Timestamp Approximate Nearest Neighbor Search over High-Dimensional Vector Data

Yuxiang Wang†, Ziyuan He†, Yongxin Tong†, Zimu Zhou‡, Yiman Zhong†
† State Key Laboratory of Complex and Critical Software Environment,
Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Beihang University, Beijing, China
‡ City University of Hong Kong, Hong Kong, China
†{yuxiangwang, hzy_he, yxtong, yeeman}@buaa.edu.cn, ‡zimuzhou@cityu.edu.hk

***Abstract*—Unstructured data, such as images and texts, are increasingly represented as high-dimensional vectors for emerging AI applications like retrieval-augmented generation. A key operation in these applications is querying for vectors that are both semantically similar and temporally relevant. This operation can be formulated as Timestamp Approximate Nearest Neighbor Search (TANNS), where both the vectors and the query incorporate temporal attributes, aiming to retrieve the approximate nearest neighbors valid at the given timestamp. A naive solution is to create separate indexes for each timestamp, which enables accurate and fast searches but incurs high update latency and excessive storage demands.**

In this paper, we introduce the timestamp graph, a novel structure that supports rapid index updates while minimizing storage costs. Exploiting the temporal locality of changes in valid vectors, our timestamp graph effectively manages a unified index across all historical timestamps, thereby substantially reducing storage overhead. Moreover, we design the historic neighbor tree, which further compresses the space complexity to that of a single-timestamp index. Extensive evaluations on four standard datasets show that our method achieves over 99% accuracy while improving the query efficiency by 4.4× to 138.1× than existing solutions.

***Index Terms*—high-dimensional data, approximate nearest neighbor search**

I. Introduction

Vector databases, which store and manage embeddings of diverse unstructured data (*e.g.*, text, images, audio) as high-dimensional vectors [1], [2], have become indispensable in emerging AI workflows such as retrieval-augmented generation (RAG) [3]. Acting as external knowledge repositories, they provide AI models with contextually relevant information to mitigate hallucinations and enhance in-context learning. Such information is efficiently retrieved via Approximate Nearest Neighbor Search (ANNS), an operation that returns semantically similar records for a given query vector.

Beyond semantic similarity, *temporal relevance* is also critical for practical vector database applications. On the one hand, the *vectors* often have a *validity* period, as new embeddings are continuously added to the database and outdated ones expire. For instance, in a knowledge base, the embeddings of current knowledge evolve over time [4], necessitating searching over valid data only to return accurate results. On the other hand, many *queries* are tied to specific *timestamps*, demanding answers that correspond to a particular moment. This is common in applications like financial analysis and fact verification, where aligning the retrieved vectors with the queried time reference is necessary to avoid misleading results [5]. We illustrate this insight in the following motivating example:

Example 1 (Answering Time-relevant Questions): Although large language models (LLMs) exhibit strong capabilities in question answering, they often struggle with time-relevant problems [6], [7], such as "What was the highest closing value of the NASDAQ-100 index before October 2024?" and "What was the prize pool for the Counter-Strike Major Championship in 2024 Summer?" Retrieval-Augmented Generation (RAG) can enhance the accuracy of answers by supplementing the LLM with text chunks that are both relevant to the query and temporally accurate [3], [8]. Since the retrieval step in RAG typically uses embedding similarity to identify relevant chunks [9], it is important to consider approximate nearest neighbor search of embedded data with timestamp constraint.

To meet these needs, we introduce Timestamp Approximate Nearest Neighbor Search (TANNS), a new query that incorporates *time attributes* into both the data vectors and the query. Given a query vector paired with a timestamp, the TANNS query returns approximate nearest neighbors among all vectors valid at that timestamp. Our goal is to enable *accurate* and *efficient* TANNS queries, as applications like chat bots demand real-time, high-quality results over millions of high-dimensional vectors [9]–[11].

Prior research fails to support efficient TANNS queries. Early studies on nearest neighbor search with temporal constraints [12]–[15] build temporal indexes to track valid objects per timestamp for rapid search. However, they struggle for the dense vector embeddings due to the curse of dimensionality [16]–[18]. A seemingly plausible solution is the recently proposed hybrid search upon high-dimensional vectors associated with attributes [19]–[21]. However, they are unfit for TANNS because they implicitly assume attributes with predetermined values, whereas the expiration timestamp of a vector is typically unknown when it is added to the database.

To handle the high-dimensional vectors with nondeterministic validity, we employ graph-based indexes, which prevail in ANNS over high-dimensional data [22], [23], and build a separate index per historical timestamp of interest. While this strategy enables accurate and fast searches, it incurs notable update latency due to frequent index updates whenever the validity of any vector changes. More critically, maintaining

TABLE I: Time and space complexity of methods (N: vector number in the dataset; M: neighbor number in the graph index).

	Searching Time	Updating Time	Space
Naive Graph-based TANNS (Sec. III-B)	$O(M \log N)$	$O(MN \log N)$	$O(MN^2)$
TANNS with Timestamp Graph (Sec. IV)	$O(\log^2 N)$	$O(\log^2 N)$	$O(M^2N)$
TANNS with Compressed Timestamp Graph (Sec. V)	$O(\log^2 N)$	$O(\log^2 N)$	$O(MN)$

an index for all valid vectors per timestamp consumes unacceptably high storage, with a space complexity that grows quadratically with the number of vectors (see Sec. III).

To enhance the efficiency of TANNS queries, we propose the *timestamp graph* (see Sec. IV), a novel structure that supports rapid index updates and requires low index storage. It maintains a *single* graph index for *all* historical timestamps, thus significantly reducing storage overhead. The feasibility of this approach is underpinned by the high *temporal locality* of the changes in valid vectors, resulting in considerable overlap in their indices across time. The cost of index updates is also reduced, as only the neighbors of changed vectors need modification, which constitute a small fraction of the graph. However, directly implementing this idea is challenging, as vector expiration can decrease graph connectivity and compromise search accuracy. To address this issue, we introduce *backup neighbors* and develop efficient algorithms to manage point insertion and expiration in the timestamp graph. Recognizing the redundancy in storing neighbor information, we further propose the *historic neighbor tree* (see Sec. V) to compress the timestamp graph. Remarkably, the compressed timestamp graph tracks valid vectors across all timestamps while achieving the *same space complexity* as a *single-timestamp* index. Table I summarizes the complexity of our proposed methods.

Our major contributions are as follows:

- We investigate the timestamp approximate nearest neighbor search (TANNS), a new query in vector databases for emerging AI applications.
- We propose the timestamp graph, a novel structure to manage valid vectors across timestamps by a single index. It enables accurate and efficient TANNS queries with fast index updating and low index storage overhead.
- We design the historic neighbor tree, which compresses the timestamp graph to the limit, achieving the same space complexity as index for a single timestamp.
- Extensive experiments show that our solution yields a recall rate of over 99% on four standard datasets, while improving the query efficiency by 4.4× to 138.1× over the state-of-the-arts.

II. Problem Statement

We first review the nearest neighbor search, a primitive in vector databases [1], [2], [24], before defining our timestamp approximate nearest neighbor search (TANNS) query.

Definition 1 ((Exact) k Nearest Neighbor (kNN)): Given N high-dimensional vectors $\mathcal{D} = \{u_1, u_2, ..., u_N\}$ and a query vector q of the same dimension, the (exact) kNN query $kNN(\mathcal{D}, q, k)$ retrieves a subset of k vectors that are closest to q, *i.e.*, $\forall u \in kNN(\mathcal{D}, q, k)$ and $\forall v \in \mathcal{D} \setminus kNN(\mathcal{D}, q, k)$, $dis(q, u) \leq dis(q, v)$, where $dis(\cdot, \cdot)$ is the distance function.

Unlike its low-dimensional counterpart [25], kNN on high-dimensional data often prioritizes *efficiency* over *accuracy* to align with the targeting vector database applications [1], [2], [24], which results in the following query optimization scope:

- *Approximate Queries.* Most studies [2], [26], [27] focus on approximate nearest neighbor search (ANNS) with its accuracy assessed by the *recall rate*, a key metric for downstream vector database tasks such as text retrieval [9] and recommendation [28].
- *Dedicated Indexes.* Data in vector database applications is not only high-dimensional but also large-scale, making indexing indispensable for efficient queries. Graph-based indexes, *e.g.*, the hierarchical navigable small-world graph (HNSW) prevail in ANNS queries on high-dimensional vectors [26], [27]. They yield high accuracy by visiting merely $O(\log N)$ vectors [2], [22], [23].

We follow the above scope yet extend it with *time attributes* in both the vectors and the query, as defined below.

Definition 2 (Timestamp Approximate Nearest Neighbor Search (TANNS)): Let $\mathcal{D} = \{u_1, u_2, ..., u_N\}$ be N high-dimensional vectors. Each vector u_i is associated with two timestamps, $u_i.s$ and $u_i.e$ ($u_i.s < u_i.e$), representing the start and end of its validity. Let q be a query vector of the same dimension, which is associated with a timestamp ts. The timestamp approximate nearest neighbor search $TANNS(\mathcal{D}, ts, q, k)$ returns a subset of k valid vectors that are approximately closest to q.

This new query captures the data dynamics and time-aware queries in emerging vector database applications, *e.g.*, memory controls and time references in chat bots [10], [11]. Specifically, we are interested in the following setups.

- Without loss of generality, we assume discrete timestamps, meaning that exactly one vector becomes valid or invalid at each timestamp. The largest timestamp in the dataset is denoted as T.
- We follow the setup of a transaction-time database [29]. A vector u_i becomes valid when it is newly added to $\mathcal{D}$ at $u_i.s$, and expires when it is marked obsolete or replaced by another vector at $u_i.e$. Notably, the end timestamp of one vector's validity is unknown when it is added to $\mathcal{D}$.
- We evaluate query accuracy by recall rate $\frac{|r \cap r^*|}{k}$, where r is the result of the TANNS query, and r^* is the result of the corresponding exact kNN query over all the valid vectors at timestamp ts, *i.e.*, $kNN(\mathcal{D}(ts), q, k)$, where $\mathcal{D}(ts)$ represents all the valid vectors at ts ($1 \leq ts \leq T$).

Example 2: Assume $\mathcal{D}$ contains five vectors whose valid times are shown in Fig. 1, and a query $TANNS(\mathcal{D}, t, q, 3)$, where the distances between q and the vectors in $\mathcal{D}$ are listed in Table II. We have $kNN(\mathcal{D}(t), q, 3) = \{u_1, u_3, u_4\}$ since the valid vectors at t are $\{u_1, u_3, u_4, u_5\}$, and u_5 is further from q than the other three vectors. If the result of TANNS query is $\{u_1, u_3, u_5\}$, then the recall rate is 66.7%, since two out of three vectors in the result appear in the ground truth set.

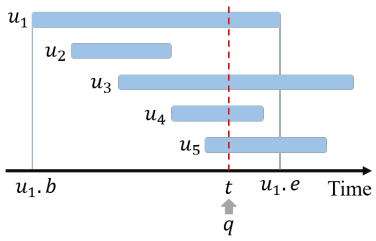


Fig. 1: Example of TANNS.

Distance	q
u_1	2.5
u_2	1.9
u_3	2.1
u_4	1.1
u_5	3.2

TABLE II: Distances between vectors.

We aim to optimize the *efficiency* of TANNS queries while retaining high accuracy over *large-scale*, *dynamic* data [28], [30]. Hence, we explore designs that support efficient index *update* and *search* in terms of both *time* and *memory*. Table III summarizes the major notations that will be used in this paper.

TABLE III: Summary of major notations.

Notations	Description
$\mathcal{D}$	vector dataset
N	total vector number in $\mathcal{D}$
T	the maximal timestamp in $\mathcal{D}$
G	HNSW index
$\mathcal{TG}$	timestamp graph
B	backup neighbor list
k	required result size of TANNS query
k'	parameter controlling searching scope
M	neighbor number
M'	candidate neighbor number during construction
HNT	historic neighbor tree
μ	leaf node size for historic neighbor tree
L_{now}	current neighbor list
L_t	neighbor list at timestamp t

III. Naive Graph-Based TANNS

This section presents a naive solution to TANNS by adopting graph-based indexes. We first review HNSW, a popular graph-based index for ANNS, and then apply it to TANNS.

A. Graph-based ANNS and HNSW

Graph-based indexes are widely adopted for efficient ANNS [23]. Given a set of vectors $\mathcal{D}$, a graph-based index constructs a proximity graph G for $\mathcal{D}$, where each point in G corresponds to a vector in $\mathcal{D}$, and each edge connects a pair of points satisfying certain neighborhood conditions. Then the ANNS can be performed by greedy routing towards the query vector following the graph. These indexes achieve high search efficiency and accuracy since their structures effectively capture the nearby relation between high-dimensional vectors [26].

Algorithm 1: ***HNSW Search***

Input: HNSW index G, query q, query number k
Output: a set of k ANN of q

```
1  ep ← entry point of G;
2  Mark ep as visited;
3  pool ← {ep};
4  ann ← {ep};
5  while pool is not empty do
6      u ← point closest to q in pool;
7      Remove u from pool;
8      v ← point furthest from q in ann;
9      if dis(q, v) > dis(q, u) then break;
10     foreach unvisited point o ∈ G[u] do
11         Mark o as visited;
12         v ← point furthest from q in ann;
13         if |ann| < k' or dis(q, o) < dis(q, v) then
14             Insert v into ann and pool;
15             if |ann| > k' then
16                 Remove point furthest from q in ann;
17 return k points closest to q in ann;
```

The hierarchical navigable small-world graph (HNSW) is a popular graph-based index in the academia and industry [22]. It employs a multilayer graph structure and performs searches from the top layer down to the base layer. We briefly review HNSW's *search* and *construction* algorithms below. For simplicity, we only explain the operations at the base layer. Operations at other layers are similar.

- **HNSW Search**. Searching over the HNSW index follows the greedy routing principle, yet expands the search scope from k to k' (where $k' > k$ is a parameter) to avoid local minima and boost query accuracy (see Algorithm 1). The search begins from a predefined entry point ep and maintains two lists: $pool$ for candidate points and ann for potential nearest neighbors. In each iteration, it picks point v from $pool$ that is closest to q and adds v's neighbors to $pool$. The search terminates when ann reaches the size of k' (*efSearch* in [22]) and all points in ann are closer to q than those in $pool$. Finally, k points in ann that are closest to q are returned as query result.
- **HNSW Construction**. The HNSW index is built via incremental insertion (see Algorithm 2). To insert a point o, we retrieve M' (*efConstruction* in [22]) candidate neighbors and select M of them as its final neighbors $G[o]$ (lines 3-4). The same neighbor selection function (*i.e.*, *Select-Nbrs*) is applied to points in $G[o]$ to ensure that the number of neighbors is within M (lines 5-8). The neighbor selection strategy used in HNSW is heuristic, which prioritizes points closer to o and excluding dominated points (lines 9-16). A point u is dominated by v if $dis(o, v) < dis(o, u)$ and $dis(u, v) < dis(o, u)$.

The time complexity of searching N points by the HNSW index is $O(M \log N)$ since it visit $O(\log N)$ points [22] and

Algorithm 2: *HNSW Construction*

Input: dataset $\mathcal{D}$
Output: HNSW index G for $\mathcal{D}$

```
1  Initialize an empty graph G;
2  foreach point o ∈ D do
3      cand ← Search (G, o, M');
4      G[o] ← Select-Nbrs (o, cand, M);
5      foreach point u ∈ G[o] do
6          Add o to G[u];
7          if u has more than M neighbors then
8              G[u] ← Select-Nbrs (u, G[u], M);
9  return G;
10 Function Select-Nbrs (o, cand, M):
11     Sort cand by ascending distance to o;
12     nbr ← φ;
13     foreach point u ∈ cand do
14         if u is not dominated by points in nbr then
15             Insert u into nbr;
16         if |nbr| = M then break;
17     return nbr;
```

access their M neighbors. The time complexity for HNSW index construction is $O(MN \log N)$, as each point is inserted sequentially, and each insertion involves a search operation which takes $O(M \log N)$ time. The space complexity for the HNSW index is $O(MN)$, as M neighbors are stored per point.

B. Adopting HNSW Index for TANNS Query

Basic Idea. Although HNSW is efficient for ANNS, it assumes a static vector set $\mathcal{D}$, whereas the *valid* vector set varies over time in TANNS. Hence, an intuitive solution to TANNS is to build one HNSW index for the valid vectors at each timestamp, resulting in T distinct HNSW indexes $G_1, G_2, ..., G_T$. Each G_t is constructed with all valid vectors at timestamp t via Algorithm 2. For a given query $TANNS(\mathcal{D}, ts, q, k)$, we pick the corresponding index G_{ts}, on which we search for k approximate nearest vectors via Algorithm 1.

Complexity Analysis. The time complexity for updating the index is $O(MN \log N)$, as a new index is built from scratch using Algorithm 2 each time the list of valid points changes. As each point enters and expires for at most one time, the maximal timestamp T is bounded by $2N$. Thus, the total space usage for T distinct HNSW indexes is $O(MN^2)$.

Remarks. The naive method is inefficient in terms of the index updating time and the index storage, as practical applications may contain millions of vectors [30]. Although our design and analysis is presented in the context of HNSW, it also applies to other graph-based indexes constructed by incremental insertion [23], *e.g.*, NSW [31] and PANNG [32].

IV. Timestamp Graph for TANNS

This section introduces TANNS queries based on the timestamp graph, a novel structure that supports fast index updating and consumes low index storage. We first present the structure of the timestamp graph, then explain its search and construction algorithms, and finally analyze its complexity.

A. Timestamp Graph

Rationale. Data dynamics in practical vector database applications are mild [33], [34], which implies that the valid vector sets at nearby timestamps largely overlap. Hence, it is unnecessary to maintain a separate index per timestamp, a key efficiency bottleneck in the naive solution. We take advantage of such *temporal locality* and manage valid vectors across all T timestamps by a single structure, as explained below.

Structure. A timestamp graph $\mathcal{TG}$ is a proximity graph defined as in Sec. III-A yet over all historic valid vector sets $\mathcal{D}(1)$, ..., $\mathcal{D}(T)$. It can be considered as aggregating the T per-timestamp HNSW indexes into a single graph by removing the unchanged valid vectors between adjacent timestamps. Unlike the HNSW index that only stores the current neighbor list for each point, the timestamp graph tracks neighbor lists for all historic timestamps. By scanning the historic neighbor lists, it can trace the neighbors of a point at any timestamp.

Example 3: In Table IV, the column $\mathcal{TG}[o]$ represents the historic neighbor list of point o. At timestamp t_1, u_3 and u_4 are chosen as o's neighbors, and they remain as neighbors until t_2, when o's neighbors are updated to u_1 and u_3. Thus, when visiting point o at timestamp between t_1 and t_2, we regard u_3 and u_4 as its two neighbors, while u_1 and u_3 are treated as neighbors between t_2 and t_3.

B. TANNS Query on Timestamp Graph

To process a query $TANNS(\mathcal{D}, ts, q, k)$ with a timestamp graph $\mathcal{TG}$, we search for the approximate nearest neighbors in a similar way to Algorithm 1. The main difference is that when visiting point u in line 10, its neighbors $G[u]$ are not fixed but dependent on the query timestamp ts. The neighbors of u at timestamp ts can be obtained by a binary search on u's historic neighbor list. All other steps remains the same as standard HNSW search. The algorithm yields high accuracy and efficiency since it specifically searches within the graph built using vectors valid at the query timestamp.

C. Timestamp Graph Construction

In the naive solution (Sec. III-B), an entire HNSW index for $\mathcal{D}(t)$ points is built from scratch using valid vectors per timestamp t. In contrast, the construction of the timestamp graph only involves one point per timestamp. However, it must handle not only point *insertion*, but also point *expiration*. We show that point expiration impairs the graph connectivity and thus the search accuracy, and solve the problem with *backup neighbors*, as explained below.

Need for Backup Neighbors. To update the timestamp graph at a specific timestamp t, if a new point becomes valid, we

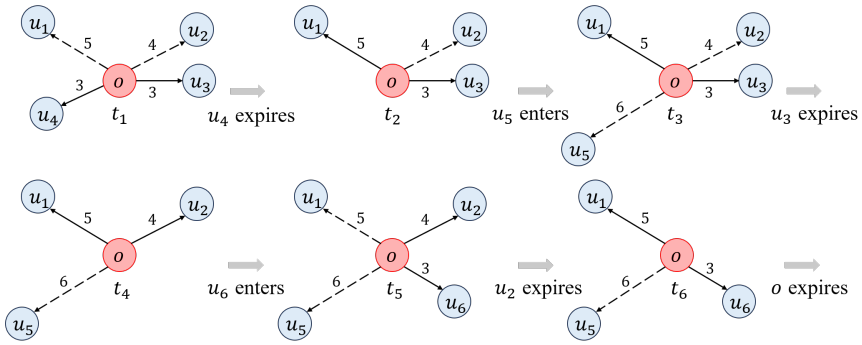


Fig. 2: Neighbors of point o during timestamp graph updating.

Timestamp	$\mathcal{TG}[o]$	$B[o]$
t_1	u_3, u_4	u_1, u_2
t_2	u_1, u_3	u_2
t_3	u_1, u_3	u_2, u_5
t_4	u_1, u_2	u_2
t_5	u_2, u_6	u_1, u_5
t_6	u_1, u_6	u_5
t_7	None	None

TABLE IV: Primary neighbors and backup neighbors of point o.

add it to the graph and connect it with M neighbors. If otherwise a point expires, we mark it as invalid and disconnect it from its current M neighbors. Then we append a new neighbor list (with the invalid point removed) to all the affected neighbors. Over time, the neighbor list of certain points may become empty, reducing the overall graph connectivity, which decreases the search accuracy [34].

We try to maintain the graph connectivity by introducing a backup neighbor list per point. Specifically, for each point u, we store M backup neighbors $B[u]$ in addition to M primary neighbors in $\mathcal{TG}[u]$. Consequently, when a point in $\mathcal{TG}[u]$ expires, certain point in $B[u]$ can take up its place, avoiding decreased connectivity. With the backup neighbors in mind, we now describe the algorithm details for handling point insertion and expiration to update the timestamp graph at a given timestamp t.

Point Insertion. Adding a point to the timestamp graph $\mathcal{TG}$ is similar to the insertion in the HNSW, except for an extra step to maintain the backup neighbor list (see Algorithm 3). Concretely, when a new point o is added, we search the current graph and retrieve M' candidate neighbors as in the HNSW, but select $2M$ points from the candidates. The M points closest to o are allocated to $\mathcal{TG}[o]$ as primary neighbors, while the remaining points are stored in $B[o]$ as backup neighbors. For any point u whose neighbor list $\mathcal{TG}[u]$ is modified after point insertion, we append a new version with timestamp t in u's historic neighbor list (lines 4-12).

Point Expiration. When a point expires, we disconnect the expired point from its current neighbors and select new neighbors for these points (see Algorithm 4). Specifically, we process all neighbors u of o as follows: If o is a backup neighbor for u, we simply discard o from $B[u]$ (lines 2-3). If o is in $\mathcal{TG}[u]$, *i.e.*, a primary neighbor, we replace o with a point from $B[u]$ (lines 7-9). In case the backup neighbor list $B[u]$ is empty, we reselect neighbors for u as if u were a newly added point (lines 11-12). Since the frequency of neighbor reselection is low, such time-consuming operation does not notably affect the overall efficiency.

Example 4: Fig. 2 and Table IV illustrate the changes to point o's neighbors when updating the timestamp graph. Initially, o selects $\mathcal{TG}[o] = \{u_3, u_4\}$ as primary neighbors (connected with full line) and $B[o] = \{u_1, u_2\}$ as backup neighbors (connected with dashed line). At timestamp t_1, point u_4 expires and o chooses u_1 from $B[o]$ to replace u_4 as its neighbor (while u_2 is skipped as it is dominated by u_3). Then point u_5 appears at timestamp t_3. Although it is excluded from $\mathcal{TG}[o]$ because its distance from o is larger than other o's neighbors, we keep it in $B[o]$ as a backup neighbor. At timestamp t_4, u_3 expires and u_1 takes over its place. Then, u_1 is replaced by u_6 at timestamp t_5 and added into the backup list. Finally, point o expires at t_7. We clear $B[o]$ but keep its historic neighbor list, ensuring that queries with timestamps before t_7 can be performed correctly.

Algorithm 3: ***Point Insertion***

Input: timestamp graph $\mathcal{TG}$, point o, timestamp t

1 $cand \leftarrow$ ***Search*** $(\mathcal{TG}, o, M')$;
2 $\mathcal{TG}[o], B[o] \leftarrow$ ***Select-Nbrs*** $(o, cand, 2M)$;
3 Initialize o's historic neighbor list with $\mathcal{TG}[o]$;
4 **foreach** $u \in \mathcal{TG}[o]$ **do**
5 **if** *o is not dominated by points in $\mathcal{TG}[u]$ and closer to u than furthest points in $\mathcal{TG}[u]$* **then**
6 Move the furthest point in $\mathcal{TG}[u]$ into $B[u]$;
7 Add o to $\mathcal{TG}[u]$;
8 Append $\mathcal{TG}[u]$ to the historic neighbor list of u, associated with timestamp t;
9 **else**
10 Add o to $B[u]$;
11 **if** $|B[u]| > M$ **then**
12 Remove the furthest point from $B[u]$;

D. Complexity Analysis of Timestamp Graph

For each point in timestamp graph, finding all neighbors at a given timestamp via binary search on the historic neighbor list takes $O(\log N)$ time, resulting in an overall search complexity of $O(\log^2 N)$. For graph updating, we perform a search operation at first, which takes $O(\log^2 N)$ time. Although a search may also be triggered when handling point expiration, this only occurs once every M updates in the worst case, since

Algorithm 4: ***Point Expiration***

Input: timestamp graph $\mathcal{TG}$, point o, timestamp t

```
foreach u ∈ TG[o] do
    if o ∈ B[u] then
        Remove o from B[u];
    else if o ∈ TG[u] then
        Remove o from TG[u];
        if B[u] is not empty then
            cand ← TG[u] ∪ B[u];
            TG[u] ← Select-Nbrs (u, cand, M);
            Remove all elements in TG[u] from B[u];
        else
            cand ← Search (TG, u, M');
            TG[u], B[u] ← Select-Nbrs (u, cand, 2M);
        Append TG[u] to the historic neighbor list of
          u, associated with timestamp t;
```

only one point expires each time. Thus the time complexity for index updating is $O(\log^2 N) + \frac{M \cdot O(\log^2 N)}{M} = O(\log^2 N)$.

Due to the locality of timestamp graph updating, both point insertion and expiration only affect M points, *i.e.*, only the neighbor lists of these M points are modified, and we only store the new neighbor lists for them. Since each update requires $O(M^2)$ additional storage, the overall space complexity of the timestamp graph is $O(M^2N)$.

V. Compressing Timestamp Graph

This section pushes the space complexity of the timestamp graph to the limit (*i.e.*, same as HNSW index for a single timestamp) by compressing and managing the neighbor lists in the timestamp graph via the historic neighbor tree. We explain its structure, introduce the algorithms for neighbor list reconstruction and compression, and analyze the complexity of the compressed timestamp graph.

A. Historic Neighbor Tree

Motivation. Although the timestamp graph reduces the index storage from $O(MN^2)$ to $O(M^2N)$, there is still redundancy in the *neighbor lists* of the timestamp graph. As it is unlikely to replace the entire neighbor list at once, the same points tend to appear in multiple neighbor lists. For example, in Table IV, u_1 appears in o's neighbor list for four times, thereby occupying redundant space.

The design of historic neighbor tree is inspired by the classic interval tree [35], but it differs in two key aspects: *(i)* The historic neighbor tree manages points whose presence in the neighbor list are unknown in advance, unlike the interval tree, where both endpoints of intervals are predefined. *(ii)* The historic neighbor tree employs a bottom-up construction approach to ensure the tree remains balanced.

Structure. A historic neighbor tree HNT for a point $o \in \mathcal{D}$ is a binary tree with multiple nodes. Fig. 3 illustrates the structure of an internal node of a historic neighbor tree. It contains a unique node identifier, a timestamp t, and the points valid at t. The leaf nodes share a similar structure but lack the timestamp, with the points contained determined by the construction process (see Sec. V-C). Both node types maintain two sequences for the points they contain: one sorted by the points' start times and the other by their end times. Furthermore, a point valid in multiple nodes is stored only *in the highest node* where it is valid. For example, in Fig. 4, although u_3 is valid at timestamp t_3, it should not be stored in node 1 since it can be placed in the higher node 3.

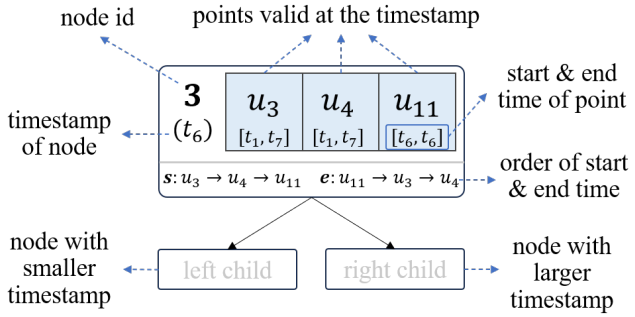


Fig. 3: Structure of a node in historic neighbor tree.

In addition to the historic neighbor tree, we use L_{now} to record points in the current neighbor list. These points are sorted by their start times. Once points are removed from L_{now}, they are added to the historic neighbor tree.

Next, we explain how to *(i)* reconstruct the neighbor list at a given timestamp from the historic neighbor tree, and *(ii)* construct the historic neighbor tree from all neighbor lists.

Example 5: Suppose we aim to reconstruct L_{t_2}, the neighbor list at timestamp t_2. We first scan the current neighbor list and add u_6, as it is valid at t_2. Then we stop at u_{10} since its start time exceeds t_2. Afterwards, we traverse the historic neighbor tree from the root node, adding u_3 and u_4 as they are valid at t_2. We proceed to node 1 (since $t_2 < t_6$), adding u_1, then to node 0, where u_2 is the only valid point. In the end, we reconstruct L_{t_2} as $\{u_1, u_2, u_3, u_4, u_6\}$.

B. Neighbor List Reconstruction

Since the historic neighbor list may grow as large as N, a fast reconstruction is necessary. Algorithm 5 outlines the steps to reconstruct the neighbor list L_t at a given timestamp t from the historic neighbor tree. We first scan the current neighbors L_{now} and add points valid at t until encountering a point with a start time later than t (lines 2-5). Next, we start from the root and traverse the historic neighbor tree (lines 6-12). At each node, valid points are added to L_t. If t matches the node's timestamp, traversal stops as no valid points exist in lower nodes. Otherwise, traversal continues to the left or right child, depending the relation between t and the node's timestamp. Reconstruction is completed after reaching a leaf node and including all valid points stored in it.

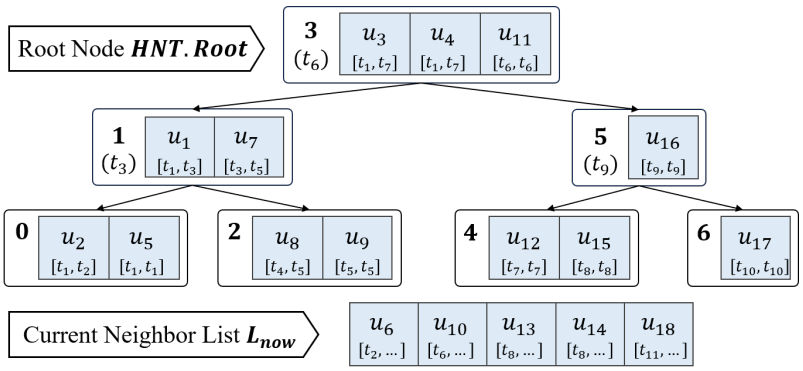


Fig. 4: Compressed historic neighbor list for point o.

Timestamp	$\mathcal{TG}[o]$
t_1	u_1 , u_2 , u_3 , u_4 , u_5
t_2	u_1 , u_2 , u_3 , u_4 , u_6
t_3	u_1 , u_3 , u_4 , u_6 , u_7
t_4	u_3 , u_4 , u_6 , u_7 , u_8
t_5	u_3 , u_4 , u_7 , u_8 , u_9
t_6	u_3 , u_4 , u_6 , u_{10}, u_{11}
t_7	u_3 , u_4 , u_6 , u_{10}, u_{12}
t_8	u_6 , u_{10}, u_{13}, u_{14}, u_{15}
t_9	u_6 , u_{10}, u_{13}, u_{14}, u_{16}
t_{10}	u_6 , u_{10}, u_{13}, u_{14}, u_{17}
t_{11}	u_6 , u_{10}, u_{13}, u_{14}, u_{18}

TABLE V: Historic neighbor lists for point o.

Algorithm 5: ***Neighbor List Reconstruction***

Input: historic neighbor tree HNT, timestamp t
Output: neighbor list at timestamp t

```
1  L_t ← φ;
2  foreach point p ∈ L_now do
3      if p is valid at t then
4          Insert p into L_t;
5      else break;
6  Node n ← HNT.Root;
7  while n is not leaf do
8      For node n, add all valid points at t to L_t;
9      if t = n.t then return L_t;
10     if t < n.t then n ← n.left;
11     else n ← n.right;
12 For node n, add all valid points at t to L_t;
13 return L_t;
```

C. Historic Neighbor Tree Construction

We now describe how to compress the neighbor lists of a point o into a historic neighbor tree. We dynamically build the historic neighbor tree since the neighbors at each timestamp are unknown in advance. Upon receiving the latest neighbor list at t, we append it to the existing historic neighbor tree.

Algorithm 6 outlines the procedures. Consider the points in $L_{now} \setminus L_t$, which have just been removed from the current neighbor list. We first check whether point p already exists in the historic neighbor tree. If not, we execute the routine for adding a new point to the historic neighbor tree (lines 3-12). Otherwise, we simply adjust the position of the existing point in the tree (lines 14-17). Finally, we assign the current neighbor list to L_{now} (line 18). The details of adding a new point and adjusting an existing point are as follows.

Adding New Point. First, we identify the *active path* in the historic neighbor tree, *i.e.*, the path from the newest leaf node n_c to the root node $HNT.Root$. A new point p is added by scanning this path for the highest node where p can be placed (lines 3-6). If p is placed in a leaf node, we check whether the node exceeds the size threshold μ. If yes, we consider this

Algorithm 6: ***Append Neighbor List***

Input: historic neighbor tree HNT,
the latest neighbor list L_t

```
1  foreach point p ∈ L_now \ L_t do
2      if p ∉ HNT then // Adding New Point
3          Node n_c ← the newest leaf node of HNT;
4          foreach n in path from HNT.Root to n_c do
5              if n = n_c or p is valid at n.t then
6                  Insert p into n and break;
7          if n_c has more than μ points then
8              Subtree tree_c ← the maximal complete
                binary tree containing n_c;
9              Create internal node n_i, with tree_c as its
                left subtree and t as its timestamp;
10             Create leaf node n_l as right sibling of n_c;
11             if n_i is higher than HNT.Root then
12                 HNT.Root ← n_i;
13     else // Adjusting Existing Point
14         Find node n_p which p located in;
15         foreach n in path from HNT.Root to n_p do
16             if p is valid at n.t then
17                 Move p from n_p into n and break;
18 L_now ← L_t;
```

node as filled and create two new nodes, an internal node n_i and a leaf node n_l to accommodate future points (lines 7-10). Finally, n_i becomes the new root node if it is higher than the current root, and the tree height increases by one (lines 11-12). In this way, the historic neighbor tree is constructed from bottom to top.

Example 6: Fig. 5 (a)-(f) illustrates the evolution of the historic neighbor tree as points are added. The active path (highlighted in green) is the path from the root node to the newest leaf node, where the new point will be inserted. When the newest leaf node is filled, two new nodes are created at the same time, as shown in lines 10-12 in Algorithm 6. For

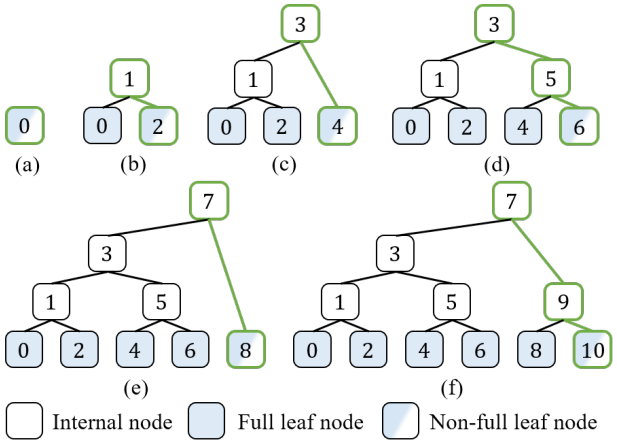


Fig. 5: Evolution of tree structure as new points are added to the historic neighbor tree.

example, in the transition from (b) to (d), when node 2 is full, the subtree rooted at node 1 is complete, and thus the parent node for this subtree is created, which is node 3. Besides, the sibling node 4 is created. Similarly, when node 4 is full, node 5 is created as its parent because the maximal complete tree containing 4 is node 4 itself, and node 6 becomes its sibling.

Adjusting Existing Point. It handles the cases where a point is removed from the neighbor list and then re-enters. For instance, in Table IV, point u_1 enters the neighbor list at timestamp t_2 and t_6, and adjusting should be applied at t_2. The operation begins by locating point p in the historic neighbor tree, which is denoted as n_p. Then we attempt to lift point p as high as possible along the path from n_p to the root node. We will prove in Lemma 1 that a point should not be adjusted to any other position in the tree. Although the adjusting operation may cause point p to appear in more versions of neighbor list than it is actually in, it does not affect the search accuracy as p remains valid at each reconstructed list it appears.

Lemma 1: If point p is adjusted from node n_p to node n'_p, then n'_p is an ancestor of n_p.

Proof: We prove the lemma by contradiction. Assume that node n'_p is not an ancestor or descendant of n_p. Then n_p and n'_p has a lowest common ancestor node c and c is neither n_p nor n'_p. Since point p is valid at both $n_p.t$ and $n'_p.t$, it must be valid at $c.t$ as well, because $c.t$ lies between $n_p.t$ and $n'_p.t$. Therefore, the point should be placed at a higher node c, rather than at node n'_p, which violates our assumption. This contradiction shows that n'_p must be n_p's ancestor or descendant. Moreover, node n_p cannot be an ancestor of n'_p, since p is still valid at $n_p.t$, and it should be placed as high as possible in the tree, which completes the proof. ■

D. Complexity Analysis of Compressed Timestamp Graph

We first characterize the structure of historic neighbor tree and then analyze the complexity of the compressed timestamp graph. We begin by showing the historic neighbor tree is balanced, whose height is logarithmic relative to the number of points it contains.

Lemma 2: If a historic neighbor tree has n points, then its height is at most $\lceil \log n \rceil$.

Proof: Suppose a historic neighbor tree has h levels, then its left subtree must be full (according to lines 7-10 in Algorithm 6), meaning that the subtree contains 2^{h-1} leaf nodes. Thus, the entire tree contains at least $\mu \cdot 2^{h-1}$ points, where the leaf node size μ is chosen to be greater than 2. Given that, a historic neighbor tree with 2^h points has a height of at most h, which completes our proof. ■

Time Complexity. We begin by analyzing the complexity of historic neighbor tree.

Lemma 3: The time complexity of neighbor list reconstruction of point u at a given timestamp is $O(\log n + M_r)$, where n is the number of points that ever appear in u's history neighbor list and M_r is the size of the reconstructed neighbor list.

Proof: We prove the lemma by showing that almost all the points visited in Algorithm 5 appear in the reconstructed neighbor list. The exceptions are one point in the current neighbor list L_{now} and at most one points in each layer of the historic neighbor tree. The reasons are as follows:

- In lines 2-5, all points visited are valid until we encounter the first invalid point, at which the scanning stops.
- In lines 7-14, we traverse the historic neighbor tree to add valid points from the nodes we visit. By using either the start time or end time order, we guarantee to access at most one invalid point per node. If $t < n.t$, *i.e.*, the query timestamp is smaller than the node's timestamp, we scan the list in the order of the start time and stop once we encounter an invalid point. Similarly, if $t > n.t$, we scan the list in the order of the end time.

In summary, the time complexity of neighbor list reconstruction is the summation of the layer number in historic neighbor tree and the reconstructed neighbor list size, which is $O(\log n + M_r)$. ■

Our experimental results (see Sec. VI-B) indicate that the maximal size of the M_r is $1.31M$, which is slightly larger than the original size of neighbor list. Likewise, we can conclude that the time complexity for inserting into a point's historic neighbor tree is $O(\log n)$, since the insertion operation visits only one node at each layer.

We continue to analyze the compressed timestamp graph. Since the number of visited nodes during both search and update is $O(\log N)$, the time complexity for searching and updating the compressed timestamp graph is $O(\log^2 N)$.

Space Complexity. We proceed to analyze the space complexity for the compressed timestamp graph.

Theorem 1: The space complexity of compressed timestamp graph is $O(MN)$, where N is the size of dataset and M is the neighbor number for each point.

Proof: For a point with n points ever appearing in its historic neighbor list, the space complexity of its historic neighbor tree is $O(n)$, because each point in the historic neighbor list is stored in only one location, either in the current neighbor list L_{now} or in a node of the historic neighbor tree. Thus, the total space required for all points' neighbors

is $\sum_{i=1}^{N} O(n_i)$, where n_i is the number of history neighbors of point u_i. This corresponds to the number of edges created during the index construction. Since each point creates at most $2M$ edges as it is inserted into the timestamp graph, the upper bound for $\sum_{i=1}^{N} n_i$ is $2MN$. Therefore, the space complexity of the compressed timestamp graph is $O(MN)$. ■

Note that the space complexity of the compressed timestamp graph has reached the optimal, as the space complexity of HNSW index at a single timestamp is $O(MN)$.

VI. Experimental Study

This section presents the evaluations of our method on standard benchmarks.

A. Experiment Setup

Datasets. We use four common high-dimensional vector datasets in our experiment.

- **SIFT [36]**: Each point represents an 128-dimensional image descriptor, extracted from INRIA Holidays images using the Scale-Invariant Feature Transform algorithm.
- **GIST [36]**: Each point is a 960-dimensional image descriptor derived from the Holidays image set and Flickr images, capturing global image properties.
- **DEEP [28]**: Each point is a 96-dimensional feature vector embedded using the GoogLeNet model, representing image features extracted from web images.
- **GloVe [30]**: Each data point is a 200-dimensional word vector generated from tweets using the Global Vectors for Word Representation model.

By default, 1 million vectors are used as data vectors for each dataset. Euclidean distance is used to measure similarity in SIFT, GIST, and DEEP, while cosine distance is used in GloVe.

Workloads. To assess the performance of TANNS across various application scenarios, we assign data vectors to four types of randomly generated temporal data patterns.

- **Short**: All points have valid time ranges less than $0.05T$.
- **Long**: All points have valid time ranges exceeding $0.4T$.
- **Mixed**: Points are divided into two groups of roughly equal size: one with valid time ranges over $0.4T$, and the other under $0.05T$.
- **Uniform**: Points have valid time ranges uniformly distributed between 1 and T.

Standard query sets from the four datasets are used, with query timestamps randomly selected between 1 and T. The number of TANNS results required for each query is set to $k = 10$.

Baselines. We compare our method with following solutions.

- **Pre-Filtering [33]**: It first filters the dataset to retrieve all vectors valid at the query timestamp, then scans the valid vectors to find vectors closest to the query. The method always guarantees exact results.
- **Post-Filtering (HNSW) [33]**: It first searches the vector index without considering timestamps to retrieve a large candidate set, then filters the results based on validity at the query timestamp. HNSW [22] is used as the vector index, configured with $M = 16$, $M' = 200$.
- **ACORN [37]**: It constructs the graph index in a predicate-agnostic manner, supporting ANNS with diverse query predicates. However, ACORN needs to determine the construction parameter γ based on the attribute distribution in the dataset, which is unavailable in our scenario. To adapt ACORN for TANNS, we assume that the validity period of each vector and each query timestamp are known in advance. We then set γ as the inverse of the average query selectivity. Other parameters are set to their default values, *i.e.*, $M = 32$ and $M_\beta = 64$.
- **SeRF [21]**: It is designed for range-filtering ANNS queries, where each vector is associated with an attribute, and the query searches among data vectors whose attributes fall within the specified range. However, TANNS queries differ as each vector's validity is determined by two timestamps, with the end timestamp unknown in advance. To adapt SeRF for TANNS, we perform queries on the 1D segment graph using the start time as the attribute and apply post-filtering based on the end time. The default construction parameters for 1D segment graph are used, *i.e.*, $M = 16$, $K = 200$.

Implementation. All methods are implemented in C++ and compiled using GCC 9.4.0 with -Ofast optimization. For the timestamp graph, we use the following default parameters: neighbor number $M = 16$, candidate neighbor number $M' = 200$ and leaf node size for historic neighbor tree $\mu = 8$. Experiments are conducted using a single thread. We repeat each experiment multiple times and report the average result.

Environment. All the experiments are carried on a server with Intel Xeon(R) Gold 6240 2.60GHz CPU processors and 768GB RAM.

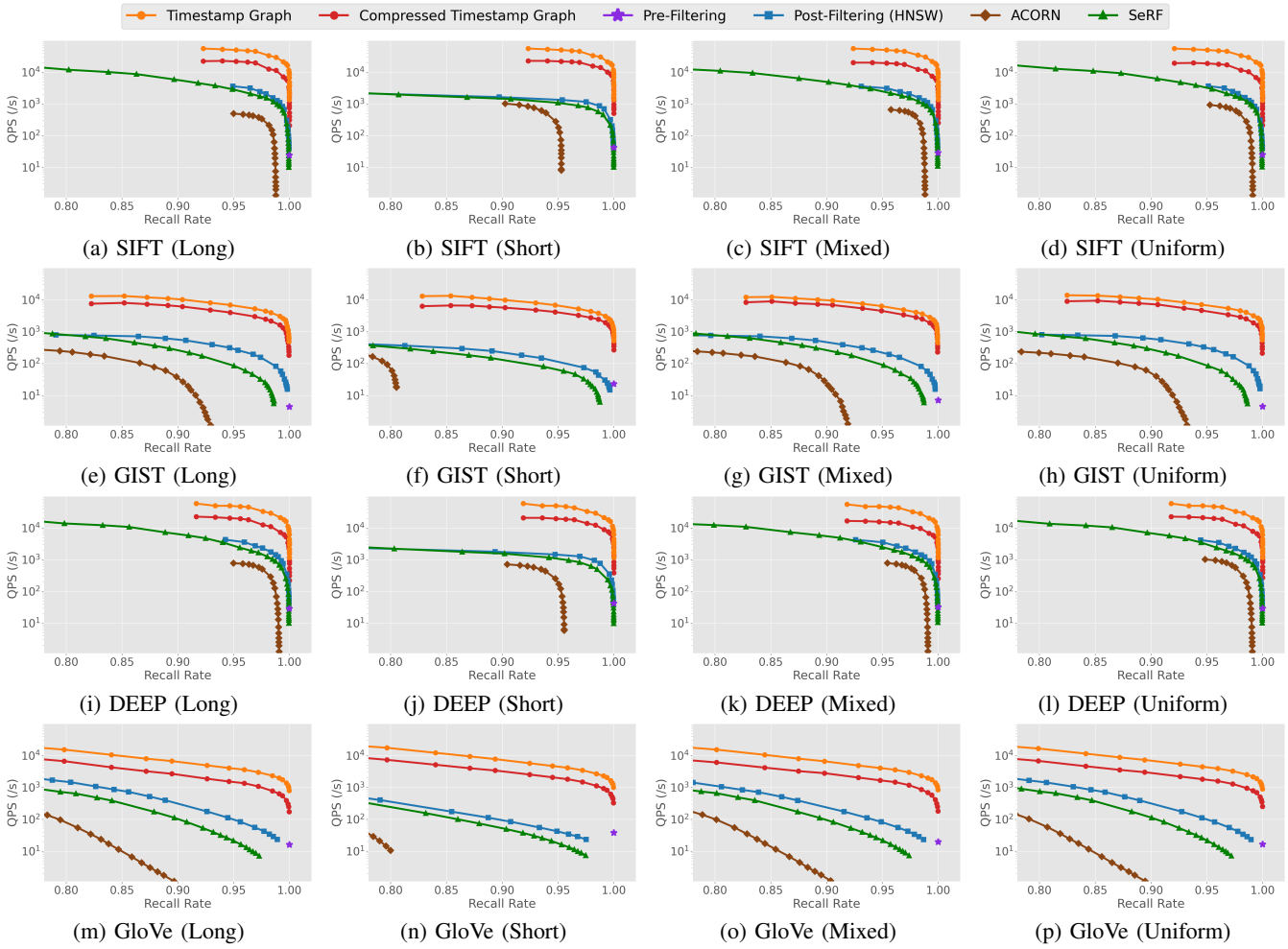
B. Experiments on Search Performance

In this section, we evaluate the search performance of our method on all four datasets using the following metrics:

- **Queries Per Second (QPS)**: It measures the average number of queries executed per second. The metric reflects the search efficiency.
- **Recall Rate**: As described in Sec. II, the recall rate is calculated as $\frac{|r \cap r^*|}{k}$, where r represents the TANNS query result, r^* is the ground truth for the query, and k is the required result size (set to $k = 10$ in the experiment). This metric measures the search accuracy.

Following established benchmarks for ANNS [23], [26], [27], we use QPS vs. Recall Rate plots to show the trade-off between search speed and accuracy. Different data points in these plots are generated by varying k', the parameter controlling the search scope.

Comparison Between Methods. Fig. 6 presents the relationship between QPS and recall rate. For all methods, QPS and recall rate are inversely related, indicating that search efficiency decreases as recall rate improves. Across all datasets and data patterns, both timestamp graph and compressed timestamp graph consistently outperform baseline algorithms in QPS at the same recall rate. The QPS of timestamp graph



(a) SIFT (Long) (b) SIFT (Short) (c) SIFT (Mixed) (d) SIFT (Uniform)

(e) GIST (Long) (f) GIST (Short) (g) GIST (Mixed) (h) GIST (Uniform)

(i) DEEP (Long) (j) DEEP (Short) (k) DEEP (Mixed) (l) DEEP (Uniform)

(m) GloVe (Long) (n) GloVe (Short) (o) GloVe (Mixed) (p) GloVe (Uniform)

Fig. 6: Searching performance on four datasets under different data patterns.

has an advantage of 4.4-138.1× over the existing solutions at the recall rate of 95%. For example, on the GloVe dataset with a uniform data pattern, timestamp graph achieves a QPS of 4464, offering an 54.0× advantage over to the best baseline which processes 82 TANNS queries per second.

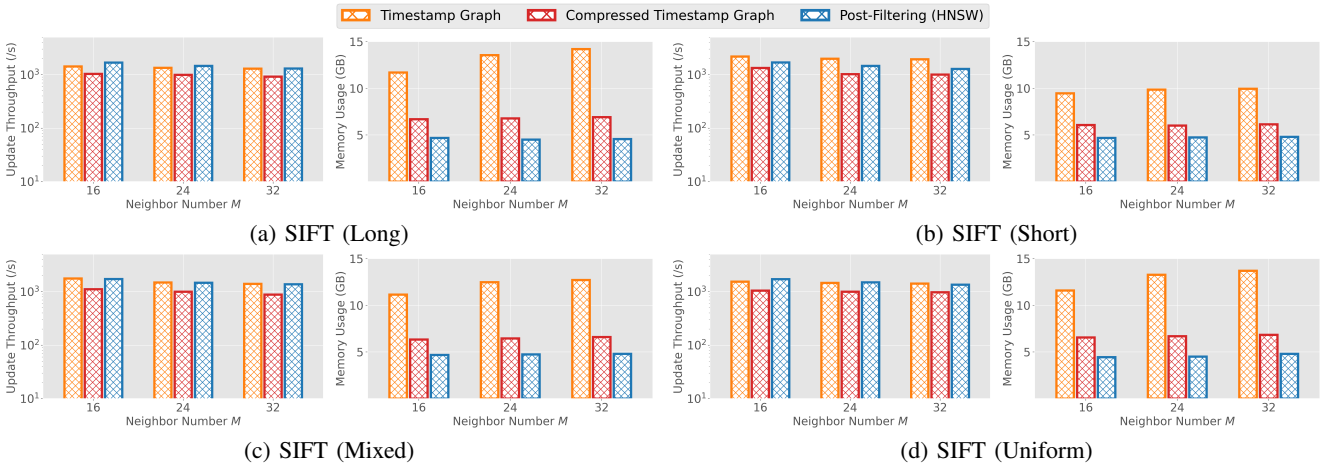
Varying Datasets. As shown in Fig. 6, both timestamp graph and compressed timestamp graph achieve a recall rate exceeding 99% across all four datasets. Besides, the timestamp graph offers the highest QPS across all datasets when the recall rate is constrained to be above 95%. For the uniform data pattern at a fixed recall rate of 95%, the timestamp graph achieves QPS of 49751 for SIFT, 7042 for GIST, 49875 for DEEP, and 4464 for GloVe. Among the datasets using euclidean distance, GIST has the lowest QPS due to its higher dimensionality: as dimensions increase, the computational cost of calculating vector distances rises, leading to longer search times.

Varying Data Patterns. Data patterns affect search performance by influencing the *selectivity* of the query timestamp, *i.e.*, the ratio of valid vectors at a given timestamp to the total vectors in $\mathcal{D}$. In the GIST dataset and at the recall rate of 95%, the QPS of Post-Filtering and SeRF decreases from 260 and 84 to 74 and 57, respectively, when transitioning from the long to short data pattern. This decline occurs because lower selectivity forces these methods to expand their search scope to satisfy temporal constraints, thus reducing efficiency. In contrast, the influence of data pattern on our method is minimal. Fixing the recall rate at 95%, timestamp graph processes 3984 searches per second in GIST (Long) dataset and 4149 searches per second in GIST (Short) dataset. The reason behind is that timestamp graph exclusively searches valid points, avoiding performance degradation even under low selectivity conditions.

C. Experiments on Index Construction

In this part, we evaluate the index construction performance of the timestamp graph using the following metrics:

- **Update Throughput**: Similar to QPS, we use throughput to measure the efficiency of index updates. Specifically, it reflects the average number of updates to the dataset that can be processed per second.
- **Memory Usage**: This metric reports the total memory consumption of the method, including both the raw vector data and the constructed index.



(a) SIFT (Long) (b) SIFT (Short)

(c) SIFT (Mixed) (d) SIFT (Uniform)

Fig. 7: Indexing construction performance on the SIFT dataset under different data patterns.

The experiments are conducted on the SIFT dataset with four temporal data patterns. We compare the performance of timestamp graph with the Post-Filtering (HNSW) method, which constructs an HNSW index using all vectors in $\mathcal{D}$ without considering the valid time ranges of the vectors.

Comparison Between Methods. Fig. 7 presents the update throughput and memory usage of methods under various data patterns. The timestamp graph achieves update throughput comparable to the original HNSW index, ranging between 0.8× and 1.5× of the corresponding HNSW values, indicating the efficiency of our update method. The compressed timestamp graph can support a throughput of over 1000 updates per second when the neighbor number is 16, which is sufficient for real-world vector search applications [33], [34]. Regarding memory usage, the compressed timestamp graph reduces memory consumption by 35.9%-51.4% compared to the timestamp graph, with the reduction being more pronounced when the graph is constructed with a larger neighbor number.

Varying Data Pattern. HNSW's update throughput and memory usage remain unaffected by data patterns, as it does not account for time attributes during index construction. In contrast, the update throughput for the timestamp graph is influenced by the data pattern. For example, compressed timestamp graph built for short data pattern exhibits an 1.3× higher throughput and consumes 90.7% of the memory used by the graph constructed for long data pattern. This is because timestamp graphs for short patterns contain fewer valid vectors, which reduces the time required to search for candidate neighbors during index construction, thereby improving update throughput. Besides, vectors with shorter valid ranges have fewer historic neighbors, leading to reduced memory usage.

D. Experiments on Scalability Test

In this part, we evaluate the scalability of the timestamp graph using the DEEP dataset. Experiments are conducted with dataset sizes ranging from 2 million to 10 million vectors. Four metrics are used to assess performance, *i.e.*, QPS, recall rate, update throughput, and memory usage.

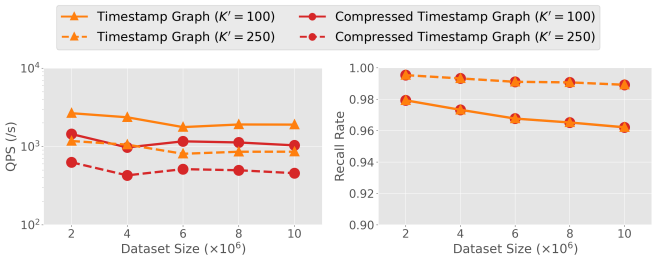


Fig. 8: Search performance of scalability test.

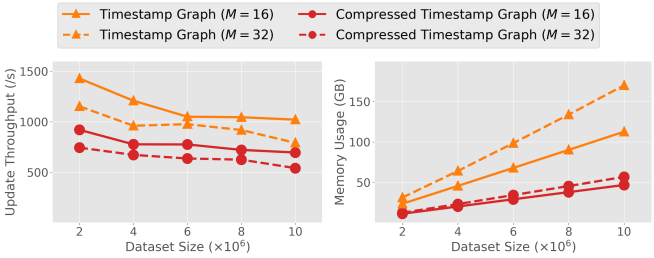


Fig. 9: Index construction performance of scalability test.

Search Performance. Fig. 8 shows the recall rate and QPS of TANNS when varying dataset size. The neighbor number M is fixed at 16, with the search scope parameter k' set to 100 and 250 for the experiment. As the dataset size increases, the recall rate of the timestamp graph shows a slight decline for a fixed search scope. Besides, even with the largest dataset containing 10 million vectors, the timestamp graph achieves a QPS of 857 while maintaining a recall rate of 99%, showing good scalability of timestamp graph in search performance.

Index Construction Performance. Fig. 9 illustrates the update throughput and memory usage of timestamp graph as dataset size increases, with neighbor numbers set to 16 or 32. We observe that the update throughput decreases gradually as the dataset size grows. Moreover, memory usage grows nearly linearly with the dataset size. The compressed timestamp graph

requires 11.2GB of memory for 2 million vectors and 46.7GB for 10 million. Besides, compressed timestamp graph reduces memory usage by up to 66.8%, demonstrating the effectiveness of historic neighbor tree in optimizing the storage overhead.

E. Summary of Major Experimental Findings

Previous experimental results are summarized as follows:

- Timestamp graph exhibits strong performance in both search efficiency and accuracy. It achieves a recall rate exceeding 99% across all four datasets and provides a 4.4× to 138.1× improvement in search efficiency compared to existing methods at similar recall levels.
- Experiments on index construction show that our method processes over 1000 updates per seconds. Besides, the historic neighbor tree efficiently compresses the timestamp graph, reducing memory usage by up to 51.4%.
- Our method demonstrates good scalability both in both search performance and index construction. In the DEEP dataset with 10 million vectors, timestamp graph achieves a QPS of 857 while maintaining a recall rate of 99%.

VII. Related Work

Approximate Nearest Neighbor Search. ANNS is a core operation in vector databases [1], [24], [33], [38]. Indexes for ANNS mainly fall into three categories: partition-based [39]–[44], quantization-based [36], [45]–[48], and graph-based [22], [23], [49]–[53], where the last one best balances search accuracy and efficiency [26], [27]. HNSW and NSG are two representative graph-based indexes. HNSW [22] constructs a graph by incremental point insertion, *i.e.*, each point is successively added into the graph and connected to the existing points close to it. NSG [50] first initializes a KGraph as the base graph and then refines it using the MRNG neighbor selection strategy. However, these two indexes struggle to support point deletion. FreshDiskANN [54] is a graph-based method that supports point updates via periodical point removal and reconnection. Yet it cannot be adapted to our problem as it fails to achieve the fine granularity as the timestamp graph.

Another relevant research is hybrid approximate nearest neighbor search [19]–[21], [37], [55]–[60], which performs ANNS among vectors with specified attributes. However, these solutions are mainly designed for static vector data stores, and most cannot support the semantics of timestamp filtering in TANNS. Among them, we empirically compared our method with ACORN [37] and SeRF [21], two state-of-the-art methods adaptable for TANNS.

Temporal Data Management. Temporal databases store and retrieve data records that changes over time [61]. There are many dedicated indexes for temporal data [29]. Among them, MVB-Tree [62], HV-Tree [63], Timeline Index [64], and LIT [65] are indexes designed for time-varied relational data. Other indexes are intended for querying spatial-temporal objects, such as HR-Tree [12], TPR-Tree [13] and MV3R-Tree [14]. Yet they struggle to support similarity searches in the high-dimensional space due to the "curse of dimensionality" [18].

VIII. Conclusion

This paper defines the Timestamp Approximate Nearest Neighbor Search (TANNS) and designs both time- and space-efficient solutions for TANNS queries. We propose the timestamp graph to effectively manage valid vectors across all timestamps via a unified structure. We further develop the historic neighbor tree to compress neighbor lists in the timestamp graph and push its space complexity to the limit (same as index at a single timestamp). Extensive evaluations show that our solution achieves up to 138.1× higher QPS at 99% recall compared to prior proposals in TANNS query processing. Our future work will focus on time interval query for vectors with temporal attributes, which can be used in financial or recommendation applications to retrieve relevant content valid during a specific time period.

Acknowledgment

This work was partially supported by National Key Research and Development Program of China under Grant No. 2023YFF0725103, National Science Foundation of China (NSFC) (Grant Nos. 62425202, U21A20516, 62336003), the Beijing Natural Science Foundation (Z230001), the Fundamental Research Funds for the Central Universities No. JK2024-03, the Didi Collaborative Research Program and the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE). Zimu Zhou's research is supported by Chow Sang Sang Group Research Fund No. 9229139. Yongxin Tong is the corresponding author.

References

[1] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu *et al.*, "Milvus: A purpose-built vector data management system," in *SIGMOD*, 2021.

[2] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," *The VLDB Journal*, 2024.

[3] A. Asai, S. Min, Z. Zhong, and D. Chen, "Retrieval-based language models and applications," in *ACL*, 2023.

[4] J. Jang, S. Ye, C. Lee, S. Yang, J. Shin, J. Han, G. Kim, and M. Seo, "Temporalwiki: A lifelong benchmark for training and evaluating ever-evolving language models," in *EMNLP*, 2022.

[5] R. Campos, G. Dias, A. M. Jorge, and A. Jatowt, "Survey of temporal information retrieval and related applications," *ACM Computing Surveys*, 2014.

[6] Y. Qiu, Z. Zhao, Y. Ziser, A. Korhonen, E. Ponti, and S. B. Cohen, "Are large language model temporally grounded?" in *NAACL*, 2024.

[7] J. Wallat, A. Jatowt, and A. Anand, "Temporal blind spots in large language models," in *WSDM*, 2024.

[8] D. Schumacher, F. Haji, T. Grey, N. Bandlamudi, N. Karnik, G. U. Kumar, J. C.-Y. Chiang, P. Rad, N. Vishwamitra, and A. Rios, "Context matters: An empirical study of the impact of contextual information in temporal question answering systems," *arXiv preprint arXiv:2406.19538*, 2024.

[9] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *EMNLP*, 2020.

[10] B. Dhingra, J. R. Cole, J. M. Eisenschlos, D. Gillick, J. Eisenstein, and W. W. Cohen, "Time-aware language models as temporal knowledge bases," *Transactions of the Association for Computational Linguistics*, 2022.

[11] G. Izacard, P. Lewis, M. Lomeli, L. Hosseini, F. Petroni, T. Schick, J. Dwivedi-Yu, A. Joulin, S. Riedel, and E. Grave, "Atlas: Few-shot learning with retrieval augmented language models," *Journal of Machine Learning Research*, 2023.

[12] M. A. Nascimento and J. R. Silva, "Towards historical r-trees," in *SAC*, 1998.

[13] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," in *SIGMOD*, 2000.

[14] Y. Tao and D. Papadias, "The mv3r-tree: A spatio-temporal access method for timestamp and interval queries," in *VLDB*, 2001.

[15] Y. Tao and D. Papadias, "Efficient historical r-trees," in *SSDBM*, 2001.

[16] S. Arya, D. M. Mount, and O. Narayan, "Accounting for boundary effects in nearest neighbor searching," in *SoCG*, 1995.

[17] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *VLDB*, 1996.

[18] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC*, 1998.

[19] M. Wang, L. Lv, X. Xu, Y. Wang, Q. Yue, and J. Ni, "An efficient and robust framework for approximate nearest neighbor search with attribute constraint," in *NeurIPS*, 2023.

[20] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan *et al.*, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," in *WWW*, 2023.

[21] C. Zuo, M. Qiao, W. Zhou, F. Li, and D. Deng, "Serf: Segment graph for range-filtering approximate nearest neighbor search," in *SIGMOD*, 2024.

[22] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[23] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," in *PVLDB*, 2021.

[24] C. Chen, C. Jin, Y. Zhang, S. Podolsky, C. Wu, S.-P. Wang, E. Hanson, Z. Sun, R. Walzer, and J. Wang, "Singlestore-v: An integrated vector database system in singlestore," in *PVLDB*, 2024.

[25] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "Multidimensional access methods: Trees have grown everywhere," in *VLDB*, 1997.

[26] 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 Transactions on Knowledge and Data Engineering*, 2019.

[27] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," in *SISAP*, 2017.

[28] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *CVPR*, 2016.

[29] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys*, 1999.

[30] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014.

[31] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, 2014.

[32] M. Iwasaki, "Pruned bi-directed k-nearest neighbor graph for proximity search," in *SISAP*, 2016.

[33] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: a hybrid analytical engine towards query fusion for structured and unstructured data," in *PVLDB*, 2020.

[34] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang *et al.*, "Spfresh: Incremental in-place update for billion-scale vector search," in *SOSP*, 2023.

[35] H. Edelsbrunner, "A new approach to rectangle intersections," *International Journal of Computer Mathematics*, 1983.

[36] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010.

[37] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "Acorn: Performant and predicate-agnostic search over vector embeddings and structured data," in *SIGMOD*, 2024.

[38] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," *arXiv preprint arXiv:2401.08281*, 2024.

[39] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, 1999.

[40] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *PVLDB*, 2007.

[41] 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," in *PVLDB*, 2013.

[42] Y. Tian, X. Zhao, and X. Zhou, "Db-lsh 2.0: Locality-sensitive hashing with query-based dynamic bucketing," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[43] J. Wei, B. Peng, X. Lee, and T. Palpanas, "Det-lsh: A locality-sensitive hashing scheme with dynamic encoding tree for approximate nearest neighbor search," in *PVLDB*, 2024.

[44] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[45] Y. Matsui, Y. Uchida, H. Jegou, and S. Satoh, "A survey of product quantization," *ITE Transactions on Media Technology and Applications*, 2018.

[46] D. Xu, I. W. Tsang, and Y. Zhang, "Online product quantization," *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[47] J. Paparrizos, I. Edian, C. Liu, A. J. Elmore, and M. J. Franklin, "Fast adaptive similarity search through variance-aware quantization," in *ICDE*, 2022.

[48] J. Gao and C. Long, "Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search," in *SIGMOD*, 2024.

[49] I. Azizi, K. Echihabi, and T. Palpanas, "Graph-based vector search: An experimental evaluation of the state-of-the-art," in *SIGMOD*, 2025.

[50] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," in *PVLDB*, 2019.

[51] Y. Peng, B. Choi, T. N. Chan, J. Yang, and J. Xu, "Efficient approximate nearest neighbor search in multi-dimensional databases," in *SIGMOD*, 2023.

[52] 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," in *PVLDB*, 2024.

[53] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie, "Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment," in *SIGMOD*, 2024.

[54] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search," *arXiv preprint arXiv:2105.09613*, 2021.

[55] Y. Matsui, R. Hinami, and S. Satoh, "Reconfigurable inverted index," in *MM*, 2018.

[56] J. Mohoney, A. Pacaci, S. R. Chowdhury, A. Mousavi, I. F. Ilyas, U. F. Minhas, J. Pound, and T. Rekatsinas, "High-throughput vector similarity search in knowledge graphs," in *SIGMOD*, 2023.

[57] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang *et al.*, "Vbase: Unifying online vector similarity search and relational queries via relaxed monotonicity," in *OSDI*, 2023.

[58] C. Han, S. Kim, and H.-M. Park, "Efficient proximity search in time-accumulating high-dimensional data using multi-level block indexing." in *EDBT*, 2024.

[59] Y. Cai, J. Shi, Y. Chen, and W. Zheng, "Navigating labels and vectors: A unified approach to filtered approximate nearest neighbor search," in *SIGMOD*, 2025.

[60] Y. Xu, J. Gao, Y. Gou, C. Long, and C. S. Jensen, "irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search," in *SIGMOD*, 2025.

[61] C. S. Jensen and R. T. Snodgrass, "Temporal database," in *Encyclopedia of Database Systems*, 2018.

[62] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," *The VLDB Journal*, 1996.

[63] R. Zhang and M. Stradling, "The hv-tree: a memory hierarchy aware version index," in *PVLDB*, 2010.

[64] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May, "Timeline index: a unified data structure for processing queries on temporal data in sap hana," in *SIGMOD*, 2013.

[65] G. Christodoulou, P. Bouros, and N. Mamoulis, "Lit: Lightning-fast in-memory temporal indexing," in *SIGMOD*, 2024.