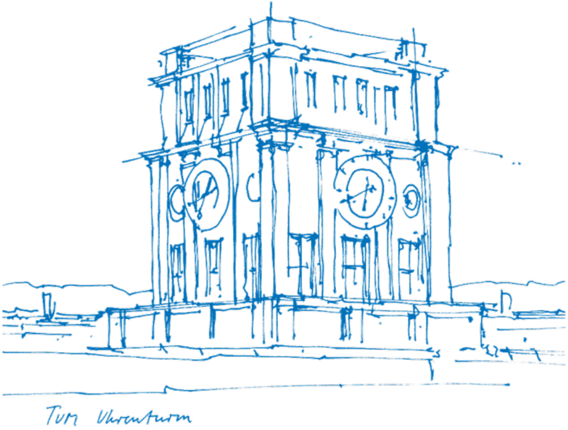
Hist-Tree

An Efficient Indexing Data Structure

Mert Sidal

School of Computation, Information and Technology
Decentralized Information Systems and Data Management
Technical University of Munich

February 6th, 2025



Outline

Hist-Tree

- **Index** for fast approximate Lookups
- **Basic Assumptions**: Sortedness and Range of Data
- **Idea**: Histogram to partition Data into equal-sized Bins
- Physically organized into **two** Arrays of 32-bit Integers
    1. Inner Nodes **with** Child Pointers
    2. Leaf Nodes

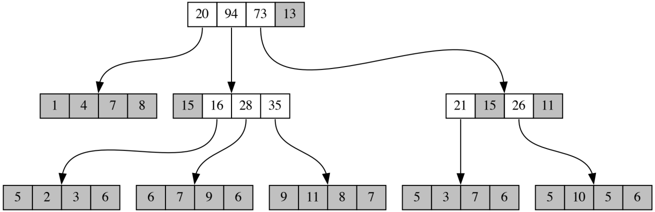


Figure 1 Example Hist-Tree with 200 keys in the range $[0, 1000)$

Outline

1 Background

2 Implementation Details

3 Testing & Benchmarking

4 Conclusion

Components

Builder.h:

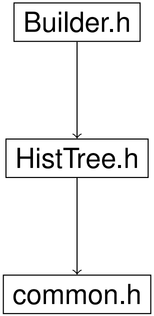
- `build()`

HistTree.h:

- `getSearchBound(key)`
- `remove(key)`
- `insert(key)`

common.h:

- Utilites: `SearchBound` struct, `Visualizer` class and `RebuildContext` struct



Tree Construction Algorithm

Algorithm 1 HistTree Construction

```
create bit vector from sorted keys
initial partition into bins
while nodes need processing do
    count keys in current bins
    if bin count < error bound then
        stop and create leaf
    else
        create inner node and split further
    end if
end while
```

Lookup Algorithm

Algorithm 2 HistTree Lookup

```
handle edge cases
if inner nodes is empty then
    return direct bin lookup in leaf nodes
end if
while not at leaf do
    calculate bin for current level
    accumulate counts of previous bins
    traverse to next node or return
    adjust key and bin width for next level
end while
return search bound in sorted array
```

Remove Algorithm

Algorithm 3 HistTree Remove

```
if key is min/max bound then
    rebuild
    return
end if
check if key exists
reset bit in bit vector
while traversing tree do
    decrement bin count
    if node becomes sparse then
        convert to leaf and cleanup children
    end if
end while
```

Insert Algorithm

Algorithm 4 HistTree Insert

```
check if key already exists
set bit in bit vector
while traversing tree do
    increment bin count
    if count exceeds error bound then
        rebuild
        return
    end if
end while
```

A Collection of Problems & Solutions

Problems	Solutions
No clear construction concept in paper	Developed custom bit vector approach
Limitations of `std::bitset` leading to use of `std::vector<bool>`	Transitioned to more performant `boost::dynamic_bitset`
Complexity of keeping the Tree Structure during Updates	Hybrid Implementation of Updates that trigger a Rebuild **if needed**

Hotspot Analysis: Partition Bit Vector

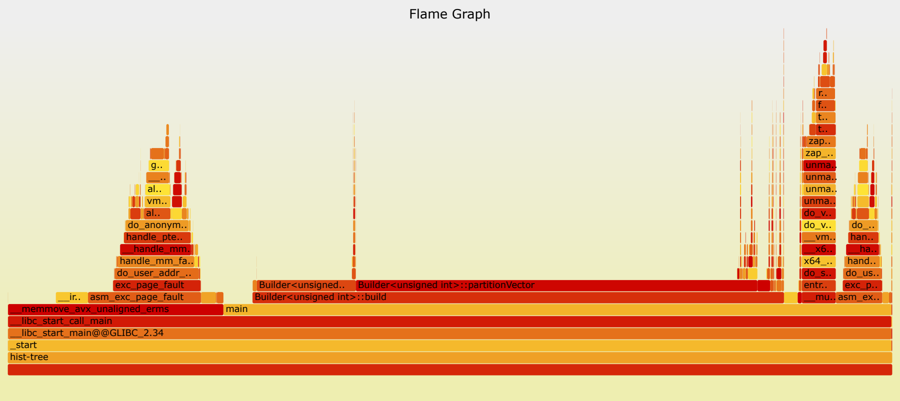


Figure 2 Performance Profiling of Build Function

Bottleneck: partitionVector()

- Repeatedly invoked during Tree Construction
- No Standardized Implementation
- **Base Implementation**: Manual bit-by-bit Copying ($\mathcal{O}(n)$)
- **Optimization Approaches**:
  - `memcpy` → no support for `boost::dynamic_bitset<>`
  - Bitwise Operations
  - SIMD
  - OpenMP
  - **Final**: Hybrid consisting of SIMD and OpenMP
- **Speedup for large Datasets** approximately 30%

Outline

Testing Approach

Testing Methodology

- Google Test Framework for Unit-Tests
- **Builder Tests:** Constructor, Bit Vector Computations, Build Mechanism
- **HistTree Tests:** Lookup, Insert, Remove
- **Key Challenge:** No Reference Implementation

Areas for Improvement

- Cover more Edge Cases and Boundary Conditions
- Add Tests for Utility Functions in `common.h`

SOSD Benchmark

Situation

- SOSD: Benchmark Suite for Learned Indexes
- Execution Blocked by Memory Constraints

Why?

- Not enough RAM
- `partitionVector()` and `createBitVector()` allocate vectors/bitsets partly across entire `range_`
- Datasets utilize full Key Range, causing exponential Memory Growth

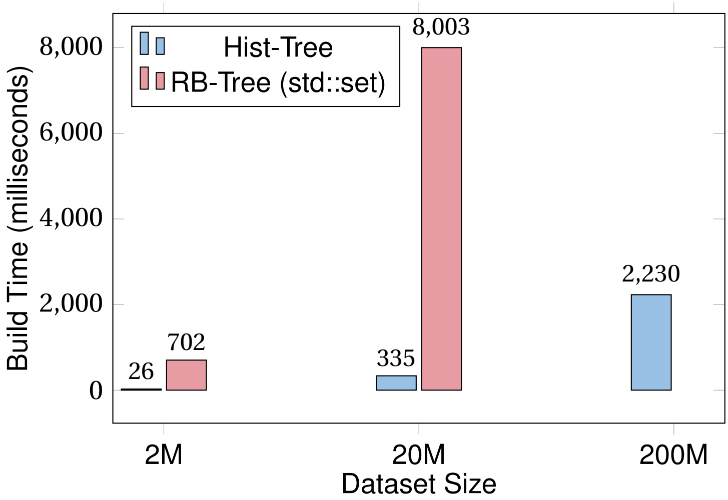
Possible Solutions

- Alternative Build Process without bit vectors
- Bit Vector Compression using Run-Length Encoding (RLE)

Benchmark Approach

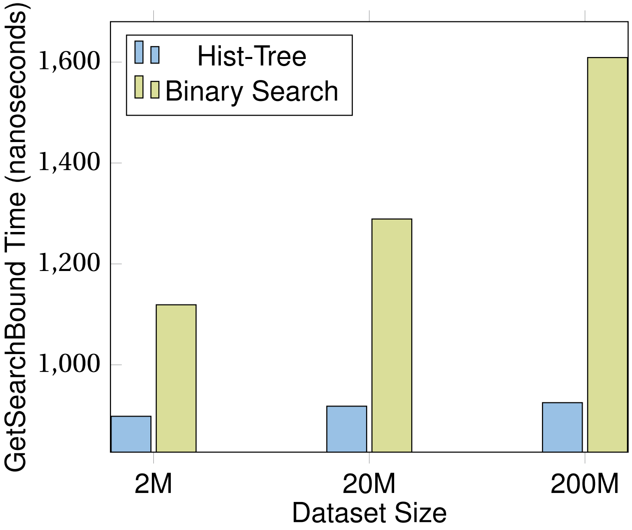
- Benchmarking Framework: Google Benchmark
- Hardware Specifications:
  - CPU: 8 x 2650.12 MHz cores
  - RAM: 8 GB
- Methodology:
  - **Dense** Synthetic Data: sizes 2M–200M
  - Operations Tested:
    - Tree Construction
    - Search Bound Retrieval
    - Insertion & Removal
  - Parameters:
    - Bins: 32–128
    - MaxError: 1024–8192
- **Following Results shown for Bins=64, MaxError=2048**

Benchmark: Construction Time Comparison

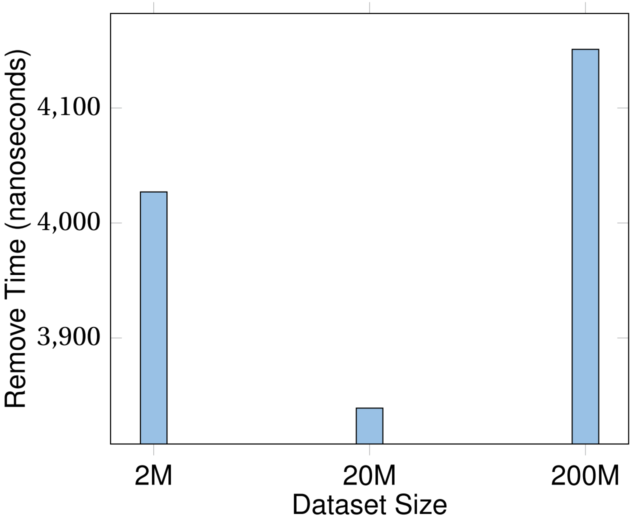


- Hist-Tree Build-Time grows non-linearly
- RB-Tree Construction:
  - 27x slower for 2M
  - 24x slower for 20M
  - Fails at 200M due to memory constraints

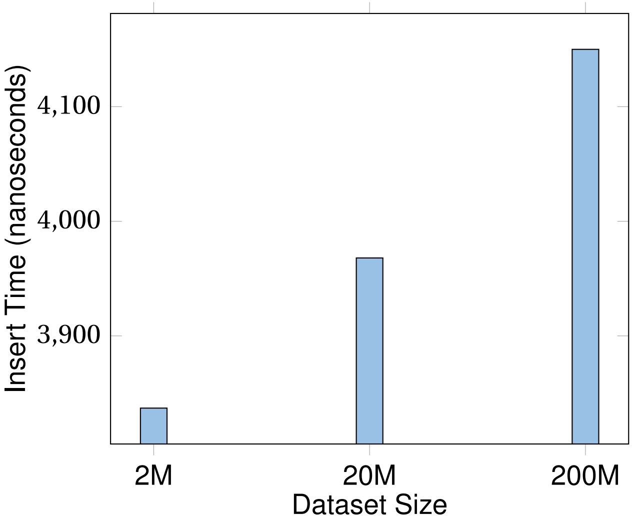
Benchmark: GetSearchBound



Benchmark: Remove

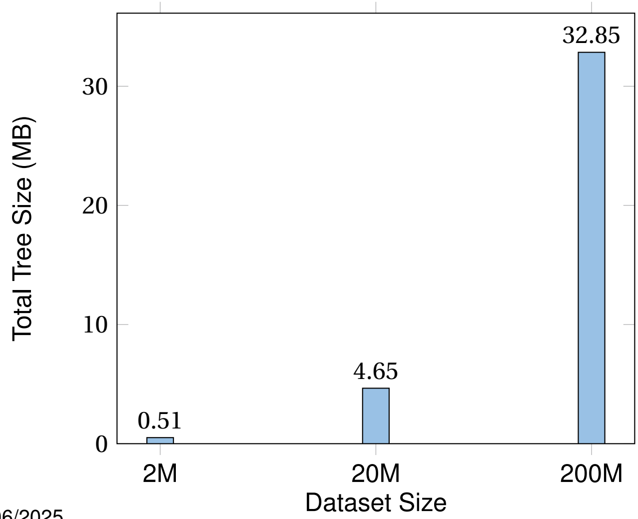


Benchmark: Insert



Benchmark: Memory Footprint of Resulting Hist-Tree

Compression ratio for 200M dataset: 32.846 MB vs. 800 MB raw data size (96% space reduction)



Backup: SOSD Competitors' Memory Footprint

Index / Index Size	XS Up to 0.01% of data size	S Up to 0.1% of data size	M Up to 1% of data size	L Up to 10% of data size	XL No limit
RMI	24.59 KB	24.59 KB	24.59 KB	24.59 KB	24.59 KB
RS	0.17 KB	0.17 KB	0.17 KB	0.17 KB	0.17 KB
PGM	0.37 KB	0.37 KB	0.37 KB	0.37 KB	0.37 KB
BTree	43.07 KB	680.93 KB	5.44 MB	43.54 MB	174.15 MB
FAST		416.77 KB	6.67 MB	6.67 MB	1.71 GB
ALEX		423.1 KB	6.77 MB	54.13 MB	866.1 MB
BinarySearch	0.0 KB	0.0 KB	0.0 KB	0.0 KB	0.0 KB

Figure 3 SOSD Competitors' Memory Footprint (Synthetic Uniform Dense Data (200M))

Outline

Challenges & Potential Improvements

Current Limitations

- High Memory overhead during Construction
- Costly full Rebuilds during **special** Updates
- Potential Data Fragmentation after multiple Removes

Future Steps

- Optimize Build Process
  - Implement RLE (Run-Length Encoding) approach
  - Rethink the Paper's Build Approach
- Partial Rebuild Strategies
- Defragmentation Techniques
- SOSD: The Revenge

Key Takeaways

- Result has small Memory Footprint
- Near-constant getSearchBound (around 900 ns)
- Needs scalable Build