Project Proposal

Applicability of Non-Asymptotic Optimizations in Range Filters

Programme of Study: BSc Computer Science and Artificial Intelligence

Surname: Ogurtsov

Name: Andrei

Student ID: 1220718867

University Email Address: a.ogurtsov@nup.ac.cy

Supervisor: Artem Anisimov

Word count: 1,780 (excl. refs)

Abstract

In this dissertation, I implement and evaluate **Range Filters** for LSM-tree–based (Log-Structured Merge-tree) storage systems, focusing on real CPU performance. The focus of the research is on the implementation of the **Hollow Z Fast Trie** [1] structure, combined with Monotone Minimal Perfect Hashing (**MMPH**) algorithms [2], with a performance study.

Unlike most academic works that focus on improving theoretical asymptotics, I measure and reduce CPU-level costs, doing "non-asymptotic" optimizations, working with constant-factor costs on real CPUs: data placement in memory (*Memory Layout*), minimizing branch prediction errors (*branch mispredictions*), and using specialized BMI instructions of processors (**BMI2, AVX512**).

My goal is to research is that possible to reduce CPU operation time in range filters using specific optimizations. The data structure, that was chosen for experiments has the lowest possible memory usage limit. I expect the application of hardware and algorithmic optimizations to allow maintaining the compactness of the structure, while maximizing lookups per second for single-core CPU, which is very important for LSM engines where metadata checks are CPU-bound, even if storage is fast enough.

The project repository is available at:
`https://github.com/OgurtsovAndrei/Thesis`

Contents

1 Purpose and Objectives

Storage engines (NoSQL databases, search engines, time-series processing systems) such as RocksDB, LevelDB, Cassandra, and HBase, use the **LSM-tree** (Log-Structured Merge-tree) structure at their core. This concept, formalized by Patrick O'Neil[3], changes random writes into sequential IO by accumulating data in RAM (*Memtable*) and periodically flushing it to disk as immutable Sorted String Tables (**SSTables**). This matches the strengths of SSD/HDD throughput, where sequential writes are much cheaper than random writes.

SSTables are organized in levels. On reads, a missing key may require checking multiple SSTables across levels, which leads to "read amplification": confirming that data is absent requires performing many Disk IO operations (we should check entry absence in all levels, all files). LSM engines attach filters like **Bloom Filters**[4] to each SSTable to avoid disk IO. A negative result guarantees the key is absent; a positive result may be a false positive (FP). However, classic Bloom Filters only support point queries. Modern applications often require range queries (e.g., `SELECT * WHERE key BETWEEN 'A' AND 'B'`). Using point filters for range search tasks leads to performance degradation, from $O(1)$ to $O(L)$, where L is the range length.

The purpose of this research is to address the performance limitations of existing **Approximate Range Filters** (like SuRF[5] or Rosetta[6]), where constant-factor overheads often dominate. While asymptotically fast, they remain CPU-heavy in practice. This study focuses on the **Hollow Z-Fast Trie**[1], adapting this theoretically optimal structure using "non-asymptotic" hardware optimizations to minimize real-world query processing time.

Objectives

Main Goal: To verify the applicability of non-asymptotic optimizations for reducing CPU query time in Range Filter data structures, specifically adapting the Hollow Z Fast Trie for modern CPU architectures.

Specific Research Objectives:

1. **Theoretical Analysis:** Study algorithms for constructing **Approximate Range Emptiness (ARE)** in linear time and analyze existing Z Fast Trie implementations.

2. **Succinct Structures Development:** Implement a **Succinct Bit Vector** supporting `Rank` and `Select` operations in constant time $O(1)$ using $N + o(N)$ bits of memory.

3. **Hollow Z Fast Trie Implementation:**

   - Implement the deterministic version of the structure.
   - Develop a probabilistic extension (*Probabilistic Trie*) to ensure a given ϵ.
   - Integrate the table of Encoded Handles.

4. **MMPH Development:** Implement monotone minimal perfect hashing (MMPH) to effectively map keys to ranks. Investigate *Time-optimized* (bucketing and LCP) and *Space-optimized* (relative ranking) variants.

5. **Hardware Optimization:**

   - Utilize **BMI2** instructions (`PDEP`/`PEXT`) to speed up bit index manipulations.
   - Experiment with **AVX512-BITALG** for parallel bit counting (Popcount).
   - Optimize the *Memory Layout* to ensure sequential access and reduce cache misses.

6. **Evaluation:** Conduct a comparative analysis of the developed solution against existing filters (SuRF, Rosetta, Memento) regarding query latency (p50, p99), memory overhead, and build time.

2 Research Questions

My work evaluates the following questions:

- **RQ1:** Does using specialized processor instructions (like AVX512, BMI2) allow us to significantly reduce the constant factor in $O(1)$ access time?
- **RQ2:** How much does limiting to fixed key lengths help in optimizing MMPH? How does it affect the final metadata size?
- **RQ3:** Is that possible to write *branchless* algorithm for Trie traversal? What impact does a *branchless* approach to Trie traversal have on the filter‘s overall throughput?
- **RQ4:** Is it possible to implement the purely theoretical Hollow Z Fast Trie structure efficiently enough to be competitive?
- **RQ5:** What is the minimum achievable memory overhead when implementing Hollow Trie compared to the theoretical information limit?
- **RQ6:** How do specific key distribution patterns (e.g., clustering) affect the FP probability in Hollow Trie?

3 Impact of the Study

The scientific and practical importance of this research is determined by modern trends in hardware development and data processing systems.

1. The Problem of Constants in Data Structures. Many data structures that are theoretically optimal (Big-O) and space-efficient often have high computational complexity when running on real hardware. For example, comparing MMPH implementations, we can notice that some structures with worse memory asymptotics $N \log N + O(N)$[7] actually use less memory than $N \log W + O(N)$[2], which has better asymptotics, his happens because of big "hidden" constant in $O(N)$. Our research aims to reduce these "hidden" constant time factors through Hardware-Software Co-design.

2. CPU Evolution. For the last 20 years, CPU core performance has almost stopped growing due to frequency. The main growth comes from parallel instruction execution. This is mainly achieved through *out-of-order execution*. And Single Instruction Multiple Data (SIMD) CPU instructions. To do this, it is necessary to have minimal number of *branch mispredictions*, CPU-friendly *memory layout*, and minimal number of *memory random accesses*, and also use special CPU bit manipulation instructions like **BMI2** and **AVX512-VBMI**. Thus, building CPU-friendly algorithms is crucial for performance.

3. Economic Efficiency of Storage Systems. In cloud infrastructures, storage costs and Disk I/O are the main expenses. Optimal Range Filters allow:

- Reducing throughput load on HDDs and SSDs, extending their lifespan by reducing unnecessary reads.
- Speeding up real-time analytical queries.
- Reducing RAM and DISK consumption for SSTable metadata.

4 Methodology

To achieve the set goals, I will use a methodology combining theoretical analysis and low-level system optimization. The focus is not on improving asymptotic complexity, which is already optimal, but on reducing time constants in Big-O notation through processor cache optimization (Memory Layout) and SIMD instruction usage.

Tools and Technology Stack

The project will be implemented in **Go**, leveraging its facilities for direct memory manipulation and assembly integration.

- **Language:** Go, using pointers for direct memory management and bypassing Garbage Collection (GC) in critical sections.
- **Assembly:** Using `Go ASM` to access **BMI2** (`PDEP`, `PEXT`) and **AVX-512 BMI** instructions.
- **Profiling:** Using `perf` (Linux) to analyze L1/L2 cache misses and branch prediction errors.
- **Test Stand:** Server based on Intel Xeon architecture supporting AVX512-BITALG and AVX512-VBMI instruction sets.

Metrics and Success Criteria

The solution will be evaluated based on the following parameters:

- **Query Latency:** Measuring the 50th and 99th percentiles of query execution time. The goal is to achieve less than 1000ns latency per request with a filter size of 10^7 keys.
- **Space Efficiency:** Measuring memory consumption in bits per key. The target is to achieve better compression than SuRF-compatible accuracy levels.
- **Build Time:** The time required to construct the structure from a sorted iterator must be linear. This is critical for the Compaction process in LSM-trees.

5 Project Plan

The development process is structured into sequential stages. The project progress is monitored against these key milestones, with a buffer included for contingencies.

- **Stage 1: Hollow Z-Fast Trie (Completed).** Implementation of deterministic and probabilistic Z-Fast Tries, including MMPH and Encoded Handles.
- **Stage 2: Succinct Structures (In Progress).** Development of Succinct Bit Vectors and Range Locator integration.
- **Stage 3: LocalExactRange & ARE (Planned).** Assembling the final Approximate Range Emptiness structure using the "Universe Reduction" mechanism.
- **Stage 4: Evaluation (Planned).** Comparative benchmarking against RocksDB filters and AVX-512 optimization analysis.

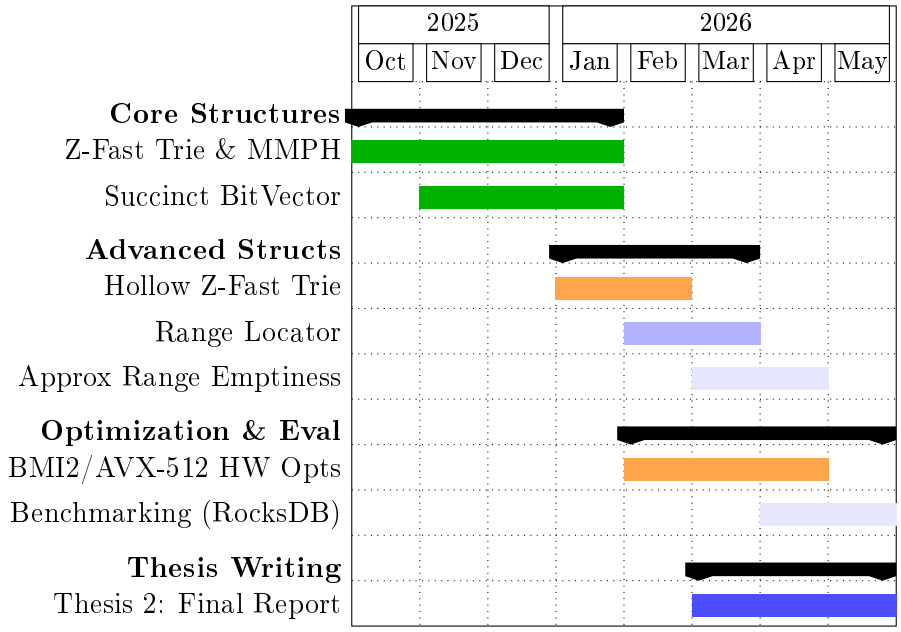

Figure 1: Project Roadmap and Progress (as of Jan 2026)

Key Milestones

- **Dec 2025:** Completion of Core Data Structures (Z-Fast Trie, MMPH).
- **Jan 2026:** Submission of Project Proposal (Thesis I).
- **Mar 2026:** Integration of Approximate Range Emptiness (ARE) structure.
- **Apr 2026:** Completion of Benchmarking and Hardware Optimization analysis.
- **May 2026:** Submission of Final Report (Thesis II) and Viva preparation.

6 Summarised Related Literature Review

This part looks at three directions: the theoretical limits of the Approximate Range Emptiness problem, the evolution of filters in LSM storages, and the monotone hashing algorithms.

6.1 Theoretical Foundations

The key work is the study by Goswami et al.[8] strictly formalized the "ARE" problem. The authors proved the information-theoretical lower memory limit: $B \geq \Omega(n \log(L/\epsilon))$, where L is the range length. (But no real implementation.) This proof refuted the possibility of using standard Bloom filters, because to check a range of length L, a Bloom filter requires L independent checks.

Belazzougui et al.[1] proposed the **Hollow Z-Fast Trie** structure. The basis of the structure is "blind" search using path hashes from root to leaf. (Hashes of paths with "2-fattest" numbers lengths) "Z-Fast" optimization applies properties of 2-adict (2-fattest numbers) for navigation, allowing skipping tree levels. The "hollowing" mechanism removes all internal nodes, replacing them with a compact transition hash table, which reduces memory to $O(n)$ words, making the structure asymptotically optimal.

6.2 Practical LSM Filters

Key-Value DB storages (RocksDB, LevelDB) use structures for block access.

- **SuRF (Succinct Range Filter) [5]:** Implements Fast Succinct Trie (FST). The key idea is hybrid tree encoding: upper levels are stored as *LOUDS-Dense* (bit map supporting rank/select), and lower ones as *LOUDS-Sparse*. This hack allows efficiently cutting off empty ranges, but it has two drawbacks: search time dependence on key length $O(k)$ and complexity of implementing iterators on compressed bit maps.
- **Rosetta [6]:** Abandons the tree structure in favor of a hierarchy of Bloom filters (Prefix Bloom Filters). A separate filter is build for each prefix length. Although this provides $O(1)$ access for the specific prefix, total memory overhead grows linearly with the number of hierarchy levels. Rosetta is effective only with the small number of unique key lengths. Which limits the number of prefixes sizes, which can be optimized.
- **REMIX [9]:** Proposes "Range Indexing" approach instead of filtering. REMIX builds sparse index over sorted SST files, allows quickly defining ranges but requires rebuilding during every data compaction, that slows down writing.

6.3 Learned Filters and Adaptive Structures

With appearance of "Learned Index Structures" (Kraska et al.) [10], filters using ML models appeared.

- **SNARF [10]:** Builds cumulative distribution function (CDF) model of keys. The filter checks if the query range falls into the predicted key region. SNARF is effective on data with high correlation (e.g., timestamps), but on random (uniformly distributed) keys its model degrades, and it switches to a structure similar to SuRF.
- **Proteus [11]:** Tries to solve the Model Selection problem. Proteus dynamically profiles the load and "assembles" filter from primitives (Bloom, Prefix BF, SuRF). However, overhead on training and rebuilding in real-time often exceeds the gain from adaptation.
- **Oasis [12]:** Segments the key space into disjoint intervals, training a simple linear model for each segment. This allows achieving better data locality but complicates the logic of processing boundary queries (range query boundary conditions).

6.4 Monotone Hashing

To transform node hash into physical rank (Range Locator), MMPH algorithms are used. The **HDC (Hash, Displace, Compress)** method [7] builds function in three stages: hashing into buckets, resolving collisions by displacement, and compressing the displacement table. The recent **LeMonHash** [13] tries to replace the displacement table with neural network, but in practice sometimes (for $N < 10^8$) the tabular HDC method remains faster due to the absence of floating-point operations. (However, replacement partly succeeded)

6.5 Research Gap

Today we see two groups: theoretically optimal structures (ARE, Hollow Trie) are considered "complex" for implementation, while practical ones (SuRF, Rosetta) have suboptimal asymptotics ($O(k)$ or $O(L)$). Thus, we need an "engineering adaptation" of the theoretical ideas of Hollow Z-Fast Trie, which will allow obtaining constant search time $O(1)$ without overhead on any ML models.

7 Subject Field

Computer Science / System Software / Database Internals

References

[1] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *ESA (1)*, volume 6346 of *LNCS*, pages 427–438. Springer, 2010.

[2] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with o(1) accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*, pages 785–794. SIAM, 2009.

[3] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[4] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*, pages 323–336. ACM, 2018.

[6] Siqiang Luo, Subhadeep Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, pages 2071–2086. ACM, 2020.

[7] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.

[8] Mayank Goswami, Allan Grønlund Jørgensen, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '15)*, pages 848–858. SIAM, 2015.

[9] Wenyu Zhong, Chen Chen, Xing Wu, and Song Jiang. Remix: Efficient range query for lsm-trees. In *19th USENIX Conference on File and Storage Technologies (FAST '21)*, pages 1–15. USENIX Association, 2021.

[10] Kapil Vaidya, Subhadeep Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. Snarf: A learning-enhanced range filter. *Proceedings of the VLDB Endowment*, 15(8):1632–1644, 2022.

[11] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*, pages 1667–1681. ACM, 2022.

[12] G Chen, Z He, M Li, and S Luo. Oasis: An optimal disjoint segmented learned range filter. *Proceedings of the VLDB Endowment*, 17(8):1911–1924, 2024.

[13] Paolo Ferragina, Hans-Peter Lehmann, Peter Sanders, and Giorgio Vinciguerra. Learned monotone minimal perfect hashing. In *31st Annual European Symposium on Algorithms (ESA 2023)*, 2023.