HNSW-B+tree Filter: Adaptive Pre-filtering for Approximate Nearest Neighbor Search

Project Report

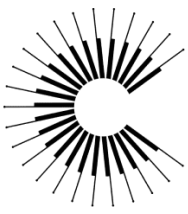
Data Management 2

By: **Ilyas Hakkou**

Supervised by:

Mr. Anas Ait Omar

Dr. Karima Echihabi



College of Computing

UM6P

University Mohammed VI Polytechnic

Table of Contents

Project Overview

This project implements an adaptive filtering mechanism that enhances HNSWLib's approximate nearest neighbor (ANN) search capabilities by incorporating a B+ tree for efficient pre-filtering based on attribute ranges. The system dynamically switches between brute-force and HNSW-based ANN search depending on the size of the filtered dataset, optimizing query performance across different scenarios.

Key Features

1. Custom B+ tree implementation optimized for range queries
2. Adaptive search strategy selection
3. Integration with HNSWLib
4. Support for high-dimensional vector data
5. Efficient handling of duplicate keys
6. Performance metrics tracking

Technical Architecture

The system consists of several key components working together:

1. **B+ Tree Index**

- Handles range-based filtering
- Maintains sorted keys for efficient range queries
- Supports duplicate keys through Alternative 3 implementation
- Provides O(log n) search complexity

2. **HNSW Index**

- Manages high-dimensional vector search
- Provides approximate nearest neighbor search capabilities
- Integrated with the filtering mechanism

3. **Adaptive Query Processor**

- Dynamically selects between brute-force and HNSW search
- Uses a threshold-based approach (30% of total data points)
- Optimizes query performance based on filtered set size

Implementation Details

B+ Tree Implementation

The B+ tree implementation (BPlusTree.h) is specifically designed for this use case with the following features:

1. **Key Operations**

    - Insert: O(log n)
    - Search: O(log n)
    - Range Query: O(log(n) + m), where m is the number of elements in range

2. **Duplicate Key Handling**

    - Uses Alternative 3 approach
    - Multiple record IDs can be associated with a single key

Query Processing Pipeline

1. **Initial Filtering**

```
vector<int> candidatesIDs = bptree->rangeSearch(lower, upper);
float filteredRatio = static_cast<float>(candidatesIDs.size()) / data.size();
```

1. **Search Strategy Selection**

```
if (filteredRatio > HNSW_THRESHOLD) {
// Use HNSW with post-filtering

} else {

// Use brute-force on filtered subset

}
```

Key Components

1. B+ Tree

The B+ tree implementation provides efficient range-based filtering through:

1. Leaf node linking for sequential access
2. Dynamic node splitting
3. Parent pointer maintenance
4. Optimized key distribution

2. HNSW Integration

The HNSW integration includes:

1. Dynamic index construction
2. Configurable search parameters
3. Post-filtering capability

4. Fallback mechanisms

3. Metrics Tracking

The system includes comprehensive metrics tracking implemented in the **metrics.h** file:

```
struct QueryMetrics {
	double qps;
	double totalRuntime;
	double avgRuntime;
	double recall;
	int totalQueries;
};
```

Build and Run Instructions

Prerequisites

1. C++ compiler with C++11 support

Building the Project

1. **Clone the repository**:

```
git clone https://github.com/IlyasIsHere/hnsw-bptree-filter.git
cd hnsw-bptree-filter
```

1. **Compile the project:**

```
g++ -o program main.cpp -Ihnswlib -std=c++11 -O3
```

1. **Run the program:**

```
./program.exe
```

Challenges and Solutions

To efficiently manage multiple record IDs for the same key, I implemented Alternative 3 (using a recordIDs vector in each node).

Another challenge was that at first, I didn't keep track of the parent of each node in the B+tree, but since I'm splitting recursively bottom-up to implement the insert operation, I needed to keep track of the parents, and update them accordingly.

During B+ tree testing, I discovered that the search() function failed to find vectors even when their keys existed in the tree. This occurred because direct floating-point comparisons in C++ are unreliable due to binary representation limitations. To solve this, we implemented an **areAlmostEqual()** function that considers two floats equal if their difference is smaller than 0.000001, ensuring reliable key matching throughout the tree operations.

Another major challenge is the weak recall for queries that use HNSW. I still haven't found a solution to it, but I think that is because HNSW doesn't return exactly the k nearest neighbors, but rather it only approximates them, which causes inconsistencies. However, using the B+Tree + brute-force approach I get a recall of 100%.

On the other hand, to guarantee that the HNSW returns vectors that satisfy the range constraint, I developed a two-phase filtering mechanism to guarantee range constraints are satisfied. First, I use the B+ tree to identify the valid range of vectors, but instead of immediately filtering the HNSW search space, I perform a wider HNSW search (getting k * 10 neighbors) on the full dataset. This oversampling approach ensures I have enough candidates that might fall within the range. Then, in the post-processing phase, I filter these candidates based on their range attribute (data[id][0]), only keeping those that satisfy the range constraints (lower <= value <= upper). If this filtered set doesn't yield enough neighbors (less than k), I have a fallback mechanism that switches to brute force search on the B+ tree filtered subset. This approach maintains HNSW's efficiency while guaranteeing that all returned vectors satisfy the range constraints, effectively combining the speed of approximate nearest neighbor search with exact range filtering.

Finally, to optimize the operation of getting the nearest k neighbors after calculating all the distances, I used a max-heap to efficiently get the k vectors with the smallest distances, instead of sorting them all.

Test Cases for B+ Tree

I created unit tests for the B+Tree operations to ensure correct functioning. The tests are in the **BPlusTreeTest.cpp** file.

1. **Insertion and Traversal**:
    - Validated that keys remain sorted in leaf nodes.
    - Checked linked-leaf traversal for full coverage of keys.
2. **Range Search**:
    - Tested retrieval of points within specific ranges.
    - Verified efficiency and accuracy for overlapping and non-overlapping ranges.
3. **Duplicate Keys**:
    - Ensured all associated records are retrieved correctly for duplicate keys.

To compile and run the tests, you can use these commands:

```
g++ .\BPlusTreeTest.cpp -o test

./test
```

Performance Evaluation

The system's performance is evaluated using several metrics:

- Queries Per Second (QPS)
- Average query runtime
- Total runtime (exluding index building phase)
- Recall rate (number of correct results / 100, as described in the dataset website)

To compute the recall, and since I do not have the exact results in some file, I assumed that the correct results are the ones I would get using a brute-force approach. So for each query I also ran a brute-force to get the ground truth results, and then compute the recall. At the end, the final recall is the average recall across all queries.

Performance Results

Below are the performance results for the "contest-data-release-1m.bin" dataset (1 million):

```
=== Performance Metrics ===
Queries Per Second (QPS): 11.5364
Total Runtime: 432.369 s
Average Runtime: 0.0866818 s
Recall: 51.8083%
Total Queries: 4988
============================
```

The small recall value comes mainly from the queries that were processed using HNSW. In fact when I only use the B+tree and brute-force approach, here are the results I get:

```
=== Performance Metrics ===
Queries Per Second (QPS): 4.49589
Total Runtime: 1109.46 s
Average Runtime: 0.222425 s
Recall: 99.9749%
Total Queries: 4988
============================
```

AI Usage

I used Claude AI to help me write the report, and it also helped me break down the functions I need to implement the insert() function in the B+Tree. I also used it to automate the repetitive tasks like printing/debugging, and to write documentation for my functions.

Conclusion

The HNSW-B+tree Filter project successfully combines traditional data structures with modern search techniques, demonstrating how a B+ tree can enhance approximate nearest neighbor search.

Implementing the B+ tree from scratch provided me valuable insights into advanced data structure design, while its integration with HNSW showed how hybrid approaches can significantly improve real-world search performance, despite having a trade-off between recall and runtime.

References

- https://dbgroup.cs.tsinghua.edu.cn/sigmod2024/task.shtml?content=description
- https://github.com/nmslib/hnswlib
- https://www.geeksforgeeks.org/cpp-program-to-implement-b-plus-tree/