Hybrid B+ Tree and HNSW-based k-Nearest Neighbors Retrieval System

Ayman Youss
UM6P College of Computing
Supervised by :
Prof. Karima Echihabi
Anas Ait Aomar

December 28, 2024

Abstract

This paper presents a hybrid indexing and retrieval framework for performing k-Nearest Neighbors (kNN) search with an additional range constraint on a continuous attribute (e.g., timestamp). Specifically, a B+ tree is used to filter candidates by timestamp, and HNSW (Hierarchical Navigable Small World) graphs are employed to perform approximate kNN queries in high-dimensional space. Our Experimental results have proved that combining a B+ tree filter with HNSW-based retrieval yields efficiency gains for large candidate sets while preserving high recall.

1 Introduction

Finding the k nearest neighbors (kNN) subject to additional filtering constraints remains a core challenge in high-dimensional data retrieval. Exact indexing structures (e.g., KD-trees or R-trees) often yield results very slowly in high-dimensional spaces. On the other hand, approximate nearest neighbor (ANN) methods excel at rapid retrieval but typically do not natively support attribute-based filtering.

To bridge this gap, we propose a hybrid approach that combines a B+ tree for continuous-attribute filtering (in our case, timestamp ranges) with a Hierarchical Navigable Small World (HNSW) graph for approximate similarity search in the vector space.

Our work consists of:

- A hybrid pipeline that first applies B+ tree filtering for timestamp constraints, then switches between brute force and HNSW-based ANN depending on the candidate set size.
- An in-memory B+ tree implementation designed to handle continuous attributes and large insertion volumes without excessive rebalancing overhead.
- A single global HNSW index for all data points, eliminating the need to rebuild the graph for each query subset.
- A thorough evaluation of the system's trade-off between accuracy (recall) and speed, across varying timestamps and query loads.

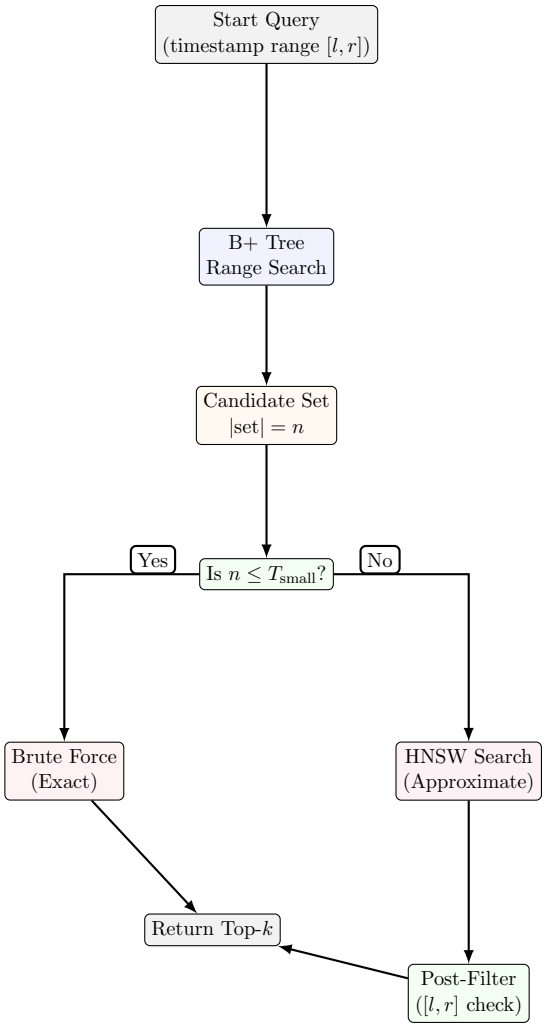

Figure 1: Hybrid flow diagram

2 Background and Related Work

2.1 B+ Tree Overview

B+ trees have been widely used in database systems to index numeric attributes (e.g., primary keys, timestamps) due to their balanced nature and efficient logarithmic-time search. Each internal node maintains a sorted array of keys, pointing to subtrees or leaves, while leaf nodes store actual data references or pointers. Range queries exploit the sequential ordering of leaf nodes, traversing from a starting key to an ending key in $O(\log N + K)$ time, where N is the total number of entries and K is the number of results.

2.2 HNSWlib for ANN

Approximate nearest neighbor (ANN) methods have gained popularity for handling large-scale high-dimensional data, where exact methods often become slow. HNSW (Hierarchical Navigable Small World) graphs represent one of the state-of-the-art ANN techniques, offering sub-linear query times while maintaining strong recall rates.

The core idea of HNSW is to construct a layered graph structure where nodes at higher layers connect to fewer neighbors, forming "shortcut links." During search, the algorithm first navigates from the top layer to the lower layers, progressively refining the candidate set. HNSWlib is a commonly used implementation that provides:

- Support for various distance metrics (we focus on squared Euclidean).
- Controllable parameters, such as:
  - `M`, dictating the maximum number of edges per node.
  - `efSearch`, controlling how many candidates are explored during query traversal.

3 B+ Tree Logic

3.1 Classes, Attributes, and Methods

3.1.1 Node Structure (`struct Node`)

Fields:

- **maxKeys:** Maximum number of keys the node can store (equals $2 \times d$ in the B+ Tree definition of the class).
- **numberOfKeys:** Current count of keys in this node.
- **isLeaf:** Whether the node is a leaf.
- **parent:** Pointer to the parent node (used when we insert and need to propagate splits upwards).
- **next:** Pointer to the next leaf node (used for fast range scans).
- **children:** Vector of child pointers (size = `maxKeys + 1`).
- **keys:** Vector of key values (size = `maxKeys + 1` so we have space when inserting).

3.1.2 BPlusTree Class

Fields:

- **root:** Pointer to the root `Node`.
- **degree:** Defines the node capacity. (equal to $2 \times$ order.)

Key Operations:

- `insert(value)`: Insert a key (float) into the B+ Tree.
- `search(value)`: Check if a key exists in the tree.
- `rangeSearch(low, high)`: Collect all keys within the interval [*low*, *high*].
- `printTree()`: Print the tree structure in a hierarchical format.
- `generateDot(std::ostream &os)`: Generate Graphviz DOT code for visualization.

3.2 Logic

3.2.1 Insertion Logic (`insert(float value)`)

Traverse Down to Leaf

- Start from the root and follow child pointers until you reach a leaf.
- For an internal node, pick the child whose key range would contain the value.

Insert Into Leaf

- In the leaf node, find the proper spot for `value` and shift existing keys to the right to maintain sorted order.
- Increase the `numberOfKeys`.

Check for Overflow

- If the leaf has more keys than `degree`, split the node:
  1. Create a new leaf `newLeaf`.
  2. Move half of the keys to `newLeaf`.
  3. The first key of `newLeaf` is the separator key.
  4. Update pointers (`leaf->next` and `newLeaf->next`).
  5. If the original leaf had no parent (meaning it was the root), create a new root. Otherwise, insert the separator key into the parent (via `insertInParent`).

3.2.2 Splitting Nodes

Splitting a Leaf Node (`splitLeaf`)

- Gather all the leaf's keys in a temporary vector.
- Divide the leaf's keys roughly in half:
  - `leaf` keeps the first half.
  - `newLeaf` gets the second half.
- Insert the "first key" of `newLeaf` into the parent (or make a new root if parent is `NULL`).

Splitting an Internal Node (`splitInternal`)

- Similar approach, but now we have *keys* and *children*.
- Copy the node's keys and children into temporary vectors.
- Find the median key (the "separator").
- `internal` keeps the left half (including children).
- `newInternal` gets the right half.
- The median key is pushed up to the parent. If the parent is `NULL`, create a new root.

3.2.3 Search Logic (`search(float value)`)

- Start at the root and move down to the leaf.
- In each internal node, choose the child where `value` could reside (compare with keys).
- Once in the leaf, do a simple linear scan for `value`.
- Return `true` if found; otherwise `false`.

3.2.4 Range Search Logic (`rangeSearch(float low, float high)`)

- Find the leftmost leaf that could contain `low`.
- Starting from there, traverse leaves via their `next` pointers.
- Collect all keys that fall within [*low*, *high*].

3.2.5 Tree Printing and DOT Generation

- `printTree():` Recursively prints each node's keys; indenting by level to visualize the structure.
- `generateDot(std::ostream &os):` Recursively creates a node label (with all keys), then draws edges to children. This can be fed into Graphviz to produce a graphical representation of the B+ Tree.

4 Dataset and Query Format

This work builds upon the dataset and query sets provided by the SIGMOD 2024 Programming Competition. The competition setting requires handling both a categorical attribute and a timestamp attribute in conjunction with high-dimensional vectors. Our project focuses on the timestamp-based range filtering.

4.1 Datasets Derived from YFCC100M

Our primary datasets and query sets are adapted from subsets of the *YFCC100M Dataset*. Each data point in YFCC100M is represented by:

- A high-dimensional vector derived via the CLIP model on images, then projected down to 100 dimensions via Principal Component Analysis (PCA).
- A discretized **categorical attribute** C, encoded as an integer beginning from 0.
- A **timestamp attribute** T, normalized into the range $[0, 1]$.

Table 1 lists two of the dataset-query pairs we used in this project. The smaller *dummy* files were utilized for debugging, packaging, and demonstration purposes; the larger *contest* files offer a medium-scale testing environment closer to real-world dimensions.

Table 1: Overview of Experimental Datasets and Queries

Name	Description	Dataset Size	Query Set Size
`dummy-data.bin` `dummy-queries.bin`	Dummy data/queries	10^4	10^2
`contest-data-release-1m.bin` `contest-queries-release-1m.bin`	Medium-scale data	10^6	10^4

4.2 Dataset Structure

Binary File Layout. The dataset file begins with a 4-byte integer `num_vectors` (`uint32_t`), which specifies the total number of data points. Then, for each data point, 102 floats are stored consecutively:

- **Category (C)**: 1 float.
- **Timestamp (T)**: 1 float.
- **100-Dimensional Vector**: 100 floats.

Hence, a single record is $1 + 1 + 100 = 102$ floats, and the entire file is:

`num_vectors` $\times$ 102 $\times$ `sizeof(float)` + 4 bytes (for the header).

Usage in Our System. When loading the dataset, we only retain the timestamp (the second float) as the continuous attribute (`dp.attribute`) and the last 100 floats as the high-dimensional embedding (`dp.coords`). The first float (categorical attribute) is ignored.

4.3 Query Format

Binary File Layout. The query file begins with a 4-byte integer `num_queries`. Each query occupies 104 floats:

1. `query_type` (1 float, can be 0,1,2,3).
2. `v` (1 float, categorical value or -1 if unused).
3. `l` (1 float, lower timestamp bound or -1).
4. `r` (1 float, upper timestamp bound or -1).
5. 100 floats representing the query vector.

Types of Queries. There are four possible query types:

- Type 0: Vector-only query (no additional constraints).
- Type 1: Vector + categorical (C = v).
- Type 2: Vector + timestamp ($l \leq T \leq r$).
- Type 3: Vector + categorical + timestamp ($C = v$ and $l \leq T \leq r$).

Since our system targets timestamp filtering, we only handle Type 2 and Type 3 queries.

5 Implementation Details (Data Loading)

5.1 Implementation Files Overview

A brief outline of our repository's core files is as follows:

- `bplustreecpp.h` : Defines and implements the B+ tree data structure, including range-search functionality.
- `dataset.h` / `dataset.cpp`: Handles loading of the dataset from binary files, parsing of queries, and related utility functions.
- `hnsw_wrapper.h` / `hnsw_wrapper.cpp`: Interfaces with the HNSW library, providing an easy way to construct, search, and manage the global approximate nearest neighbor index.
- `main.cpp`: Contains the `main` entry point for running the complete pipeline, parsing command-line arguments, and performing the hybrid query pipeline.

5.2 Dataset Loading (`loadDatasetFromBin`)

We open the dataset file in binary mode and read num_vectors. For each data record, we read 102 floats into a temporary buffer:

- The first float (buffer[0]) is the categorical attribute (ignored).
- The second float (buffer[1]) is the timestamp, which we store in DataPoint.attribute.
- The subsequent 100 floats (buffer[2..101]) are stored in DataPoint.coords.

5.3 Query Loading (`loadQueriesFromBin`)

We open the query file in binary mode and read num_queries. Each query has 104 floats, which we place into a buffer:

$$\begin{aligned}
\text{buffer}[0] &= \text{query_type } (0,1,2,3).\\
\text{buffer}[1] &= \text{categorical value (or -1)}.\\
\text{buffer}[2] &= \text{lower timestamp bound (l) (or -1)}.\\
\text{buffer}[3] &= \text{upper timestamp bound (r) (or -1)}.\\
\text{buffer}[4..103] &= \text{100-dimensional query vector}.
\end{aligned}$$

Only queries of type 2 or 3 are appended to our in-memory queries list. For each valid query, we store the 100-dim query vector in q.queryVec and the timestamp bounds in $q.a\_min$ and $q.a\_max$.

6 Hybrid Query Pipeline Design

6.1 B+ Tree for Attribute Filtering

We use a B+ tree to manage the continuous (timestamp) attribute. Each leaf node in this tree stores timestamps in ascending order, allowing efficient range queries. For fast retrieval of data

point identifiers (IDs), we maintain an additional lookup structure, which can be conceptualized as a map from attribute values to a list of vector IDs:

$$\texttt{attributeMap}[t] \rightarrow \{\texttt{ID}_1, \texttt{ID}_2, \dots\}.$$

Here, t is a timestamp value, and each ID corresponds to a high-dimensional vector whose timestamp field matches t. A typical workflow for constructing and querying the B+ tree is as follows:

1. **Insertion:** For each data point, we insert its timestamp t into the B+ tree. We also append the data point's ID to the corresponding list in `attributeMap`.
2. **Range Search:** Given a query range $[l, r]$, the B+ tree is traversed from the first timestamp greater than or equal to l to the last timestamp less than or equal to r. The resulting set of timestamps is used to retrieve all associated IDs from the `attributeMap`.

6.2 HNSWlib for Approximate kNN

Once the candidate set of point IDs has been identified via the B+ tree, our system may perform approximate k-nearest neighbor search using HNSWlib. In our implementation, we build a *single global* HNSW index:

- **Distance Function:** HNSWlib naturally returns the squared Euclidean distance, which is then converted to the true Euclidean distance by taking the square root in post-processing.
- **Parameters (`M` and `efSearch`):** During index construction, we set the maximum number of edges per node using a parameter `M`, which controls the connectivity. At query time, we use `efSearch` to configure the search beam width within the HNSW graph. Larger values of `efSearch` lead to more accurate but slower searches.
- **Index Build:** The global HNSW graph is built once for the entire dataset. This avoids reducing query latency by reconstructing the index for every query.
- **Post-Filtering:** After retrieving the nearest neighbors from the HNSW index, post-filtering is applied to the results, to ensure that the datapoints fall into the timestamp range. The number of points retrieved is equal to C * k where k is the number of neighbors intended and C is a positive integer value. This ensures that there are always potential candidates.

6.3 Multithreading

While the procedure above describes a single query, most real-world systems must handle many queries concurrently. To increase overall throughput, our design optionally employs a **multi-threaded** approach:

- **Thread Pool:** We create a pool of worker threads that each retrieve and process a single query.
- **Shared Data Structures:** The B+ tree and global HNSW index are shared read-only across threads (once built).

7 Benchmarking and Evaluation

7.1 Experimental Setup

- Processor: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (12 CPUs), 2.6GHz
- Memory (RAM): 16GB (16384MB)
- Page File: 15538MB used, 29109MB available
- Operating System: Windows 11 Home Single Language 64-bit (10.0, Build 22631).

7.2 Experimental Results

Our experimental evaluation considers various parameter settings, including:

- `efSearch` , influencing HNSW's retrieval accuracy.
- `Threshold` (T_{small}), determining when the system chooses brute force over ANN.
- **B+ tree order**, impacting how many keys each node can hold and thus affecting insertion and range query costs.

All the detailed results can be found on the github repository. Here, We measure:

- **Total Time**: The sum of loading time and the time spent on all queries.
- **Loading Time**: How long it takes to build the B+ tree and HNSW structures.
- **Query Time**: The time to process all queries, broken down by range-filtering and neighbor-search operations.
- **Total Recall**: The fraction of true neighbors retrieved (averaged over queries).

`efSearch`	`threshold`	`order`	**Loading Time**	**Query Time**	**Total Time**	**Recall (%)**
100	100	200	7380	118	7498	0.997021
100	100	300	7700	34	7734	0.998723
100	1000	200	7655	10	7665	1
100	1000	300	7616	7	7623	1
200	100	200	7753	117	7870	1
200	500	300	7617	7	7624	1
200	1000	200	7570	11	7581	1
200	1000	300	7570	7	7577	1

Illustrative results for different combinations of `efSearch`, `threshold`, and B+ tree `order`.

8 Conclusion and Future Work

Our proposed hybrid system of B+ tree attribute filtering and HNSW-based approximate nearest neighbor search has demonstrated good performance. Nevertheless, there remain various avenues to explore for further optimization and scalability:

- **Parallel Query Execution:** While we already process multiple queries in parallel, we can further optimize B+ tree insertions by multithreading them *when node splitting is not expected*, thereby reducing potential lock contentions. In high-performance computing environments such as the *Toubkal* cluster at SimLab, distributing queries across multiple nodes may yield near-linear performance gains.
- **GPU-based Distance Calculation:** To accelerate distance computations on large candidate sets, we can offload brute-force distance calculations or HNSW graph traversals to the GPU.
- **Additional Attributes:** We can extend the B+ tree to support multiple attributes (or building additional indexes).
- **Larger-Scale Experiments:** We could evaluate the system on bigger, more diverse datasets (potentially in the hundreds of millions of points) to ensure reliability and throughput.
- **Bulk Loading:** Bulk loading can significantly enhance the performance of our B+ tree construction.

Acknowledgments

Throughout the project, AI tools (specifically gpt4o model) were occasionally used for debugging, and documenting code and, after completing the initial draft of this report, for refining the language and improving clarity.

The logic for reading and processing input files (binary files) was inspired by the winning solution from the competition (Team Alaya), as it was very hard to parse a binary file without knowledge of exact size of the fields. Their code can be found in this link :
https://dbgroup.cs.tsinghua.edu.cn/sigmod2024/leaders.shtml .
I have also run the experiment on a bigger scale dataset (1m data and queries from sigmod dataset), it takes about 20 minutes to complete. I did not have enough time to add its benchmark to the report, but the results are pushed to the github repository.

I would like to express appreciation to Anas Ait Aomar for his support and responsiveness during the project. He was helpful in overcoming key challenges about the project. I would like to thank also Prof. Karima Echihabi for her guidance.