ATOMG: Adaptively Decomposed Graph Index for Vector Search

Anxin Tian
HKUST
atian@connect.ust.hk

Lei Chen
Huawei Noah's Ark Lab
lc.leichen@huawei.com

Mingxuan Yuan
Huawei Noah's Ark Lab
yuan.mingxuan@huawei.com

Haoyang Li
Hong Kong Polytechnic University
haoyang-comp.li@polyu.edu.hk

Alexander Zhou
Hong Kong Polytechnic University
alexander.zhou@polyu.edu.hk

Yue Wang
Shenzhen Institute of Computing Sciences
yuewang@sics.ac.cn

Lei Chen
HKUST & HKUST(GZ)
leichen@ust.hk

ABSTRACT

Approximate nearest neighbor search (ANNS) serves as a fundamental problem on high-dimensional and large-volume datasets across diverse applications. While existing graph-based indexes demonstrate superior search performance compared to other methods, they face severe scalability challenges: constructing indexes for large-scale datasets requires excessive time and memory resources. More critically, in dynamic scenarios where data updates are frequent, these methods either require complete reconstruction with additional time and memory overhead, or suffer significant accuracy degradation after updates. This work presents ATOMG (AdapTively decOMposed Graph), a novel decomposion-based graph index that addresses these scalability challenges in both static and dynamic scenarios. ATOMG introduces an adaptive layer decomposition strategy that uniformly partitions nodes based on distance distributions, complemented by efficient inter-layer guidance through representative nodes. Our approach achieves a 10× reduction in construction time compared to state-of-the-art methods, with a 2-3× reduction in memory cost on average. More importantly, ATOMG with efficient batch-update significantly outperforms existing methods, achieving a 20× improvement in update time, while offering stable search performance.

PVLDB Reference Format:
Anxin Tian, Lei Chen, Mingxuan Yuan, Haoyang Li, Alexander Zhou, Yue Wang, and Lei Chen. ATOMG: Adaptively Decomposed Graph Index for Vector Search. PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at .

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Approximate nearest neighbor search (ANNS) serves as a fundamental operation on high-dimension and large-volume vector datasets where exact nearest neighbor search becomes intractable, spanning diverse applications such as retrival-augmented generation [5, 15, 26, 48], image retrieval [27, 41], pattern recognition [47], and bio-sequence matching [9].

In recent decades, many methods have been developed for efficiently solving the ANNS problem: inverted indexes [7, 25, 29], tree-based indexes [28, 31], hash-based indexes [32, 45, 57], and quantization-based indexes [22, 24, 35, 53]. Graph-based ANNS algorithms construct a proximity graph where vertices represent data points and edges connect similar points based on their distances, then perform search by iteratively moving from a seed vertex to closer neighbors until no better candidates are found, they show the overwhelming improved search performance or similar compared to other methods [6, 30, 52]. According to this experimental survey [52], HNSW [34], NSG [20], NSSG [19], and HCNNG [36] are comparable across all datasets in the aspect of search performance. For latest works, τ-MNG [39] provides the theoretical guarantee and reaches SOTA search performance.

However, all existing graph-based indexes encounter the problem of construct-inefficient or memory-hungry when facing large-volume datasets. The experimental results in Section 8 reveal that HNSW [34] exhibits the highest construction overhead, peaking at longer than 24 hours on the Crawl dataset, a set of 2 million 300-dimensional vectors occupying around 3 GB memory. τ-MNG [39] follows closely, requiring similar magnitude of construction time on large datasets like Crawl and GIST1M, with GIST1M being a set of 1 million 960-dimensional vectors taking up around 4 GB memory. In terms of memory consumption, even for static scenarios, the footprint reaches its peak at approximately 20 GB for NSG [20], NSSG [19] and τ-MNG [39].

While such memory footprint might seem manageable on modern commercial hardware in static scenarios, this considerable overhead is still undesirable and becomes a critical bottleneck in dynamic scenarios. When a dataset is dynamically expanding, as is common in real-world applications [5, 24, 46, 48], existing methods either require complete reconstruction with substantial additional time and memory overhead [19, 20, 36, 39], or suffer significant accuracy degradation after updates [34, 55]. The reconstruction approach means that systems must either frequently reallocate memory and rebuild indexes, leading to significant performance degradation [19, 20], or maintain excessive memory reserves in anticipation of dynamic updates, resulting in poor resource utilization [34, 39].

Challenges. The severe scalability challenges of existing graph-based indexes in both construction overhead and dynamic update make them impractical for real-world applications that require large-scale data processing and dynamic updates [26, 48]. We analyze the specific weaknesses of existing methods from three aspects, a more detailed discussion is conducted in Section 4.

• *Long Construction Time.* Building an efficient graph index requires carefully structured graph properties, which leads to substantial construction overhead. This challenge is evident in existing approaches: HNSW's [34] sequential node insertion process, HC-NNG's [36]minimum spanning tree partitioning, and τ-MNG's [39] two-phase construction with graph pruning. These inherent limitations cannot be resolved through trivial modifications, as their long build time stems from the need to satisfy strict graph properties.

• *High Memory Footprint.* Maintaining optimal search performance requires complex graph structures with specific properties, which imposes significant memory overhead during construction. For instance, NSG [20] requires maintaining monotonic paths, NSSG [19] preserves omni-directional connections, and τ-MNG [39] enforces the τ-monotonic property. Their memory consumption stems from complex edge pruning strategies that require maintaining intermediate graphs and connectivity information during the pruning process. Directly removing or relaxing these constraints will severely impair the search performance.

• *The Inefficiency in Dynamic Scenario.* Maintaining comparable search performance given batch updates is very challenging, as the dataset distribution might drift after updates, potentially invalidating the original index structure's assumptions and degrading search efficiency [55]. Existing solutions either suffer from degrading performance with dataset growth (HNSW [34]), require frequent rebuilding (τ-MNG [39]), or simply do not support updates.

Contributions. In Figure 1, we compare index construction cost based on empirical evidence from Section 8, with black/white marks indicating whether dynamic updates are supported. Even though τ-MNG claims its SOTA search performance [39], we can see that it requires high overhead in construction time and memory. Our proposed ATOMG achieves both efficient construction time and low memory usage while exhibiting comparable search performance, where the cost complexities are shown in Table 1. We conclude our contributions as follows.

• We propose ATOMG, a novel decomposition-based graph index that leverages the key insight of reducing both construction overhead and update complexity through distribution-based graph decomposition.

• By maintaining only essential local connectivity within each layer instead of enforcing global graph properties, ATOMG significantly reduces construction time by avoiding complex pruning procedures, while simultaneously lowering memory overhead by reducing auxiliary data structures typically required for global property guarantee.

• We develop an efficient batch-update algorithm that first identifies which layers would be affected by new updates and assesses potential skewness across layers, enabling ATOMG to efficiently localize maintenance operations to only the affected layers, thus avoiding the global reorganization and verification required by traditional monolithic graphs.

• Extensive experiments demonstrate that ATOMG achieves a 10× reduction in construction time and a 2-3× reduction in memory cost on average while maintaining competitive search quality. In dynamic scenarios, our batch-update algorithm achieves a 20× speedup on average in update time compared to the static solution while preserving stable search performance.

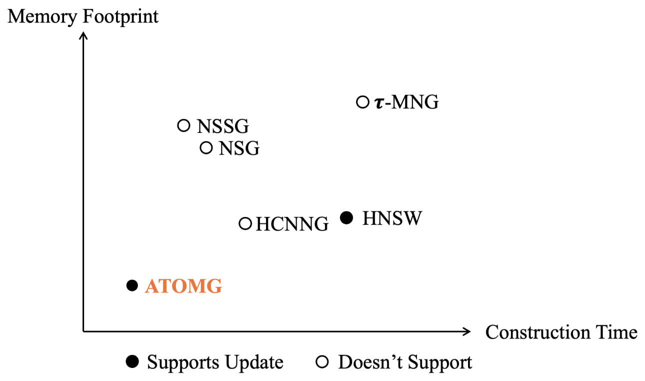


Figure 1: The performance comparison over construction cost based on empirical evidence

Table 1: Comparison over cost complexities, where d denotes the dimension of any vector in the dataset, $|S|$ denotes the number of vectors in the dataset, and deg_{max} is the maximum degree of the graph-based index. The index size can be described in $O(deg_{max} \cdot |S|)$ for all indexes.

Index	Construction time	Query time
HNSW [34]	$O(\|S\| \cdot \log \|S\|)$	$O(\log \|S\|)$
NSG [20]	$O(\|S\| \cdot \log \|S\| + \|S\|^{1.14})$	$O(\log \|S\|)$
NSSG [19]	$O(\|S\| + \|S\|^{1.14})$	$O(\log \|S\|)$
HCNNG [36]	$O(\|S\| \cdot \log \|S\|)$	$O(\|S\|^{0.4})$ [52]
τ-MNG [39]	$O(deg_{max}^2 \cdot \log deg_{max})$	$O(\log \|S\|)$
ATOMG	$O(\|S\| \cdot \log \|S\| + \|S\|^{1.14})$	$O(\log \|S\|)$

Outline. In Section 2, we review the related work on this problem. Section 3 defines concepts and problems. Section 4 reveals the inefficiency of existing graph-based indexes in both static and dynamic scenarios. We overview the index structure and the search process in Section 5. In Section 6, we present detailed construction algorithms of our index. We show how to maintain our proposed index dynamically given to-be-updated vectors in Section 7. Finally, we evaluate our methods with extensive experiments in Section 8 and conclude this work in Section 9.

2 RELATED WORK

In this section, we review closely related work to this problem. We conclude the related work in two aspects: graph-based methods and non-graph-based methods.

Graph-based Methods. For different graph-based methods, they may deploy different base graphs, we can classify the base graphs into several basic types:

• *Delaunay Graph (DG)-based:* The Delaunay graph [17] $G(V, E)$ in E^d has the empty circle property: for each edge, there exists a circle passing through its endpoints with no other vertices inside, and at most three vertices can lie on the circle simultaneously. DG guarantees precise ANNS results [34], but its near-complete connectivity in high dimensions leads to a large search space [20, 23]. Several methods have been proposed to address the high degree issue of DG in high dimensions. NSW [33] constructs an approximate DG with global connectivity, balancing search efficiency and accuracy through long edges formed early and short-range edges added later. However, its search complexity is poly-logarithmic [37]. HNSW [34] improves upon NSW by generating a hierarchical graph with a fixed upper bound on the number of neighbors per vertex, enabling logarithmic search complexity.

• *K-Nearest Neighbor Graph (KNNG)-based:* Naive KNNG construction via exhaustive pairwise comparisons is prohibitively slow for large datasets. Early solutions using additional indexes (e.g. trees [38], hashes [56]) for ANNS to find neighbors suffer from high construction complexity. SPTAG [13] is a representative divide-and-conquer approach for KNNG construction. It hierarchically divides the dataset into subsets using Trinary-Projection Trees [51], builds exact KNNGs on each subset, and repeats the process to improve the overall KNNG accuracy. SPTAG further enhances accuracy through neighborhood propagation. NN-Descent [14] is based on the idea that neighbors are more likely to be neighbors of each other. It adopts this principle to reduce KNNG construction complexity to $O(|S|^{1.14})$ on dataset S.

• *Relative Neighborhood Graph (RNG)-based:* RNG [50] in Euclidean space connects vertices x and y only if there is no other vertex z that is closer to both x and y, reducing redundant neighbors compared to DG and ensuring omnidirectional distribution of remaining neighbors. FANNG [23] proposes an occlusion rule to approximate RNG by cutting off redundant neighbors. To improve accuracy, FANNG employs a backtrack to the second-closest vertex and considers its unexplored edges. NSG [20] addresses the large index issue and enhances search performance by proposing an edge selection strategy based on monotonic RNG (MRNG). It ensures high construction efficiency by executing ANNS to obtain candidate neighbors. NSSG [19] further explores edge pruning and proposes an edge selection strategy. Both NSSG and NSG approximate RNG, NSSG has a larger out-degree compared to NSG because NSSG is relatively relaxed when cutting redundant neighbors. DiskANN [44] is designed for billion-scale data on SSDs. Inspired by NSG's framework but using random initialization, DiskANN enhances HNSW's neighbor selection strategy by introducing a flexibility parameter. SoarGraph [12] is proposed for out-of-distribution vector search problem and shows the effectiveness for multi-modal scenario, which is not suitable for the in-distribution setting.

• *Minimum Spanning Tree (MST)-based:* HCNNG [36] is a method that differs from the aforementioned techniques by using MST to connect points in the dataset. It follows the same divide-and-conquer framework as SPTAG [13] but divides the dataset through multiple hierarchical clusters, with all points in each cluster connected via MST. HCNNG obtains seeds using multiple global KD-trees [8], similar to SPTAG [13] and EFANNA [18]. To improve search efficiency, it employs an efficient guided search instead of the traditional greedy search.

Non-graph-based Methods. Other than graph-based indexes, many other methods are also proposed. Recent studies [6, 30, 52] show that non-graph-based methods are outperformed by graph-based methods, thus we introduce them generally.

• *Inverted Index:* The inverted file index [7, 25, 29] decomposes the feature space into orthogonal subspaces, independently partitioning each subspace into Voronoi regions, forming a fine-grained space partition that creates candidate lists.

• *Tree-based Index:* Tree-based indexes [28, 31] transform datasets into an organized tree structure where similar vectors are grouped together, enabling efficient search operations. This structure works like a decision tree, where each internal node represents a splitting criterion that helps narrow down the search space.

• *Hashing-based Index:* Hashing-based indexes [32, 45, 57] ensure that similar vectors in the original space have a high probability of being mapped to the same hash bucket. When searching for similar vectors, instead of examining the entire dataset, it only needs to search vectors that share the similar hash codes as the query vector.

• *Quantization-based Index:* Quantization-based indexes [22, 24, 35, 53] works by breaking down high-dimensional vectors into smaller subvectors and quantizing each subvector independently. Each subspace is assigned a smaller codebook of representative values. The search can be performed using pre-computed lookup tables of distances between codebook entries.

3 PRELIMINARIES

In this section, we give the formal definition of concepts and the problem statement. The table of all notations can be found in Table 2. The proofs of this work can be referred to our technical report [49].

3.1 Problem Definition

Our problem is studied on a a dataset $S = \{s_0, s_1, \ldots, s_{n-1}\}$ consisting of n vectors, where each element s_i in S is represented by a vector $\mathbf{x} = [x_0, x_1, \ldots, x_{d-1}]$ with d dimensions. By employing a similarity measurement function of vectors on S, we can effectively analyze and retrieve the corresponding data.

We use $V(G)$ to denote the set of vertices and $E(G)$ to denote the set of edges that $E(G) = \{\langle u, v\rangle \mid u, v \in V(G)\}$, where $\langle u, v\rangle$ implies the direction from u to v.

Definition 1. ***Euclidean Distance*** [42]. For the two points x, y on dataset S, a variety of applications employ a distance function to calculate the similarity between the two points x and y. The distance function this work used is the Euclidean distance $\delta(x, y)$ (l_2 norm), which is given below,

$$\delta(x, y) = \sqrt{\sum_{i=0}^{d-1}(x_i - y_i)^2}$$

where x and y correspond to the vectors $x = [x_0, x_1, \ldots, x_{d-1}]$, and $y = [y_0, y_1, \ldots, y_{d-1}]$, respectively, here d represents the vectors' dimension. Note that our work is distance metric agnostic, i.e. one may substitute other metrics in with no effect on performance.

Table 2: Notations

Notation	Description
E^d	The euclidean space with dimension of d
S	The vector dataset
$\lvert \cdot \rvert$	The cardinality of a set
$\delta(\cdot, \cdot)$	The Euclidean distance between two vectors
G	A graph-based index
$V(G)$	The set of vertices of the graph-based index
$E(G)$	The set of edges of the graph-based index
q	The query vector in E^d
$N(v)$	The neighbor set of the vector v
$\mathcal{L}$	The number of layers in ATOMG
$\mathcal{B}$	The edge budget for Intra-Layer Connection

Problem Statement 1. ***Nearest Neighbor Search (NNS)*** [16, 52]. Given a finite dataset S in Euclidean space E^d and a query q, NNS obtains k nearest neighbors Z of q by evaluating $\delta(x, q)$, where $x \in S$. Z is described as follows:

$$Z = \arg \min_{Z \subset S, |Z|=k} \sum_{x \in Z} \delta(x, q).$$

Problem Statement 2. ***Approximate Nearest Neighbor Search (ANNS)*** [4, 10, 52]. Given a finite dataset S in Euclidean space E^d, and a query q, ANNS builds an index I on S. It then gets a subset C of S by I, and evaluates $\delta(x, q)$ to obtain the approximate k nearest neighbors $\tilde{Z}$ of q, where $x \in C$.

Generally, we use the recall rate $Recall@k = \frac{|Z \cap \tilde{Z}|}{k}$ to evaluate the search results' accuracy. ANNS algorithms aim to maximize $Recall@k$ while making C as small as possible (e.g., $|C|$ is only a few thousand when $|S|$ is millions on the SIFT1M dataset). We define graph-based ANNS as follows.

Definition 2. ***Graph-based ANNS*** [52]. Given a finite dataset S in Euclidean space E^d, $G(V, E)$ denotes a graph constructed on S, where $\forall v \in V$ uniquely corresponds to a point x in S. Here, $\forall (u, v) \in E$ represents the neighbor relationship between u and v, and $u, v \in V$. Given a query q, seeds $\hat{S}$, routing strategy, and termination condition, the graph-based ANNS initializes approximate k nearest neighbors $\tilde{Z}$ of q with $\hat{S}$, then conducts a search from $\hat{S}$ and updates $\tilde{Z}$ via a routing strategy. Finally, it returns the query result $\tilde{Z}$ once the termination condition is met.

4 INEFFICIENCY OF EXISTING GRAPH-BASED INDEXES

In this section, we explore the existing graph-based indexes and investigate the reasons for their inefficiency in real-world applications. We introduce the causes from two aspects: high construction cost and the drawbacks in dynamic scenario.

4.1 High Construction Cost

In order to investigate the reasons behind the long construction times of existing graph-based indexes, we conducted an in-depth analysis of the construction process of each index.

For HNSW [34], it hierarchically organizes neighbors based on distance. Layers closer to the top retain long-distance connections, while layers nearer to the bottom retain short-distance connections. Its construction process requires dynamically inserting each node into the index. During insertion, it dynamically adjusts the number of layers and maintains the navigable small world properties at each layer [33]. Although such a design ensures a logarithmic complexity scaling of search, it results in a construction time of $O(|S| \cdot \log |S|)$. What's more, its highly cohesive layered structure requires high memory resources, thereby limiting its scalability for extensive datasets [52].

For HCNNG [36], it employs a divide-and-conquer framework for the construction. It partitions the input dataset into multiple stratified clusters, where vertices within each cluster are interconnected through a minimum spanning tree (MST) structure. These MSTs are merged into the final graph. In addition, it needs to construct an additional tree-like structure to improve search performance. It also results in a construction time of $O(|S| \cdot \log |S|)$.

NSG [20] implements an edge-pruning mechanism derived from monotonic RNG. Its construction necessitates edge removal on KNNG to maintain monotonic paths between the entry point and all nodes. However, this node-level link constraint imposes significant memory requirements. To alleviate the stringent constraint in NSG of ensuring a monotonic path for each node, NSSG [19] tries to improve the edge pruning strategy. Instead of preserving monotonic paths, it aims to retain omni-directional connections among 2-hop neighbors. However, according to results from this experimental survey [52], NSSG's construction overhead and search performance remain comparable to those of NSG.

τ-MNG [39] demonstrates state-of-the-art search performance and provides theoretical guarantees on the approximated search results. However, the construction process that needs to satisfy the theoretical guarantee also incurs substantial overhead. It has to first initialize using NSG or HNSW as the base graph. Then, it needs to check and insert new edges to ensure that the final graph satisfies the τ-monotonic property [39]. As a result, its construction process inevitably leads to high construction overhead exceeding that of NSG and HNSW.

4.2 The Drawbacks in Dynamic Scenario

To the best of our knowledge, among existing graph indexes, only a few possess the ability to handle dynamic updates. However, they all have obvious disadvantages. We provide a detailed introduction to existing dynamic maintenance methods below and explain why they are not satisfactory in dynamic update scenarios.

• *SPFresh* [54]: SPFresh enables the DiskANN algorithm [44] to support dynamic updates. However, both DiskANN and SPFresh are designed for on-SSD computation. The contribution of SPFresh lies in rebalancing techniques for on-disk vector partitions. This rebalancing algorithm are tightly hardware-coupled and lack the capability to migrate to in-memory computing. Overall, SPFresh cannot be directly employed to maintain these various in-memory graph indexes.

• *HNSW* [34]: Due to its inherent construction process, HNSW naturally supports vector insertion operations. However, its update efficiency decreases as the dataset grows larger, and HNSW only supports single-unit insertions, which cannot meet the demands for batch updates in real-world applications.

• *τ-MNG* [39]: τ-MNG claims that it can support vector insertion and deletion. For insertions, it directly adopts HNSW's insertion technique. However, it faces the risk of degraded performance after a certain number of updates, so it must periodically rebuild from scratch after $O(\log|S|)$ updates [39]. Such a high reconstruction overhead is unacceptable in real-world scenarios.

5 INDEX OVERVIEW

In this section, we provide an overview of our ATOMG index, including its structure and search process. The index structure is based on the layerization performed on a base graph, with connections reorganized both within and between layers. Unlike other indexes, to avoid issues of long construction times or high memory consumption, we relax the connection conditions and specify a search algorithm that focuses on rapidly narrowing down the search scope, thereby providing comparable search performance.

5.1 Index Structure

Based on the findings of this experimental survey [52], among the indexes we discussed, NSG [20] and NSSG [19] have the fastest construction time. This heuristic verifies that using k Nearest Neighbour Graph (KNNG) [14] as the base graph and then adjusting the edges is relatively time-efficient. Following this idea, we also adopt KNNG for initialization, allowing us to quickly obtain the big picture of the distance distribution among vectors in the dataset.

Intuitively, a well-designed index should be able to rapidly narrow down the search scope based on the query vector. Existing hierarchical indexes such as HNSW [34] and HCNNG [36] leverage the divide-and-conquer strategy to achieve this goal. By partitioning the dataset into multiple clusters or layers, these indexes enable the search algorithm to focus on the most relevant partitions, significantly reducing the number of nodes that need to be explored during a query. However, as described in Section 4.1, their hierarchical construction design leads to long construction time and degrades the scalability on large datasets.

Based on above considerations, our AdapTively decOMposed Graph (ATOMG) index is proposed. In Figure 2, we show the structure of our ATOMG index.

Starting from the initialized KNNG, we decompose the nodes uniformly into disjoint layers according to the distance distribution between nodes (See Section 6.2). We can adaptively set the number of layers $\mathcal{L}$ based on the volume of the dataset. This structure maintains that nodes closer to the top layer have larger distances, while those closer to the bottom layer have shorter distances. Unlike HNSW's neighbor-based organization, which results in redundant node storage across different layers, we decompose the nodes into disjoint sets to limit the construction overhead and index size.

After obtaining the layers, we select a representative node for each layer and organize connections among these representative nodes to serve as layer guidance (See Section 6.3). This design helps the query vector rapidly locate the target layer during the search process. Connections within each layer will also be adjusted for improving the search process (See Section 6.4). We show the size of the ATOMG index in Proposition 5.1.

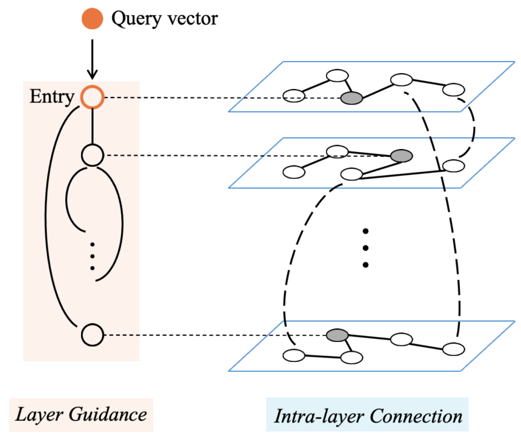


Figure 2: The ATOMG index structure

Proposition 5.1. *The ATOMG size is bounded by $O(deg_{max} \cdot |S|)$.*

Proof. Suppose that the maximum degree in the graph index is deg_{max}, and the number of layers is $\mathcal{L}$. The size of the layer guidance is bounded by $O(deg_{max} \cdot \mathcal{L})$. The size of each layer is bounded by $O(deg_{max} \cdot \frac{|S|}{\mathcal{L}})$. Therefore, the overall index size is bounded by $O(deg_{max} \cdot \mathcal{L} \cdot \frac{|S|}{\mathcal{L}}) = O(deg_{max} \cdot |S|)$. □

5.2 Search Algorithm

We propose the search algorithm that works with the ATOMG index in Algorithm 1. The algorithm takes as input a query vector q, an ATOMG index G and desired result size k. The output will be the top k nearest neighbors of q in G. First, it gets the entry point e as the representative node of the top layer. After that, it calls the LayerGuidance function to determine the target layer. Then, it calls the InLayerSearch function to find nearest neighbors and finally returns the found nearest neighbors.

For the LayerGuidance Function (lines 5-16): it starts with a queue containing the entry point and its neighbors (lines 5-6). Then it uses an iterative process to explore nodes based on their distance to the query point and maintains a candidate set of promising nodes (line 7-15). For each iteration, it selects the closest node to the query from the queue (lines 8-11) Then it explores its neighbors and updates the candidate set with better neighbors (lines 12-15). Finally, returns the best target node to guide the search (line 16). For the InLayerSearch Function (lines 18-21), it performs the actual nearest neighbor search within the target layer by using a similar BFS-based exploration strategy as LayerGuidance. We analyze the time cost of Algorithm 1 in Proposition 5.2.

Proposition 5.2. *For Algorithm 1, the time complexity is $O(\log|S|)$.*

```
Algorithm 1: Search Algorithm on ATOMG
Input: Query vector $q \in E^d$, the ATOMG index $G$, the desired size
       of the result $k$
Output: Top $k$ nearest neighbor of $q$ in $G$
1  Entry point $e \leftarrow$ the representative node of the top layer;
2  $R \leftarrow$ layer representatives; $tar \leftarrow$ LayerGuidance($q, R, e$);
3  return InLayerSearch($q, tar, k$);
   Function LayerGuidance($q, R, e$):
5      $Q \leftarrow \{e\} \cup N(e)$ ;  // Queue with entry and neighbors
6      $C \leftarrow \emptyset$ ;                              // Candidate set
7      while $C$ not updated do
8          $v \leftarrow \arg\min_{x \in Q} \delta(x, q)$;
9          $Q \leftarrow Q \setminus \{v\}$;
10         $Q \leftarrow Q \cup N(v)$ ;
11         Resize $Q$;
12         foreach $u \in N(v)$ do
13             $w \leftarrow \arg\max_{x \in C} \delta(x, q)$;
14             if $\delta(u, q) < \delta(w, q)$ then
15                 $C \leftarrow (C \setminus \{w\}) \cup \{u\}$;
16     return $\arg\min_{x \in C} \delta(x, q)$;
   Function InLayerSearch($q, layer, k$):
18     Queue initialized with $layer$'s representative node;
19     $R \leftarrow \emptyset$ ;
20     Perform similar BFS as in the LayerGuidance function;
21     return $R$;
```

Proof. For Algorithm 1, the search time of the guidance procedure is bounded by $O(\log \mathcal{L})$, the time cost in the target layer is bounded by $O(\log \frac{|S|}{\mathcal{L}})$. Thus, the time complexity of Algorithm 1 is $O(\log \mathcal{L} + \log \frac{|S|}{\mathcal{L}}) = O(\log(\mathcal{L} \cdot \frac{|S|}{\mathcal{L}})) = O(\log |S|)$ in total. □

6 INDEX CONSTRUCTION

In this section, we provide a detailed description of the index construction process. First, we introduce the overall construction procedure. Then, we discuss three important components: Layerization, Layer Guidance Connection, and Intra-Layer Connection. Finally, we elaborate the optimizations to enhance construction efficiency and search accuracy.

6.1 Overall Procedure

We present the construction algorithm of the ATOMG index in Algorithm 2. The process first constructs a base graph by K-Nearest Neighbor Graph (KNNG) [14], which establishes initial connections between vector nodes (line 1), where K follows the optimal setting from this experimental survey[52]. Upon this foundation, the algorithm performs a layerization process that decomposes the graph into multiple layers, bounded by the number of layers $\mathcal{L}$ (line 2). This layered structure is then enhanced through two crucial connection phases: first, the Layer Guidance Connection creates pathways between different layers, enabling efficient navigation through all layers (line 3); second, the Intra-Layer Connection optimizes the connections within each individual layer (lines 4-5). This dual-connection strategy ensures both efficient inter-layer movement and thorough intra-layer exploration. Finally, all these carefully constructed layers and their connections are consolidated into the final graph index structure (lines 6-7).

```
Algorithm 2: Construction of ATOMG
Input: Dataset $S$, The number of layers $\mathcal{L}$;
Output: The ATOMG index $G$;
1  Construct the base graph by KNNG [14];
2  $layers \leftarrow$ Layerization($KNNG, \mathcal{L}$) ;          // Algorithm 3
3  Layer Guidance Connection($layers$) ;             // Algorithm 4
4  foreach $layer$ in $layers$ do
5      Intra-Layer Connection($layer$) ;             // Algorithm 5
6  Conclude all layers into the final graph index $G$;
7  return $G$;
```

Next, we will introduce the three key components: Layerization (Section 6.2), Layer Guidance Connection (Section 6.3), and Intra-Layer Connection (Section 6.4) in order.

6.2 Layerization

The Layerization aims at decomposing the base graph KNNG into a hierarchical structure of multiple layers. We present this process in Algorithm 3. It begins by establishing a critical tail threshold through the estimation of distance distribution. Based on the observation that real-world datasets follow the long-tail distribution, we devise a threshold determination function to make the layerization be distance-distribution-aware (lines 1, 13-20). The DetermineTailThreshold function begins by randomly sampling a subset of nodes from the adjacency list of KNNG, specifically collecting the distances to their first (i.e. nearest) neighbors (lines 13-16). Then it proceeds by constructing an empirical cumulative distribution function (ECDF) from these sorted distance samples, the distance value at the point with the maximum change of derivative of ECDF is set to be the tail threshold (lines 17-20).

This threshold serves as a decisive boundary for categorizing nodes into two distinct groups: tail elements and normal elements. The categorization is based on each node's relationship with its furthest neighbor - nodes whose last neighbor exceeds the tail threshold are classified as tail elements, while the remaining nodes are marked as normal elements (lines 3-7). After this initial classification, the algorithm proceeds to organize these nodes into $\mathcal{L}$ layers. The layer structure is initialized with equal capacity allocations, ensuring a balanced distribution of nodes across layers. The algorithm then follows a two-phase distribution strategy: first, it processes the tail elements, assigning them to the upper layers of the hierarchy by their distance values in descending order (line 9). This ensures that nodes with more distant connections are positioned in the higher layers. Subsequently, the normal elements are distributed across the remaining layers with their neighbors from KNNG in priority (line 10). The complexity analysis is given in Lemma 6.1.

Lemma 6.1. *For Algorithm 3, the time complexity is $O(|S|)$, the space complexity is $O(|S|)$.*

6.3 Layer Guidance Connection

According to Algorithm 1, the entry point is set to be the representative node of the top layer. For optimizing the guidance among layers,

```
Algorithm 3: Layerization
  Input: The base graph KNNG $G$, number of layers $\mathcal{L}$
  Output: Layered decomposition of $V(G)$
1 $\tau \leftarrow$ DetermineTailThreshold($G$, $sampleSize$);
2 $V_{tail}, V_{normal} \leftarrow \emptyset, \emptyset$;
3 foreach $v \in V(G)$ do
4     if $\delta(v, N(v)_{last}) > \tau$ then
5         $V_{tail} \leftarrow V_{tail} \cup \{v\}$;
      else
7         $V_{normal} \leftarrow V_{normal} \cup \{v\}$;
8 Initialize $\mathcal{L}$ layers with capacity $|S|/\mathcal{L}$;
9 Distribute $V_{tail}$ and their neighbors to top layers by descending $\delta$;
10 Fill remaining layers with $V_{normal}$;
11 return $layers$ ;
  Function DetermineTailThreshold($G$, $sampleSize$):
13    $D \leftarrow \emptyset$ ;                         // Sampled distances
14    for $i \leftarrow 1$ to $sampleSize$ do
15        $v \leftarrow$ Random vertex from $V(G)$;
16        $D \leftarrow D \cup \{\delta(v, N(v)_1)\}$;
17    Sort $D$ in ascending order;
18    Compute Empirical Cumulative Distribution Function of $D$;
19    $x \leftarrow \arg_{\max}$ derivate change in ECDF;
20    return corresponding distance at $x$;
```

we formulate the Optimial Layer Guidance Connection Problem (OLGCP) below, and its hardness is shown in Theorem 6.2.

Definition 4. ***Optimial Layer Guidance Connection Problem (OLGCP)***. Given the representative nodes of layers, the Optimial Layer Guidance Connection Problem (OLGCP) is to insert edges such that: the expectation of the sum of the out-degree along the path between the entry point to the target layer is minimized and the out-degree of the entry point is minimized.

Theorem 6.2. *OLGCP is NP-Hard.*

Due to the NP-hardness of OLGCP, it is computationally infeasible to find a globally optimal solution, thus we adopt a greedy local optimization strategy that continuously refines the guidance through local improvements in Algorithm 4.

The Layer Guidance Connection algorithm takes as input the layered structure produced by the previous algorithm and outputs the same structure enhanced with optimized guidance connections. For the generation of the representative nodes, the random selection is deployed by default, with the top layer's representative serving as the entry point (lines 1-2). It proceeds in three distinct phases. In the first phase, it establishes initial connections by connecting each node to its nearest neighbor among nodes that are closer to the entry point, ensuring basic reachability throughout the structure (lines 3-6). The second phase focuses on optimizing these connections through an iterative process (lines 7-19). For each node except the entry point, it explores alternative connections to candidates that are closer to the entry point than the current node (lines 8-10). When evaluating each potential connection change, it temporarily modifies the graph structure and uses a cost evaluation function to determine if the change improves overall performance (lines 11-19). This evaluation function samples queries near the target node and calculates the average path cost based on node degrees along the path (see [49]). The final phase specifically optimizes the entry point's connections to reduce its outdegree while maintaining performance (lines 20-32). The complexity analysis of Algorithm 4 is presented in Lemma 6.3.

```
Algorithm 4: Layer Guidance Connection
  Input: Layered decomposition from Algorithm 3
  Output: Layers with guidance connections
1 $R \leftarrow$ Generate layer representatives;
2 $e \leftarrow$ representative of top layer;
  /* Phase 1: Initial Connection                        */
3 foreach $v \in R \setminus \{e\}$ do
4     $S \leftarrow \{u \in R \mid \delta(e,u) < \delta(e,v)\}$;
5     $u_{min} \leftarrow \arg\min_{u \in S} \delta(u,v)$;
6     $E(G) \leftarrow E(G) \cup \{(u_{min}, v)\}$;
  /* Phase 2: Connection Optimization                   */
7 while improvements possible do
8     foreach $v \in R \setminus \{e\}$ do
9         Let $u_c$ be the node where $(u_c, v) \in E(G)$;
10        $C \leftarrow \{u \in R \mid \delta(e,u) < \delta(e,v)\}$;
11        foreach $u \in C$ do
12            $E(G) \leftarrow E(G) \setminus \{(u_c, v)\}$;
13            $E(G) \leftarrow E(G) \cup \{(u, v)\}$;
14            $cost_{new} \leftarrow$ EvaluateCost($G$, $v$);
15            if $cost_{new}$ decreases then
16                keep $(u, v)$;
              else
18                $E(G) \leftarrow E(G) \cup \{(u_c, v)\}$;
19                $E(G) \leftarrow E(G) \setminus \{(u, v)\}$;
  /* Phase 3: Entry Point Optimization                  */
20 while $|N(e)|$ reducible do
21    foreach $v \in N(e)$ do
22        $S \leftarrow \{u \in R \setminus \{e\} \mid \delta(e,u) < \delta(e,v)\}$;
23        foreach $u \in S$ do
24            $E(G) \leftarrow E(G) \setminus \{(e, v)\}$;
25            $E(G) \leftarrow E(G) \cup \{(u, v)\}$;
26            $cost_{new} \leftarrow$ EvaluateCost($G$, $v$);
27            if $cost_{new}$ decreases then
28                keep changes;
29                break;
              else
31                $E(G) \leftarrow E(G) \cup \{(e, v)\}$;
32                $E(G) \leftarrow E(G) \setminus \{(u, v)\}$;
33 return layers;
```

Lemma 6.3. *For Algorithm 4, the average time complexity is $O(|\mathcal{L}|^2)$, the space complexity is $O(\mathcal{L}^2)$.*

6.4 Intra-Layer Connection

For optimizing the connections within each layer. Similar to OLGCP, we need to optimize the connections from each layer's representative node to other nodes within the layer. However, unlike the guidance, the number of nodes in a layer is usually much greater than $\mathcal{L}$. To effectively keep the index size under control and optimize the complexity of the construction time, we introduce the edge budget constraint and define the Optimal Intra-Layer Connection Problem (OILCP) below. The hardness of the problem is proved by Theorem 6.4.

Definition 5. ***Optimal Intra-Layer Connection Problem (OILCP).*** Given the representative node r, the maximum edge budget $\mathcal{B}$, and any layer in the ATOMG index, the Optimal Intra-Layer Connection Problem (OILCP) is to adjust edges such that: the expectation of the sum of the out-degree along the path between r to the target answer is minimized, the out-degree of r is minimized, and the edge number is no larger than $\mathcal{B}$.

Theorem 6.4. *OILCP is NP-Hard.*

The Intra-Layer Connection algorithm aims to optimize connections within a given layer while respecting an edge budget $\mathcal{B}$. We present this procedure in Algorithm 5. In the initialization phase, the algorithm begins by collecting all nodes within the target layer and their existing connections from the base graph. A priority queue is initialized to manage edge selection (lines 1-3). During the edge evaluation phase (lines 4-11), it processes two types of edges. First, it evaluates all existing edges in the graph by calculating a score for each edge using the EvaluateEdge function, which combines edge usage frequency and distance benefit (lines 4-6). Second, it considers potential new edges between nodes within a 2-hop distance from the representative node r, but only if the destination node is farther from r than the source node. These edges are also scored and added to the priority queue (lines 7-11).

The edge optimization phase builds a new edge set E' by selecting edges from the priority queue in order of their scores (lines 12-17). For each selected edge, the algorithm checks if adding it would improve the query cost while respecting the edge budget $\mathcal{B}$ by the function in our report [49]. When an edge is added, the priority queue is updated (lines 15-17). Finally, in the representative node optimization phase, the algorithm attempts to reduce the degree of the representative node r (lines 18-24). For each edge connected to r, it searches for alternative paths through intermediate nodes that could provide better overall path costs. If such improvements are found, the direct connection from r is replaced with the optimized path. The algorithm concludes by returning the layer with its optimized edge set E' (line 25). The theoretical analysis of Algorithm 5 is shown in Lemma 6.5.

Lemma 6.5. *For Algorithm 5, the time complexity is $O(\frac{|S|}{\mathcal{L}}\log(\frac{|S|}{\mathcal{L}}))$, and the space complexity is $O(\frac{|S|}{\mathcal{L}})$.*

By Lemma 6.1, 6.3 and 6.5, the time complexity of Algorithm 2 is given in Theorem 6.6.

Theorem 6.6. *The time complexity of Algorithm 2 is $O(|S| \cdot \log|S| + |S|^{1.14})$.*

Proof. The dominant cost of Algorithm 2 consists of the construction of base graph and layer-related connection, which takes $O(|S|^{1.14})$ and $O(\frac{|S|}{\mathcal{L}} \cdot \log\frac{|S|}{\mathcal{L}} \cdot \mathcal{L}^2) = |S| \cdot \log\frac{|S|}{\mathcal{L}}$, where $\mathcal{L}$ is a constant. Thus the overall time complexity is $O(|S| \cdot \log|S| + |S|^{1.14})$. □

Algorithm 5: Intra-Layer Connection

```
Input: Layer from Algorithm 4, edge budget $\mathcal{B}$
Output: Layer with intra-layer connections
/* Phase 1: Initialization */
1  $V \leftarrow V(layer)$; $E \leftarrow E(G_{base})$; $r \leftarrow$ layer representative;
2  $d \leftarrow$ ShortestPaths($G$, $r$); $P \leftarrow$ CriticalPaths($G$, $r$);
3  $f \leftarrow$ EdgeFrequency($P$); Priority queue $Q \leftarrow \emptyset$;
/* Phase 2: Edge Evaluation */
4  foreach $(u, v) \in E$ do
5      $s \leftarrow$ EvaluateEdge($u, v, f, d$);
6      $Q$.insert($\{(u, v), s\}$);
7  foreach $u \in \{x \in V \mid d(r, x) \leq 2\}$ do
8      foreach $v \in \{y \in V \mid d(r, y) > d(r, u)\}$ do
9          if $(u, v) \notin E$ then
10             $s \leftarrow$ EvaluateEdge($u, v, f, d$);
11             $Q$.insert($\{(u, v), s\}$);
/* Phase 3: Edge Optimization */
12 $E' \leftarrow \emptyset$;
13 while $|E'| < \mathcal{B}$ and $Q \neq \emptyset$ do
14     $(u, v), s \leftarrow Q$.max();
15     if IsValidAddition ($G', u, v, \mathcal{B}$) then
16         $E' \leftarrow E' \cup \{(u, v)\}$;
17         UpdatePriorityQueue($Q, G', u, v$);
/* Phase 4: Representative Node Optimization */
18 while $|N(r)|$ reducible do
19     foreach $v \in N(r)$ do
20         $C \leftarrow \{u \in V \mid d(r, u) < d(r, v)\}$;
21         $u^* \leftarrow \arg\min_{u \in C}$ PathCost($r \rightsquigarrow v$ via $u$);
22         if PathCost via $u^*$ improves then
23             $E' \leftarrow E' \setminus \{(r, v)\}$;
24             $E' \leftarrow E' \cup \{(u^*, v)\}$;
25 return layer with $(V, E')$;
```

6.5 Optimizations

Representative nodes selection. Selecting representative nodes within each layer is crucial for efficient navigation and search. While our previous approach relied on random selection for simplicity, we propose a more effective optimization based on the KNNG that naturally exists within each layer. The key insight is that good representative nodes should be located in dense regions of the data space, as these nodes are more likely to serve as effective entry points for searching their surrounding areas. Our optimization leverages the degree information from the base graph as a simple yet effective measure of local density. The selection process iteratively chooses the node with the highest local density as a centroid, then removes both this node and its nearest neighbors from consideration before selecting the next centroid. This removal step ensures that the selected centroid is well-distributed across the layer rather than clustering in a single dense region. The process continues until the desired number of centroids is reached.

Margin nodes preservation. When conducting the intra-layer connection, there may exist situations where margin nodes are assigned to different layers. Simply maintaining connections within each layer based on the base graph may lead to loss of important cross-layer relationships, especially for nodes near layer boundaries. To address this issue, we propose a margin-aware enhancement optimization that identifies and preserves critical cross-layer connections. The approach first identifies and preserves margin nodes, i.e. nodes that have strong connections to different layers in the base graph. Furthermore, we extend the margin region by sampling the neighbors of identified margin nodes. When searching, these preserved cross-layer connections allow the algorithm to explore relevant nodes in adjacent layers, reducing the risk of missing important results that happened to be assigned to different layers.

7 INDEX MAINTENANCE

In this section, we consider the scenario where the dataset undergoes dynamic updates, i.e., it may be necessary to delete outdated vectors and insert new ones. Starting from the most basic case, we first examine the impact of single-vector deletion and insertion on the index structure and respectively propose algorithms to handle single-unit updates. Then, to achieve more efficient updates, we endow the index with the capability to process batch updates.

7.1 Single Deletion

The deletion operation in the ATOMG index requires careful handling to maintain the index's connectivity and structural properties. We propose the single deletion algorithm in Algorithm 6. It consists of three main phases: node removal, connectivity restoration, and representative node updates.

In the first phase, when deleting a vector, we first locate its corresponding node in the graph index (line 1). If the node is found, we remove all its connections across all layers it exists in by updating the neighbor lists of all connected nodes. The node is then removed from the graph's node set (lines 2-4). The second phase focuses on maintaining graph connectivity in the affected area (lines 5-6, see [49]). The final phase handles the special case where the deleted node was a layer representative (lines 7-8). In this situation, we need to reset the representative nodes in the affected area. For each affected node, starting from its highest layer, we evaluate whether it should become a representative node. If selected as a representative, the node restores connections with representatives for the layer guidance (Refer to [49] for details). The overall time complexity is dominated by the connectivity restoration phase, resulting in $O(L_{aff} \times |S|/\mathcal{L})$, where L_{aff} denotes the number of layers that covered by the affected area.

Algorithm 6: Delete a Single Vector from ATOMG

```
Input: Vector to delete $v_{del}$, graph index $G$
Output: Updated graph index $G_{new}$
1 $u_{del} \leftarrow$ FindNode$(G, v_{del})$;
2 foreach $w \in N_l(u_{del})$ do
3     $N_l(w) \leftarrow N_l(w) \setminus \{u_{del}\}$;
4 $V(G) \leftarrow V(G) \setminus \{u_{del}\}$;
5 $A \leftarrow \{u_{del}\} \cup \bigcup_l N_l(u_{del})$ ;                // Affected area
6 $G \leftarrow$ RestoreConnectivity$(G, A)$;
7 if $u_{del} \in R(G)$ then
8     $G \leftarrow$ UpdateLayerRepresentatives$(G, A)$;
9 return $G$;
```

7.2 Single Insertion

For the case of single insertion, we propose the insertion algorithm for our ATOMG index in Algorithm 7, consisting of three phases.

First, it determines the optimal insertion point by searching from the entry point to find both the nearest existing node and the best layer for insertion, followed by creating a new node with the input vector at this determined layer (lines 1-4). Then, to ensure proper connectivity, the algorithm establishes connections in a three-layer window (the best layer and its adjacent layers), where for each layer it identifies potential neighbors within a distance threshold and selects the limited closest nodes while maintaining maximum degree constraints through reciprocal updates (lines 7-9). Finally, the algorithm maintains graph quality by enforcing degree constraints when adding reverse connections, ensuring each affected node keeps only its closest neighbors if its degree exceeds the maximum allowed value deg_{max} (lines 10-13). The complete functions can be found in our technical report [49]. The overall time complexity is $O(\log |S| + N_l)$ where N_l represents the maximum layer size among the processed layers.

Algorithm 7: Insert a Single Vector into ATOMG

```
Input: New vector $v_{new}$, graph index $G$
Output: Updated graph index $G_{new}$
1 $e \leftarrow$ GetEntryPoint$(G)$;
2 $u_{nn}, l_{best} \leftarrow$ Search$(G, v_{new}, e)$;
3 $u_{new} \leftarrow$ CreateNode$(v_{new}, l_{best})$;
4 $V(G) \leftarrow V(G) \cup \{u_{new}\}$;
5 $l_{lower} \leftarrow \max(0, l_{best} - 1)$;
6 $l_{upper} \leftarrow \min(l_{best} + 1, L_{max} - 1)$;
7 for $l \in \{l_{lower}, l_{best}, l_{upper}\}$ do
8     $N_l(u_{new}) \leftarrow$ FindNeighbors$(G, u_{new}, l)$;
9     $N_l(u_{new}) \leftarrow$ SelectTop$(N_l(u_{new}), k_{max})$;
10    foreach $w \in N_l(u_{new})$ do
11        $N_l(w) \leftarrow N_l(w) \cup \{u_{new}\}$;
12        if $|N_l(w)| > k_{max}$ then
13            $N_l(w) \leftarrow$ SelectTop$(N_l(w), k_{max})$;
14 return $G$;
```

7.3 Batch Update

To more efficiently handle updates in real-world scenarios, we propose a batch update algorithm in Algorithm 8. The batch update algorithm for ATOMG handles multiple insertions and deletions through a lazy update mechanism that operates in three main phases. In the first phase, it processes deletion requests by marking nodes for removal (lines 1-5) and insertion requests by finding appropriate layers for new nodes (lines 6-10), accumulating these operations in a lazy update queue rather than executing them immediately. The second phase evaluates whether the graph rebalancing is needed based on structural criteria or if the lazy update queue has exceeded a threshold size (lines 11-12). This evaluation triggers either structural redistribution across layers or the execution of accumulated deferred updates (lines 13-19). The complete functions of the algorithm can be found in our technical report [49]. The overall time complexity is $O(B_{up} \log |S| + B_{up} \times N_l)$, where B_{up}

is the batch size of the update and N_l is the maximum layer size among the processed layers.

Algorithm 8: Batch Update for ATOMG

```
   Input: Insert set $V_i$, delete set $V_d$, graph $G$
   Output: Updated graph $G$
1  foreach $v \in V_d$ do
2      $u \leftarrow \text{Find}(G, v)$;
3      if $u \neq \emptyset$ then
4          $\mathcal{U}_{lazy} \leftarrow \mathcal{U}_{lazy} \cup \{(d, u)\}$;
5          $V(G) \leftarrow V(G) \setminus \{u\}$;
6  foreach $v \in V_i$ do
7      $e \leftarrow \text{Entry}(G)$;
8      $u, l \leftarrow \text{Search}(G, v, e)$;
9      $n \leftarrow \text{New}(v, l)$;
10     $\mathcal{U}_{lazy} \leftarrow \mathcal{U}_{lazy} \cup \{(i, n)\}$;
11 $needRebalance \leftarrow \text{CheckRebalance}(G)$;
12 $needLazyUpdate \leftarrow |\mathcal{U}_{lazy}| > \theta_{lazy}$;
13 if $needRebalance$ then
14     $G \leftarrow \text{Rebalance}(G, \mathcal{U}_{lazy})$;
15     $\mathcal{U}_{lazy} \leftarrow \emptyset$;
   else
17     if $needLazyUpdate$ then
18         $G \leftarrow \text{ApplyLazyUpdates}(G, \mathcal{U}_{lazy})$;
19         $\mathcal{U}_{lazy} \leftarrow \emptyset$;
20 return $G$;
```

8 EXPERIMENTS

In this section, we evaluate the effectiveness and efficiency of our proposed solutions. First, we introduce the datasets, the algorithms compared in this work, and the implementation details in Sec. 8.1. Then we demonstrate the construction evaluation and search performance in Sec. 8.2 and 8.3. We verify the efficiency of the proposed dynamic solutions in Sec. 8.4 and conclude the experimental results in 8.5. We validate the effectiveness of optimizations, conduct parameter sensitivity analysis, and present complete experimental results in our technical report [49].

8.1 Experimental Setups

Datasets. The experimental evaluation encompasses eight widely-adopted real-world datasets spanning multiple domains: We use eight real-world datasets which are widely-adopted by existing works. These datasets represent diverse domains: Msong [11] and Audio [3] for acoustic processing, UQ-V [43] for video analysis, Crawl [1], GloVe [40], and Enron [2] for textual data, and SIFT1M [24] and GIST1M [24] for image features. The summary of datasets is shown in Table 3, where # Base and # Query denote the size of the base dataset and the query dataset respectively. This table is sorted by # Base firstly, then they are sorted by Dimension.

Methods. Our empirical studies are conducted based on five state-of-the-art methods: HNSW [34], NSG [20], NSSG [19], HCNNG [36], and τ-MNG [39]. Our ATOMG index incorporates all optimizations outlined in this work, unless specified otherwise.

Table 3: Summary of Datasets

Dataset	# Base	# Query	Dimension	LID
Audio	53, 387	200	192	5.6
Enron	94, 987	200	1, 369	11.7
Msong	992, 272	200	420	9.5
SIFT1M	1, 000, 000	10, 000	128	9.3
UQ-V	1, 000, 000	10, 000	256	7.2
GIST1M	1, 000, 000	10, 000	960	18.9
GloVe	1, 183, 514	10, 000	100	20.0
Crawl	1, 989, 995	10, 000	300	15.7

Metric. Multiple evaluation metrics are employed to assess both the index construction efficiency and search performance, which follow this experimental survey [52]. For construction, we evaluate the construction time, the memory footprint and the index size. For search, we use Recall@k = $\frac{|RA| \cap |GT|}{k}$ to measure the search accuracy, where RA is the set of returned answer and GT is the set of ground truth. We the Queires Per Second (QPS) and the Speedup = $\frac{|S|}{\text{NDC}}$ to measure the search efficiency, where NDC is the Number of Distance Computation. We focus on the high-recall region of search performance evaluation for real-world scenarios.

Implementation. The algorithms are implemented in C++. All experiments are performed on a Linux cluster with eight machines. Each machine is equipped with an Intel Xeon CPU (24 cores) and 128GB memory.

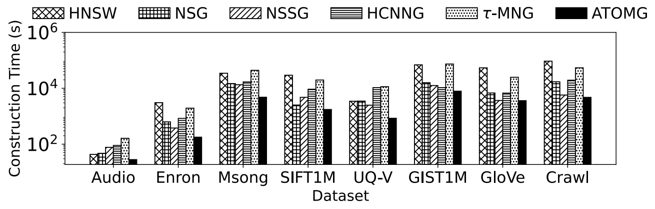
8.2 Construction Evaluation

Keeping the algorithms in their optimized versions, we compare the performance of the ATOMG on all datasets with all competitors.

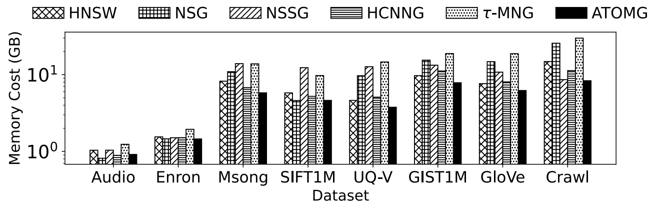
Construction Time. The construction time comparison in Figure 3 (a) reveals ATOMG's significant efficiency across different datasets. For SIFT1M, ATOMG completes construction in approximately 2×10^3 seconds, while HNSW requires 10× time cost. This performance gap is consistently maintained or widened for larger datasets. The improvement is particularly evident on Crawl, where ATOMG completes construction in around 5×10^3 seconds, significantly outperforming τ-MNG by 14× and HNSW by 18×. Even on smaller datasets like Audio and Enron, ATOMG maintains consistent performance advantages, though the absolute time differences are smaller due to the reduced data volume.

Memory Footprint. The memory footprint comparison in Figure 3 (b) demonstrates ATOMG's memory efficiency across different datasets. For SIFT1M, ATOMG requires around 5GB of memory, while competing methods like τ-MNG and NSSG consume around 10GB, representing a 2× reduction in memory usage. The largest dataset Crawl shows the most significant benefits, where ATOMG demonstrates around 3-3.5× improvement in memory efficiency than τ-MNG and NSG. ATOMG maintains consistent memory advantages over other indexes.

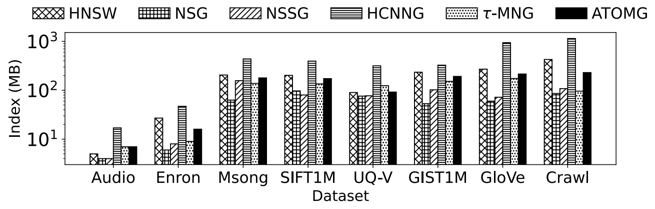
Index Size. The index size comparison in Figure 3 (c) shows ATOMG's effectiveness in maintaining compact index structures across datasets. For SIFT1M, ATOMG's index size is comparable to NSG and τ-MNG, while being significantly smaller than HCNNG. On larger datasets like GIST1M, ATOMG maintains an index size of about 200MB,



(a) Construction Time



(b) Memory Footprint



(c) Index Size

Figure 3: The construction evaluation

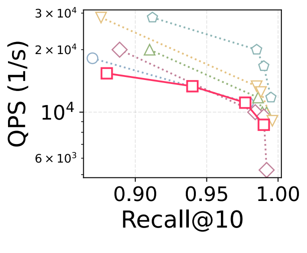
showing substantial improvement over HCNNG and HNSW. For smaller datasets like Audio and Enron, ATOMG maintains competitive or smaller index sizes compared to other methods.

8.3 Search Performance

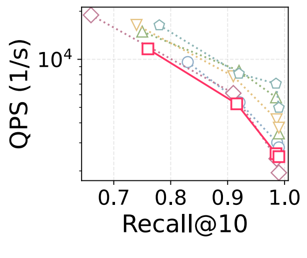
Following this experimental survey [52], we focus on $k = 10$ in the experiments, and only high precision region is considered here.

QPS vs Recall. In Figure 4, we show the QPS versus Recall curves on two datasets, others can be found in our report [49]. For smaller datasets like Audio, ATOMG maintains around 1.5×10^4 QPS at Recall 0.95, while for medium-sized datasets such as Enron and Msong, it achieves approximately 1.5-2 $\times 10^3$ QPS at high recall values (0.9 – 0.94). On larger datasets, ATOMG continues to show robust performance - SIFT1M maintains about 10^4 QPS at Recall 0.8, GIST1M achieves 5×10^3 QPS at Recall 0.95, and even the challenging Crawl dataset maintains 10^3 QPS at a high Recall of 0.97. While ATOMG doesn't consistently lead in raw search speed, its performance remains competitive across all datasets, typically falling within a factor of 1.2-1.5× of the best-performing methods. The QPS versus Recall curves across eight datasets demonstrate ATOMG's competitive search performance relative to existing methods.

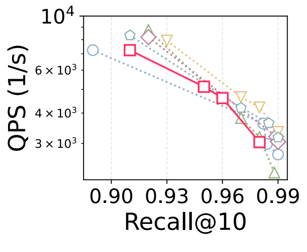
Speedup vs Recall. The Speedup versus Recall curves in Figure 5 demonstrate ATOMG's acceleration performance compared to exhaustive search across different datasets, complete results can be



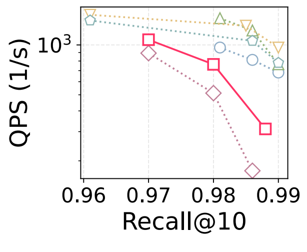
(a) Audio



(b) SIFT1M



(c) GIST1M



(d) Crawl

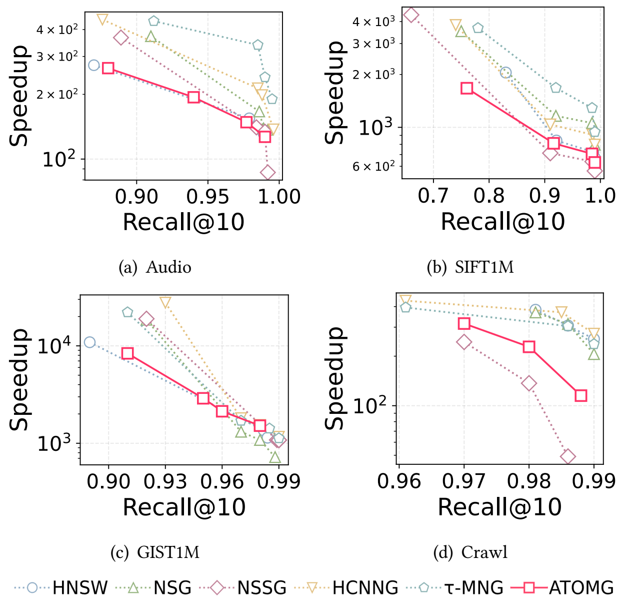
Figure 4: QPS vs Recall

refered to [49]. For the SIFT1M dataset, ATOMG achieves approximately $10^3\times$ speedup at Recall 0.9, though this is somewhat lower than other methods like NSG and HNSW which achieve around 20^3-$30^3\times$ speedup at the same recall level. On smaller datasets like Audio and Enron, ATOMG shows more modest speedups (200-500× at high recall values), but maintains stable performance across different recall levels. For the challenging Crawl dataset, ATOMG achieves around $200x$ speedup even at very high recall (0.97-0.98). While ATOMG generally shows lower absolute speedup numbers compared to other methods, it is important to note that these speedups are achieved while maintaining significantly better construction efficiency and memory usage, representing a practical trade-off between search acceleration and resource consumption.

8.4 Maintenance Evaluation

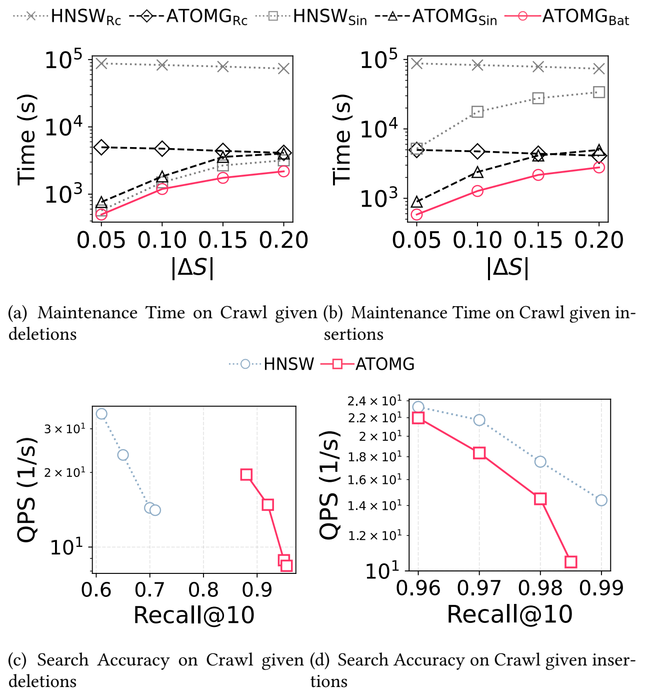
For simulating the real-world dynamic scenario, the dataset is divided into continuous segments, where the tail part serves as ΔS to be updated. We vary the size of ΔS as the proportion of $|S|$ to see how it affects the maintenance time and the search accuracy after updating. Note that for the ground truth file, if a vector is not visible before or after updating, then it would not be counted into the correct answer for fairness. The maintenance time comparison focuses on ATOMG and HNSW, as HNSW is the only inherently maintainable method among existing approaches. Different variants are tested, denoted by subscripts: Rc for re-construction, Sin for single-update method, and Bat for batch-update method.

Maintenance Time. The maintenance time analysis across different update scenarios reveals interesting patterns for ATOMG and HNSW variants. For the Crawl dataset, in deletion scenarios (Figure 6 (a)), ATOMG$_{Bat}$ maintains relatively stable performance around 1-2 $\times 10^3$ seconds across different $|\Delta S|$ values up to 0.20, while HNSW$_{Rc}$ requires consistently high maintenance time around



(a) Audio (b) SIFT1M (c) GIST1M (d) Crawl

Figure 5: Speedup vs Recall



(a) Maintenance Time on Crawl given deletions (b) Maintenance Time on Crawl given insertions

(c) Search Accuracy on Crawl given deletions (d) Search Accuracy on Crawl given insertions

Figure 6: Maintenance Evaluation

8×10^4 seconds. For insertions (Figure 6 (b)), the performance gap is even more larger, ATOMG$_{Bat}$'s maintenance time gradually increases from 5×10^2 to 3×10^3 seconds, while HNSW$_{Rc}$ maintains a high overhead of about 8×10^4 seconds, and HNSW$_{Sin}$ shows dramatic deterioration from 5×10^3 to 3×10^4 seconds as $|\Delta S|$ increases. Other results exhibit similar patterns. All variants of ATOMG (ATOMG$_{Rc}$, ATOMG$_{Sin}$, ATOMG$_{Bat}$) consistently outperform their HNSW counterparts, with ATOMG$_{Bat}$ showing the best overall maintenance efficiency across datasets and update types.

Search Accuracy. For evaluating the search accuracy fairly, we fix ΔS for two datasets respectively. For Msong, the ΔS is set to 0.15. For Crawl, the ΔS is set to 0.10. The search accuracy comparison between ATOMG and HNSW after updates shows distinct patterns for different scenarios. For the Crawl dataset, with deletions (Figure 6 (c)), HNSW starts at $\sim$ 30 QPS at recall 0.6 but degrades quickly, while ATOMG maintains $\sim$ 20 QPS up to recall 0.8, though dropping to $\sim$ 8 QPS at recall 0.9. For insertions (Figure 6 (d)), both methods show steady degradation in the high-recall region (0.96-0.99), with QPS decreasing from $\sim$ 24 to $\sim$ 10. Other results demonstrate similar patterns at different scales. These results suggest that while HNSW might perform better at lower recall levels, ATOMG provides more stable performance across different recall ranges, particularly maintaining reasonable QPS at higher recall values after updates.

8.5 Summary

The comprehensive experimental evaluation demonstrates ATOMG's balanced performance across multiple metrics. In construction efficiency, ATOMG shows a consistent 10× improvement over competitors, completing SIFT1M in 2×10^3 seconds compared to HNSW's 2×10^4 seconds. Memory footprint analysis reveals 2-3× reductions, using only 5GB for SIFT1M versus 10GB for competing methods, while maintaining comparable index sizes across diverse datasets. Search performance shows competitive QPS-recall tradeoffs, achieving 10^4 QPS at Recall 0.8 for SIFT1M, typically within a factor of 1.2-1.5× of the best performers. Most notably, in maintenance scenarios, ATOMG$_{Bat}$ significantly outperforms HNSW variants, requiring only 1-2 $\times 10^3$ seconds for updates where HNSW$_{Rc}$ needs 8×10^4 seconds, while maintaining stable search accuracy ($\sim 10^2$ QPS at Recall 0.9 for Msong). These results collectively establish ATOMG as a practical solution that effectively balances construction efficiency, memory usage, search performance, and maintenance capability.

9 CONCLUSION

In this work, we present ATOMG, a novel decomposed graph index that effectively addresses the scalability challenges in both static and dynamic scenarios. By introducing an adaptive layer decomposition strategy with uniform node partitioning and representative-based inter-layer navigation, ATOMG achieves substantial improvements in multiple aspects. In static scenarios, ATOMG reduces construction time by 10× and memory consumption by 2-3× while maintaining competitive search quality. More importantly, in dynamic scenarios, ATOMG supports efficient batch updates with a 20× speedup compared to existing methods, while preserving stable search performance. These results demonstrate that ATOMG successfully addresses the efficiency bottlenecks in index construction and dynamic maintenance, while delivering competitive search performance.

REFERENCES

[1] Anon. Common crawl. https://commoncrawl.org/overview [Online]. Accessed: 2024-07-09.

[2] Anon. Enron email dataset. https://www.cs.cmu.edu/ ./enron/ [Online]. Accessed: 2024-07-09.

[3] Anon. Timit audio. https://www.cs.princeton.edu/cass/demos.htm [Online]. Accessed: 2024-07-09.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching. In D. D. Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*, pages 573–582. ACM/SIAM, 1994.

[5] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[6] M. Aumüller, E. Bernhardsson, and A. J. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87, 2020.

[7] D. Baranchuk, A. Babenko, and Y. Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XII*, volume 11216 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2018.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[9] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.*, 33(6):623–630, June 2015.

[10] M. W. Bern. Approximate closest-point queries in high dimensions. *Inf. Process. Lett.*, 45(2):95–99, 1993.

[11] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and P. Lamere. The million song dataset. In A. Klapuri and C. Leider, editors, *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011, Miami, Florida, USA, October 24-28, 2011*, pages 591–596. University of Miami, 2011.

[12] M. Chen, K. Zhang, Z. He, Y. Jing, and X. S. Wang. Roargraph: A projected bipartite graph for efficient cross-modal approximate nearest neighbor search. *Proc. VLDB Endow.*, 17(11):2735–2749, 2024.

[13] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang. *SPTAG: A library for fast approximate nearest neighbor search*, 2018. https://github.com/Microsoft/SPTAG [Online]. Accessed: 2024-07-09.

[14] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 577–586. ACM, 2011.

[15] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, and J. Larson. From local to global: A graph RAG approach to query-focused summarization. *CoRR*, abs/2404.16130, 2024.

[16] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[17] S. Fortune. Voronoi diagrams and delaunay triangulations. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry, Second Edition*, pages 513–528. Chapman and Hall/CRC, 2004.

[18] C. Fu and D. Cai. EFANNA : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *CoRR*, abs/1609.07228, 2016.

[19] C. Fu, C. Wang, and D. Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(8):4139–4150, 2022.

[20] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, 2019.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[22] G. Þ. Gudmundsson, B. Þ. Jónsson, L. Amsaleg, and M. J. Franklin. Prototyping a web-scale multimedia retrieval service using spark. *ACM Trans. Multim. Comput. Commun. Appl.*, 14(3s):65:1–65:24, 2018.

[23] B. Harwood and T. Drummond. FANNG: fast approximate nearest neighbour graphs. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 5713–5722. IEEE Computer Society, 2016.

[24] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[25] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*, pages 861–864. IEEE, 2011.

[26] Z. Jing, Y. Su, Y. Han, B. Yuan, H. Xu, C. Liu, K. Chen, and M. Zhang. When large language models meet vector databases: A survey. *CoRR*, abs/2402.01763, 2024.

[27] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 2130–2137. IEEE Computer Society, 2009.

[28] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(5):869–883, 2009.

[29] C. Li, M. Zhang, D. G. Andersen, and Y. He. Improving approximate nearest neighbor search through learned adaptive early termination. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2539–2554. ACM, 2020.

[30] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement. *IEEE Trans. Knowl. Data Eng.*, 32(8):1475–1488, 2020.

[31] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, pages 825–832, 2004.

[32] W. Liu, H. Wang, Y. Zhang, W. Wang, L. Qin, and X. Lin. EI-LSH: an early-termination driven I/O efficient incremental c-approximate nearest neighbor search. *VLDB J.*, 30(2):215–235, 2021.

[33] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.*, 45:61–68, 2014.

[34] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[35] Y. Matsui, Y. Uchida, H. Jégou, and S. Satoh. A survey of product quantization. *ITE Transactions on Media Technology and Applications*, 6(1):2–10, 2018.

[36] J. A. V. Muñoz, M. A. Gonçalves, Z. Dias, and R. da Silva Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognit.*, 96, 2019.

[37] B. Naidan, L. Boytsov, and E. Nyberg. Permutation search methods are efficient, yet faster search is possible. *Proc. VLDB Endow.*, 8(12):1618–1629, 2015.

[38] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In C. Àlvarez and M. J. Serna, editors, *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, volume 4007 of *Lecture Notes in Computer Science*, pages 85–97. Springer, 2006.

[39] Y. Peng, B. Choi, T. N. Chan, J. Yang, and J. Xu. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proc. ACM Manag. Data*, 1(1):54:1–54:27, 2023.

[40] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[41] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007.

[42] L. C. Shimomura, R. S. Oyamada, M. R. Vieira, and D. S. Kaster. A survey on graph-based methods for similarity searches in metric spaces. *Inf. Syst.*, 95:101507, 2021.

[43] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong. Multiple feature hashing for real-time large scale near-duplicate video retrieval. In K. S. Candan, S. Panchanathan, B. Prabhakaran, H. Sundaram, W. Feng, and N. Sebe, editors, *Proceedings of the 19th International Conference on Multimedia 2011, Scottsdale, AZ, USA, November 28 - December 1, 2011*, pages 423–432. ACM, 2011.

[44] S. J. Subramanya, Devvrit, H. V. Simhadri, R. Krishnaswamy, and R. Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13748–13758, 2019.

[45] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proc. VLDB Endow.*, 8(1):1–12, 2014.

[46] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, 2013.

[47] B. Tang and H. He. ENN: extended nearest neighbor method for pattern recognition [research frontier]. *IEEE Comput. Intell. Mag.*, 10(3):52–60, 2015.

[48] A. Tayal and A. Tyagi. Dynamic contexts for generating suggestion questions in RAG based conversational systems. In T. Chua, C. Ngo, R. K. Lee, R. Kumar, and

H. W. Lauw, editors, *Companion Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, Singapore, May 13-17, 2024*, pages 1338–1341. ACM, 2024.

[49] A. Tian, L. Chen, M. Yuan, H. Li, A. Zhou, Y. Wang, and L. Chen. Atomg: Adaptively decomposed graph index for vector search [technical report]. https://github.com/ExpCodeBase/atom/blob/main/full.pdf [Online]. Accessed: 2024-11-20.

[50] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognit.*, 12(4):261–268, 1980.

[51] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X. Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(2):388–403, 2014.

[52] M. Wang, X. Xu, Q. Yue, and Y. Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, 2021.

[53] X. Wu, R. Guo, A. T. Suresh, S. Kumar, D. N. Holtmann-Rice, D. Simcha, and F. X. Yu. Multiscale quantization for fast similarity search. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5745–5755, 2017.

[54] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, P. Cheng, and M. Yang. Spfresh: Incremental in-place update for billion-scale vector search. In J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 545–561. ACM, 2023.

[55] X. Zeng, Z. Wu, X. Hu, X. Shi, S. Sun, and S. Zhang. CANDY: A benchmark for continuous approximate nearest neighbor search with dynamic data ingestion. *CoRR*, abs/2406.19651, 2024.

[56] Y. Zhang, K. Huang, G. Geng, and C. Liu. Fast knn graph construction with locality sensitive hashing. In H. Blockeel, K. Kersting, S. Nijssen, and F. Zelezný, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part II*, volume 8189 of *Lecture Notes in Computer Science*, pages 660–674. Springer, 2013.

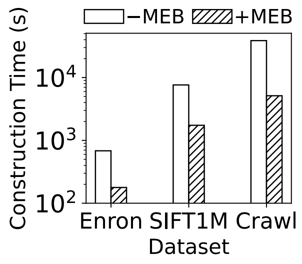
[57] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2023–2037. ACM, 2016.

APPENDICES

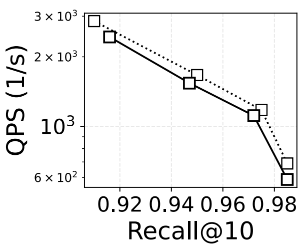
A EXPERIMENTS

A.1 Effectiveness of Optimizations

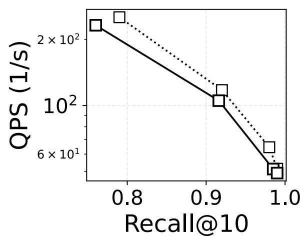
Maximum Edge Budget. In Figure 7 (a), MEB substantially reduces construction time across all datasets, approximately one order of magnitude reduction in construction time. In Figure 7 (b)-(d), the impact on query efficiency is minimal. For the Enron dataset, the QPS values remain nearly identical across different recall levels. The results on SIFT1M and Crawl datasets exhibit similar minor performance degradation. These results indicate that MEB successfully achieves significant construction time optimization while maintaining comparable search performance.



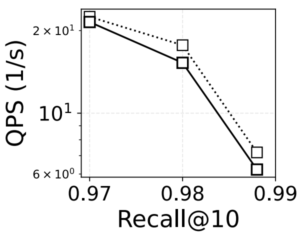
(a) Construction Time



(b) Search Performance on Enron



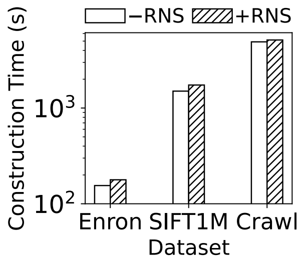
(c) Search Performance on SIFT1M



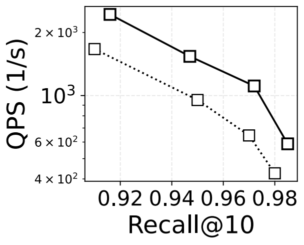
(d) Search Performance on Crawl

Figure 7: The effectiveness of MEB

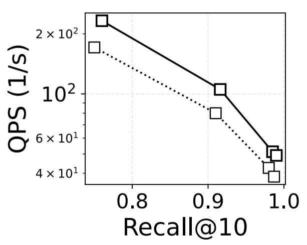
Representative Node Selection. We evaluate the effectiveness of Representative Node Selection (RNS) and we report the results of three representative datasets Enron, SIFT1M and Crawl, the results on other datasets show similar trends. In Figure 8 (a), while RNS increases construction time, the overhead is approximately 20-30% additional time across the tested datasets. However, this increased construction cost is offset by significant improvements in search performance. In Figure 8 (b)-(d), the comparison of search efficiency reveals that RNS consistently enhances query throughput (QPS) while maintaining recall accuracy. When examining the search performance at high recall values (Recall@10 around 0.98), both Enron and SIFT1M datasets show approximately 1.2-1.7× improvement in QPS with RNS enabled. The enhancement is more pronounced for Crawl, where RNS achieves a 2.6× increase in QPS. The results on other datasets show similar trends, which indicate that RNS introduces minimal construction overhead while delivering significant and sustained improvements in search efficiency.



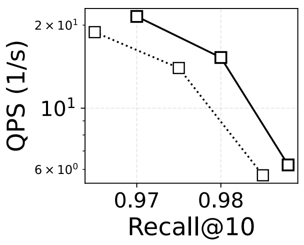
(a) Construction Time



(b) Search Performance on Enron



(c) Search Performance on SIFT1M



(d) Search Performance on Crawl

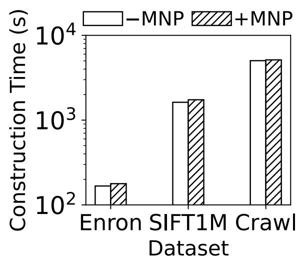
Figure 8: The effectiveness of RNS

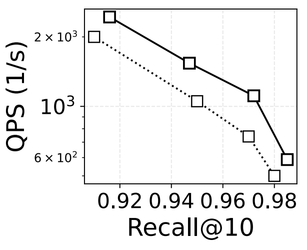
Margin Node Preservation. In Figure 9 (a), MNP has negligible impact on construction time across all tested datasets. However, the search performance improvements are substantial, as illustrated in Figure 9 (b)-(d). For the Enron dataset, MNP achieves approximately 2× higher QPS at high recall around 0.98. Similar improvements are observed in SIFT1M. The Crawl dataset shows the most impressive enhancement, with MNP achieving nearly 3× better QPS at high recall around 0.98, the results on other datasets show similar trends. These results indicate that MNP is a highly effective optimization that significantly improves search efficiency without introducing additional construction overhead.

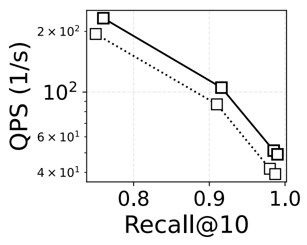
A.2 Parameter Sensitivity Analysis

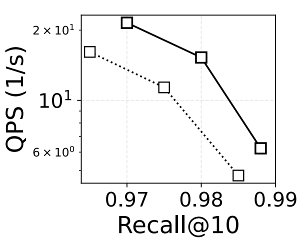
We conduct the parameter sensitivity analysis and we report the results of construction time on three representative datasets Enron, SIFT1M and Crawl. We report the search performance result on SIFT1M only. The results on other datasets show similar trends.

Varying number of layers $\mathcal{L}$. The impact of varying the number of layers ($\mathcal{L}$) reveals important trade-offs in ATOMG's performance characteristics. The construction time analysis in Figure 10 (a) shows substantial increases across all datasets as $\mathcal{L}$ grows. For Enron, raising $\mathcal{L}$ from 10^2 to 10^4 increases construction time from approximately 100 seconds to 2×10^3 seconds. SIFT1M and Crawl exhibit similar scaling behavior, with construction time rising from 10^3 seconds to 10^4 seconds. The search performance impact on SIFT1M, shown in Figure 10 (b), demonstrates a clear degradation as $\mathcal{L}$ increases. At Recall 0.8, QPS diminishes from 200 to 40 as $\mathcal{L}$ grows from 10^2 to 10^4. These results suggest that while larger $\mathcal{L}$ values provide finer index granularity, they introduce significant overhead in both construction and search phases.


(a) Construction Time


(b) Search Performance on Enron


(c) Search Performance on SIFT1M


(d) Search Performance on Crawl

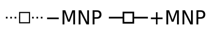
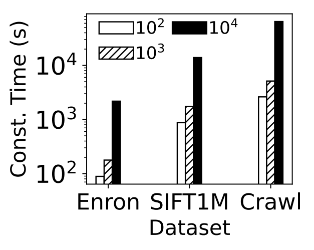
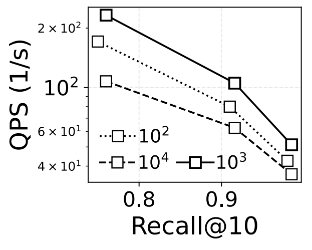
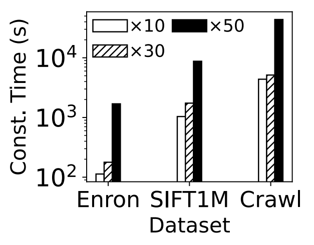


Figure 9: The effectiveness of MNP


(a) Construction Time, varying $\mathcal{L}$


(b) Search Performance, varying $\mathcal{L}$


(c) Construction Time, varying $\mathcal{B}$

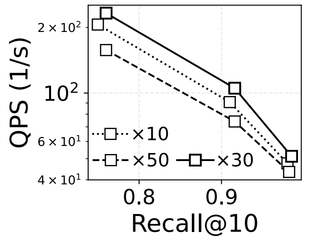

(d) Search Performance, varying $\mathcal{B}$

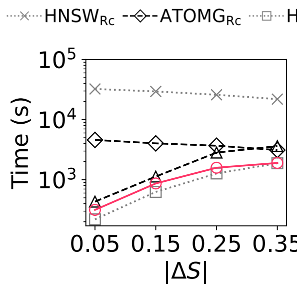
Figure 10: Parameter Sensitivity Analysis

Varying the edge budget $\mathcal{B}$. The impact of varying the edge budget ($\mathcal{B}$) shows significant effects on ATOMG's performance characteristics. Figure 10 (c) demonstrates how construction time scales with different $\mathcal{B}$ values (×10, ×30, and ×50) across datasets. For Enron, increasing $\mathcal{B}$ from ×10 to ×50 causes construction time to rise from roughly 100 seconds to 1,500 seconds. SIFT1M and Crawl follow the similar scaling pattern. The search performance impact on SIFT1M is illustrated in Figure 10 (d). At Recall 0.8, QPS decreases from about 200 to 150 as $\mathcal{B}$ increases from ×10 to ×50. This performance degradation becomes more noticeable at higher recall levels, with Recall 0.9 showing QPS reduction from approximately 100 to 50. These results suggest that while larger edge budgets significantly increase construction overhead, they have a more modest impact on search performance, indicating that moderate $\mathcal{B}$ values (around ×30) strike the balance between construction costs and query efficiency.

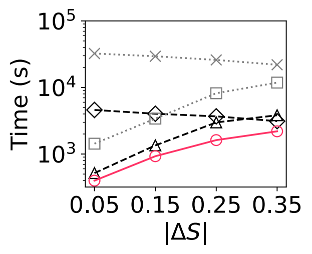
A.3 Maintenance Evaluation

Maintenance Time. The maintenance time analysis across different update scenarios reveals interesting patterns for ATOMG and HNSW variants. For the Msong dataset, Figure 11 (a) shows that with deletions, ATOMG$_{Bat}$ maintains the best performance, requiring only about 1-2×10^3 seconds even as $|\Delta S|$ increases to 0.35, while HNSW$_{Rc}$ consistently requires around 2-3 × 10^4 seconds. For insertions (Figure 11 (b)), ATOMG$_{Bat}$ maintains its efficiency advantage, since HNSW$_{Sin}$ shows increasing maintenance time from 10^3 to 10^4 seconds as $|\Delta S|$ grows. The Crawl dataset exhibits similar patterns but at larger scales. In deletion scenarios (Figure 11 (c)), ATOMG$_{Bat}$ maintains relatively stable performance around 1-2 × 10^3 seconds across different $|\Delta S|$ values up to 0.20, while HNSW$_{Rc}$ requires consistently high maintenance time around 8×10^4 seconds. For insertions (Figure 11 (d)), the performance gap is even more larger, ATOMG$_{Bat}$'s maintenance time gradually increases from 5×10^2 to 3×10^3 seconds, while HNSW$_{Rc}$ maintains a high overhead of about 8×10^4 seconds, and HNSW$_{Sin}$ shows dramatic deterioration from 5×10^3 to 3×10^4 seconds as $|\Delta S|$ increases. All variants of ATOMG (ATOMG$_{Rc}$, ATOMG$_{Sin}$, ATOMG$_{Bat}$) consistently outperform their HNSW counterparts, with ATOMG$_{Bat}$ showing the best overall maintenance efficiency across both datasets and operation types.

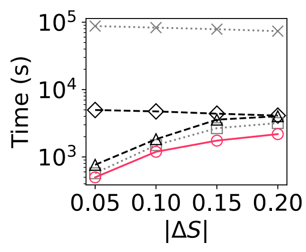
Search Accuracy. For evaluating the search accuracy fairly, we fix ΔS for two datasets respectively. For Msong, the ΔS is set to 0.15. For Crawl, the ΔS is set to 0.10. The search accuracy comparison between ATOMG and HNSW after updates shows distinct patterns for different scenarios. On the Msong dataset, for deletions (Figure 11 (e)), HNSW achieves higher QPS ($\sim 2 \times 10^2$) at lower recall levels (0.6-0.7), but its performance drops sharply. ATOMG maintains more stable performance, achieving $\sim 10^2$ QPS even at higher recall (0.9). For insertions (Figure 11 (f)), both methods show more comparable performance in the high-recall region (0.96-0.99), with QPS gradually decreasing from 2.2×10^3 to 1.5×10^3. The Crawl dataset demonstrates similar patterns at different scales. With deletions (Figure 11 (g)), HNSW starts at $\sim 3 \times 10^1$ QPS at recall 0.6 but degrades quickly, while ATOMG maintains $\sim 2 \times 10^1$ QPS up to recall 0.8, though dropping to ~ 8 QPS at recall 0.9. For insertions (Figure 11 (h)), both methods show steady degradation in the high-recall region (0.96-0.99), with QPS decreasing from $\sim 2.4 \times 10^1$ to $\sim 1.0 \times 10^1$. These results suggest that while HNSW might perform better at lower recall levels, ATOMG provides more stable performance across different recall ranges, particularly maintaining reasonable QPS at higher recall values after updates.



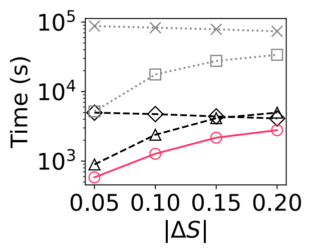
(a) Maintenance Time on Msong given deletions



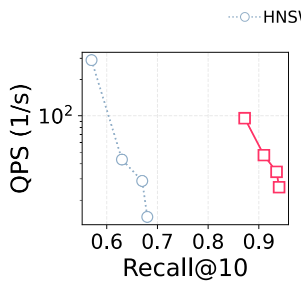
(b) Maintenance Time on Msong given insertions



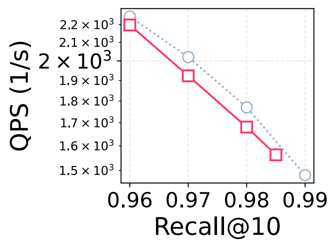
(c) Maintenance Time on Crawl given deletions



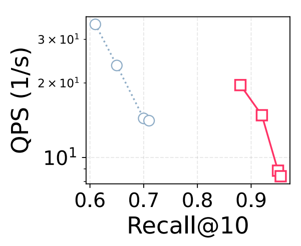
(d) Maintenance Time on Crawl given insertions



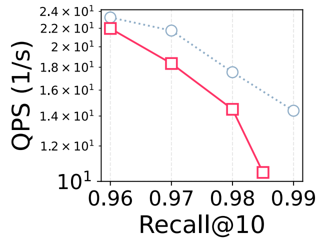
(e) Search Accuracy on Msong given deletions



(f) Search Accuracy on Msong given insertions

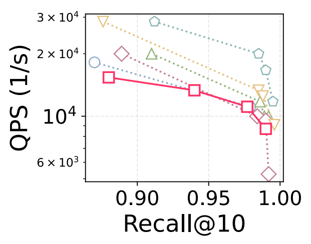


(g) Search Accuracy on Crawl given deletions

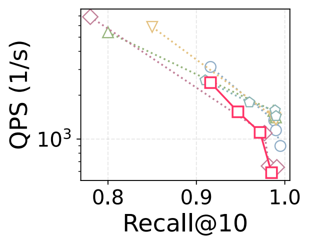


(h) Search Accuracy on Crawl given insertions

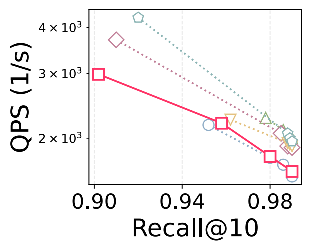
Figure 11: Maintenance Evaluation



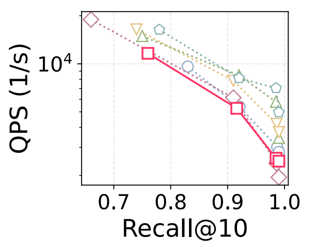
(a) Audio



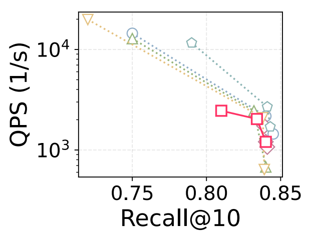
(b) Enron



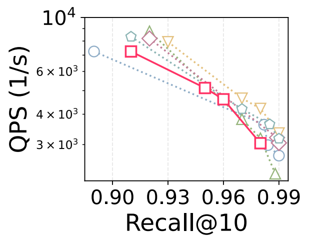
(c) Msong



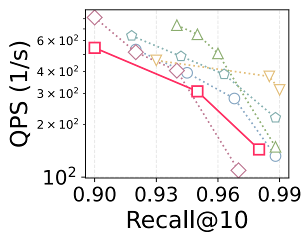
(d) SIFT1M



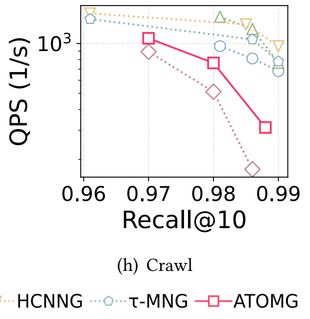
(e) UQ-V



(f) GIST1M



(g) Glove-100



(h) Crawl

Figure 12: QPS vs Recall

A.4 Search Performance

The complete results on search performance is shown in Figure 12 and 13.

B PROOFS

B.1 Proof of Proposition 5.1

Proof. Suppose that the maximum degree in the graph index is deg_{max}, and the number of layers is $\mathcal{L}$. The size of the layer guidance is bounded by $O(deg_{max} \cdot \mathcal{L})$. The size of each layer is bounded by $O(deg_{max} \cdot \frac{|S|}{\mathcal{L}})$. Therefore, the overall index size is bounded by $O(deg_{max} \cdot \mathcal{L} \cdot \frac{|S|}{\mathcal{L}}) = O(deg_{max} \cdot |S|)$. □

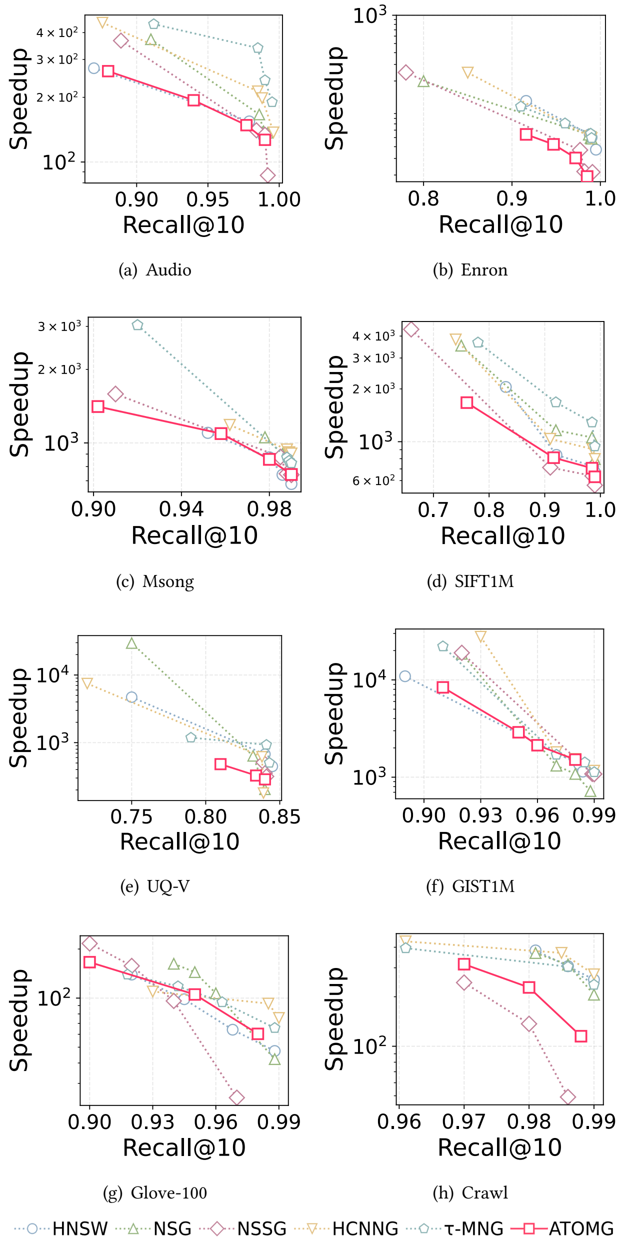


Figure 13: Speedup vs Recall

B.2 Proof of Proposition 5.2

Proof. For Algorithm 1, the search time of the guidance procedure is bounded by $O(\log \mathcal{L})$, the time cost in the target layer is bounded by $O(\log \frac{|S|}{\mathcal{L}})$. Thus, the time complexity of Algorithm 1 is $O(\log \mathcal{L} + \log \frac{|S|}{\mathcal{L}}) = O(\log(\mathcal{L} \cdot \frac{|S|}{\mathcal{L}})) = O(\log |S|)$ in total. □

B.3 Proof of Lemma 6.1

Proof. For the Layerization algorithm reveals its computational efficiency across different stages. The tail threshold determination process takes $O(sampleSize \cdot \log(sampleSize))$ time due to the sorting operation before the ECDF analysis, where $sampleSize$ is much smaller than $|S|$. The layer assignment process examines each node once, contributing a linear $O(|S|)$ complexity. The algorithm's overall time complexity is $O(|S|)$. In terms of space requirements, the algorithm maintains several key data structures: lists for tail and normal elements, the layer structure itself, and temporary storage for sampling, leading to a linear space complexity of $O(|S|)$. □

B.4 Proof of Theorem 6.2

Proof. First, let's formulate the decision version of OLGCP: Given a set of representative nodes R, an entry point e, and two parameters k and L, determine if there exists a connection strategy such that (1) the out-degree of e is no more than k, and (2) the expected path length from e to all nodes in R is no more than L.

We prove this is NP-hard through reduction from the Dominating Set Problem (DSP) [21]. Given a DSP instance $G = (V, E)$ and parameter k, we construct an instance of OLGCP as follows: let the representative nodes set $R = V$; create an entry point e; set path length threshold $L = 2$; keep the same degree constraint k; for any two nodes in R, they are connected *iff* they are adjacent in G.

We now prove that DSP has a dominating set of size $\leq k$ if and only if OLGCP has a valid solution:

($\Rightarrow$) If there exists a dominating set S in G where $|S| \leq k$, connecting e to all nodes in S gives a valid OLGCP solution because:

• The out-degree of e is $|S| \leq k$; • For any node $v \in R$, either $v \in S$ (path length = 1) or v is adjacent to some node in S (path length = 2), making the expected path length $\leq 2 = L$.

($\Leftarrow$) If OLGCP has a solution with out-degree $\leq k$ and expected path length ≤ 2, let S be the set of nodes that e connects to. Then: • $|S| \leq k$ by the degree constraint; • S must be a dominating set in G, otherwise there would exist some node with path length > 2, violating the path length constraint.

The reduction can be constructed in polynomial time $O(|V|+|E|)$. Since DSP is NP-complete [21], OLGCP is NP-hard. □

B.5 Proof of Lemma 6.3

Proof. First, the initial graph construction requires $O(|\mathcal{L}|^2)$ operations. As the optimization progresses, the number of nodes requiring examination reduces to approximately $O(|\mathcal{L}|^{0.5})$. The candidate set for each node can be reduced to $O(\log |\mathcal{L}|)$. The entry point optimization phase also benefits from a limited candidate set of around $O(\log |\mathcal{L}|)$ nodes, with most improvements occurring in the first few iterations. Furthermore, since query points are sampled near their target nodes, the actual path lengths and $|Q|$ in cost evaluation can be regarded as constant. Taking all these factors into account, the initial graph construction requires $O(|\mathcal{L}|^2)$ operations, while the optimization phases contribute $O(|\mathcal{L}|^{0.5} \log^3 |\mathcal{L}|)$, resulting in a total average time complexity of $O(|\mathcal{L}|^2)$. It requires $O(\mathcal{L}^2)$ space for the connection information and temporary storage. □

B.6 Proof of Theorem 6.4

Proof. First, let's formulate the decision version of OILCP: Given a representative node r, a set of nodes V, and parameters k, L, and edge budget $\mathcal{B}$, determine if there exists a connection strategy such that: the out-degree of r is $\leq k$; the expected path length from r to any node in V is $\leq L$; the total number of edges used is $\leq \mathcal{B}$.

Given an instance of Set Cover Problem [21] with universe $U = u_1, ..., u_n$, collection of subsets $\mathcal{S} = S_1, ..., S_m$, and parameter k, we construct an instance of OILCP as follows: create representative node r; for each element $u_i \in U$, create a target node v_i; for each subset $S_j \in \mathcal{S}$, create an intermediate node s_j; set $L = 2$ (path length threshold); set edge budget $\mathcal{B} = k + |U|$; keep the same degree constraint k; allow edges between: 1) r and any intermediate node s_j, 2)Each s_j and v_i if $u_i \in S_j$.

We prove that Set Cover has a solution of size $\leq k$ if and only if OILCP has a valid solution:

($\Rightarrow$) If there exists a set cover $C \subseteq \mathcal{S}$ with $|C| \leq k$: • Connect r to all intermediate nodes corresponding to sets in C; • Connect these intermediate nodes to their covered target nodes; • This gives: 1) Out-degree of r is $|C| \leq k$, 2) Every target node is reachable in exactly 2 hops, 3) Total edges used = $|C| + |U| \leq k + |U| = \mathcal{B}$.

($\Leftarrow$) If OILCP has a valid solution: • Let S' be the set of intermediate nodes that r connects to; • $|S'| \leq k$ by degree constraint; • The corresponding sets in $\mathcal{S}$ must cover all elements in U; • Otherwise, some target nodes would require > 2 hops; • Thus, S' corresponds to a valid set cover of size $\leq k$.

The reduction can be constructed in polynomial time $O(|U| + |\mathcal{S}| + \sum_{S \in \mathcal{S}} |S|)$. Since Set Cover is NP-complete [21], OILCP is NP-hard. □

B.7 Proof of Lemma 6.5

Proof. The time complexity of our algorithm can be analyzed by examining its four main phases. In the initialization phase, computing shortest paths requires $O(\frac{|S|}{\mathcal{L}} \log \frac{|S|}{\mathcal{L}})$ time as each node performs a logarithmic search operation. The algorithm then processes edge frequencies and critical paths in $O(\frac{|S|}{\mathcal{L}} \cdot deg_{max})$ time, where deg_{max} represents the maximum degree of any node in the graph. The edge evaluation phase forms the computational bottleneck, requiring $O(\frac{|S|}{\mathcal{L}} deg_{max}^2 \log \frac{|S|}{\mathcal{L}})$ operations since it needs to evaluate all potential edges within 2-hop distance - each node can have up to deg_{max} first-hop neighbors, and each of these neighbors can have up to deg_{max} second-hop neighbors. The edge optimization phase processes at most $\mathcal{B}$ edges with priority queue operations, contributing $O(\mathcal{B} deg_{max} \log \frac{|S|}{\mathcal{L}})$ to the complexity. Finally, the representative node optimization phase runs in $O(deg_{max}^2 \log \frac{|S|}{\mathcal{L}})$ time. The overall time complexity becomes $O(\frac{|S|}{\mathcal{L}} deg_{max}^2 \log \frac{|S|}{\mathcal{L}})$, as this term dominates all others given that $\mathcal{B}$ is typically $O(\frac{|S|}{\mathcal{L}} \cdot deg_{max})$, where deg_{max} are set to be small constant. Thus, the complexity can be simplified to $O(\frac{|S|}{\mathcal{L}} \log \frac{|S|}{\mathcal{L}})$. For space complexity, the graph structure itself needs $O(\frac{|S|}{\mathcal{L}} \cdot deg_{max})$ space to store all edges. The priority queue can grow up to $O(\frac{|S|}{\mathcal{L}} \cdot deg_{max}^2)$ in size. Therefore, the total space complexity is $O(\frac{|S|}{\mathcal{L}} \cdot deg_{max}^2)$. □

B.8 Proof of Theorem 6.6

Proof. The dominant cost of Algorithm 2 consists of the construction of base graph and layer-related connnection, which takes $O(|S|^{1.14})$ and $O(\frac{|S|}{\mathcal{L}} \cdot \log \frac{|S|}{\mathcal{L}} \cdot \mathcal{L}^2) = |S| \cdot \log \frac{|S|}{\mathcal{L}}$, where $\mathcal{L}$ is a constant. Thus the overall time complexity is $O(|S| \cdot \log |S| + |S|^{1.14})$. □

C FUNCTION LIST FOR ALGORITHMS

C.1 Layer Guidance Connection

Algorithm 9: Functions for Algorithm 4

```
Function EvaluateCost($G$, $v$):
    $Q$ ← Generate sample queries near $v$;
    $cost$ ← 0;
    foreach $q \in Q$ do
        $P$ ← Search path in $G$;
        $path\_cost \leftarrow \sum_{x \in P} |N(x)|$;
        $cost \leftarrow cost + path\_cost$;
    return $cost/|Q|$;
```

C.2 Intra-Layer Connection

Algorithm 10: Functions for Algorithm 5

```
Function EvaluateEdge($u$, $v$, $f$, $d$):
    $usage \leftarrow$ Norm$(f[(u, v)])$;
    $\Delta d \leftarrow$ Norm$(d(r, v) - d(r, u))$;
    return $\alpha \cdot usage + (1 - \alpha) \cdot \Delta d$;
Function IsValidAddition($G'$, $u$, $v$, $\mathcal{B}$):
    if $|N_{G'}(u)| \geq \mathcal{B}$ or $|N_{G'}(v)| \geq \mathcal{B}$ then
        return false;
    if $(u, v) \in E(G')$ then
        return false;
    $E_{old} \leftarrow E(G')$;
    $E(G') \leftarrow E(G') \cup \{(u, v)\}$;
    $cost_{new} \leftarrow$ EvaluateCost($G'$, $v$);
    $E(G') \leftarrow E_{old}$;
    return $cost_{new}$ does not increase;
```

C.3 Single Deletion

Algorithm 11: Functions for Algorithm 6

```
Function RestoreConnectivity($G, A$):
    foreach $u \in A$ do
        if $|N_l(u)| < k_{min}$ then
            $N_{new} \leftarrow$ FindNewNeighbors$(G, u, l)$;
            $N_l(u) \leftarrow N_l(u) \cup N_{new}$;
    return $G$;
Function UpdateLayerRepresentatives($G, A$):
    foreach $u \in A$ do
        if ShouldBeRepresentative$(G, u, l)$ then
            $R_l(G) \leftarrow R_l(G) \cup \{u\}$;
            if $l < L_{max} - 1$ then
                $R_{up} \leftarrow R_{l+1}(G)$;
                $N_{l+1}(u) \leftarrow$ SelectTop$(R_{up}, k_{max})$;
                foreach $r \in N_{l+1}(u)$ do
                    $N_l(r) \leftarrow N_l(r) \cup \{u\}$;
        else
            break;
    return $G$;
```

C.4 Single Insertion

Algorithm 12: Functions for Algorithm 7

```
Function CreateNode($v, l$):
    $n.v \leftarrow v$;
    $n.l \leftarrow l$;
    $n.N \leftarrow \emptyset$;
    return $n$;
Function FindNeighbors($G, n, l$):
    $N \leftarrow \emptyset$;
    $C \leftarrow G.V_l$;
    foreach $c \in C$ do
        if $\delta(n.v, c.v) \leq \theta_d$ then
            $N \leftarrow N \cup \{c\}$;
    return $N$;
Function SelectTop($V, k$):
    $V_{sort} \leftarrow \text{Sort}(V)$;
    $V_k \leftarrow V_{sort}[1:k]$;
    return $V_k$;
```

C.5 Batch Update

Algorithm 13: Functions for Algorithm 8

```
Function ApplyLazyUpdates($G, \mathcal{U}_{lazy}$):
    foreach $(op, v) \in \mathcal{U}_{lazy}$ do
        if $op = d$ then
            $G \leftarrow \text{DeleteNode}(G, v)$;
        else
            $G \leftarrow \text{InsertNode}(G, v)$;
    return $G$;
Function Rebalance($G, \mathcal{U}_{lazy}$):
    $\{n_l\} \leftarrow \text{Count}(G)$;
    $L_{unbal} \leftarrow \{l | n_l / n_{l+1} > \theta_b\}$;
    $L_{affected} \leftarrow$ The union of connected layers with $L_{unbal}$;
    $V_{affect} \leftarrow \{v \in V(G) | v.l \in L_{affected}\}$;
    $V_{lazy} \leftarrow \{v | (i, v) \in \mathcal{U}_{lazy}\} \setminus \{v | (d, v) \in \mathcal{U}_{lazy}\}$;
    $V_{rebal} \leftarrow V_{affect} \cup V_{lazy}$;
    $V_{sort} \leftarrow \text{Sort}(V_{rebal})$;
    $G \leftarrow \text{AdjustGraph}(G, V_{sort}, L_{affected})$;
    $G \leftarrow \text{UpdateLayerRepresentatives}(G, L_{affected})$;
    return ApplyLazyUpdates($G, \mathcal{U}_{lazy}$);
Function AdjustGraph($G, V, L_{affected}$):
    $N \leftarrow |V|$;
    $S \leftarrow$ empty array of size $|L_{affected}|$;
    foreach $l \in L_{affected}$ do
        $S[l] \leftarrow \lfloor N(1 - e^{-1/\alpha}) e^{-l/\alpha} \rfloor$;
    $S[min(L_{affected})] \leftarrow S[min(L_{affected})] + (N - \sum S)$;
    $L_{new} \leftarrow \text{Distribute}(V, S)$;
    foreach $v \in V$ do
        $v.l \leftarrow L_{new}[v]$;
        $v.N \leftarrow \emptyset$;
    foreach $v \in V$ do
        for $l \leftarrow 0$ to $v.l$ do
            $N_l \leftarrow \text{FindNeighbors}(G, v, l)$;
            foreach $u \in N_l$ do
                $v.N[l] \leftarrow v.N[l] \cup \{u\}$;
                $u.N[l] \leftarrow u.N[l] \cup \{v\}$;
    return $G$;
Function CheckRebalance($G$):
    if $|V(G)| < \theta_m \cdot L_m$ then
        return false;
    $\{n_l\} \leftarrow \text{Count}(G)$;
    return $\max(\{n_l\}) / \min(\{n_l\}) > \theta_b$;
Function UpdateLayerRepresentatives($G, L_{affected}$):
    foreach $l \in L_{affected}$ do
        $V_l \leftarrow \{v \in V(G) | v.l = l\}$;
        $G.rep[l] \leftarrow \text{SelectRepresentative}(V_l)$;
    return $G$;
```