# MVG Index: Empowering Multi-Vector Similarity Search in High-Dimensional Spaces

Mengzhao Wang
Zhejiang University
wmzssy@zju.edu.cn

Xiangyu Ke
Zhejiang University
xiangyu.ke@zju.edu.cn

Lu Chen
Zhejiang University
luchen@zju.edu.cn

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

## ABSTRACT

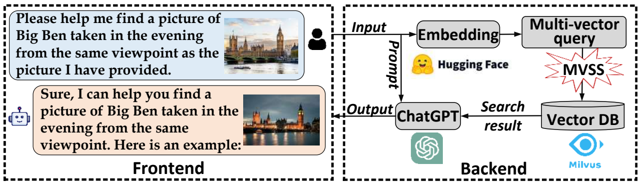
In the realm of high-dimensional space, Vector Similarity Search (VSS) has gained increasing significance, particularly in the context of unstructured data processing. However, existing advancements predominantly assume that each object comprises only a single vector, limiting their applicability to single-modality scenarios. The surge in multi-modal data processing, exemplified by multi-modal large language models, underscores the need for Multi-Vector Similarity Search (MVSS), where each object is composed of multiple vectors. Current attempts to address this challenge through various VSS integrations often suffer from inefficiency and inaccuracy, primarily due to the intrinsic limitations of single-vector indexes.

This study introduces a specialized **M**ulti-**V**ector **G**raph index, denoted as MVG, explicitly designed to tackle the MVSS problem. Notably, MVG effectively consolidates all vector combination relationships within each object pair into a single index, enabling seamless processing of both VSS and MVSS. Furthermore, MVG incorporates a set of computational acceleration methods that integrate characteristics specific to MVSS scenarios into both index construction and search procedures. Three index compression algorithms, grounded in the well-designed layout of the multi-vector index, empower MVG to adapt to various scenarios with specific requirements for index size and search efficiency. Theoretical validations confirm that all acceleration and compression techniques are lossless, ensuring that index quality and search result accuracy remain unaffected. Extensive experiments on real-world datasets substantiate the superiority of MVG in terms of index construction efficiency, space cost, MVSS performance, VSS performance, and scalability when compared to the state-of-the-art VBase. Remarkably, MVG exhibits up to 96.5% lower query latency with a higher recall rate than VBase.

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/ZJU-DAILY/MVG.

## 1 INTRODUCTION

The use of high-dimensional vector-based representation for unstructured data, such as documents and images, has become a key building block for training and deploying Artifical Intelligence (AI) models like GPT and CLIP [15, 45, 58]. This paradigm has shown significant promise in various emerging AI applications, sparking interest in Vector Similarity Search (VSS) in high-dimensional spaces within academic and industrial communities. For instance, applications like Bing Chat benefit from large language models (LLMs) that utilize VSS components for retrieval augmented generation (RAG), achieved by vectorizing documents [3, 64]. VSS has long been a pivotal topic in domains like information retrieval [52, 55], recommendation systems [18, 42], and databases [27, 69]. In VSS, given an object set $S$ where each object is represented as a vector, a query vector $q$, and a vector distance metric $\delta(.)$, the aim is to find the top-$k$ similar objects with the smallest distances to $q$. As the cardinality $n$ and dimensionality $d$ of vectors increase, exact VSS necessitates $O(n \cdot d)$ search time complexity, which becomes prohibitively time-consuming. Therefore, an approximate version[1], also known as Approximate Nearest Neighbor Search (ANNS) [50, 56], is preferred in real-world scenarios. To expedite search procedure, advanced VSS techniques build a vector index, balancing accuracy and efficiency. Numerous studies have indicated that graph-based vector index (e.g., HNSW [40]) is the most promising and has consequently become a mainstream research direction and practical choice in the VSS field [33, 54].


Please help me find a picture of Big Ben taken in the evening from the same viewpoint as the picture I have provided.

Sure, I can help you find a picture of Big Ben taken in the evening from the same viewpoint. Here is an example:

**Figure 1: An example of text-image chat by MVSS.**

In order to enhance the capabilities of AI applications, Multi-Vector Similarity Search (MVSS) has been proposed for processing multi-modal or multi-view data [50, 65]. In this scenario, an object or query contains multiple vectors, and the similarity between objects incorporates distances from multiple vector pairs. Specifically, each object in an object set $S$ consists of $m$ vectors (e.g., $m = 2$), and the query inputs $t$ ($1 \leq t \leq m$) vectors with corresponding weights. The multi-vector distance metric is an aggregate function of multiple single-vector distance metrics (e.g., weighted sum). For instance, users might input a reference image and a modified text to form two query vectors for retrieval in a multi-modal object set [51]. Additionally, a single-modal object may construct multiple vectors from different views [23]; MVSS ensures more accurate search results by combining information from different views. Given the significance of MVSS, recent research surveys and vector database products emphasize the clear demand for a fast and accurate MVSS solution [27, 43, 47, 48, 57]. However, current VSS methods are only capable of addressing VSS with one vector in a query or object.

Example 1. In a text-image chat scenario, users can input a reference image and an instructional text to obtain a desired result

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] Hereafter, we refer to approximate VSS simply as VSS unless otherwise specified.

using MVSS. As depicted in Figure 1, a user can acquire an evening image of Big Ben by submitting a reference daytime picture of Big Ben along with a natural language description of his intention ("Frontend"). In the backend, MVSS plays a pivotal role in quickly and precisely responding to this request. Initially, the input text and image are embedded into two distinct high-dimensional spaces to create a multi-vector query with two vectors using the Hugging Face Embedding API [1]. Subsequently, MVSS is executed on a vector database (e.g., Milvus [50]), and the search result is combined with the user input to prompt ChatGPT [6]. Finally, a user-friendly response is presented in the frontend.

Recent studies address MVSS by optimizing the search process across multiple single-vector indexes. Specifically, they construct $m$ vector indexes on an object set $S$, where each object owns $m$ vectors; a query with $t$ vectors prompts the scanning of $t$ corresponding indexes. A straightforward method, Merging, acquires $t$ candidate sets via VSS on $t$ indexes, culminating in the merging of sets to provide final results [66]. Yet, deciding on an optimal number of returned candidates per VSS remains a conundrum [65]. An excess or shortage of candidates leads to a computational load and complex merging, or incomplete, inaccurate results respectively [50]. Milvus [50] confronts this dilemma by trialing various candidate sizes and resorting the candidates via the NRA algorithm[2] [22], enhancing search accuracy. However, each trial for a larger candidate size necessitates distinct vector index traversal, causing considerable vector access and computation. Alternatively, VBase [65] circumvents NRA reiteration by round-robin traversal of each index. This process is fragmented into rounds and incorporates a mechanism to decide more frequently traversed indexes based on preliminary findings. Thus, results are obtained during traversal, omitting additional merging steps. VBase outperforms Milvus in terms of search efficiency and accuracy. Nevertheless, its reliance on multi-index searching invariably leads to unnecessary vertex visitation, impairing efficiency, given single-vector but not multi-vector object similarity for each index.

We illustrate the primary limitation of current methods using the HNSW index[3]. First, in a single-vector HNSW index, the neighbors of a vertex are determined solely based on the single-vector distance, neglecting the multi-vector distance. To illustrate, we construct two HNSW indexes on a million-scale Recipe dataset [46]: one based on single-vector distance (denoted as HNSW*) and the other based on multi-vector distance (referred to as HNSW**). We observe that the average neighbor overlap ratio is only *18%* between the two indexes, indicating that the single-vector HNSW is inadequate in capturing the multi-vector similarity that is crucial for MVSS. Second, when executing multi-vector search on a single-vector HNSW, the neighbors of a visited vertex may guide the traversal along an incorrect path, resulting in wrong search results. Our evaluation demonstrates that, for identical search parameters, HNSW** achieves a recall rate of *0.98*, while the recall rate of HNSW* is only *0.65*. Furthermore, the computational requirements for serving a query are heavier in HNSW*, suggesting its subpar navigation and neighbor similarity in the context of answering a multi-vector query. In a netshell, the main reason constraining the MVSS performance of existing methods is the drawback of the single-vector index.

In this paper, we present a **M**ulti-**V**ector **G**raph (MVG) index to facilitate fast and accurate MVSS. MVG constructs a HNSW graph on multi-vector objects, covering all potential vector combinations. Within this index, a vertex's neighbors are obtained by computing both single-vector distance and multi-vector distance with any combinations, supporting VSS and MVSS simultaneously. By adaptively visiting relevant neighbors based on query vector combinations, MVG ensures efficient and accurate searching. Specifically, we address the following three research questions.

**(i)** ***Efficient construction of a high-quality multi-vector index.*** Directly constructing a multi-vector index that incorporates all vector combinations is time-consuming. Our evaluation shows that it takes *31.5*× more time than building current single-vector indexes with the same index parameters on the MIT-States dataset. To address this, MVG optimizes multi-vector neighbor generation and employs several lossless acceleration strategies that leverage the characteristics of index construction and multi-vector distance computation. Consequently, MVG can be rapidly constructed, comparable to or even faster than current single-vector indexes.

**(ii)** ***Lossless compression of the multi-vector index.*** A vertex in the multi-vector graph index stores more information that consider multi-vector relationship, resulting in *9.7*× larger index size than current single-vector indexes with identical parameters on MIT-States. To solve this problem, MVG offers three lossless index compression algorithms, grounded in the well-designed layout of the multi-vector index. This allows for compact storage of the compressed index without losing search accuracy for multi-vector queries. Moreover, the decompression procedure mainly involves efficient binary operations, enabling online search at almost the same speed as that on the uncompressed index.

**(iii)** ***Fast and accurate processing of any type of query.*** Serving *any* type of query on a single index requires both diverse neighbors in the index and adaptive neighbor access during search. MVG addresses this challenge by seamlessly processing VSS and MVSS according to query vector combinations on a multi-vector index. By incorporating the characteristics of a multi-vector query, MVG also integrates the lossless acceleration into the search procedure and further enhances it with query input weights. This simplifies numerous multi-vector distance computations, significantly improving search efficiency without sacrificing accuracy.

This work is the first exploration, to the best of our knowledge, to design multi-vector index and search strategies that exploit the intrinsic features of the MVSS problem. Our evaluation shows that MVG outperforms state-of-the-art methods in terms of index construction efficiency, index size, latency, and recall rate, achieving up to *96.5%* reduction in query latency. Notably, MVG easily attains a recall rate over *0.99*, a challenging accuracy level for other methods. The primary contributions of our research are as follows:

- We introduce MVG, a high-performance multi-vector graph index that offers efficient index processing and compact index layout (**§3**). With the ability to support both VSS and MVSS simultaneously using a single index, MVG demonstrates remarkable improvements in search efficiency and accuracy compared to existing methods.

[2] NRA can efficiently obtains the final results from multiple ordered candidate lists by establishing upper and lower bounds of multi-vector distances [22].

[3] Milvus and VBase both utilize HNSW [40] as their underlying indexes.

**Table 1: Frequently used notations.**

| Notations | Descriptions |
|---|---|
| $S, o$ | A set of multi-vector objects, an object in $S$ |
| $q$ | A multi-vector query |
| $\boldsymbol{o_i}, \boldsymbol{q_i}$ | The $i$-th vectors in $o$ and $q$, respectively |
| $d_i$ | The dimension of the $i$-th vectors |
| $m, t$ | The number of vectors in $o$ and $q$, respectively |
| $D_m, D_t$ | The total vector dimension in $o$ and $q$, respectively |
| $\delta(.)$ | The Euclidean distance between two vectors |
| $w_i$ | The weight of the $i$-th distance $\delta(\boldsymbol{o_i}, \boldsymbol{q_i})$ |
| $g(.)$ | The aggregation function of $\delta(.)$ |
| $\|\| \cdot \|\|$ | The $l_2$-norm of a vector |

- We propose a series of acceleration techniques tailored for index construction and search in multi-vector scenarios (**§4.2** and **§5.2**). They leverage the features of building multi-vector indexes and conducting multi-vector searches. We prove that these techniques do not damage the index quality and the search accuracy.
- We develop three compression algorithms grounded in the well-designed layout of the multi-vector index (**§4.3**). They accommodate diverse scenarios with specific index size and search efficiency requirements. Importantly, all of these compression algorithms are lossless, ensuring no loss of search accuracy.
- We provide comprehensive theoretical analysis and empirical verification for each technique. Extensive evaluation on six real-world datasets shows that MVG outperforms state-of-the-art methods in terms of efficiency, accuracy, and scalability (**§6**).

## 2 PRELIMINARIES

In this section, we formally define the Multi-Vector Similarity Search (MVSS) problem. Then, we outline the motivation behind our research. Please refer to Table 1 for frequently used notations.

### 2.1 Problem Definition

We define MVSS following Vector Similarity Search (VSS).

Definition 1. ***VSS.*** *Given an object set $S$ where each object is a single vector, a query vector $\boldsymbol{q}$, and the distance metric $\delta(.)$, VSS identifies the top-$k$ objects most similar to $\boldsymbol{q}$, denoted by $R$. Formally:*

$$R = \arg\min_{R \subset S} \sum_{o \in R} \delta(\boldsymbol{o}, \boldsymbol{q}) \quad . \tag{1}$$

VSS aims to find vectors similar to a given query vector, a problem commonly addressed by advanced graph-based index techniques such as HNSW [33, 40].

Definition 2. ***MVSS.*** *Given an object set $S$ where each object has $m$ vectors, a multi-vector query $q$ with $t$ vectors and distance weights, the distance metric $\delta(.)$, and the aggregation function $g(.)$, which is monotonic and non-decreasing with respect to each $\delta(.)$, MVSS identifies the top-$k$ objects most similar to $q$, denoted by $R$. Formally:*

$$R = \arg\min_{R \subset S} \sum_{o \in R} g(\delta(\boldsymbol{o_0}, \boldsymbol{q_0}), \cdots, \delta(\boldsymbol{o_{t-1}}, \boldsymbol{q_{t-1}})) \quad . \tag{2}$$

The current focus primarily revolves around the case where $m = 2$ for MVSS [50, 65]. Few instances consider $m$ up to 4 [51]. In this paper, we extend to the range of $m$ from 1 to 6, covering most real-world scenarios [17, 30, 46, 62]. For a multi-vector query, the meaningful value of $t$ ranges from 1 to $m$. The corresponding relationship of vectors in $q$ and $o$ is provided alongside $q$. For simplicity, we assume that the $i$-th query vector with $1 \leq i \leq t$ corresponds to the $i$-th vector of object $o$ with $1 \leq i \leq t$. We use the weighted sum for $g(.)$, as it is widely used in current MVSS scenarios [50, 65]:

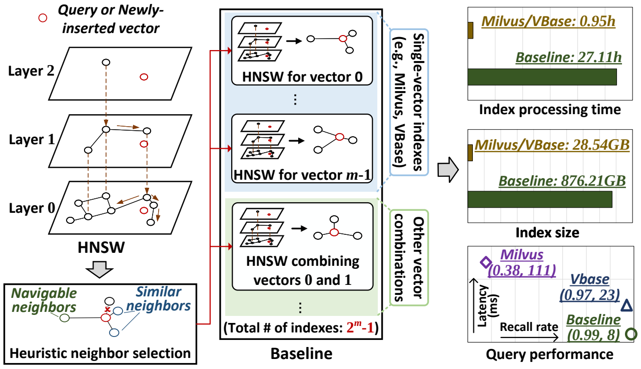


**Figure 2: HNSW-based MVSS methods and performance.**

$$g(\delta(\boldsymbol{o_0}, \boldsymbol{q_0}), \cdots, \delta(\boldsymbol{o_{t-1}}, \boldsymbol{q_{t-1}})) = \sum_{i=0}^{t-1} w_i \cdot \delta(\boldsymbol{o_i}, \boldsymbol{q_i}) \quad . \tag{3}$$

When $m = t = 1$, the MVSS problem reduces to VSS. Thus, MVSS is a more general problem. As the emergence of versatile AI applications (e.g., Copilot [2], Gemini [4]), MVSS is increasingly vital.

**Remarks.** Following current MVSS research [50, 65], we highlight: (1) Unless otherwise specified, the default distance between two vectors is the square Euclidean distance. (2) When considering an object set, the vectors within an object have default weights of 1. Hence, the aggregate distance during the index construction is the sum of the distances between each pair of vectors.

### 2.2 Motivation Illustration on HNSW

We begin by outlining HNSW's workflow and its time and space complexity. Then, we explain the operational principles of current MVSS methods, which rely on HNSW. This enables us to establish a baseline and identify its limitations, steering our research.

*2.2.1* ***HNSW Algorithm.*** In graph-based VSS methods, Hierarchical Navigable Small World graph (HNSW) is well-studied in academia [25, 35] and widely deployed in industrial vector databases [9, 12, 59]. Figure 2 (left) delineates HNSW's multiple layers, where the base layer (layer 0) contains all vector data and the upper layer keeps a subset of the lower layer's vectors. In the offline stage, the HNSW index is built incrementally by inserting vectors one by one. Each vector's maximal layer follows an exponentially decaying probability distribution. The newly-inserted vector is treated as a query and finds the top-$c$ closest vertices (candidate neighbors) by greedy search in each layer. This iterates from the vector's maximal layer to the base layer. In each layer, HNSW selects diverse neighbors that include both similar and navigable neighbors based on a heuristic strategy (the navigable neighbors are from different isolated clusters and act as "expressways" for search efficiency [40]). In online query serving, it starts greedy search from the top layer and gets the entry vertex for the next layer. Then, the search moves to the lower layer and repeats the greedy search. This continues until the base layer, where the top-$k$ closest vertices are returned.

In HNSW, the index construction and search share a key parameter: the candidate set size $c$ ($c_1$ and $c_2$, respectively). At each search iteration, the closest vertex to the query is extracted from the candidate set, and its neighbors are visited. Additionally, the maximal number of neighbors $r$ and the result set size $k$ are unique parameters in index construction and search, respectively. Let $c \cdot \theta(n)$ be the scale of visited vertices for obtaining the top-$c$ closest vertices[4]. For each newly-inserted vector, the time complexity of obtaining its $c_1$ candidates is $O(c_1 \cdot d \cdot \theta(n))$, and producing its $r$ neighbors is $O(r \cdot c_1 \cdot d)$, where $d$ is the vector dimension. Hence, the time complexity of constructing an HNSW index on $n$ vectors is $O((\theta(n)+r) \cdot c_1 \cdot d \cdot n)$. The time complexity of executing a query on HNSW is $O(c_2 \cdot d \cdot \theta(n))$. In the HNSW index, we store $r$ neighbor IDs and $d$ values of vector data for each base layer vertex. We omit the higher layers in complexity analysis as their size is negligible. Thus, the space complexity of an HNSW index is $O((r+d) \cdot n)$.

*2.2.2* ***HNSW-Based MVSS.*** For an object set $S$ with $m$ vectors per object, current methods create $m$ vector sets. The $i$-th vector set comprises the $i$-th vectors of all objects, leading to $m$ single-vector HNSW indexes for the $m$ vector sets (Figure 2). The time complexity is $O(\sum_{i=0}^{m-1}(\theta(n)+r) \cdot c_1 \cdot d_i \cdot n)$, or equivalently $O((\theta(n)+r) \cdot c_1 \cdot D_m \cdot n)$, where $d_i$ is the vector dimension in the $i$-th vector set, and $D_m$ is the total dimension of an object's vectors. The space complexity is $O((r \cdot m + D_m) \cdot n)$. Recent advancements, such as `Milvus` [50] and `VBase` [65], optimize search process on the $m$ single-vector HNSW indexes. For a multi-vector query, the time complexity of obtaining $t$ candidate sets is $O(\sum_i^{t-1} c_2 \cdot d_i \cdot \theta(n))$ per iteration for `Milvus`. Thus, the search time complexity of `Milvus` is $O(c_2 \cdot D_t \cdot s \cdot \theta(n)$, where $s$ is the iteration number, and $D_t$ is the total dimension of a query's vectors. Notably, `Milvus` necessitates a larger $c_2$ as the iteration grows, and a reordering procedure to obtain the final results by NRA [22]. On the other hand, `VBase` scans $t$ HNSW indexes in a round-robin manner to avoid repetitive access for a vertex. During each scanning, it directly computes the multi-vector distance between the visited vertices and the query to update the result set. Thus, the search time complexity of `VBase` is $O(c_2 \cdot D_t \cdot t \cdot \theta(n))$.

Current MVSS methods have significantly optimized the search strategies on multiple single-vector HNSW indexes, but they still suffer from subpar efficiency and accuracy, as illustrated in Figure 2 (right). We observe clear limitations of multiple single-vector HNSW indexes for a multi-vector query. This stems from the fact that the navigation and similarity of neighbors are solely based on one vector of an object in each single-vector HNSW.

*2.2.3* ***Baseline.*** A straightforward optimization builds an HNSW index for each vector combination (cf. Definition 3), yielding $2^m - 1$ indexes (the middle of Figure 2). Thus, each multi-vector query matches an HNSW index, and the scanned index depends on the query's vector combination. The search time complexity over such an index is $O(c_2 \cdot D_t \cdot \theta(n))$, lower than current MVSS methods. However, the time complexity of index construction is $O((\theta(n)+r) \cdot c_1 \cdot D_m \cdot 2^{m-1} \cdot n)$ as each vector is included in $2^{m-1}$ combinations (cf. Lemma 2). The space complexity is $O(((2^m-1) \cdot r + 2^{m-1} \cdot D_m) \cdot n)$, since it has $2^m - 1$ indexes (cf. Lemma 1) and each vector of an object is in $2^{m-1}$ indexes.

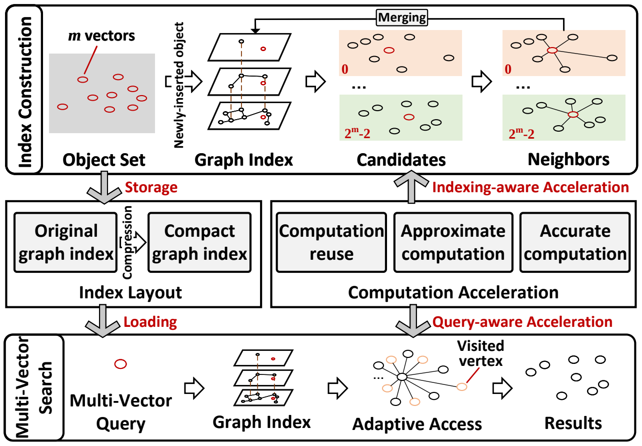


**Figure 3: The execution flow of `MVG`.**

Example 2. Figure 2 (right) shows the evaluation of three HNSW-based MVSS methods on the MIT-States dataset [31]. We use identical HNSW parameters (including $c_1$ and $r$) for index construction. We find that `Baseline` necessitates significantly more index processing time and storage space. However, the current methods exhibit higher query latency and lower recall rate. `Baseline` demonstrates noticeable search efficiency and accuracy superiority.

To summarize, current MVSS methods use single-vector HNSW indexes, which can be efficiently built and have a small index size. However, they exhibit low search efficiency and accuracy. The baseline approach can significantly enhance search accuracy and efficiency but requires very high index construction time complexity and index space complexity. This motivates the design of a new index that offers efficient index processing and compact index layout while maintaining high search efficiency and accuracy.

## 3 MVG: AN OVERVIEW

We introduce a new Multi-Vector Graph (`MVG`) index to optimize index assembly and search procedure, offering efficient index processing, compact index layout, and superior search performance, all at once. Here, we outline the essential components of `MVG`.

### 3.1 Index Construction

Figure 3 (top) illustrates `MVG`'s gradual construction of a graph index via incremental addition of objects, each bearing $m$ vectors and $2^m - 1$ vector combinations (see Definition 3 and Lemma 1). Initially, `MVG` treats each vector combination as a multi-vector query and collects candidate sets by executing MVSS on the existent graph index built by previously-added objects. To gather candidates efficiently, `MVG` implements an indexing-aware acceleration module considering varying computation features. Then, `MVG` performs heuristic neighbor selection on each candidate set, determining the ultimate neighbors per vector combination. We ensure that each vector combination has both navigable and similar neighbors on `MVG`. The indexing-aware acceleration module also assists `MVG` in fast neighbor selection. Unified index management is achieved by merging all neighbor IDs of each object across vector combinations, facilitating single instance sharing of the object's vector data in the index. Grounded on the merged neighbor IDs, `MVG` features an index compression module that compresses neighbor IDs by ID residual.

[4] In the HNSW paper, $\theta(n)$ is roughly $\log(n)$ [40].

This compression ensures lossless recovery of the neighbor IDs without exhaustive computations.

**Remarks.** MVG modularizes the computation acceleration and index compression, allowing easy plug-in/plug-out. We implement multiple optimizations in these modules. Moreover, potential optimizations, beyond this paper, may be integrated into MVG smoothly.

### 3.2 Search Procedure

As shown in Figure 3 (bottom), for any multi-vector query, MVG executes MVSS on a unified graph index. The search process follows the up-and-down greedy route (see §2.2.1). MVG employs an adaptive neighbor access mechanism based on the query's vector combination. Note that we can choose between the original or the compact graph index, depending on performance and memory requirements. The original graph index, though larger, demonstrates better search performance as it avoids decompression. Conversely, the compact graph index, though smaller, exhibits suboptimal search performance compared to the original graph index. Furthermore, MVG deploys a query-aware computation acceleration module considering multi-vector query features.

**Remarks.** The search procedure is adaptive and can accommodate any vector combination of a multi-vector query within a single index. Additionally, we decouple the index layout and computation acceleration from the search procedure, ensuring the flexibility of diverse selections and further optimizations.

## 4 INDEX CONSTRUCTION

Recall that current MVSS indexes use only a single vector of each object to determine neighbors. This limits their MVSS performance. In our baseline (cf. §2.2.3), we build extra indexes with neighbors based on different vector combinations, improving MVSS performance. However, this approach needs high index processing time and large index size, as shown in Figure 2. To solve this, MVG combines all vector combinations into one graph index, where each vertex has diverse neighbors covering all possible object pair relationships. In MVG, a vertex corresponds to an object, unlike the current methods or our baseline where a vertex corresponds to a vector or combination. This paradigm shift presents opportunities to accelerate index processing and compress the index.

### 4.1 Basic Process

In MVG, the graph index is multi-layered and built by adding objects incrementally. For a new object $o$, the maximum layer $l_{max}$ ($\geq 0$) follows an exponentially decaying probability distribution. Let $L$ be the current graph index's top layer. If $l_{max} > L$, MVG updates $L$ to $l_{max}$ and inserts $o$ into all layers from high to low using Algorithm 1. If $l_{max} \leq L$, MVG inserts $o$ into layers from $l_{max}$ to 0 using the same algorithm, but $o$ is not in layers from $L$ to $l_{max}+1$. When inserting $o$ at layer $l$, MVG evaluates all vector combinations to find entry points, candidate neighbors, and final neighbors (lines 2-10). Note that the distance between vertices depends on the vector combination for each traversal (see Example 3). For a vector combination, MVG treats it as a multi-vector query with all weights as 1. To get the entry point at layer $l$, MVG executes a greedy search (see Algorithm 2) at the upper layer, where the nearest neighbor is the entry point (line 3). To find the candidates at layer $l$, MVG also performs a greedy search at layer $l$ from the entry point (line 4). Finally, the ultimate neighbors are those candidates that are not closer to any already-selected neighbor (lines 5-10).

**Algorithm 1:** Insertion of New object at Layer $l$

```
Input: graph index at layer $l$, newly-inserted object $o$, #
       candidates per vector combination $c_1$, maximal #
       neighbors per vector combination $r$
Output: all neighbors of $o$
1  $R \leftarrow \emptyset$;                                 ▷ final set of neighbors
2  for each vector combination in $o$ do
3      $e \leftarrow$ the entry point at layer $l$;          ▷ refer to Algorithm 2
4      $C \leftarrow$ select top-$c_1$ nearest neighbors using $e$ and
        Algorithm 2;                                  ▷ current set of candidates
5      $x \leftarrow$ extract the candidate nearest to $o$ from $C$;
6      $R' \leftarrow x$;                                    ▷ current set of neighbors
7      while $|R'| < r$ and $|C| \neq \emptyset$ do
8          $x \leftarrow$ extract the nearest candidate to $o$ from $C$;
9          if $x$ is closer to $o$ compared to any neighbor from $R$
            then                                      ▷ heuristic neighbor selection
10             $R' \leftarrow R' \cup x$;
11     $R \leftarrow R \cup R'$;                              ▷ merge all neighbors
12 return $R$
```

Definition 3. ***Vector Combination.*** *Given an object with $m$ vectors, a vector combination is an arrangement that selects from 1 to $m$ vectors out of the total $m$ vectors.*

Lemma 1. *An object with $m$ vectors has $2^m - 1$ vector combinations.*

Proof. (*Sketch.*) For an object with $m$ vectors, each vector can be selected or not, yielding $2^m$ subsets. Excluding the empty set, we get $2^m - 1$ nonempty combinations. Due to space limitations, we include the detailed proof in our technical report [8]. □

Example 3. Let each object have two vectors. It has three vector combinations: 0, 1, and 0-1. Thus, MVG performs three traversals, using each vector combination as a multi-vector query to obtain the final neighbors. It is worth noting that each combination involves distinct components for distance calculation. For the first, the inter-object distance is the distance of their first vectors. For the second, it is the distance of their second vectors. For the third, it is the sum of the distances of their first and second vectors.

**Complexity Analysis.** MVG explores $c_1 \cdot \theta(n)$ [4] vertices to find top-$c_1$ candidates. It visits $c_1 \cdot r$ neighbors to generate $r$ final neighbors. As per Lemma 2, each vector in an object joins $2^{m-1}$ traversals (lines 2-11 in Algorithm 1). Let $D_m$ be the total dimension of all vectors in an object. The insertion time complexity of an object is $O((\theta(n) + r) \cdot c_1 \cdot D_m \cdot 2^{m-1})$. The space cost of an object in the index is $O((2^m - 1) \cdot r + D_m)$.

Lemma 2. *For an object $o$ with $m$ vectors, each vector is included in $2^{m-1}$ vector combinations.*

Proof. (*Sketch.*) Given a set of $m$ vectors, pick one vector. There are $2^{m-1}$ combinations with this vector, as each of the other $m - 1$ vectors can be included or not. Due to space limitations, the detailed proof can be found in our technical report [8]. □

## 4.2 Indexing-Aware Acceleration

Multi-vector distance computation consumes much time in index construction. It is required between the inserted object and each visited vertex for all vector combinations to get entry points or candidates. It is also needed between selected neighbors and candidates to obtain final neighbors. Our evaluation shows that these computation operations take up to *77.8%* of the construction time on the MIT-States dataset. Obviously, distance computation is a major bottleneck, limiting the construction efficiency. We investigate the features of different multi-vector distance calculations in index construction and identify three classes of computation. Therefore, we propose acceleration methods based on these features to improve the construction process without compromising index quality.

*4.2.1* ***Computation Reuse***. For each inserted object $o$, MVG may visit vertex $x$ multiple times for different vector combinations. For example, MVG computes $\delta(\boldsymbol{o_0}, \boldsymbol{x_0})$ for vector 0, and again computes $\delta(\boldsymbol{o_0}, \boldsymbol{x_0})$ for vectors 0-1 to obtain $g(\delta(\boldsymbol{o_0}, \boldsymbol{x_0}), \delta(\boldsymbol{o_1}, \boldsymbol{x_1}))$. MVG avoids such redundant computations by reusing $\delta(\boldsymbol{o_0}, \boldsymbol{x_0})$. It first finds the final neighbors for the combinations with one vector, and caches the single-vector distance between $o$ and visited vertices. For subsequent combinations with more than one vector, MVG checks the cached distance, and if matched, fetches the value to speed up multi-vector computation. Computation reuse improves the construction efficiency by optimizing $D_m$ in the complexity formula. On the MIT-States dataset, this optimization reduces computations for each inserted object by *72.1%* on average.

Lemma 3. *The computation reuse does not impact the index quality.*

Proof. (*Sketch.*) Let the current vector combination have $m'$ vectors. To get the multi-vector distance for a visited vertex $x$, we compute each $\delta(\boldsymbol{o_i}, \boldsymbol{x_i})$ for $0 \leq i \leq m'-1$. Computation reuse fetches some $\delta(\boldsymbol{o_i}, \boldsymbol{x_i})$ from cache, skipping their computation. Please refer to our technical report for the detailed proof [8]. □

**Optimization.** We observe that most multi-vector distance calculations can be simplified by the cached single-vector distance (e.g., up to *77.6%* on the MIT-States dataset), indicating a large overlap between vertex access for one-vector and multi-vector combinations. Note that the state-of-the-art graph-based vector index only requires 50% similar neighbors [54]. Hence, we can directly generate the candidates for multi-vector combinations from the visited vertices while getting the candidates for one-vector combinations. This further reuses distance computations, leading to more efficient index construction while preserving the index quality. The insertion time complexity of an object becomes $O((\theta(n) \cdot m + r \cdot 2^{m-1}) \cdot c_1 \cdot D_m)$, which is lower than before.

*4.2.2* ***Approximate Computation***. Most distance values during index construction are compared with a threshold $T$. For example, MVG adds a visited vertex $x$ to the candidate set $C$ if the distance between $x$ and the inserted object $o$ is less than $T$, which is the farthest distance from $o$ for the elements in $C$. Likewise, all distance computations for final neighbor acquisition require comparisons. On the MIT-States dataset, comparison computations account for *90.4%* of the total, and *83.4%* of these can accomplish a right comparison by using partial distances. An accurate distance, especially for multi-vector case, is often unnecessary for comparison. A simple optimization is to incrementally scan a vector combination's dimensions and stop when the partial distance exceeds $T$ or all dimensions have been scanned [25, 44]. This reduces the time complexity of distance computation. However, for high-dimensional vector combinations, values in different dimensions contribute differently to the final distance; incremental scanning from low to high dimension may scan many low-contribution values.

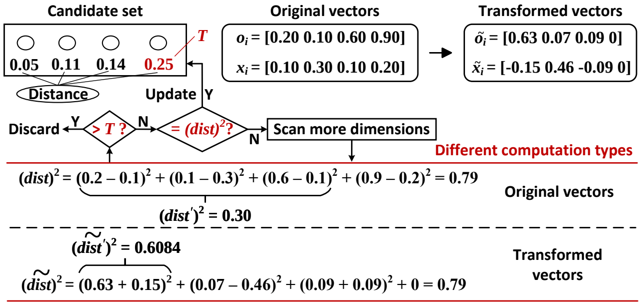

Candidate set
0.05
0.11
0.14
0.25
T
Distance
Update
Y
Original vectors
$o_i$ = [0.20 0.10 0.60 0.90]
$x_i$ = [0.10 0.30 0.10 0.20]
Transformed vectors
$\tilde{o}_i$ = [0.63 0.07 0.09 0]
$\tilde{x}_i$ = [-0.15 0.46 -0.09 0]
Discard
Y
> T ?
N
= (dist)²?
N
Scan more dimensions
Different computation types
$(dist)^2 = \underbrace{(0.2 - 0.1)^2 + (0.1 - 0.3)^2 + (0.6 - 0.1)^2}_{(dist')^2 = 0.30} + (0.9 - 0.2)^2 = 0.79$
Original vectors
$(\widetilde{dist})^2 = \overbrace{(0.63 + 0.15)^2}^{(\widetilde{dist}')^2 = 0.6084} + (0.07 - 0.46)^2 + (0.09 + 0.09)^2 + 0 = 0.79$
Transformed vectors

**Figure 4: An example of approximate computation.**

Example 4. In Figure 4, the current distance threshold $T$ in the candidate set is 0.25. To add a visited vertex $x$ to the candidate set, the conventional method computes the accurate square distance $(dist)^2$ and compares it with $T$. If $(dist)^2 > T$, $x$ is discarded; otherwise, $x$ updates the candidate set. This requires scanning four dimensions to calculate $(dist)^2$. Alternatively, computing the partial distance $(dist')^2$ by incrementally scanning vectors avoids scanning all dimensions. Here, scanning the first three dimensions suffices for a correct comparison. However, checking the fourth dimension directly, i.e., $(dist')^2 = (0.9 - 0.2)^2 = 0.49$, needs only one dimension for a correct comparison. This is because the last dimension contributes the most. Hence, scanning the important dimensions first is vital for better efficiency.

MVG transforms the vector values by their relative importance and assigns the higher importance values to the lower dimensions of a vector combination, to prioritize scanning the high-contribution dimensions. For a set of objects $S$, each object $o$ has $m$ vectors, and the $i$-th vectors form the $i$-th vector set $S_i$. MVG treats $S_i$ as an $n \times d_i$ matrix $\boldsymbol{S_i}$, where $n$ is the number of objects and $d_i$ is the vector dimension of $S_i$. The vector transformation in $S_i$ is as follows.

**(i)** MVG calculates the mean vector $\boldsymbol{\mu_i}$ for all vectors in $S_i$ as

$$\boldsymbol{\mu_i} = \frac{1}{n} \sum_{\boldsymbol{o_i} \in S_i} \boldsymbol{o_i} \quad . \tag{4}$$

**(ii)** MVG centralizes the vectors in $S_i$ to get the centered vector set $\hat{S_i}$, with its corresponding matrix $\hat{\boldsymbol{S_i}}$ computed as

$$\hat{\boldsymbol{S_i}} = \boldsymbol{S_i} - \boldsymbol{1}\boldsymbol{\mu_i}^\top \quad , \tag{5}$$

where $\boldsymbol{1}$ is an $n$-dimension vector with all elements equal to 1.

**(iii)** MVG calculates the covariance matrix $\boldsymbol{\Sigma_i}$ of the centered data as

$$\boldsymbol{\Sigma_i} = \frac{1}{n}(\hat{\boldsymbol{S_i}}^\top \hat{\boldsymbol{S_i}}) \quad . \tag{6}$$

**(iv)** MVG computes the eigenvalues $(\lambda_0, \cdots, \lambda_{d_i - 1})$ and the eigenvectors $(\boldsymbol{z_0}, \cdots, \boldsymbol{z_{d_i-1}})$ of $\boldsymbol{\Sigma_i}$. It ensures each eigenvector is normalized to unit length, and is orthogonal to all other eigenvectors.

**(v)** MVG sorts the eigenvalues in descending order and constructs a $d_i \times d_i$ projection matrix $\boldsymbol{M_i}$ with the eigenvectors arranged in columns, according to the ordered eigenvalues:

$$\boldsymbol{M_i} = (\boldsymbol{z_0}, \boldsymbol{z_1}, \cdots, \boldsymbol{z_{d_i-1}}) \quad , \tag{7}$$

where $\boldsymbol{z_0}$ corresponds to the largest eigenvalue $\lambda_0$, $\boldsymbol{z_1}$ corresponds to the second largest eigenvalue $\lambda_1$, and so on.

**(vi)** MVG employs $\boldsymbol{M_i}$ to transform each vector in $S_i$, forming the transformed data set $\tilde{S_i}$, with the same dimensionality. For a vector $\boldsymbol{o_i}$, the transformed vector $\tilde{\boldsymbol{o}}_{\boldsymbol{i}}$ can be calculated as:

$$\tilde{\boldsymbol{o}}_{\boldsymbol{i}} = {\boldsymbol{M_i}}^\top \boldsymbol{o_i} \quad . \tag{8}$$

The above steps are conducted on all $S_i$ for $0 \leq i \leq m-1$. In our implementation, we randomly sample some vectors (1%) from $S_i$ to establish the projection matrix $\boldsymbol{M_i}$. As per our experiments, this strategy enhances the efficiency of vector transformation, while preserving a comparable vector transformation quality.

Lemma 4. *The projection matrix $\boldsymbol{M_i}$ is orthonormal.*

Proof. (*Sketch.*) $\boldsymbol{M_i}$ has orthogonal, normalized eigenvectors as columns. Their dot products are zero or one. Due to the space limitation, we put the detailed proof in our technical report [8]. □

Lemma 5. *Transforming vectors $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ in $S_i$ by the orthonormal matrix $\boldsymbol{M_i}$ does not alter the distance between them.*

Proof. (*Sketch.*) An orthogonal matrix holds inverse equal to transpose. Transformation by $\boldsymbol{M_i}$ keeps vector length and distance. Kindly refer to our technical report for the detailed proof [8]. □

In $\tilde{S_i}$, we measure the significance of the $j$-th dimension by its variance $\sigma_j^2$. Larger $\sigma_j^2$ dominates the distance magnitude, indicating a more distinct position in the $j$-th dimension for vector pairs.

Lemma 6. *With $\lambda_j$ as the $j$-th largest eigenvalue of $\Sigma_{\boldsymbol{i}}$, and $\sigma_j^2$ as the variance of the $j$-th dimension of $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$, it holds: $\sigma_j^2 = \lambda_j$.*

Proof. (*Sketch.*) The transformed data $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$ are zero-mean. We have $Var(\tilde{S_i}) = E(\tilde{S_i}^2)$ by the identity $Var(\tilde{S_i}) = E(\tilde{S_i}^2) - (E(\tilde{S_i}))^2$, where $Var(\tilde{S_i})$ is the variance of $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$. $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}[j] = {\boldsymbol{M_i}}^\top[j]\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$ is the $j$-th dimension of $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$. We have $\sigma_j^2 = {\boldsymbol{M_i}}^\top[j]\Sigma_{\boldsymbol{i}}({\boldsymbol{M_i}}^\top[j])^\top$. $({\boldsymbol{M_i}}^\top[j])^\top$ is the $j$-th eigenvector of $\Sigma_{\boldsymbol{i}}$ with eigenvalue $\lambda_j$. Due to space limitations, we put the detailed proof in our technical report [8]. □

Theorem 1. *After transforming $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ using $\boldsymbol{M_i}$, the contribution to the distance between $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ is non-increasing from the first dimension to the last dimension.*

Proof. (*Sketch.*) $\tilde{\boldsymbol{o}}_{\boldsymbol{i}}$ and $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ are projections of $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ by $\boldsymbol{M_i}$. Distance contribution of dimension $j$ is:

$$(\Delta_j)^2 = (\tilde{\boldsymbol{o}}_{\boldsymbol{i}}[j] - \tilde{\boldsymbol{x}}_{\boldsymbol{i}}[j])^2 \quad . \tag{9}$$

$\Delta_j$ is proportional to the square root of the eigenvalue. Thus, $(\Delta_j)^2 \geq (\Delta_{j+1})^2$. Refer to our full report for the detailed proof [8]. □

For each inserted object $o$, MVG transforms each vector in $o$ by the corresponding orthogonal matrix. When computing the distance for comparison, MVG incrementally scans the transformed vectors, computing higher-priority values first. The transformation allows the direct calculation of original accurate distances on the transformed vectors (see Lemma 5). This optimization improves index construction efficiency by optimizing the $D_m$ term. Our evaluation on the MIT-States dataset shows a *22.8%* reduction in the number of dimensions required for distance computations per inserted object.

Lemma 7. *The optimization of approximate computation does not impact the index quality.*

Proof. (*Sketch.*) According to Lemma 5, orthogonal transformation preserves the original distance. The partial distance $\widetilde{dist}'$ on transformed vectors ranges from 0 to $dist$. We can always ensure a correct comparison by using $\widetilde{dist}'$, whether $dist > T$ or $dist \leq T$. Kindly refer to our technical report for the detailed proof [8]. □

Example 5. Figure 4 depicts an example of two transformed vectors obtained by applying $\boldsymbol{M_i}$ to the original vectors. In the transformed vectors, the first dimension holds the most significance, followed by the second, and so on. In this example, one dimension is scanned on the transformed vectors to compute the partial distance, yielding $(\widetilde{dist'})^2 = 0.6084$. Since this value exceeds $T$, $x$ is discarded without further scanning. Note that the accurate distance remains unchanged by the transformation.

#### 4.2.3 *Accurate Computation*.

In situations where a candidate set remains unfilled, it is necessary to directly compute the accurate distance between the inserted object $o$ and a visited vertex $x$. The following formula establishes a relationship between Euclidean distance and inner product:

$$||\boldsymbol{o_i} - \boldsymbol{x_i}||^2 = ||\boldsymbol{o_i}||^2 + ||\boldsymbol{x_i}||^2 - 2 \cdot \boldsymbol{o_i} \cdot \boldsymbol{x_i} \quad . \tag{10}$$

Motivated by this, MVG reformulates distance computation as fast inner product computation. It calculates and stores the square $l_2$-norms ($||\boldsymbol{o_i}||^2$) of the vectors in $o$. When computing the accurate distance between $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$, it first calculates the inner product $\boldsymbol{o_i} \cdot \boldsymbol{x_i}$. The square Euclidean distance is then calculated by Equation 10 using $||\boldsymbol{o_i}||^2$ and $||\boldsymbol{x_i}||^2$ (the square $l_2$-norms of all vectors in each object are stored upon insertion). Therefore, MVG obtains the exact distance solely by computing the inner product, optimizing the $D_m$ term in the time complexity formula. Our evaluation on the MIT-States dataset demonstrates that, on average, *9.6%* of distance computations per inserted object can be accelerated using this way.

Lemma 8. *The optimization of accurate computation does not impact the index quality.*

Proof. According to Equation 10, this lemma is evident. □

## 4.3 Index Compression

To unify index management, MVG consolidates all data into a single index. As depicted in Figure 5, MVG sequentially stores $m$ vectors, $m$ $l_2$-norms, and the neighbor IDs of $2^m - 1$ vector combinations for each vertex. It allocates equal storage space for each object to store its vector data and $l_2$-norms, facilitating sequential access through offset. Note that the vector data and $l_2$-norms of each vertex are stored only once. The $l_2$-norms expedite accurate computation in §4.2.3. Regarding neighbor IDs, different layers are stored separately, following the same format. Each layer contains $2^m - 1$ neighbor ID lists from all vector combinations. Each list maintains a consistent size for direct access by offset. This layout enables quick data access when answering a multi-vector query, which involves access to

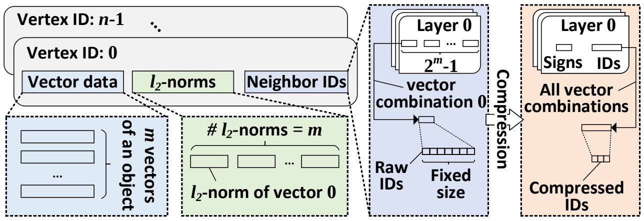


**Figure 5: Overview of index layout.**

vector data and $l_2$-norms by vertex ID and the access of neighbor IDs by both the vertex ID and the vector combination of the query.

Despite the careful design of this index layout, the number of neighbor IDs per vertex becomes substantial due to numerous vector combinations. This escalation raises the storage cost, particularly for large-scale data. To mitigate this issue, MVG incorporates a neighbor ID compression module by sorting IDs and appending a sign for each ID to indicate the vector combination. Only $\lceil \log_2(2^m - 1) \rceil$ bits are needed for each sign. Next, we explore three compression algorithms for neighbor ID lists in MVG.

### 4.3.1 *Sequential ID (Seq-ID)*.

Seq-ID compresses a neighbor ID list by retaining the first ID and computing the difference of consecutive IDs. That is, the second ID minus the first ID, the third ID minus the second ID, and so on, until the last ID minus the penultimate ID. Resultantly, MVG stores only the first ID and the differences. Note that MVG allocates equal bits for each difference, enabling easy access. To recover a raw ID, Seq-ID takes a time complexity of $O(h)$, where $h$ is the neighbor list length.

Example 6. Figure 6(a) provides an illustration of compressing an ordered neighbor ID list by Seq-ID. Before compression, nine IDs require 288 bits, as each ID is 32 bits. With Seq-ID, the maximum difference is 90, necessitating 7 bits per difference and 32 bits for the first ID. Consequently, 88 bits are adequate to store the list, resulting in saved bits at the cost of decompression. In the worst case, all IDs need to be decompressed to access the last raw ID.

### 4.3.2 *Minimum ID (Mini-ID)*.

Mini-ID eliminates the need to decompress irrelevant IDs for a specific ID access. It achieves this by subtracting the first ID from all other IDs to obtain the differences. Consequently, MVG stores the first ID and the differences. Mini-ID allows access to a raw ID in $O(1)$ time complexity. However, it requires more storage space than Seq-ID.

Example 7. Figure 6(b) depicts the compression of a neighbor ID list using Mini-ID. The maximum difference is 390, requiring 9 bits per difference. Consequently, 104 bits are needed to store the list, which is more than Seq-ID. However, this method enables direct raw ID recovery using the first raw ID and the difference.

### 4.3.3 *Median ID (Med-ID)*.

To balance decompression efficiency and compression ratio, Med-ID subtracts all smaller IDs from the median ID and subtracts the median ID from all larger IDs to obtain the differences. Therefore, MVG stores the median ID and the differences. Similar to Mini-ID, Med-ID recovers a raw ID in $O(1)$ time complexity. Due to its smaller differences compared to Mini-ID, Med-ID leads to less storage space.

Example 8. As shown in Figure 6(c), the median ID is 200, and the maximum difference is 250. Thus, each difference storage requires 8 bits, and the total number of bits to store such a neighbor ID list is 96. Med-ID achieves a better trade-off between decompression efficiency and compression ratio.

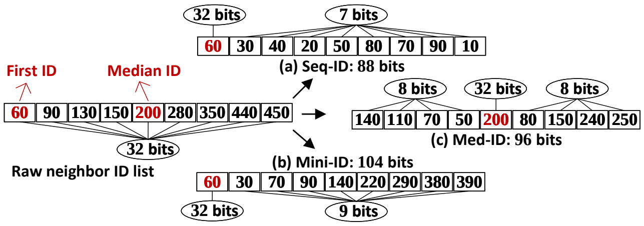


**Figure 6: Illustration of neighbor ID list compression.**

Lemma 9. *The compression algorithms are lossless.*

Proof. Since any raw ID can be accurately recovered using the stored raw ID and the differences between them, we can conclude that the compression process is indeed lossless. □

In MVG, index compression reduces storage space by optimizing the term $(2^m - 1) \cdot r$ in the complexity formula. For instance, on the MIT-States dataset, the space saving of neighbor lists can reach values of *52.1%*, *51.7%*, and *51.9%*, for Seq-ID, Mini-ID, and Med-ID, respectively. It is worth noting that the decompression process, especially for Mini-ID and Med-ID, employs rapid bit operations and minimally affects search efficiency for multi-vector queries.

# 5 SEARCH PROCEDURE

In this section, we delve into the search procedure on the unified index, encompassing both the original index and the compact index (refer to Figure 3). We outline the basic process and discuss optimizations with query-aware acceleration. Note that MVG accommodates any vector combination in a multi-vector query.

## 5.1 Basic Process

Recall that MVG is a multi-layer graph structure. The search starts at the top layer, using the greedy route to find the nearest vertex. Each layer's nearest vertex is the entry point for the next layer. This repeats until the base layer, where the search follows Algorithm 2. Finally, the top-$k$ results are returned. We emphasize that the search in the upper layers follows Algorithm 2 with $c_2 = k = 1$. Next, we describe the search at a specific layer by Algorithm 2.

The search process begins by initializing the candidate set $C$ and the result set $R$ with the entry point $e$ and tracks the visited elements with a set $H$ to avoid repetitive access. It extracts the nearest vertex $x$ to the query $q$ from $C$ and the furthest vertex $y$ to $q$ from $R$. If $y$ is closer to $q$ than $x$, the search terminates, as all vertices in $R$ are closer to $q$ than those in $C$. Otherwise, it visits each unvisited neighbor $p$ of $x$ and updates $C$ and $R$ with $p$. This process repeats until $C$ is empty or $y$ is closer to $q$ than $x$. Throughout this process, the distance between $q$ and a visited vertex is computed by Equation 3, and the neighbor access adapts to the vector combination of $q$. According to performance and memory requirements, we can choose between the original or the compact index to execute the search (cf. Figure 3). In the original index, the neighbor IDs are obtained by the offset. In the compact index, the neighbor ID's decompression and access are determined by the neighbor sign bit. The decompression depends on the compression algorithms. For example, to decompress an ID, Seq-ID may require several ID decompressions, but Mini-ID and Med-ID need only one.

**Algorithm 2:** Search Top-$k$ Neighbors At Layer $l$

```
Input: graph index at layer $l$, multi-vector query $q$, entry
       point $e$ at layer $l$, # candidates $c_2$, # results $k$
Output: top-$k$ neighbors of $q$
$H \leftarrow e$;                                    ▷ set of visited elements
$C \leftarrow e$;                                    ▷ candidate set
$R \leftarrow e$;                                    ▷ result set
while $|C| > 0$ do
    $x \leftarrow$ extract nearest vertex to $q$ from $C$;
    $y \leftarrow$ get furthest vertex to $q$ from $R$;
    if $y$ is closer to $q$ than $x$ then
        break;   ▷ all vertices in $R$ are closer than those in $C$
    for each neighbor $p$ of $x$ at layer $l$ do
        if $p \notin H$ then
            $H \leftarrow H \cup p$;
            $y \leftarrow$ get furthest vertex from $R$ to $q$;
            if $|R| < k$ or $p$ is closer to $q$ than $y$ then
                $C \leftarrow C \cup p$; $R \leftarrow R \cup p$;
                if $|R| > k$ then
                    remove furthest element to $q$ from $R$;
return $R$
```

## 5.2 Query-Aware Acceleration

In Algorithm 2, there are two types of distance computations. The first type aims to compare with a threshold $T$ ($T$ is the distance between $q$ and the furthest vertex in $R$), and the second type requires an accurate distance value (when $R$ is not filled). On the MIT-States dataset, distance computation accounts for *91.3%* of the search time, with *82.1%* for comparison and *17.9%* for accurate distance. Clearly, we can still utilize the optimizations of approximate and accurate computation from §4.2.2 and §4.2.3.

MVG also employs a new acceleration optimization, exploiting the distinct aspect of a multi-vector query. This idea stems from the observation that different vectors carry different weights in a multi-vector query. Thus, we infuse the weight information into the approximate computation, prioritizing the scanning of vectors with higher weights. Specifically, MVG orders the distance computation for each vector pair by the weights when computing the aggregation distance between the query $q$ and a visited vertex. Vectors with higher weights get precedence in distance calculation. According to our evaluation, this optimization saves *29.5%* computation on the MIT-States dataset, further reducing the computations compared to original approximate computation method.

Lemma 10. *The query-aware acceleration does not result in loss of search accuracy.*

Proof. By Lemma 7 and Lemma 8, approximate or accurate computation does not impact the result. The weight-based optimization only changes the computation order. Therefore, all query-aware acceleration methods are lossless. □

**Complexity Analysis.** To obtain the top-$k$ nearest neighbors, MVG visits vertices on a $c_2 \cdot \theta(n)$ scale. Let $D_t$ be the total dimension of all vectors in a multi-vector query. The time complexity of getting top-$k$ results is $O(\theta(n) \cdot c_2 \cdot D_t)$. The query-aware acceleration improves search efficiency by optimizing the $D_t$ term.

**Table 2: Statistics of experimental datasets.**

| Datasets | $n$ | $m$ / $t$ | $D_m$ | #$q$ | $w_i$ |
|---|---|---|---|---|---|
| Recipe | 1.3M | 2 | 2,048 | $10^4$ | 0.1/0.2 |
| MIT-States | 2.1M | 6 | 3,456 | $10^3$ | 0.2/0.15/0.2/0.1/0.2/0.15 |
| CelebA | 1M | 4 | 2,304 | $10^3$ | 0.2/0.3/0.4/0.1 |
| FashionIQ | 1M | 2 | 1,024 | $10^3$ | 0.7/0.3 |
| MS-COCO | 1M | 3 | 1,536 | $10^3$ | 0.4/0.3/0.3 |
| Shopping | 1M | 2 | 1,024 | $10^3$ | 0.7/0.3 |
| MIT-States+ | 16M | 6 | 3,456 | $10^3$ | 0.2/0.15/0.2/0.1/0.2/0.15 |

# 6 EXPERIMENTS

To conduct a comprehensive evaluation of MVG, we carry out the following experiments: (i) Multi-Vector Query Performance (**§6.2**), (ii) Efficiency of Index Construction (**§6.3**), (iii) Index Size (**§6.4**), (iv) Query Workloads (**§6.5**), (v) Scalability (**§6.6**), and (vi) Ablation Study (**§6.7**). All source codes, datasets, and additional evaluations can be accessed publicly at: https://github.com/ZJU-DAILY/MVG.

## 6.1 Experimental Setting

*6.1.1* ***Datasets.*** We employ six real-world datasets, each featuring objects with more than two vectors from different modalities or encoders. As portrayed in Table 2, these datasets present variability in the number of vectors ($m$), dimensions ($D_m$), and scale ($n$). The ground truth is derived from executing a brute-force search using a batch of multi-vector queries. Since the original data scale is small (e.g., the scale of CelebA is merely 200K), we expand the datasets using generative models [21, 60], which allows us to create additional samples from the learned distribution of the real data.

*6.1.2* ***Query Type.*** Multi-vector queries contain the same number of vectors as objects ($t = m$) by default, with each vector having a distinct weight. We evaluate the ability to answer any query by testing various weight configurations on a specific dataset. We also account for single-vector queries, where other vectors carry zero weight. This reflects realistic situations where users may submit different number of vectors ($1 \sim m$) or favor certain vectors ($w_i$).

*6.1.3* ***Compared Methods.*** We evaluate three existing methods, one baseline we devise, and two variants of MVG. All methods employ HNSW algorithm. **(i)** VBase. It leverages index scanning optimization with multiple single-vector indexes [65]. It is developed by Microsoft and is the state-of-the-art MVSS method. **(ii)** Milvus. It applies candidate merging optimization with multiple single-vector indexes [50], released by Zilliz. **(iii)** Merging. It is a previous MVSS method for hybrid queries [51, 66] and also depends on multiple single-vector indexes. **(iv)** Baseline. Our naive optimization, which builds an index for each vector combination in objects (see §2.2.3). **(v)** MVG*. Our MVSS method that adopts indexing-aware and query-aware acceleration, excluding index compression. **(vi)** MVG. Our optimal MVSS method that utilizes all proposed techniques. For index compression, it employs Med-ID algorithms.

*6.1.4* ***Performance Measure.*** We measure the search efficiency and accuracy by *Latency* and *Recall*, respectively. For $R$, the top-$k$

Table 3: Multi-vector query performance (*Latency*: ms).

| Methods | Recipe |  | MIT-States |  | CelebA |  | FashionIQ |  | MS-COCO |  | Shopping |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | *recall* | *latency* | *recall* | *latency* | *recall* | *latency* | *recall* | *latency* | *recall* | *latency* | *recall* | *latency* |
| Merging | 0.38 | 3.0 | 0.36 | **4.2** | 0.51 | 9.5 | 0.31 | 5.6 | 0.34 | 16.8 | 0.12 | **0.6** |
| Milvus | 0.81 | 1,079.6 | 0.38 | 110.9 | 0.65 | 4,312.5 | 0.79 | 1,280.3 | 0.58 | 2,779.3 | 0.43 | 317.1 |
| VBase | 0.98 | 40.0 | 0.97 | 23.1 | **0.99** | 159.2 | 0.98 | 61.4 | 0.92 | 91.8 | 0.94 | 48.5 |
| Baseline | **0.99** | 3.3 | **0.99** | 8.1 | **0.99** | 7.8 | **0.99** | 9.1 | **0.93** | 18.8 | 0.94 | 10.7 |
| MVG* | **0.99** | **2.1** | **0.99** | 5.5 | **0.99** | **5.6** | **0.99** | **4.1** | **0.93** | **10.8** | **0.95** | 8.3 |
| MVG | **0.99** | 3.0 | **0.99** | 6.7 | **0.99** | 6.5 | **0.99** | 4.7 | **0.93** | 11.6 | **0.95** | 11.1 |

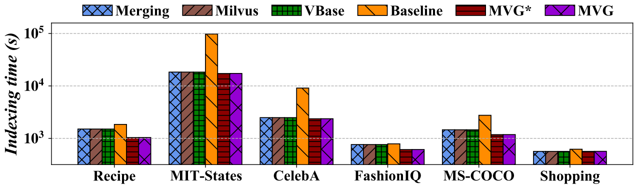


Figure 7: Efficiency of index construction.

result set from an MVSS method, the *Recall* is:

$$Recall = \frac{|R \cap R'|}{k} \quad , \tag{11}$$

where $R'$ represents the exact result from a brute-force search. We set $k$ to 10 by default, unless stated otherwise.

*6.1.5* ***Environment Configuration.*** We run the experiments on a machine with an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 128GB of memory, on CentOS 7.9. All methods are coded in C++ and compiled with g++ 4.8 and -O3 optimization. We use OpenMP for parallel index construction, with 48 threads for all methods. For query execution, we use one thread for all methods. This thread setup follows common practices in vector search research [24, 54].

## 6.2 Multi-Vector Query Performance

Table 3 shows the multi-vector query performance of different methods. We observe that: **(i)** MVG* balances accuracy and efficiency optimally. For example, against the leading VBase, MVG* cuts latency by up to 96.5%, with a higher recall rate. **(ii)** MVG has higher latency than MVG*. This is due to the index compression module of MVG, which needs an extra decompression step for multi-vector queries. **(iii)** The last trio of methods outperforms the first trio in the *Recall*-vs-*Latency* trade-off. This highlights the importance of using multi-vector neighbor relationships. **(iv)** Current search strategy optimizations also improve performance effectively. For instance, compared to Merging, Milvus enhances query accuracy, and VBase beats Milvus in both accuracy and efficiency.

## 6.3 Efficiency of Index Constuction

Figure 7 illustrates the index construction time for different methods. We adjust the parameters for index construction to achieve optimal search performance. We observe that: **(i)** MVG* has the fastest construction time, due to its indexing-aware acceleration component. MVG takes slightly longer than MVG*, because of the extra compression step. **(ii)** During parameter adjustment, we find that single-vector indexes (including Merging, Milvus, VBase) require a larger candidate neighbor set ($c_1$) and a bigger maximal number of neighbors ($r$) than multi-vector indexes (including MVG* and MVG), to reach optimal search performance. **(iii)** Baseline's construction time is much higher than others, as it needs to build indexes for many vector combinations. **(iv)** The existing three methods use single-vector indexes and have identical construction times.

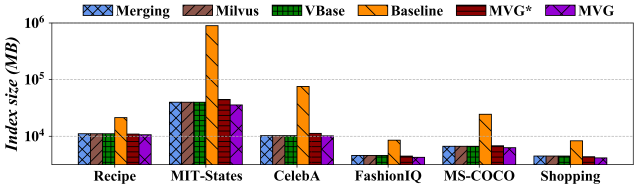
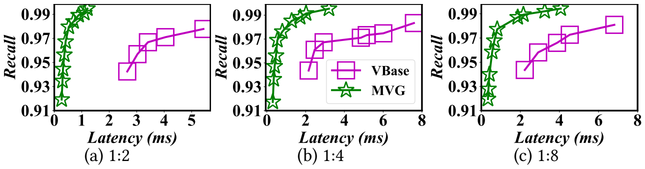


Figure 8: Index size.



(a) 1:2
(b) 1:4
(c) 1:8

Figure 9: Different weight ratios on MIT-States.

## 6.4 Index Size

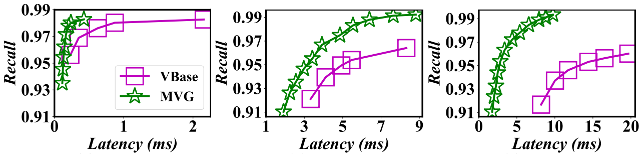
Figure 8 delineates the index size of different methods, following the same parameter setup from §6.3. The results reveal that: **(i)** MVG exhibits the smallest index size, due to its index compression component. **(ii)** Baseline has the largest index size, as each vector combination needs an index, multiplying the indexes. **(iii)** Our multi-vector index, despite more neighbors, shows a smaller index size than the current single-vector indexes, due to the well-designed index layout and neighbor ID list compression. **(iv)** All existing methods use the single-vector indexes, having the same index size.

## 6.5 Query Workloads

In this section, we compare MVG and VBase across various query workloads. To ensure impartiality, identical HNSW settings are maintained for both. We exclude other extant methods as their pronounced limitations in efficiency (Merging) and accuracy (Milvus).

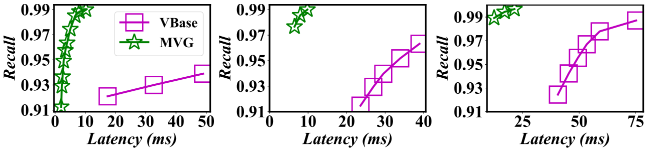
*6.5.1* ***Weight Ratio.*** We modulate the weight ratio within each dual-vector query ($t = 2$) on MIT-States and assess the MVSS performance of MVG and VBase. Figure 9 shows that MVG surpasses VBase consistently, regardless of the weight ratios. Notably, MVG performs better when the weight bias is minimal (Figure 9(a)). A previous study suggests that minor weight bias is common in real-world conditions [51].

*6.5.2* ***Number of Query Vectors (t).*** In Figure 10, we examine VSS ($t = 1$) and MVSS ($t > 1$) performance. The results indicate that MVG achieves superior performance in both query cases. Notably, the performance margin between MVG and VBase widens as $t$ increases.



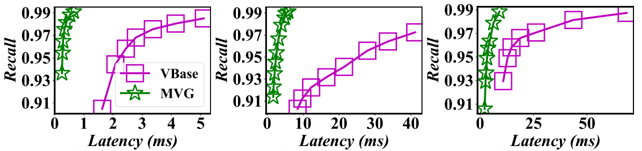
(a) $t$ = 1 (b) $t$ = 3 (c) $t$ = 5

**Figure 10: Different number of query vectors on MIT-States.**



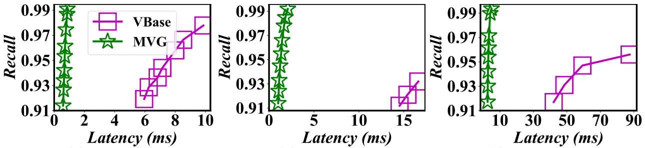
(a) $k$ = 1 (b) $k$ = 100 (c) $k$ = 200

**Figure 11: Different number of search results on MIT-States.**



(a) $m$ = 2 (b) $m$ = 4 (c) $m$ = 6

**Figure 12: Different number of vectors within an object.**



(a) $n$ = 4$M$ (b) $m$ = 8$M$ (c) $m$ = 16$M$

**Figure 13: Different data scales on MIT-States+.**

*6.5.3* ***Number of Results (k).*** Figure 11 shows MVSS performance of MVG and VBase on MIT-States for different $k$. MVG consistently outperforms VBase for any $k$. Additionally, whether it is a smaller $k$ ($k$ = 1) or a larger $k$ ($k$ = 200), VBase experiences a drastic performance reduction. In constrat, MVG showcases robust applicability under such extreme $k$ values.

## 6.6 Scalability

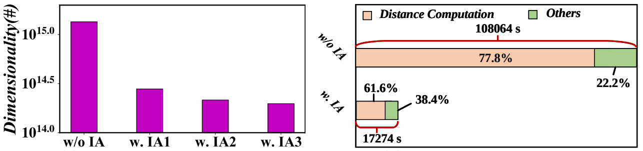
In this section, we test MVG's scalability on MIT-States with different numbers of vectors per object and data scales. For fairness, we use identical HNSW settings for both MVG and VBase.

*6.6.1* ***Number of Vectors in an Object (m).*** In real-world scenarios, various use cases may have different numbers of vectors, $m$, in an object. As Figure 12 depicts, we adjust $m$ and assess the MVSS performance on MIT-States. The results indicate that MVG outperforms, particularly in the high recall rate region, demonstrating its robust scalability to the number of vectors within an object.

*6.6.2* ***Data Scale (n).*** Figure 13 illustrates MVSS performance of MVG and VBase for different $n$. We note that the latency of VBase increases linearly with $n$. In contrast, MVG displays only a minor latency elevation even with large $n$ values, reducing latency by up to 96.5% when $n$ is 16 million. Along with this, MVG also showcases a significant accuracy superiority. These results confirm the superior scalability of MVG in large-scale situations.

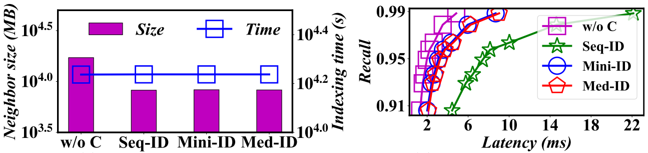
## 6.7 Ablation Study

In this section, we verify the effect of each standalone technique by conducting an ablation study on MIT-States.



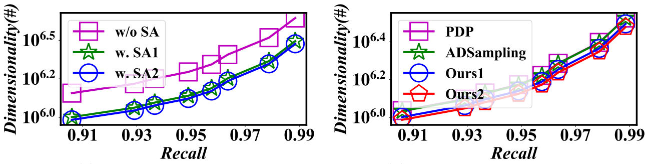
(a) Acceleration methods (b) Index processing time

**Figure 14: Effect of indexing-aware acceleration.**



(a) Compression methods (b) Search performance

**Figure 15: Effect of index Compression.**



(a) Acceleration effect (b) Comparison of methods

**Figure 16: Effect of query-aware acceleration.**

*6.7.1* ***Indexing-aware Acceleration.*** We evaluate the indexing-aware acceleration on MIT-States. In Figure 14(a), we record the total dimensionality needed for calculating the square Euclidean distance. The acceleration methods are gradually integrated into the original construction process. For instance, *w/o IA* means the process without acceleration, *w. IA1* is the enhanced process with computation reuse, *w. IA2* adds approximate computation optimization to *w. IA1*, and *w. IA3* refines *w. IA2* with accurate computation optimization. Our three acceleration methods exhibit a substantial reduction in the evaluated dimensionality compared to *w/o IA*. In Figure 14(b), we can see that the original process (*w/o IA*) demands considerable construction time, predominantly due to the extensive distance computation. In constrast, with all three acceleration methods (*w. IA*), MVG greatly reduces the index processing time.

*6.7.2* ***Index Compression.*** We study how index compression impacts the neighbor size (i.e., the space cost of neighbor ID list), the index processing time, and the search performance. Figure 15(a) shows that our compression techniques reduce the neighbor size by nearly 50%. Moreover, the index compression phase is fast, with index processing time similar to the uncompressed one. In Figure 15(b), we see that Seq-ID has higher query latency than the uncompressed one at the same *Recall* due to decompressing all neighbor IDs for each visited vertex. In contrast, both Mini-ID and Med-ID offer search performance on par with the uncompressed one, with a slight increase in latency. Med-ID is the default compression method in MVG, due to its better neighbor size reduction.

*6.7.3* ***Query-Aware Acceleration.*** Figure 16 illustrates the dimensionality used to calculate the square Euclidean distance in the search process. From Figure 16(a), we see that our search acceleration techniques (*w. SA1* and *w. SA2*) reduce the dimensionality by 32.9% compared to the initial method (*w/o SA*). This significant improvement is due to the approximate computation acceleration (*w. SA1*), which considers the dimensionality importance. Despite the small accurate computation fraction, our accurate computation acceleration still brings noticeable improvement. In Figure 16(b),

**Table 4: Complexities of different methods (refer to Table 1 and §2.2.2 for symbols).**

| Methods | Construction time | Space | Search |
|---|---|---|---|
| Milvus | $O((\theta(n)+r)c_1D_mn)$ | $O((mr+D_m)n)$ | $O(sc_2D_t\theta(n))$ |
| VBase | $O((\theta(n)+r)c_1D_mn)$ | $O((mr+D_m)n)$ | $O(tc_2D_t\theta(n))$ |
| Baseline | $O((\theta(n)+r)c_1D_m2^{m-1}n)$ | $O(((2^m-1)r+2^{m-1}D_m)n)$ | $O(c_2D_t\theta(n))$ |
| MVG | $O((\theta(n)m+2^{m-1}r)c_1D_mn)$ | $O(((2^m-1)r+D_m)n)$ | $O(c_2D_t\theta(n))$ |

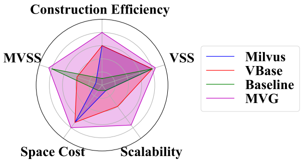


**Figure 17: Ability Comparison.**

our approximate computation acceleration methods are compared with two state-of-the-art methods: PDP [44] and ADSampling [25]. The former scans the dimensionality on raw vector incrementally, while the latter on a randomly projected vector. Our method beats both of these methods. By using weight-based optimization (Ours2), our method further reduces dimensionality, based on the initial approximate computation optimization (Ours1).

## 7 SUMMARY

We summarize our experiments and discuss MVG's possible optimizations and limitations. Table 4 catalogs the complexities of different methods, and Figure 17 compares their overall capabilities.

**Overall Performance.** Current MVSS methods support VSS efficiently, but suffer from inefficient and inaccurate MVSS. Moreover, MVSS performance worsens when there are more vectors in an object (e.g., $m > 2$). We pinpoint the performance bottleneck in these methods: *the single-vector index lacks the neighbors with multi-vector similarity and navigation.* Initially, we develop a baseline that uses multiple indexes to include similar and navigable neighbors for each vector combination, which improves MVSS performance. However, it increases the indexing time and space cost. Sequentially, we design MVG, a multi-vector graph index for MVSS. It includes diverse neighbors in a single index and integrates computation acceleration and index compression components. It offers efficient index processing, compact index layout, and high search efficiency and accuracy for both MVSS and VSS. MVG also shows state-of-the-art scalability to handle more vectors in an object and larger data scales.

**Potential Optimizations.** Current VSS methods' progress drives many potential optimizations for MVG. **(i)** We can use the GPU's parallel computation power to speed up the index construction and multi-vector search of MVG. **(ii)** We can deploy MVG on a high-performance SSD to optimize the index layout and I/O operations for over billion-scale data. **(iii)** We can integrate machine learning algorithms into MVG to predict the next hop or termination decision during multi-vector searching.

**Limitations.** Table 4 shows that MVG's construction time and space cost grow exponentially with the number of vectors $m$. For larger $m$ (e.g., $m > 10$), this may be infeasible. We highlight that most real-world scenarios do not need a large $m$ (e.g., $m$ is usually 2 [16, 20, 32]). For example, a multi-modal object has text and image modalities ($m = 2$) [51]; in video surveillance, three vectors represent each person's front face, side face, and posture ($m = 3$) [50]. To the best of our knowledge, current MVSS research mainly considers $m = 2$ [50, 65]. According to our evaluation, current MVSS methods need a larger number of neighbors $r$ for optimal search performance, leading to lower construction efficiency and higher space cost. In contrast, MVG has a smaller $r$ and reduces construction time and space cost with computation acceleration and index compression, optimizing $D_m$ and $r$ in the complexity formulas. In this paper, we show MVG's superiority for $m \leq 6$. For larger $m$, current methods and MVG all face construction efficiency and space cost challenges. We mark this as a future open problem.

## 8 RELATED WORK

**Vector Search Algorithms.** Vector search algorithms (a.k.a., approximate nearest neighbor search) aim to find the top-$k$ nearest vectors to a query vector from a large-scale vector set. Current research has four categories: Tree-based [19, 39, 41], Hashing-based [26, 29, 36], Quantization-based [14, 28, 34], and Graph-based methods [24, 38, 40]. Graph-based methods are promising, offering a state-of-the-art balance of efficiency and accuracy [37]. Various efforts have optimized graph algorithms from different perspectives, such as external storage [33, 53], GPU acceleration [49, 68], learn to route [35, 63]. These optimizations improve graph algorithms for VSS. However, current graph algorithms face performance bottlenecks for MVSS due to the drawback of single-vector index.

**Vector Databases.** Vector databases are built on vector search algorithms, offering versatile features for industrial applications [27]. Many vector databases, such as Milvus [12], Pinecone [10], and Weaviate [10], are essential for unstructured data management due to the rise of large language models (LLMs) [6, 67]. Current vector databases serve VSS tasks well for unstructured data under a specific modality. However, for MVSS on unstructured data with multiple modalities (needed for multi-modal LLMs [61]), existing databases lack support or have limitations. Extending vector databases for fast and accurate MVSS is challenging due to the absence of efficient, effective, and scalable index and search algorithms [43, 57].

**Multi-Vector Search.** Recent surveys [43, 47, 48, 57] and industrial scenarios [5, 7, 11, 13, 27] highlight the need for fast and accurate MVSS solutions. Existing methods manage MVSS by fine-tuning search strategies on multiple single-vector indexes [50, 65]. They improve performance to some extent, but still face efficiency, accuracy, and scalability issues. This is mainly due to the limitations of single-vector index for multi-vector queries.

## 9 CONCLUSION

We investigate the MVSS problem and introduce an innovative solution, MVG. Our method encompasses all vector combinations in a well-designed index to efficiently support both VSS and MVSS. We develop computation acceleration techniques and index compression algorithms, integrating them into MVG to facilitate rapid index processing, a compact index layout, and efficient search. Furthermore, we reinforce our method with rigorous theoretical analysis. Our experiments demonstrate the superiority of our method in index construction efficiency, space cost, MVSS performance, VSS performance, and scalability. In future research, we plan to explore potential optimizations discussed in §7 to further enhance MVG.

# REFERENCES

[1] 2023. Hugging Face Embeddings API. https://huggingface.co/blog/getting-started-with-embeddings.

[2] 2023. Reinventing search with a new AI-powered Bing and Edge, your copilot for the web. https://news.microsoft.com/the-new-Bing/.

[3] 2023. Vector search in Azure AI Search. https://learn.microsoft.com/en-us/azure/search/vector-search-overview.

[4] 2023. Welcome to the Gemini era. https://deepmind.google/technologies/gemini/.

[5] 2024. Can we query on multiple vector embedding fields in vector search? https://www.mongodb.com/community/forums/t/can-we-query-on-multiple-vector-embedding-fields-in-vector-search/244155.

[6] 2024. ChatGPT plugins. https://openai.com/blog/chatgpt-plugins.

[7] 2024. MultiVector Retriever. https://python.langchain.com/docs/modules/data_connection/retrievers/multi_vector.

[8] 2024. MVG Index: Empowering Multi-Vector Similarity Search in High-Dimensional Spaces (Technical Report). https://github.com/ZJU-DAILY/MVG/blob/main/MVG_technical_report.pdf.

[9] 2024. pgvector. https://github.com/pgvector/pgvector.

[10] 2024. pinecone. https://www.pinecone.io/.

[11] 2024. Querying with multiple vectors during embedding nearest neighbor search? https://www.reddit.com/r/MachineLearning/comments/10rvkru/d_querying_with_multiple_vectors_during_embedding/.

[12] 2024. Vector database built for scalable similarity search. https://milvus.io/.

[13] 2024. What is tensor search? https://medium.com/@jesse_894/introducing-marqo-build-cloud-native-tensor-search-applications-in-minutes-9cb9a05a1736.

[14] 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. *PVLDB* 9, 4 (2015), 288–299.

[15] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. ACL 2023 Tutorial: Retrieval-based Language Models and Applications. *ACL*.

[16] Alberto Baldrati, Marco Bertini, Tiberio Uricchio, and Alberto Del Bimbo. 2022. Conditioned and composed image retrieval combining and partially fine-tuning CLIP-based features. In *CVPR*. 4959–4968.

[17] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. 2018. Multimodal machine learning: A survey and taxonomy. *TPAMI* 41, 2 (2018), 423–443.

[18] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, and Bo Zheng. 2022. Approximate Nearest Neighbor Search under Neural Similarity Metric for Large-Scale Recommendation. In *CIKM*. 3013–3022.

[19] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *SOTC*. 537–546.

[20] Ginger Delmas, Rafael Sampaio de Rezende, Gabriela Csurka, and Diane Larlus. 2022. ARTEMIS: Attention-based Retrieval with Text-Explicit Matching and Implicit Similarity. In *ICLR*.

[21] Aleksandra Edwards, Asahi Ushio, José Camacho-Collados, Hélène de Ribaupierre, and Alun D. Preece. 2021. Guiding Generative Language Models for Data Augmentation in Few-Shot Text Classification. *arXiv:2111.09064* (2021).

[22] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal aggregation algorithms for middleware. In *PODS*. 102–113.

[23] Yue Fan and Xiuli Ma. 2022. Multi-Vector Embedding on Networks with Taxonomies. In *IJCAI*. 2944–2950.

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

[25] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proc. ACM Manag. Data* 1, 2 (2023), 137:1–137:27.

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

[27] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *PVLDB* 15, 12 (2022), 3548–3561.

[28] 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 *ICML*. 3887–3896.

[29] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.

[30] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. 2019. FiBiNET: combining feature importance and bilinear feature interaction for click-through rate prediction. In *RecSys*. 169–177.

[31] Phillip Isola, Joseph J Lim, and Edward H Adelson. 2015. Discovering states and transformations in image collections. In *CVPR*. 1383–1391.

[32] Surgan Jandial, Pinkesh Badjatiya, Pranit Chawla, Ayush Chopra, Mausoom Sarkar, and Balaji Krishnamurthy. 2022. SAC: Semantic attention composition for text-conditioned image retrieval. In *CVPR*. 4021–4030.

[33] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS*, Vol. 32.

[34] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *TPAMI* 33, 1 (2011), 117–128.

[35] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *SIGMOD*. 2539–2554.

[36] 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 *ICDE*. 289–300.

[37] 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. *TKDE* 32, 8 (2020), 1475–1488.

[38] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2022. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *PVLDB* 15, 2 (2022), 246–258.

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

[40] Yury A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *TPAMI* 42, 4 (2020), 824–836.

[41] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *TPAMI* 36, 11 (2014), 2227–2240.

[42] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based news recommendation for millions of users. In *SIGKDD*. 1933–1942.

[43] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of Vector Database Management Systems. *arXiv:2310.14021* (2023).

[44] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27.

[45] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *ICML*. 8748–8763.

[46] 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 *CVPR*. 3020–3028.

[47] Toni Taipalus. 2023. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *arXiv:2309.11322* (2023).

[48] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *Data Engineering* (2023), 39–54.

[49] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. Fast k-NN Graph Construction by GPU based NN-Descent. In *CIKM*. 1929–1938.

[50] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. 2614–2627.

[51] Mengzhao Wang, Xiangyu Ke, Xiaoliang Xu, Lu Chen, Yunjun Gao, Pinpin Huang, and Runkai Zhu. 2024. MUST: An Effective and Scalable Framework for Multimodal Search of Target Modality. In *ICDE*.

[52] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint. In *NeurIPS*.

[53] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1 (2024), 14:1–14:27.

[54] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978.

[55] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. 2022. LIDER: An Efficient High-dimensional Learned Index for Large-scale Dense Passage Retrieval. *PVLDB* 16, 2 (2022), 154–166.

[56] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph-and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *Data Engineering* (2023), 3–21.

[57] Renzhi Wu, Jingfan Meng, Jie Jeff Xu, Huayi Wang, and Kexin Rong. 2023. Rethinking Similarity Search: Embracing Smarter Mechanisms over Smarter Data.

*arXiv:2308.00909* (2023).

[58] Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *NeurIPS*.

[59] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *SIGMOD*. 2241–2253.

[60] Yiben Yang, Chaitanya Malaviya, Jared Fernandez, Swabha Swayamdipta, Ronan Le Bras, Ji-Ping Wang, Chandra Bhagavatula, Yejin Choi, and Doug Downey. 2020. Generative Data Augmentation for Commonsense Reasoning. In *EMNLP*. 1008–1025.

[61] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. 2023. A Survey on Multimodal Large Language Models. *arXiv:2306.13549* (2023).

[62] 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 *SIGKDD*. 974–983.

[63] Qiang Yue, Xiaoliang Xu, Yuxiang Wang, Yikun Tao, and Xuliyuan Luo. 2023. Routing-Guided Learned Product Quantization for Graph-Based Approximate Nearest Neighbor Search. *arXiv:2311.18724* (2023).

[64] Minjia Zhang, Jie Ren, Zhen Peng, Ruoming Jin, Dong Li, and Bin Ren. 2023. Exploiting Modern Hardware Architectures for High-Dimensional Vector Search at Speed and Scale. *Data Engineering* (2023), 22–38.

[65] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. {VBASE}: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *OSDI*.

[66] Shaoting Zhang, Ming Yang, Timothee Cour, Kai Yu, and Dimitris N Metaxas. 2014. Query specific rank fusion for image retrieval. *TPAMI* 37, 4 (2014), 803–815.

[67] Yi Zhang, Zhongyang Yu, Wanqi Jiang, Yufeng Shen, and Jin Li. 2023. Long-Term Memory for Large Language Models Through Topic-Based Vector Database. In *IALP*. 258–264.

[68] Weijie Zhao, Shulong Tan, and Ping Li. 2020. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1033–1044.

[69] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *PVLDB* 16, 8 (2023), 1979–1991.

# APPENDIX I: PROOFS

Due to space limitations in the main text, we provide detailed proofs of several significant theorems and lemmas from §4.1 and §4.2 here for reference.

## A PROOF OF LEMMA 1

Lemma 1. *An object with $m$ vectors has $2^m - 1$ vector combinations.*

Proof. Each vector from the object can either be included in a combination or not. Hence, for each vector, there are two choices: include it or exclude it. Therefore, with $m$ vectors, there are $2^m$ potential combinations (since each of the $m$ vectors can be in one of two states - selected or not). However, this count includes the empty set (where no vectors are selected). Since the problem requires nonempty subsets, we subtract 1 from $2^m$ to exclude the empty set. Thus, there are $2^m - 1$ nonempty combinations of vectors for an object with $m$ vectors. □

## B PROOF OF LEMMA 2

Lemma 2. *For an object $o$ with $m$ vectors, each vector is included in $2^{m-1}$ vector combinations.*

Proof. Considering a set of $m$ vectors, select one specific vector from this set. We then count all combinations that include this particular vector. As we include this vector in every combination, we are left with $m-1$ vectors. For each remaining $m-1$ vectors, we have two options - include in the current combination or exclude it. This implies that for the $m-1$ vectors left, we have $2^{m-1}$ potential combinations. As each of these combinations unquestionably includes our selected vector, it follows that each vector from the set of $m$ vectors is included in the $2^{m-1}$ combinations. □

## C PROOF OF LEMMA 3

Lemma 3. *The computation reuse does not impact the index quality.*

Proof. Suppose the current vector combination consists of $m'$ vectors. For simplicity, we assume that they are the first $m'$ vectors in the inserted object $o$. To calculate the multi-vector distance $g(\delta(o_0, x_0), \cdots, \delta(o_{m'-1}, x_{m'-1}))$ for an accessed vertex $x$, we need to compute each $\delta(o_i, x_i)$ for $0 \leq i \leq m'-1$ (refer to Equation 3). The method of computation reuse directly obtain some $\delta(o_i, x_i)$ values from the cache upon matching, thus avoiding the computation of $\delta(o_i, x_i)$. Therefore, the multi-vector distance is identical before and after the application of computation reuse. □

## D PROOF OF LEMMA 4

Lemma 4. *The projection matrix $\boldsymbol{M_i}$ is orthonormal.*

Proof. Given that the eigenvectors are orthogonal and normalized, the dot product of any pair of distinct columns equals zero, and the self dot product of a column equals one. Therefore, the matrix $\boldsymbol{M_i}$, composed of these eigenvectors, is confirmed to be an orthonormal matrix. □

## E PROOF OF LEMMA 5

Lemma 5. *Transforming vectors $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ in $S_i$ by the orthonormal matrix $\boldsymbol{M_i}$ does not alter the distance between them.*

Proof. One fundamental property of orthogonal matrices asserts that the transpose of the matrix equals its inverse:

$$\boldsymbol{M_i}^\top = \boldsymbol{M_i}^{-1} \quad . \tag{12}$$

Transformation of a vector $\boldsymbol{o_i}$ by $\boldsymbol{M_i}$ does not change its length. Formally written as $||\boldsymbol{M_i}^\top \boldsymbol{o_i}||^2 = (\boldsymbol{M_i}^\top \boldsymbol{o_i})^\top (\boldsymbol{M_i}^\top \boldsymbol{o_i}) = \boldsymbol{o_i}^\top (\boldsymbol{M_i} \boldsymbol{M_i}^\top) \boldsymbol{o_i} = \boldsymbol{o_i}^\top \boldsymbol{I} \boldsymbol{o_i} = \boldsymbol{o_i}^\top \boldsymbol{o_i} = ||\boldsymbol{o_i}||^2$, where $\boldsymbol{I}$ denotes the identity matrix. Similarly, for two vectors $\boldsymbol{o_i}, \boldsymbol{x_i}$, their distance stays constant when the vectors are multiplied by an orthonormal matrix: $\delta(\boldsymbol{M_i}^\top \boldsymbol{o_i}, \boldsymbol{M_i}^\top \boldsymbol{x_i}) = ||\boldsymbol{M_i}^\top \boldsymbol{o_i} - \boldsymbol{M_i}^\top \boldsymbol{x_i}|| = ||\boldsymbol{M_i}^\top (\boldsymbol{o_i} - \boldsymbol{x_i})|| = ||\boldsymbol{o_i} - \boldsymbol{x_i}|| = \delta(\boldsymbol{o_i}, \boldsymbol{x_i})$. Therefore, the distance between vectors remains unchanged when transforming them by an orthonormal matrix. □

## F PROOF OF LEMMA 6

Lemma 6. *With $\lambda_j$ as the $j$-th largest eigenvalue of $\Sigma_{\boldsymbol{i}}$, and $\sigma_j^2$ as the variance of the $j$-th dimension of $\tilde{S_{\boldsymbol{i}}}$, it holds: $\sigma_j^2 = \lambda_j$.*

Proof. We refer to $\hat{S_{\boldsymbol{i}}}$ as our original zero-centered data, leading to its mean, $E(\hat{S_{\boldsymbol{i}}})$, is 0. Following the transformations carried under Equation 8, the mean of the projected data, $E(\tilde{S_{\boldsymbol{i}}})$, is also 0. To connect the variance, $\sigma_j^2$, to the $j$-th largest eigenvalue, $\lambda_j$, we use the identity $Var(\tilde{S_{\boldsymbol{i}}}) = E(\tilde{S_{\boldsymbol{i}}}^2) - (E(\tilde{S_{\boldsymbol{i}}}))^2$, where $Var(\tilde{S_{\boldsymbol{i}}})$ is the variance of $\tilde{S_{\boldsymbol{i}}}$. It simplifies to $Var(\tilde{S_{\boldsymbol{i}}}) = E(\tilde{S_{\boldsymbol{i}}}^2)$ for zero-mean data. The $j$-th dimension of the transformed data is identified as $\tilde{S_{\boldsymbol{i}}}[j] = \boldsymbol{M_i}^\top [j] \hat{S_{\boldsymbol{i}}}$ where $\boldsymbol{M_i}^\top [j]$ represents the $j$-th row of $\boldsymbol{M_i}^\top$. The variance can be computed as follows:

$$\begin{aligned} \sigma_j^2 = Var(\tilde{S_{\boldsymbol{i}}}[j]) &= E((\boldsymbol{M_i}^\top [j] \hat{S_{\boldsymbol{i}}})^2) \\ &= \boldsymbol{M_i}^\top [j] E(\hat{S_{\boldsymbol{i}}} \hat{S_{\boldsymbol{i}}}^\top)(\boldsymbol{M_i}^\top [j])^\top \\ &= \boldsymbol{M_i}^\top [j] \Sigma_{\boldsymbol{i}} (\boldsymbol{M_i}^\top [j])^\top \quad . \end{aligned} \tag{13}$$

Given that $(\boldsymbol{M_i}^\top [j])^\top$ is the $j$-th eigenvector of $\Sigma_{\boldsymbol{i}}$ and $\lambda_j$ is the corresponding eigenvalue, it follows that

$$\sigma_j^2 = \lambda_j \boldsymbol{M_i}^\top [j] (\boldsymbol{M_i}^\top [j])^\top = \lambda_j ||\boldsymbol{M_i}^\top [j]||^2 = \lambda_j \quad . \tag{14}$$

Thus, the variance of the $j$-th dimension of the transformed data $\tilde{S_{\boldsymbol{i}}}$ equals the $j$-th largest eigenvalue of the covariance matrix $\Sigma_{\boldsymbol{i}}$. □

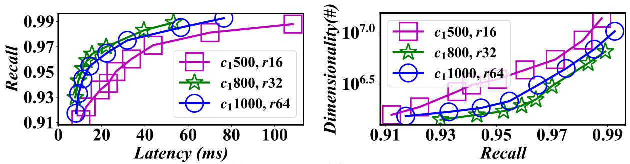
## G PROOF OF THEOREM 1

Theorem 1. *After transforming $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ using $\boldsymbol{M_i}$, the contribution to the distance between $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ is non-increasing from the first dimension to the last dimension.*

Proof. Let us designate $\tilde{\boldsymbol{o_i}}$ and $\tilde{\boldsymbol{x_i}}$ as projections of vectors $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ onto the new space defined by $\boldsymbol{M_i}$. Assume $\tilde{o_i}[j]$ and $\tilde{x_i}[j]$ as the $j$-th elements of these vectors. The contribution of the $j$-th dimension to the distance is measured as:

$$(\Delta_j)^2 = (\tilde{\boldsymbol{o_i}}[j] - \tilde{\boldsymbol{x_i}}[j])^2 \quad . \tag{15}$$

Recall that the variance associated with each eigenvector (each dimension in the new space) reduces in sequence. Hence, for each dimension $j$, the variance linked to this dimension is greater or equivalent to the variance of dimension $j+1$. The measurement $\Delta_j$ represents the projection of the difference between $\boldsymbol{o_i}$ and $\boldsymbol{x_i}$ onto the $j$-th eigenvector. It is a multiple of the standard deviation of the data, projected onto this dimension, which is the square root of the corresponding eigenvalue. This means that each dimension's



(a) *Latency*-vs-*Recall* (b) Evaluated demensionality per query

**Figure 18: Effect of different parameters.**

contribution to the distance between $\tilde{\boldsymbol{o}}_i$ and $\tilde{\boldsymbol{x}}_i$ is proportional to the relevant eigenvalue. Thus, the contribution of the $j$-th dimension to distance, $(\Delta_j)^2$, is larger or equal to $(\Delta_{j+1})^2$, which constitutes the contribution of the dimension $j+1$. This proves our assertion, indicating that the contribution to the distance between vector pairs continuously lessens from the first dimension to the last after their projection using $\boldsymbol{M_i}$. □

## H PROOF OF LEMMA7

Lemma 7. *The optimization of approximate computation does not impact the index quality.*

Proof. According to Lemma 5, the distance remains unchanged before and after the execution of an orthogonal transformation. As a result, the partial distance $\widetilde{dist}'$ on the transformed vectors ranges from 0 to $dist$, where $dist$ represents the accurate distance between vectors. If $\widetilde{dist}' > T$, indicating $dist > T$, it implies a correct comparison can be made using the partial distance; otherwise, MVG continues to scan more dimensions to assess the relationship between $\widetilde{dist}'$ and $T$. The scan continues until either $\widetilde{dist}' > T$ emerges or all dimensions have been scanned. When scanning all dimensions, $\widetilde{dist}' = dist$, at which point a correct comparison can also be made. □

# APPENDIX II: ADDITIONAL EVALUATIONS

In this section, we present additional experiments aimed at comprehensively evaluating the methods.

## I PARAMETER CONFIGURATION

In line with HNSW, two key parameters are considered: the candidate set size, $c_1$, and the maximal number of neighbors, $r$ (layer 0). $c_1$ determines a trade-off between construction time and index quality, while $r$ defines the maximum number of outgoing connections in the graph. Taking VBase as an example, we test the impact of different parameters on search performance, as depicted in Figure 18. In our main text experiments, we employ the parameters that yield the best search performance. The detailed parameter values are provided in Table 5.

## J CONSTRUCTION TIME PROFILE

Table 6 outlines the construction time profile of MVG across various datasets. *Matrix Generation* refers to the time taken to derive a projection matrix (§4.2.2) for facilitated computation acceleration. *Transformation* denotes the time required to modify the vectors of an inserted object based on the projection matrix. *Object Insertion* corresponds to the time it takes to identify candidate and final neighbors. Finally, *Compression* signifies the time consumed for neighbor list compression.

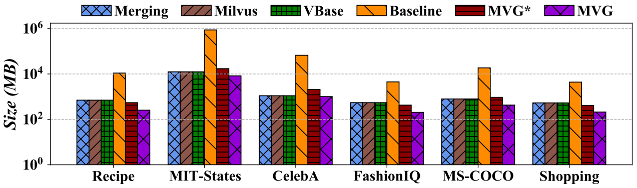
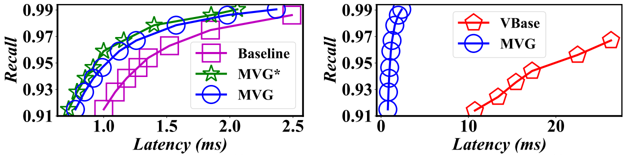


**Figure 19: Index size.**

## K INDEX SIZE WITHOUT RAW VECTOR STORAGE

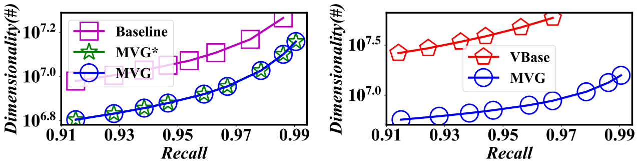
Figure 19 illustrates the index sizes of various methods, excluding the storage of raw vectors in its calculations.

## L PERFORMANCE CURVES OF MVSS



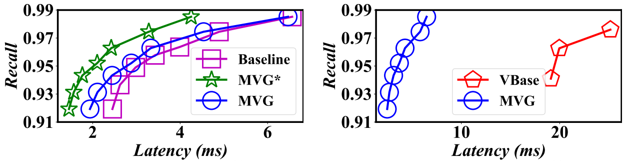
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

**Figure 20: *Latency*-vs-*Recall* curve on Recipe.**



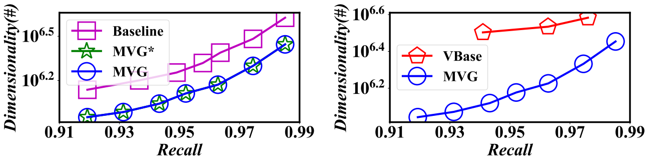
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

**Figure 21: *Dimensionality*-vs-*Recall* curve on Recipe.**



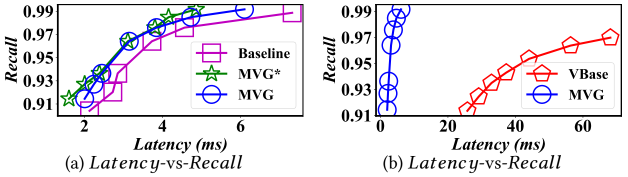
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

**Figure 22: *Latency*-vs-*Recall* curve on MIT-States.**



(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

**Figure 23: *Dimensionality*-vs-*Recall* curve on MIT-States.**



(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

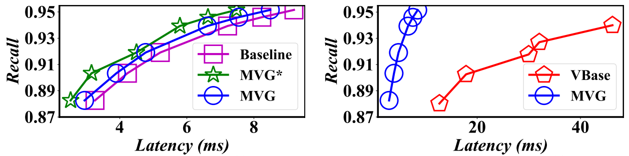
**Figure 24: *Latency*-vs-*Recall* curve on CelebA.**

**Table 5: Parameter values used in our experiments.**

| Methods | | Recipe | MIT-States | CelebA | FashionIQ | MS-COCO | Shopping |
|---|---|---|---|---|---|---|---|
| `Merging`/`Milvus`/`VBase` | $c_1$ | 800 | 2,000 | 800 | 800 | 800 | 800 |
| | $r$ | 32 | 128 | 32 | 32 | 32 | 32 |
| `Baseline`/`MVG*`/`MVG` | $c_1$ | 500 | 500 | 500 | 500 | 500 | 500 |
| | $r$ | 16 | 16 | 16 | 16 | 16 | 16 |

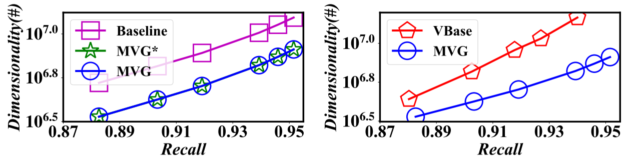
**Table 6: Construction time profile for `MVG` (Time: s).**

| | Recipe | MIT-States | CelebA | FashionIQ | MS-COCO | Shopping |
|---|---|---|---|---|---|---|
| *Matrix Generation* | 3.17 | 2.62 | 1.96 | 0.89 | 1.18 | 0.76 |
| *Transformation* | 172.42 | 569.16 | 194.25 | 99.84 | 157.13 | 101.14 |
| *Object Insertion* | 860.61 | 16,675.90 | 2,168.70 | 510.76 | 1,020.10 | 462.93 |
| *Compression* | 1.66 | 26.35 | 3.62 | 1.27 | 1.59 | 1.26 |
| *Total Time* | 1,037.86 | 17,274.03 | 2,368.53 | 612.76 | 1,180.00 | 1,123.26 |



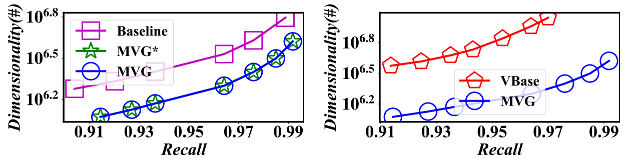
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

**Figure 30: *Latency*-vs-*Recall* curve on Shopping.**



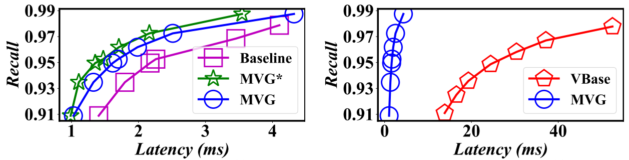
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

**Figure 31: *Dimensionality*-vs-*Recall* curve on Shopping.**



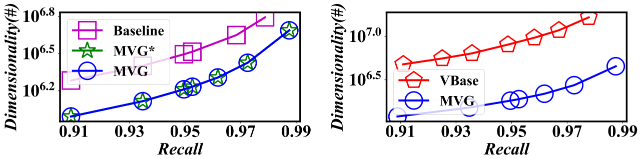
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

**Figure 25: *Dimensionality*-vs-*Recall* curve on CelebA.**



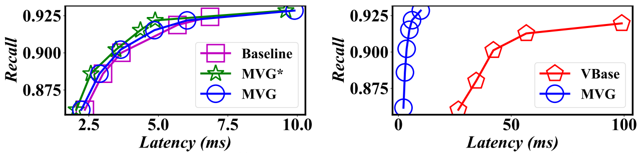
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

**Figure 26: *Latency*-vs-*Recall* curve on FashionIQ.**



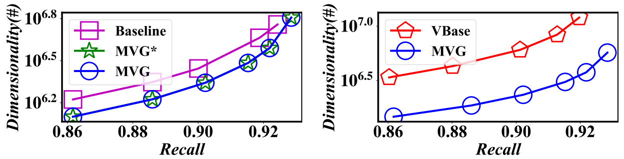
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

**Figure 27: *Dimensionality*-vs-*Recall* curve on FashionIQ.**



(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

**Figure 28: *Latency*-vs-*Recall* curve on MS-COCO.**



(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

**Figure 29: *Dimensionality*-vs-*Recall* curve on MS-COCO.**

To clearly illustrate the trade-off between search efficiency and accuracy, we present the performance curve in Figure 20–31. For a detailed overview of the compared methods, please refer to §6.1.3. The curves for `Merging` and `Milvus` are not included due to their significantly low search efficiency or accuracy.