Towards Space-Efficient High-Performance In-Memory Search Structures

Huanchen Zhang

April 17, 2018

Thesis Proposal

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Michael Kaminsky
Andrew Pavlo
Kimberly Keeton

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Abstract

This thesis seeks to address the challenge of building space-efficient yet high-performance in-memory search structures, including indexes and filters, to allow more efficient use of memory in OLTP databases. We show that we can achieve this goal by first designing fast static structures that leverage succinct data structures to approach the information-theoretic optimum in space, and then using the "hybrid index" architecture to obtain dynamicity with bounded and modest cost in space and performance.

To obtain space-efficient yet high-performance static data structures, we first introduce the Dynamic-to-Static rules that present a systematic way to convert existing dynamic structures to smaller immutable versions. We then present the Fast Succinct Trie (FST) and its application, the Succinct Range Filter (SuRF), to show how to leverage theories on succinct data structures to build static search structures that consume space close to the information-theoretic minimum while performing comparably to uncompressed indexes. To support dynamic operations such as inserts, deletes, and updates, we introduce the dual-stage hybrid index architecture that preserves the space efficiency brought by a compressed static index, while amortizing its performance overhead on dynamic operations by applying modifications in batches.

In the proposed work, we seek opportunities to further shrink the size of in-memory indexes by co-designing the indexes with the in-memory tuple storage. We also propose to complete the hybrid index work by extending the techniques to support concurrent indexes.

Contents

Chapter 1

Introduction

The overhead of managing disk-resident data has given rise to a new class of online transaction processing (OLTP) database management systems (DBMSs) that store the entire database in main memory [14, 15, 18, 29]. These systems outperform traditional disk-oriented DBMSs because they eschew the legacy components that manage data stored on slow, block-based storage [20]. Unfortunately, this improved performance is achievable only when the database is smaller than the amount of physical memory available in the system. If the database does not fit in memory, then the operating system will move virtual memory pages out to disk, and memory accesses will cause page faults [28]. Because these page faults are transparent to the DBMS, the threads executing transactions will stall while the page is fetched from disk, degrading the system's throughput and responsiveness. Thus, the DBMS must use memory efficiently to avoid this performance bottleneck.

Although the price of DRAM dropped drastically between 2005 and 2011 and stablized since [7], it is neither free nor infinite. Memory is a non-trivial capital cost when purchasing new equipment, and it incurs real operational costs in terms of power consumption. Studies have shown that DRAM can account for up to 40% of the overall power consumed by a server [23]. Improving memory efficiency in a DBMS, therefore, has two benefits. First, for a fixed working set size, reducing the required memory can save money, both in capital and operating expenditures. Second, improving memory efficiency allows the DBMS to keep more data resident in memory. Higher memory efficiency allows for a larger working set, which enables the system to achieve higher performance with the same hardware.

Search structures in a DBMS, including indexes and filters, consume a large portion of the database's total memory. The situation is particularly bad in OLTP databases because OLTP applications often maintain several indexes per table to ensure that queries execute quickly. To illustrate this point, Table 1.1 shows the relative amount of storage used for indexes in several OLTP benchmarks deployed in a main-memory DBMS [2]. We used the DBMS's internal statistics API to collect these measurements after running the workloads on a single node until the database size approximates 10 GB. Indexes consume up to 58% of the total database size for these benchmarks, which is commensurate with our experiences with real-world OLTP systems.

Traditional index compression techniques leverage general-purpose block compression algo-

	Tuples	Primary Indexes	Secondary Indexes
TPC-C	42.5%	33.5%	24.0%
Articles	64.8%	22.6%	12.6%
Voter	45.1%	54.9%	0%

Table 1.1: Percentage of the memory usage for tuples, primary indexes, and secondary indexes in H-Store using the default indexes (DB size $\approx$ 10 GB).

rithms such as LZ77 [1], Snappy [4] and LZ4 [3]. For example, InnoDB uses the zlib library [5], which implements LZ77, to compress the B+tree nodes before they are written to disk. This approach reduces the I/O cost of fetching pages from disk, but the nodes must be decompressed once they reach memory so that the system can interpret their contents. Applying traditional block compression algorithms on in-memory indexes, however, are inappropriate especially for the small-footprint queries of OLTP applications. As we will show later in Chapter 3 (Section 3.1.5), block-compressed indexes perform poorly due to the overhead of decompressing an entire block to access a small number of tuples.

This thesis seeks to address the challenge of building space-efficient yet high-performance in-memory search structures to allow more efficient use of memory in data processing systems. The primary target of the thesis are order-preserving indexes in main-memory OLTP databases. Many of the solutions that we provide in this thesis, however, can be generalized to cover other types of in-memory search structures such as range-filters.

This thesis provides evidence to support the following statement:

Thesis Statement: *We can compress in-memory search structures while retaining their high performance in OLTP databases by first designing fast static structures that leverage succinct data structures to approach the information-theoretic optimum in space and then using the hybrid index architecture to obtain dynamicity with bounded and modest cost in space and performance.*

We have completed the following contributions for this thesis:

C1. A new set of guidelines called the Dynamic-to-Static rules (D-to-S) that identifies the major sources of wasted space in dynamic index structures and helps convert existing dynamic structures to smaller immutable versions. (Chapter 3, Section 3.1)

C2. A new data structure called the Fast Succinct Trie (FST) whose space consumption is close to the minimum number of bits required by information theory. Its performance, however, is comparable to uncompressed order-preserving indexes. (Chapter 3, Section 3.2)

C3. A new (and probably the first practical) general-purpose range-filter called the Succinct Range Filter (SuRF) that supports membership test for both single-key and range queries efficiently. (Chapter 3, Section 3.3)

C4. A new way to build indexes for main-memory OLTP databases called the hybrid index approach, which includes a dual-stage index architecture that substantially reduces the per-tuple index space with only modest costs in throughput and latency even when the reclaimed memory is not exploited to improve system performance. (Chapter 4)

We propose to explore the following ideas to complete this thesis:

P1. Value Deduplication: To reduce the space taken by the value pointers in an index. The idea is to store all the 64-bit tuple addresses in a table once together in a compressed address trie. Each tuple is implicitly assigned a ($log_2 N$-bit) tuple ID according to its rank in the address trie. All the indexes of that table then use the tuple ID to refer to a tuple. (Chapter 5, Section 5.1)

P2. Permutation Indexes: To further shrink index size by not repeatedly storing the actual key strings because they can be extracted from the tuple storage that is already in memory. Instead, an index only stores a permutation of the tuples according to the indexed-key order, and uses binary search to look for tuples. (Chapter 5, Section 5.2)

P3. Concurrent Hybrid Index: To extend the hybrid index architecture to support building space-efficient concurrent indexes. The challenge is to design a non-blocking merge algorithm to smoothly migrate index entries between the dynamic and the static stages in hybrid indexes. (Chapter 5, Section 5.3)

As C1 – C4 significantly reduce the structural overhead of an in-memory index, the actual content (i.e., the keys and the values) stored in the index starts to dominate its space consumption. P1 and P2 attempt to solve this problem and try to answer the question: what is the minimal information an index needs to stay high-performance in an in-memory row-store? P3 is a necessary extension/completion of the hybrid index approach because it allows the hybrid index technique to be used in a concurrent environment which is prevalent in today's multi-core era.

Chapter 2

Index Microbenchmarks

This chapter introduces the index microbenchmarks that we developed and used throughout the thesis. This set of microbenchmarks are based on the Yahoo! Cloud Serving Benchmark (YCSB) [12] which is a key-value workload generation tool that models large-scale cloud services. We have leveraged three of its default workloads: workload **A** (*read/write*, 50/50), workload **C** (*read-only*), and workload **E** (*scan/insert*, 95/5). We call the initialization phase (i.e., populating the index) in each workload the *insert-only* workload. The *scan-only* workload is derived from workload E by removing the 5% insert operations. We replace the default YCSB-key with three representative key types: 64-bit random integers (rand-int), 64-bit monotonically increasing integers (mono-inc-int), and email addresses (host reversed, e.g., “com.domain@foo”) drawn from a real-world dataset (average length = 22 bytes, max length = 129 bytes). The key distribution in each workload is either Zipfian or uniform. All values are 64-bit integers to represent tuple pointers.

This set of index microbenchmarks currently only supports single-threaded evaluation without any network activity. For each workload, it first inserts 50 million (default, configurable) entries into the index under test. Then it executes 10 million (default, configurable) key-value queries according to the workload. The throughput numbers reported by the benchmark are the number of operations divided by the execution time. The memory consumption numbers are measured at the end of the initialization phase.

To summarize:

Workloads:	insert-only, read-only, read/write, scan/insert, scan-only
Key Types:	64-bit rand-int, mono-inc-int, email
Value:	64-bit integer (tuple pointers)
Key Distributions:	Zipfian, uniform

The server used to execute the index microbenchmarks has the following configuration:

CPU:	2×Intel® Xeon® E5-2680 v2 CPUs @ 2.80 GHz
DRAM:	4×32 GB DDR3 RAM
Cache:	256 KB L2-cache, 26 MB L3-cache
Disk:	500 GB, 7200 RPM, SATA

Chapter 3

Compressing Static Structures

In this chapter, we consider the problem of building fast and space-efficient static search structures. The word “static” here means that the data structure is optimized for read-only workloads. An insert or update operation on a static structure will cause a significant part of the data structure to be rebuilt. We study static structures because they are more compressible than their dynamic counterparts. Later in Chapter 4, we will use the results in this chapter as building blocks to construct space-efficient high-performance dynamic indexes. Section 3.1 describes a set of general rules to help generate smaller immutable versions of dynamic data structures. Section 3.2 provides a case study of compressing static tries, which leverages Succinct Data Structures [6] to approach information-theoretic optimum in space while retaining high-performance. Section 3.3 uses the compressed trie created in Section 3.2 to solve the range-filtering problem.

3.1 Dynamic-to-Static Rules

Most of the existing index structures such as Btrees and radix trees used in main-memory DBMSs are dynamic. They support fast reads and updates but are far from being space-efficient. We observe two major sources of wasted space in dynamic data structures. First, dynamic data structures allocate memory at a coarse granularity to minimize the allocation/reallocation overhead. They usually allocate an entire node or memory block and leave a significant portion of that space empty for future entries. Second, dynamic data structures contain a large number of pointers to support fast modification of the structures. These pointers not only take up space but also slow down certain operations due to pointer-chasing.

Given an existing dynamic data structure, how can we obtain a space-efficient static structure out of it? We present the Dynamic-to-Static (D-to-S) rules to guide this process:

- **Rule #1: Compaction** – Remove duplicated entries and make every allocated memory block 100% full.
- **Rule #2: Structural Reduction** – Remove pointers and structures that are unnecessary for efficient read-only operations.
- **Rule #3: Compression** – Compress parts of the data structure using a general-purpose compression algorithm.

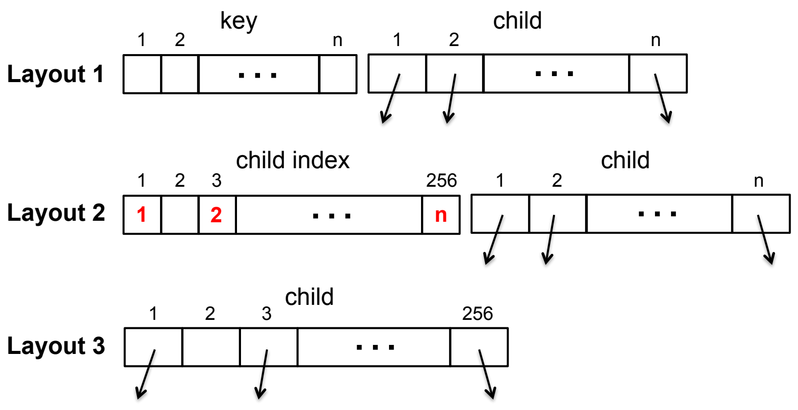


Figure 3.1: ART Node Layouts – Organization of the ART index nodes. In Layout 1, the key and child arrays have the same length and the child pointers are stored at the corresponding key positions. In Layout 2, the current key byte is used to index into the child array, which contains offsets/indexes to the child array. The child array stores the pointers. Layout 3 has a single 256-element array of child pointers as in traditional radix trees [24].

In the rest of this section, we describe how to apply the D-to-S rules to two representative indexes: B+tree and the Adaptive Radix Tree (ART). Two additional examples (i.e., Masstree and Skip List) can be found in [33].

3.1.1 Example Data Structures

B+tree: The B+tree is the most common index structure that is used in almost every OLTP DBMS [11]. It is a self-balancing search tree, usually with a large fanout. Although originally designed for disk-oriented databases to minimize disk seeks, B+trees have maintained their prevalence in the main-memory DBMSs. For our analysis, we use the STX B+tree [9] as our baseline implementation.

Adaptive Radix Tree: The Adaptive Radix Tree (ART) [24] is a fast and space-efficient data structure designed for in-memory databases. ART is a 256-way radix tree. Unlike traditional radix trees (or tries) where each node is implemented as a fixed-size (256 in this case) array of child pointers, ART uses four node types with different layouts and capacities adaptively to achieve better memory efficiency and better cache utilization. Figure 3.1 illustrates the three node layouts used in ART.

3.1.2 Rule #1: Compaction

This first rule seeks to generate a more efficient layout of an index's entries by minimizing the number of memory blocks allocated. This rule includes two steps. The first is to remove duplicate content. For example, to map multiple values to a single key (for secondary indexes), dynamic data structures often store the same key multiple times with different values. Such key duplication is unnecessary in a static data structure because the number of values associated with

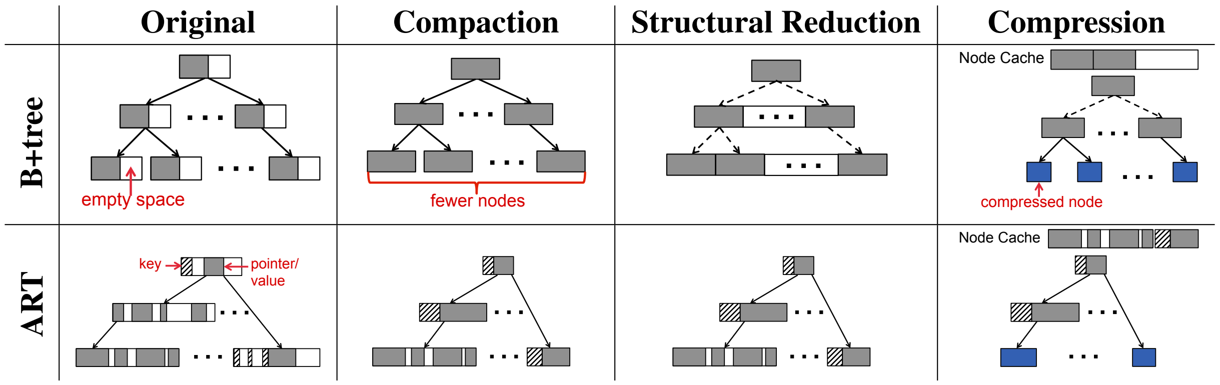


Figure 3.2: Examples of Applying the Dynamic-to-Static Rules – Solid arrows are pointers; dashed arrows indicate that the child node location is calculated rather than stored in the structure itself. The second column shows the starting points: the original dynamic data structures. After applying the Compaction Rule, we get intermediate structures in column three. We then applied the Structural Reduction Rule on those intermediate structures and obtain more compact final structures in column four. Column five shows the results of applying the Compression Rule, which is optional depending on workloads.

each key is fixed. The second step is to fill all allocated memory blocks to 100% capacity. This step may include modifications to the layouts of memory blocks/nodes. Memory allocation is done at a fine granularity to eliminate gaps between entries; furthermore, leaving spacing for future entries is unnecessary since the data structure is static. The resulting data structure thus uses fewer memory blocks/nodes for the same entries.

As shown in Figure 3.2, a major source of memory waste in both index types is the empty space in each node. For example, the expected node occupancy of a B+tree is only 69% [32]. For ART, our results show that its node occupancy is only 51% for 50 million 64-bit random integer keys. This empty space is pre-allocated to ingest incoming entries efficiently without frequent structural modifications (i.e., node splits). For B+tree, filling every node to 100% occupancy, therefore, reduces space consumption by 31% on average without any structural changes to the index itself. ART's prefix tree structure prevents us from filling the fixed-sized nodes to their full capacity. We instead customize the size of each node to ensure minimum slack space. This is possible because the content of each node is fixed and known when building the static structure. Specifically, let n denote the number of key-value pairs in an ART node ($2 \leq n \leq 256$). We choose the most space-efficient node layout in Figure 3.1 based on n. If $n \leq 227$, Layout 1 with array length n is used; otherwise, Layout 3 is used.

3.1.3 Rule #2: Reduction

The goal of this rule is to minimize the overhead inherent in the data structure. This rule includes removing pointers and other elements that are unnecessary for read-only operations. For example, the pointers in a linked list are designed to allow for fast insertion or removal of entries. Thus, removing these pointers and instead using a single array of entries that are stored contiguously in memory saves space and speeds up linear traversal of the index. Similarly, for a tree-based index with fixed node sizes, we can store the nodes contiguously at each level and remove pointers from the parent nodes to their children. Instead, the location of a particular node is calculated based on in-memory offsets. Thus, in exchange for a small CPU overhead to compute the location of nodes at runtime we achieve memory savings. Besides pointers, other redundancies include auxiliary elements that enable functionalities that are unnecessary for static indexes (e.g., transaction metadata).

The resulting data structures after applying the Reduction Rule to B+tree and ART are shown in Figure 3.2 (column four). We note that after the reduction, the nodes in B+tree are stored contiguously in memory. This means that unnecessary pointers are gone (dashed arrows indicate that the child nodes' locations in memory are calculated rather than stored). There are additional opportunities for reducing the space overhead when applying this rule to B+tree. For example, the internal nodes in the B+tree can be removed entirely. This would provide another reduction in space but it would also make point queries slower. Thus, we keep these internal nodes in B+tree. For ART, however, because its nodes have different sizes, finding a child node requires a "base + offset" or similar calculation, so the benefit of storing nodes contiguously is not clear. We, therefore, keep ART unchanged here and leave exploring other layouts as future work.

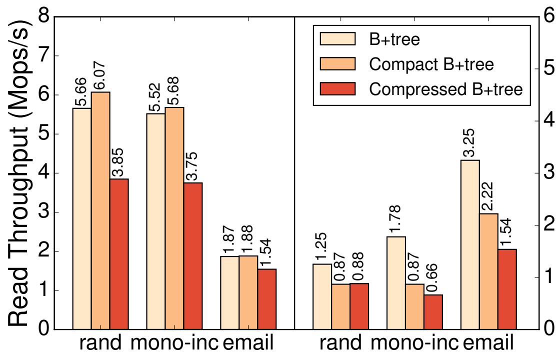
3.1.4 Rule #3: Compression

This final rule is to compress internal nodes or memory pages used in the index. For this step, we can use any general-purpose compression algorithm. We choose the ones that are designed to have fast decompression methods in exchange for a lower compression rate, such as Snappy [4] or LZ4 [3]. Column five of Figure 3.2 shows how we apply the Compression Rule to the B+tree and ART. Only the leaf nodes are compressed so that every point query needs to decompress at most one node. To minimize the cost of an expensive decompress-node operation, we maintain a cache of recently decompressed nodes. The node cache approximates LRU using the CLOCK replacement algorithm.

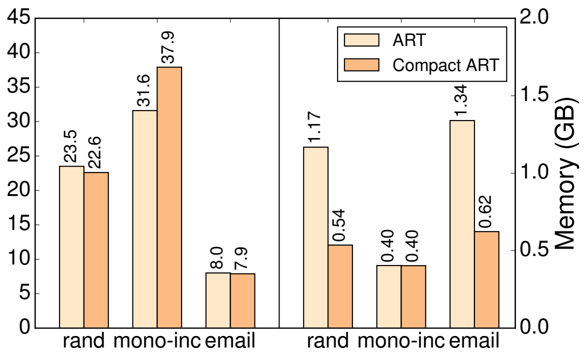
This rule is not always necessary for hybrid indexes because of its performance overhead. Our results in Section 3.1.5 show that using compression for in-memory data structures is expensive even with performance optimizations, such as node caching. Furthermore, the compression ratio is not guaranteed because it depends heavily on the workload, especially the key distribution. For many applications, the significant degradation in throughput may not justify the space savings; nevertheless, compression remains an option for environments with significant space constraints.

3.1.5 Evaluation

In this section, we evaluate the above static data structures created by applying the D-to-S rules. For the original data structure X, where X represents either B+tree or ART, Compact-X refers to the resulting data structure after applying the Compaction and Reduction rules to X (i.e., column 4 of Figure 3.2), while Compressed-X means that the data structure is also compressed using Snappy [4] according to the Compression Rule. We used the index microbenchmarks introduced



(a) B+tree



(b) ART

Figure 3.3: D-to-S Rules Evaluation – Read performance and storage overhead for the compacted and compressed data structures generated by applying the D-to-S rules. Note that the figures have different Y-axis scales (rand=random integer, mono-inc=monotonically increasing integer).

in Chapter 2. Specifically, we tested Zipf-distributed rand-int, mono-inc-int and email keys under the read-only workload.

As Figure 3.3 shows, the read throughput for the compact indexes is up to 20% higher compared to their original data structures. This is not surprising because these compact versions inherit the core design of their original data structures but achieve a more space-efficient layout with less structural overhead. This results in fewer nodes/levels to visit per look-up and better cache performance.

Figure 3.3 also shows that the compact indexes reduce the memory footprint by greater than 30% in all but one case. The savings come from higher data occupancy and less structural waste (e.g., fewer pointers). In particular, the Compact ART is only half the size for random integer and email keys because ART has relatively low node occupancy (54%) compared to B+tree and Skip List (69%) in those cases. For monotonically increasing (mono-inc) integer keys, the original ART is already optimized for space.

We also tested the Compression Rule on the B+tree. As shown in Figure 3.3a, although the Compressed B+tree saves additional space for the mono-inc (24%) and email (31%) keys, the throughput decreases from 18–34%. Since the other data structures have the same problems, we choose not to evaluate compressed versions of them and conclude that naïve compression is a poor choice for in-memory OLTP indexes.

3.2 Fast Succinct Tries

The above section describes how to get relatively space-efficient static data structures based on existing dynamic ones. It also shows that applying general-purpose compression algorithms to in-memory indexes hurts performance and does not guarantee a good compression ratio. In this section, we introduce a better way to further reduce the memory footprint of static data structures. Specifically, we provide a case study on Succinct Data Structures [6]: the Fast Succinct Trie

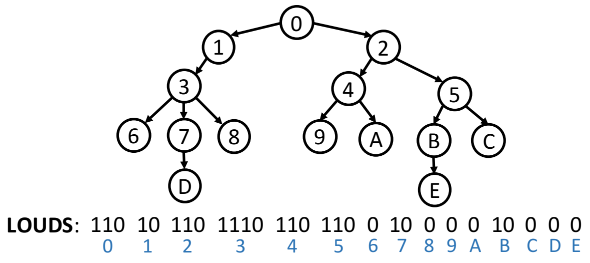

LOUDS: 110 10 110 1110 110 110 0 10 0 0 0 10 0 0 0
0 1 2 3 4 5 6 7 8 9 A B C D E

Figure 3.4: An example ordinal tree encoded using LOUDS.

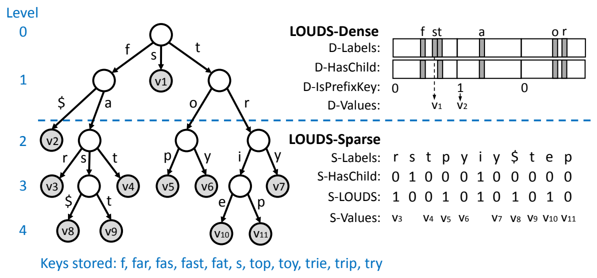

Level
LOUDS-Dense
D-Labels:
D-HasChild:
D-IsPrefixKey:
D-Values:
LOUDS-Sparse
S-Labels: r s t p y i y $ t e p
S-HasChild: 0 1 0 0 0 1 0 0 0 0 0
S-LOUDS: 1 0 0 1 0 1 0 1 0 1 0
S-Values: v3 v4 v5 v6 v7 v8 v9 v10 v11
Keys stored: f, far, fas, fast, fat, s, top, toy, trie, trip, try

Figure 3.5: LOUDS-DS – `$` represents `0xFF` and is used to indicate the situation where the prefix leading to a node is also a valid key.

(FST). To the best of our knowledge, FST is the first succinct trie that matches the performance of the state-of-the-art pointer-based index structures (existing succinct trie implementations are usually at least an order of magnitude slower).

3.2.1 Design

A tree representation is “succinct” if the space taken by the representation is close to the information-theoretic lower bound, which is the minimum number of bits needed to distinguish any object in a class. A trie of degree k is a rooted tree where each node can have at most k children with unique labels selected from set $\{0, 1, \ldots, k-1\}$. The information-theoretic lower bound (Z) of an n-node trie of degree k is approximately $n(k \log_2 k - (k-1)\log_2(k-1))$ bits [8]. In FST, we choose $k = 256$, then $Z \approx 9.44n$ bits.

The encoding schemes used in FST originated from the ***L**evel-**O**rdered **U**nary **D**egree **S**equence* (LOUDS), which is an encoding scheme for ordinal trees (i.e., tries without labels) [21]. LOUDS traverses the nodes in a breadth-first order and encodes each node’s degree using the unary code. For example, node 3 in Figure 3.4 has three children and is thus encoded as `‘1110’`.

FST’s design is based on the observation that the upper levels of a trie comprise few nodes but incur many accesses, while the lower levels comprise the majority of nodes, but are relatively “colder”. We, therefore, encode the upper levels using a fast bitmap-based encoding scheme, called LOUDS-Dense as shown in the top half of Figure 3.5, in which a child node search requires only one array lookup, choosing performance over space. Meanwhile, we encode the lower levels using a more space-efficient scheme, called LOUDS-Sparse as shown in the lower half of Figure 3.5, in which encoding a trie node only consumes **10 bits** (close to the information-theoretic optimum 9.44 bits), choosing space over performance. A detailed description of LOUDS-Dense and LOUDS-Sparse can be found in [34].

The combined encoding scheme is called LOUDS-DS. The dividing point between the upper and lower levels is tunable to trade performance and space. FST keeps the number of upper levels small in favor of the space efficiency provided by LOUDS-Sparse. We use the size ratio R ($R = 64$ by default) between LOUDS-Sparse and LOUDS-Dense to determine the dividing point. We restrict the size of the LOUDS-Dense levels to be less than $1/R$th of that of the

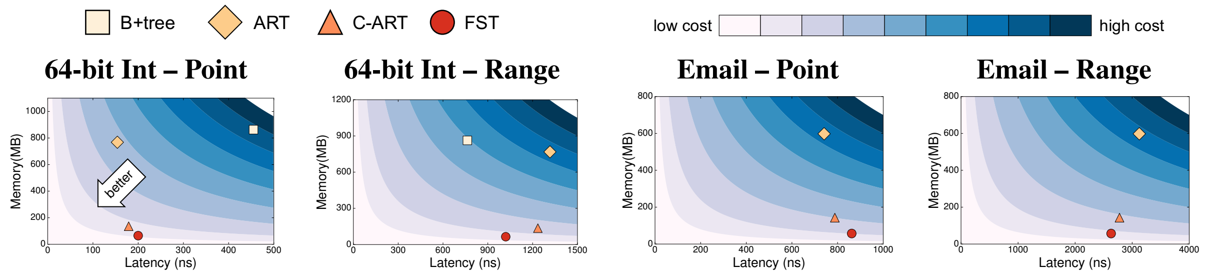


Figure 3.6: FST vs. Pointer-based Indexes

LOUDS-Sparse levels so that the overall size of the encoded trie is small and bounded.

Tree navigation in FST relies on the rank & select primitives. Given a bit-vector, $rank_1(i)$ counts the number of 1's up to position i, while $select_1(i)$ returns the position of the i-th 1. Modern rank & select implementations [19, 27, 31, 35] achieve constant time by using look-up tables (LUTs) to store a sampling of precomputed results so that they only need to count between the samples. With the rank & select support, FST can perform tree navigation operations such as "move to child" and "move to parent" in constant time.

[34] describes additional performance optimizations in FST including light-weight rank & select structures, vectorized label search, and prefetching.

3.2.2 Evaluation

In this section, we compare FST to B+tree, ART and the static Compact-ART (C-ART) obtained from Section 3.1. Again, we use the YCSB-based microbenchmarks introduced in Chapter 2. We begin each experiment by bulk-loading a sorted key list of 50M random integers or 25 emails into the index. The experiment then executes the read-only (point queries) or the scan-only (range queries) workload with Zipf key distribution. Note that ART, C-ART, and FST store only the unique key prefixes while B+tree must store the full keys in this experiment. Also note that the reported memory numbers exclude the space taken by the value pointers.

Figure 3.6 shows the comparison results. Each subfigure plots the locations of the four (three for email keys) indexes in the performance-space (latency vs. memory) map. We observe that FST is among the fastest choices in all cases while consuming less space. To better understand this trade-off, we define a cost function $C = P^r S$, where P represents performance (latency), and S represents space (memory). The exponent r indicates the relative importance between P and S. $r > 1$ means that the application is performance critical, and $0 < r < 1$ suggests otherwise. We define an "indifference curve" as a set of points in the performance-space map that have the same cost. We draw the equi-cost curves in Figure 3.6 using cost function $C = PS$ ($r = 1$), assuming a balanced performance-space trade-off. We observe that FST has the lowest cost (i.e., is most efficient) in all cases. In order for the second place (C-ART) to have the same cost as FST in the first subfigure, for example, r needs to be 6.7 in the cost function, indicating an extreme preference for performance.

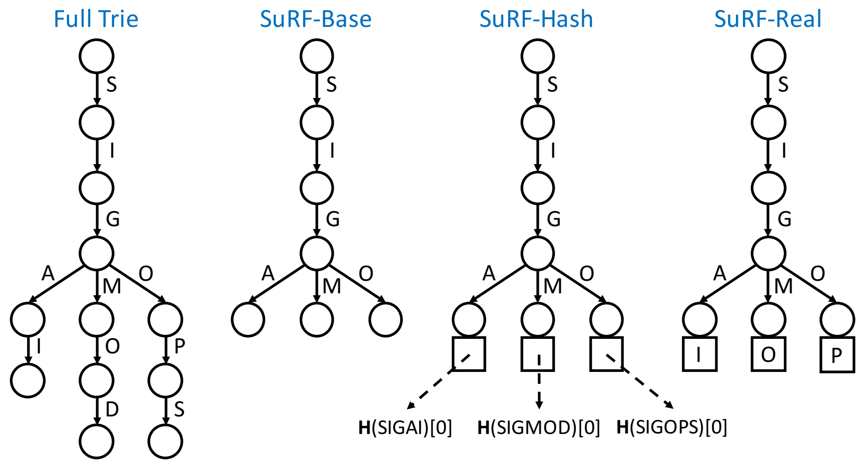


Figure 3.7: An example of deriving SuRF variations from a full trie.

3.3 Succinct Range Filters

Using FST, we have built the **Succinct Range Filter** (SuRF), a fast and compact filter that provides exact-match filtering ("Is key 42 in the SSTable?") and range filtering ("Are there keys between 42 and 1000 in the SSTable?"). Like Bloom filters [10], SuRF guarantees one-sided errors for membership tests. The key insight in SuRF is to transform the FST into an approximate (range) membership filter by replacing lower levels of the trie by suffix bits extracted from the key (either the key itself or a hash of the key). The next section presents three variations of SuRF derived from FST, as shown in Figure 3.7.

3.3.1 Design

SuRF-Base: Unlike a full FST that stores complete keys, the basic version of SuRF (SuRF-Base) stores the minimum-length key prefixes such that it can uniquely identify each key, trading accuracy for space. Specifically, SuRF-Base only stores an additional byte for each key beyond the shared prefixes. Figure 3.7 shows an example. SuRF-Base truncates the full trie by including only the shared prefix (`'SIG'`) and one more byte for each key (`'C'`, `'M'`, `'O'`). Our evaluation shows that SuRF-Base uses only 10 bits per key (BPK) for 64-bit random integers and 14 BPK for emails [34].

A false positive in SuRF-Base occurs when the prefix of the non-existent query key coincides with a stored key prefix. For example, in Figure 3.7, querying key `'SIGMETRICS'` will cause a false positive in SuRF-Base. Evaluation shows that SuRF-Base incurs a 4% false positive rate (FPR) for integer keys and a 25% FPR for email keys [34]. To improve FPR, we include different forms of key suffixes to allow SuRF to better distinguish between the stored key prefixes.

SuRF-Hash: As shown in Figure 3.7, SuRF with hashed key suffixes (SuRF-Hash) adds a few hash bits per key to SuRF-Base to reduce its FPR. Let H be the hash function. For each key K, SuRF-Hash stores the n (n is fixed) least-significant bits of $H(K)$ in FST's value array (which is empty in SuRF-Base). When a key (K') lookup reaches a leaf node, SuRF-Hash extracts the n least-significant bits of $H(K')$ and performs an equality check against the stored hash bits associated with the leaf node. Using n hash bits per key guarantees that the point query FPR of SuRF-Hash is less than 2^{-n} (the partial hash collision probability). Our evaluation shows that SuRF-Hash requires only 2–4 hash bits to reach 1% FPR. These extra hash bits, however, do not help range queries because they do not provide ordering information on keys.

SuRF-Real: Instead of hash bits, SuRF with real key suffixes (SuRF-Real) stores the n key bits immediately following the stored prefix of a key. Figure 3.7 shows an example when $n = 8$. SuRF-Real includes the next character for each key (`'I'`, `'O'`, `'P'`) to improve the distinguishability of the keys: for example, querying `'SIGMETRICS'` no longer causes a false positive. Unlike in SuRF-Hash, both point and range queries benefit from the real suffix bits to reduce false positives. For point queries, the real suffix bits are used the same way as the hashed suffix bits. For range queries when reaching a leaf node, SuRF-Real compares the stored suffix bits s to key bits k_s of the query key at the corresponding position. If $k_s \leq s$, the iterator points to the current key; otherwise, it advances to the next key in the trie.

Although SuRF-Real improves FPR for both point and range queries, the trade-off is that using real keys for suffix bits cannot provide as good FPR as using hashed bits because the distribution correlation between the stored keys and the query keys weakens the distinguishability of the real suffix bits.

3.3.2 Example Application: RocksDB

We integrated SuRF with RocksDB as a replacement for its Bloom filter. We modified RocksDB's point (*Get*) and range (*Seek*) query implementations to use SuRF.

For *Get(key)*, SuRF is used exactly like the Bloom filter. Specifically, at each level, RocksDB locates the candidate SSTable(s) and block(s) (level 0 may have multiple candidates) via the block indexes in the table cache. For each candidate SSTable, RocksDB queries the filter first and fetches the SSTable block only if the filter result is positive. If the filter result is negative, the candidate SSTable is skipped and the unnecessary I/O is saved.

For *Seek(lk, hk)*, if *hk* (high key) is not specified, we call it an *Open Seek*. Otherwise, we call it a *Closed Seek*. To implement *Seek(lk, hk)*, RocksDB first collects the candidate SSTables from all levels by searching for *lk* (low key) in the block indexes. Absent SuRFs, RocksDB must read block(s) from each candidate SSTable to find the global smallest key $K \geq$ *lk*. With SuRFs, however, instead of reading the actual blocks, which causes I/O, RocksDB queries the (in-memory) filters associated with the candidate SSTables and compares the returned partial keys to determine which SSTable the global smallest key $K \geq$ *lk* locates.

To illustrate how SuRFs benefit range queries, suppose a RocksDB instance has three levels (L_N, L_{N-1}, L_{N-2}) of SSTables on disk. L_N has an SSTable block containing keys 2000, 2011, 2020 with 2000 as the block index; L_{N-1} has an SSTable block containing keys 2012, 2014, 2029 with 2012 as the block index; and L_{N-2} has an SSTable block containing keys 2008, 2021, 2023 with 2008 as the block index. Consider the range query [2013, 2019]. Using only block

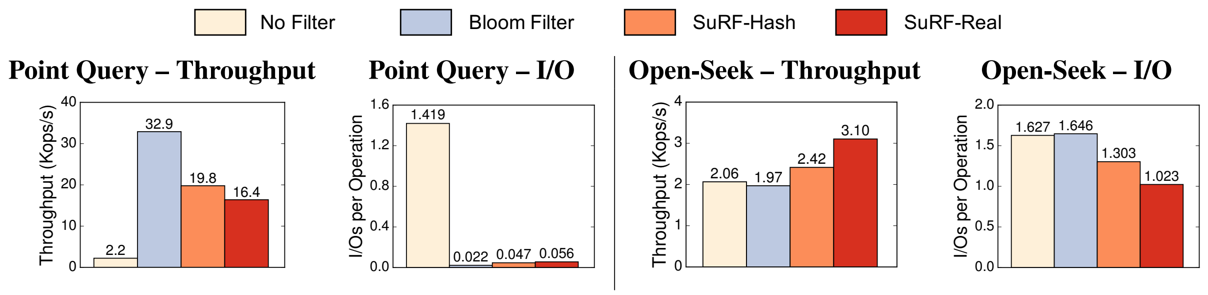


Figure 3.8: RocksDB point query and Open-Seek query evaluation under different filter configurations

indexes, RocksDB has to read all three blocks from disk to verify whether there are keys between 2013 and 2019. Using SuRFs eliminates the blocks in L_N and L_{N-2} because the filters for those SSTables will return false to query [2013, 2019] with high probability. The number of I/Os is likely to drop from three to one.

3.3.3 Evaluation

We evaluate SuRF-equipped RocksDB using a synthetic time-series benchmark. The key for each entry includes a 64-bit timestamp followed by a 64-bit tie-breaker. The associated value in the record is 1 KB long. The timestamps follow a Poisson distribution. The total size of the raw dataset is approximately 100 GB. We tested three types of database queries:

- **Point Query**: Given a full key, return the record if it exists.
- **Open-Seek Query**: Given a starting timestamp, return an iterator pointing to the earliest entry after that time.
- **Closed-Seek Query**: Given a time range, determine whether there are any entries within that time period. If yes, return an iterator pointing to the earliest entry in the range.

We configured[1] RocksDB according to Facebook's recommendations [16, 17]. We create four instances of RocksDB with different filter options: (1) no filter, (2) Bloom filter, (3) SuRF-Hash, and (4) SuRF-Real. We configure each filter to use an equal amount of memory: Bloom filters use 14 bits per key while SuRF-Hash and SuRF-Real include a 4-bit suffix per key. We first warm the cache with 1M uniformly-distributed point queries to existing keys. We then execute 50K application queries, recording the end-to-end throughput and I/O counts. The query keys are randomly generated. Our test machine has an Intel Core i7-6770HQ CPU, 32 GB RAM, and an Intel 540s 480 GB SSD. The resulting RocksDB instance has four levels (including Level 0) and uses 52 GB of disk space.

Figure 3.8 (left two figures) shows the result for point queries. Because the query keys are randomly generated, almost all queries return false. Using filters in point queries reduces I/O because they prevent unnecessary block retrieval. Using SuRF-Hash or SuRF-Real is slower

[1]Block cache size = 1 GB; OS page cache $\leq$ 3 GB. Enabled `pin_l0_filter_and_index_blocks_in_cache` and `cache_index_and_filter_blocks`.

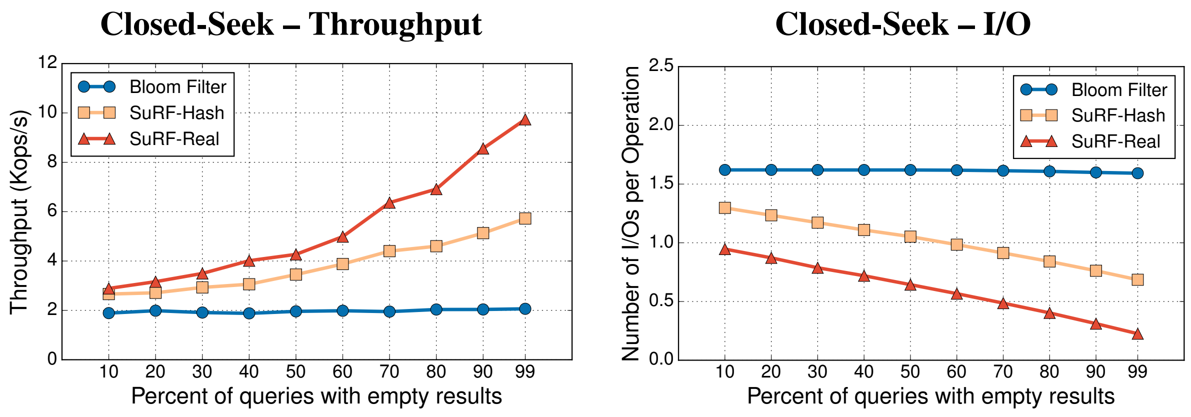


Figure 3.9: RocksDB Closed-Seek query evaluation under different filter configurations and range sizes

than using the Bloom filter because the 4-bit suffix does not reduce false positives as low as the Bloom filter configuration. SuRF-Real provides similar benefit to SuRF-Hash because the key distribution is sparse.

The main benefit of using SuRF is speeding range queries. Figure 3.8 (right two figures) shows that using SuRF-Real can speed up Open-Seek queries by 50%. SuRF-Real cannot improve further because an Open-Seek query requires reading at least one SSTable block and that SSTable block read is likely to occur at the last level where the data blocks are not available in cache. In fact, the I/O figure (rightmost) shows that using SuRF-Real reduces the number of I/Os per operation to 1.023, which is close to the maximum I/O reduction for Open-Seeks.

Figure 3.9 shows the throughput and I/O count for Closed-Seek queries. On the x-axis, we control the percent of queries with empty results by varying the range size. Similar to the Open-Seek query results, the Bloom filter does not help range queries and is equivalent to having no filter. Using SuRF-Real, however, speeds up the query by $5\times$ when 99% of the queries return empty. Without a range filter, every query must fetch candidate SSTable blocks from each level to determine whether there are keys in the range. Using the SuRF variants, however, avoids many of the unnecessary I/Os; Using SuRF-Real is more effective than SuRF-Hash in this case because the real suffix bits help reduce false positives at the range boundaries.

Chapter 4

The Hybrid Index Architecture

In the previous chapter, we studied how to compress in-memory static index structures while retaining high-performance. In this chapter, we shift our focus to dynamic indexes. We present the hybrid index architecture that uses the compressed static indexes developed in the previous chapter to gain space efficiency, while amortizing their performance overhead on dynamic operations by applying updates in batches. Our evaluation shows that the hybrid index architecture substantially reduces the per-tuple index space with only modest costs in throughput and latency even when the reclaimed memory is not exploited to improve system performance.

4.1 The Dual-Stage Architecture

As shown in Figure 4.1, the hybrid index architecture is comprised of two stages: the *dynamic* stage and the *static* stage. New entries are added to the dynamic stage. This stage uses a regular dynamic index structure (e.g., a B+tree or ART) and is kept small so that queries to the most recent entries, which are likely to be accessed and modified in the near future, are fast. As the size of the dynamic stage grows, the index periodically triggers the merge process (discussed in Section 4.2) and migrates aged entries from its dynamic stage to the static stage which uses a more space-efficient data structure (e.g., a Compact B+tree or Compact ART) to hold the bulk of the index entries. The static stage does not support direct key additions or modifications. It can only incorporate key updates in batches through the merge process.

A hybrid index serves read requests by searching the stages in order. To speed up this process, it maintains a Bloom filter for the keys in the dynamic stage so that most point queries search only one of the stages. Specifically, for a read request, the index first checks the Bloom filter. If the result is positive, it searches the dynamic stage and the static stage (if necessary) in sequence. If the result is negative, the index bypasses the dynamic stage and searches the static stage directly. The space overhead of the Bloom filter is negligible because the dynamic stage only contains a small subset of the index's keys.

A hybrid index handles value updates differently for primary and secondary indexes. To update an entry in a primary index, a hybrid index searches the dynamic stage for the entry. If the target entry is found, the index updates its value in place. Otherwise, the index inserts a

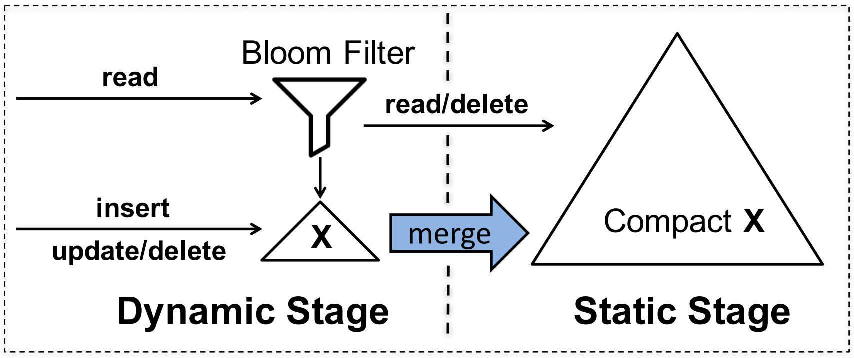


Figure 4.1: Dual-Stage Hybrid Index Architecture – All writes to the index first go into the dynamic stage. As the size of the dynamic stage grows, it periodically merges older entries to the static stage. For a read request, it searches the dynamic stage and the static stage in sequence.

new entry into the dynamic stage. This insert effectively overwrites the old value in the static stage because subsequent queries for the key will always find the updated entry in the dynamic stage first. Garbage collection for the old entry is postponed until the next merge. We chose this approach so that recently modified entries are present in the dynamic stage, which speeds up subsequent accesses. For secondary indexes, a hybrid index performs value updates in place even when the entry is in the static stage, which avoids the performance and space overhead of having the same key valid in both stages.

For deletes, a hybrid index first locates the target entry. If the entry is in the dynamic stage, it is removed immediately. If the entry is in the static stage, the index marks it "deleted" and removes it at the next merge. Again, depending on whether it is a unique index or not, the DBMS may have to check both stages for entries.

This dual-stage architecture has two benefits over the traditional single-stage indexes. First, it is space-efficient. The periodically-triggered merge process guarantees that the dynamic stage is much smaller than the static stage, which means that most of the entries are stored in a compact data structure that uses less memory per entry. Second, a hybrid index exploits the typical access patterns in OLTP workloads where tuples are more likely to be accessed and modified soon after they were added to the database. New entries are stored in the smaller dynamic stage for fast reads and writes, while older (and therefore unchanging) entries are migrated to the static stage only for occasional look-ups.

4.2 Merge

This section focuses on the merge process in the hybrid index architecture (i.e., merging tuples from the dynamic stage to the static stage). Although the merge process happens infrequently, it should be fast and efficient on temporary memory usage. Instead of using standard copy-on-write techniques, which would double the space during merging, we choose a more space-efficient merge algorithm that blocks all queries temporarily. In the proposed work (Chapter 5), we will propose a space-efficient non-blocking merge algorithm for concurrent indexes.

Our evaluation shows that the (blocking) merge algorithm takes 60 ms to merge a 10 MB B+tree into a 100 MB Compact B+tree. The merge time increases linearly as the size of the index grows. The space overhead of the merge algorithm, however, is only the size of the largest array in the dynamic stage structure, which is almost negligible compared to the size of the entire dual-stage index.

4.2.1 Merge Algorithm

Even though individual merge algorithms can vary significantly depending on the complexity of the data structure, they all have the same core component. The basic building blocks of a compacted data structure are sorted arrays containing all or part of the index entries. The core component of the merge algorithm is to extend those sorted arrays to include new elements from the dynamic stage. When merging elements from the dynamic stage, we control the temporary space penalty as follows. We allocate a new array adjacent to the original sorted array with just enough space for the new elements from the dynamic stage. The algorithm then performs in-place merge sort on the two consecutive sorted arrays to obtain a single extended array. The temporary space overhead for merging in this way is only the size of the smaller array, and the in-place merge sort completes in linear time.

Taking B+trees and ARTs for example. The steps for merging B+tree to Compact B+tree are straightforward. It first merges the leaf-node arrays using the algorithm described above and then rebuild the internal nodes. The internal nodes are constructed based on the merged leaf nodes so that the balancing properties of the structures are maintained. Merging ART to Compact ART, however, is more complicated. When merging two trie nodes, the (depth-first) algorithm recursively creates new merging tasks when two child nodes require further merging. The detailed algorithm can be found in [33].

4.2.2 Merge Strategy

What to Merge: On every merge operation, the system must decide which entries to move from the dynamic stage to the static stage. We choose a strategy, called merge-all, that merges the entire set of dynamic stage entries. This choice is based on the observation that many OLTP workloads are insert-intensive with high merge demands. Moving everything to the static stage makes room for the incoming entries and alleviates the merge pressure as much as possible.

When to Merge: The second design decision is what event triggers the merge process to run. We choose to use a ratio-based trigger: merge occurs whenever the size ratio between the dynamic and the static stages reaches a threshold. The advantage of a ratio-based trigger is that it automatically adjusts the merge frequency according to the index size. This strategy prevents write-intensive workloads from merging too frequently. Although each merge becomes more costly as the index grows, merges happen less often. One can show that the merge overhead over time is constant.

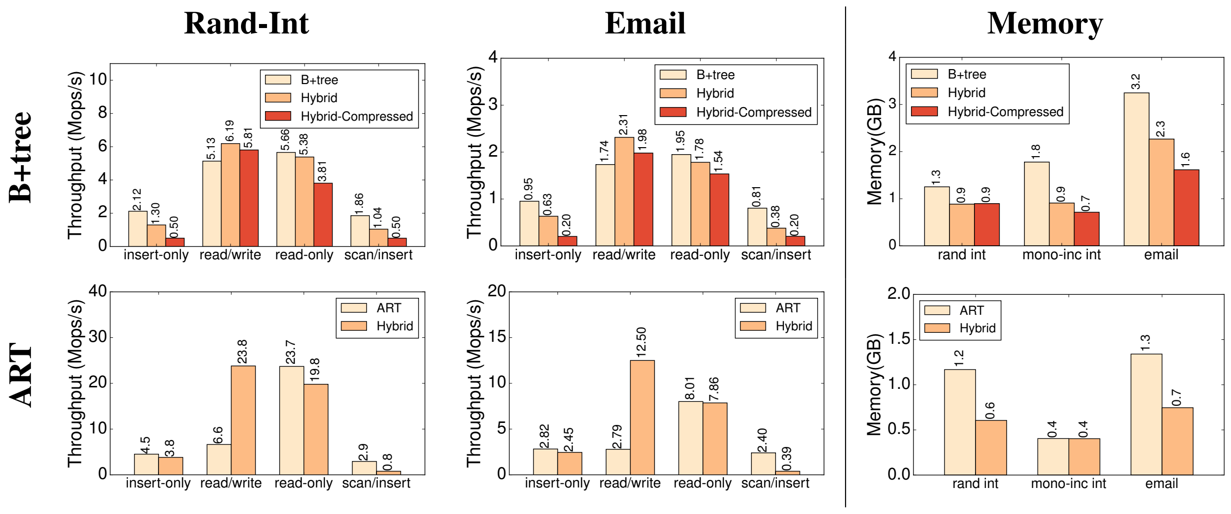


Figure 4.2: Hybrid Index vs. Original (Primary Indexes) – Throughput and memory measurements for the different YCSB workloads and key types when the data structures are used as primary key (i.e., unique) indexes. Note that the figures have different Y-axis scales.

4.3 Microbenchmark Evaluation

In this section, we compare the hybrid indexes to their corresponding original structures to show the trade-offs of adopting the hybrid index architecture. We have created Hybrid (-Compressed) B+tree, where a regular B+tree is used in the dynamic stage and a Compact (Compressed) B+tree (refer to Chapter 3) is used in the static stage, and Hybrid ART, where a regular ART is used in the dynamic stage and a Compact ART is used in the static stage. We evaluate these indexes using the index microbenchmarks introduced in Chapter 2. Specifically, we tested Zipf-distributed rand-int and email keys under the insert-only, read/write, read-only and scan/insert workloads.

Figure 4.2 shows the throughput and memory consumption results. The main takeaway is that both the hybrid indexes provide comparable throughputs (faster in some workloads, slower in others) to their original structures while consuming 30–50% less memory. Hybrid-Compressed B+tree achieves up to 30% additional space saving but loses a significant fraction of the throughput. This trade-off might only be acceptable for applications with tight space constraints.

Insert-only: Both hybrid indexes are slower than their original structures under the insert-only workloads since they have the highest merge demand. Besides merge, a hybrid index must check both the dynamic and static stages on every insert to verify that a key does not already exist to satisfy the primary key constraint. Such key uniqueness check causes about a 30% insert throughput drop.

Read/Write: The hybrid indexes' dual-stage architecture is better at handling skewed updates. Both hybrid indexes outperform their original structures under this workload because they store newly updated entries in the smaller (and therefore more cache-friendly) dynamic stage.

Read-only: Hybrid indexes' point-query performance is only slightly slower than the dy-

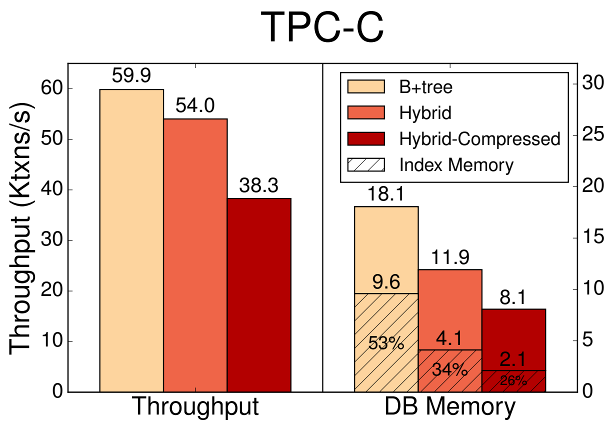


Figure 4.3: In-Memory Workloads – Throughput and memory measurements of the H-Store DBMS using the default B+tree, Hybrid, and Hybrid-Compressed B+tree when running the TPC-C workload that fit entirely in memory.

namic stage alone. This is because 1. the static stage structure is fast (refer to the evaluation in Section 3.1.5); 2. the workload is skewed, resulting in a large portion of the queries hit in the dynamic stage; and 3. the Bloom filter in front of the dynamic stage ensures that most reads only search one of the stages.

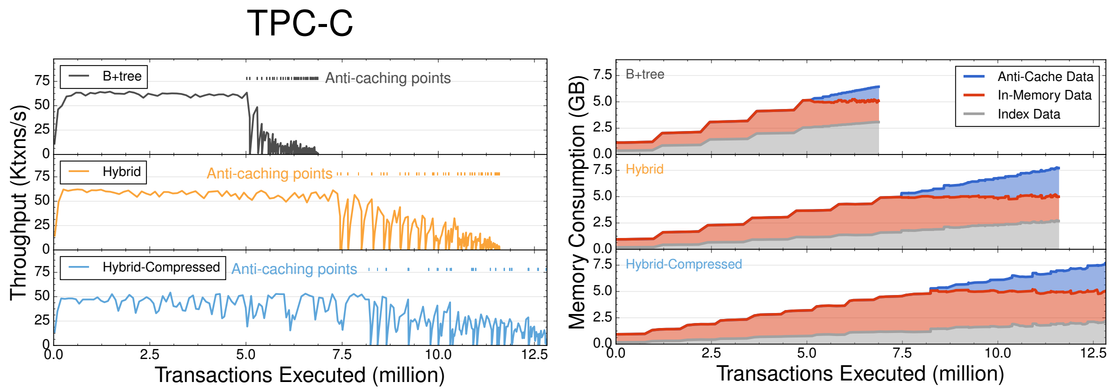
Scan/Insert: The hybrid indexes have lower throughput for range queries. This is expected because the dual-stage design requires comparing keys from both the dynamic and static stages to determine the "next" entry when advancing the iterator.

4.4 Full DBMS Evaluation

This section shows the effects of integrating hybrid indexes into the in-memory H-Store OLTP DBMS [2]. H-Store is a distributed, row-oriented DBMS that supports serializable execution of transactions over main memory partitions [22] The experiments in this section use the standard TPC-C benchmark [30]. We show that switching from regular B+tree to hybrid B+tree reduces the DBMS's footprint in memory and enables it to process transactions for longer without having to use secondary storage.

4.4.1 In-Memory Workloads

We first consider the case when the entire database fits in memory. We show that using hybrid indexes helps H-Store save a significant amount of memory. We ran TPC-C benchmark on H-Store with three different index types: (1) B+tree, (2) Hybrid B+tree, and (3) Hybrid-Compressed B+tree. The benchmark warms up for one minute after the initial load and then runs for five minutes on an 8-partition H-Store instance (one CPU core per partition). We report the average throughput (warm-up period excluded) and the memory consumption (measured at end of the benchmark).



(a) TPC-C Throughput Timelines **(b)** TPC-C Memory Breakdown

Figure 4.4: Larger-than-Memory Workloads – Throughput and memory measurements of the H-Store DBMS using B+tree, Hybrid, and Hybrid-Compressed B+tree as index structures when running workloads that are larger than the amount of memory available to the system. H-Store uses its anti-caching component to evict cold data from memory out to disk.

As shown in Figure 4.3, Hybrid B+tree and Hybrid-Compressed B+tree reduce the percentage of memory taken by indexes (from 53% to 34% and 26%, respectively). The trade-off is that Hybrid B+tree incurs a 10% average throughput drop compared to the original, which is modest considering the memory savings. Hybrid-Compressed B+tree, however, sacrifices throughput more significantly to reap its additional memory savings.

4.4.2 Larger-than-Memory Workloads

We then show that hybrid indexes can further help H-Store expand its capacity when the size of the database goes beyond physical memory. When both memory and disk are used, the memory saved by hybrid indexes allows the database to keep more hot tuples in memory. The database thus can sustain a higher throughput because fewer queries must retrieve tuples from disk.

We ran TPC-C on H-Store with anti-caching enabled. Anti-caching is a database-managed paging system that allows the DBMS to manage databases that are larger than the amount of memory available without incurring the performance penalty of a disk-oriented system [13]. We set the anti-caching eviction threshold to be 5 GB. The system's eviction manager periodically checks whether the total amount of memory used by the DBMS is above this threshold. If it is, H-Store selects the coldest data to evict to disk.

Figure 4.4 shows the experiment results. Using hybrid indexes, H-Store with anti-caching executes more transactions than the original B+tree index during the same execution time. Two features contribute to H-Store's improved capacity. First, with the same anti-caching threshold, hybrid indexes consume less memory, allowing the database to run longer before the first anti-caching eviction occurs. Second, even during periods of anti-caching activity, H-Store with hybrid indexes sustains higher throughput because the saved index space allows more tuples to remain in memory.

Chapter 5

Proposed Work

An index is comprised of the keys, the values, and the data structure. The work we have completed so far pushes hard on reducing the memory overhead of the data structures alone. As the sizes of our data structures get closer and closer to the information-theoretic optima, the content (i.e., the keys and the values) stored in the index starts to dominate the memory consumption. My proposed work, therefore, seeks opportunities to further shrink the size of in-memory indexes by rethinking what format of keys and values should appear in those indexes. We also propose to complete the hybrid index work by extending the techniques to support concurrent indexes.

5.1 Value Deduplication

An index value for an in-memory row store is often a 64-bit pointer. This pointer has two purposes: (1) to uniquely identify a tuple, and (2) to record the tuple's location in the virtual memory space. The key insight in the first proposed idea is to separate the two functionalities to achieve compression. Specifically, an index no longer stores 64-bit virtual addresses (VAs) as values. Instead, it stores (shorter) unique identifiers called tuple IDs or TIDs. Suppose that a table contains N entries. To uniquely identify each tuple in the table, we only need N TIDs (i.e., from 0 to $N - 1$), which requires $log_2 N$ bits instead of 64 bits per TID. Even for a table with one billion entries, using a 30-bit TID is enough. Meanwhile, each database table maintains an additional data structure that maps TIDs to VAs. An index search, thus, first obtains a TID from the index structure and then translate the TID to tuple pointers via the TID-to-VA map.

The idea of value indirection is not new. It is typically used to allow transparent relocation of physical storage blocks. For example, Bw-tree [25] maintains a mapping table that maps logical page IDs to either a flash page address or a memory pointer. What's new in this proposal is to leverage this basic idea to achieve more.

In the rest of this section, we assume a table with N tuples, M indexes (one primary index), and a fixed tuple size L.

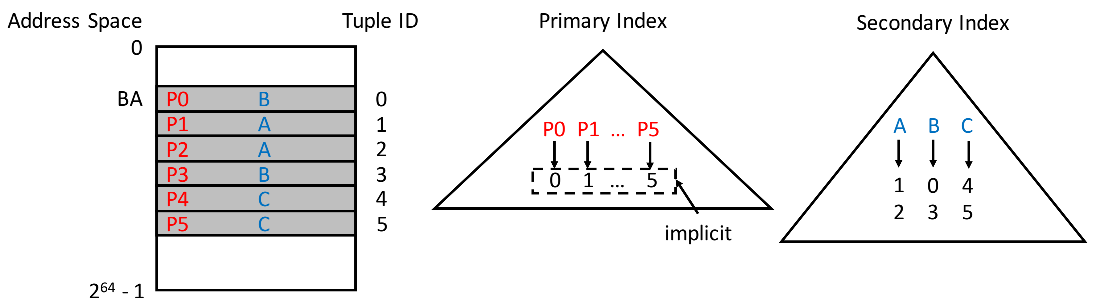


Figure 5.1: Value Deduplication: Static Case:

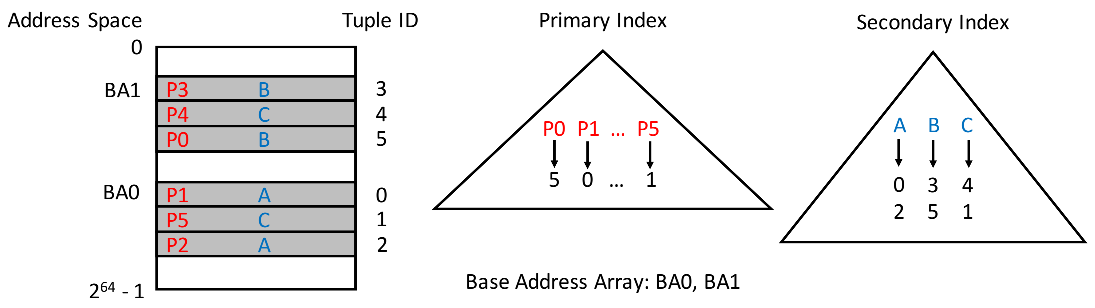


Figure 5.2: Value Deduplication: Semi-Static Case

5.1.1 Static Case

Let's first consider the case where the table is immutable. In this case, we store all the tuples in a contiguous memory block according to their primary key order, as shown in Figure 5.1. We assign TIDs in primary key order, too. The TID-to-VA map then is implicit. Suppose that the base address of the table is BA, then $VA = BA + TID \times L$ for each tuple. Moreover, the primary index can omit storing the TIDs as values because the TIDs are the same as the key orders.

The total space taken by the index values, therefore, is $(M-1)Nlog_2N$ bits ($(M-1)log_2N$ bits per tuple). Comparing to $64M$ bits per tuple when using pointers, the space saving is significant. For example, for $N = 1$ billion and $M = 3$, the space saving is 69%. The performance impact is negligible in this case because computing $VA = BA + TID \times L$ is cheap as the constants BA and L are almost guaranteed to be in cache.

5.1.2 Semi-Static Case

We then consider the semi-static case, where table inserts come in batch. We assume in-place deletes (via atomic tombstones) and in-place updates (via locks or atomically appending delta records). Under the semi-static case, if we continue to assign TIDs based on the primary key order, the newly arrived tuples will invalidate the previously assigned TIDs and cause all indexes

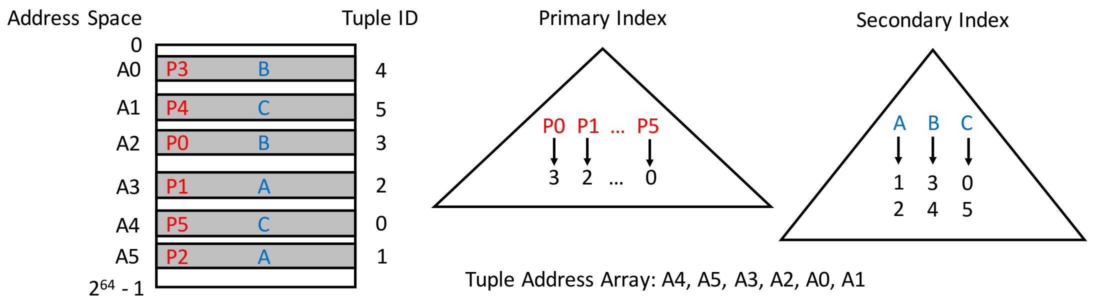


Figure 5.3: Value Deduplication: Dynamic Case

to rebuild. We instead assign TIDs in tuple arriving order.

Because the table size may grow (when a new batch of tuples arrives) or shrink (when garbage collection is invoked), storing all tuples in a contiguous memory block as in the (absolute) static case is no-longer memory-management-friendly. We, therefore, allocate a number of fixed-sized (S) memory blocks (like pages), where each block holds $K = S/L$ tuples, as shown in Figure 5.2. The tuple blocks are ordered: the i-th ($0 \leq i \leq \lceil N/K \rceil - 1$) tuple block stores tuples with $iK \leq TID \leq (i+1)K - 1$. The base addresses of the tuple blocks are recorded in order in an address array. The TID-to-VA map in this case is implicit, too. Suppose that the base address of the i-th tuple block is BA(i). Given TID, $VA =$ BA(TID/K) $+(TID - TID/K)L$.

The total space taken by the index values is $MNlog_2N + 64 \times (N/K)$ bits, where $64 \times (N/K)$ represents the space (almost negligible) for storing the base addresses of the tuple blocks. Taking $N = 1$ billion, $M = 3$, and $K = 10,000$ as an example, the space saving is 53% compared to using 64-bit pointers for values.

The performance impact during normal index lookups is negligible because the base addresses for tuple blocks are very likely cached. Garbage collection takes tuples with the largest TIDs (i.e., most recently inserted) to filling the "holes" created by deleted entries. Because of the TID change for those tuples, the values of the corresponding index entries must update accordingly.

5.1.3 Dynamic Case

Finally, we consider the (absolute) dynamic case, where we have no control over the table layout: individual tuples are scattered in the virtual memory space. In this case, we continue to assign TIDs in tuple arriving order. The TID-to-VA map, however, requires a dynamic address array to implement, as shown in Figure 5.3. Garbage collection in the address array follows a similar process as in the semi-static case.

The total space taken by the index values is $MNlog_2N + 64N$ bits. Taking $N = 1$ billion and $M = 3$ as an example, the space saving is 20% compared to using 64-bit pointers for values. Performance-wise, each index lookup requires an additional memory reference to translate TIDs to pointers.

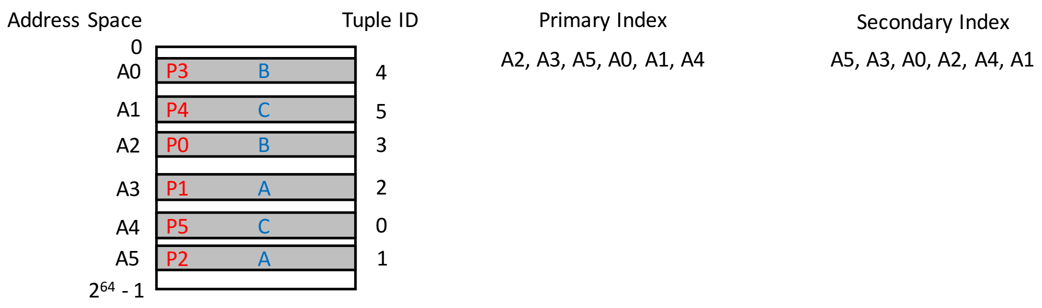


Figure 5.4: Permutation Index

5.2 Permutation Indexes

This proposed idea is inspired by the observation that the role of indexes has changed when the underlying database moves from disk to memory. Indexes in traditional disk-oriented databases are designed to reduce the number of I/Os by as many as possible. They, therefore, keep the full keys in memory so that no disk access is required during the index search before reaching the leaf page. For in-memory databases, however, accessing a tuple is as cheap as accessing an index node (excluding cache effects). Referring to the tuple storage is no longer forbidden and an index is mainly used to reduce the search complexity and to facilitate scans.

Based on this observation, we propose the permutation index that only stores a permutation of tuple pointers (or TIDs, see Section 5.1) in the indexed-key order. Figure 5.4 shows an example. An index lookup is accomplished by performing a binary search on the permutation: upon reaching a tuple pointer, the index dereferences it and extracts the key from the actual tuple for comparison. The permutation index completely eliminates the need to (duplicately) store keys in an index. The trade-off is that it doubles the number of memory references in a cache-unfriendly way during an index search.

We preliminarily evaluate the performance of permutation indexes. We created a table of N items with a 64-byte tuple size, as shown in Figure 5.5. The tuples are stored contiguously in memory. The first column contains primary keys from 0 to $N - 1$. A primary index is created on this column. The permutation index for this column is implicit because it is a simple arithmetic sequence (i.e., $BA, BA + 64, ..., BA + 64(N - 1)$). The second column is populated with a permutation of the primary keys. The column is indexed by a secondary index. The workload contains 10 million random point queries on existing keys. We tested three index types: the permutation index, the stx-btree, and the ART.

Figure 5.6 presents the results. ART continues to dominate as we have shown in [33]. Compared to the secondary permutation index, the implicit primary permutation index is twice as fast because it saves half of the memory references by directly binary searching the table. The takeaway from these experiments is that the performance penalty caused by permutation indexes is likely in the acceptable range and can be justified by their memory savings.

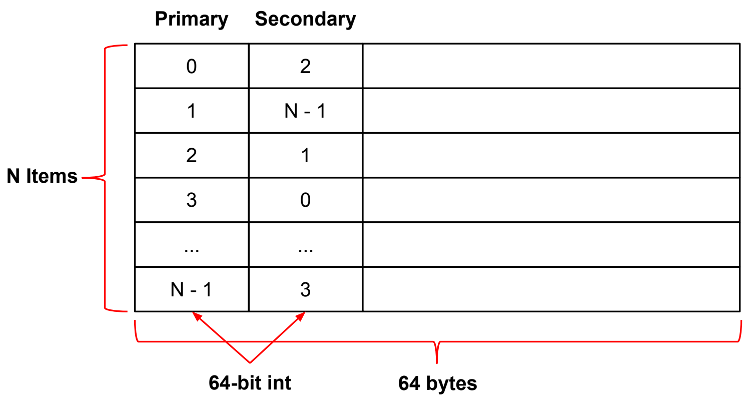


Figure 5.5: Permutation Index Evaluation: Table Layout

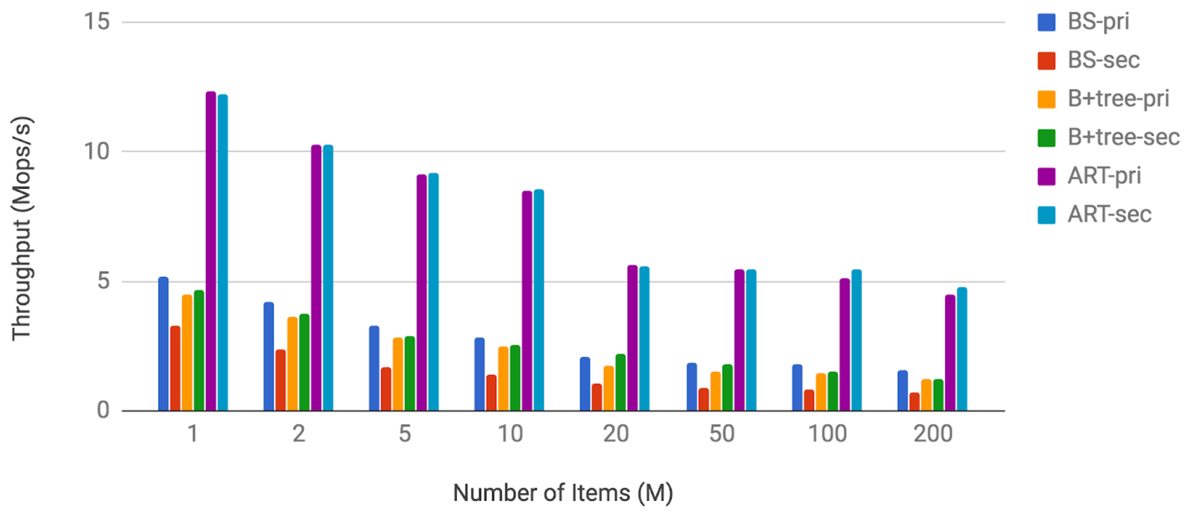


Figure 5.6: Permutation Index Evaluation: Results

5.3 Concurrent Hybrid Indexes

Finally, we propose to extend the hybrid index architecture to support concurrent indexes. Referring to Chapter 4, the hybrid index architecture includes three main components: the dynamic stage, the static stage, and the merge process. In order to support multiple readers and writers at the same time, each of the three components must be concurrent. For the dynamic stage, we can use an existing concurrent index such as Bw-tree [25], Masstree [26] or ART [24]. The static stage is perfectly concurrent because it is read-only. The challenge in building concurrent hybrid indexes is to design a space-efficient non-blocking merge algorithm.

The proposed non-blocking merge algorithm works as follows. We introduce a temporary third stage between the dynamic and the static stage, called the intermediate stage. When a merge is triggered, the current dynamic stage freezes and becomes the intermediate stage. Meanwhile, a new empty dynamic stage is created at the front to continue receiving writes. In this way, all the writes are independent of the merge process and are thus non-blocking.

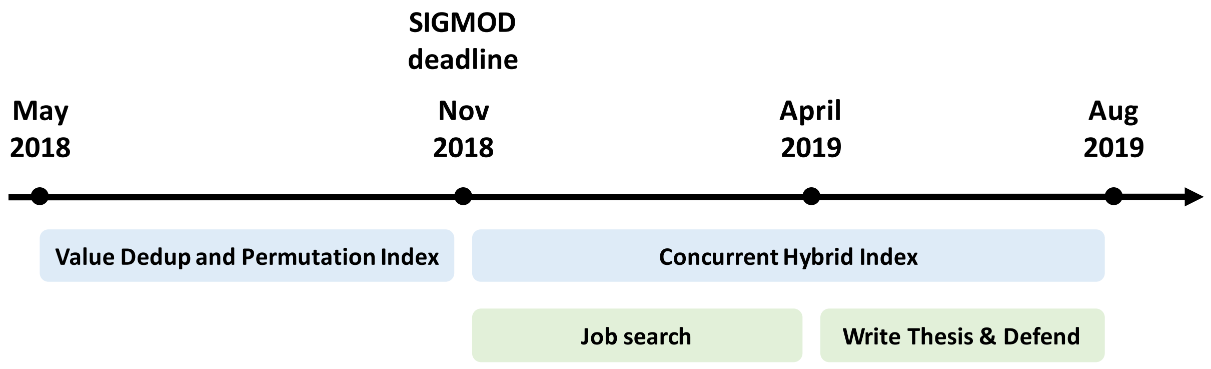
The problem reduces to merging two read-only data structures (from the intermediate stage to the static stage) without blocking access to any index item. A naive solution is to create an entire copy of the static stage and perform the merge on the copy. This is a poor solution because it doubles the memory use during a merge.

The key observation here is that the merged-in items in the static-stage will not be accessed until the intermediate stage is deleted because read requests for those newly merged-in items will hit in the intermediate stage first without touching the final stage. This means that the static stage structure can be at any partially-merged state as long as we keep the existing index items accessible. Based on this observation, we can merge the two structures incrementally by performing atomic updates at the node level: creating a new version of the node and then atomically swapping out the old version.

Garbage collecting the swapped-out old node is non-trivial because concurrent threads may still hold references to it. To determine when the old node is safe to reclaim, we propose to introduce, for each thread, a local operation counter that is incremented every time before an index operation starts. Because the counters are thread-local, they are cheap. Using these thread-local counters, we can generate a GC time tag for the old node (when it is swapped out) by reading all the thread-local counters and compute the maximum (C_{max}). The GC condition for this old node is when counters from all threads become larger than C_{max} because any operation started after C_{max} must be directed to the new version of the node. We write C_{max} to a globally visible location at each GC request and use that to synchronize the thread-local counters so that reclaiming the old node will not be held by slow threads.

Chapter 6

Timeline



Bibliography

[1] LZ77. https://en.wikipedia.org/wiki/LZ77_and_LZ78#LZ77. 1

[2] H-Store. http://hstore.cs.brown.edu. 1, 4.4

[3] LZ4. https://code.google.com/p/lz4/. 1, 3.1.4

[4] Snappy. https://github.com/google/snappy. 1, 3.1.4, 3.1.5

[5] zlib. http://www.zlib.net. 1

[6] Succinct data structures. https://en.wikipedia.org/wiki/Succinct_data_structure, 2016. 3, 3.2

[7] Memory prices (1957-2017). https://jcmit.net/memoryprice.htm, 2017. 1

[8] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. 3.2.1

[9] Timo Bingmann. STX B+ tree C++ template classes. http://panthema.net/2007/stx-btree/. 3.1.1

[10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. 3.3

[11] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. 3.1.1

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010. 2

[13] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A new approach to database management system architecture. *VLDB*, 6(14): 1942–1953, September 2013. 4.4.2

[14] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984. 1

[15] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1–12, 2013. 1

[16] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017. 3.3.3

[17] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/

wiki/RocksDB-Tuning-Guide, 2015. 3.3.3

[18] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. 1

[19] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Proc. of WEA*, pages 27–38, 2005. 3.2.1

[20] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008. ISBN 978-1-60558-102-6. 1

[21] Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science*, pages 549–554. IEEE, 1989. 3.2.1

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *VLDB*, 1(2):1496–1499, 2008. 4.4

[23] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12). 1

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. ICDE, pages 38–49, 2013. 3.1, 3.1.1, 5.3

[25] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 302–313. IEEE, 2013. 5.1, 5.3

[26] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. EuroSys, pages 183–196, 2012. 5.3

[27] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proc. of SEA '12*, pages 295–306, 2012. 3.2.1

[28] Radu Stoica and Anastasia Ailamaki. Enabling efficient os paging for main-memory oltp databases. DaMoN, 2013. 1

[29] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007. ISBN 978-1-59593-649-3. 1

[30] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007. 4.4

[31] Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 154–168, 2008. 3.2.1

[32] Andrew Chi-Chih Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978. 3.1.2

[33] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes.

In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 1567–1581. ACM, 2016. 3.1, 4.2.1, 5.2

[34] Huanchen Zhang, Hyeontaek Lim, Leis Viktor, 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)*, pages 1567–1581. ACM, 2018. 3.2.1, 3.3.1

[35] Dong Zhou, David G Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *International Symposium on Experimental Algorithms*, pages 151–163. Springer, 2013. 3.2.1