A Small-Scale Evaluation of Learned Indexes vs. B+Trees on SOSD Datasets

Zhengrui Wang

1 Introduction

Traditional ordered index structures such as B+Trees have been the default choice in database systems for decades. They provide logarithmic-time lookups over sorted keys and are well understood, robust, and easy to reason about. However, they are also oblivious to the actual distribution of keys and suffer from branch mispredictions and pointer chasing on modern CPUs. Recent work on learned index structures argues that an index can be viewed as a model that maps keys to positions in a sorted array, and that replacing hand-designed trees with machine-learned models can yield faster and smaller indexes in certain settings. In particular, Kraska et al. introduce the idea of learned indexes and propose the Recursive Model Index (RMI), while the SOSD benchmark and follow-up work provide a systematic evaluation of learned and traditional indexes on several large real-world datasets.

The goal of this project is to reproduce, at a smaller scale and with simpler code, some of the qualitative behavior reported in that line of work. Concretely, I implement a static B+Tree and a simple two-stage RMI in C++, and compare them on four real-world integer datasets from the SOSD suite: `books_200M_uint64`, `fb_200M_uint64`, `osm_cellids_200M_uint64`, and `wiki_ts_200M_uint64`. Each dataset consists of densely packed, sorted 64-bit keys. I focus on read-only equality lookups and ignore updates and range queries.

My experiments examine three main questions: when a simple RMI can outperform a basic B+Tree on real data, how their build times and index memory compare, and how sensitive the RMI is to the number of leaf models. The experiments use subsamples of size 1M, 5M, and 10M keys from each dataset and measure mean, p95, and p99 lookup latency as well as build time and index metadata size.

2 Design

The overall setting is intentionally restricted to match the core SOSD scenario: an in-memory, read-only index over a static array of sorted 64-bit keys with dense IDs and no duplicates. The underlying data structure is a `std::vector<uint64_t>` that stores all keys. Both the B+Tree and the RMI are built on top of this vector and only store additional metadata; neither structure owns a separate copy of the keys. The workload consists of equality lookups on keys that are guaranteed to exist in the array, so I do not consider insertions, deletions, or range scans.

On top of this base, I implement and evaluate two index designs. The first is a static B+Tree with fixed node fanout (order) equal to 64. Leaf nodes store up to 64 keys and their positions in the base array, while internal nodes store separator keys and pointers to child nodes. The tree is bulk-loaded bottom-up from the sorted key array and never modified afterward. Lookup in this tree follows the standard pattern: binary search among the separators at each internal node to choose the child, descend to a leaf, and binary search over the leaf's key array to find the target.

The second design is a two-stage Recursive Model Index. This RMI treats the index as a regression problem: it tries to learn a function mapping a key to its position in the sorted array. The structure consists of a single root model and a set of leaf models. Both root and leaf models are simple linear regressions of the form

$$\text{pos} \approx a \cdot \text{key} + b.$$

The root model provides a coarse prediction of the position and is used to select a leaf; the leaf model then refines the prediction. To make the structure usable as an index, each leaf also maintains an error bound that represents the maximum absolute prediction error on training keys. At lookup time, the RMI uses the leaf model to predict a position and then performs a binary search over the key array within a small window around that prediction, sized by the error bound.

To understand the impact of model capacity, I vary the number of leaf models in the RMI. For the `fb` and `wiki` datasets, I fix the number of leaves L at 64, a common choice in prior work. For `books` and `osm`, I run a small hyper-parameter sweep with $L \in \{32, 64, 128, 256\}$ to see how increasing the number of leaf models affects both accuracy and performance. In all cases, the RMI is read-only and shares the base key array with the B+Tree.

3 Implementation

Both index structures are implemented in C++17. The B+Tree is represented by a node structure that stores whether the node is a leaf, its fanout, a `min_key` for the entire subtree rooted at this node, a vector of separator keys, and either a vector of positions (for leaves) or a vector of child pointers (for internal nodes). Leaf nodes also have an optional `next` pointer to their right sibling, which could be used for efficient range scans, although range queries are not evaluated in this project. Memory usage for the B+Tree is estimated by recursively counting the number of nodes and multiplying by a fixed per-node cost (512 bytes), which roughly accounts for the node header, keys, and pointers.

The B+Tree is bulk-loaded from the sorted key array in a bottom-up manner. I first scan the array in blocks of at most 64 keys, creating a leaf node for each block. Each leaf copies its keys, records their indices in the base array, and sets `min_key` to its first key. The leaves are linked together as a list. I then construct internal levels by grouping nodes from the current level into blocks of up to 64 children, allocating a parent for each block, and filling its `child_ptrs` with the children. The parent's `min_key` is set to the `min_key` of the first child, and its separator keys are set to the `min_key` of each child starting from the second one. This process repeats until only a single node remains, which becomes the root. A correctness bug in my initial implementation, where internal separators were not aligned with subtree minima, caused lookups to occasionally miss existing keys; this was later fixed by explicitly tracking and using each child's `min_key` when constructing parents.

The RMI is implemented as a class that holds a root linear model and a vector of leaf models. Training starts by treating the sorted key array as pairs $(\text{key}_i, \text{pos}_i)$ where $\text{pos}_i = i$. The root model is trained by fitting a least-squares linear regression over all keys and positions, using analytic formulas for the regression coefficients. After training the root, each key is assigned to a leaf based on the root's predicted position: the continuous prediction $\hat{y}$ is clamped to the range $[0, N-1]$ and mapped to an integer leaf index

$$\ell = \left\lfloor \hat{y} \cdot \frac{L}{N} \right\rfloor,$$

where L is the number of leaves. For each leaf, I then fit a separate linear regression over its assigned keys and positions. Finally, for every key assigned to a given leaf, I compute the absolute prediction error under that leaf's model and store the maximum as the leaf's error bound.

Lookup with the RMI uses both model stages. Given a query key, I first evaluate the root model to obtain a coarse position prediction and select a leaf index as in training. The corresponding leaf model is then evaluated to get a refined position prediction, which is again clamped to $[0, N-1]$. I then define a search interval centered at this predicted position with radius equal to the leaf's error bound, adjusted to remain within the array bounds. A standard binary search is then executed over the base key array restricted to this interval. In the common case where the models are accurate and the error bound is small, this interval contains only dozens of keys and the search is fast. In worst cases with large error bounds the RMI effectively degrades to a binary search over a substantial portion of the array.

To ensure that both implementations are correct, I added a sanity check routine that runs after building the indexes. It samples 100 random existing keys and verifies that both the B+Tree and the RMI report success and return positions where the key matches. It also probes 100 "non-keys" formed by adding one to randomly chosen keys and ensures that no crashes occur. The separator bug in the initial B+Tree implementation was caught by this sanity check, since the B+Tree would sometimes fail to find a key that the RMI could find correctly.

4 Experimental Setup

The experiments use four 64-bit integer datasets from the SOSD benchmark: `books_200M_uint64`, `fb_200M_uint64`, `osm_cellids_200M_uint64`, and `wiki_ts_200M_uint64`. These datasets are stored as binary files of sorted, unique unsigned 64-bit integers. For each dataset, I load only a prefix of the file for each experiment. Specifically, I consider three sizes: the first one million keys (1M), the first five million keys (5M), and the first ten million keys (10M). This keeps build times and memory usage manageable on a personal laptop while still providing realistic scales and allowing me to explore how performance changes with the number of keys.

For each combination of dataset and size, I build both the B+Tree and the RMI on the same base key array. For the RMI, the number of leaf models is fixed to 64 for the `fb` and `wiki` datasets. For `books` and `osm`, I sweep over 32, 64, 128, and 256 leaves to study the effect of model capacity on performance. After building both indexes and running the sanity check, I generate an equality lookup workload of 100,000 queries by sampling positions uniformly at random from the $[0, N-1]$ range and using the corresponding keys as queries. This ensures that all queries are hits and that the workload is evenly distributed over the key space.

I measure lookup latency using `std::chrono::high_resolution_clock` around each individual query and record the durations in nanoseconds. From these per-query measurements, I compute the mean latency as well as the empirical 95th and 99th percentiles (p95 and p99). I also record build time for each index as the wall-clock time to construct the structure from the key array, and index memory as the estimated metadata size using the counters described earlier. All experiments are run in a single-threaded C++17 binary compiled with `-O3`, on an ARM-based macOS laptop, with no other heavy applications running during measurement.

The experimental pipeline writes all metrics into CSV files. For each run, one file captures lookup metrics with columns for dataset, index type (B+Tree or RMI), number of keys, number of leaves (for RMIs), and mean/p95/p99 latency. Another file captures build metrics with dataset, index type, number of keys, number of leaves, build or training time in seconds, and index metadata size in bytes. These CSV files make it easy to inspect the data, compute speedup ratios, and plot trends in external tools.

5 Results

The first set of results compares lookup latency for the B+Tree and the RMI as the number of keys grows from 1M to 10M. On the `books` dataset, which appears to have a very smooth key distribution, the RMI consistently outperforms the B+Tree by a large margin. Across all three sizes, the best RMI configuration (typically with 64–256 leaves) achieves roughly three to four times lower mean latency than the B+Tree. For example, at 10M keys the B+Tree's mean per-query latency is in the few-hundred-nanosecond range, while the RMI's mean is closer to a hundred nanoseconds. The RMI also offers significantly lower p95 and p99 tail latencies on `books`, indicating that its predictions rarely lead to large search windows.

On the `fb` and `wiki` datasets, the RMI still improves lookup latency, but the gains are more modest and shrink with scale. At 1M keys the RMI is roughly 1.5–2 times faster than the B+Tree on both datasets. At 5M keys, the speedup drops to about 1.3–1.4 times. By 10M keys, on `fb` the RMI still maintains a small advantage, while on `wiki` the mean latencies of the two structures are essentially the same, although the RMI retains a slightly shorter tail (lower p95 and p99). This suggests that the key distributions in `fb` and `wiki` are somewhat learnable but more complex than `books`, so a simple two-stage linear RMI cannot consistently maintain a large advantage as the dataset grows.

The `osm` dataset behaves quite differently. At 1M keys, a reasonably tuned RMI (for example with 128 leaves) can still match or very slightly outperform the B+Tree, but the speedup is small. As the number of keys increases to 5M and 10M, even the best RMI configurations become slower than the B+Tree. At 10M keys, the RMI's mean latency is clearly higher, and its p95 and p99 latencies are also worse, indicating that prediction errors cause the search windows to expand and negate the benefits of the model. This is consistent with the expectation that some real-world key distributions are harder to learn and that simple linear models cannot approximate their CDF well enough.

The second group of results examines how the number of leaf models affects RMI performance. On the `books` dataset, increasing the number of leaves from 32 to 64 or 128 reduces mean latency, as each leaf can focus on a narrower slice of the key space and fit it more accurately. Beyond 128 leaves, the benefits plateau; 256 leaves perform similarly to 128, with small fluctuations depending

on the dataset size. Importantly, all configurations with 64 or more leaves substantially outperform the B+Tree on `books`. On `osm`, increasing the number of leaves also reduces RMI latency at a given scale, but even the best configuration at 5M and 10M keys remains slower than the B+Tree. This shows that while model capacity matters, there are datasets where adding more leaf models cannot overcome the inherent difficulty of the distribution.

Build time and index memory follow intuitive patterns. B+Tree build time grows roughly linearly with the number of keys and stays in the tens of milliseconds at 10M keys. The RMI's training time is also linear in the number of keys but tends to be around twice as large as the B+Tree's build time at the largest scale, due to the cost of fitting regression models. However, in absolute terms, both build and training times are small compared to the cost of loading the data. For memory, the B+Tree's estimated metadata size grows to tens of megabytes as the dataset reaches 10M keys, driven by the number of internal and leaf nodes. In contrast, the RMI's model memory is in the kilobyte range: with 64 leaves it uses only a few kilobytes, and even with 256 leaves it remains around ten kilobytes. Since both structures share the same base key array, this confirms that the learned index can be orders of magnitude more compact in terms of additional index metadata.

6 Discussion

The experiments suggest a nuanced answer to the question of when a simple RMI is preferable to a B+Tree. On smooth, learnable distributions such as `books`, a two-stage linear RMI works extremely well: it can provide three to four times lower lookup latency while using dramatically less index memory. On moderately complex datasets like `fb` and `wiki`, the RMI still offers improvements, especially at smaller scales, but the advantages diminish as the dataset grows. On harder distributions such as `osm`, the RMI's modeling error leads to larger search windows, and the learned index becomes slower than even a fairly simple, unoptimized B+Tree.

These findings fit well with the picture painted by prior work on learned indexes. They confirm that the potential benefits are real in some regimes, but also that learned indexes are not universally superior. The performance of an RMI depends strongly on the data distribution, on how many models are used, and on the details of the search strategy around the predictions. My experiments also highlight that hyper-parameters like the number of leaf models matter, but they are not magic: increasing the number of leaves on `books` provides clear improvements and reaches a comfortable plateau, while on `osm` it only reduces the gap to the B+Tree without overturning it.

The project also surfaced some engineering challenges and aspects that did not work as smoothly as expected. The B+Tree implementation initially had subtle bugs in how internal separators were constructed. Without sanity checks that compared the B+Tree's answers to those of the RMI on random keys, these bugs could have gone unnoticed, since many queries would still succeed. Fixing them required carefully tracking and propagating subtree minima through the tree. On the RMI side, my implementation uses only linear models and simple least-squares training and does not include any of the more advanced ideas from the literature, such as deeper model hierarchies, piecewise linear approximations, or adaptive error bounds. As a result, it is not surprising that the RMI struggles on more complex distributions; the project nevertheless provides useful intuition for why that happens.

There are also important limitations in the experimental setup. The entire study is conducted

in a single-threaded, in-memory environment, with equality lookups only and no updates. I did not evaluate range queries, mixed workloads, or the impact of concurrency and cache behavior in multi-core systems. The B+Tree implementation is straightforward and not optimized with cache-aware layouts, SIMD, or manual prefetching, and the RMI is similarly unoptimized. This means that absolute latency numbers cannot be directly compared to those in the SOSD benchmark or in the original papers; only qualitative trends are meaningful. Despite these limitations, the main qualitative conclusions line up with those in the literature, which increases confidence that the implementation and methodology are reasonable.

7 Conclusion

This project implemented and evaluated a static B+Tree and a simple two-stage RMI in C++ on four real-world SOSD datasets. Both indexes operate over the same sorted arrays of 64-bit keys and are evaluated in a single-threaded, read-only setting with equality lookups. The experiments measure lookup latency, build time, and index metadata size for subsamples of 1M, 5M, and 10M keys and, for two datasets, explore the effect of the number of leaf models in the RMI.

The main conclusions are that a simple RMI can substantially outperform a basic B+Tree on some real-world datasets, particularly when the key distribution is smooth and learnable, but that its advantage shrinks or vanishes on more complex or irregular distributions. On the `books` dataset, the RMI achieves roughly three to four times lower mean lookup latency than the B+Tree while using orders of magnitude less index memory. On `fb` and `wiki`, it still improves latency, especially at smaller sizes, but the gains become small at the largest scale. On `osm`, the RMI fails to beat the B+Tree at 5M and 10M keys even with more leaf models, underscoring that learned indexes are not a drop-in replacement for traditional structures. Overall, the project confirms both the promise and the limitations of learned index structures and provides a concrete, small-scale implementation that reproduces many of the qualitative behaviors reported in prior work.