College of Computing

Combining B+ Trees and HNSW for Accelerated Approximate Nearest Neighbor Search

A Study on Index Structures for Efficient ANN Retrieval

Report Submitted by:
Mahmoud Maftah

Course:
Database Management Systems 2

Supervisor:
Karima Echihabi
Anas Ait Aomar

Date:
December 29, 2024

Contents

1 Introduction

With the rise of high-dimensional data in fields like computer vision, natural language processing, and recommender systems, the *k-Nearest Neighbors* (k-NN) problem has become a cornerstone of many applications. Traditionally, k-NN focuses on finding the k closest points in a high-dimensional space based on some distance function (e.g., Euclidean). However, in many real-world scenarios, one also needs to impose additional *scalar filters* to further restrict which points are candidates for matching. For instance, a scalar filter might be a numerical property (e.g., timestamp, score, or geographical coordinate) that disqualifies items outside a certain range.

This work explores a combined method to efficiently handle the k-NN search *and* the scalar filtering by integrating two data structures:

1. A **B+ Tree** to manage scalar range queries,
2. An **HNSW** (Hierarchical Navigable Small World) index to efficiently approximate nearest neighbors in high-dimensional space.

We compare two different approaches for deciding how many neighbors to retrieve from HNSW: a fixed-threshold method and a probabilistic method. The latter ensures, with high confidence, that at least k valid points (those satisfying the scalar filter) appear among the HNSW candidates.

2 Problem Description

2.1 Overview of the k-NN and Scalar Filtering Task

Suppose we have a database of M items, each item represented by:

- A *vector* in R^d (e.g., image features, word embeddings),
- A *scalar value* s (e.g., a relevance score, timestamp, or price).

We want to process queries of the form:

1. **k-NN Search**: Given a query vector $\mathbf{q}$, find the k closest items (in ℓ_2 distance or another metric).
2. **Scalar Filtering**: Restrict to items whose scalar s lies in the interval $[S_{\min}, S_{\max}]$.

The problem is that a standard approximate nearest neighbor algorithm (like HNSW) has no direct mechanism to *only* retrieve vectors with $s \in [S_{\min}, S_{\max}]$. Consequently, we must either:

- Retrieve a large set of neighbors from HNSW and then *filter* them by checking the scalar condition,
- Or avoid approximate indexing and do a brute force approach for all valid items.

Both can be inefficient: The former can yield many irrelevant neighbors if the filter is very restrictive; the latter can be slow if the database is large.

2.2 Role of the B+ Tree

To handle the scalar condition efficiently, we maintain a **B+ Tree** keyed by the scalar s. A B+ Tree is a self-balancing tree data structure commonly used in databases to organize and retrieve data with good I/O performance. Key benefits and time complexities include:

- **Search Complexity**: $O(\log n)$ for searching a single key in a B+ Tree with n entries.
- **Range Queries**: B+ Trees natively support range queries by traversing leaf nodes, often achieving $O(\log n + k)$ complexity for returning k matching entries.
- **Insertion / Deletion**: Also $O(\log n)$ on average, maintaining balanced structure through splits and merges.

In our use case, the B+ Tree can quickly count or list all items whose scalar s is in the range $[S_{\min}, S_{\max}]$. This helps decide whether we should rely on brute force (when the range is small) or an approximate approach (when the range is large).

2.3 Two B+ Tree Implementations

1) General-Purpose B+ Tree. Our first implementation is a fairly conventional B+ Tree supporting:

- **Insertion, Deletion, Search**: All in $O(\log n)$ average complexity.
- **Range Queries**: Allows fetching all items with keys within $[S_{\min}, S_{\max}]$.
- **Subtree Size Maintenance (Optional)**: In some versions, each node tracks the total number of entries in its subtree, enabling faster counting operations like how many keys are $\leq x$.

2) B+ Tree with Enhanced Range Query Support. The second implementation is optimized toward *range queries* and counting within an interval. It may:

- Store extra metadata (like subtree sizes) to quickly compute how many keys fall below a certain threshold,
- Provide specialized methods, e.g., `countInRange(`$S_{\min}, S_{\max}$`)` and `rangeQuery(`$S_{\min}, S_{\max}$`)` that run efficiently without fully iterating through all leaf nodes.

By specifically addressing the needs of range-based lookups, this B+ Tree variant can deliver even more efficient filtering decisions, which is essential in our hybrid approach combining B+ Trees and HNSW.

2.4 Bringing It All Together

With the B+ Tree in place for scalar filtering and HNSW for high-dimensional neighbor search, the system proceeds as follows:

1. The B+ Tree determines the count of items in the range $[S_{\min}, S_{\max}]$.
2. Depending on this count, we either do a brute force among those items or retrieve candidates from HNSW, then apply a probabilistic or fixed-threshold approach to balance efficiency and accuracy.

In the following sections, we compare two approaches for deciding the number of HNSW candidates to retrieve (O): the *fixed-threshold* method versus a *probabilistic* strategy. We also provide detailed formulas for the latter, show how to implement it, and analyze its benefits over the simpler fixed threshold.

3 Comparative Analysis of Fixed-Threshold and Probabilistic Approaches

In this section, we discuss two approaches for determining the number of neighbors O to retrieve from a Hierarchical Navigable Small World (HNSW) graph when performing a k-nearest neighbors (k-NN) query under a scalar filter (i.e., only certain items are valid based on a scalar property). Specifically, we compare the *fixed-threshold* approach with a more *probabilistic* strategy.

3.1 Fixed-Threshold Approach

3.1.1 Overview

In the fixed-threshold approach, a constant integer O (e.g., 1000) is chosen *a priori* to decide how many candidates the HNSW index should return for each query. Once these O candidates are retrieved, we apply a *scalar filter* to discard points outside the range $[S_{\min}, S_{\max}]$ and then take the top-k among those remaining points (if at least k are available).

3.1.2 Drawbacks

While straightforward, this method can fail if too many of the O candidates do not meet the scalar condition. For instance, suppose there are S points in the dataset that satisfy the filter out of a total of M points. If S is small relative to M, the probability that none (or fewer than k) of the O retrieved items falls in $[S_{\min}, S_{\max}]$ can be high. Consequently, we might miss valid items even if they exist in the database.

3.2 Probabilistic Approach

3.2.1 Motivation

To alleviate the limitations of a fixed threshold, we propose a *probabilistic* approach that dynamically determines O based on:

1. The fraction $p = S/M$ of the dataset that meets the scalar filter.
2. The desired confidence that we retrieve *at least* k valid neighbors.

By modeling the retrieval from HNSW as if drawing O samples from the dataset (and assuming each sample independently meets the condition with probability p), we can employ a binomial distribution to estimate how large O must be to meet our desired probability of success.

3.2.2 Binomial Model

Consider a random variable $X \sim \text{Binomial}(O, p)$, where O is the number of draws (i.e., neighbors requested from HNSW), and $p = \frac{S}{M}$ is the probability that a randomly drawn point satisfies the scalar condition. Under this model:

$$P(X = i) \;=\; Oi\, p^i\, (1-p)^{O-i},$$

where $0 \le i \le O$ and Oi is the binomial coefficient.

Ensuring at Least One Valid Point As a simple example, to ensure at least one valid point with probability $1-\alpha$, we need: $\mathrm{P(X} \geq 1) = 1 - P(X = 0) \;=\; 1 - (1-p)^O \;\geq\; 1-\alpha$,
$\Rightarrow \quad (1-p)^O \leq \alpha$, which implies:

$$O \;\geq\; \frac{\ln(\alpha)}{\ln(1-p)}.$$

Ensuring at Least k Valid Points In the more general case, we want *at least* k valid points. The probability of having fewer than k valid items in the O draws is:

$$P(X < k) = \sum_{i=0}^{k-1} Oi\, p^i\, (1-p)^{O-i}.$$

Hence, the probability of getting at least k valid points is:

$$P(X \geq k) = 1 - \sum_{i=0}^{k-1} Oi\, p^i\, (1-p)^{O-i}.$$

For a desired confidence level $1-\alpha$, we want

$$P(X \geq k) \;\geq\; 1-\alpha \iff \sum_{i=0}^{k-1} Oi\, p^i\, (1-p)^{O-i} \;\leq\; \alpha. \tag{1}$$

We solve the equation above to obtain the smallest integer O that satisfies this condition. This ensures that, with probability at least $1-\alpha$, we will retrieve at least k valid neighbors from the scalar-filtering perspective, note that this approach is based on a very strong assumption, according to which there is no correlation between our filtering index, and the distance from the query vector.

3.2.3 Algorithmic Steps

The main steps of the probabilistic approach are:

1. Use the B+ Tree to compute $S = \texttt{countInRange}(S_{\min}, S_{\max})$, the number of points that satisfy the scalar filter.
2. Calculate $p = \frac{S}{M}$, where M is the total number of points in the dataset.
3. Choose a confidence parameter α (e.g., 0.01 for a 1% chance of failure).
4. Solve the binomial inequality and get the number of instances to get from the HNSW index, later in this report we will present an efficient way to compute it.

3.2.4 Discussion of Practical Considerations

Although HNSW's retrieval mechanism is not purely random, the binomial model often serves as a good approximation in many high-dimensional or lightly correlated scenarios. As p grows smaller (i.e., $S \ll M$), the required O to maintain the same confidence $1-\alpha$ will grow larger, thus increasing the computational overhead. However, this overhead is precisely what allows the user to *guarantee* a small probability of returning fewer than k valid neighbors when they do in fact exist in the dataset.

In practice, the choice of α and k must balance computational efficiency with application-specific requirements. For example, lower values of α (greater confidence) increase O, which may not always be practical

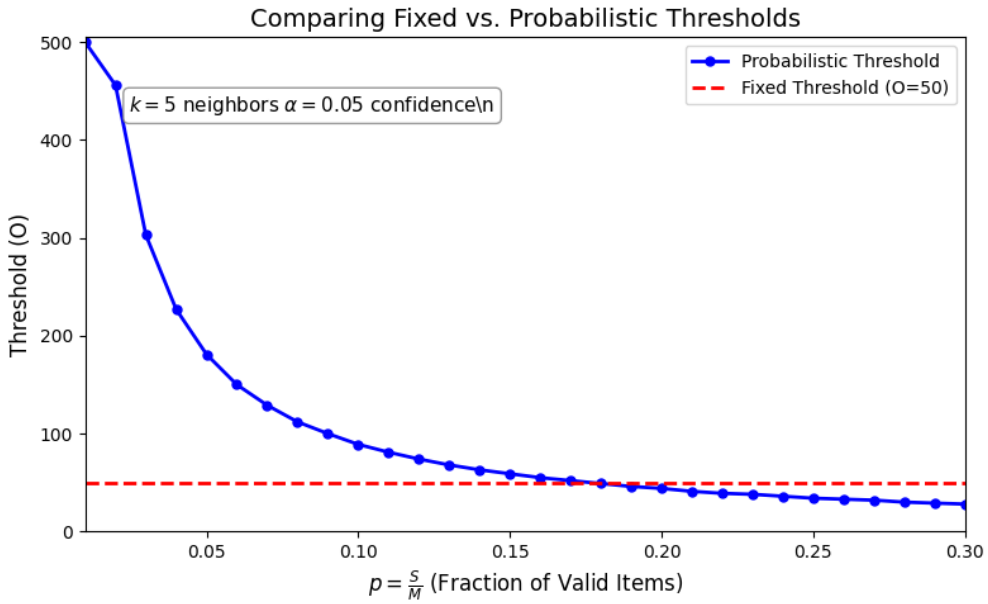

Figure 1: Illustration of how the probabilistic approach chooses O based on $p = \frac{S}{M}$ and a confidence parameter α. As p gets smaller, O increases to ensure the probability of retrieving at least k valid items remains high.

for real-time systems. Similarly, larger k requires higher O, further compounding the computational cost. Therefore, these parameters should be fine-tuned based on the trade-offs between accuracy, confidence, and performance demands.

```cpp
/**
 * @brief Binomial PMF: P(X = k) for X ~ Binomial(n, p).
 */
double binomialPMF(int n, int k, double p) const {
    if (p < 0.0 || p > 1.0) return 0.0;
    double c = binomialCoefficient(n, k);
    double pk  = std::pow(p,   k);
    double pmk = std::pow(1.0 - p, n - k);
    return c * pk * pmk;
}
```

Listing 1: Binomial Probability Mass Function (PMF)

```cpp
/**
 * @brief Probability that a Binomial(n, p) random variable is less than k.
 *        i.e., P(X < k).
 */
double binomialCDFLessThan(int n, int k, double p) const {
    double sumProb = 0.0;
    for (int i = 0; i < k; i++) {
        sumProb += binomialPMF(n, i, p);
    }
    return sumProb;
}
```

Listing 2: Binomial Cumulative Distribution Function (CDF)

4 Complexity Analysis

4.1 Insertion

To insert a key, we first find the target leaf by descending from the root in $O(\log_d(N))$. Within the leaf, we insert via a binary search among up to $d-1$ keys, which is $O(\log d) \approx O(1)$ for small d. If the leaf overflows, it splits—potentially propagating upward one level at a time, at most reaching the root. Each split or insertion into an internal node costs $O(d)$ and can happen over $O(\log_d(N))$ levels. Overall, insertion is $O(\log_d(N))$.

4.2 Deletion

Deletion similarly begins by locating the target leaf in $O(\log_d(N))$. Removing the key from the leaf costs $O(\log d) \approx O(1)$. If underflow occurs, merges or borrows may propagate upward, each step costing $O(d)$ over at most $O(\log_d(N))$ levels. Thus, deletion also runs in $O(\log_d(N))$.

4.3 Search (Single Value)

A single-value search descends the tree in $O(\log_d(N))$ and then does a quick binary search ($O(\log d) \approx O(1)$) in the leaf. Hence, search is $O(\log_d(N))$.

4.4 searchAll (All Values for a Key)

Finding the leaf for a given key takes $O(\log_d(N))$. Returning the values in that leaf is $O(1)$ plus whatever time is needed to output the entire vector of stored values.

4.5 countLessOrEqual

We traverse down the tree in $O(\log_d(N))$, using `subtree_size` metadata to decide how many keys to count in each subtree. Any extra work per node is $O(d) \approx O(1)$. Thus, `countLessOrEqual` is $O(\log_d(N))$.

4.6 countInRange($S_{\min}, S_{\max}$)

This uses two `countLessOrEqual` calls—one for $S_{\max}$ and one for $S_{\min} - 1$. Each call is $O(\log_d(N))$, so counting in a range remains $O(\log_d(N))$.

4.7 rangeQuery($S_{\min}, S_{\max}$)

We locate the first leaf containing $S_{\min}$ in $O(\log_d(N))$. From there, we scan leaf nodes (linked by `next`) until we exceed $S_{\max}$. Collecting K matching entries adds $O(K)$, making `rangeQuery` $O(\log_d(N) + K)$.

4.8 Subtree Size Updates

Whenever keys are inserted or deleted, `subtree_size` is updated from the modified leaf up to the root. This process touches $O(\log_d(N))$ nodes; each node's update takes $O(d) \approx O(1)$. Hence, maintaining `subtree_size` during insertions or deletions also costs $O(\log_d(N))$.

Operation	**Time Complexity**
Insertion	$O(\log_d(N))$
Deletion	$O(\log_d(N))$
Search (single)	$O(\log_d(N))$
SearchAll	$O(\log_d(N))$
countLessOrEqual	$O(\log_d(N))$
countInRange	$O(\log_d(N))$
rangeQuery	$O(\log_d(N) + K)$

5 Challenges Faced and Design Considerations

During the implementation of our B^+ Tree for storing and querying items, we encountered several challenges, both conceptual and practical, as detailed below.

5.1 Segment Tree for Scalar Filtering

The primary goal of our data structure was to handle two types of queries on a database of M items:

1. **k-NN Search:** Given a query vector $\mathbf{q} \in R^d$, return the k closest items with respect to some distance metric (e.g., ℓ_2).
2. **Scalar Filtering:** Restrict to items whose scalar score s (e.g., a relevance score, timestamp, or price) lies in an interval $[S_{\min}, S_{\max}]$.

Initially, we contemplated augmenting the B^+ Tree with a *segment tree* to quickly count and retrieve items in a given scalar range. A segment tree excels at range-count and range-sum queries in $\mathcal{O}(\log M)$ time and can help prune searches when the number of items satisfying $[S_{\min}, S_{\max}]$ is large. For disk-based systems, such pruning could drastically reduce disk I/O by immediately switching to a specialized index structure (e.g., an HNSW graph for k-NN) when the result set grows beyond a certain threshold. However, in our in-memory B^+ Tree, the performance gains from segment trees turned out to be relatively modest compared to the additional complexity of maintaining such a secondary structure.

5.2 Insertion Overhead and Node Splits

B^+ Trees require special handling when a node becomes overfull or underfull. In particular:

- **Splitting Nodes:** When an insertion causes a leaf node to exceed the allowed maximum of $d - 1$ keys, the node is split into two. This split operation involves copying roughly half of the keys (and associated values) into a new node, which can lead to temporary overheads.
- **Borrowing from Siblings:** Deletions may cause *underflow*, triggering a borrow or merge operation. Borrowing may require shifting elements between adjacent leaf nodes, again incurring a cost of $\mathcal{O}(d)$.
- **Inserting into Arrays:** Each node maintains its keys in a simple array (or `std::vector`). Inserting a new key in sorted order is typically $\mathcal{O}(d)$, since it can require shifting a portion of the array.

We explored using a `map` (e.g., a balanced tree such as a Red-Black Tree) for storing node keys to achieve faster inserts at sorted positions. However, splitting a `map` into two parts is nontrivial, since the underlying data structure is not designed for mid-point splits. Many balanced-tree containers (like `std::map` in C++) do not expose efficient mid-split functionality, making node splitting a complicated or inefficient operation.

5.3 Alternative Data Structures: Treaps

Another structure we considered was a *treap*, which supports efficient splits and merges by randomly assigning priorities to each key. Treaps can efficiently handle range queries and dynamic splits. However, integrating a treap into the B^+ Tree framework complicates the code significantly, and would require customizing both the node layout and the balancing routines. Given the additional complexity and our aim for a simpler textbook-style B^+ Tree, we decided to stick with the array-based approach.

5.4 Empirical Evidence of Low Overhead

To further justify our choice of storing keys in a simple array, we performed empirical tests of insertion time as a function of the order d. For orders up to around 200, our measurements showed that insertion times decrease as d increases, reaching a minimum at $d = 200$. Beyond this point, the overhead of binary search (`lower_bound()` in C++) and node splitting—requiring the copying of large keys/values arrays—becomes significant.

However, this is not a major concern. Unlike disk-based B+ trees, where large orders are needed to maximize the base of the logarithm and minimize the number of I/Os, in-memory B+ trees perform well with orders greater than 9. This is because multiple nodes from the B+ tree can reside on the same disk page, avoiding space waste. In contrast, in disk-based B+ trees, each node must occupy a single disk page, necessitating larger orders to optimize space and I/O performance.

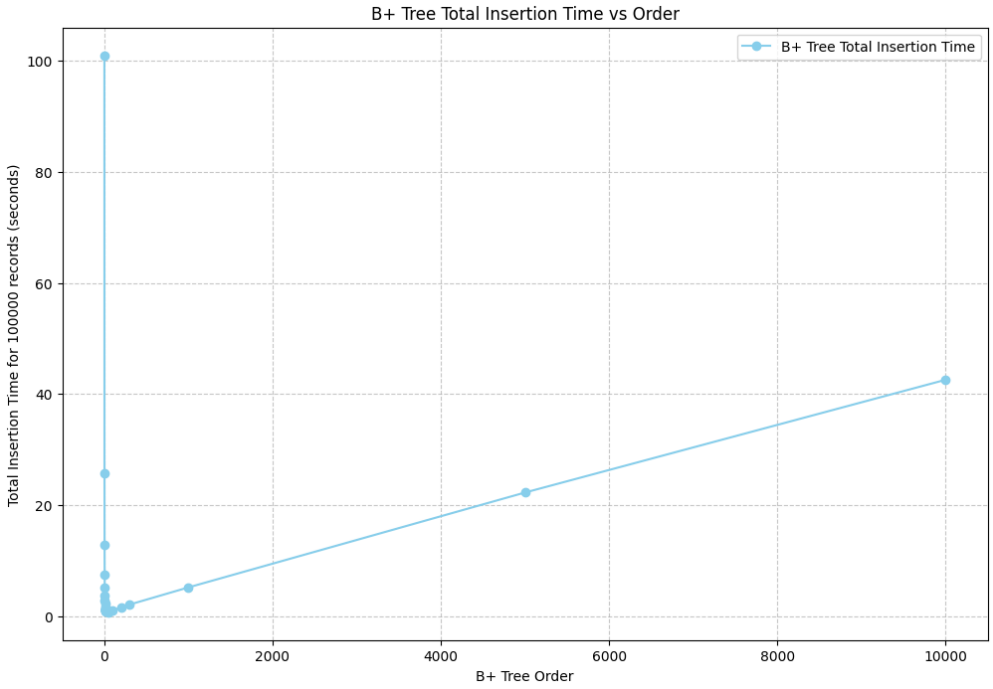

Figure 2: Insertion Time vs. B^+ Tree Order d

As depicted in Figure 2, the insertion time increases with the order d of the B^+ Tree. However, this increase is not a significant concern for our in-memory B^+ Tree implementation. Unlike disk-based B^+ Trees, where maximizing the order d is crucial to minimize disk I/O operations by fitting multiple nodes into a single disk page, our in-memory approach benefits from the high-speed access of RAM.

5.5 Benchmarking B^+ Tree Insertion Performance

To assess the impact of the B^+ Tree order d on insertion efficiency, we conducted benchmark tests by inserting up to 10^5 records using trees with orders 3, 5, 7, and 9. The insertion times recorded for each order are presented in Figure 3.

The benchmark results reveal that lower-order B^+ Trees, such as order 3, exhibit significantly higher insertion times, especially as the number of records increases. Specifically, for inserting 10^5 records:

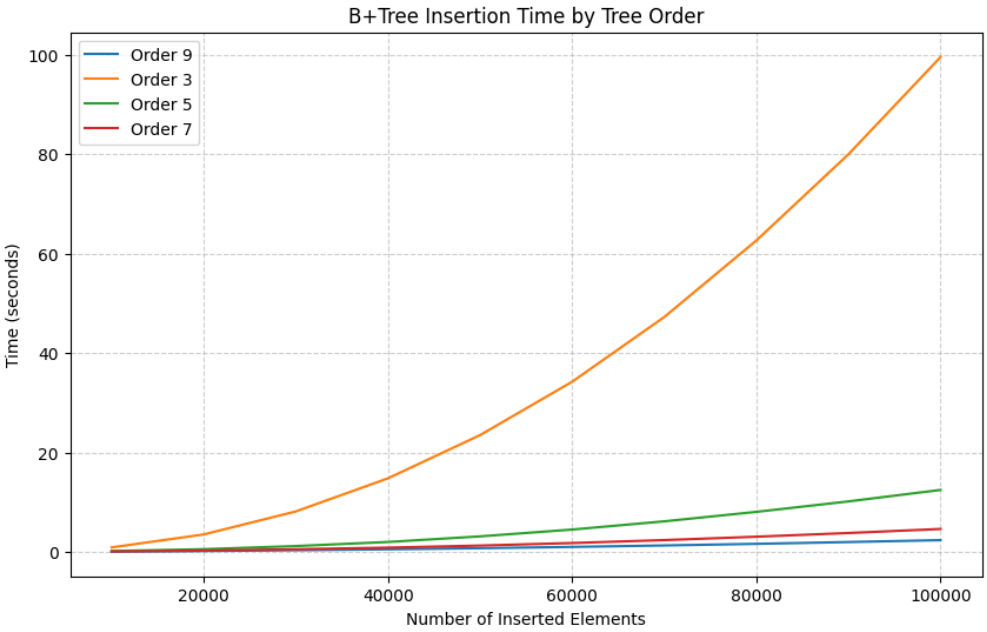

Figure 3: Insertion Time vs. B^+ Tree Order d for 10^5 Records

- **Order 9**: Achieved the best performance, completing the insertion of 10^5 records in approximately 2 seconds. This made it six times more efficient than order 5 and twice as efficient as order 7.

Interpretation of Results:

The observed performance improvements with higher orders are primarily attributable to the in-memory nature of our B^+ Tree implementation. Unlike disk-based B^+ Trees, where maximizing the order d is essential to reduce disk I/O by storing multiple nodes within a single disk page, our in-memory trees leverage the high-speed access of RAM. This allows for higher orders without incurring significant overhead, as memory access times are negligible compared to disk access latencies.

Moreover, an order of 200, while seemingly large, is not problematic in an in-memory context. Higher orders result in shallower tree structures, reducing the number of node accesses required during insertions and searches. In-memory storage ensures that these accesses are swift, and the ability to store multiple nodes in contiguous memory regions enhances cache efficiency, further optimizing performance.

5.6 Design Considerations and Problems Faced

In our probabilistic vector index, we need to determine how many candidate points, O, to fetch from the HNSW structure such that with high probability we retrieve at least k valid neighbors (i.e., those whose scalar value s lies in the required interval). Initially, we implemented a *linear* search to find this O: starting from $O = k$ and incrementing until

$$\Pr\big(X < k\big) \;=\; \sum_{i=0}^{k-1} Oip^i(1-p)^{O-i} \quad \leq \quad \alpha,$$

where $X \sim \text{Binomial}(O, p)$ and p is the fraction of valid items. Although conceptually straightforward, this *linear* approach is *slow* for large M (total dataset size). In the worst case, it can require up to $O(M)$ checks, which becomes a bottleneck as M grows.

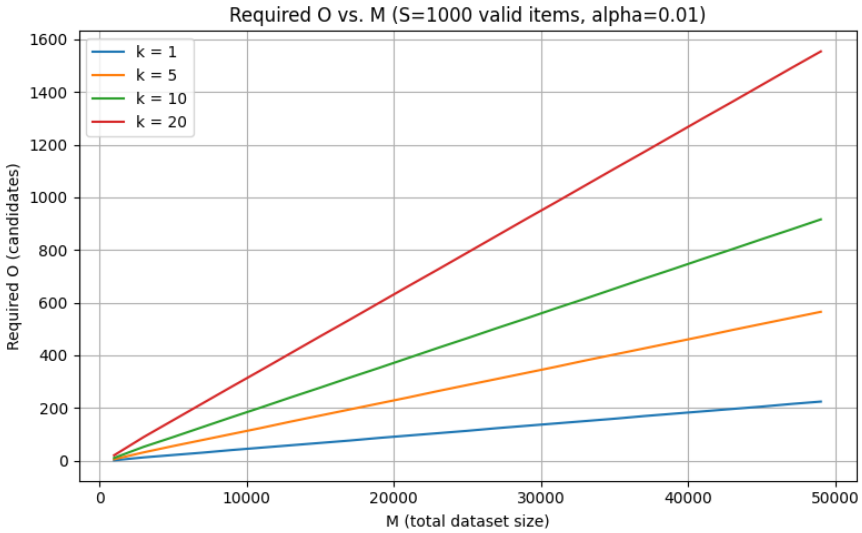

Figure 4: Number of candidates O required for varying dataset sizes M and neighbor count k. Each curve plots the smallest O that ensures, with confidence $1 - \alpha$, we retrieve at least k valid neighbors.

Monotonicity. The key insight to improve performance is that the function we evaluate, $\Pr(X < k)$, is *monotonically decreasing* in O. Intuitively, if we draw more samples O, the probability of getting fewer than k valid items should go down. Since the target condition $\Pr(X < k) \leq \alpha$ transitions from *false* to *true* exactly once as O increases, it is an ideal scenario for *binary search*. By doing a binary search over $O \in [k, M]$, we reduce the complexity from linear in M to logarithmic in M, greatly speeding up the query planning phase.

Required Number of Instances. Figure 4 shows how the required number of candidates O grows with different values of M and k. As expected, when k is larger (we need more valid neighbors) or when M is larger (the probability p might drop), the required O rises. Once M approaches the total number of valid items, or if k is small, O saturates toward a smaller value.

In our final design, therefore, we **(1)** rely on the *monotonicity* of the binomial cumulative distribution to apply a binary search for O rather than a linear scan, and **(2)** add a small safety margin (e.g., +100) to account for the approximate nature of HNSW retrieval. This ensures both *high performance* and *high recall* of the scalar-filtered neighbors.

```cpp
int computeRequiredO_BinarySearch(int M, int S, int k, double alpha) const {
    // Edge cases
    if (k <= 0) return 0;
    if (S <= 0) return k;
    if (S >= M) return k;
    if (alpha <= 0.0) return k;

    double p = static_cast<double>(S) / M;
    int left = k;
    int right = M;
    int best = M;

    while (left <= right) {
        int mid = (left + right) / 2;
        double probFewerThanK = binomialCDFLessThan(mid, k, p);

        if (probFewerThanK <= alpha) {
            best = mid;      // mid is good enough
            right = mid - 1; // try to find a smaller O
        } else {
            left = mid + 1;  // need bigger O
        }
    }
    return best;
}
```

Listing 3: Compute Required O Using Binary Search

5.7 Other Practical Issues

Additional nuances included:

- **Memory Layout:** Storing child pointers, key arrays, and value vectors contiguously helped keep memory accesses efficient, a notable plus in an in-memory setup.
- **Metadata Updates:** Maintaining `subtree_size` upon insertions and deletions added a small per-insertion overhead. However, since our tree height is $O(\log_d N)$ and d is relatively small, the overall additional cost is manageable.
- **Disk vs. In-Memory Concerns:** Many optimizations (like partial node caching or asynchronous splits) are more relevant in disk-based B$^+$ Tree systems. In an in-memory deployment, simpler designs can be sufficiently fast without extensive caching layers.

Overall, these considerations guided us toward a design that balances simplicity and efficiency. Although advanced data structures like segment trees, treaps, or other specialized indices may improve certain operations in specific scenarios, the *array-based* leaf and internal node structure, commonly presented in textbooks, remains an effective solution up to fairly large order sizes.

5.8 AI Tools usage

In the development and documentation of this project, AI tools were utilized to assist with code refinement, design optimization, node structure development, and the enhancement of the report's language.

However, the original idea, benchmarking processes, and the proposal of three distinct B^+ Tree implementations—each tailored for specific use cases—were conceived and executed independently by the author. The strategic decisions, performance evaluations, and customization of these data structures reflect the author's own research and expertise.

6 Useful Resources

This section provides a list of resources, that can be used to further understand concepts introduced in this report, to explore some external dependencies used (HNSWLib), or to further push this solution by investigating other solutions and possibly integrate ideas from existing solutions into this one.

6.1 Libraries and Tools

- **HNSWLib**
  A lightweight and efficient library for Approximate Nearest Neighbor (ANN) search using HNSW graphs. It is widely used for its simplicity and high performance.
  https://github.com/nmslib/hnswlib

- **HNSWlib Documentation**
  Authors: Yury Malkov and Danylo Yashunin
  Link: `https://github.com/nmslib/hnswlib/blob/master/README.md`
  Accessed: April 27, 2024.

- **SPTAG (Microsoft)**
  A library by Microsoft that supports fast ANN search using both graph-based and tree-based methods. It is particularly useful for hybrid indexing systems.
  https://github.com/microsoft/SPTAG

6.2 Research Papers

- **Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs**
  This foundational paper introduces the HNSW algorithm, which achieves logarithmic complexity scaling and outperforms traditional ANN methods.
  https://arxiv.org/abs/1603.09320