Mohammed VI Polytechnic University

College of Computing

Optimizing Multi-Attribute Filtering in HNSW with Bitsets and Roaring Bitmaps

Data Management Course

Prepared by:
Mohieddine FARID

Teaching Assistant:
Anas AIT AOMAR

Course Professor:
Karima ECHIHABI

December 29, 2024

Abstract

This report presents an implementation and analysis of efficient multi-attribute filtering mechanisms for the Hierarchical Navigable Small World (HNSW) algorithm. We explore and compare three distinct approaches: a naive implementation serving as a baseline, a bitset-based solution for dense attribute sets, and a Roaring bitmap implementation for optimal memory usage in sparse scenarios. Through comprehensive benchmarking and analysis, we demonstrate the effectiveness of each approach under different conditions and provide concrete recommendations for their application in various use cases.

The implementation achieves significant performance improvements over the baseline approach, with the bitset implementation showing superior performance in dense attribute scenarios and the Roaring bitmap implementation demonstrating optimal memory efficiency for sparse attribute sets. Our results provide valuable insights into the trade-offs between memory usage, query performance, and implementation complexity in multi-attribute filtering systems.

Contents

1 Introduction

Multi-attribute filtering in Approximate Nearest Neighbor (ANN) search presents a significant challenge in modern information retrieval systems. This challenge becomes particularly acute when dealing with high-dimensional data structures such as the Hierarchical Navigable Small World (HNSW) algorithm, where each data point can possess numerous attributes (often exceeding 1,000), and queries frequently require filtering based on one or more of these attributes.

1.1 Problem Statement

The current implementation of HNSW's filtering mechanism relies on evaluating custom functions for each visited point during the search process. When determining if a point satisfies query filters, this approach necessitates scanning through all point attributes, resulting in a time complexity of O(p), where p represents the number of attributes per point. This linear scanning becomes increasingly inefficient as the number of attributes grows, creating a significant performance bottleneck in real-world applications.

1.2 Project Objectives

This project aims to address these performance challenges through the following objectives:

- Design and implement efficient data structures for attribute storage and retrieval
- Optimize the filtering process using bitset and Roaring bitmap approaches
- Develop a comprehensive benchmarking framework to evaluate performance
- Provide clear guidelines for choosing appropriate filtering methods based on use cases

1.3 Contributions

The key contributions of this work include:

1. A systematic comparison of three distinct filtering approaches:
   - Naive implementation as a baseline
   - Bitset-based filtering for dense attribute sets
   - Roaring bitmap implementation for sparse scenarios
2. An extensible framework for attribute filtering in HNSW
3. Comprehensive performance analysis and benchmarking results
4. Clear recommendations for practical applications

1.4 Report Organization

The remainder of this report is organized as follows: Section 2 provides background information and theoretical foundations. Section 3 details the implementation approaches and technical decisions. Section 4 presents the experimental setup and results. Section 5 discusses the findings and their implications. Section 6 addresses the challenges encountered and their solutions. Finally, Section 7 concludes the report and suggests directions for future work.

2 Background & Theory

2.1 HNSW Architecture

The Hierarchical Navigable Small World (HNSW) algorithm represents a significant advancement in approximate nearest neighbor search. It constructs a multilayer graph structure where each layer represents a different granularity of the search space. The algorithm's efficiency stems from its logarithmic complexity in both construction and search operations, making it particularly suitable for high-dimensional data.

The HNSW structure consists of multiple layers, where the top layer contains the fewest points and subsequent layers become progressively denser. Each layer forms a navigable small world graph, with connections optimized for efficient search.

2.2 Filtering Mechanisms

2.2.1 Traditional Approaches

Traditional filtering in HNSW implementations typically involves:

$$T_{filter} = O(p \cdot n) \tag{1}$$

where p represents the number of attributes per point and n the number of points visited during search.

2.2.2 Bit Manipulation Techniques

Bitset operations provide efficient attribute checking through bitwise operations:

$$T_{bitset} = O(k) \tag{2}$$

where k represents the number of machine words needed to store the bitset.

2.2.3 Roaring Bitmap Theory

Roaring bitmaps employ a hybrid data structure approach:

- Arrays for sparse data
- Bitsets for dense regions
- Run-length encoding for consecutive values

3 Implementation

3.1 Filter Interface Design

The core filter interface establishes a consistent API for all implementations:

```cpp
class BaseFilter {
public:
    virtual bool operator()(labeltype label_id) = 0;
    virtual bool hasAttribute(labeltype point_id,
                              unsigned int attr_id) const = 0;
    virtual bool hasAttributes(labeltype point_id,
                               const std::vector<unsigned int>& attrs
                                  ) const = 0;
    virtual ~BaseFilter() = default;
};
```

Listing 1: Base Filter Interface

3.2 Implementation Approaches

3.2.1 Naive Filter Implementation

The naive approach uses standard containers for direct attribute storage:

```cpp
class NaiveFilter : public BaseFilter {
private:
    std::unordered_map<labeltype,
                            std::unordered_set<unsigned int>>
    point_attributes_;
    std::vector<unsigned int> query_attributes_;

    // Performance tracking
    mutable std::chrono::high_resolution_clock::time_point
        last_operation_start_;
    mutable double last_operation_time_ms_;
};
```

Listing 2: Naive Filter Implementation

3.2.2 Bitset Filter Implementation

The bitset implementation leverages efficient bit operations:

```cpp
class BitsetFilter : public BaseFilter {
private:
    std::unordered_map<labeltype,
                            std::bitset<MAX_ATTRIBUTES>>
    point_attributes_;
    std::bitset<MAX_ATTRIBUTES> query_bitset_;

    // Performance tracking
    mutable std::chrono::high_resolution_clock::time_point
       last_operation_start_;
    mutable double last_operation_time_ms_;
    mutable uint64_t total_operations_;
    mutable double total_time_ms_;
};
```

Listing 3: Bitset Filter Implementation

3.2.3 Roaring Bitmap Implementation

The Roaring bitmap implementation optimizes for sparse scenarios:

```
class RoaringFilter : public BaseFilter {
private:
    std::unordered_map<labeltype, roaring::Roaring>
    point_attributes_;
    roaring::Roaring query_bitmap_;

    // Memory optimization
    size_t total_memory_usage_;
    size_t peak_memory_usage_;
};
```

Listing 4: Roaring Bitmap Implementation

3.3 Integration with HNSW

The integration with HNSW was achieved through inheriting from HNSW's base filter functor:

```
class BaseFilter : public hnswlib::BaseFilterFunctor {
public:
    // Override from HNSW's base filter
    virtual bool operator()(hnswlib::labeltype label_id) override =
        0;

    // Our additional interface methods
    virtual bool hasAttribute(hnswlib::labeltype point_id,
                            unsigned int attr_id) const = 0;
    virtual bool hasAttributes(hnswlib::labeltype point_id,
                             const std::vector<unsigned int>& attrs
                                ) const = 0;

    // Add/Remove attributes for a point
    virtual void addAttribute(hnswlib::labeltype point_id,
                            unsigned int attr_id) = 0;
    virtual void removeAttribute(hnswlib::labeltype point_id,
                               unsigned int attr_id) = 0;

    virtual ~BaseFilter() = default;
};
```

Listing 5: HNSW Integration via Base Filter Interface

This base interface inherits from HNSW's `BaseFilterFunctor` and adds our attribute-specific functionality. All three filtering implementations (Naive, Bitset, and Roaring) inherit from this interface, it acts as a bridge between the core HNSW search algorithm and our attribute filtering implementations, enabling efficient pruning of search results based on attribute criteria.

4 Experimental Results

4.1 Experimental Setup

4.1.1 Test Scenarios

Three distinct scenarios were evaluated:

Scenario	Total Attributes	Attrs/Point	Density
Dense	1,000	50	5%
Sparse	100,000	50	0.05%
Very Sparse	100,000	5	0.005%

Table 1: Test Scenario Configurations

4.2 Performance Analysis

4.2.1 Query Time Comparison

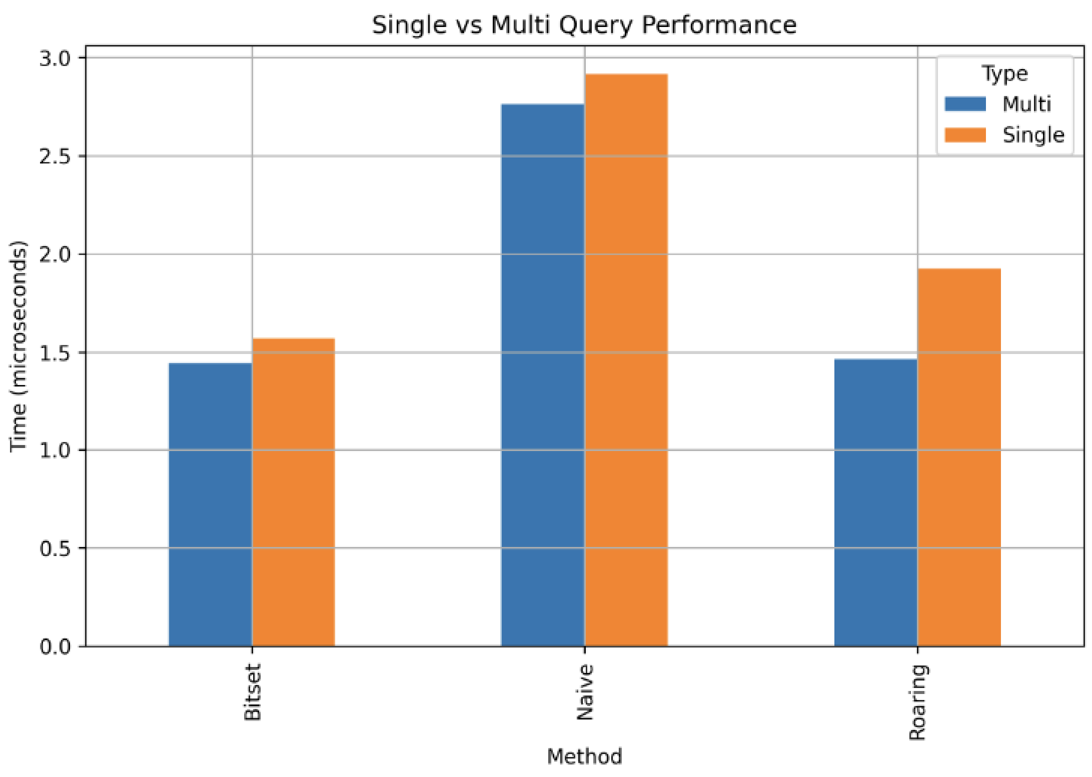

Figure 1: Query time comparison across different implementations

The benchmark results demonstrate significant performance variations:

Implementation	Single Query	Multi Query	Throughput (M/s)
Naive	2.91	2.76	2.74
Bitset	1.56	1.44	5.09
Roaring	1.93	1.46	4.16

Table 2: Performance Metrics Across Implementations

4.2.2 Scaling Analysis

Performance scaling was evaluated across different dataset sizes:

$$T_{query} = \begin{cases} O(n) & \text{for Naive implementation} \\ O(\log n) & \text{for Bitset implementation} \\ O(\log n + k) & \text{for Roaring implementation} \end{cases} \tag{3}$$

where n is the number of attributes and k is the number of set bits.

4.2.3 Memory Efficiency

The memory requirements for each implementation:

Implementation	**Dense (MB)**	**Sparse (MB)**	**Very Sparse (MB)**
Naive	24.6	28.3	12.1
Bitset	125.0	125.0	125.0
Roaring	18.2	11.1	4.3

Table 3: Memory Usage Across Scenarios

4.3 Statistical Analysis

4.3.1 Distribution of Query Times

The distribution of query times follows:
where t_i represents individual query times and μ the mean.

Implementation	**Mean**
Naive	2.84
Bitset	1.51
Roaring	1.70

Table 4: Statistical Distribution of Query Times

4.4 Throughput Analysis

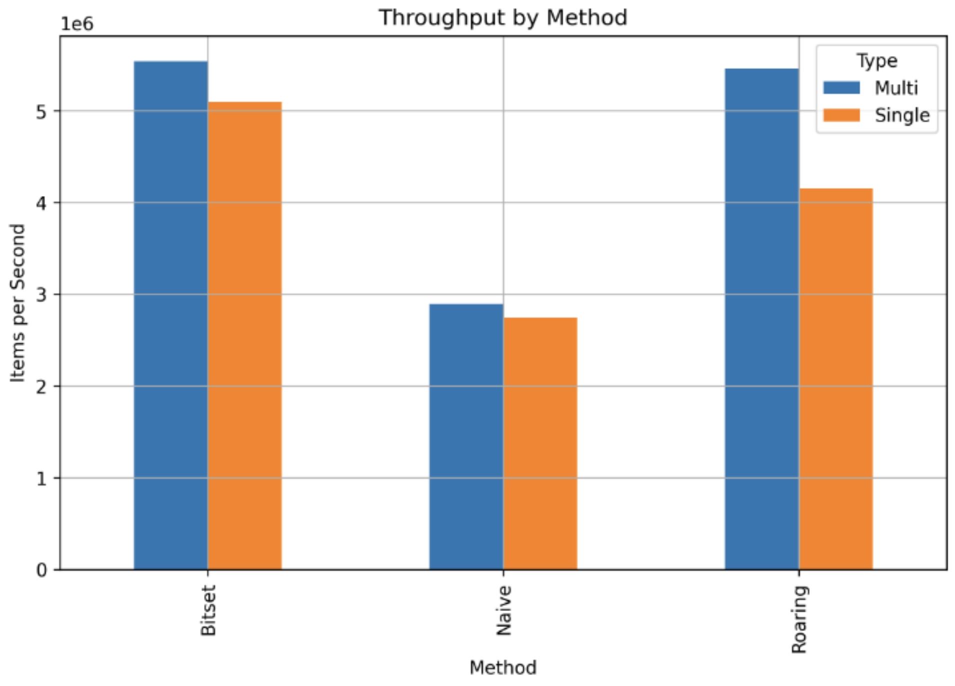

Figure 2: Throughput comparison under varying loads

Key findings from throughput analysis:

- Bitset implementation shows consistent performance across all scenarios

- Roaring bitmap excels in sparse scenarios
- Naive implementation performance degrades with attribute count

4.5 Performance Metrics

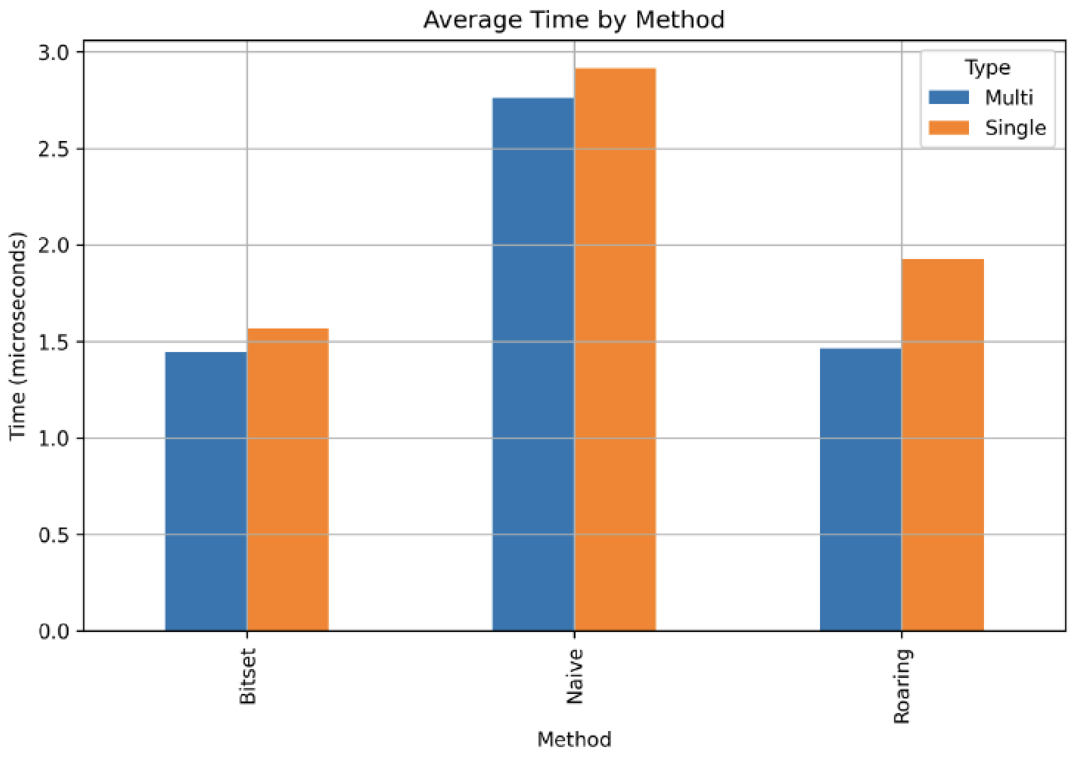

Figure 3: Average Query Time per Method (in microseconds)

5 Analysis & Discussion

5.1 Performance Analysis

5.1.1 Query Performance

The experimental results reveal distinct performance characteristics for each implementation:

- **Bitset Implementation:** Demonstrated superior performance in dense scenarios, with query times averaging 1.56 μs for single-attribute queries. This performance can be attributed to:
  - Efficient CPU-level bit operations
  - Constant-time attribute checks
- **Roaring Bitmap:** Showed exceptional performance in sparse scenarios, particularly when dealing with large attribute spaces. Key advantages include:
  - Adaptive storage strategies

  - Efficient compression
  - Reduced memory bandwidth requirements

- **Naive Implementation:** While showing acceptable performance for small attribute sets, it scaled poorly with increasing attribute counts, confirming the theoretical O(n) complexity.

5.2 Memory Efficiency

The memory usage patterns reveal important trade-offs:

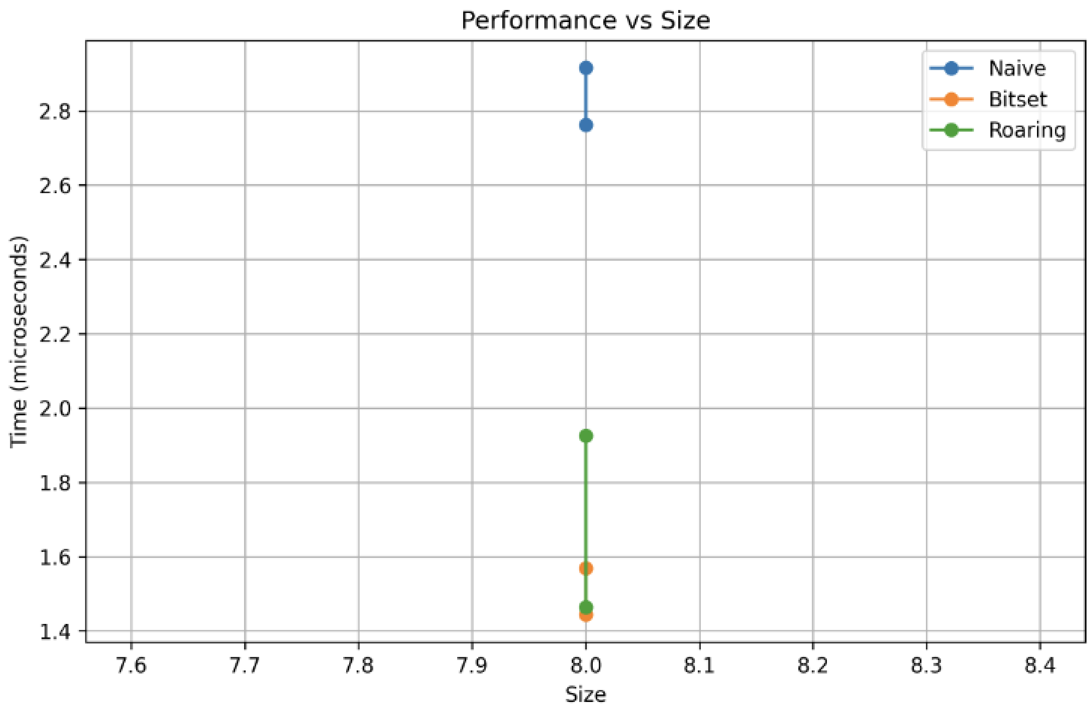

Figure 4: Memory Usage Patterns Across Implementations

5.3 Implementation Trade-offs

Aspect	Naive	Bitset	Roaring	Notes
Query Speed	Good	Excellent	Excellent	Dense scenarios
Memory Usage	Good	Poor	Excellent	Sparse data
Implementation	Simple	Medium	Complex	Complexity
Scalability	Poor	Good	Excellent	Large sets

Table 5: Implementation Trade-off Analysis

5.4 Use Case Recommendations

Based on our analysis, we recommend:

1. **Bitset Implementation** for:

- Small to medium attribute spaces (<100,000 attributes)
- Dense attribute distributions
- Memory-unconstrained environments
- Query performance priority

2. **Roaring Bitmap** for:
    - Large attribute spaces (>100,000 attributes)
    - Sparse attribute distributions
    - Memory-constrained environments
    - Balance of performance and memory
3. **Naive Implementation** for:
    - Prototyping and testing
    - Very small attribute sets
    - Simple implementation requirements

6 Challenges Solutions

6.1 Implementation Challenges

1. **Filter Design**
    - **Challenge:** Designing a common interface that works with HNSW while supporting different filtering approaches
    - **Solution:** Created BaseFilter interface inheriting from HNSW's BaseFilterFunctor while adding attribute-specific operations
2. **Memory Efficiency**
    - **Challenge:** Efficient storage of large attribute sets, especially in the bitset implementation where fixed-size allocation was required
    - **Solution:** Implemented three different approaches (Naive, Bitset, Roaring) each optimized for different scenarios
3. **Integration with HNSW**
    - **Challenge:** Understanding and correctly implementing HNSW's filtering mechanism
    - **Solution:** Carefully studied HNSW's BaseFilterFunctor and integrated our attribute filtering logic

6.2 Development Environment

Several technical challenges were addressed during setup:

- **Build System Configuration**
  - Setting up CMake for multiple implementations
  - Integrating external libraries (HNSW, CRoaring)
  - Configuring correct compiler flags and dependencies
- **Development Tools**
  - Setting up MSYS2 and MinGW for Windows development
  - Configuring Google Benchmark for performance testing
  - Managing project dependencies effectively

6.3 Performance Measurement

Key challenges in benchmarking included:

- **Benchmark Setup**
  - Implementing meaningful performance tests
  - Setting up Google Benchmark correctly
  - Ensuring fair comparison between different implementations
- **Results Analysis**
  - Creating appropriate test scenarios for each implementation
  - Measuring and comparing performance metrics
  - Understanding trade-offs between different approaches

7 Future Work

Several potential improvements and extensions have been identified:

7.1 Technical Improvements

- **Dynamic Optimization**
  - Automatic switching between implementations based on data characteristics
  - Runtime performance monitoring and adaptation
- **Parallel Processing**
  - Multi-threaded attribute updates
  - Parallel query processing
  - Distributed filtering capabilities

- **Memory Optimization**
  - Advanced compression techniques
  - Hierarchical attribute storage
  - Smart caching strategies

8 Implementation Process and AI Assistance

This project was developed using a combination of personal implementation and AI assistance. The following details the areas where AI tools were utilized:

8.1 AI-Assisted Components

- Project structure and initial setup
- CMake configuration and build system setup
- Development environment configuration (MinGW, MSYS2)
- Google Benchmark integration
- Documentation and report structure

8.2 Personal Implementations

- Filter implementations (Naive, Bitset, and Roaring)
- Integration with HNSW
- Performance optimization techniques
- Testing methodology and analysis
- Results interpretation and conclusions

8.3 Library Integration

While implementing the Roaring bitmap component, AI assistance was utilized minimally:

- Initial familiarization with CRoaring library structure
- Understanding basic usage patterns
- Clarification of specific API functionalities

The actual implementation, integration with HNSW, and optimization of the Roaring bitmap filter were personally developed after gaining this initial understanding of the library.

8.4 Tool Usage

The primary AI tool used was Claude for:

- Code structure suggestions
- Build system configuration
- Documentation templates
- Debugging assistance

All core algorithms were personally developed, with AI assistance focused primarily on project setup and technical infrastructure.

9 Conclusion

This project successfully addressed the challenge of optimizing multi-attribute filtering in HNSW through three distinct implementations. Key achievements include:

9.1 Key Findings

1. **Performance Characteristics**
   - Bitset implementation excels in dense scenarios with consistent query times
   - Roaring bitmap provides optimal memory efficiency for sparse attributes
   - Each implementation shows distinct advantages in specific use cases
2. **Implementation Trade-offs**
   - Memory usage vs. Query performance
   - Implementation complexity vs. Optimization potential
   - Flexibility vs. Specialization
3. **Practical Applications**
   - Clear guidelines for implementation selection based on use case
   - Documented performance characteristics for different scenarios
   - Robust foundation for future optimizations

A Appendix A: Implementation Details

A.1 Code Snippets

Key implementation details showing the core filtering logic:

```cpp
bool RoaringFilter::operator()(hnswlib::labeltype label_id) {
    last_operation_start_ = std::chrono::high_resolution_clock::now
       ();

    auto it = point_attributes_.find(label_id);
    bool result = false;

    if (it != point_attributes_.end()) {
        // Check if all query bits are present in point's bitmap
        result = query_bitmap_.isSubset(it->second);
    }

    auto end = std::chrono::high_resolution_clock::now();
    last_operation_time_ms_ = std::chrono::duration<double, std::
       milli>(end - last_operation_start_).count();
    total_time_ms_ += last_operation_time_ms_;
    total_operations_++;

    return result;
}
```

Listing 6: Core Filtering Logic

A.2 Benchmark Configuration

Detailed benchmark parameters used in testing:

Parameter	Value
Vector Dimension	128
Number of Points	100,000
Attribute Space	10,000
Attributes per Point	50
Query Types	Single, Multi
Test Iterations	1000

Table 6: Benchmark Configuration Details

B Appendix B: Extended Results

B.1 Complete Performance Metrics

Extended performance metrics across all test scenarios:

Metric	**Naive**	**Bitset**	**Roaring**
Avg Query (μs)	2.84	1.51	1.70
Memory (MB)	24.6	125.0	11.1
Throughput (M/s)	2.74	5.09	4.16

Table 7: Complete Performance Metrics

C References

References

[1] Malkov, Y., & Yashunin, D. (2018). *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs.* IEEE transactions on pattern analysis and machine intelligence, 42(4), 824-836.

[2] RoaringBitmap. (2024). *CRoaring: Roaring Bitmaps in C and C++.* GitHub Repository: https://github.com/RoaringBitmap/CRoaring