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, most current advancements 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 efforts 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 pair of objects into a single index, enabling seamless processing of both VSS and MVSS. Furthermore, MVG integrates three computational acceleration methods that leverage characteristics specific to MVSS scenarios into 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 index size and search efficiency requirements. Theoretical validations confirm that all acceleration and compression techniques are lossless, ensuring unaltered index quality and search accuracy. Extensive experiments on real-world datasets demonstrate that MVG outperforms the leading VBase in terms of index construction efficiency, space cost, search performance, and scalability, exhibiting up to 96.5% reduction in query latency while maintaining a higher recall rate.

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

1 INTRODUCTION

High-dimensional vector representation for unstructured data, such as documents and images, has become a key building block for training and deploying Artificial Intelligence (AI) models like GPT and CLIP [15, 49, 66]. This paradigm has shown significant promise in various emerging AI applications, sparking interest in Vector Similarity Search (VSS) 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, 72]. VSS has long been a pivotal topic in domains like information retrieval [57, 60], recommendation systems [18, 46, 62], and databases [27, 77]. 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) [55, 61], is preferred in real-world scenarios. To expedite the 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 [44]) is the most promising and has consequently become a mainstream research direction and practical choice in the VSS field [34, 59].

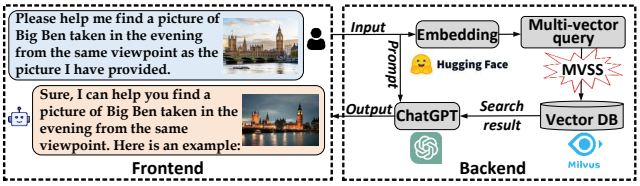


Figure 1: An example of text-image chat by MVSS.

To enhance the capabilities of AI applications, Multi-Vector Similarity Search (MVSS) has been proposed for processing multi-modal or multi-view data [55, 73]. In this context, an object or query contains multiple vectors, and object similarity 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 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 modified text to form two query vectors for retrieval in a multi-modal object set [56]. Additionally, a single-modal object can also construct multiple vectors from different views, with MVSS ensuring more accurate search results [23, 31, 78]. Given the significance of MVSS, recent research surveys and vector database products emphasize the demand for a fast and accurate MVSS solution [27, 47, 51, 53, 63]. However, current VSS methods can only address VSS with one vector in a query or object.

Example 1. In a text-image chat scenario, users can input a reference image and instructional text to get the desired result using MVSS. As shown 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

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

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

("Front- end"). 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 separate 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 [55]), and the search result is combined with the user input to prompt ChatGPT [6]. Finally, a user-friendly response is displayed in the frontend.

Recent studies address MVSS by optimizing the search process across multiple single-vector indexes. They build m indexes on an object set S, where each object has m vectors; a query with t vectors initiates the scanning of t corresponding indexes. A straightforward method, `Merging`, acquires t candidate sets via VSS on t indexes, yielding final results by merging these sets [74]. However, determining the optimal number of candidates per VSS is challenging [73]. Too many or too few candidates can lead to either an increased computational burden or inaccurate results, respectively [55]. `Milvus` [55] addresses this issue by trialing various candidate sizes and reordering the candidates via the NRA algorithm[2] [22]. Nevertheless, each trial requires repetitive traversals of indexes, resulting in significant vector access and computation. Alternatively, `VBase` [73] circumvents repeated NRA iterations by employing a round-robin traversal of each index. It includes a mechanism to determine which indexes to traverse more frequently based on current results. Consequently, it retrieves results during traversal without the need for additional merging steps. `VBase` outperforms `Milvus` in search efficiency and accuracy. Despite its advantages, `VBase`'s dependence on multi-index scanning can lead to unnecessary vector visitation, which hampers efficiency, given single-vector but not multi-vector object similarity for each index.

We elucidate 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 single-vector distance, neglecting multi-vector distance. To illustrate, we build two HNSW indexes on a million-scale Recipe dataset [50]: one using single-vector distance (HNSW*) and another using multi-vector distance (HNSW**). We observe that the average neighbor overlap ratio is a mere *18%* between the two indexes, indicating that the single-vector HNSW is inadequate in capturing the multi-vector similarity essential for MVSS. Second, conducting a multi-vector search on a single-vector HNSW can misguide traversal, leading to incorrect search outcomes. Our tests show that with the same search parameters, HNSW** achieves a recall rate of *0.98*, significantly outperforming HNSW*'s *0.65*. Additionally, HNSW* requires more computational load to process a query, suggesting its subpar navigation and neighbor similarity for multi-vector queries. In a nutshell, the main reason constraining the MVSS performance of existing methods stems from the drawback of the single-vector index.

In this paper, we present a **M**ulti-**V**ector **G**raph (`MVG`) index designed to enhance the speed and accuracy of MVSS. `MVG` builds a graph structure on multi-vector objects, covering all possible vector combinations. This index determines the neighbors of a vertex by computing the similarity between objects across various combinations, thereby supporting queries involving any vector combinations. Through adaptable navigation to pertinent neighbors depending on the query's vector combination and weights, `MVG` achieves efficient and accurate search capabilities. We specifically explore the following three research questions.

(i) *Efficient construction of a high-quality multi-vector index.* The assembly of a multi-vector index that encompasses all vector combinations is notably time-consuming. Our evaluation indicates that it takes *31.5* × more time than building single-vector indexes on the MIT-States dataset. To address this, `MVG` employs three lossless acceleration strategies that leverage the characteristics of index construction and multi-vector distance computation.

(ii) *Lossless compression of the multi-vector index.* Considering multi-vector relationships, a vertex in the multi-vector graph index contains more neighbor data, resulting in an index size that is *9.7* × greater than that of single-vector indexes on MIT-States. For this issue, `MVG` introduces three lossless compression techniques for the index, based on its well-designed layout.

(iii) *Fast and accurate processing of any query type.* `MVG` addresses this challenge by employing a unified graph index that accommodates various neighbor types for different vector combinations and facilitates adaptive neighbor access during the search process. By exploiting the features of multi-vector queries, `MVG` effectively integrates lossless acceleration techniques into the search procedure and enhances it with the inclusion of query input weights.

To the best of our knowledge, this work is the first exploration into designing a multi-vector index and search strategy that exploits the intrinsic features of MVSS. Our evaluations show that `MVG` outperforms state-of-the-art methods in terms of index construction efficiency, index size, query latency, and recall rate, achieving up to *96.5%* reduction in query latency. Notably, `MVG` easily attains a recall rate over *0.99*, a level of accuracy challenging 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 a compact index layout (**§3**). Capable of supporting queries with any vector combinations and weights using a single index, `MVG` demonstrates remarkable improvements in search efficiency and accuracy.
- We propose three acceleration techniques tailored for index construction and search in multi-vector scenarios (**§4.2** and **§5.2**). These techniques leverage the features of building multi-vector indexes and conducting multi-vector searches, and we prove that they do not compromise index quality or search accuracy.
- We develop three compression algorithms grounded in the well-designed layout of the multi-vector index (**§4.3**). They can 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]NRA can efficiently derive 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 [44] 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
n	The number of objects in S
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 aggregate function of $\delta(.)$
$\|\| \cdot \|\|$	The l_2-norm of a vector

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 [34, 44].

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 aggregate 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:*

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

Current studies primarily evaluate cases where $m = 2$ for MVSS in their experiments [55, 73], with only a few instances considering m up to 4 [56]. In this paper, we expand the range of m from 1 to 6, covering most real-world applications [17, 30, 50, 70]. For a multi-vector query, the meaningful value of t varies from 1 to m, and the corresponding relationship of vectors in q and o is provided alongside q. Without loss of generality, we assume that the i-th query vector, with $1 \le i \le t$, corresponds to the i-th vector of object o. We employ the weighted sum for $g(.)$, as it is widely used in current MVSS scenarios [55, 73]:

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

When $m = t = 1$, the MVSS problem is simplified to VSS, making MVSS a more general problem. With the rise of versatile AI applications such as Copilot [2] and Gemini [4], the importance of MVSS continues to grow. For the sake of illustration, we give the definition of vector combination within a multi-vector object.

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.*

The following lemma presents the number of vector combinations given the number of vectors within an object.

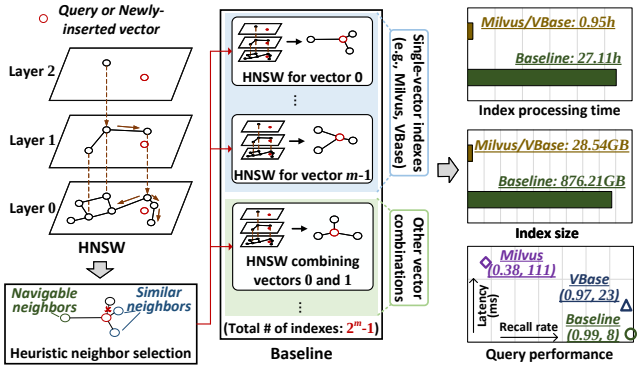


Figure 2: HNSW-based MVSS methods.

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]. □

Remarks. Following current MVSS research [55, 73], 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 default to weights of 1. Therefore, the aggregate distance during index construction is the sum of the distances between each pair of vectors. (3) Different queries may have varying vector combinations and weights.

2.2 Motivation Illustration on HNSW

We outline the workflow of HNSW along with its time and space complexity. Then, we delve into the operational principles of current MVSS methods, which rely on HNSW. This guides us to establish a baseline and identify its issues, thus steering our research.

2.2.1 ***HNSW Algorithm.*** Among graph-based VSS methods, Hierarchical Navigable Small World graph (HNSW) is well-studied in academia [25, 38] and widely deployed in industrial vector databases [9, 12, 67]. Figure 2 (left) illustrates HNSW's hierarchical structure, 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. The maximum layer of each vector 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 neighbors that are both similar and navigable, based on a heuristic strategy (the navigable neighbors are from different isolated clusters and act as "expressways" for search efficiency [44]). During online query serving, the greedy search starts from the top layer and gets the entry vertex for the next layer. The search then proceeds to the lower layer and repeats the same process. 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 vertex closest to the query is extracted from the candidate set, and its neighbors are subsequently 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 building 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` [55] and `VBase` [73], have optimized the 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]. `VBase` scans t 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 $O(c_2 \cdot D_t \cdot t \cdot \theta(n))$.

According to the complexity analysis and empirical results depicted in Figure 2 (right), current MVSS methods exhibit subpar efficiency and accuracy. We observe clear limitations of multiple single-vector HNSW indexes when handling 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 index.

2.2.3 ***Baseline.*** A straightforward optimization involves building an HNSW index for each vector combination, yielding 2^m-1 indexes (the middle of Figure 2). Thus, each multi-vector query matches an index according to 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 that of 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.

Example 2. Figure 2 (right) shows the evaluation of three HNSW-based MVSS methods on the MIT-States dataset [32]. 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 superior search efficiency and accuracy.

[4]In the HNSW paper, $\theta(n)$ is roughly $\log(n)$ [44].

Algorithm 1: Index Construction

Input: An object set S
Output: The multi-vector graph index G

1 **for** *each object o in S* **do**
2 $l_{max} \leftarrow$ the maximum layer of o; ▷ exponential decaying
3 $L \leftarrow$ the current graph index's top layer;
4 **if** $l_{max} > L$ **then** ▷ update the top layer
5 $L \leftarrow l_{max}$;
6 **for** $l \in \{l_{max}, \cdots, 0\}$ **do**
7 insert o at layer l; ▷ §4.1 (Alg. 3), §4.2
8 compress the neighbor IDs of o; ▷ §4.3
9 store o's vectors and neighbor IDs in index G;
10 **return** multi-vector index G

Algorithm 2: Search Procedure

Input: A query q
Output: Top-k results R

1 $e \leftarrow$ entry point at top layer; ▷ it is fixed
2 **for** $l \in \{L, \cdots, 1\}$ **do**
3 $r \leftarrow$ top-1 nearest vertex to q at layer l; ▷ §5.1 (Alg. 4), §5.2
4 $e \leftarrow r$; ▷ r is the entry point of the next layer
5 $R \leftarrow$ top-k nearest vertices to q at layer 0; ▷ §5.1 (Alg. 4), §5.2
6 **return** R

In summary, while current MVSS methods utilize single-vector indexes for efficient construction and small index size, they exhibit low search efficiency and accuracy. The baseline approach enhances search accuracy and efficiency, but at the cost of high index construction time and space. 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 construction and search procedure, offering efficient index processing, compact index layout, and superior search performance, all at once. Here, we outline the workflow of `MVG`.

3.1 Index Construction

`MVG` builds a hierarchical graph index for a collection of objects. The construction process is outlined in Algorithm 1. It begins with an empty graph to which objects are progressively added, forming a sub-graph with the already added objects (lines 1-9). When a new object is inserted, each vector combination of the object is treated as a query. These queries are used to obtain respective similar objects relative to their vector combinations by executing a greedy search on the current graph index; neighbors of each vector combination are selected from their similar objects based on the heuristic rule of HNSW (§4.1). During this process, we identify three types of distance computations and develop three corresponding lossless acceleration strategies (§4.2). After receiving the adjacency list of the new object, it is processed through the compression component, where the neighbor IDs are compressed using ID residual (§4.3).

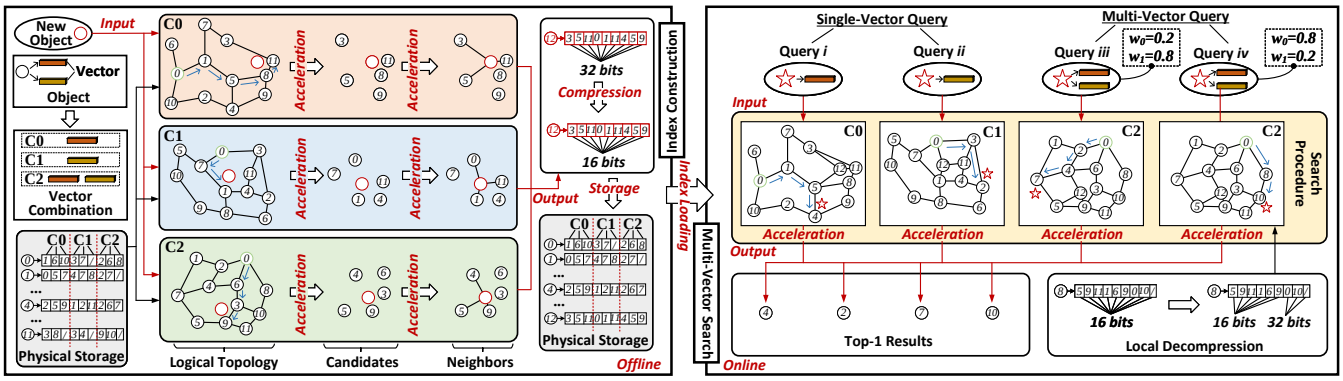


Figure 3: The execution flow of index construction and search procedure in MVG.

Example 3. As Figure 3 (left) shows, each object is associated with two vectors, resulting in three vector combinations: C0, C1, and C2. Consequently, MVG performs three traversals, each employing a particular vector combination as a query to retrieve similar neighbors. To simplify, the insertion process for objects on the graph index of a specific layer is shown, with the observation that all layers operate similarly. Notably, each combination uses distinct elements for calculating distances. For C0, the inter-object distance is determined by the distance between their first vectors. For C1, it is based on the distance between their second vectors. For C2, it is the sum of the distances of their first and second vectors. The final adjacency list merges neighbor IDs from all vector combinations, with each ID taking up 32 bits. By compressing these IDs, we manage to reduce the space needed for neighbor storage, using smaller bit sizes (e.g., 16 bits, based on the ID residual).

Remarks. (1) In real-world scenarios, users typically employ different weights in multi-vector queries. However, during the index construction phase, weight information often is not available in advance [55, 73]. Consequently, MVG does not specify weights as input for index construction. It aims to establish similarity relationships between objects based on different vector combinations. The core operation in this process is the similarity measurement of objects. Previous studies [43, 52, 64] have demonstrated the effectiveness of computing aggregate distance between objects with a weight of 1. Inspired by this, we adopt a similar strategy to measure object similarity in MVG's construction process. **(2)** We build a single index over all vector combinations, instead of building separate indexes for each one. The reasons for this approach are threefold. Firstly, as shown in Figure 2, separate indexes (i.e., Baseline) result in a large index size due to redundant vector data. Secondly, our evaluation shows that combining neighbor IDs of all vector combinations into a single neighbor list is advantageous for our ID compression (§6.7.2). Lastly, in dynamic update scenarios, a single index simplifies index maintenance, whereas separate indexes introduce complex logic.

3.2 Search Procedure

MVG is capable of answering diverse queries with different vector combinations and weights on a unified index. For any given query, the procedure to retrieve the top-k results is outlined in Algorithm 2. The search initiates at the top layer and iteratively progresses to lower layers until it reaches the bottom layer (lines 1-5). The entry point for the top layer remains fixed (line 1), whereas the entry points for subsequent layers are determined by the nearest vertices to the query in the upper layers (lines 2-4). The top-k results are derived by executing the search at the bottom layer (§5.1). Throughout the process, the neighbor visit adapts to the vector combination of the query. For every vertex visited, only the neighbors that match the query's vector combination are decompressed and explored. The computation of distances between the query and the visited vertices is based on the query's vector combination and weights. We identify two types of distance computations and implement two specialized acceleration strategies tailored to their features to enhance the search speed without sacrificing accuracy (§5.2).

Example 4. Figure 3 (right) illustrates how MVG uniformly processes four queries (queries i-iv) with different vector combinations or weights. For simplicity, we only showcase the search procedure on the graph structure of a specific layer. Each query leads to unique neighbor visits, and only a subset of neighbor IDs corresponding to the query's vector combination are decompressed. Specifically, query i exclusively visits the neighbors linked to vector combination C0. It starts from vertex 0, measures the distances from the neighbors of vertex 0 to the query using their first vectors, and then moves to vertex 1, which is closer to the query. This sequence continues until no closer neighbors are located, at which point the final result is returned. Although both query iii and query iv interact with neighbors corresponding to vector combination C2, their different weights guide them along specialized search paths. For example, when visiting the neighbors of vertex 0, query iii jumps to vertex 2, while query iv moves to vertex 8.

Remarks. Our central objective is to perform an adaptive search procedure for queries with varying vector combinations and weights on a weight-independent multi-vector index. This leverages the fundamental principle of graph index: "a neighbor's neighbor is likely to be a neighbor as well". Given a query, even when the search starts from a vertex that is far away from the query, it can explore the vertex's neighbors and hop to closer neighbors, and continue visiting their neighbors. The graph index's greedy search mechanism guides this exploration, ultimately converging to vertices with small distances from the query.

4 INDEX CONSTRUCTION

In this section, we explore the index construction of MVG. Initially, we elucidate the basic construction process. Next, we introduce three indexing-aware acceleration techniques aiming at enhancing construction efficiency. Lastly, to minimize the index size, we refine the index layout and implement three compression algorithms.

Algorithm 3: 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 Alg. 2
4 $C \leftarrow$ select top-c_1 nearest neighbors; ▷ refer to Alg. 4
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** $\forall y \in R', f(x,o) < f(x,y)$ **then** ▷ $f(,)$ is the distance
10 $R' \leftarrow R' \cup x$; ▷ heuristic neighbor selection
11 $R \leftarrow R \cup R'$; ▷ merge all neighbors
12 **return** R

4.1 Basic Process

MVG builds its index by progressively adding objects. As outlined in Algorithm 1, each new object o is assigned a maximum layer l_{max} (≥ 0) determined by an exponentially decaying probability distribution. If l_{max} exceeds the graph index's current highest layer L, MVG updates L to l_{max}. Subsequently, o is inserted from layer l_{max} down to layer 0 as per Algorithm 3. In the process of insertion at a given layer l, MVG explores all vector combinations to identify their entry points, similar candidates, and eventual neighbors (lines 2-10). Note that the distance between vertices varies according to the vector combination used (see Example 3). When handling a specific vector combination, MVG treats it as a query with equal weights to all vectors. The entry point at layer l is determined by executing a greedy search at layer $l+1$, using the nearest neighbor as the entry point (line 3). Candidates at layer l are then identified by initiating a greedy search from this entry point (line 4). Ultimately, neighbors are selected from these candidates that are closer to o than their distances to any previously chosen neighbors of o, adhering to the heuristic rule of HNSW [44] (lines 5-10).

Complexity Analysis. To identify the top-c_1 candidates, MVG explores $c_1 \cdot \theta(n)$ vertices [4]. Additionally, it visits $c_1 \cdot r$ neighbors to generate the final set of r neighbors. According to Lemma 2, each vector within an object participates in 2^{m-1} traversals. Let D_m denote the total dimension of all vectors in an object. The insertion time complexity for an object is $O((\theta(n)+r) \cdot c_1 \cdot D_m \cdot 2^{m-1})$. The space cost associated with 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, when selecting one vector, there are 2^{m-1} combinations involving this vector, as each of the other $m-1$ vectors can be either included or excluded. The detailed proof can be found in our technical report [8]. □

4.2 Indexing-Aware Acceleration

Multi-vector distance computation profoundly impacts index construction efficiency. It is necessary for determining entry points, candidates, and final neighbors. Our evaluation on the MIT-States dataset reveals that these computation operations account for *77.8%* of the overall construction time. Clearly, distance computation represents a major bottleneck, constraining construction efficiency. To address this, we delve into the characteristics of various multi-vector distance calculations in index construction and classify them into three categories. Subsequently, we propose three acceleration methods based on these features to enhance the construction process while maintaining index quality.

4.2.1 ***Computation Reuse.*** For each inserted object o, it may visit vertex x multiple times for different vector combinations. For example, it computes $\delta(\boldsymbol{o_0}, \boldsymbol{x_0})$ for vector combination C0 (Example 3), and then computes $\delta(\boldsymbol{o_0}, \boldsymbol{x_0})$ again for C1 to obtain the aggregate distance. To avoid such redundant computations, MVG reuses $\delta(\boldsymbol{o_0}, \boldsymbol{x_0})$. Initially, it identifies the final neighbors for the combinations with one vector and caches the single-vector distance between o and visited vertices. For subsequent combinations involving more than one vector, MVG checks the cached distance and, if matched, uses the value to expedite multi-vector computation. This computation reuse enhances construction efficiency by optimizing D_m in the complexity formula. On MIT-States, this optimization reduces computations for each inserted object by an average of *72.1%*.

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 reusing the cached single-vector distance. This indicates a significant overlap between visited vertices for single-vector and multi-vector combinations. Notably, the state-of-the-art graph index only requires 50% similar neighbors [59]. Consequently, we can directly generate candidates for multi-vector combinations from the visited vertices when getting the candidates for single-vector combinations. This approach further reuses distance computations, leading to more efficient index construction. The insertion time complexity for an object now is $O((\theta(n) \cdot m + r \cdot 2^{m-1}) \cdot c_1 \cdot D_m)$, which is lower than before.

4.2.2 ***Approximate Computation.*** During index construction, most distance values are compared against a threshold T. For instance, if the distance between a visited vertex x and the inserted object o is less than T, x is added to the candidate set C. Here, T represents the farthest distance from o for the elements in C. Additionally, all distance computations for final neighbor acquisition involve comparisons (line 9 in Algorithm 3). On MIT-States, comparisons make up *90.4%* of the total computations. Out of these, *83.4%* can be successfully performed using partial distances. For comparison purposes, an exact distance is often unnecessary. To optimize this, it can incrementally scan a vector combination's dimensions and stop when the partial distance exceeds T or when all dimensions have been scanned. This approach reduces the time complexity of distance computation. However, for high-dimensional vector combinations, values in different dimensions contribute differently to the final distance. Therefore, incremental scanning from low to high dimensions might involve many low-contribution values.

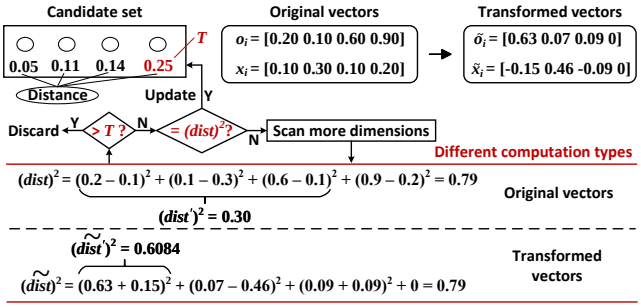


Figure 4: An example of approximate computation.

Example 5. In Figure 4, the candidate set's distance threshold T is 0.25. For a visited vertex x, it computes the accurate square distance $(dist)^2$ and compares it with T. If $(dist)^2 > T$, x is discarded; otherwise, it updates the candidate set with x. This requires scanning all four dimensions. Alternatively, computing the partial distance $(dist')^2$ by incrementally scanning vectors circumvents full-dimension scanning. Here, scanning the first three dimensions suffices for a correct comparison. However, checking the fourth dimension directly, i.e., $(dist')^2 = (0.9 - 0.2)^2 = 0.49$, also leads to a correct comparison, as the last dimension contributes the most. Thus, scanning the key dimensions first is vital for better efficiency.

MVG transforms vectors based on the relative importance of their values, assigning higher importance to lower dimensions. For a set of objects S, each object o has m vectors, with the i-th vectors forming 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. The vector transformation is executed through the following steps.

(i) Calculate mean vector $\boldsymbol{\mu_i}$:

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

(ii) Centralize vectors in S_i:

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

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

(iii) Compute covariance matrix $\Sigma_{\boldsymbol{i}}$:

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

(iv) Determine eigenvalues and eigenvectors of $\Sigma_{\boldsymbol{i}}$: $(\lambda_0, \cdots, \lambda_{d_i-1})$ and $(\boldsymbol{z_0}, \cdots, \boldsymbol{z_{d_i-1}})$. Each eigenvector is normalized to unit length, and is orthogonal to all other eigenvectors.

(v) Sort eigenvalues and construct projection matrix $\boldsymbol{M_i}$:

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

where $\boldsymbol{z_0}$ corresponds to the largest eigenvalue λ_0, $\boldsymbol{z_1}$ corresponds to the second largest eigenvalue λ_1, and so on.

(vi) Apply $\boldsymbol{M_i}$ to transform vectors in S_i: Obtain the transformed data set $\tilde{S_i}$ with the same dimensionality. For a vector $\boldsymbol{o_i}$, calculate the transformed vector $\tilde{\boldsymbol{o}}_{\boldsymbol{i}}$:

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

The above steps are conducted for all S_i with $0 \le i \le m-1$. In our implementation, we randomly sample a subset of vectors (1%) from S_i to establish the projection matrix $\boldsymbol{M_i}$, enhancing vector transformation efficiency while maintaining effectiveness.

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 σ_j^2. Larger σ_j^2 dominates the distance magnitude, indicating a more distinct position in the j-th dimension for vector pairs.

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

Proof. (*Sketch.*) The transformed data $\tilde{\boldsymbol{S_i}}$ are zero-mean. We have $Var(\tilde{\boldsymbol{S_i}}) = E(\tilde{\boldsymbol{S_i}}^2)$ by the identity $Var(\tilde{\boldsymbol{S_i}}) = E(\tilde{\boldsymbol{S_i}}^2) - (E(\tilde{\boldsymbol{S_i}}))^2$, where $Var(\tilde{\boldsymbol{S_i}})$ is the variance of $\tilde{\boldsymbol{S_i}}$. $\tilde{\boldsymbol{S_i}}[j] = \boldsymbol{M_i}^\top[j]\hat{\boldsymbol{S_i}}$ is the j-th dimension of $\tilde{\boldsymbol{S_i}}$. 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 λ_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:

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

Δ_j is proportional to the square root of the eigenvalue. Thus, $(\Delta_j)^2 \ge (\Delta_{j+1})^2$. 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. This 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 $dist \le T$. Kindly refer to our technical report for the detailed proof [8]. □

Example 6. Figure 4 showcases an example of two transformed vectors obtained by applying $\boldsymbol{M_i}$ to the original vectors. In the transformed vectors, the first dimension is of primary significance, succeeded by the second, and so forth. 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.

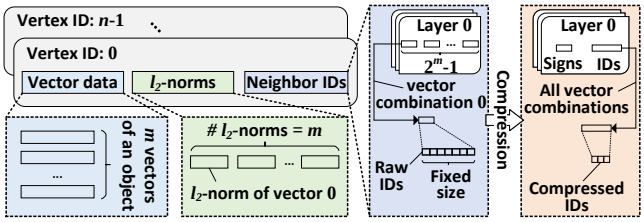


Figure 5: Overview of index layout.

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 connection between the square Euclidean distance and the 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 the precomputed $||\boldsymbol{o_i}||^2$ 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 MIT-States shows 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 facilitate unified index management, MVG consolidates all data into a single index. As illustrated in Figure 5, the index organizes m vectors, m l_2-norms, and the neighbor IDs for $2^m - 1$ vector combinations for each vertex in a sequential manner. A equal amount of storage space is allocated to store each object's vector data and l_2-norms, enabling sequential access through offsets. Note that the l_2-norms play a crucial role in accurate computations, as discussed in §4.2.3. In terms of neighbor IDs, each layer is stored separately in a consistent pattern. Every layer contains $2^m - 1$ lists of neighbor IDs for all vector combinations, with each list being of consistent size to allow direct access through offsets. Overall, this layout promotes rapid data access, including accessing vector data and l_2-norms by vertex ID, as well as neighbor IDs depending on the vertex ID and the specific vector combination of the query.

Despite the careful design of the index layout, the proliferation of neighbor IDs per vertex due to a multitude of vector combinations escalates storage cost. To address this challenge, MVG integrates a neighbor ID compression module, sorting IDs and appending a sign to each ID signifying the vector combination. A mere $\lceil \log_2(2^m - 1) \rceil$ bits are required for each sign. Subsequently, we delve into three compression algorithms for neighbor ID lists.

4.3.1 ***Sequential ID (Seq-ID).*** Seq-ID compresses a neighbor ID list by retaining the first ID and calculating sequential ID differences, hence storing only the first ID and these differences. Equal bits are allocated for each difference, facilitating access. ID recovery incurs a time complexity of $O(h)$, where h signifies neighbor list length.

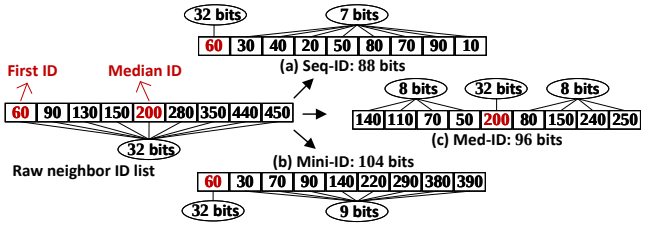


Figure 6: Illustration of neighbor ID list compression.

Example 7. Figure 6(a) illustrates the compression process using Seq-ID for an ordered neighbor list. Prior to compression, nine IDs require 288 bits (each ID being 32 bits). With Seq-ID, the maximum difference is 90, requiring 7 bits per difference and 32 bits for the first ID. Consequently, 88 bits suffice to store the entire list. In the worst case, all IDs must be decompressed to access a raw ID.

4.3.2 ***Minimum ID (Mini-ID).*** Mini-ID simplifies ID access by avoiding the need to decompress irrelevant IDs. It achieves this by computing differences between the first ID and all other IDs. Thus, it stores the first ID and these differences. Mini-ID enables raw ID access in $O(1)$ time complexity, but at a higher space cost.

Example 8. Figure 6(b) depicts the compression of a neighbor list using Mini-ID. The maximum difference is 390, requiring 9 bits per difference. Consequently, 104 bits are needed to store the list, which exceeds the storage requirement of Seq-ID. Nevertheless, this method enables direct raw ID recovery.

4.3.3 ***Median ID (Med-ID).*** Med-ID balances decompression efficiency and compression ratio by subtracting smaller IDs from the median ID and the median ID from larger IDs to calculate differences. Thus, it stores the median ID and the differences. Med-ID recovers a raw ID in $O(1)$ time complexity and uses less storage space due to smaller differences compared to Mini-ID.

Example 9. As illustrated in Figure 6(c), the median ID is 200, the maximum difference is 250, requiring 8 bits for each difference storage. The total bits needed to store such a neighbor list is 96. Hence, Med-ID achieves a better trade-off between decompression efficiency and compression ratio.

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. □

Remarks. In MVG, index compression optimizes the $(2^m - 1) \cdot r$ term of the complexity formula, effectively reducing storage space. For example, on the MIT-States dataset, neighbor lists' space saving hits *52.1%*, *51.7%*, and *51.9%* for Seq-ID, Mini-ID, and Med-ID, respectively. While Seq-ID follows popular delta encoding methods such as PforDelta [37, 65, 79], it incurs notable online decompression overhead during query execution. The proposed Mini-ID and Med-ID avoid unnecessary decompression and employ rapid bit operations, thus minimally affecting search efficiency.

5 SEARCH PROCEDURE

In this section, we explore the search procedure for the unified index, which includes both the original index and the compact index. We provide an overview of the fundamental process and discuss optimizations related to query-aware acceleration.

Algorithm 4: SEARCH TOP-k OBJECTS 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 objects of q

```
1  C ← e; R ← e; H ← e;   ▷ candidate set, result set, and visit set
2  while |C| > 0 do
3      x ← extract nearest vertex to q from C;
4      y ← get furthest vertex to q from R;
5      N ← neighbors of x corresponds to q's vector combination;
6      if y is closer to q than x then
7          break;        ▷ all vertices in R are closer than those in C
8      for each neighbor p in N do
9          if p ∉ H then
10             H ← H ∪ p;
11             y ← get furthest vertex from R to q;
12             if |R| < k or p is closer to q than y then
13                 C ← C ∪ p; R ← R ∪ p;
14                 if |R| > k then
15                     remove furthest element to q from R;
16 return R
```

5.1 Basic Process

Recall that MVG constitutes a multi-layer graph structure. The search initiates at the top layer, employing a greedy approach to locate the nearest vertex. Each layer's closest vertex serves as the entry point for the subsequent layer. This iterative process continues until reaching the base layer, then the top-k results are returned. We elucidate the search process at a specific layer by Algorithm 4.

It initializes the candidate set C and the result set R with the entry point e, while tracking the visited elements using set H to avoid repetitive access (line 1). It extracts the nearest vertex x to the query q from C (line 3) and the furthest vertex y to q from R (line 4). If y is closer to q than x, the search terminates, as all vertices in R are closer to q than those in C (lines 6-7). Otherwise, it visits x's each unvisited neighbor p based on q's vector combination (line 5) and updates C and R with p (lines 8-15). This process iterates 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. The decompression of neighbor IDs 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.

5.2 Query-Aware Acceleration

In Algorithm 2, two types of distance computations are performed. The first type involves comparison with a threshold (it is the distance between q and the furthest vertex in R). The second type requires an accurate distance value (when R is not filled). On MIT-States, the proportion of search time taken up by distance computation is *91.3%*, of which *82.1%* is for comparison and *17.9%* for accurate distance. Evidently, we can still leverage the optimizations of approximate and accurate computations from §4.2.2 and §4.2.3.

We introduce a new acceleration optimization technique that exploits the unique aspect of a multi-vector query. This idea stems from the observation that different vectors within a multi-vector query carry different weights. Consequently, we incorporate weight information into the approximate computation process. Specifically, MVG arranges the distance computation for each vector pair by their weights, prioritizing the scanning of vectors with higher weights. Our evaluation demonstrates that this optimization yields a *29.5%* reduction in computation on MIT-States, 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 at a scale of $c_2 \cdot \theta(n)$. Let D_t denote the total dimension of all vectors in a multi-vector query. The time complexity for getting the 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	D_m	$\#q$	t	w_i
Recipe	1.3M	2	2,048	10^4	2	0.1/0.2
MIT-States	2.1M	6	3,456	10^3	6	0.2/0.15/0.2/0.1/0.2/0.15
CelebA	1M	4	2,304	10^3	4	0.2/0.3/0.4/0.1
FashionIQ	1M	2	1,024	10^3	2	0.7/0.3
MS-COCO	1M	3	1,536	10^3	3	0.4/0.3/0.3
Shopping	1M	2	1,024	10^3	2	0.7/0.3
MIT-States+	16M	6	3,456	10^3	6	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 utilize six real-world datasets, each containing objects with at least two vectors from different modalities or encoders. As shown in Table 2, these datasets exhibit variations in the number of vectors (m), dimensions (D_m), and scale (n). The ground truth is obtained through a brute-force search using queries. Given that the original data scale is relatively small (e.g., the scale of the CelebA dataset is merely 200K), we expand the datasets using generative models [21, 68], which allows us to create additional samples from the learned distribution of real data.

6.1.2 ***Query Type.*** By default, multi-vector queries consist of the same number of vectors as objects ($t = m$), with each vector assigned a unique weight. Building upon related work [55, 73], we maintain fixed weights for each vector pair across all queries within a specific dataset. Additionally, we explore alternative weight configurations by adjusting weight ratios across all queries and introducing dynamic weights at the per-query level. We also consider single-vector queries, where certain vectors carry zero weight.

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

6.1.3 ***Compared Methods.*** We evaluate six methods, all of which utilize the HNSW algorithm. **(i)** VBase. Developed by Microsoft, it leverages index scanning optimization with multiple single-vector indexes [73]. **(ii)** Milvus. Released by Zilliz, it applies candidate merging optimization with multiple single-vector indexes [55]. **(iii)** Merging. A previous MVSS method for hybrid queries [56, 74], it also relies on multiple single-vector indexes. **(iv)** Baseline. Our naive optimization builds an index for each vector combination in objects (see §2.2.3). **(v)** MVG. It deploys all proposed techniques, including Med-ID algorithms for index compression. **(vi)** MVG*. It is a variant of MVG without index compression.

6.1.4 ***Performance Measure.*** We measure the search efficiency and accuracy by *Latency* and *Recall*, respectively. For R, the top-k result set from an MVSS method, the *Recall* is given by:

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

where R' denotes the exact result. By default, we set k to 10.

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, 59].

6.2 Multi-Vector Query Performance

Table 3 presents the multi-vector query performance of various methods. Our observations are as follows: **(i)** MVG* achieves an optimal balance between accuracy and efficiency. For instance, when compared to the leading VBase, MVG* reduces latency by up to 96.5%, while maintaining a higher recall rate. **(ii)** MVG exhibits higher latency than MVG* due to MVG necessitates an additional decompression step. **(iii)** The last trio of methods surpasses the first trio in the *Recall*-vs-*Latency* trade-off, emphasizing the significance of utilizing multi-vector neighbor relationships. **(iv)** Current search strategy optimizations effectively enhance performance. For example, in comparison to Merging, Milvus improves query accuracy, and VBase outperforms Milvus in both accuracy and efficiency.

6.3 Efficiency of Index Constuction

Figure 7 shows index construction times for different methods. We adjust the parameters of index construction to achieve optimal search performance. Detailed parameter values are documented in the technical report [8]. Our observations are as follows: **(i)** MVG* is fastest due to its smaller maximum number of neighbors (r) and indexing-aware acceleration component. MVG is slightly slower due to an extra compression step. **(ii)** Single-vector indexes (including Merging, Milvus, VBase) require a larger candidate neighbor set (c_1) and a greater maximal number of neighbors (r) than multi-vector indexes (including MVG* and MVG) to achieve optimal search performance. **(iii)** Baseline has the longest construction time, as it needs to build indexes for numerous vector combinations.

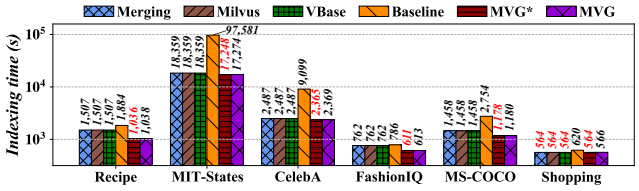


Figure 7: Efficiency of index construction.

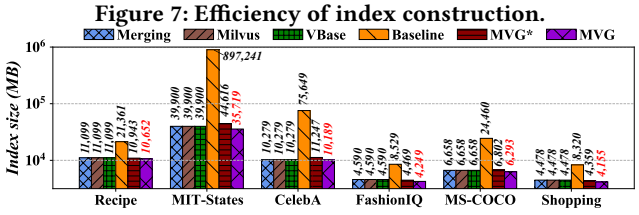


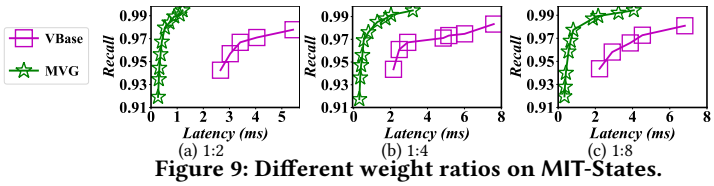
Figure 8: Index size.

6.4 Index Size

Figure 8 shows index sizes of different methods, following the parameter setup outlined in §6.3. Key findings are as follows: **(i)** MVG exhibits the smallest index size due to two factors: optimal search performance requires a smaller number of neighbors (r) and effective index compression. Specifically, index size comprises neighbor size (space cost of neighbor IDs) and vector size (space cost of vector data). Given that VBase and MVG share the same vector size, the difference in index size is determined solely by neighbor size. The space complexities of VBase and MVG for neighbor size are $O(mrn)$ and $O((2^m - 1)rn)$, respectively, where m represents the number of modalities and n denotes the number of objects. Larger r necessitates more storage space. Our evaluation [8] demonstrates that VBase requires significantly larger r than MVG. **(ii)** Baseline has the highest index size. Recall that a vertex's vector data and neighbor IDs are accessed simultaneously in HNSW. To maintain this feature, it stores vector data and neighbor IDs together on an individual index of each vector combination. Consequently, each vector within an object is stored 2^{m-1} times across the indexes of all combinations (see Lemma 2), resulting in high space cost.

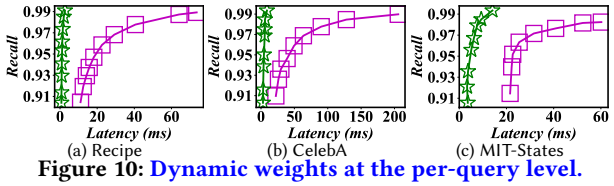
6.5 Query Workloads

In this section, we compare MVG and VBase across different query workloads, maintaining identical HNSW settings for both. We exclude other existing methods (e.g., Merging and Milvus) due to their pronounced limitations in efficiency and accuracy.



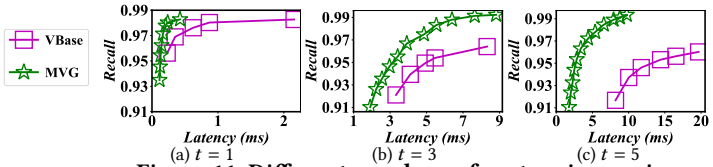
(a) 1:2 (b) 1:4 (c) 1:8

Figure 9: Different weight ratios on MIT-States.



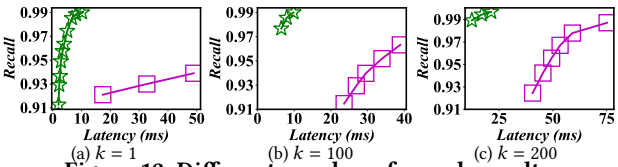
(a) Recipe (b) CelebA (c) MIT-States

Figure 10: Dynamic weights at the per-query level.



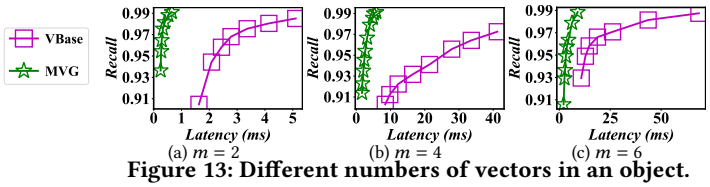
(a) $t = 1$ (b) $t = 3$ (c) $t = 5$

Figure 11: Different numbers of vectors in queries.



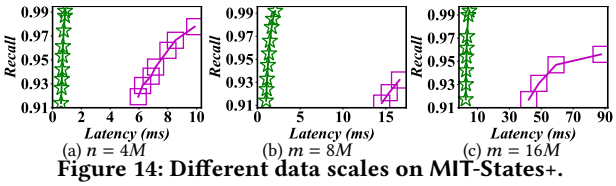
(a) $k = 1$ (b) $k = 100$ (c) $k = 200$

Figure 12: Different number of search results.



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

Figure 13: Different numbers of vectors in an object.



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

Figure 14: Different data scales on MIT-States+.

6.5.1 ***Weight Ratio.*** We adjust the weight ratio for each dual-vector query on MIT-States. Figure 9 demonstrates that MVG consistently outperforms VBase. Notably, MVG exhibits superior performance when the weight bias is minimal. Prior research has indicated that minor weight bias is prevalent in real-world scenarios [56].

6.5.2 ***Dynamic Weight.*** We evaluate the search performance of VBase and MVG using queries with dynamic weights at the per-query level. Figure 10 shows the results across three datasets. Our findings align with those from our previous experiments: MVG consistently outperforms VBase by a substantial margin. For instance, at the same recall rate of 0.99 on Recipe, MVG is 31 × faster than VBase. This further underscores the robustness of MVG across various scenarios.

6.5.3 ***Number of Query Vectors (t).*** In Figure 11, we analyze the performance of VSS ($t = 1$) and MVSS ($t > 1$). The results show that MVG is more superior in both query scenarios. Notably, the performance gap between MVG and VBase widens as t increases.

6.5.4 ***Number of Results (k).*** Figure 12 illustrates the search performance of MVG and VBase on MIT-States for varying values of k. MVG proves more robust than VBase across all values of k.

6.6 Scalability

In this section, we evaluate the scalability of MVG on MIT-States across varying numbers of vectors per object and data scales.

6.6.1 ***Number of Vectors in an Object (m).*** In Figure 13, we adjust the parameter m and evaluate the search performance on MIT-States. The results demonstrate that MVG consistently outperforms VBase, particularly in the high recall rate region. Additionally, Table 10 presents the variations in construction time and index size as m increases. Notably, the number of neighbors (r) significantly impacts both construction time and index size. We provide results for both VBase and MVG with r corresponding to their respective optimal search performance. For further reference, our technical report [8] includes detailed results for identical r values. The findings reveal that as m increases, the neighbor size and construction time for MVG grow more rapidly than those for VBase at the same r. Consequently, MVG exhibits a larger index size and construction time compared to VBase when m is large (e.g., $m > 3$). However, under optimal search performance, VBase requires a larger index size and construction time due to a significantly larger r than MVG.

Table 4: Index size and construction time with varying numbers of modalities (m) on MIT-States.

m	Index size (MB)		Neighbor size (MB)		Construction time (s)	
	VBase	MVG	VBase	MVG	VBase	MVG
1	5,380	5,283	295	198	1,121.92	774.86
2	10,250	9,613	1,098	460	2,908.09	1,536.56
3	15,884	15,227	1,646	990	3,793.87	3,109.47
4	22,533	20,360	4,228	2,055	8,360.92	5,041.57
5	33,759	27,536	6,301	4,146	15,934.00	9,276.43
6	39,900	35,719	12,443	8,261	18,359.40	17,274.03

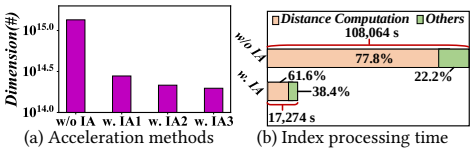
6.6.2 ***Data Scale (n).*** Figure 14 illustrates the search performance of MVG and VBase for different values of n. While the latency of VBase increases linearly with n, MVG exhibits only a minor latency increase even with large n values. Additionally, MVG demonstrates a significant accuracy superiority over VBase.

6.7 Ablation Study

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

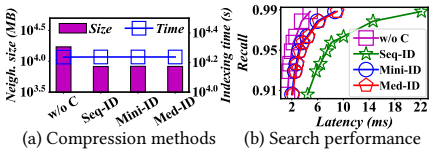
6.7.1 ***Indexing-aware Acceleration.*** Figure 15(a) records the total dimension required for calculating the square Euclidean distance during index construction. We progressively integrate acceleration methods into the original construction process. Specifically, *w/o IA* denotes the process without acceleration, *w. IA1* represents the enhanced process with computation reuse, *w. IA2* further incorporates approximate computation optimization, and *w. IA3* includes accurate computation optimization. Our three acceleration methods significantly reduce the evaluated dimension. In Figure 15(b), we observe that the original process (*w/o IA*) necessitates substantial construction time, primarily due to extensive distance computation. Conversely, when all three acceleration methods are applied (*w. IA*), MVG considerably reduces the index processing time.

6.7.2 ***Index Compression.*** We investigate the impact of index compression on neighbor size, index processing time, and search performance. Figure 16(a) demonstrates that our compression techniques reduce neighbor size by nearly 50%. Furthermore, the index compression phase operates fast, with merely 0.11%~0.21% of the



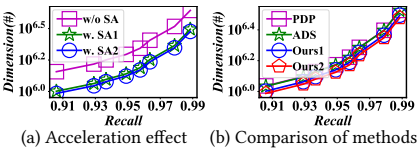
(a) Acceleration methods (b) Index processing time

Figure 15: Indexing-aware acceleration.



(a) Compression methods (b) Search performance

Figure 16: Index Compression.



(a) Acceleration effect (b) Comparison of methods

Figure 17: Query-aware acceleration.

Table 5: 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	$O((\theta(n)m+2^{m-1}r)c_1D_mn)$	$O(((2^m-1)r+D_m)n)$	$O(c_2D_t\theta(n))$

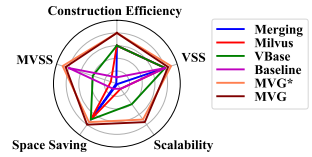


Figure 18: Ability Comparison.

total index processing time. In Figure 16(b), we observe that Seq-ID exhibits higher query latency than the uncompressed method, attributed to the decompression of all neighbor IDs for each visited vertex. In contrast, both Mini-ID and Med-ID deliver search performance comparable to the uncompressed version.

6.7.3 ***Query-Aware Acceleration.*** Figure 17 depicts the dimension used for calculating the square Euclidean distance in the search process. Figure 17(a) shows our search acceleration techniques (*w. SA1* and *w. SA2*) reduce the dimension by 32.9% compared to the original method (*w/o SA*). This notable decrease results from the approximate computation acceleration (*w. SA1*), accounting for dimension importance. Despite the small fraction of accurate computations, our method (*w. SA2*) still brings significant improvement. Figure 17(b) compares our methods to two state-of-the-art techniques: PDP [48] and ADS [25], with ours surpassing both. Employing weight-based optimization (Ours2) further reduces dimension based on initial approximate computation optimization (Ours1).

7 SUMMARY

We summarize our experiments and delve into possible optimizations and limitations of MVG. Table 5 catalogs the complexities of different methods, and Figure 18 compares their overall capabilities.

Overall Performance. Figure 18 uses a radar chart to visualize the overall capabilities of all evaluated methods. Construction efficiency is gauged using construction time; less time equates to a higher score. Space saving is measured by index size; smaller size scores higher. VSS and MVSS are scored on the trade-off between efficiency and accuracy, with better trade-offs scoring higher. Scalability is assessed on construction time, index size, and search performance as the number of vectors per object and date scale increase, with smaller times and sizes, and better performance scoring higher. Our proposed methods, MVG* and MVG, exhibit the best overall capabilities, offering efficient index processing, compact layout, and high search efficiency and accuracy for both MVSS and VSS, with excellent scalability for handling more vectors and larger data scales. Other methods show limitations in various aspects.

Potential Optimizations. VSS methods' advancements offer potential optimizations for MVG. **(i)** Using GPU's parallel computation could accelerate MVG's index construction and multi-vector search. **(ii)** Deploying MVG on a high-performance SSD could optimize index layout and I/O operations for billion-scale data. **(iii)** Machine learning algorithms could be integrated into MVG to predict next steps or termination condition during multi-vector searches.

Limitations. Table 5 reveals that MVG's construction time and space cost grow exponentially with the number of vectors m, resulting in excessive overhead for larger m (e.g., $m > 10$). Notably, we observe that most real-world situations require a smaller m such as $m = 2$ [16, 20, 33]. For instance, a multi-modal object has text and image modalities [36]; in video surveillance, three vectors represent a person's front face, side face, and posture [55]. Current research only evaluates the case of $m = 2$ [55, 73]. For larger m, current methods need higher offline overhead for optimal search performance. Thus, as m increases, current methods and MVG face overhead challenges. We mark this as an open problem for future research.

8 RELATED WORK

Vector Search Algorithms. Current algorithms is categorized into Tree-based [19, 42, 45], Hashing-based [26, 29, 39], Quantization-based [14, 28, 35], and Graph-based methods [24, 41, 44]. Graph-based methods achieve the state-of-the-art balance of efficiency and accuracy [40]. Various optimizations focus on different aspects, like external storage [34, 58], GPU acceleration [54, 76], and learning to route [38, 71]. However, current algorithms face performance bottlenecks for MVSS due to the drawback of single-vector index.

Vector Databases. Vector databases, built on vector search algorithms, offer versatile features for industrial applications [27]. Essential for unstructured data management with the surge of large language models (LLMs) [6, 75], many vector databases like Milvus [12], Pinecone [10], and Weaviate [10] effectively serve VSS tasks. However, for MVSS on unstructured data across multiple modalities [69], existing databases lack adequate support [47, 63].

Multi-Vector Search. Recent surveys [47, 51, 53, 63] and industrial scenarios [5, 7, 11, 13, 27] emphasize the requirement for efficient and accurate MVSS solutions. Existing methods handle MVSS by refining search strategies on multiple single-vector indexes [55, 73], enhancing performance but still facing efficiency, accuracy, and scalability challenges due to the limitations of single-vector index.

9 CONCLUSION

We study the MVSS problem and introduce an innovative solution, MVG. Our method integrates all vector combinations within a well-designed index, efficiently supporting both VSS and MVSS. We develop computation acceleration techniques and index compression algorithms, encompassing them into MVG to facilitate rapid index processing, a compact index layout, and efficient search. Furthermore, we bolster 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. *Proceedings of the VLDB Endowment (PVLDB)* 9, 4 (2015), 288–299.

[15] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based Language Models and Applications. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*. 41–46.

[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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4959–4968.

[17] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. 2018. Multimodal machine learning: A survey and taxonomy. *IEEE Transactions on Pattern Analysis and Machine Intelligence (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 *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM)*. 3013–3022.

[19] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *Proceedings of the 40th annual ACM Symposium on Theory of Computing (STOC)*. 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 *The Tenth International Conference on Learning Representations (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 *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 102–113.

[23] Yue Fan and Xiuli Ma. 2022. Multi-Vector Embedding on Networks with Taxonomies. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (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. *Proceedings of the VLDB Endowment (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. *Proceedings of the ACM on Management of Data (PACMMOD)* 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. *Proceedings of the VLDB Endowment (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. *Proceedings of the VLDB Endowment (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 *Proceedings of the 37th International Conference on Machine Learning (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. *Proceedings of the VLDB Endowment (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 *Proceedings of the 13th ACM Conference on Recommender Systems (RecSys)*. 169–177.

[31] Yefan Huang, Feng Luo, Xiaoli Wang, Zhu Di, Bohan Li, and Bin Luo. 2023. A one-size-fits-three representation learning framework for patient similarity search. *Data Science and Engineering (DSE)* 8, 3 (2023), 306–317.

[32] Phillip Isola, Joseph J Lim, and Edward H Adelson. 2015. Discovering states and transformations in image collections. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 1383–1391.

[33] Surgan Jandial, Pinkesh Badjatiya, Pranit Chawla, Ayush Chopra, Mausoom Sarkar, and Balaji Krishnamurthy. 2022. SAC: Semantic attention composition for text-conditioned image retrieval. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. 4021–4030.

[34] 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 *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS)*. 13748–13758.

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

[36] Zhi Lei, Guixian Zhang, Lijuan Wu, Kui Zhang, and Rongjiao Liang. 2022. A multi-level mesh mutual attention model for visual question answering. *Data Science and Engineering (DSE)* 7, 4 (2022), 339–353.

[37] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience (SPE)* 45, 1 (2015), 1–29.

[38] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2539–2554.

[39] 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 *IEEE International Conference on Data Engineering (ICDE)*. 289–300.

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

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

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

[43] Chuwei Luo, Changxu Cheng, Qi Zheng, and Cong Yao. 2023. GeoLayoutLM: Geometric Pre-training for Visual Information Extraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7092–7101.

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

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

[46] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based news recommendation for millions of users. In *Proceedings of the 23rd ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 1933–1942.

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

[48] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 1 (2023), 54:1–54:27.

[49] 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 *Proceedings of the 38th International Conference on Machine Learning (ICML)*. 8748–8763.

[50] 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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 3020–3028.

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

[52] Ryota Tanaka, Kyosuke Nishida, Kosuke Nishida, Taku Hasegawa, Itsumi Saito, and Kuniko Saito. 2023. SlideVQA: A Dataset for Document Visual Question Answering on Multiple Images. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 13636–13645.

[53] 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. *Bulletin of the Technical Committee on Data Engineering (TCDE)* 47, 3 (2023), 39–54.

[54] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. Fast k-NN Graph Construction by GPU based NN-Descent. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM)*. 1929–1938.

[55] 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 *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2614–2627.

[56] 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 *IEEE International Conference on Data Engineering (ICDE)*.

[57] 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 *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems (NeurIPS)*. 15738–15751.

[58] 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. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 1 (2024), 14:1–14:27.

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

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

[61] 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. *Bulletin of the Technical Committee on Data Engineering (TCDE)* 47, 3 (2023), 3–21.

[62] Jiancan Wu, Xiangnan He, Xiang Wang, Qifan Wang, Weijian Chen, Jianxun Lian, and Xing Xie. 2022. Graph convolution machine for context-aware recommender system. *Frontiers of Computer Science (FCS)* 16, 6 (2022), 166614.

[63] 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).

[64] Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. 2020. LayoutLM: Pre-training of Text and Layout for Document Image Understanding. In *The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1192–1200.

[65] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*. 401–410.

[66] 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 *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems (NeurIPS)*. 21573–21612.

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

[68] 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 *Findings of the Association for Computational Linguistics: EMNLP*. 1008–1025.

[69] 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).

[70] 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 *Proceedings of the 24th ACM International Conference on Knowledge Discovery & Data Mining (KDD)*. 974–983.

[71] 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).

[72] 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. *Bulletin of the Technical Committee on Data Engineering (TCDE)* 47, 3 (2023), 22–38.

[73] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 377–395.

[74] Shaoting Zhang, Ming Yang, Timothee Cour, Kai Yu, and Dimitris N Metaxas. 2014. Query specific rank fusion for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 37, 4 (2014), 803–815.

[75] 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 *International Conference on Asian Language Processing (IALP)*. 258–264.

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

[77] 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. *Proceedings of the VLDB Endowment (PVLDB)* 16, 8 (2023), 1979–1991.

[78] Susu Zheng, Weiwei Yuan, and Donghai Guan. 2022. Heterogeneous information network embedding with incomplete multi-view fusion. *Frontiers of Computer Science (FCS)* 16, 5 (2022), 165611.

[79] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *IEEE International Conference on Data Engineering (ICDE)*. 59.

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 λ_j as the j-th largest eigenvalue of $\boldsymbol{\Sigma_i}$, and σ_j^2 as the variance of the j-th dimension of $\tilde{\boldsymbol{S_i}}$, it holds: $\sigma_j^2 = \lambda_j$.*

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

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

Given that $(\boldsymbol{M_i}^\top[j])^\top$ is the j-th eigenvector of $\boldsymbol{\Sigma_i}$ and λ_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 $\tilde{\boldsymbol{S_i}}$ equals the j-th largest eigenvalue of the covariance matrix $\boldsymbol{\Sigma_i}$. □

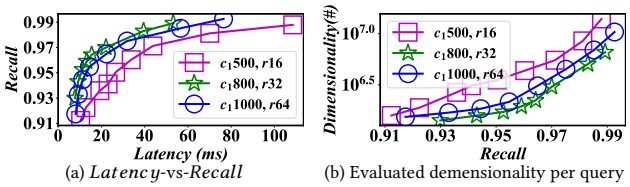
G PROOF OF THEOREM 1

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:

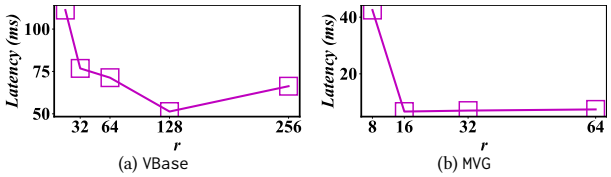
$$(\Delta_j)^2 = (\tilde{\boldsymbol{o_i}}[j] - \tilde{\boldsymbol{x_i}}[j])^2 \quad . \tag{15}$$

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 Δ_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 19: Effect of different parameters.



(a) VBase (b) MVG

Figure 20: Search performance with varying numbers of neighbors on MIT-States.

contribution to the distance between $\tilde{o_i}$ and $\tilde{x_i}$ 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 MIT-States with four vectors per object, as depicted in Figure 19. In our main text experiments, we employ the parameters that yield the best search performance. The detailed parameter values are provided in Table 8.

Figure 20 illustrates the search performance of VBase and MVG with varying r values on MIT-States, with all query latency results recorded at a recall rate of 0.99. We noted that the query latency

Table 6: Neighbor size and index construction time of VBase with varying numbers of neighbors on MIT-States. Bold values indicate the instances with the best search performance.

#Neighbors (r)	Neighbor size (MB)	Construction time (s)
16	1,770	3,428.53
32	3,292	6,856.93
64	6,342	9,812.80
128	**12,443**	**18,359.40**
256	24,646	22,515.60

Table 7: Neighbor size and index construction time of MVG with varying numbers of neighbors on MIT-States. MVG* represents the MVG index without compression. Bold values indicate the instances with the best search performance.

#Neighbors (r)	Neighbor size (MB)		Construction time (s)	
	MVG*	MVG	MVG*	MVG
8	9,225	5,161	8,123.45	8,138.12
16	**17,158**	**8,261**	**17,247.68**	**17,274.03**
32	33,144	11,768	34,530.44	34,559.39
64	65,161	14,616	42,031.27	42,075.86

first decreases and then increases with increasing r for both VBase and MVG. This occurs because a smaller r lengthens the search path, while a larger r increases the number of neighbor visits per hop. Consequently, an optimal r value is crucial for balancing search efficiency and accuracy. For VBase, neighbor selection is based solely on an object's single vector in each single-vector HNSW, requiring a larger r to gather more candidates and ensure a high recall rate for MVSS. In contrast, MVG allocates specific neighbors to each vector combination for each vertex, thus needing a smaller r to achieve a high recall rate. r was determined through grid search in our experiments, choosing the r that yielded the best search performance. For instance, the optimal r is 16 for MVG and 128 for VBase on MIT-States. We present the neighbor size and construction time for VBase and MVG under different r values in Table 6 and Table 7, respectively. VBase exhibits a smaller neighbor size than MVG at the same r, while MVG has a smaller neighbor size than VBase at their respective optimal r. Additionally, MVG significantly reduces the neighbor size compared to its non-compressed counterpart (MVG*).

J CONSTRUCTION TIME PROFILE

Table 9 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.

K INDEX SIZE WITHOUT RAW VECTOR STORAGE

Figure 21 illustrates the index sizes of various methods, excluding the storage of raw vectors in its calculations.

Table 8: 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 9: 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

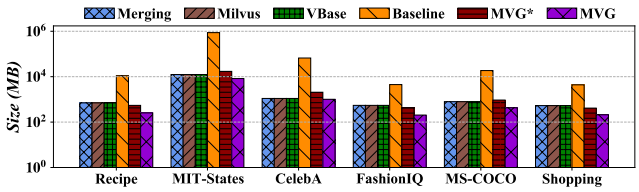
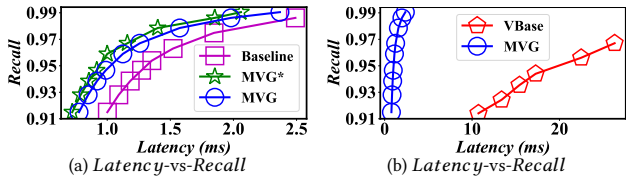


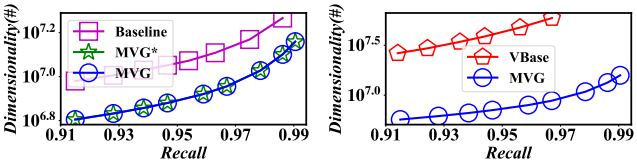
Figure 21: Index size.

L PERFORMANCE CURVES OF MVSS



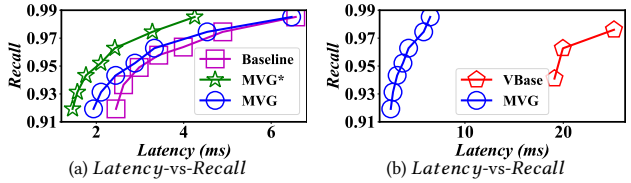
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

Figure 22: *Latency*-vs-*Recall* curve on Recipe.



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

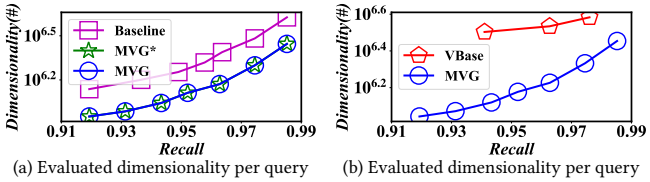
Figure 23: *Dimensionality*-vs-*Recall* curve on Recipe.



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

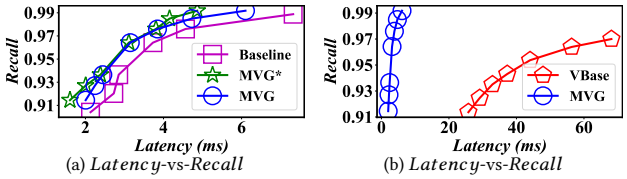
Figure 24: *Latency*-vs-*Recall* curve on MIT-States.

To clearly illustrate the trade-off between search efficiency and accuracy, we present the performance curve in Figure 22–33. 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.



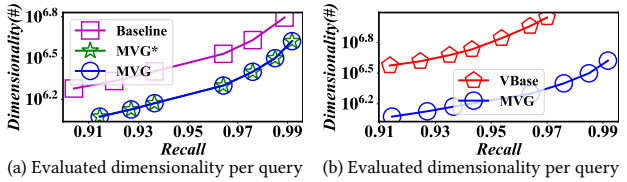
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

Figure 25: *Dimensionality*-vs-*Recall* curve on MIT-States.



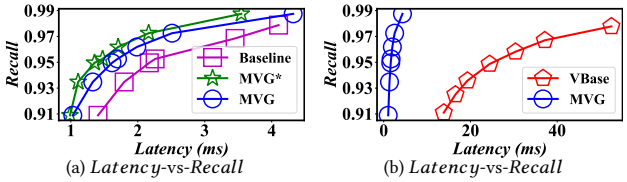
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

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



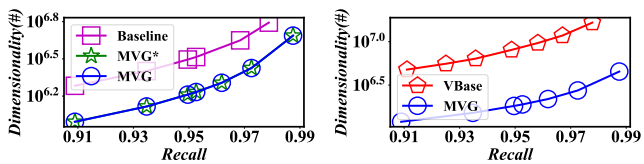
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

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



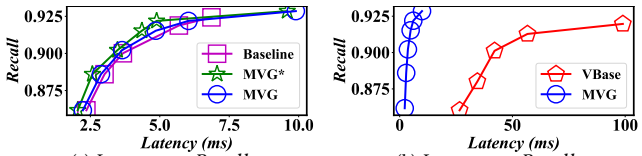
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

Figure 28: *Latency*-vs-*Recall* curve on FashionIQ.



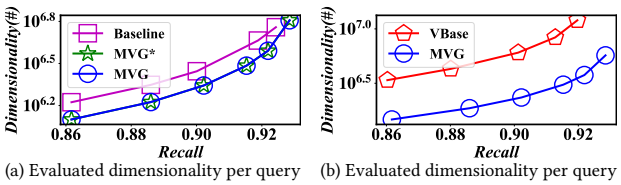
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

Figure 29: *Dimensionality*-vs-*Recall* curve on FashionIQ.



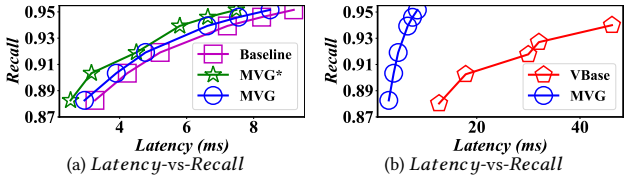
(a) *Latency*-vs-*Recall* (b) *Latency*-vs-*Recall*

Figure 30: *Latency*-vs-*Recall* curve on MS-COCO.



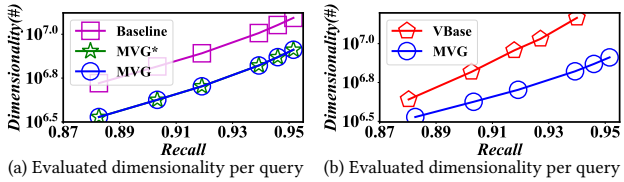
(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query

Figure 31: *Dimensionality*-vs-*Recall* curve on MS-COCO.



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

Figure 32: *Latency*-vs-*Recall* curve on Shopping.



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

Figure 33: *Dimensionality*-vs-*Recall* curve on Shopping.

Table 10: Index size and construction time with varying numbers of modalities (m) on MIT-States. The results are obtained under the number of neighbors that yield the best search performance.

m	Index size (MB)		Neighbor size (MB)		Construction time (s)	
	VBase	MVG	VBase	MVG	VBase	MVG
1	5,380	5,283	295	198	1,121.92	774.86
2	10,250	9,613	1,098	460	2,908.09	1,536.56
3	15,884	15,227	1,646	990	3,793.87	3,109.47
4	22,533	20,360	4,228	2,055	8,360.92	5,041.57
5	33,759	27,536	6,301	4,146	15,934.00	9,276.43
6	39,900	35,719	12,443	8,261	18,359.40	17,274.03

Table 11: Index size and construction time with varying numbers of modalities (m) on MIT-States. The results are obtained under the number of neighbors is 16 ($r = 16$).

m	Index size (MB)		Neighbor size (MB)		Construction time (s)	
	VBase	MVG	VBase	MVG	VBase	MVG
1	5,380	5,283	295	198	1,121.92	774.86
2	9,743	9,613	590	460	1,860.16	1,536.56
3	15,123	15,227	885	990	3,002.19	3,109.47
4	19,485	20,360	1,180	2,055	3,974.51	5,041.57
5	24,865	27,536	1,475	4,146	5,120.84	9,276.43
6	29,228	35,719	1,770	8,261	6,632.99	17,274.03

Table 12: Index size and construction time with varying numbers of modalities (m) on MIT-States. The results are obtained under the number of neighbors is 32 ($r = 32$).

m	Index size (MB)		Neighbor size (MB)		Construction time (s)	
	VBase	MVG	VBase	MVG	VBase	MVG
1	5,634	5,336	549	251	1,727.47	1,402.85
2	10,250	9,775	1,098	622	2,908.09	2,709.23
3	15,884	15,626	1,646	1,388	3,793.87	5,593.62
4	20,500	21,217	2,195	2,911	4,707.66	9,333.05
5	26,134	29,293	2,744	5,903	5,957.10	17,146.33
6	30,751	39,226	3,292	11,768	6,856.93	34,559.39

Table 13: Index size and construction time with varying numbers of modalities (m) on MIT-States. The results are obtained under the number of neighbors is 64 ($r = 64$).

m	Index size (MB)		Neighbor size (MB)		Construction time (s)	
	VBase	MVG	VBase	MVG	VBase	MVG
1	6,142	5,378	1,057	293	3,139.19	1,951.86
2	11,267	9,904	2,114	752	5,390.97	3,514.92
3	17,408	15,962	3,171	1,725	6,609.23	7,719.42
4	22,533	21,895	4,228	3,589	8,360.92	12,547.16
5	28,675	30,724	5,285	7,334	9,872.41	23,381.82
6	33,800	42,073	6,342	14,616	19,338.23	42,075.86

M EFFECT OF DIFFERENT NUMBERS OF VECTORS

Table 10 illustrates how the construction time and index size change as the number of modalities (m) increases on MIT-States. It is important to note that the number of neighbors (r) has a substantial impact on both the construction time and index size. Therefore, we present the results for VBase and MVG, each with their respective r, to ensure optimal search performance. For further reference, we also include the data for construction time and index size with the same r in Tables 11, 12, and 13. The results indicate that with an increase in m, both the neighbor size and construction time for MVG escalate more swiftly compared to VBase for the same r. Consequently, MVG exhibits a larger index size and longer construction time than VBase when m exceeds three. Nonetheless, for optimal search conditions, VBase demands a larger index size and more construction time than MVG for all values of m. This occurs because as m increases, VBase requires a considerably larger r than MVG to achieve the best search performance. Thus, MVG maintains an better equilibrium between the indexing and searching processes across various m values.

N EFFECT OF INDEX COMPRESSION

Table 14 displays the index size, neighbor size, and vector size without compression (notated as "w/o C") and with compression ("w. C"), respectively. We utilized the Med-ID algorithm to compress the index. From the results, our index compression method decreases the neighbor size by a minimum of 49.3%. Table 15 illustrates the effectiveness of index compression considering different m on MIT-States. It is evident that as m escalates, both the percentages of index

Table 14: Index storage statistics on different datasets (Unit: MB). The percentages enclosed in parentheses denote the rate of neighbor size reduction.

Datasets	Index size		Neighbor size		Vector size	
	w/o C	w. C	w/o C	w. C	w/o C	w. C
Recipe	10,943	10,652	545	254 (**53.4%**)	10,398	10,398
MIT-States	44,616	35,718	17,158	8,260 (**51.9%**)	27,458	27,458
CelebA	11,247	10,189	2,068	1,010 (**51.2%**)	9,179	9,179
FashionIQ	4,469	4,249	424	204 (**51.9%**)	4,045	4,045
MS-COCO	6,802	6,293	935	426 (**54.4%**)	5,867	5,867
Shopping	4,359	4,155	414	210 (**49.3%**)	3,945	3,945

Table 15: Index storage statistics of different vector counts per object (Unit: MB). The percentages enclosed in parentheses denote the rate of reduction in index size or neighbor size.

#Vectors (m)	Index size		Neighbor size	
	w/o C	w. C	w/o C	w. C
1	5,388	5,283 (**1.9%**)	303	198 (**34.7%**)
2	10,006	9,613 (**3.9%**)	853	460 (**46.1%**)
3	16,184	15,227 (**5.9%**)	1,946	990 (**49.1%**)
4	22,429	20,360 (**9.2%**)	4,124	2,055 (**50.2%**)
5	31,861	27,536 (**13.6%**)	8,471	4146 (**51.1%**)
6	44,616	35,718 (**19.9%**)	17,158	8260 (**51.9%**)

Table 16: Offline overhead of index compression (Unit: s). The percentages in parentheses represent the proportion of compression time in total index processing time.

Datasets	Total time	Compression time
Recipe	1,037.86	1.66 (**0.16%**)
MIT-States	17,274.03	26.35 (**0.15%**)
CelebA	2,368.53	3.62 (**0.15%**)
FashionIQ	612.76	1.27 (**0.21%**)
MS-COCO	1,180.00	1.59 (**0.13%**)
Shopping	1,123.26	1.26 (**0.11%**)

Table 17: Neighbor size and compression processing time for different compression methods on MIT-States.

	w/o C	Seq-ID	Mini-ID	Med-ID
Neighbor size (MB)	17,158	8213	8,293	8,261
Compression time (s)	0	29.17	28.00	26.35

size and neighbor size reduction amplify. This indicates our compression method can yield more substantial benefits in scenarios involving a greater m.

Table 16 provides the total index processing times and the corresponding index compression times across various datasets. From these results, it is clear that index compression contributes merely 0.11% to 0.21% of the total index processing time. Thus, the index compression is a lightweight component within our index framework. Table 17 presents the neighbor size and compression processing time for various methods. For the sake of comparison, we have also included the results for the counterpart without compression, labeled as "w/o C".

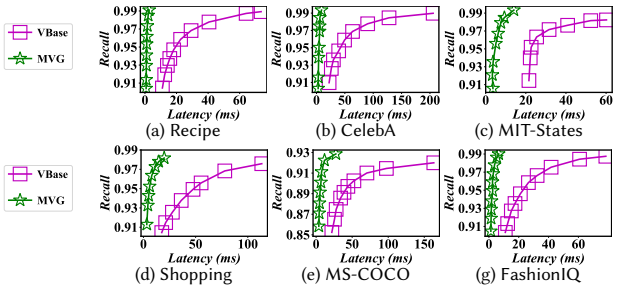


Figure 34: Search performance under dynamic weights at the per-query level.

O DYNAMIC WEIGHT AT THE PER-QUERY LEVEL

Figure 34 presents the results on six different datasets. From these results, we infer the same conclusion as our earlier experiments, i.e., MVG consistently surpasses VBase by a significant margin. For instance, MVG is 31 × faster than VBase at the same recall rate of 0.99 on the Recipe dataset. This further validates the robustness of MVG in various scenarios.