

MSG: Empowering High-Dimensional Multi-Vector Search

ABSTRACT

In the realm of high-dimensional space, Vector Search (VS) 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 Search (MVS), where each object is composed of multiple vectors. Current efforts to address this challenge through VS on multiple single-space indexes often suffer from inefficiency and inaccuracy, primarily due to the intrinsic limitations of single-space indexes.

This study introduces a **Multi-Space Graph** index, named MSG, explicitly designed to tackle the MVS problem. Notably, MSG effectively consolidates all neighbor relationships from different vector spaces within each pair of objects into a single index, enabling seamless processing of any multi-vector query with various vector combinations and weights. Additionally, MSG integrates three computational acceleration methods that leverage MVS-specific characteristics into index construction and search procedures. Three index compression algorithms, grounded in the well-designed index layout, allow MSG 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 MSG outperforms the leading VBase in index construction efficiency, space cost, search performance, and scalability, exhibiting up to 96.5% reduction in query latency while maintaining a higher recall rate.

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 [16, 56, 73]. This paradigm has shown significant promise in various emerging AI applications, sparking interest in Vector Search (VS) within academic and industrial communities. For instance, applications like Bing Chat benefit from large language models (LLMs) that utilize VS components for retrieval augmented generation (RAG), achieved by vectorizing documents [12, 79]. VS has long been a pivotal topic in domains like information retrieval [64, 67], recommendation systems [19, 51, 69], and databases [32, 84]. In VS, given an object set S where each object is represented as a vector,

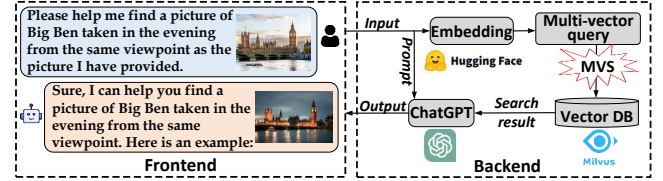


Figure 1: An example of text-image chat by MVS.

a query vector q , and a vector distance metric $\delta(\cdot)$, 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 VS necessitates $O(n \cdot d)$ search time complexity, which becomes prohibitively time-consuming. Therefore, an approximate version¹, also known as Approximate Nearest Neighbor Search (ANNS) [62, 68], is preferred in real-world scenarios. To expedite the search procedure, advanced VS techniques build a vector index, balancing accuracy and efficiency. Numerous studies have indicated that graph-based index (e.g., Hierarchical Navigable Small World graph (HNSW) [49]) is the most promising and has consequently become a mainstream research direction and practical choice in the VS field [39, 66, 87].

To enhance the capabilities of AI applications, Multi-Vector Search (MVS) has been proposed for processing multi-modal or multi-view data [62, 80]. 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 [63]. Additionally, a single-modal object can also construct multiple vectors from different views, with MVS ensuring more accurate search results [25, 36, 85]. Given the significance of MVS, recent research surveys and vector database products emphasize the demand for a fast and accurate MVS solution [32, 53, 58, 60, 70]. As highlighted in a recent tutorial at SIGMOD'24 [52], *MVS requires significant computations and increases query latency, and current VS methods are not readily applicable to this problem, thus new techniques are needed.*

EXAMPLE 1. In a text-image chat scenario, users can input a reference image and instructional text to get the desired result using MVS. 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 (“Frontend”). In the backend, MVS 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 [5]. Subsequently, MVS is executed on a vector database (e.g., Milvus [62]), and the search result is combined with the user input to prompt ChatGPT [1]. Finally, a user-friendly response is displayed in the frontend.

¹Hereafter, we refer to approximate VS simply as VS unless otherwise specified.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '25, June 22–27, 2025, Berlin, Germany

© 2025 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/XXXXXXX.XXXXXXX>

Current research [24, 62, 80, 81] endeavors to address MVS by amalgamating outcomes from VS. The underlying concept posits that a multi-vector query can be processed through VS in multiple vector spaces. For a given set of objects S , vectors corresponding to each modality reside within an identical vector space. Consequently, a single-space index is constructed for each modality, culminating in multiple indexes on S . For instance, there may be an index for the text vector space and another for the image vector space. In response to a multi-vector query, current methods dissect it into several single-vector queries, which are then executed on their respective indexes. Finally, search candidates from each index are reordered based on the aggregate score with query weights to produce query results. Despite the ability to leverage well-established single-space indexes, these methods demonstrate limited efficiency and accuracy as observed in recent literature [62, 80] and our evaluation (§6.2). For example, with million-scale data, the recall rate of the state-of-the-art method falls below **0.90** when the query latency remains under **10ms** on most real-world datasets. In contrast, existing methods for single-vector queries easily attain a recall rate exceeding **0.99** within a mere **1ms** [14, 30].

We identify the primary limitations of current methods as follows. First, they lack indexes for vector spaces in which a multi-vector query is located. For example, when a query has two modalities of text and image, current methods only build single-space indexes for respective vector spaces. However, a multi-vector query comprising text and image inputs resides within a new vector space, formed by concatenating text and image vector spaces. It is widely acknowledged that *a vector index is efficacious only for queries positioned within the same vector space as the indexed vectors* [27, 54, 55]. Therefore, it is insufficient to answer a multi-vector query through VS in vector spaces different from the space where the query is located (**Figure 3**). Second, they lack a search strategy for the dynamic weights inherent to a multi-vector query. To illustrate, consider two multi-vector queries with the same vector data but differing in weights. Current methods implement an identical search procedure on single-space indexes and thus obtain the same candidates for both queries, owing to their *weight-agnostic search strategies*. Obviously, these two queries ought to have different search processes and candidates by virtue of their weight variance (**Figure 3**). In a nutshell, current index and search strategies are incompatible with the MVS feature, presenting an avenue for further optimization.

In this paper, we present a **Multi-Space Graph (MSG)** index, which is specifically designed to enhance the speed and accuracy of MVS. MSG builds a graph structure on a set of objects, establishing neighbor relationships between objects across all possible vector spaces where multi-vector queries may occur. It efficiently answers multi-vector queries within various spaces based on the vector space's neighbor relationships. To further support dynamic weights in multi-vector queries, we develop an adaptable navigation strategy for flexible searches on MSG, considering weights and vector data. Consequently, queries with different weights can start from the same vertex, explore its neighbors, and proceed to respective closer neighbors guided by distinct weights and vector data, ultimately leading to respective query results. With this new design principle, we address the following three research questions.

(i) Efficient construction of a high-quality multi-space index. The assembly of a multi-space index that encompasses all vector

spaces is notably time-consuming. Our evaluation indicates that it takes **31.5** \times more time than building multiple single-space indexes on the MIT-States dataset. To address this, MSG employs three lossless acceleration strategies that leverage the characteristics of graph index construction and multi-vector distance computation.

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

(iii) Fast and accurate processing of any multi-vector query. MSG addresses this challenge by employing a multi-space index that accommodates various neighbor types for different vector spaces and executes adaptive neighbor access during the search process. By exploiting the features of multi-vector queries, MSG 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 specialized index and search strategy that exploits the intrinsic features of MVS. Our evaluations show that MSG 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, MSG 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 MSG, a multi-space graph index that offers efficient index processing and a compact index layout (§3). Capable of supporting any multi-vector queries across various vector combinations and weights using a single index, MSG demonstrates remarkable improvements in search efficiency and accuracy.
- We propose three acceleration techniques tailored for index construction and search in MVS (§4.2 and §5.2). These techniques leverage the features of building multi-space indexes and performing 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-space 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 evaluations on six real-world datasets show that MSG 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 Search (MVS) problem. Then, we outline the motivation behind our research. Please refer to Table 1 for frequently used notations.

2.1 Problem Definition

We define MVS following current Vector Search (VS).

DEFINITION 1. VS. Given an object set S where each object is a single vector, a query vector q , and the distance metric $\delta(\cdot, \cdot)$, VS

Table 1: Frequently used notations.

Notations	Descriptions
S, o	A set of multi-vector objects, an object in S
q	A multi-vector query
o_i, 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
$\delta(\cdot)$	The Euclidean distance between two vectors
w_i	The weight of the i -th distance $\delta(o_i, q_i)$
$g(\cdot)$	The aggregate function of $\delta(\cdot)$
$\ \cdot\ $	The l_2 -norm of a vector

identifies the top- k objects most similar to q , denoted by R . Formally:

$$R = \arg \min_{R \subseteq S} \sum_{o \in R} \delta(o, q) \quad (1)$$

VS aims to find vectors similar to a given query vector, and it is addressed by advanced graph-based indexes such as HNSW [39, 49].

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

$$R = \arg \min_{R \subseteq S} \sum_{o \in R} g(\delta(o_0, q_0), \dots, \delta(o_{t-1}, q_{t-1})) \quad (2)$$

Current studies primarily evaluate cases where $m = 2$ for MVS in their experiments [62, 80], with only a few instances considering m up to 4 [63]. In this paper, we expand the range of m from 1 to 6, covering most real-world applications [18, 35, 57, 77]. 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 $1 \leq i \leq t$, corresponds to the i -th vector of object o . We employ the weighted sum for $g(\cdot)$, as it is widely used in current MVS scenarios [62, 80]:

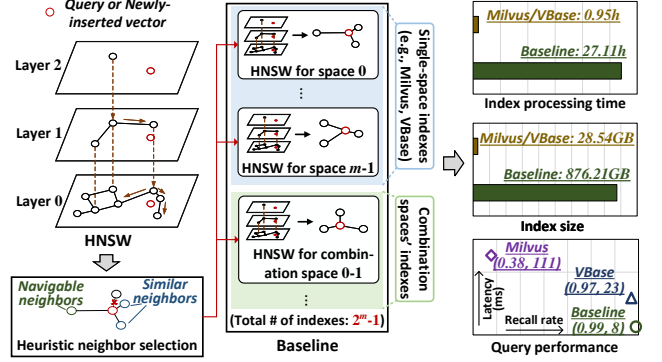
$$g(\delta(o_0, q_0), \dots, \delta(o_{t-1}, q_{t-1})) = \sum_{i=0}^{t-1} w_i \cdot \delta(o_i, q_i) \quad (3)$$

When $m = t = 1$, the MVS problem is simplified to VS, making MVS a more general problem. With the rise of versatile AI applications such as Copilot [10] and Gemini [3], the importance of MVS continues to grow. In the MVS problem, we give the definition of vector combination within a multi-vector object or query.

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. Recall that different vectors within an object originate from different modalities, views, or models [62]. Obviously, a vector combination determines a specific vector space based on the included vectors [28]. When there is no ambiguity, we interchangeably use the terms “vector combination” and its corresponding “vector space” in this paper.

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

**Figure 2: HNSW-based MVS methods.**

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 [6]. \square

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 [6]. \square

Remarks. Following current MVS research [62, 80], we highlight: (1) Unless otherwise specified, the default distance between two vectors is the square Euclidean distance. (2) The absence of weights in an object set is common practice. Typically, different weights, if provided, are integrated into vectors during the embedding phase [18], making it unnecessary to explicitly maintain weights within the object set. Therefore, most applications compute aggregate distances for evaluating similarity between objects without weights [48, 59, 71]. We assume that all objects have been aligned and possess the same number of vectors. (3) For user input queries, due to information misalignment and personalized requirements, different queries may exhibit varying vector combinations (e.g., lacking certain modalities) and weights (e.g., favoring specific modalities).

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 MVS 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 vector search (VS) methods, HNSW is well-studied in academia [29, 43] and widely deployed in industry [7, 11, 74]. Figure 2 (left) illustrates HNSW’s hierarchical structure: the base layer (layer 0) contains all vectors, while the upper layers keep subsets of the lower layer’s vectors. The index is incrementally built offline by inserting vectors individually, each following an exponentially decaying probability distribution for its maximum layer. Each vector acts as a query, finding the top- c closest vertices via a greedy search from the highest to the base layer, selecting both similar and navigable neighbors heuristically. Upon query, the search starts from the top layer and proceeds downward, obtaining the top- k closest vertices at the base layer. Recent studies [66] suggest that HNSW approximates the Monotonic

Relative Neighborhood Graph (MRNG), a type of Monotonic Search Networks (MSNET) [27]. According to Theorem 1, when object vectors and a query vector reside in the same space, the query can efficiently and accurately find its nearest object through greedy search on MSNET. While constructing an MSNET is impractical ($O(n^3)$) [21], HNSW offers a practical construction overhead while maintaining search performance advantages [29, 43, 74].

THEOREM 1. *Suppose that the query q is from the set of objects, meaning that $\delta(o, q) = 0$. Given a set of object vectors and a query vector in the same space, the nearest object to the query can be identified through a greedy search [27] on an MSNET constructed from the set of vectors, with a time complexity approximating $O(\log(n))$.*

PROOF. (Sketch.) We can prove that a monotonic path between any two vertices on an MSNET can be found using a greedy search [27]. The monotonic path has an average length of $O(\log(n))$ [27]. Thus, for the nearest vertex o to the query ($\delta(o, q) = 0$), we can start at any vertex and navigate to o along the monotonic path. Kindly refer to our technical report for the detailed proof [6]. \square

Theorem 1’s assumption has been recently relaxed [26, 54], allowing HNSW to support queries not included in the object set (i.e., $\delta(o, q) \geq 0$) but residing in the same space as object vectors. This is backed by both theoretical and empirical research [26, 54, 55]. Furthermore, Theorem 1 can be extended to retrieve the k nearest objects ($k \geq 1$) with high efficiency and accuracy [27].

HNSW’s index construction and search share a key parameter: candidate set size c (c_1 and c_2 respectively). Each search iteration extracts the vertex closest to the query from the candidate set and visits its neighbors. Unique parameters include the maximal number of neighbors r and the result set size k in index construction and search, respectively. The time complexity of obtaining c_1 candidates for each query 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 dimensionality². Thus, building an HNSW has a time complexity of $O((\theta(n) + r) \cdot c_1 \cdot d \cdot n)$. Executing a query has a time complexity of $O(c_2 \cdot d \cdot \theta(n))$. Storing neighbor IDs and vector data for each base layer vertex results in a space complexity of $O((r + d) \cdot n)$, excluding higher layers due to their negligible size.

2.2.2 Existing HNSW-Based MVS Methods. Current methods for an object set S with m vectors per object generate m vector sets, each containing the corresponding vectors of all objects. This results in m single-space HNSW indexes (Figure 2). The time complexity is $O((\theta(n) + r) \cdot c_1 \cdot D_m \cdot n)$, where d_i is the vector dimension in the i -th vector set, and D_m is the total object vector dimension. The space complexity is $O((r \cdot m + D_m) \cdot n)$. For Merging [81], a multi-vector query acquires multiple candidate sets via VS on these single-space indexes, merging these sets for final results. However, determining the optimal candidate number per VS is challenging. Recent advancements like Milvus [62] and VBase [80] optimize the search process on multiple single-space HNSW indexes. Milvus trials various candidate sizes and reorders candidates via NRA³, with a search time complexity of $O(c_2 \cdot D_t \cdot s \cdot \theta(n))$, where s is the iteration number, and D_t is the total query vector dimension. As iterations increase, a larger c_2 is required. VBase scans multiple indexes in a round-robin manner to avoid repetitive vertex access.

²In the HNSW paper, $\theta(n)$ is roughly $\log(n)$ [49].

³NRA can efficiently derive final results from multiple ordered candidate lists by establishing upper and lower bounds of multi-vector distances [24].

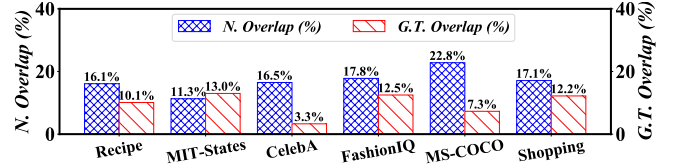


Figure 3: Neighbor overlap ratio (N. overlap (%)) in different vector spaces and ground-truth overlap ratio (G.T. overlap (%)) for queries with identical vectors but different weights.

It computes the aggregate distance between visited vertices and the query to update the result set during each scan, resulting in a search time complexity of $O(c_2 \cdot D_t \cdot t \cdot \theta(n))$.

Complexity analysis and empirical results (Figure 2, right) reveal limitations of multiple single-space HNSW indexes for multi-vector queries. This is because the incompatibility of current index and search strategies with the MVS feature (see §1 and Example 2).

2.2.3 Baseline. A straightforward optimization is to build an HNSW for each vector space, yielding $2^m - 1$ indexes (the middle of Figure 2). Each multi-vector query matches an index based on the query’s vector space. During the search, aggregate distances between the query and visited vertices are calculated using vector data and query weights. The search time complexity is $O(c_2 \cdot D_t \cdot \theta(n))$, lower than current MVS 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 in 2^{m-1} combinations (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 (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 MVS methods on the MIT-States dataset [37]. Baseline outperforms in search efficiency and accuracy but demands more processing time and storage. Figure 3 assesses the neighbor overlap ratio for HNSW in different vector spaces and the ground-truth overlap ratio for queries with identical vectors but varying weights across six real-world datasets. Results reveal a significant disparity in neighbor relationships in different spaces (e.g., N. Overlap Ratio is **11.3%** on MIT-States), indicating the limitation of using HNSW for specific spaces to serve multi-vector queries in other spaces. Despite identical query vector data, varying weights lead to low overlap in query results (e.g., G.T. Overlap Ratio is **3.3%** on CelebA), suggesting the current weight-agnostic search strategy’s unsuitability for multi-vector queries with dynamic weights.

In summary, while current MVS methods utilize single-space indexes for efficient construction and small index size, they exhibit low search efficiency and accuracy. Baseline enhances search accuracy and efficiency, but requires high construction time and index storage. This trade-off underpins the research in this paper.

3 MSG: AN OVERVIEW

We introduce a new Multi-Space Graph (MSG) 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 MSG.

3.1 Index Construction

MSG builds a hierarchical graph index for a set of objects, as outlined in Algorithm 1. It begins with an empty graph to which objects are

Algorithm 1: INDEX CONSTRUCTION

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

```

1 for each object  $o$  in  $S$  do
2    $l_{max} \leftarrow$  the maximum layer of  $o$ ;  $\triangleright$  exponential decaying[49]
3    $L \leftarrow$  the current graph index's top layer;
4   if  $l_{max} > L$  then  $\triangleright$  update the top layer
5      $L \leftarrow l_{max}$ ;
6   for  $l \in \{l_{max}, \dots, 0\}$  do
7     insert  $o$  at layer  $l$ ;  $\triangleright$  §4.1 (Alg. 3), §4.2
8     compress the neighbor IDs of  $o$ ;  $\triangleright$  §4.3
9     store  $o$ 's vectors and neighbor IDs in index  $G$ ;
10 return multi-space index  $G$ 

```

progressively added, forming a sub-graph with the already added objects (lines 1-9). When a new object is inserted, $2^m - 1$ queries are created based on vectors within the object, each corresponding to a vector combination. These queries are utilized to obtain similar objects in their respective vector spaces by a greedy search on the existing sub-graph index; neighbors within each vector space are selected from their similar objects based on the heuristic rule of HNSW (§4.1). In this process, we identify three types of distance computations and develop three corresponding lossless acceleration strategies (§4.2). Upon obtaining 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).

PROPOSITION 1. *In MSG, neighbor relationships between objects exist in all vector spaces where a multi-vector query may reside.*

PROOF. According to Lemma 1 and Algorithms 1 and 3, the validity of this proposition is evident. \square

EXAMPLE 3. As Figure 4 (left) shows, each object is associated with two vectors, resulting in three vector combinations: C0, C1, and C2. MSG performs three traversals for object insertion, each obtaining candidates and neighbors in the corresponding vector space. To simplify, the insertion process for objects on a specific layer is shown, with the observation that all layers operate similarly. Each traversal uses distinct vectors for distance computation. 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 (motivated by previous studies [48, 59, 71], see “Remarks”-(2) in §2.1). The final adjacency list merges neighbor IDs from all vector spaces, with each ID taking up 32 bits. By compressing these IDs, it reduces the space needed for neighbor storage, using fewer bits (e.g., 16 bits, based on the ID residual).

Remarks. (1) Users often employ different weights in multi-vector queries. However, weight information is usually unavailable during index construction [62, 80]. Hence, recent advancements [62, 80] focus on building separate indexes for each modality in an object set without weights. These indexes only organize neighbor relationships within individual vector spaces for each modality. However, in MVS, a multi-vector query is typically located in a new vector space determined by the query's vector combination. Recognizing that an object with m vectors can produce up to $2^m - 1$ vector combinations (Lemma 1), we developed a graph index linking each vertex

Algorithm 2: SEARCH PROCEDURE

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

```

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

```

to various neighbors across all vector spaces, which establishes a direct correspondence from vertices to objects. This approach ensures neighbor relationships are created across all possible vector spaces where a multi-vector query might occur (Proposition 1). Thus, our index can serve a query based on the neighbors matching the query's vector space. If two queries reside in the same vector space but have different weights (situated in different areas of the same space), they can obtain respective results through our new search strategy within the same space (see §3.2). (2) We build a single index over all vector spaces, instead of building separate indexes for each one. The reasons for this approach are threefold. Firstly, separate indexes (i.e., Baseline) result in a large index size due to redundant vector data (Figure 2). Secondly, our evaluation shows that combining neighbor IDs of all vector spaces into a single neighbor list is advantageous for our ID compression (§6.7.2). Lastly, a single index simplifies index maintenance in dynamic update scenarios, whereas separate indexes introduce complex logic.

3.2 Search Procedure

Algorithm 2 outlines the procedure to retrieve the top- k results for a query on MSG. The search initiates at the top layer and iteratively progresses to lower layers until it reaches the bottom layer (lines 1-5). The top layer's entry point is fixed (line 1), and the entry points for lower 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. Throughout the process, an adaptive navigation strategy is implemented, so that the neighbor visit adapts to the vector space and weights of the query (§5.1). For every visited vertex, only neighbors matching the query's vector space are decompressed and explored. Distance computations between the query and visited vertices consider both vector data and query weights. We identify two types of distance computations and implement two specialized acceleration strategies to enhance the search speed without sacrificing accuracy (§5.2).

PROPOSITION 2. *Any multi-vector query in dynamic vector spaces and weights can efficiently and accurately obtain its results on MSG.*

Proposition 1 indicates that MSG builds neighbor relationships between objects across all vector spaces where a query may reside, similar to having an HNSW index for each space. Thus, for multi-vector queries in different vector spaces, a corresponding HNSW is available for search. In addition, the query's weights and vector data specify its location in this space (Figure 4). Theorem 1 suggests that queries situated in different locations of the same space can efficiently and accurately search its nearest object on HNSW.

EXAMPLE 4. Figure 4 (right) illustrates how MSG uniformly processes four queries (i-iv) with different vector combinations or

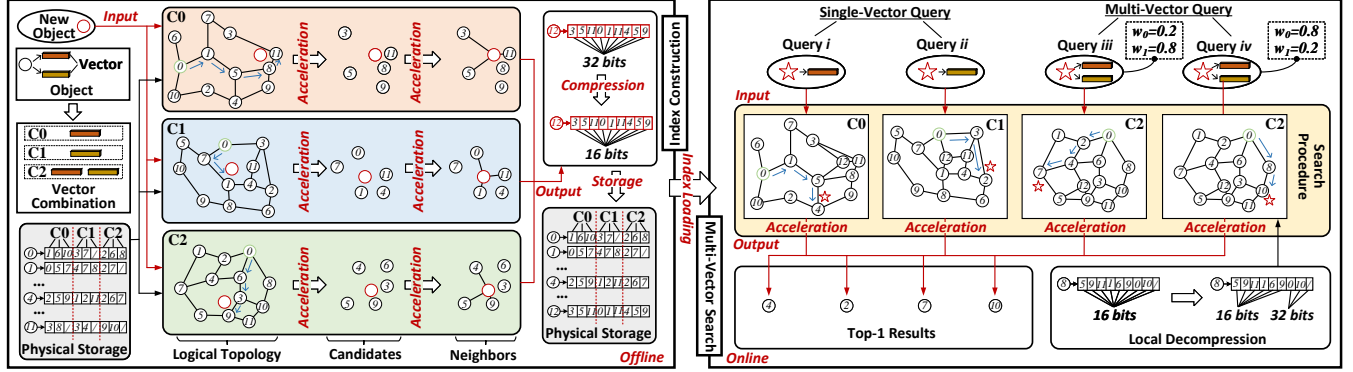


Figure 4: The execution flow of index construction and search procedure in MSG.

weights. The search procedure on a specific layer’s graph structure is highlighted for simplicity. Each query has unique neighbor visits, decompressing only a subset of neighbor IDs relevant to the query’s vector space. For instance, query i only visits neighbors within C0. It starts from vertex 0, measures the distances from its neighbors to the query using their first vectors, and then moves to vertex 1, which is closer to the query. This continues until no closer neighbors are found, at which point the final result is returned. Although both query iii and query iv interact with neighbors corresponding to C2, their unique weights and vector data guide them along distinct search paths. For example, query iii jumps to vertex 2 from vertex 0’s neighbors, while query iv moves to vertex 8.

Remarks. Our search strategy employs an adaptive navigation procedure for queries with varying vector spaces and weights on a weight-agnostic multi-space index. This leverages the fundamental principle of graph index: “a neighbor’s neighbor is likely to be a neighbor as well” [27]. Given a query, even when the search starts from a distant vertex, it can explore the vertex’s neighbors and hop to closer ones. The graph index’s greedy search mechanism guides this exploration to converge on vertices close to the query.

4 INDEX CONSTRUCTION

In this section, we explore the index construction of MSG. 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.

4.1 Basic Process

MSG 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 , MSG updates L to l_{max} . Subsequently, o is inserted from layer l_{max} down to layer 0 as per Algorithm 3. During insertion at a given layer l , MSG explores all vector spaces to identify entry points, candidates, and eventual neighbors (lines 2-10). Note that the distance between vertices varies according to the vector space (see Example 3). When handling a specific vector combination, MSG treats it as a query and determine its entry point at layer l by 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 far from any previously chosen neighbors of o , adhering to the heuristic rule of HNSW [49] (lines 5-10).

Complexity Analysis. To identify the top- c_1 candidates, MSG explores $c_1 \cdot \theta(n)$ vertices⁴. 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 $O((2^m - 1) \cdot r + D_m)$.

4.2 Indexing-Aware Acceleration

Index construction efficiency is significantly influenced by distance computation, which is crucial for identifying entry points, candidates, and final neighbors. Our analysis on the MIT-States dataset shows that the computation operations constitute 77.8% of the total construction time, indicating a major bottleneck. To improve construction efficiency while maintaining index quality, we examine the characteristics of various distance calculations in index construction, categorize them into three groups, and propose three lossless acceleration methods based on these characteristics.

4.2.1 Computation Reuse. For each inserted object o , it may visit vertex x multiple times in different vector spaces. For example, it computes $\delta(o_0, x_0)$ for vector combination C0 (Example 3), and then computes $\delta(o_0, x_0)$ again for C2 to obtain the aggregate distance. To avoid such redundant computations, MSG reuses $\delta(o_0, x_0)$. Initially, it identifies the final neighbors for the spaces from each modality (e.g., C0) and caches the single-vector distances between o and visited vertices. For combination spaces from more than one modality (e.g., C2), MSG checks cached distances and uses the matched values to calculate aggregate distances directly. 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 $\delta(o_i, x_i)$ for $0 \leq i \leq m' - 1$. Computation reuse fetches some $\delta(o_i, x_i)$ from cache, skipping their computation. Please refer to our technical report for the detailed proof [6]. \square

Optimization. Most aggregate distance calculations can be simplified by reusing cached distances, indicating a significant overlap (always $> 50\%$) between visited vertices for spaces involving the

Algorithm 3: INSERTION OF NEW OBJECT AT LAYER l

Input: graph index at layer l , newly-inserted object o , # candidates per vector space c_1 , maximal # neighbors per vector space 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  $\triangleright f(\cdot, \cdot)$  is the distance
10       $R' \leftarrow R' \cup x$ ; ▷ heuristic neighbor selection
11    $R \leftarrow R \cup R'$ ; ▷ merge all neighbors
12 return  $R$ 
```

same modalities (e.g., C0-C1 and C2). We observe that the state-of-the-art graph index requires approximately 50% similar neighbors [66]. Consequently, we can directly generate candidates for C2 from the visited vertices when obtaining candidates for C0 and C1. This approach further reuses distance computations, enhancing index construction efficiency. The insertion time complexity for an object is now $O((\theta(n) \cdot m + r \cdot 2^{m-1}) \cdot c_1 \cdot D_m)$, which is lower than before.

4.2.2 Approximate Computation. In index construction, many distances are typically compared to 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 maximum distance from o for the elements in C . Additionally, all distance computations for final neighbor acquisition aim for comparisons (line 9 in Algorithm 3). On MIT-States, comparisons constitute **90.4%** of total computations, with **83.4%** successfully performed using approximate distances. Thus, exact distances are often unnecessary. To optimize this, we can incrementally scan a vector combination's dimensions, stopping when the partial distance exceeds T or all dimensions are scanned [29, 54]. This reduces the time complexity of distance computation. However, different dimensions contribute differently to the final distance. Therefore, incremental scanning might involve many low-contribution values.

EXAMPLE 5. In Figure 5, the candidate set's distance threshold T is 0.25. For a visited vertex x , the accurate square distance $(dist)^2$ is computed and compared with T . Since $(dist)^2 > T$, x is discarded. This requires the scanning of 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, directly checking the fourth dimension, i.e., $(dist')^2 = (0.9 - 0.2)^2 = 0.49$, also leads to a correct comparison, as it contributes the most. Thus, scanning the key dimensions first is vital for efficiency.

MSG transforms vectors based on value importance, assigning higher importance to lower dimensions. For an object set S of size n , each object o has m vectors, with the i -th vectors forming the i -th vector set S_i . MSG treats S_i as an $n \times d_i$ matrix \tilde{S}_i , where d_i is dimensionality. The vector transformation follows these steps.

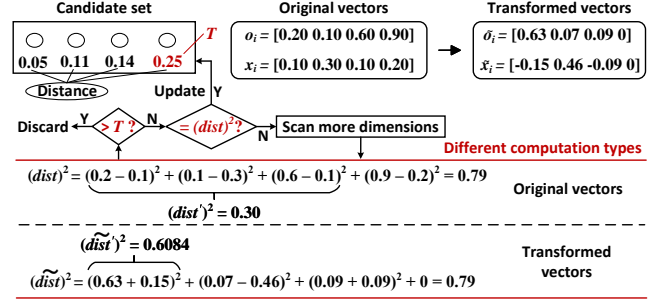


Figure 5: An example of approximate computation.

(i) Calculate mean vector μ_i :

$$\mu_i = \frac{1}{n} \sum_{o_i \in S_i} o_i \quad (4)$$

(ii) Centralize vectors in S_i :

$$\hat{S}_i = S_i - \mathbf{1} \mu_i^\top, \quad (5)$$

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

(iii) Compute covariance matrix Σ_i :

$$\Sigma_i = \frac{1}{n} (\hat{S}_i^\top \hat{S}_i) \quad (6)$$

(iv) Determine eigenvalues and eigenvectors of Σ_i : $(\lambda_0, \dots, \lambda_{d_i-1})$ and (z_0, \dots, 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 M_i :

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

where z_0 corresponds to the largest eigenvalue λ_0 , z_1 corresponds to the second largest eigenvalue λ_1 , and so on.

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

$$\tilde{o}_i = M_i^\top o_i \quad (8)$$

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

LEMMA 4. The projection matrix M_i is orthonormal.

PROOF. (Sketch.) 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 [6]. \square

LEMMA 5. Transforming vectors o_i and x_i in S_i by the orthonormal matrix M_i does not alter the distance between them.

PROOF. (Sketch.) An orthogonal matrix holds inverse equal to transpose. Transformation by M_i keeps vector length and distance. Kindly refer to our technical report for the detailed proof [6]. \square

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 Σ_i , and σ_j^2 as the variance of the j -th dimension of \tilde{S}_i , it holds: $\sigma_j^2 = \lambda_j$.

PROOF. (Sketch.) The transformed data \tilde{S}_i are zero-mean. We have $\text{Var}(\tilde{S}_i) = E(\tilde{S}_i^2)$ by the identity $\text{Var}(\tilde{S}_i) = E(\tilde{S}_i^2) - (E(\tilde{S}_i))^2$, where $\text{Var}(\tilde{S}_i)$ is the variance of \tilde{S}_i . $\tilde{S}_i[j] = M_i^\top[j] \hat{S}_i$ is the j -th dimension of \tilde{S}_i . We have $\sigma_j^2 = M_i^\top[j] \Sigma_i (M_i^\top[j])^\top \cdot (M_i^\top[j])^\top$

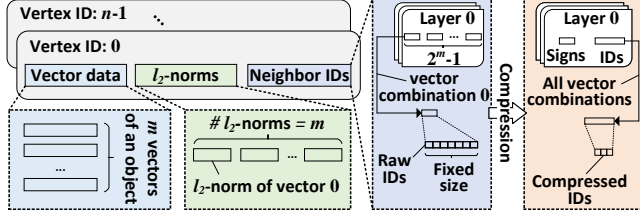


Figure 6: Overview of index layout.

is the j -th eigenvector of Σ_i with eigenvalue λ_j . Due to space limitations, we put the detailed proof in our technical report [6]. \square

THEOREM 2. *After transforming \mathbf{o}_i and \mathbf{x}_i using \mathbf{M}_i , the contribution to $\delta(\mathbf{o}_i, \mathbf{x}_i)$ is non-increasing from the first to the last dimensions.*

PROOF. (Sketch.) $\tilde{\mathbf{o}}_i$ and $\tilde{\mathbf{x}}_i$ are projections of \mathbf{o}_i and \mathbf{x}_i by \mathbf{M}_i . Distance contribution of dimension j is:

$$(\Delta_j)^2 = (\tilde{\mathbf{o}}_i[j] - \tilde{\mathbf{x}}_i[j])^2. \quad (9)$$

Δ_j is proportional to the square root of the eigenvalue. Thus, $(\Delta_j)^2 \geq (\Delta_{j+1})^2$. Refer to our full report for the detailed proof [6]. \square

MSG transforms each vector in an inserted object using the corresponding orthogonal matrix. When computing the distance for comparison, it incrementally scans transformed vectors and prioritizes high-importance values. MSG also allows accurate distance calculation on transformed vectors (see Lemma 5). This optimization improves construction efficiency by optimizing the D_m term. Our evaluation on the MIT-States dataset shows a **22.8%** reduction in dimensions 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 $\widehat{dist'}$ on transformed vectors ranges from 0 to \widehat{dist} . We can always ensure a correct comparison by using $\widehat{dist'}$, whether $\widehat{dist} > T$ or $\widehat{dist} \leq T$. Kindly refer to our technical report for the detailed proof [6]. \square

EXAMPLE 6. Figure 5 showcases the transformation of original vectors using \mathbf{M}_i , yielding vectors with primary significance in the first dimension, succeeded by the second, and so forth. In this example, one dimension is scanned on the transformed vectors to compute the partial distance $(\widehat{dist'})^2 = 0.6084$. Since this value exceeds T , \mathbf{x} is discarded without further scanning. Note that the accurate distance remains unchanged by the transformation.

4.2.3 Accurate Computation. When a candidate set is unfilled, it is necessary to compute the accurate distance between the inserted object \mathbf{o} and a visited vertex \mathbf{x} . The following formula connects square Euclidean distance and inner product:

$$\|\mathbf{o}_i - \mathbf{x}_i\|^2 = \|\mathbf{o}_i\|^2 + \|\mathbf{x}_i\|^2 - 2 \cdot \mathbf{o}_i \cdot \mathbf{x}_i. \quad (10)$$

Motivated by this, MSG reformulates distance computation as fast inner product computation. It calculates and stores the square l_2 -norms ($\|\mathbf{o}_i\|^2$) of vectors in \mathbf{o} . When computing the accurate distance between \mathbf{o}_i and \mathbf{x}_i , it first calculates the inner product $\mathbf{o}_i \cdot \mathbf{x}_i$. The square Euclidean distance is then calculated by Equation 10 using the precomputed $\|\mathbf{o}_i\|^2$ and $\|\mathbf{x}_i\|^2$ (the square l_2 -norms of all vectors in an object are stored upon insertion). Therefore, MSG obtains the exact distance solely by computing the inner product,

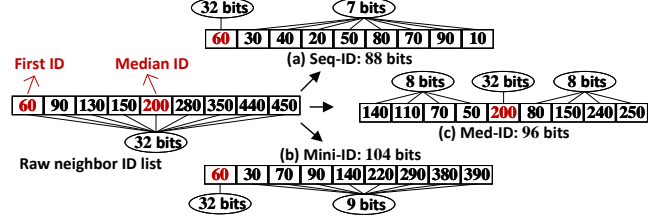


Figure 7: Illustration of neighbor ID list compression.

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. \square

4.3 Index Compression

MSG consolidates all data into a single index for unified management. As illustrated in Figure 6, it sequentially organizes m vectors, m l_2 -norms, and the neighbor IDs for $2^m - 1$ vector spaces per vertex. Equal storage is allocated for each object's vector data and l_2 -norms, facilitating sequential access via offsets. 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 spaces, with each list being of consistent size to allow direct access through offsets.

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

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.

EXAMPLE 7. Figure 7(a) shows the compression process using Seq-ID. Prior to compression, nine IDs require 288 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 are 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 7(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. 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.

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

```

EXAMPLE 9. As illustrated in Figure 7(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.

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 raw and compressed IDs, we conclude that the compression process is indeed lossless. \square

Remarks. MSG optimizes the $(2^m - 1) \cdot r$ term in the complexity formula through index compression, effectively reducing storage space. For example, on MIT-States, neighbor lists' space savings reach **52.1%**, **51.7%**, and **51.9%** with Seq-ID, Mini-ID, and Med-ID, respectively. While Seq-ID follows popular delta encoding such as PforDelta [42, 72, 86], it incurs notable decompression overhead during search. Mini-ID and Med-ID avoid unnecessary decompression and employ rapid bit operations, thus maintaining search efficiency.

5 SEARCH PROCEDURE

In this section, we explore the search procedure on MSG. We provide an overview of the fundamental process (§5.1) and discuss optimizations related to query-aware acceleration (§5.2).

5.1 Basic Process

Recall that MSG 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 farthest 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 4, two types of distance computations are performed. The first type involves comparison with a threshold (it is the distance between q and the farthest 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, MSG prioritizes 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 of vectors. Therefore, all query-aware acceleration methods are lossless. \square

Complexity Analysis. To obtain the top- k nearest neighbors, MSG visits vertices at a scale of $c_2 \cdot \theta(n)$. 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.

6 EXPERIMENTS

To conduct a comprehensive evaluation of MSG, we carry out the following experiments: (i) Query Performance (§6.2), (ii) Construction Efficiency (§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, additional evaluations, and the technical report are publicly accessible at: <https://anonymous.4open.science/r/MSG>.

6.1 Experimental Setting

6.1.1 Datasets. We use six real-world datasets, each with objects having at least two vectors from different modalities or encoders. Table 2 exhibits variations in the number of vectors (m), dimensionality (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 CelebA is merely 200K), we expand the datasets using generative models [23, 75], which creates additional samples from the learned distribution of real data.

Table 2: Statistics of experimental datasets.

Datasets	n	m	D_m	$\#q$	t
Recipe	1.3M	2	2,048	10^4	1~2
MIT-States	2.1M	6	3,456	10^3	1~6
CelebA	1M	4	2,304	10^3	1~4
FashionIQ	1M	2	1,024	10^3	1~2
MS-COCO	1M	3	1,536	10^3	1~3
Shopping	1M	2	1,024	10^3	1~2
MIT-States+	16M	2	1,152	10^3	1~2

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. Similar to related work [62, 80], 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.

6.1.3 Compared Methods. We evaluate six methods based on HNSW. (i) VBase. Developed by Microsoft, it leverages index scanning optimization on multiple single-space indexes [80]. (ii) Milvus. Released by Zilliz, it applies candidate merging optimization with multiple single-space indexes [62]. (iii) Merging. A previous MVS method for hybrid queries [63, 81], it also relies on multiple single-space indexes. (iv) Baseline. Our naive optimization builds an index for each vector space (see §2.2.3). (v) MSG. It deploys all proposed techniques, including Med-ID algorithms for index compression. (vi) MSG*. It is a variant of MSG 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 MVS method, the *Recall* is given by:

$$\text{Recall} = \frac{|R \cap R'|}{k}, \quad (11)$$

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

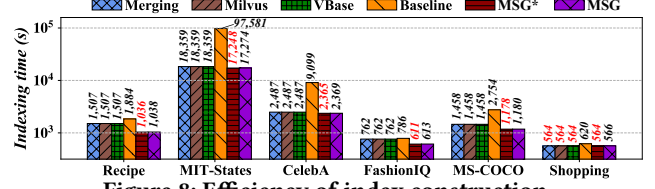
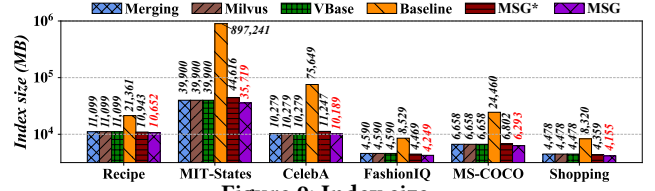
6.1.5 Environment Configuration. We run the experiments on a machine with an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 128GB of memory, on CentOS 7.9. All methods are coded in C++ and compiled with g++ 4.8 and -O3 optimization. We use OpenMP for parallel index construction, with 48 threads for all methods. For query execution, we use one thread for all methods [27, 66].

6.2 Multi-Vector Query Performance

Table 3 presents the multi-vector query performance of various methods. Our observations are as follows: (i) MSG* achieves an optimal balance between accuracy and efficiency. For instance, when compared to the leading VBase, MSG* reduces latency by up to **96.5%**, while maintaining a higher recall rate. (ii) MSG exhibits higher latency than MSG* due to MSG 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-space 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 8 shows index construction times for different methods. We adjust the parameters of index construction to achieve optimal

**Figure 8: Efficiency of index construction.****Figure 9: Index size.**

search performance. Detailed parameter values are documented in the technical report [6]. Our observations are as follows: (i) MSG* is fastest due to its smaller maximum number of neighbors (r) and indexing-aware acceleration component. MSG is slightly slower due to an extra compression step. (ii) Single-space indexes (Merging, Milvus, VBase) require a larger candidate neighbor set (c_1) and a greater r than multi-space indexes (MSG* and MSG) to achieve optimal search performance. (iii) Baseline has the longest construction time, as it needs to build indexes for each vector spaces.

6.4 Index Size

Figure 9 shows index sizes of different methods, following the parameter setup in §6.3. Key findings are as follows: (i) MSG exhibits the smallest index size due to two factors: a smaller number of neighbors r and effective index compression. Specifically, index size comprises neighbor size (neighbor IDs) and vector size (vector data). Given that VBase and MSG share the same vector size, the difference in index size is determined solely by neighbor size. The space complexities of VBase and MSG for neighbor size are $O(mrn)$ and $O((2^m - 1)rn)$, respectively. Our evaluation [6] demonstrates that VBase requires significantly larger r than MSG for the optimal search performance. (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 space. Consequently, each vector within an object is stored 2^{m-1} times (see Lemma 2), resulting in high space cost.

6.5 Query Workloads

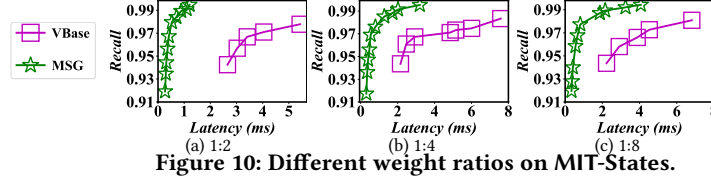
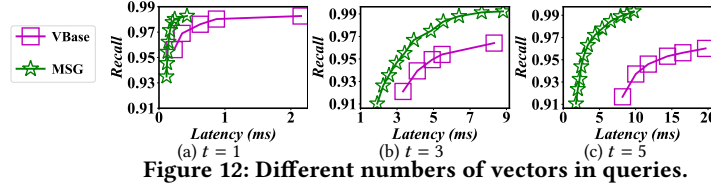
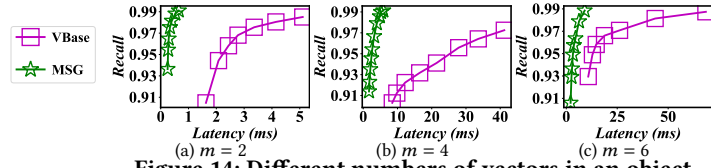
In this section, we compare MSG and VBase across different query workloads, maintaining identical HNSW settings for both. We exclude other existing methods (e.g., Merging and Milvus) due to their pronounced limitations in efficiency and accuracy.

6.5.1 Weight Ratio. We adjust the weight ratio for each dual-vector query on MIT-States. Figure 10 demonstrates that MSG consistently outperforms VBase. Notably, MSG exhibits superior performance when the weight bias is minimal. Prior research has indicated that minor weight bias is prevalent in real-world scenarios [63].

6.5.2 Dynamic Weight. We evaluate the search performance of VBase and MSG using queries with dynamic weights at the per-query level. As shown in Figure 11, MSG consistently outperforms VBase by a substantial margin. For instance, at the same recall rate of 0.99 on Recipe, MSG is $31 \times$ faster than VBase.

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
MSG*	0.99	2.1	0.99	5.5	0.99	5.6	0.99	4.1	0.93	10.8	0.95	8.3
MSG	0.99	3.0	0.99	6.7	0.99	6.5	0.99	4.7	0.93	11.6	0.95	11.1

**Figure 10: Different weight ratios on MIT-States.****Figure 12: Different numbers of vectors in queries.****Figure 14: Different numbers of vectors in an object.**

6.5.3 Number of Query Vectors (t). Figure 12 shows the performance results across varying values of t . MSG is more superior for all values of t , and the performance gap widens as t increases.

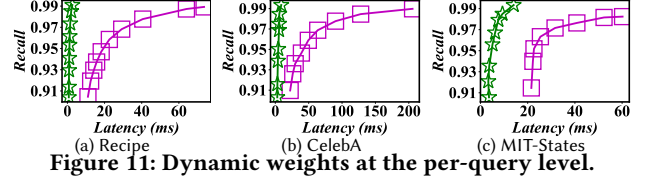
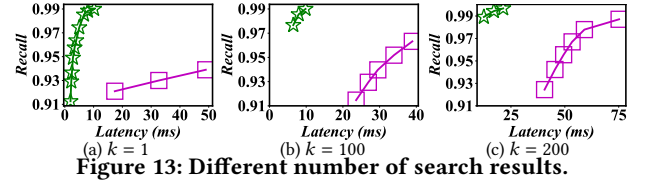
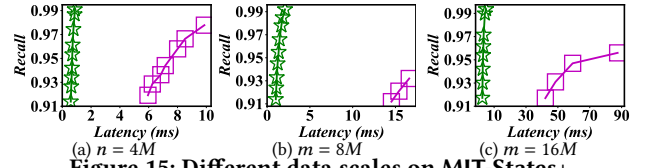
6.5.4 Number of Results (k). Figure 13 illustrates the search performance of MSG and VBase on MIT-States for varying values of k . MSG proves more robust than VBase across all values of k .

6.6 Scalability

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

6.6.1 Number of Vectors in an Object (m). Figure 14 shows the impact of parameter m on search performance. The results demonstrate that MSG consistently outperforms VBase, particularly at the high recall rates. 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 MSG with r corresponding to their respective optimal search performance. For further reference, our technical report [6] includes detailed results for identical r values. As m increases, MSG's neighbor size and construction time grow faster than VBase's at the same r . However, under optimal search performance, VBase requires a larger index size and construction time due to a significantly larger r than MSG.

6.6.2 Data Scale (n). Figure 15 illustrates the search performance of MSG and VBase for different values of n . While the latency of VBase increases linearly with n , MSG exhibits only a minor latency increase even with large n values. Additionally, MSG demonstrates a significant accuracy superiority over VBase.

**Figure 11: Dynamic weights at the per-query level.****Figure 13: Different number of search results.****Figure 15: Different data scales on MIT-States+.****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	MSG	VBase	MSG	VBase	MSG
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

6.7 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 16(a) shows the total dimensions needed for distance calculation during index construction. We gradually integrate acceleration methods into the original process. Specifically, *w/o IA* denotes the process without acceleration, *w. IA1* adds computation reuse optimization, *w. IA2* further incorporates approximate computation optimization, and *w. IA3* includes accurate computation optimization. These optimizations significantly reduce the evaluated dimensions. Figure 16(b) shows 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*), MSG 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

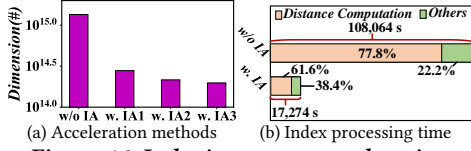


Figure 16: Indexing-aware acceleration.

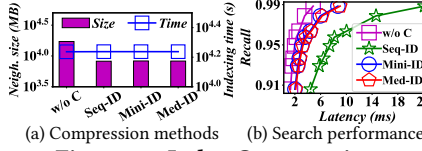


Figure 17: Index Compression.

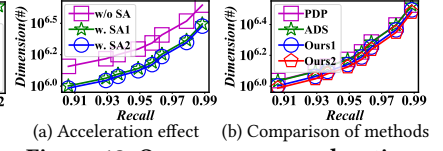


Figure 18: Query-aware acceleration.

Table 5: Complexities of different methods (refer to Table 1 and §2.2.2 for symbols).

Methods	Construction time	Space	Search
Milvus	$O((\theta(n) + r)c_1 D_m n)$	$O((mr + D_m)n)$	$O(sc_2 D_t \theta(n))$
VBase	$O((\theta(n) + r)c_1 D_m n)$	$O((mr + D_m)n)$	$O(tc_2 D_t \theta(n))$
Baseline	$O((\theta(n) + r)c_1 D_m 2^{m-1} n)$	$O(((2^m - 1)r + 2^{m-1} D_m)n)$	$O(c_2 D_t \theta(n))$
MSG	$O((\theta(n)m + 2^{m-1}r)c_1 D_m n)$	$O(((2^m - 1)r + D_m)n)$	$O(c_2 D_t \theta(n))$

performance. Figure 17(a) shows that our compression techniques reduce neighbor size by nearly 50%, with merely 0.11%~0.21% of the total index processing time. In Figure 17(b), we observe that Seq-ID exhibits higher query latency than the uncompressed method, due to the decomposition 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 18 depicts the dimensions used for distance calculation in the search process. Figure 18(a) shows our approximate computation optimization (w. SA1) reduces the dimensions by 32.9% compared to w/o SA. Despite the small fraction of accurate computations, our accurate computation optimization still brings obvious improvement (w. SA1 vs. w. SA2). Figure 18(b) compares our methods with two SOTA techniques: PDP [54] and ADS [29], with ours surpassing both. Employing weight-based optimization (Ours2) further reduces dimensions on initial approximate computation optimization (Ours1).

7 SUMMARY

We summarize our experiments and delve into possible optimizations and limitations of MSG. Table 5 catalogs the complexities of different methods, and Figure 19 compares their overall capabilities. **Overall Performance.** Figure 19 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. VS and MVS 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 data scale increase, with smaller times and sizes, and better performance scoring higher. Our proposed methods, MSG* and MSG, exhibit the best overall capabilities, offering efficient index processing, compact layout, and high search efficiency and accuracy for both MVS and VS, with excellent scalability for handling more vectors and larger data scales.

Potential Optimizations. VS methods' advancements offer potential optimizations for MSG. (i) Using GPU's parallel computation could accelerate MSG's index construction and multi-vector search. (ii) Deploying MSG 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 MSG to predict next steps or termination condition during multi-vector searches.

Limitations. Table 5 reveals that MSG'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 $m = 2$ [17, 22, 38]. For instance, a multi-modal object has text and image modalities [41]; in video surveillance, three vectors represent a person's front face, side face, and posture [62]. Current research only evaluates the case of $m = 2$ [62, 80]. For larger m , current methods need higher offline overhead for optimal search performance (Table 10). Thus, as m increases, current methods and MSG 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 [20, 47, 50], Hashing-based [31, 34, 44], Quantization-based [15, 33, 40], and Graph-based methods [27, 46, 49]. Graph-based methods achieve the state-of-the-art balance of efficiency and accuracy [45]. Various optimizations focus on different aspects, like external storage [39, 65], GPU acceleration [61, 83], and learning to route [43, 78]. However, current algorithms face performance bottlenecks for MVS due to the drawback of single-space index.

Vector Databases. Vector databases, built on vector search algorithms, offer versatile features for industrial applications [32]. Essential for unstructured data management with the rise of large language models (LLMs) [1, 82], many vector databases like Milvus [11], Pinecone [8], and Weaviate [8] effectively serve VS tasks. However, for MVS on unstructured data across multiple modalities [76], existing databases lack adequate support [53, 70].

Multi-Vector Search. Recent surveys [53, 58, 60, 70] and industrial scenarios [2, 4, 9, 13, 32] emphasize the requirement for efficient and accurate MVS solutions. Existing methods handle MVS by refining search strategies on multiple single-space indexes [62, 80], enhancing performance but still facing efficiency, accuracy, and scalability challenges due to the limitations of single-space index.

9 CONCLUSION

We study the MVS problem and introduce an innovative solution, MSG. Our method integrates all vectors per object in a multi-space index, efficiently supporting both VS and MVS. We develop computation acceleration techniques and index compression algorithms, encompassing them into MSG to improve index processing and search procedure. Furthermore, we bolster our method with rigorous theoretical analysis. Our experiments demonstrate the superiority of our method in construction efficiency, space cost, search performance, and scalability. In future research, we plan to explore potential optimizations discussed in §7 to further enhance MSG.

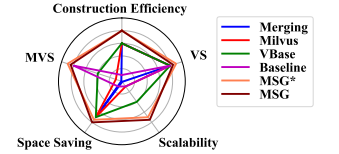


Figure 19: Ability Comparison.

REFERENCES

- [1] 2023. ChatGPT plugins. <https://openai.com/blog/chatgpt-plugins>. [Online; accessed 10-October-2023].
- [2] 2023. MultiVector Retriever. https://python.langchain.com/docs/modules/data_connection/retrievers/multi_vector. [Online; accessed 10-October-2023].
- [3] 2023. Welcome to the Gemini era. <https://deepmind.google/technologies/gemini/>. [Online; accessed 07-November-2023].
- [4] 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>. [Online; accessed 15-February-2024].
- [5] 2024. Hugging Face Embeddings API. <https://huggingface.co/blog/getting-started-with-embeddings>. [Online; accessed 20-January-2024].
- [6] 2024. MSG: Empowering High-Dimensional Multi-Vector Search (Technical Report). <https://anonymous.4open.science/r/MSG>. [Online; accessed 07-July-2024].
- [7] 2024. pgvector. <https://github.com/pgvector/pgvector>. [Online; accessed 11-March-2024].
- [8] 2024. pinecone. <https://www.pinecone.io/>. [Online; accessed 11-March-2024].
- [9] 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/. [Online; accessed 15-March-2024].
- [10] 2024. Reinventing search with a new AI-powered Bing and Edge, your copilot for the web. <https://news.microsoft.com/the-new-Bing/>. [Online; accessed 05-January-2024].
- [11] 2024. Vector database built for scalable similarity search. <https://milvus.io/>. [Online; accessed 11-March-2024].
- [12] 2024. Vector search in Azure AI Search. <https://learn.microsoft.com/en-us/azure/search/vector-search-overview>. [Online; accessed 07-February-2024].
- [13] 2024. What is tensor search? https://medium.com/@jesse_894/introducing-marqo-build-cloud-native-tensor-search-applications-in-minutes-9cb9a05a1736. [Online; accessed 20-January-2024].
- [14] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore L. Willke. 2023. Similarity search in the blink of an eye with compressed indices. *Proceedings of the VLDB Endowment (PVLDB)* 16, 11 (2023), 3433–3446.
- [15] 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.
- [16] 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.
- [17] Alberto Baldri, 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.
- [18] 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.
- [19] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jin 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.
- [20] 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.
- [21] D.W. Dearholt, N. Gonzales, and G. Kurup. 1988. Monotonic Search Networks For Computer Vision Databases. In *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, Vol. 2. 548–553.
- [22] Ginger Delmas, Rafael Sampaio de Rezende, Gabriela Csúrka, and Diane Larlus. 2022. ARTEMIS: Attention-based Retrieval with Text-Explicit Matching and Implicit Similarity. In *The Tenth International Conference on Learning Representations (ICLR)*.
- [23] 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).
- [24] 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.
- [25] 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.
- [26] Cong Fu, Changxu Wang, and Deng Cai. 2021. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 44, 8 (2021), 4139–4150.
- [27] 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.
- [28] Kata Gábor, Haifa Zargayouna, Isabelle Tellier, Davide Buscaldi, and Thierry Charnois. 2017. Exploring Vector Spaces for Semantic Relations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1814–1823.
- [29] 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 (PACMOD)* 1, 2 (2023), 137:1–137:27.
- [30] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data (PACMOD)* 2, 3 (2024), 167:1–137:27.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] Sargan 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.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience (SPE)* 45, 1 (2015), 1–29.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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.
- [47] 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.
- [48] 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.
- [49] Yuri A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 42, 4 (2020), 824–836.
 - [50] 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.
 - [51] 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.
 - [52] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Tutorial Track.
 - [53] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of Vector Database Management Systems. *arXiv:2310.14021* (2023).
 - [54] 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 (PACMOD)* 1, 1 (2023), 54:1–54:27.
 - [55] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. 2020. Graph-based Nearest Neighbor Search: From Practice to Theory. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, Vol. 119. 7803–7813.
 - [56] 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.
 - [57] 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.
 - [58] Toni Taipalus. 2023. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *arXiv:2309.11322* (2023).
 - [59] 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.
 - [60] 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.
 - [61] 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.
 - [62] 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.
 - [63] 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)*.
 - [64] 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.
 - [65] 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 (PACMOD)* 2, 1 (2024), 14:1–14:27.
 - [66] 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.
 - [67] 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.
 - [68] 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.
 - [69] 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.
 - [70] Renzhi Wu, Jingfan Meng, Jie Jeff Xu, Huayi Wang, and Kexin Rong. 2023. Re-thinking Similarity Search: Embracing Smarter Mechanisms over Smarter Data. *arXiv:2308.00909* (2023).
 - [71] 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.
 - [72] 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.
 - [73] 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.
 - [74] 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.
 - [75] 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.
 - [76] 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).
 - [77] 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.
 - [78] Qiang Yue, Xiaoliang Xu, Yuxiang Wang, Yikun Tao, and Xuliyan Luo. 2023. Routing-Guided Learned Product Quantization for Graph-Based Approximate Nearest Neighbor Search. *arXiv:2311.18724* (2023).
 - [79] 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.
 - [80] 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.
 - [81] 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.
 - [82] 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.
 - [83] 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.
 - [84] 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.
 - [85] 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.
 - [86] 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.
 - [87] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data (PACMOD)* 2, 1 (2024), 69:1–69:26.

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. \square

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. \square

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 $g(\delta(o_0, x_0), \dots, \delta(o_{m'-1}, x_{m'-1}))$ 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. \square

D PROOF OF LEMMA 4

LEMMA 4. *The projection matrix 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 M_i , composed of these eigenvectors, is confirmed to be an orthonormal matrix. \square

E PROOF OF LEMMA 5

LEMMA 5. *Transforming vectors \mathbf{o}_i and \mathbf{x}_i in S_i by the orthonormal matrix 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:

$$M_i^\top = M_i^{-1} \quad (12)$$

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

F PROOF OF LEMMA 6

LEMMA 6. *With λ_j as the j -th largest eigenvalue of Σ_i , and σ_j^2 as the variance of the j -th dimension of \tilde{S}_i , it holds: $\sigma_j^2 = \lambda_j$.*

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

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

Given that $(M_i^\top[j])^\top$ is the j -th eigenvector of Σ_i and λ_j is the corresponding eigenvalue, it follows that

$$\sigma_j^2 = \lambda_j M_i^\top[j] (M_i^\top[j])^\top = \lambda_j \|M_i^\top[j]\|^2 = \lambda_j \quad (14)$$

Thus, the variance of the j -th dimension of the transformed data \tilde{S}_i equals the j -th largest eigenvalue of the covariance matrix Σ_i . \square

G PROOF OF THEOREM 1

THEOREM 1. *After transforming \mathbf{o}_i and \mathbf{x}_i using M_i , the contribution to the distance between \mathbf{o}_i and \mathbf{x}_i is non-increasing from the first dimension to the last dimension.*

PROOF. Let us designate $\tilde{\mathbf{o}}_i$ and $\tilde{\mathbf{x}}_i$ as projections of vectors \mathbf{o}_i and \mathbf{x}_i onto the new space defined by 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:

$$(\Delta_j)^2 = (\tilde{o}_i[j] - \tilde{x}_i[j])^2 \quad (15)$$

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 \mathbf{o}_i and \mathbf{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

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 M_i . \square

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, MSG 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. \square

APPENDIX II: SUPPLEMENTARY INSTRUCTION

I HNSW ALGORITHM

Among graph-based VS methods, Hierarchical Navigable Small World graph (HNSW) is well-studied in academia [29, 43] and widely deployed in industrial databases [7, 11, 74]. Figure 2 (left) illustrates HNSW's hierarchical structure, where the base layer (layer 0) contains all vectors and the upper layer keeps a subset of the lower layer's vectors. In the offline stage, the 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 (candidates) by greedy search in each layer. This iterates from the vector's maximal layer to the base layer. In each layer, HNSW selects both similar and navigable neighbors based on a heuristic strategy (the navigable ones act as "expressways" for efficiency [49]). Given a query, 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 obtained. Based on recent research [66], HNSW serves as a practical approximation for the Monotonic Relative Neighborhood Graph (MRNG), a type of Monotonic Search Networks (MSNET) [27]. According to Theorem 1, when object vectors and a query vector reside in the same space, the query can efficiently and accurately find its nearest object by greedy search on MSNET. However, constructing an MSNET is impractical ($O(n^3)$) [21]; instead, HNSW offers a practical construction overhead while maintaining the advantage in search performance [29, 43, 74].

In HNSW, 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 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 vertices⁴. 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 dimensionality. Hence, the time complexity of building an HNSW is $O((\theta(n) + r) \cdot c_1 \cdot d \cdot n)$. The time complexity of executing a query on HNSW is $O(c_2 \cdot d \cdot \theta(n))$. We store r neighbor IDs and d values of vector data for each base layer vertex, omitting higher layers from complexity analysis due to their negligible size. Thus, the space complexity of HNSW is $O((r + d) \cdot n)$.

J EXISTING MVS METHODS

Recent studies address MVS 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 [81]. However, determining the optimal number of candidates per VSS is challenging [80]. Too many or too few candidates can lead to either an increased computational burden or inaccurate results, respectively [62]. Milvus [62] addresses this issue by trialing various candidate sizes and re-ordering the candidates via the NRA algorithm⁵ [24]. Nevertheless, each trial requires repetitive traversals of indexes, resulting in significant vector access and computation. Alternatively, VBase [80] 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.

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 [62] and VBase [80], 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 $O(\sum_{i=0}^{t-1} c_2 \cdot d_i \cdot \theta(n))$ 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 [24]. 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

⁴In the HNSW paper, $\theta(n)$ is roughly $\log(n)$ [49].

⁵NRA can efficiently derive final results from multiple ordered candidate lists by establishing upper and lower bounds of multi-vector distances [24].

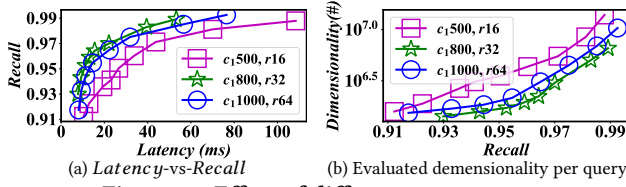


Figure 20: Effect of different parameters.

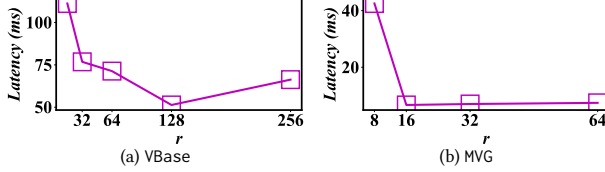


Figure 21: Search performance with varying numbers of neighbors on MIT-States.

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

update the result set. Thus, the search time complexity of VBase is $O(c_2 \cdot D_t \cdot t \cdot \theta(n))$.

APPENDIX III: ADDITIONAL EVALUATIONS

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

K 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 20. 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 21 illustrates the search performance of VBase and MVG with varying r values on MIT-States, with all query latency results

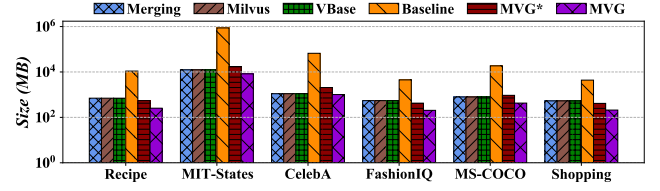


Figure 22: Index size.

recorded at a recall rate of 0.99. We noted that the query latency 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*).

L 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.

M INDEX SIZE WITHOUT RAW VECTOR STORAGE

Figure 22 illustrates the index sizes of various methods, excluding the storage of raw vectors in its calculations.

N PERFORMANCE CURVES OF MVSS

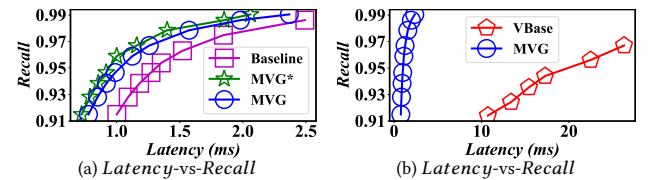


Figure 23: Latency-vs-Recall curve on Recipe.

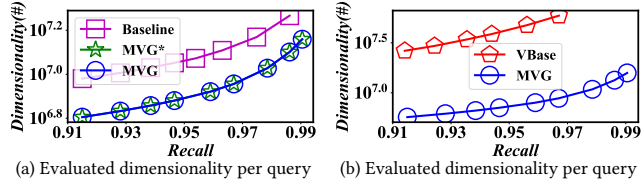
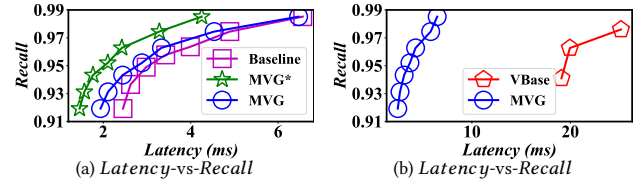
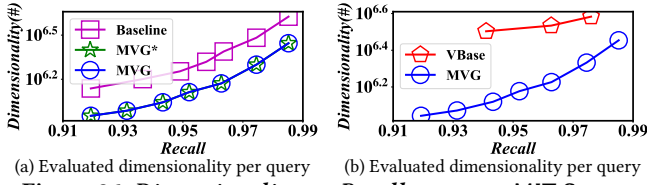
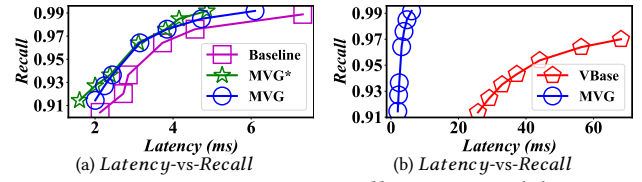
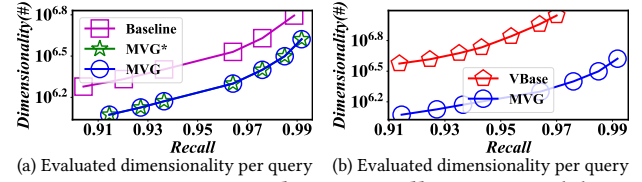
To clearly illustrate the trade-off between search efficiency and accuracy, we present the performance curve in Figure 23–34. For a detailed overview of the compared methods, please refer to §6.1.3.

Table 8: Parameter values used in our experiments.

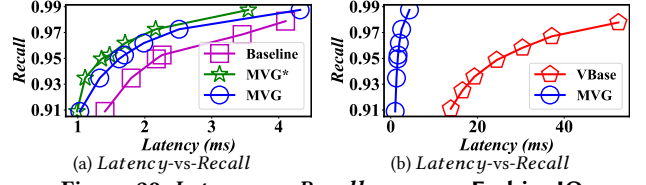
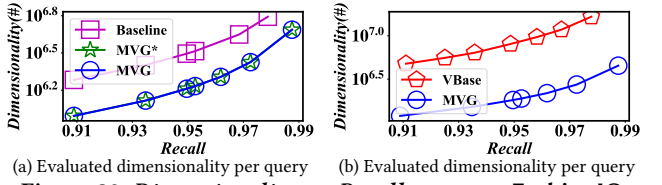
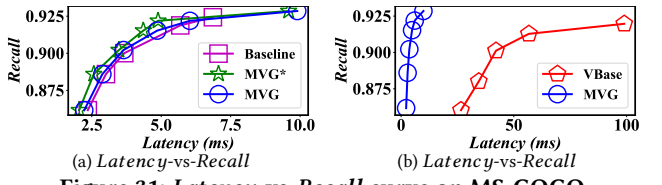
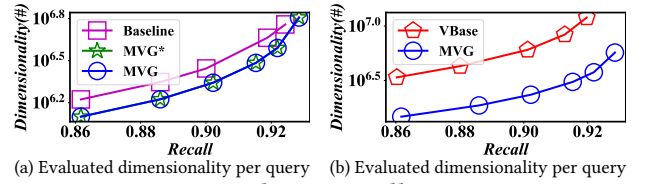
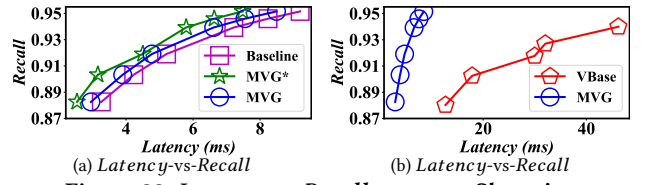
Methods		Recipe	MIT-States	CelebA	FashionIQ	MS-COCO	Shopping
Merging/Milvus/VBase	c_1	800	2,000	800	800	800	800
	r	32	128	32	32	32	32
Baseline/MVG*/MVG	c_1	500	500	500	500	500	500
	r	16	16	16	16	16	16

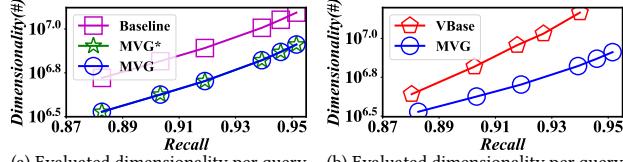
Table 9: Construction time profile for MVG (Time: s).

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

**Figure 24: Dimensionality-vs-Recall curve on Recipe.****Figure 25: Latency-vs-Recall curve on MIT-States.****Figure 26: Dimensionality-vs-Recall curve on MIT-States.****Figure 27: Latency-vs-Recall curve on CelebA.****Figure 28: Dimensionality-vs-Recall curve on CelebA.**

The curves for Merging and Milvus are not included due to their significantly low search efficiency or accuracy.

**Figure 29: Latency-vs-Recall curve on FashionIQ.****Figure 30: Dimensionality-vs-Recall curve on FashionIQ.****Figure 31: Latency-vs-Recall curve on MS-COCO.****Figure 32: Dimensionality-vs-Recall curve on MS-COCO.****Figure 33: Latency-vs-Recall curve on Shopping.**



(a) Evaluated dimensionality per query (b) Evaluated dimensionality per query
Figure 34: 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

O 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

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

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	4,146 (51.1%)
6	44,616	35,718 (19.9%)	17,158	8,260 (51.9%)

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 a better equilibrium between the indexing and searching processes across various m values.

P EFFECT OF INDEX COMPRESSION

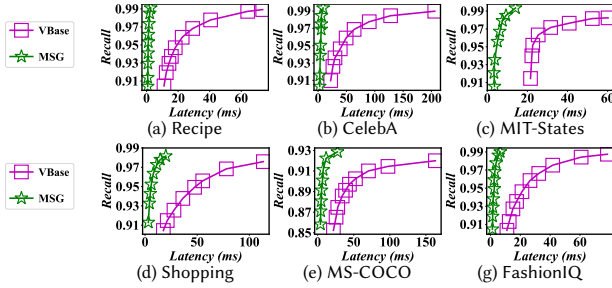
Table 14 displays the index size, neighbor size, and vector size without compression (notated as “w/o C”) and with compression (“w. C”), respectively. We utilized the Med-ID algorithm to compress the index. From the results, our index compression method decreases the neighbor size by a minimum of 49.3%. 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 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	8,213	8,293	8,261
Compression time (s)	0	29.17	28.00	26.35

**Figure 35: Search performance under dynamic weights at the per-query level.**

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”.

Q DYNAMIC WEIGHT AT THE PER-QUERY LEVEL

Figure 35 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 $31 \times$ 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.