

# Efficient Proximity Search in Time-accumulating High-dimensional Data using Multi-level Block Indexing

Changhun Han  
Kookmin University  
Republic of Korea  
codingnoye@kookmin.ac.kr

Suji Kim  
Kookmin University  
Republic of Korea  
suji2924@kookmin.ac.kr

Ha-Myung Park  
Kookmin University  
Republic of Korea  
hmpark@kookmin.ac.kr

## ABSTRACT

How can we efficiently index extensive high-dimensional vector data increasing over time, enabling quick and accurate proximity searches within designated time windows? A time-restricted  $k$ -nearest neighbor (TkNN) query aims to identify the  $k$ -nearest vectors to a query vector within a specified time window. While high-dimensional and time-accumulating data are ubiquitous and managing such data efficiently is becoming increasingly significant, TkNN search within this context has not received much attention so far. In this paper, we propose Multi-level Block Indexing (MBI), a tailored indexing method for efficient approximate TkNN search. MBI employs an incremental hierarchical index structure that divides the data into multiple blocks based on timestamps. This structure ensures efficient query processing, irrespective of the length of the query time window, and facilitates the addition of new data over time. Experimental results highlight MBI's superiority over conventional methods, achieving query processing speeds up to 10.88 times faster and offering logarithmic scaling in data insertion time as the data volume grows.

## 1 INTRODUCTION

How can we index extensive and time-accumulating vector data to enable fast and accurate proximity search in specific time windows? Time-accumulating data, which grow over time fastly, are ubiquitous. For example, GK2A, a weather satellite of South Korea, takes 30 high-resolution pictures of the Korean Peninsula every hour for weather observation<sup>1</sup>; more than 60,000 new tracks are ingested by Spotify every day<sup>2</sup>; more than 500 hours of new video content is uploaded to YouTube every minute<sup>3</sup>; and 95 million photos and videos are shared on Instagram per day<sup>4</sup>.

Proximity search is a well-known classical problem of finding data close (or similar) to a query. One popular proximity search is  $k$ -nearest neighbor ( $k$ NN) search, which aims to find the  $k$  data points closest to the query point. In  $k$ -nearest neighbor search, data is typically represented as a set of points in a high-dimensional space, where the proximity properties of the original data are maintained to enable efficient distance computation. One way to perform  $k$ NN search is to compute the distances to all data points from the query point and select the  $k$  closest ones exactly. With large datasets, this exhaustive distance computation becomes impractical. To overcome this challenge, several recent

studies [10, 14, 16, 19, 27] have proposed indexing methods for fast and approximate  $k$ NN search.

However, the aforementioned studies do not consider the temporal information of the data, and therefore do not support time-restricted  $k$ NN (TkNN) search, such as “Which 5 movies released between 1980 and 1995 are most similar to Zootopia?” or “Which 10 photos you took between January 2010 and May 2011 are most similar to the one you just took?” One way to handle TkNN queries using the above indexing methods is to perform  $k$ NN search on the entire dataset and filter the results to include only those within the time window. However, this method cannot guarantee that the number of search results is  $k$  and may even output nothing. Searching can continue until the desired number of results is obtained, but this approach may be time-consuming when the time window is short because most of the results have to be filtered out.

In this paper, we propose *Multi-level Block Indexing* (MBI), a novel indexing method for efficient approximate TkNN search on time-accumulating data with timestamps. MBI aims to ensure efficient processing of any TkNN query, regardless of whether its time window is long or short, and to ensure efficient addition of new data that continuously occurs over time to the index. For this purpose, we devise an expandable hierarchical indexing structure where the data is divided into multiple blocks based on timestamps. Then, MBI efficiently processes queries by selecting blocks that match the query time window. Our contributions are summarized as follows:

- **Method.** We propose MBI, a new indexing algorithm for efficient approximate TkNN search on time-accumulating data with timestamps.
- **Analysis.** We theoretically analyze the efficiency of MBI in terms of index size, indexing time complexity and query time complexity.
- **Experiments.** The experimental results show that query processing of MBI is up to 10.88 times faster than that of the simple methods, and the data insertion time scales logarithmically with the total size of the data.

The codes and datasets used in this paper are available at <https://github.com/tknn2023/mbi>. Frequently used symbols are listed in Table 1.

## 2 RELATED WORK

The problem of approximate  $k$ -nearest neighbor ( $k$ NN) search has been widely studied in the literature, and various approaches have been proposed to tackle it. However, the problem of approximate time-restricted  $k$ NN (TkNN) search has not received much attention so far. In this section, we review some of the relevant works on approximate  $k$ NN search and TkNN search.

<sup>1</sup><https://nmsc.kma.go.kr/enhome/html/base/cmm/selectPage.do?page=satellite.gk2a.intro>

<sup>2</sup><https://www.musicbusinessworldwide.com/over-60000-tracks-are-now-uploaded-to-spotify-daily-thats-nearly-one-per-second/>

<sup>3</sup><https://www.globalmediainsight.com/blog/youtube-users-statistics/>

<sup>4</sup><https://earthweb.com/how-many-pictures-are-on-instagram/>

**Table 1: Table of symbols**

Symbol	Definition
$\mathcal{T}$	Set of timestamps
$\mathcal{D}$	Spatio-temporal database
$(v, t)$	Timestamped vector where $v \in \mathbb{R}^d$ is a vector and $t \in \mathcal{T}$ is a timestamp
$d$	Spatial dimension of $\mathcal{D}$
$\mathcal{D}[t_a : t_b]$	Subset of $\mathcal{D}$
$S_L$	Leaf block size in MBI
$\tau$	Threshold for selecting blocks in MBI
$\mathcal{B}_i = (\mathcal{D}_i, \mathcal{G}_i)$	Block of index $i$ in MBI where $\mathcal{D}_i$ is its the timestamped vector set and $\mathcal{G}_i$ is its graph based index for TkNN queries.
$\mathcal{B}_{i.t_s}, \mathcal{B}_{i.t_e}$	The earliest and the latest timestamp of vectors in $\mathcal{B}_i$
$q = (w, k, t_s, t_e)$	TkNN query where $w$ is a vector, $k$ is the number of query results, and $t_s$ and $t_e$ are the start and the end timestamps of the query time window.

## 2.1 Approximate $k$ NN Search

As exact  $k$ NN search can be computationally intensive, especially when dealing with high-dimensional datasets, various approximate  $k$ NN search methods have been studied and developed to overcome this challenge. These methods aim to find an approximation of the  $k$ -nearest neighbors of a given query point, while minimizing the computational cost. One key idea behind these methods is to build a data structure that allows efficient and fast search of the  $k$ -nearest neighbors.

We categorize these methods based on their underlying structures and approaches: tree-based, graph-based, hashing-based, and quantization-based methods. Tree-based methods including KD-Tree [11], R\*-Tree [6], Balltree [31], Cover trees [7], RPTree [9], and VP-Tree [43] use recursive partitioning to organize the data. RPFforest [42] and FLANN [30] utilize multiple trees to improve the search efficiency and reduce the effects of data distribution or randomization on the accuracy of the search results. Graph-based methods connect nearby data points using various types of graph structures, such as  $k$ -nearest neighbor graphs ( $k$ NN graph) [32], Delaunay graphs (DG) [4], relative neighborhood graphs (RNG) [36], and minimum spanning trees (MST). Examples of the graph-based methods include NSW [26], HNSW [27], NGT [19], NNDescent [10], OL-Graph [45], FANNG [18], and Efanna [12]. A comprehensive survey that covers various graph-based methods is available in [40]. Hashing-based methods, such as LSH [15] and PUFFINN [3], groups close points together using locality sensitive hash functions. Quantization-based methods, including IVFADC [23], QuickADC [1], and ScaNN [16], represent each data point as a low-dimensional, discretized vector, which allows for fast distance computations by comparing the discretized vectors. Meanwhile, some studies improve the search performance through modern hardware support. SPANN [8] and DiskANN [20] use external memory to address the issue of memory shortage that can occur when processing large datasets. Faiss [22], PQT [41], and SONG [44] exploit GPUs to accelerate the search process. QuickerADC [2] increases the search speed of QuickADC by using SIMD instruction sets.

As such, a wide range of approximate  $k$ NN search methods have been developed, among which graph-based and quantization-based methods have demonstrated the state-of-the-art performance<sup>5</sup>. However, applying these methods directly to time-restricted  $k$ NN search is challenging. Creating indices for the entire dataset without considering timestamps, existing approximate  $k$ NN methods are hard to efficiently filter data that falls within a specific time window, which is exactly what TkNN requires. Meanwhile, those methods could be modified to address the TkNN problem by continuing the search process until the query results within the query time window reach  $k$  entries. However, this approach leads to a significant performance degradation when the query time window is short, as it requires exploring a large portion of the data. We discuss this modification further in Section 3.2.2. In this paper, we propose a novel method that empowers these methods to efficiently perform time-restricted  $k$ NN search, irrespective of the length of the query time window. We note that our method uses existing approximate  $k$ NN search methods as a module.

## 2.2 Time-Restricted $k$ NN (TkNN) Search

In the context of time-restricted  $k$ NN search, the problem becomes more complex since we need to take into account both the spatial and temporal dimensions of the data. TkNN search has been studied mainly in the fields of geoinformatics and databases with the purpose of performing queries on objects moving over time. Accordingly, most existing research assumes low dimensions, such as two or three. A simple approach is to treat time as a new spatial axis and apply tree structures such as R\*-Tree or Quadtree [24, 35, 38]. In this approach, TkNN search can be performed by conducting the range search on the time axis and the  $k$ NN search on the remaining axes. Similarly, PPR-Tree [17, 24] is specifically designed for moving objects and links ephemeral R-Trees in a directed acyclic graph (DAG) format over time for TkNN search. DISTIL+ [29] is based on the Quadtree structure and utilizes distributed clusters to process large datasets. However, none of the aforementioned methods are well-suited for TkNN search in high-dimensional data, as the underlying tree structures become inefficient due to the curse of dimensionality [28]. Consequently, it becomes inevitable to explore almost all vectors within the query time window when dealing with high-dimensional data. Meanwhile, TkNN search in high-dimensional data is becoming essential with the advancement of artificial intelligence as various types of data such as photos, music, and documents are represented as high-dimensional data [25, 37, 39]. In this paper, we propose a new method that performs approximate TkNN search efficiently even in high-dimensional data.

## 3 PRELIMINARIES

In this section, we provide a formal definition of the time-restricted  $k$ -nearest neighbor (TkNN) search, which is the problem we address in this paper. Following that, we introduce two straightforward approaches for TkNN and show their limitations.

### 3.1 Problem Definition

Before defining the TkNN search, let us first introduce several necessary terms and symbols. A timestamped vector  $(v, t)$  is a pair consisting of a vector  $v \in \mathbb{R}^d$  and a timestamp  $t \in \mathcal{T}$  where  $d$  is the dimension of the vector space and  $\mathcal{T}$  is a set of timestamps. Any two timestamps in  $\mathcal{T}$  are comparable; for  $t_a, t_b \in \mathcal{T}$ ,  $t_a < t_b$

<sup>5</sup><https://github.com/erikbern/ann-benchmarks>

---

**Algorithm 1:** BSBF Query Process

---

**Input:** A TkNN query  $q = (w, k, t_s, t_e)$ , a database  $\mathcal{D}$  in order of increasing timestamp, a distance function  $\sigma$

**Output:** TkNNs of  $q$

- 1  $\mathcal{D}[t_s : t_e] \leftarrow \text{BinarySearch}(t_s, t_e, \mathcal{D})$
  - 2 **return**  $\text{BruteForce}(w, k, \sigma, \mathcal{D}[t_s : t_e])$
- 

indicates that  $t_a$  precedes  $t_b$ . A database  $\mathcal{D} \subseteq \mathbb{R}^d \times \mathcal{T}$  is a set of timestamped vectors. For  $t_a, t_b \in \mathcal{T}$  such that  $t_a < t_b$ , we denote by  $\mathcal{D}[t_a : t_b]$  a subset of  $\mathcal{D}$  whose vectors have timestamps between  $t_a$  and  $t_b$ , i.e.,  $\mathcal{D}[t_a : t_b] = \{(v, t) \in \mathcal{D} \mid t_a \leq t < t_b\}$ . The proximity of two vectors is defined as a distance function  $\sigma : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+$ , and we denote by  $\sigma(u, v)$  the distance between vectors  $u$  and  $v$ . Any distance measure including the euclidean distance can be used for  $\sigma$  depending on datasets and applications. Then, we define a TkNN query as follows:

*Definition 3.1.* A TkNN query  $q = (w, k, t_s, t_e)$  on database  $\mathcal{D}$  is to identify a subset  $A$  of  $\mathcal{D}$  such that:

$$A = \arg \min_{X \subseteq \mathcal{D}[t_s : t_e], |X|=k} \sum_{(v, t) \in X} \sigma(w, v)$$

where  $w \in \mathbb{R}^d$  is the query vector,  $k$  is the number of timestamped vectors to identify, and  $t_s, t_e \in \mathcal{T}$  are the start and end timestamps of the query time window.

To measure the quality of an approximate answer  $\hat{A}$  compared to the true answer  $A$ , we use  $\text{recall}@k$  that is defined as follows:

$$\text{recall}@k(\hat{A}, A) = \frac{|\hat{A} \cap A|}{k}$$

For the sake of simplicity, we assume that all vectors have distinct timestamps. Even if there are vectors with identical timestamps, as in real-world datasets, our approach can be easily adapted. For vectors sharing the same timestamp, we arbitrarily assign an order. During the query process, we set the query range from the earliest ordered vector with the start timestamp to the last ordered vector with the end timestamp.

## 3.2 Two simple approaches for TkNN search

We introduce two straightforward approaches to handle TkNN queries and their limitations.

**3.2.1 Binary Search and Brute-Force (BSBF).** Binary Search and Brute-Force, shortly BSBF, is a method that combines binary search and brute-force methods for TkNN search. Algorithm 1 shows the pseudocode for BSBF's query process. BSBF sorts all timestamped vectors in database  $\mathcal{D}$  in order of increasing timestamp and uses the sorted database as its index structure. Given a TkNN query  $q = (w, k, t_s, t_e)$ , BSBF identifies  $\mathcal{D}[t_s : t_e]$  by performing binary search process and finds the  $k$  vectors in  $\mathcal{D}[t_s : t_e]$  that are closest to  $w$  using a brute-force method. The query process of BSBF is very fast when the query time window is short. Let  $n$  and  $m$  be the sizes of  $\mathcal{D}$  and  $\mathcal{D}[t_s : t_e]$ , respectively. The query process of BSBF requires  $O(\log n)$  to identify  $\mathcal{D}[t_s : t_e]$  and  $O(m \log k)$  to find  $k$  nearest points if a max-heap of size  $k$  is used for the brute-force method. This suggests that when  $m$  is close to  $n$ , BSBF is significantly inefficient because it needs to examine nearly all the timestamped vectors in  $\mathcal{D}$ .

---

**Algorithm 2:** Graph-based SF Query Process

---

**Input:** A TkNN query  $q = (w, k, t_s, t_e)$ , a graph  $\mathcal{G}$  of a database  $\mathcal{D}$ , a distance function  $\sigma$ , a maximum candidate set size  $M_C$ , a parameter  $\epsilon$  for controlling the search range

**Output:** Approximate TkNNs of  $q$

- 1  $s \leftarrow$  a randomly sampled timestamped vector from  $\mathcal{D}$
  - 2 Initialize a candidate set  $C \leftarrow \{s\}$
  - 3 Initialize a visited set  $V \leftarrow \emptyset$
  - 4 Initialize a result set  $R \leftarrow \emptyset$
  - 5 **while**  $C \setminus V \neq \emptyset$  **do**
  - 6      $p' = (v', t') \leftarrow \arg \min_{(v, t) \in C \setminus V} \sigma(v, w)$
  - 7      $V \leftarrow V \cup \{p'\}$
  - 8     **if**  $|R| < k$  **then**
  - 9          $C \leftarrow C \cup \{\text{neighbors}(v'', t'') \text{ of } p' \text{ in } \mathcal{G}\}$
  - 10     **else**
  - 11          $C \leftarrow C \cup \{\text{neighbors}(v'', t'') \text{ of } p' \text{ in } \mathcal{G} \text{ such that } \sigma(v'', w) < \epsilon \cdot \max_{(v, t) \in R} \sigma(v, w)\}$
  - 12     **if**  $t_s \leq t' < t_e$  **then**
  - 13          $R \leftarrow R \cup \{p'\}$
  - 14         **if**  $|R| > k$  **then**
  - 15             update  $R$  to retain  $k$  nearest timestamped vectors to  $w$
  - 16     **if**  $|C| > M_C$  **then**
  - 17         update  $C$  to retain  $M_C$  nearest timestamped vectors to  $w$
  - 18 **return**  $R$
- 

**3.2.2 Search and Filtering (SF).** Another approach for handling TkNN queries is Search-and-Filtering, or SF, which uses an existing approximate  $k$ NN method [19] with the following minor modification: it continues searching until it identifies  $k$  or more vectors within the query time window. Algorithm 2 provides the pseudocode for handling a TkNN query in a proximity graph  $\mathcal{G}$  using SF. Lines 12-15 are mainly different from the existing algorithm. The proximity graph  $\mathcal{G}$  is built using any proximity based indexing techniques like NNDescent to connect spatially close vertices, where each vertex corresponds to a vector in  $\mathcal{D}$  without timestamps. This algorithm starts at a random vertex  $s$  and traverses the graph towards the query vector  $w$ . During the traversal, it filters and builds a result set  $R$  from vectors that fall within the query time window. SF is effective when the query time window is long enough to cover almost all vectors in the database  $\mathcal{D}$ . However, it becomes inefficient when the query time window is short. This is because a short query time window only includes a few vectors, and most of the vectors identified during the search do not make it into the results, which significantly expands the search area.

## 4 PROPOSED METHOD

In this section, we propose Multi-level Block Indexing (MBI), a new incremental indexing method for TkNN queries that works efficiently for queries of any time windows. The technical challenges to efficiently implement MBI and our key ideas to resolve them are as follows:

- C1. How can we make the query process remain efficient regardless of the length of the query time window?** When the query time window is long, BSBF is inefficient

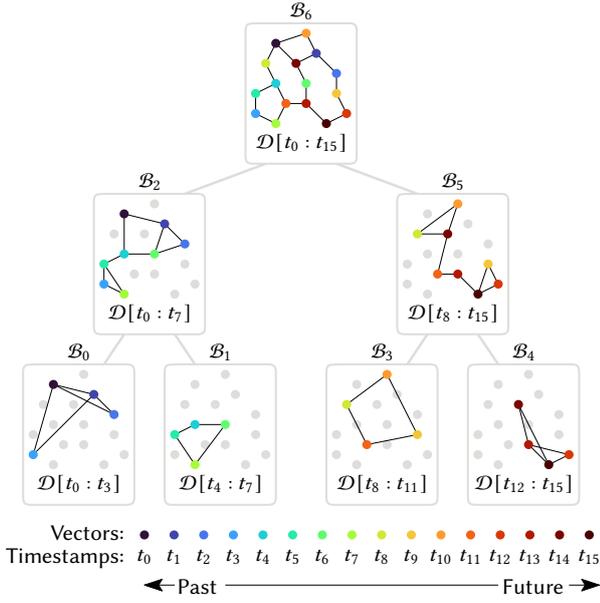


Figure 1: An example of the hierarchical index structure of MBI.

because it needs to check all the vectors in the time window. On the other hand, when the query time window is short, SF becomes inefficient as it must expand the search area until the targeted number of vectors within the time window is detected. **To ensure the efficiency for queries of any time window, we divide the database into multiple blocks based on timestamps and organize them in a hierarchical structure (Section 4.1).** Accordingly, MBI operates like BSBF when the query time window is short, and like SF when the query time window is long.

- C2. How can we efficiently add new vectors that occur over time to MBI?** Data containing timestamps usually accumulates continuously over time. Meanwhile, it's not straightforward to insert new data while maintaining the sophisticated hierarchical structure of MBI. **We enable efficient incremental construction of MBI by bottom-up block merging and postorder block numbering (Section 4.2).**
- C3. How should we select a set of blocks to process the query most efficiently?** The query processing of MBI involves finding a set of time-disjoint blocks that covers all vectors within the query time window, performing TkNN queries independently in each block, and then combining the results. The performance of the query processing depends on the selected set of blocks; if we only select small blocks, it becomes inefficient when the query time window is long, similar to BSBF, and if we only select large blocks, it becomes inefficient when the query time window is short, similar to SF. **We propose a top-down block selection method that efficiently finds a mixed set of both large and small blocks suitable for each query (Section 4.3).**

In the following subsections, we first provide an overview of MBI in Section 4.1 and describe the details of the construction and the query process of MBI in Section 4.2 and 4.3, respectively. We also provide several theoretical analyses on MBI in Section 4.4.

## 4.1 Overview

MBI is a binary tree of blocks, each containing all vectors of a specific time window and an index to efficiently handle TkNN queries on these vectors. While any index structure for efficient kNN search can be used for the index, we employ one of the graph based indexing methods, as they have shown the best search performance. Each block in MBI has the left and right children, each having half of the vectors in the parent block. The left child has the half with earlier timestamps, while the right child does the half with later timestamps. The root block holds all vectors, and each leaf block holds  $S_L$  or fewer vectors. Figure 1 shows an example of MBI. The database  $\mathcal{D} = \mathcal{D}[t_0 : t_{15}]$  contains 16 vectors, each with a timestamp from  $t_0$  to  $t_{15}$ . If  $i < j$ , then  $t_i < t_j$ . The leaf size  $S_L$  is 4. The vectors are color-coded based on the order of their timestamps.

MBI enables efficient query processing by searching only in a few selected blocks that overlap with the query's time window. For example, in Figure 1, a query with time window ( $t_s = t_5, t_e = t_7$ ) can be efficiently processed by considering only block  $\mathcal{B}_1$  whose vector set is  $\mathcal{D}[t_4 : t_7]$ . Similarly, a query with time window ( $t_s = t_1, t_e = t_{15}$ ) can be efficiently processed with  $\mathcal{B}_6$ . Meanwhile, a query with time window ( $t_s = t_4, t_e = t_{14}$ ) can be processed with different sets of blocks:  $\{\mathcal{B}_1, \mathcal{B}_5\}$ ,  $\{\mathcal{B}_1, \mathcal{B}_3, \mathcal{B}_4\}$ , or  $\{\mathcal{B}_6\}$ . We will explain our strategy for finding a set of blocks for efficient query processing in Section 4.3.

## 4.2 Multi-level Block Indexing

Multi-level Block Indexing (MBI) incrementally appends vectors in the order of increasing timestamps, that is, a new vector has a later timestamp than all existing vectors in MBI. When a new vector arrives, MBI inserts the vector into the latest leaf block. If the leaf block is full (i.e., it contains  $S_L$  vectors already), MBI creates a new leaf block and inserts the vector into it. In this case, MBI also creates virtual blocks to maintain the perfect binary tree structure. Figure 2 shows an example of adding a new vector  $v_9$  into MBI that contains 8 vectors and the leaf size  $S_L$  is 4. Because the latest leaf block  $\mathcal{B}_1$  is full, MBI creates a new leaf block  $\mathcal{B}_3$  and inserts  $v_9$  into it. The virtual blocks are created to maintain the perfect binary tree structure. If a new vector arrives again, it is added into  $\mathcal{B}_3$  as  $\mathcal{B}_3$  is the latest and not full. If the new vector makes the latest leaf block full, then MBI builds the graph-based kNN index for the leaf block and performs **bottom-up block merging** to build the graph-based kNN indices for ancestor blocks of the leaf block. Figure 3 shows an example of the bottom-up block merging process that occurs when  $v_{15}$  is added to MBI with vectors  $v_0$  to  $v_{14}$ .  $S_L$  is 4. In this example,  $v_{15}$  is added to  $\mathcal{B}_4$  as  $\mathcal{B}_4$  is the latest and not full. As  $v_{15}$

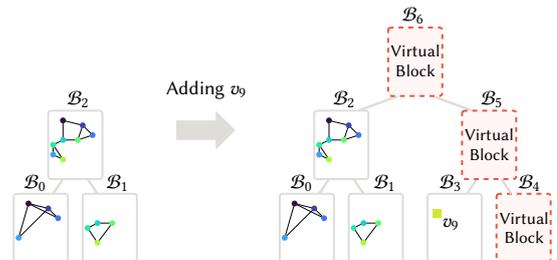


Figure 2: An example of adding a new vector into MBI whose leaf blocks are full.

---

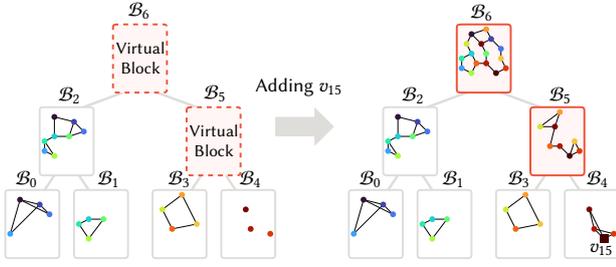
**Algorithm 3: MBI's Vector Insertion**


---

**Input:** MBI, a timestamped vector  $(v, t)$

- 1 Let  $i$  be the index of the first non-full leaf block in MBI
- 2 Let  $\mathcal{B}_i = (\mathcal{D}_i, \mathcal{G}_i)$  be the block at index  $i$  where  $\mathcal{D}_i$  is the vector set and  $\mathcal{G}_i$  is the graph-based  $k$ NN index, which is initially empty.
- 3  $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{(v, t)\}$
- 4 **if**  $|\mathcal{D}_i| = S_L$  **then**
- 5      $\mathcal{G}_i \leftarrow \text{BuildKNNIndex}(\mathcal{D}_i)$
- 6     Let  $j$  be the number of leaf blocks in MBI
- 7      $h \leftarrow 1$
- 8     **while**  $j$  is even **do**
- 9          $\mathcal{D}_{i+1} \leftarrow \mathcal{D}_{i+1-2^h} \cup \mathcal{D}_i$
- 10          $\mathcal{G}_{i+1} \leftarrow \text{BuildKNNIndex}(\mathcal{D}_{i+1})$
- 11          $\mathcal{B}_{i+1} \leftarrow (\mathcal{D}_{i+1}, \mathcal{G}_{i+1})$
- 12          $i \leftarrow i + 1$
- 13          $j \leftarrow j/2$
- 14          $h \leftarrow h + 1$

---



**Figure 3: An example of MBI's bottom-up block merging process.**

makes  $\mathcal{B}_4$  full, a  $k$ NN index is created, and the bottom-up block merging proceeds:  $\mathcal{B}_5$  is built from the vectors of  $\mathcal{B}_3$  and  $\mathcal{B}_4$ , and subsequently,  $\mathcal{B}_6$  is built from the vectors of  $\mathcal{B}_2$  and  $\mathcal{B}_5$ .

Algorithm 3 lists the pseudo-code to insert a new timestamped vector  $(v, t)$  into MBI. Let  $i$  be the index of the first non-full leaf block in MBI and  $\mathcal{B}_i = (\mathcal{D}_i, \mathcal{G}_i)$  be the block at index  $i$  where  $\mathcal{D}_i$  is the vector set and  $\mathcal{G}_i$  is the graph-based  $k$ NN index, which is initially empty (lines 1, 2). Then, the timestamped vector  $(v, t)$  is added to  $\mathcal{D}_i$  (line 3). If  $\mathcal{D}_i$  becomes full (i.e.,  $|\mathcal{D}_i| = S_L$ ), we construct a graph-based  $k$ NN index  $\mathcal{G}_i$  for  $\mathcal{D}_i$  and perform **bottom-up block merging** that involves consecutively creating ancestor blocks of  $\mathcal{B}_i$  (lines 4-14). Bottom-up block merging for  $\mathcal{B}_i$  proceeds as follows: *If  $\mathcal{B}_i$  is the right child of the prospective parent, we create the parent at index  $i + 1$ , i.e.,  $\mathcal{B}_{i+1}$  is the parent block of  $\mathcal{B}_i$ . The vector set of  $\mathcal{B}_{i+1}$  is the union of children's vector sets  $\mathcal{D}_{i+1-2^h}$  and  $\mathcal{D}_i$  and we build a  $k$ NN index  $\mathcal{G}_{i+1}$  where  $h$  is the height of the parent block in the tree. This process is repeated in the parent block.* Note that  $\mathcal{B}_i$ 's sibling block is at index  $i + 1 - 2^h$  as the blocks are numbered sequentially as they are created, which corresponds to the visiting sequence in a postorder traversal of MBI.

**Parallelization of MBI.** The bottom-up block merging process is easily parallelizable because it builds each block independently. For example, when  $v_{15}$  is added, MBI independently identifies the vector sets of  $\mathcal{B}_4$ ,  $\mathcal{B}_5$ , and  $\mathcal{B}_6$  by combining the vector sets of  $\mathcal{B}_2$ ,  $\mathcal{B}_3$ , and  $\mathcal{B}_4$ , then, the  $k$ NN indices  $\mathcal{G}_4$ ,  $\mathcal{G}_5$ , and  $\mathcal{G}_6$  are built in parallel.

---

**Algorithm 4: MBI Query Process**


---

**Input:** A TkNN query  $q = (w, k, t_s, t_e)$ , MBI, a distance function  $\sigma$

**Output:** An approximate TkNNs of  $q$

- 1 Let  $\mathcal{B}_r$  be the root block of MBI
- 2  $R \leftarrow \emptyset$
- 3  $S \leftarrow \text{BlockSelection}(\mathcal{B}_r, t_s, t_e)$
- 4 **foreach**  $\mathcal{B}' \in S$  **do**
- 5     **if**  $\mathcal{B}'$  is a non-full leaf block **then**
- 6          $R \leftarrow R \cup \text{BSBFQuery}(w, k, t_s, t_e, \mathcal{B}', \sigma)$   
        // Algorithm 1
- 7     **else**
- 8          $R \leftarrow R \cup \text{SFQuery}(w, k, t_s, t_e, \mathcal{B}', \sigma)$   
        // Algorithm 2
- 9      $R \leftarrow k$  neighbors nearest to  $w$  in  $R$
- 10 **return**  $R$
- 11 **Function**  $\text{BlockSelection}(\mathcal{B}_c, t_s, t_e)$ :
- 12     Let  $\mathcal{B}_c.t_s$  and  $\mathcal{B}_c.t_e$  be the earliest and the exclusive upper timestamps of vectors in  $\mathcal{B}_c$ , respectively.
- 13      $r_o \leftarrow \frac{\max(0, \min(\mathcal{B}_c.t_e, t_e) - \max(\mathcal{B}_c.t_s, t_s))}{\mathcal{B}_c.t_e - \mathcal{B}_c.t_s}$
- 14     **if**  $r_o = 0$  **then**
- 15         **return**  $\emptyset$
- 16     **else if**  $\mathcal{B}_c$  is a leaf block or  $r_o > \tau$  **then**
- 17         **return**  $\{\mathcal{B}_c\}$
- 18     **else**
- 19         Let  $c$  and  $h$  be the index and the height of  $\mathcal{B}_c$ , respectively.
- 20         **return**  $\text{BlockSelection}(\mathcal{B}_{c-2^h}, t_s, t_e) \cup \text{BlockSelection}(\mathcal{B}_{c-1}, t_s, t_e)$

---

### 4.3 Query Processing on MBI

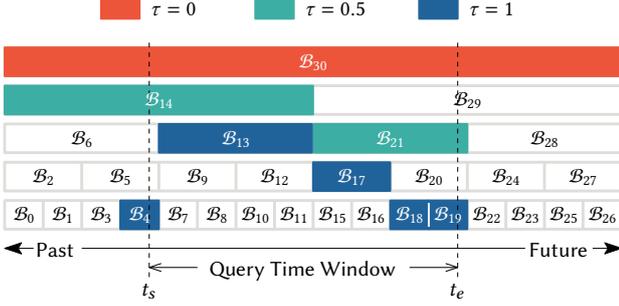
Algorithm 4 lists the pseudo-code of query processing on MBI. Given a TkNN query  $q = (w, k, t_s, t_e)$ , we first find a set of blocks, referred to as a *search block set*, that cover all vectors whose timestamps are between  $t_s$  and  $t_e$  (line 3). We then perform TkNN search for each block in the search block set and combine the results to derive the final TkNN result (lines 4-7). To choose the best search block set among all possible ones, we propose a **top-down block selection** approach. Let  $\mathcal{B}_c.t_s$  and  $\mathcal{B}_c.t_e$  be the earliest and the exclusive upper timestamp of vectors in the *current block*  $\mathcal{B}_c$ , respectively. We define the *overlap ratio*  $r_o(q, \mathcal{B}_c)$  of the query  $q$  in  $\mathcal{B}_c$  as follows:

$$r_o(q, \mathcal{B}_c) = \frac{\max(0, \min(\mathcal{B}_c.t_e, t_e) - \max(\mathcal{B}_c.t_s, t_s))}{\mathcal{B}_c.t_e - \mathcal{B}_c.t_s}$$

Then, the top-down block selection starts from the root block and proceeds according to the following three cases (lines 9-17):

- Case 1.** If  $r_o(q, \mathcal{B}_c)$  is 0,  $\mathcal{B}_c$  is not included in the search block set.
- Case 2.** If  $\mathcal{B}_c$  is a leaf block or  $r_o(q, \mathcal{B}_c)$  exceeds a threshold  $\tau$ , the block is included in the search block set.
- Case 3.** If  $\mathcal{B}_c$  is a non-leaf block and  $r_o(q, \mathcal{B}_c)$  is less than  $\tau$ , the block is not included in the search block set, and this process is repeated for its child blocks.

The value of  $\tau$  determines the composition of the search block set. As  $\tau$  approaches 0, blocks closer to the root are more likely to be selected, whereas as  $\tau$  approaches 1, the selection criteria for blocks becomes stricter, increasing the probability of selecting



**Figure 4: Examples of search block sets when  $\tau$  is 0, 0.5, and 1, respectively.**

leaf blocks. Figure 4 illustrates examples of search block sets corresponding to different  $\tau$  values. In this example, when  $\tau$  is 0, 0.5, and 1, the search block sets are  $\{\mathcal{B}_{30}\}$ ,  $\{\mathcal{B}_{14}, \mathcal{B}_{21}\}$ , and  $\{\mathcal{B}_4, \mathcal{B}_{13}, \mathcal{B}_{17}, \mathcal{B}_{18}, \mathcal{B}_{19}\}$ , respectively. A  $\tau$  value that is too low makes the query process performed only on the root block, significantly degrading the performance when the query time window is short as in the case of SF. On the other hand, a  $\tau$  value that is too high makes the query process performed on too many blocks, increasing the overhead, especially when the query time window is long. The optimal value of  $\tau$  varies according to the dataset. In Section 4.4.3, we prove that the number of blocks in which MBI performs a query is at most two if  $\tau \leq 0.5$ . Also, in Section 5.4, we experimentally show how  $\tau$  affects the search performance.

In the incremental process of adding vectors to MBI, it may not always form a complete tree as in the left tree of Figure 3. In such cases, to conduct the search as described above, we complete the tree with virtual blocks whose time window extends from  $-\infty$  to  $\infty$ . Throughout the top-down block selection process, these virtual blocks always fall into case 3 due to their infinite time window size. Consequently, they are never included in the search block set, providing opportunities for their descendants to be potentially included instead.

#### 4.4 Analysis

This section provides analytic results of MBI. The indexing methods employed for each block significantly influence the analytical results. Let  $\Phi(n)$  be the time complexity and  $\Psi(n)$  the index size of the indexing method for  $n$  vectors.

**4.4.1 Index size of MBI.** The total index size of MBI is the sum of each block:

$$\sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} 2^i \Psi(|\mathcal{D}|/2^i)$$

where  $|\mathcal{D}|$  is the number of vectors in database  $\mathcal{D}$ , and  $S_L$  is the leaf block size. The index size of a graph-based  $k$ NN index for  $n$  vectors is  $O(nk')$ , where  $k'$  is the average number of neighbors in the graph. Meanwhile,  $k'$  is usually set to be a constant value, so the required space can be seen as  $O(n)$ . Thus, indexing each block with a graph-based  $k$ NN index in MBI results in the following total index size:

$$\sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} 2^i O(|\mathcal{D}|/2^i) = \sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} O(|\mathcal{D}|) = O(|\mathcal{D}| \log |\mathcal{D}|)$$

**4.4.2 Indexing time complexity of MBI.** Since MBI indexes each block independently, the total indexing time is the sum of

the time required to build each block, as follows:

$$\sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} 2^i \Phi(|\mathcal{D}|/2^i)$$

Accordingly, the average time required to add a single vector to MBI, i.e., the amortized time complexity of inserting  $|\mathcal{D}|$ -th vector, is as follows:

$$\frac{1}{|\mathcal{D}|} \times \sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} 2^i \Phi(|\mathcal{D}|/2^i)$$

NNDescent, an approximate  $k$ NN graph build algorithm, is known to require  $O(n^{1.14})$  time to index  $n$  vectors by empirical analysis [10, 34]. Indexing each block as a  $k$ NN graph using NNDescent, the total index time is as follows:

$$\sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} 2^i O((|\mathcal{D}|/2^i)^{1.14}) \leq \sum_{i=0}^{\log \frac{|\mathcal{D}|}{S_L}} O(|\mathcal{D}|^{1.14}) = O(|\mathcal{D}|^{1.14} \log |\mathcal{D}|)$$

Similarly, the amortized vector insertion time for inserting  $|\mathcal{D}|$ -th vector is then  $O(|\mathcal{D}|^{0.14} \log |\mathcal{D}|)$ .

**4.4.3 Query time complexity of MBI.** We analyze the query time complexity of MBI by dividing it into two cases: when  $\tau \leq 0.5$  and when  $\tau > 0.5$ . The following lemma is for analyzing the query time complexity when  $\tau \leq 0.5$ , indicating that the maximum number of blocks processed in this case is two.

**LEMMA 4.1.** *In MBI, if  $\tau \leq 0.5$ , the maximum number of blocks processing a query is 2.*

**PROOF.** Consider a query with a time window of  $(t_s, t_e)$ . Let  $\mathcal{B}_p$  be the first block such that its left child block  $\mathcal{B}_l$  and right child block  $\mathcal{B}_r$  partition the query time window into two. If the ratio of the query time window to the total time window of  $\mathcal{B}_p$  exceeds  $\tau$ , the query is processed in the single block  $\mathcal{B}_p$ , thereby satisfying the lemma trivially. If the ratio is less than  $\tau$ , the query time window is divided into two,  $(t_s, t_m)$  and  $(t_m, t_e)$ , by  $\mathcal{B}_l$  and  $\mathcal{B}_r$  respectively. We first show that, for the time window  $(t_m, t_e)$ , the query is processed either in  $\mathcal{B}_r$  or in precisely one of its descendants. We define  $\alpha$  as the ratio of the query time window  $(t_m, t_e)$  to the total time window of  $\mathcal{B}_r$ . If  $\alpha \geq \tau$ , the query is processed in  $\mathcal{B}_r$ . If  $\alpha < \tau$ , the query time window  $(t_m, t_e)$  does not overlap with the right child of  $\mathcal{B}_r$ . This is because the right child of  $\mathcal{B}_r$ 's time window starts from the midpoint of  $\mathcal{B}_r$ , and  $\tau < 0.5$ . In such cases, we repeat this process in the left child of  $\mathcal{B}_r$ . Accordingly, for the time window  $(t_m, t_e)$ , the query is processed either in  $\mathcal{B}_r$  if  $\alpha \geq \tau$  or in the  $i$ -th left child block of  $\mathcal{B}_r$  if  $\tau/2^i \leq \alpha < \tau/2^{i-1}$  for any integer  $i \geq 0$ . As the query time window  $(t_s, t_m)$  is symmetric to  $(t_m, t_e)$ , it is processed in  $\mathcal{B}_l$  or in precisely one of its descendants in a similar manner, and the lemma holds.  $\square$

Now, we prove the query time complexity in MBI when  $\tau \leq 0.5$  in the following theorem.

**THEOREM 4.2.** *The query time complexity in MBI is  $O(\log |\mathcal{D}[t_s : t_e]|/\tau + k/\tau)$  when  $\tau \leq 0.5$  and the query time window is  $(t_s, t_e)$  where  $|\mathcal{D}[t_s : t_e]|$  is the number of vectors in  $\mathcal{D}[t_s : t_e]$ .*

**PROOF.** For  $n$  vectors, the time complexity of graph-based 1NN search is  $O(\log n)$  [13], and for  $k$ NN search, the time complexity is  $O(\log n + k)$  considering that  $k$  searches are performed. Let  $m$  be the number of vectors within the query time window. We

assume that both  $m$  and  $n$  are significantly larger than  $k$ . Then, the time complexity of Search and Filtering (SF) for TkNN query is  $O(\log n + kn/m)$  as the expected number of vectors to be checked is  $kn/m$ .

The query time window covers at least  $\tau$  of every block processing the query in MBI. The number of vectors in each block is less than or equal to  $|\mathcal{D}[t_s : t_e]|/\tau$ . Consequently, the time complexity to process a query using SF in a single block is  $O(\log |\mathcal{D}[t_s : t_e]|/\tau + k/\tau)$ . According to Lemma 4.1, if  $\tau \leq 0.5$ , the number of blocks processing the query is at most 2. Therefore, the query time complexity of MBI is  $O(\log |\mathcal{D}[t_s : t_e]|/\tau + k/\tau)$ .  $\square$

The following lemma is for the analysis of the query time complexity of MBI when  $\tau > 0.5$ . For this lemma, we first define an *internal left-aligned query (ILAQ) block* of a query as a block satisfying the following conditions: (1) it is not a leaf block, (2) this block covers the query time window, (3) the earliest timestamp of the block is identical to the start timestamp of the query. We also define the level of a block as the number of edges on the path from the root block to that block, with the root block itself being at level 0.

**LEMMA 4.3.** *MBI with an ILAQ block at the root processes the query using only one block per level, except at the leaf level where up to two blocks are used.*

**PROOF.** We prove the lemma by induction. Consider a query with time window  $(t_s, t_e)$  and an ILAQ block  $\mathcal{B}_p = (\mathcal{D}_p, \mathcal{G}_p)$  of the query. As the base case, assume the level of  $\mathcal{B}_p$  is 1, i.e., the children of  $\mathcal{B}_p$  are leaf blocks. If  $|\mathcal{D}[t_s : t_e]|/|\mathcal{D}_p| > \tau$ , MBI processes the query on  $\mathcal{B}_p$ , and the child blocks are not used. In the other case, the two children are used for the query instead of  $\mathcal{B}_p$ . Thus, the lemma is true in the base case. We now prove that the lemma is true when the level of  $\mathcal{B}_p$  is  $i + 1$  with the following inductive hypothesis: the lemma holds for every ILAQ block having level  $i$  or less. If  $|\mathcal{D}[t_s : t_e]|/|\mathcal{D}_p| > \tau$ , the lemma is trivially true because the query is wholly processed on  $\mathcal{B}_p$ . Let  $\mathcal{B}_l$  and  $\mathcal{B}_r$  be the left and right child blocks of  $\mathcal{B}_p$ . If  $0.5 < |\mathcal{D}[t_s : t_e]|/|\mathcal{D}_p| \leq \tau$ , the query is divided into two whose time windows are  $(t_s, t_m)$  and  $(t_m, t_e)$ , respectively, where  $t_m$  is the earliest timestamp of  $\mathcal{B}_r$ . The query of  $(t_s, t_m)$  is processed by  $\mathcal{B}_l$  as  $\mathcal{D}_l = \mathcal{D}[t_s : t_m]$ , and no descendants of  $\mathcal{B}_l$  are used.  $\mathcal{B}_r$  satisfies the conditions to be an ILAQ block for the query of  $(t_m, t_e)$ , thus, the subtree having  $\mathcal{B}_r$  as the root follows the lemma. Meanwhile,  $\mathcal{B}_l$  itself is not used, as it is not a leaf block and  $|\mathcal{D}[t_m : t_e]|/|\mathcal{D}_r| < |\mathcal{D}[t_s : t_e]|/|\mathcal{D}_p| \leq \tau$ , where  $|\mathcal{D}_r|$  is the number of vectors in  $\mathcal{B}_r$ . Thus, the lemma follows as only one block is used at level  $i$ . If  $|\mathcal{D}[t_s : t_e]|/|\mathcal{D}_p| \leq 0.5$ , the lemma also trivially holds because  $\mathcal{B}_r$  is not used as it doesn't have any vector belong to the query time window and  $\mathcal{B}_l$  is an ILAQ block of the query. Finally, by the base case and the induction step, the lemma is proven to be true.  $\square$

Using this lemma, we analyze the query time complexity of MBI when  $\tau > 0.5$  in the following theorem.

**THEOREM 4.4.** *When  $\tau > 0.5$  and the query time window is  $(t_s, t_e)$ , the query time complexity in MBI is  $O(\log^2 |\mathcal{D}[t_s : t_e]|/\tau + (k/\tau) \cdot \log |\mathcal{D}[t_s : t_e]|/\tau)$  where  $|\mathcal{D}[t_s : t_e]|$  is the number of vectors in  $\mathcal{D}[t_s : t_e]$ .*

**PROOF.** Let  $\mathcal{B}_p$  be the first block such that its left child block  $\mathcal{B}_l = (\mathcal{D}_l, \mathcal{G}_l)$  and right child block  $\mathcal{B}_r = (\mathcal{D}_r, \mathcal{G}_r)$  divide the query time window into  $(t_s, t_m)$ , and  $(t_m, t_e)$ . Then,  $\mathcal{B}_r$  is an ILAQ

block for the query of time window  $(t_m, t_e)$ . Among  $\mathcal{B}_r$  and its descendants, the largest block used for the query of  $(t_m, t_e)$  is not larger than  $|\mathcal{D}[t_s, t_e]| \cdot \tau$ . By Lemma 4.3, the worst case is that the descendants of the largest block are used for every level. As  $k$ NN search requires  $O(\log n + k/\tau)$  for  $n$  vectors and threshold  $\tau$ , we calculate the complexity for the query of  $(t_m, t_e)$  as follows:

$$\log \frac{|\mathcal{D}[t_s : t_e]|}{S_L \tau} \sum_{i=1}^{\log \frac{|\mathcal{D}[t_s : t_e]|}{S_L \tau}} \log \frac{|\mathcal{D}[t_s : t_e]|}{2^i \tau} + \frac{k}{\tau}$$

and this is bounded by  $O(\log^2 |\mathcal{D}[t_s : t_e]|/\tau + (k/\tau) \cdot \log |\mathcal{D}[t_s : t_e]|/\tau)$ . Finally, since the queries for the time windows  $(t_s, t_m)$  and  $(t_m, t_e)$  have a symmetrical structure, the overall query time complexity remains as the same, i.e.,  $\tau > 0.5$  is  $O(\log^2 |\mathcal{D}[t_s : t_e]|/\tau + (k/\tau) \cdot \log |\mathcal{D}[t_s : t_e]|/\tau)$ .  $\square$

When  $\tau \leq 0.5$ , MBI theoretically shows a better query time complexity compared to when  $\tau > 0.5$ , which is well demonstrated in Figure 9. Meanwhile, since the above analyses are about worst-cases, in some cases, the performance can be better when  $\tau > 0.5$  in practice.

## 5 EXPERIMENTS

In this section, we experimentally evaluate MBI to show its effectiveness for TkNN query. We focus on answering the following questions:

- Q1. Search Performance.** Does MBI provide the best search performance (Section 5.2)?
- Q2. Scalability.** How well does MBI scale up in terms of the data size (Section 5.3)?
- Q3. Data Insertion time.** How efficiently can new data be inserted into MBI (Section 5.4)?
- Q4. Effect of parameters.** How does MBI's parameters  $S_L$  and  $\tau$  affect the performance of MBI (Section 5.4)?

### 5.1 Experimental Setting

**5.1.1 Machine.** All experiments in this paper are performed on a machine equipped with AMD Ryzen 7 3800XT 8-Core Processor and 128GB of DDR4 RAM. Ubuntu 20.04.1 LTS is installed as the operating system. All codes are written in Rust, and compiled using Cargo v1.66.1 with option '--release'.

**5.1.2 Datasets.** We use two real-world datasets and four synthetic datasets, summarized in Table 2. The two real-world datasets are MovieLens and COMS. MovieLens is a set of movies, each represented as a 32-dimensional vector with its release year as the timestamp. The vectors are generated from users' ratings by a matrix factorization method. COMS consists of weather satellite images captured by the Communication, Ocean and Meteorological Satellite (COMS), which is the first geostationary multi-purpose satellite of South Korea. We represent each image as a 128-dimensional vector using an autoencoder. The date and time each image is captured is used as the timestamp. We also use four public large datasets that originally do not have timestamps: GloVe-100, SIFT1M, GIST1M, and DEEP1B. We consider the index of each item as its virtual timestamp. GloVe-100 is a large word embedding dataset where each word is represented as a 100-dimensional vector. SIFT1M, GIST1M, and DEEP1B are large image descriptor datasets, where images are represented as 128-dimensional, 960-dimensional, and 96-dimensional vectors, respectively.

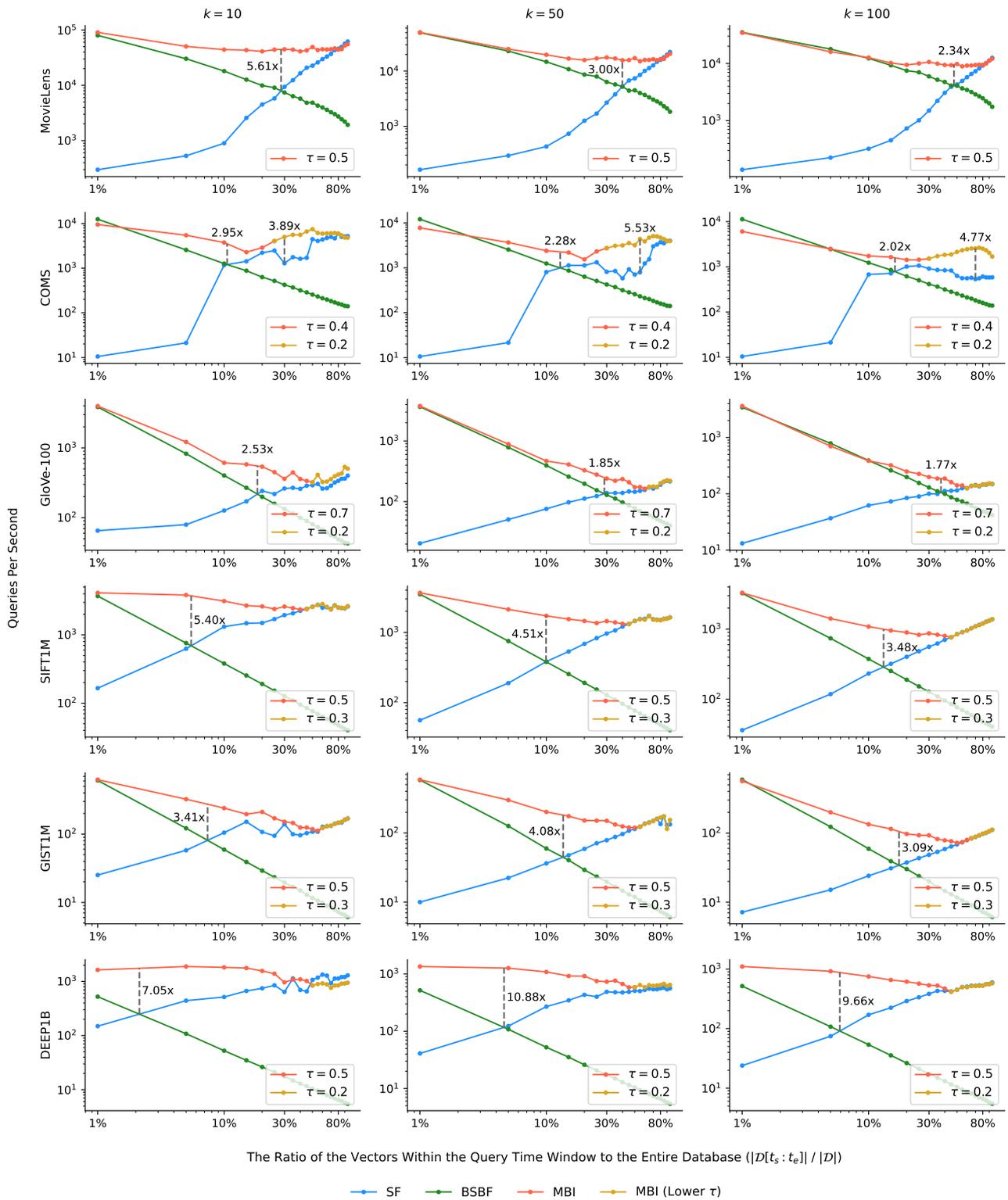


Figure 5: The ratio of the query time window vs. the number of queries per second when recall@k is set to 0.995 and k is set to 10, 50, and 100.

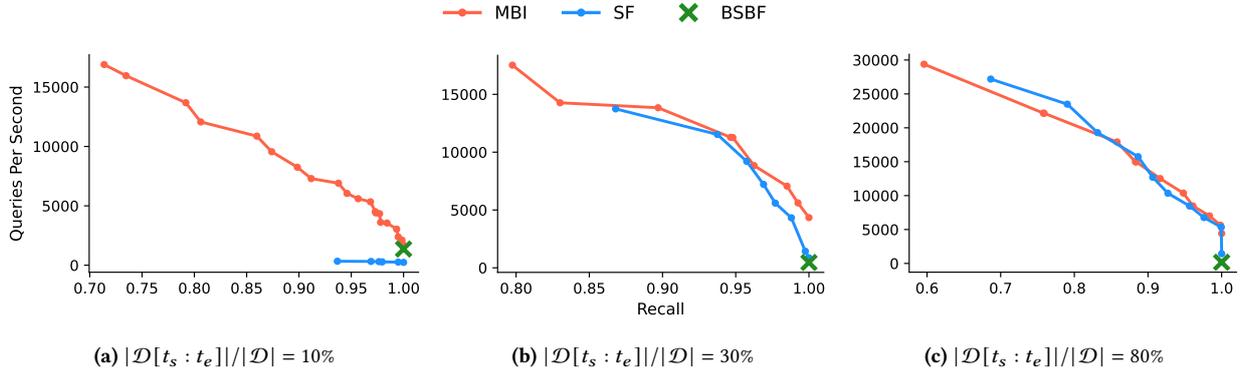


Figure 6: The recall@10 vs. queries per second on COMS dataset. The ratios of the query time window are set to 10%, 30%, and 80%.

Table 2: The summary of datasets.

Datasets	# items		Dim.	Distance	Source
	Train	Test			
MovieLens	57,571	200	32	Angular	GroupLens <sup>6</sup>
COMS	291,180	200	128	Angular	KMA <sup>7</sup>
GloVe-100	1,183,514	10,000	100	Angular	Pennington et al. <sup>8</sup> [33]
SIFT1M	1,000,000	10,000	128	Euclidean	Jégou et al. <sup>9</sup> [21]
GIST1M	1,000,000	1,000	960	Euclidean	Jégou et al. <sup>9</sup> [21]
DEEP1B	9,990,000	10,000	96	Angular	Babenko et al. <sup>10</sup> [5]

Table 3: Default parameters

Datasets	Graph Search Parameters				MBI Parameters	
	# neighbors	$M_C$	$\epsilon$	$k$	$\tau$	$S_L$
MovieLens	96	192			0.5	3550
COMS	256	256			0.2, 0.4	1000
GloVe-100	256	256	1 - 1.4	10 (default),	0.2, 0.7	36000
SIFT1M	128	128	(by 0.02)	50, 100	0.3, 0.5	15625
GIST1M	512	512			0.3, 0.5	15625
DEEP1B	64	64			0.2, 0.5	78000

5.1.3 *Parameter Settings.* For each block of MBI and SF,  $k$ NN graph based index structure, which is constructed by NNDescent, and the graph search algorithm in Algorithm 2 are used for  $Tk$ NN queries. As the number of neighbors in the graph and the number  $M_C$  of max candidates in the search algorithm significantly affect the search performance, we find the optimal values for each dataset by grid search and use them for our experiments. The search algorithm employed by both SF and MBI has another parameter  $\epsilon$ , which is involved in how far the candidate is extended during the search. The parameter  $\epsilon$  balances the trade-off between query speed and recall. We vary the value of  $\epsilon$  in increments of 0.02, ranging from 1 to 1.4, and present the optimal based on the Pareto frontier. We set the parameter  $k$ , representing the number of nearest neighbors to find, to different values (10, 50, and 100) to see how  $k$  affects the performance. If not specified, the default value for  $k$  is 10. MBI has two own parameters: the threshold  $\tau$  used for selecting blocks and the leaf size  $S_L$ . We use different  $\tau$  values for each dataset; the values are

<sup>6</sup><https://grouplens.org/datasets/movielens/>

<sup>7</sup><https://www.data.go.kr/data/15043600/fileData.do>

<sup>8</sup><https://nlp.stanford.edu/projects/glove/>

<sup>9</sup><http://corpus-texmex.irisa.fr/>

<sup>10</sup><https://sites.skoltech.ru/compvision/noimi/>

Table 4: Index sizes of MBI and SF

Datasets	Input Data Size (GB)	Index Sizes (GB)	
		MBI	SF
MovieLens	0.01	0.06 (6.08x)	0.02 (1.90x)
COMS	0.15	0.91 (6.35x)	0.27 (1.74x)
GloVe-100	0.48	3.95 (8.72x)	1.21 (2.49x)
SIFT1M	0.53	2.09 (4.28x)	0.80 (1.53x)
GIST1M	3.85	7.73 (2.15x)	4.64 (1.21x)
DEEP1B	3.92	18.24 (5.00x)	6.10 (1.56x)

selected to have the best query speed from a range of 0.1 to 0.9 (see Section 5.4.2). The default values for  $S_L$  are set according to the scale of each dataset considering the index size. The effect of  $S_L$  to the performance of MBI is studied in Section 5.4.1 in detail. The default parameters are summarized in Table 3.

## 5.2 Search Performance

Figure 5 shows the query speed of MBI, BSBF, and SF according to the size of the query time window when recall@ $k$  is 0.995 with  $k$  set to values of 10, 50, and 100. We sample 200 vectors randomly for each dataset to form queries. These queries are excluded during the indexing process. The start and end times (i.e.,  $t_s$  and  $t_e$ ) of the query time window are randomly determined to cover a fraction of the entire data. This process is repeated 5 times for fractions ranging from 1% to 95%. The x-axis represents the fraction, i.e.,  $\mathcal{D}[t_s : t_e] / |\mathcal{D}|$ , and the y-axis represents the number of queries per second. For MBI, we choose the values of  $\tau$  that result in the highest queries per second (see Section 5.4). MBI demonstrates superior query speed compared to BSBF and SF in all datasets regardless of the length of the query time window. MBI even outperforms a hypothetical method that selects the faster of BSBF and SF in most cases, showing up to 10.88 times faster speed. The tendency is similar for all datasets and the three values of  $k$ , while the query speed decreases as  $k$  increases. This figure also clearly shows that BSBF gets slower when the query time window gets longer, and SF gets slower when the query time window gets shorter, as mentioned in Sections 3.2.1 and 3.2.2. Figure 6 shows the recall@10 and queries per second of MBI, BSBF, and SF with various  $\epsilon$  values from 1 to 1.4 when  $\mathcal{D}[t_s : t_e] / |\mathcal{D}|$  is 10%, 30%, and 80%. COMS is used. Just like when recall@10 is 0.995, we observe that similar patterns mentioned above exist across other recall@10 values as well.

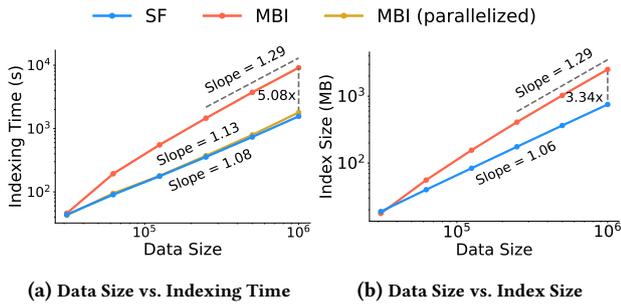


Figure 7: The scalability of MBI and SF on SIFT1M dataset.

### 5.3 Scalability

Figure 7 shows the data scalability of MBI. SIFT1M dataset is used. The slope in the plot indicates the scalability as both axes are on a log scale. Figure 7a shows the indexing time of MBI and SF according to the data size. The slope of MBI gradually decreases as the data size increases, showing a value of 1.29. This implies that to index double the data in MBI, it requires  $2^{1.29} = 2.45$  times the indexing time. This result is natural because, compared to SF, MBI exhibits a logarithmic increase in its indexing time complexity due to its hierarchical structure. We note that MBI trades off a slight increase in indexing time to significantly improve the speed of queries, which are more frequently executed. Besides, since each block in MBI is independently created, it is easily parallelized. By generating blocks in parallel, the indexing time of MBI is comparable to that of SF, resulting in a decrease of the indexing time by up to 5.08 times. Figure 7b shows the index sizes of MBI and SF according to the data size. Similarly to the indexing time results, the slope of MBI gradually decreases to 1.29 as the size of the data increases. This result matches the theoretical analysis in Section 4.4.1. The index sizes of other datasets are listed in Table 4. The number in a parenthesis indicates the relative index size compared to the input data size.

### 5.4 Effect of Parameters

MBI has two own parameters: an indexing parameter  $S_L$  of the leaf block size and a query parameter  $\tau$  for choosing the search block. We examine the effect of each parameter on the performance of MBI.

**5.4.1 Effect of leaf size  $S_L$ .** As the leaf size  $S_L$  affects the index size, the indexing time, and the query speed, we compare them in Figure 8. MovieLens is used. Figure 8a shows the cumulative indexing time measured when data are incrementally inserted into MBI. A lower  $S_L$  tends to require a higher indexing time, although the difference is not significant. As analyzed in Section 4.4.2, the results approximate  $n^{1.14} \log n$  with respect to the number of inserted data  $n$ . Figure 8b shows the query speed measured each time a vector is appended to MBI. The queries are conducted with the size of the time window randomly set from 5% to 95% of the current data size. The query speed tends to decrease slightly as  $S_L$  increases, but the difference is almost negligible. The query speed exhibits a kind of zigzag patterns but shows a steady decrease in the overall trend, which matches well with the results analyzed in the Section 4.4.3. The sudden increase in query speed occurs when MBI creates a complete tree.

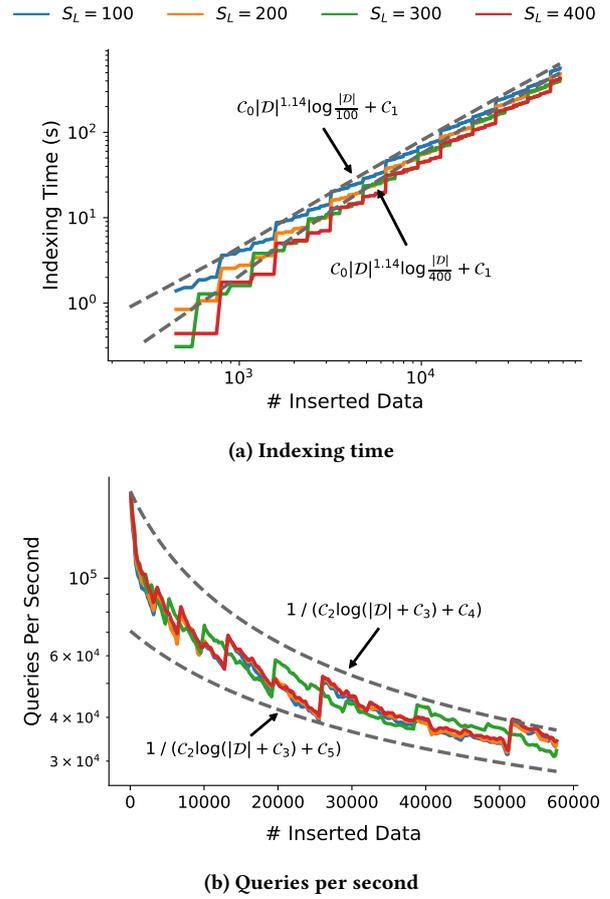
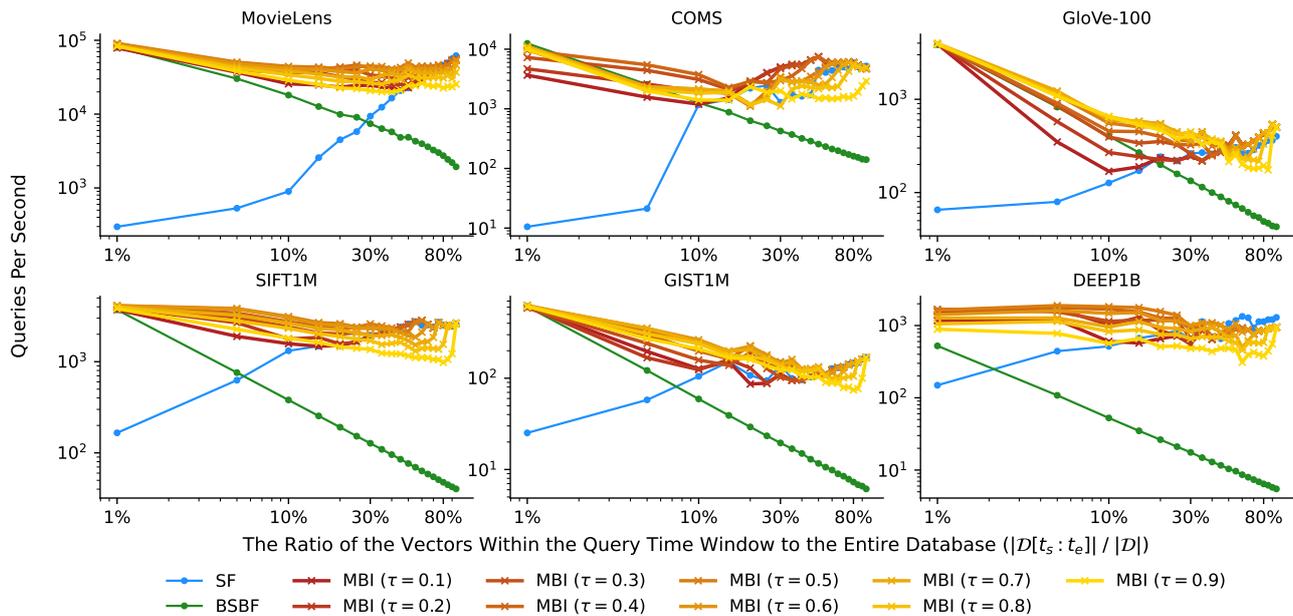


Figure 8: The indexing time and queries per second with different leaf sizes on MovieLens dataset.  $C_0$  to  $C_5$  are set as follows:  $1.2 \cdot 10^{-3}$ , 0.5,  $2.61 \cdot 10^{-5}$ , 45000,  $-2.74 \cdot 10^{-4}$ , and  $-2.655 \cdot 10^{-4}$ .

**5.4.2 Effect of threshold  $\tau$ .** Figure 9 shows the query speed of MBI with various  $\tau$  values from 0.1 to 0.9 according to the size of the query time window when recall@10 is 0.995. BSBF and SF are also displayed for reference. When  $\tau > 0.5$ , the query speed tends to decrease as  $\tau$  increases. This is because the search is performed in many blocks if  $\tau$  values is high. When  $\tau \leq 0.5$ , MBI performs well with a high  $\tau$  when the query time window is low, and conversely, performs well with a low  $\tau$  when the query time window is high. This is because it is guaranteed that queries are processed in two or fewer blocks when  $\tau$  is 0.5 or less as we prove it in Lemma 4.1, and the lower the  $\tau$  value, the more likely it is to be processed in larger blocks. The results show that the performance is satisfactory when  $\tau$  is around 0.5 across all datasets. Therefore, we recommend these values when no prior information is available. If possible, one can compute the optimal  $\tau$  for each query interval experimentally beforehand, and use the pre-computed  $\tau$  at run-time.

## 6 CONCLUSION

In this paper, we propose Multi-level Block Indexing (MBI), a novel and efficient indexing method for TkNN queries on high-dimensional and time-accumulating dataset. We devise an incremental hierarchical index structure, which organizes data into several blocks according to their timestamps so that query



**Figure 9: The ratio of the query time window vs. the number of queries per second when recall@10 is set to 0.995. MBI with different  $\tau$  from 0.1 to 0.9 is used.**

processing in MBI remains efficient, regardless of the length of the query time window. In our theoretical analysis of MBI’s efficiency, we find that the index size is  $O(|\mathcal{D}| \log |\mathcal{D}|)$ , and it requires  $O(|\mathcal{D}|^{1.14} \log |\mathcal{D}|)$  time to index the data, suggesting an amortized data insertion time of  $O(|\mathcal{D}|^{0.14} \log |\mathcal{D}|)$  where  $|\mathcal{D}|$  is the number of timestamped vectors in database  $\mathcal{D}$ . The query time is  $O(\log |\mathcal{D}| + k/\tau)$ , where  $k$  is the number of query results, and  $\tau$  is a constant parameter of MBI used for selecting blocks. Experimental results show that MBI is up to 10.88 times faster than a hypothetical method that selects the faster of conventional methods: Binary Search and Brute-Force (BSBF) and Search and Filtering (SF). Also, the index size and the time required to create the index in experiments corresponds to the theoretical results.

## ACKNOWLEDGMENTS

This work was funded by the Korea Meteorological Administration Research and Development Program "Developing Intelligent Assistant Technology and Its Application for Weather Forecasting Process" under Grant (KMA2021-00123). Ha-Myung Park is the corresponding author.

## REFERENCES

- [1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated Nearest Neighbor Search with Quick ADC. In *ICMR*. ACM, 159–166. <https://doi.org/10.1145/3078971.3078992>
- [2] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2021. Quicker ADC : Unlocking the Hidden Potential of Product Quantization With SIMD. *IEEE Trans. Pattern Anal. Mach. Intell.* 43, 5 (2021), 1666–1677. <https://doi.org/10.1109/TPAMI.2019.2952606>
- [3] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. 2019. PUFFINN: Parameterless and Universally Fast Finding of Nearest Neighbors. In *ESA*, Vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:16.
- [4] Franz Aurenhammer, Rolf Klein, and Der-Tsai Lee. 2013. *Voronoi diagrams and Delaunay triangulations*. World Scientific Publishing Company.
- [5] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.
- [6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R\*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. ACM, 322–331. <https://doi.org/10.1145/93597.98741>

- [7] Alina Beygelzimer, Sham M. Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *ICML*, Vol. 148. ACM, 97–104. <https://doi.org/10.1145/1143844.1143857>
- [8] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. *CoRR* abs/2111.08566 (2021). arXiv:2111.08566 <https://arxiv.org/abs/2111.08566>
- [9] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *STOC*. ACM, 537–546. <https://doi.org/10.1145/1374376.1374452>
- [10] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. ACM, 577–586. <https://doi.org/10.1145/1963405.1963487>
- [11] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.* 3, 3 (1977), 209–226. <https://doi.org/10.1145/355744.355745>
- [12] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. *CoRR* abs/1609.07228 (2016). arXiv:1609.07228 <http://arxiv.org/abs/1609.07228>
- [13] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *VLDB* 12, 5 (2019), 461–474. <https://doi.org/10.14778/3303753.3303754>
- [14] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
- [15] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. Morgan Kaufmann, 518–529.
- [16] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *ICML*, Vol. 119. PMLR, 3887–3896.
- [17] Marios Hadjieleftheriou, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. 2002. Efficient Indexing of Spatiotemporal Objects. In *EDBT*, Vol. 2287. Springer, 251–268. [https://doi.org/10.1007/3-540-45876-X\\_17](https://doi.org/10.1007/3-540-45876-X_17)
- [18] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *2016 (IEEE) Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Comput. Soc., 5713–5722. <https://doi.org/10.1109/CVPR.2016.616>
- [19] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *CoRR* abs/1810.07355 (2018). arXiv:1810.07355
- [20] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NIPS*, Vol. 32. Curran Associates, Inc.
- [21] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

- [22] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547. <https://doi.org/10.1109/TBDATA.2019.2921572>
- [23] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128. <https://doi.org/10.1109/TPAMI.2010.57>
- [24] George Kollios, Vassilis J. Tsotras, Dimitrios Gunopulos, Alex Delis, and Marios Hadjieleftheriou. 2001. Indexing Animated Objects Using Spatiotemporal Access Methods. *IEEE Trans. Knowl. Data Eng.* 13, 5 (2001), 758–777. <https://doi.org/10.1109/69.956099>
- [25] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *ICML*, Vol. 32. JMLR, 1188–1196. <http://proceedings.mlr.press/v32/le14.html>
- [26] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68. <https://doi.org/10.1016/j.is.2013.10.006>
- [27] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.
- [28] Rosalind B Marimont and Marvin B Shapiro. 1979. Nearest neighbour searches and the curse of dimensionality. *IMA J. Appl. Math.* 24, 1 (1979), 59–70.
- [29] Puya Memarzia, Maria Patrou, Md. Mahub Alam, Suprio Ray, Virendra C. Bhavsar, and Kenneth B. Kent. 2019. Toward Efficient Processing of Spatio-Temporal Workloads in a Distributed In-Memory System. In *MDM*. IEEE, 118–127. <https://doi.org/10.1109/MDM.2019.00-66>
- [30] Marius Muja and David G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *VISAPP. INSTICC*, 331–340.
- [31] Stephen M Omohundro. 1989. *Five balltree construction algorithms*. International Computer Science Institute Berkeley.
- [32] Rodrigo Paredes and Edgar Chávez. 2005. Using the  $k$ -Nearest Neighbor Graph for Proximity Searching in Metric Spaces. In *SPIRE*, Vol. 3772. Springer, 127–138. [https://doi.org/10.1007/11575832\\_14](https://doi.org/10.1007/11575832_14)
- [33] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *EMNLP*. 1532–1543. <http://www.aclweb.org/anthology/D14-1162>
- [34] Larissa Capobianco Shimomura, Rafael Seidi Oyamada, Marcos R. Vieira, and Daniel S. Kaster. 2021. A survey on graph-based methods for similarity searches in metric spaces. *Inf. Syst.* 95 (2021), 101507. <https://doi.org/10.1016/j.is.2020.101507>
- [35] Yufei Tao and Dimitris Papadias. 2001. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*. Morgan Kaufmann, 431–440. <http://www.vldb.org/conf/2001/P431.pdf>
- [36] Godfried T. Toussaint. 1980. The relative neighbourhood graph of a finite planar set. *Pattern Recognit.* 12, 4 (1980), 261–268. [https://doi.org/10.1016/0031-3203\(80\)90066-7](https://doi.org/10.1016/0031-3203(80)90066-7)
- [37] Trieu H. Trinh, Minh-Thang Luong, and Quoc V. Le. 2019. Selfie: Self-supervised Pretraining for Image Embedding. *CoRR* abs/1906.02940 (2019). [arXiv:1906.02940](http://arxiv.org/abs/1906.02940) <http://arxiv.org/abs/1906.02940>
- [38] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. 2000. Overlapping Linear Quadrees and Spatio-Temporal Query Processing. *Comput. J.* 43, 4 (2000), 325–343. <https://doi.org/10.1093/comjnl/43.4.325>
- [39] Dongjing Wang, ShuiGuang Deng, Xin Zhang, and Guandong Xu. 2016. Learning Music Embedding with Metadata for Context Aware Recommendation. In *ICMR*. ACM, 249–253. <https://doi.org/10.1145/2911996.2912045>
- [40] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *VLDB* 14, 11 (2021), 1964–1978. <https://doi.org/10.14778/3476249.3476255>
- [41] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik P. A. Lensch. 2016. Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU. In *CVPR*.
- [42] Donghui Yan, Yingjie Wang, Jin Wang, Honggang Wang, and Zhenpeng Li. 2018.  $K$ -nearest Neighbor Search by Random Projection Forests. In *Big Data*. IEEE Computer Society, Los Alamitos, CA, USA, 4775–4781. <https://doi.org/10.1109/BigData.2018.8622307>
- [43] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*. SIAM, 311–321.
- [44] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *ICDE*. IEEE, 1033–1044. <https://doi.org/10.1109/ICDE48307.2020.00094>
- [45] Wan-Lei Zhao, Hui Wang, and Chong-Wah Ngo. 2021. Approximate  $k$ -NN Graph Construction: A Generic Online Approach. *IEEE Transactions on Multimedia* 24 (2021), 1909–1921. <https://doi.org/10.1109/TMM.2021.3073811>