based vector search engines, e.g., [35, 72]. Of which, [72] optimizes HNSW for GPU but it assumes data to be small enough tofi t into GPU memory. Faiss [35] also supports GPU, but it loads the whole data segments on demand if data cannotfit into GPU memory, leading to low performance. Instead, Milvus develops a new hybrid index (SQ8H) that combines the best of the GPU and CPU without loading data on thefl y for fast query processing.

This work is relevant to the trend of building specialized data engines since one size does notfi t all [60], e.g., specialized graph engine [18], IoT engine [21], time series database [55], and scientific database [59]. In this regard, Milvus is a specialized data engine for managing vector data.

9 CONCLUSION

In this work, we share our experience in building Milvus over the last few years at Zilliz. Milvus has been adopted by hundreds of companies and is currently an incubation-stage project at the LF AI & Data Foundation. Looking forward, we plan to leverage FPGA to accelerate Milvus. We have implemented the IVF_PQ indexing on FPGA and the initial results are encouraging. Another interesting yet challenging direction is to architect Milvus as a cloud-native data management system and we are currently working on it.

ACKNOWLEDGMENTS

Milvus is a multi-year project that involves many engineers at Zilliz. In particular, we thank Shiyu Chen, Qing Li, Yunmei Li, Chenglong Li, Zizhao Chen, Yan Wang, and Yunying Zhang for their contributions. We also thank Haimeng Cai and Chris Warnock for proofreading the paper. Finally, we would like to thank Walid G. Aref and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2020. *Annoy: Approximate Nearest Neighbors Oh Yeah.* https://github.com/spotify/annoy

[2] 2020. *ElasticSearch: Open Source, Distributed, RESTful Search Engine.* https://github.com/elastic/elasticsearch

[3] 2020. *Facebook Faiss.* https://github.com/facebookresearch/faiss

[4] 2020. *Vearch: A Distributed System for Embedding-based Retrieval.* https://github.com/vearch/vearch

[5] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. 2007. Best Position Algorithms for Top-k Queries. In *International Conference on Very Large Data Bases (VLDB)*. 495–506.

[6] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *Proceedings of the VLDB Endowment (PVLDB)* 9, 4 (2015), 288–299.

[7] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2018. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Computing Research Repository (CoRR)* abs/1807.05614 (2018).

[8] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2055–2063.

[9] Dávid Bajusz, Anita Rácz, and Károly Héberger. 2015. Why Is Tanimoto Index An Appropriate Choice For Fingerprint-Based Similarity Calculations? *Journal of Cheminformatics* 7 (2015).

[10] Tadas Baltrusaitis, Chaitanya Ahuja, and Louis-Philippe Morency. 2019. Multimodal Machine Learning: A Survey and Taxonomy. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 41, 2 (2019), 423–443.

[11] Oren Barkan and Noam Koenigstein. 2016. ITEM2VEC: Neural Item Embedding for Collaborative Filtering. In *IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*. 1–6.

[12] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. 2011. Interval-based Pruning for Top-k Processing over Compressed Lists. In *International Conference on Data Engineering (ICDE)*. 709–720.

[13] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The Rise of Deep Learning in Drug Discovery. *Drug Discovery Today* 23, 6 (2018), 1241–1250.

[14] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A Library for Fast Approximate Nearest Neighbor Search.* https://github.com/Microsoft/SPTAG

[15] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *ACM Conference on Recommender Systems (RecSys)*. 191–198.

[16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *ACM Conference on Management of Data (SIGMOD)*. 215–226.

[17] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *ACM Symposium on Theory of Computing (STOC)*. 537–546.

[18] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *ACM Conference on Management of Data (SIGMOD)*. 377–392.

[19] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *ACM Symposium on Principles of Database Systems (PODS)*. 102–113.

[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment (PVLDB)* 12, 5 (2019), 461–474.

[21] Christian Garcia-Arellano, Adam J. Storm, David Kalmuk, Hamdi Roumani, Ronald Barber, Yuanyuan Tian, Richard Sidle, Fatma Özcan, Matt Spilchen, Josh Tiefenbach, Daniel C. Zilio, Lan Pham, Kostas Rakopoulos, Alexander Cheung, Darren Pepper, Imran Sayyid, Gidon Gershinsky, Gal Lushi, and Hamid Pirahesh. 2020. Db2 Event Store: A Purpose-Built IoT Database Engine. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3299–3312.

[22] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 36, 4 (2014), 744–755.

[23] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases (VLDB)*. 518–529.

[24] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1483–1497.

[25] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In *ACM Conference on Knowledge Discovery & Data Mining (KDD)*. 311–320.

[26] Martin Grohe. 2020. Word2vec, Node2vec, Graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *ACM Symposium on Principles of Database Systems (PODS)*. 1–16.

[27] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning (ICML)*.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[29] D. Frank Hsu and Isak Taksa. 2005. Comparing Rank and Score Combination Methods for Data Fusion in Information Retrieval. *Information Retrieval (IR)* 8, 3 (2005), 449–480.

[30] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. 2019. FiBiNET: Combining Feature Importance and Bilinear Feature Interaction for Click-through Rate Prediction. In *ACM Conference on Recommender Systems (RecSys)*. 169–177.

[31] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys (CSUR)* 40, 4 (2008), 11:1–11:58.

[32] Omid Jafari, Parth Nagarkar, and Jonathan Montaño. 2020. mmLSH: A Practical and Efficient Technique for Processing Approximate Nearest Neighbor Queries on Multimedia Data. *Computing Research Repository (CoRR)* abs/2003.06415 (2020).

[33] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 33, 1 (2011), 117–128.

[34] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in One Billion Vectors: Re-rank with Source Coding. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 861–864.

[35] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* (2019).

[36] Timothy King. 2019. *80 Percent of Your Data Will Be Unstructured in Five Years.* https://solutionsreview.com/data-management/80-percent-of-your-data-will-be-unstructured-in-five-years/

[37] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *International Conference on Machine Learning (ICML)*. 1188–1196.

[38] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *ACM Conference on Management of Data (SIGMOD)*. 835–850.

[39] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-Commerce Platform. In *Middleware*. 9–16.

[40] Mingjie Li, Ying Zhang, Yifang Sun, Wei Wang, Ivor W. Tsang, and Xuemin Lin. 2020. I/O Efficient Approximate Nearest Neighbour Search based on Learned Functions. In *International Conference on Data Engineering (ICDE)*. 289–300.

[41] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 32, 8 (2020), 1475–1488.

[42] Yuliang Li, Jianguo Wang, Benjamin Pullman, Nuno Bandeira, and Yannis Papakonstantinou. 2019. Index-Based, High-Dimensional, Cosine Threshold Querying with Optimality Guarantees. In *International Conference on Database Theory (ICDT)*. 11:1–11:20.

[43] Peng-Cheng Lin and Wan-Lei Zhao. 2019. A Comparative Study on Hierarchical Navigable Small World Graphs. *Computing Research Repository (CoRR)* abs/1904.02077 (2019).

[44] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, and Lu Qin. 2019. I-LSH: I/O Efficient c-Approximate Nearest Neighbor Search in High-Dimensional Space. In *International Conference on Data Engineering (ICDE)*. 1670–1673.

[45] Kejing Lu and Mineichi Kudo. 2020. R2LSH: A Nearest Neighbor Search Scheme Based on Two-dimensional Projected Spaces. In *International Conference on Data Engineering (ICDE)*. 1045–1056.

[46] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1443–1455.

[47] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *VLDB Journal* 29, 1 (2020), 393–418.

[48] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2017. Intelligent Probing for Locality Sensitive Hashing: Multi-Probe LSH and Beyond. *Proceedings of the VLDB Endowment (PVLDB)* 10, 12 (2017), 2021–2024.

[49] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 42, 4 (2020), 824–836.

[50] Javier Marín, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. 2018. Recipe1M: A Dataset for Learning Cross-Modal Embeddings for Cooking Recipes and Food Images. *Computing Research Repository (CoRR)* abs/1810.06553 (2018).

[51] Adam C. Mater and Michelle L. Coote. 2019. Deep Learning in Chemistry. *Journal of Chemical Information and Modeling* 59, 6 (2019), 2545–2559.

[52] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *International Conference*

on Learning Representations (ICLR).

[53] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Annual Conference on Neural Information Processing Systems (NeurIPS).* 3111–3119.

[54] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 36, 11 (2014), 2227–2240.

[55] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proceedings of the VLDB Endowment (PVLDB)* 8, 12 (2015), 1816–1827.

[56] Amaia Salvador, Nicholas Hynes, Yusuf Aytar, Javier Marin, Ferda Ofli, Ingmar Weber, and Antonio Torralba. 2017. Learning Cross-modal Embeddings for Cooking Recipes and Food Images. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 3068–3076.

[57] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for Fast Image Descriptor Matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 1–8.

[58] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR).*

[59] Michael Stonebraker, Anastasia Ailamaki, Jeremy Kepner, and Alexander S. Szalay. 2012. The Future of Scientific Data Bases. In *International Conference on Data Engineering (ICDE).* 7–8.

[60] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *International Conference on Data Engineering (ICDE).* 2–11.

[61] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Annual Conference on Neural Information Processing Systems (NeurIPS).* 13748–13758.

[62] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *ACM Conference on Management of Data (SIGMOD).* 2659–2665.

[63] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM Conference on Management of Data (SIGMOD).* 1041–1052.

[64] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *ACM Conference on Management of Data (SIGMOD).* 1041–1052.

[65] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3152–3165.

[66] Peter Willett. 2014. The Calculation of Molecular Structural Similarity: Principles and Practice. *Molecular Informatics* 33, 6–7 (2014), 403–413.

[67] Susan Wojcicki. 2020. *YouTube at 15: My Personal Journey and the Road Ahead.* https://blog.youtube/news-and-events/youtube-at-15-my-personal-journey

[68] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *ACM Conference on Management of Data (SIGMOD).* 2241–2253.

[69] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *ACM Conference on Knowledge Discovery & Data Mining (KDD).* 974–983.

[70] Shaoting Zhang, Ming Yang, Timothée Cour, Kai Yu, and Dimitris N. Metaxas. 2015. Query Specific Rank Fusion for Image Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 37, 4 (2015), 803–815.

[71] Shilin Zhang and Hangbin Yu. 2018. Person Re-Identification by Multi-Camera Networks for Internet of Things in Smart Cities. *IEEE Access* 6 (2018), 76111–76117.

[72] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *International Conference on Data Engineering (ICDE).* 1033–1044.

[73] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proceedings of the VLDB Endowment (PVLDB)* 13, 5 (2020), 643–655.