Minimal Perfect Hashing - Integration to RocksDB and Discussion on Performance

Camilla Xue

***Abstract*—**This report evaluates the integration of a Minimal Perfect Hashing index into RocksDB, aiming to eliminate hash collisions and improve lookup efficiency within data blocks.

I. Introduction

Designed to optimize write-intensive workloads, the Log-Structured Merge-tree (LSM-tree) organizes data in multiple levels of sorted runs to allow efficient sequential writes and amortized cost for merges [1]. Due to its efficient write performance and adaptability, the LSM-tree has become a cornerstone in modern key-value database systems, underpinning widely-used platforms such as Cassandra, DynamoDB, and RocksDB.

This report focuses on RocksDB, a high-performance embedded key-value store developed by Facebook that extensively employs the LSM-tree architecture. In RocksDB, data is stored within immutable Sorted String Table (SST) files, where data blocks are the fundamental storage units in the block-based table format and contain a sequence of key-value entries. Within each data block, keys are encoded using prefix compression and organized with restart intervals. Traditionally, RocksDB uses binary search over the restart points inside a data block to locate the appropriate region corresponding to a target key, followed by a linear scan within that region. To further accelerate point lookups and reduce the number of memory accesses (hops) caused by binary search, RocksDB has experimented with integrating a hash table as an auxiliary index. This hash table aims to shortcut the search process and lower search latency by providing direct access to key positions.

Building upon these ideas, our work proposes to enhance the data block search mechanism by employing Minimal Perfect Hashing (MPH). Unlike traditional hash tables, MPH guarantees collision-free key mapping, effectively eliminating collision overhead and reducing the number of memory accesses (hops) required during searches. This approach aims to improve read performance while maintaining RocksDB's storage efficiency.

The remainder of this paper is organized as follows. Section II examines the current hash index implementation in RocksDB, focusing on hash bucket utilization. Section III presents the design and performance evaluation of the Minimal Perfect Hashing (MPH) index, comparing it against both the no-hash baseline and the existing hash index. Section IV provides a detailed analysis of MPH in relation to hash and no-hash approaches, exploring the factors influencing their performance differences.

II. Hash Index in RocksDB

The hash index is constructed during each data-block building, and its control flow is summarized in Figure 1, which highlights the relevant classes, key functions, and internal variables. The *index_type* option determines whether a hash index should be created for the block. When enabled, a *DataBlockHashIndexBuilder* instance is initialized alongside the block builder. As each key–value pair is appended to the block, the builder invokes its *Add* function, which in turn calls the corresponding Add method of the hash-index builder. For every entry, this process records the hash of the key together with the associated restart index, storing these (hash, restart) pairs in *hash_and_restart_pairs_*. After all entries are inserted and the block is finalized, the *Finish* method is invoked to construct the actual hash table.

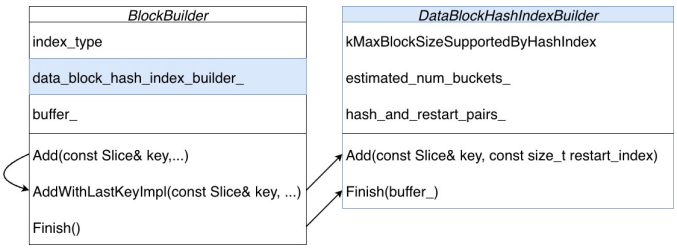

Fig. 1. Design of RocksDB's Hash Index.

The hash table size is determined by the number of key–value pairs divided by the utilization ratio, which defaults to 0.75. Each key is then placed into its hashed bucket; if a bucket is already occupied, a collision is detected, and the bucket is marked accordingly. Entries that fall into collided buckets do not participate in hash-index lookup at query time, instead, any lookup for them falls back to the standard binary search mechanism over the block's restart intervals. This design preserves correctness while attempting to benefit from the constant-time read for those valid buckets.

To better understand the behavior and effectiveness of the hash index, we analyze the distribution of its buckets which fall into either valid, collided, or unused entry. We conduct this analysis across a range of value sizes: 8B, 16B, 64B, and 100B. As the value size increases, fewer key-value pairs fit in each block, resulting in fewer entries in the hash index. Figure 2 shows the proportion of each bucket type remains relatively consistent across different value sizes, with a small reduction in collisions observed when the value size reaches 100B. Notably, approximately 50% of the hash buckets remain unused, representing a significant amount of memory allocated but not contributing to query acceleration. This inefficiency

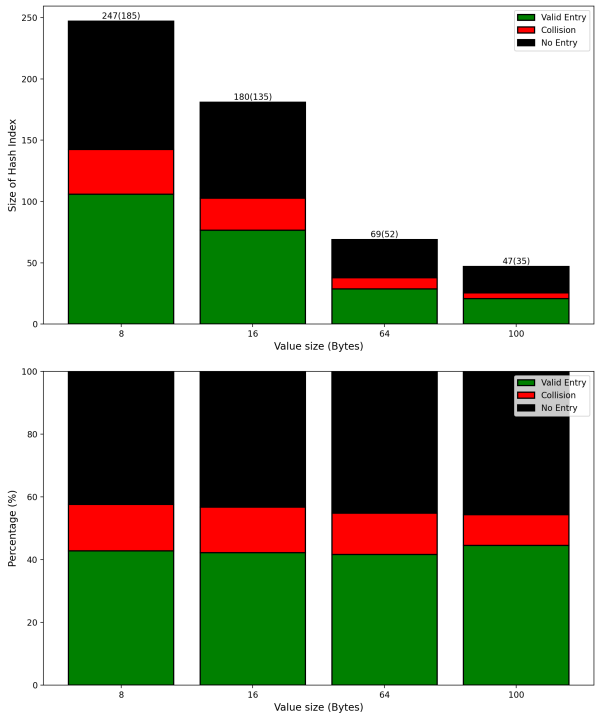

Fig. 2. Distribution of Hash Index in size(top) and in percentage(bottom). Labels above each bar indicate the block size, with the number of entries displayed in brackets.

motivates our adoption of minimal perfect hashing (MPH), which aims to eliminate collisions entirely and ensure that all allocated space directly supports faster and more effective key lookups.

III. MPH Performance Discussion

The MPH index implementation follows a structure similar to RocksDB's existing hash index. To reduce the likelihood of hash collisions at each level, we adopt a 64-bit hashing scheme in place of the 32-bit hash used by the original index. The MPH construction uses a jump-seed mechanism to deterministically assign each key to a unique bucket, ensuring a collision-free mapping. After the minimal perfect hash is generated, the MPH index is serialized into the data block in a compact format, where each entry is encoded in 1 byte (8 bits), except for the bit-vector entries encoded in 8 bytes (64 bits). The serialized components: levels, bit vectors, rank, and values, are written sequentially, with each preceded by a 1-byte field indicating the number of entries in that component. This layout minimizes storage overhead while ensuring that every stored byte directly contributes to collision-free key lookup, enabling accurate and predictable search performance.

We compare the point query performance of the MPH index with that of traditional binary-search-based and hash-based indexing schemes. All experiments are conducted with the cache size provisioned larger than the dataset size in order to isolate computational efficiency from I/O effects. By varying both the value size and the restart interval, we observe performance characteristics that differ across the three indexing methods.

As illustrated in Figure 3, the MPH index consistently outperforms both binary and hash indexing for small restart intervals, regardless of value size. However, as the restart interval increases, the throughput of MPH declines and query latency increases. Figure 4 further quantifies these results by presenting the relative throughput gain or loss of binary and hash indexing using MPH throughput as the baseline.

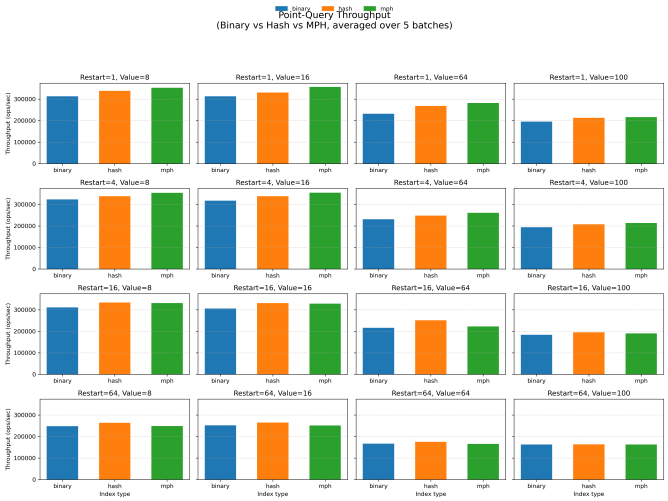

Fig. 3. Performance comparison among no hash (binary search), hash, and MPH with the size of value varying: 8, 16, 64, 100 bytes, and the block restart interval varying: 1, 4, 16, 64.

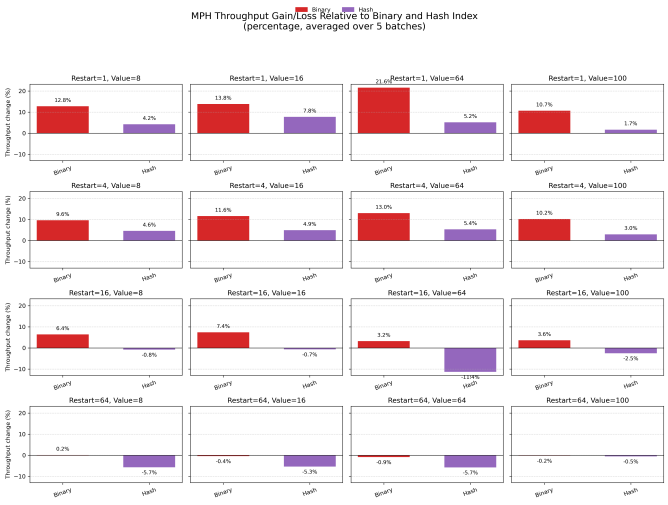

Fig. 4. Performance improvement using MPH as baseline for size of value varying: 8, 16, 64, 100 bytes, and the block restart interval varying: 1, 4, 16, 64.

The results show that MPH indexing significantly outperforms binary indexing for restart intervals of 1, 4, and 16 across all tested value sizes. Nevertheless, the magnitude of improvement diminishes as the restart interval increases. When the restart interval reaches 64, the advantage of MPH becomes marginal, and in some cases MPH performs worse than binary indexing. A similar trend is observed when comparing MPH against hash indexing: the relative performance of MPH deteriorates noticeably when the restart interval is 16 or larger.

This degradation suggests that the overhead associated with constructing and traversing the MPH index to directly identify the target restart point can exceed the cost of performing a conventional binary search or hash lookup over restart intervals. Consequently, for larger restart intervals and larger value sizes, the additional computation introduced by MPH increases query latency and reduces overall throughput.

IV. Results and Discussion

The minimal perfect hash (MPH) index is designed to reduce the number of lookup hops by directly locating the target restart index, thereby eliminating the need for binary search. The experimental results generally align with this design objective across different restart interval configurations. For small restart intervals, the MPH index consistently outperforms binary search by approximately 10% and hash indexing by approximately 5%. This behavior is consistent with the role of the restart interval: smaller restart intervals require more search steps across restart points, making MPH particularly effective in reducing CPU overhead through direct perfect-hash-based addressing.

As the restart interval increases, however, the number of required search hops decreases. Under these conditions, the overhead associated with parsing and traversing the MPH structure for each query begins to outweigh its benefits, resulting in degraded performance. Furthermore, the performance variations observed under different value sizes highlight additional trade-offs associated with adopting MPH indexing. In the following section, we discuss the potential factors contributing to these behaviors and analyze the associated performance implications.

A. Size of MPH vs Hash

We begin by analyzing the size breakdown of the MPH index compared to the Hash index under the default value of restart interval(16). Figure 5 illustrates the raw size and the proportion each component contributes to the total MPH size, alongside the size of the Hash index and the overall size increase of MPH relative to the Hash index. The value component dominates the MPH size, accounting for over 60%, which its size corresponds to the number of keys stored in the index. The Bit Vector follows as the second largest contributor at around 20%, representing the serialized bits of the hashed keys across levels. The level and rank components make up smaller proportions of the total size. Notably, as the value size increases, the MPH index grows disproportionately larger compared to the Hash index; for example, at a value size of 100 bytes, the MPH index is approximately 50% larger than the Hash index.

Given this noticeable size gap, a natural first question is whether the larger MPH index might incur additional latency due to cache behavior, such as, by causing more cache-line loads or incur more cache-line misses. However, this explanation appears insufficient upon closer inspection. The serialized MPH bytes are laid out sequentially and are read contiguously when constructing the in-memory MPH structure. Thus, the cost of fetching the MPH data is unlikely to account for the relatively large performance difference observed.

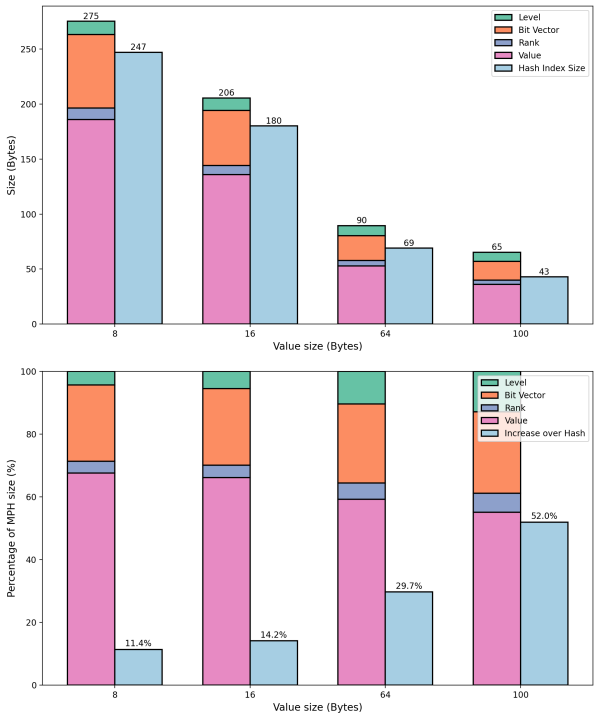

Fig. 5. Distribution of MPH index size (top) and percentage breakdown (bottom) compared to the Hash index. Both indexes store the same number of keys across all value sizes. The labels above each bar in the bottom graph indicate the percentage increase in size of the MPH index relative to the Hash index.

This shifts our attention toward instruction-level complexity, which relates to structure of the lookup logic itself. The Hash index performs a straightforward sequence: hash the key, map to a bucket, and read the restart index (falling back to a binary search if there is a collision). On the other hand, the MPH lookup executes a more involved chain of operations. It determines the correct level, interprets the corresponding bit-vector segment, calculates the index to the value, and reads the restart index. This sequence introduces more dependent instructions and conditional branches, all of which increase the data dependency and likelihood of branch mispredictions that reduce the CPU's ability to exploit instruction-level parallelism.

To better understand how these differences impact actual performance, we measure the CPU time consumed during the lookup process. The analysis is divided into two parts: index initialization time, which reflects the overhead of setting up the data structures in memory, and restart index lookup time, which captures the cost of retrieving the restart point for a given key. As shown in Figure 6, the MPH index consistently spends noticeably more time during lookup compared to the Hash index. For index initialization, the Hash index maintains a relatively constant time across different value sizes, while MPH initialization time is roughly five times higher. Regarding the restart index lookup time, MPH exhibits greater latency overall, with a particularly marked increase for larger value sizes (64 and 100 bytes) compared to smaller ones (8 and

16 bytes). This trend aligns with the declining performance observed in Figure 3, where MPH performs worse as the value size increases in the setting of default restart interval.

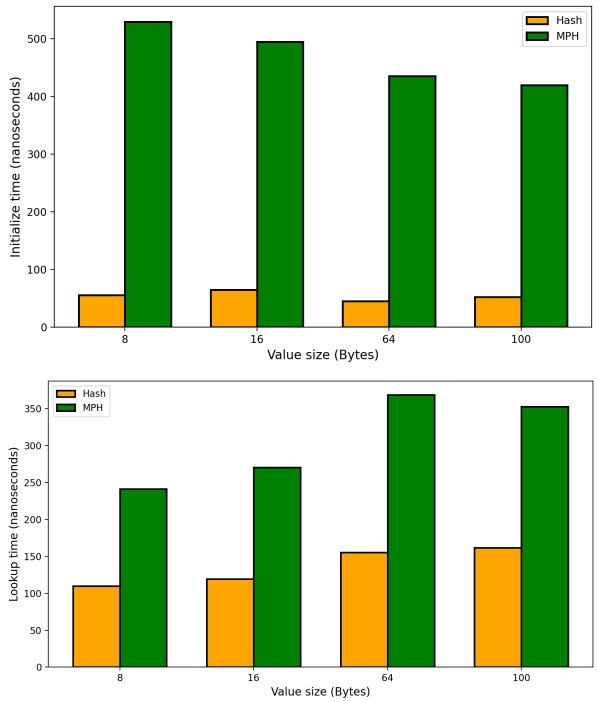

Fig. 6. Comparison of lookup time between MPH and Hash indexes, showing index initialization time (top) and restart index lookup time (bottom). For the Hash index, the lookup time is measured only in no-collision scenarios to isolate the cost of direct hash-based access.

V. Conclusion

This report evaluated the performance implications of integrating a MPH index into RocksDB's data block lookup mechanism. Although MPH offers theoretical advantages by eliminating collisions and minimizing lookup hops, our experiments reveal that these benefits diminish as the size of stored values grows. The overhead of additional computations and data dependencies in MPH outweighs its gains, resulting in slower performance compared to both the conventional hash index and binary search for larger values.

REFERENCES

[1] O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. "The Log-Structured Merge-Tree (LSM-Tree)." Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996.

[2] Camilla Xue. "RocksDB MPH Index Implementation." https://github.com/Caconoutt/rocksdb-mph/pull/3