Memory-Efficient Search Trees for Database Management Systems

Huanchen Zhang

CMU-CS-20-101

Febuary 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Michael Kaminsky
Andrew Pavlo
Kimberly Keeton, Hewlett-Packard Labs

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

This research was sponsored by the National Science Foundation under grant number CNS-1314721, Intel ISTC-CC, Intel ISTC-VCC, and the VMware University Research Fund. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: search tree, memory-efficiency, database management system, indexing, range filtering, succinct data structure, key compression

To my son.

Abstract

The growing cost gap between DRAM and storage together with increasing database sizes means that database management systems (DBMSs) now operate with a lower memory to storage size ratio than before. On the other hand, modern DBMSs rely on in-memory search trees (e.g., indexes and filters) to achieve high throughput and low latency. These search trees, however, consume a large portion of the total memory available to the DBMS. This dissertation seeks to address the challenge of building compact yet fast in-memory search trees to allow more efficient use of memory in data processing systems. We first present techniques to obtain maximum compression on fast read-optimized search trees. We identified sources of memory waste in existing trees and designed new succinct data structures to reduce the memory to the theoretical limit. We then introduce ways to amortize the cost of modifying static data structures with bounded and modest cost in performance and space. Finally, we approach the search tree compression problem from an orthogonal direction by building a fast string compressor that can encode arbitrary input keys while preserving their order. Together, these three pieces form a practical recipe for achieving memory-efficiency in search trees and in DBMSs.

Acknowledgments

I feel very fortunate for having the opportunity to work with the “Big Four” of my advising team: David G. Andersen, Michael Kaminsky, Andrew Pavlo, and Kimberly Keeton. This thesis would not have been possible without them.

I thank my advisor Dave for believing in me when I stumbled during the journey. His constant support in and beyond research has helped me overcome the hard times. Dave taught me not only the knowledge necessary for writing this thesis but also the mindset of being a qualified researcher. The best part of my time in graduate school was to figure out fun data structure problems with Dave on the whiteboard. Dave introduced me to the field of Succinct Data Structures, which eventually becomes a core component of this thesis.

I am also thankful to Michael for shepherding my research with rigorousness. The quality of this thesis would be much compromised without his thoughtful questions and suggestions. I am very grateful to Andy for leading me to the lovely database community where I find my research passion. I thank Andy for giving me countless pieces of useful advice on research and career, as well as letting me kick a yoga ball to his crotch. My debt to Kim goes beyond her being an excellent internship and research mentor. I view her as a role model, and I cherish our friendship.

Many collaborators have contributed to the content of this thesis. I thank Lin Ma and Rui Shen for their effort in evaluating the Hybrid Index. I thank Hyeontaek Lim and Viktor Leis for offering valuable input on the SuRF project. I thank Xiaoxuan Liu for implementing the ALM algorithm in HOPE and for being the coolest master student I have ever mentored.

I would like to thank my colleagues at CMU for making my graduate school experience enriching and fun. I thank Greg Ganger, Rashmi Vinayak, Justine Sherry, and Bryan Parno for helping me with my writing and speaking skills. I thank Anuj Kalia, Conglong Li, Prashanth Menon, and Jack Kosaian for making our office “legendary”. I thank the members and friends of the FAWN group – Hyeontaek Lim, Dong Zhou, Anuj Kalia, Conglong Li, Sol Boucher, Angela Jiang, Thomas Kim, Chris Canel, Giulio Zhou, Daniel Wong, and Charlie Garrod – and other fellow students in PDL– Abutalib Aghayev, Joy Arulraj, Matt Butrovich, Dominic Chen, Andrew Chung, Henggang Cui, Aaron Harlap, Kevin Hsieh, Saurabh Kadekodi, Rajat Kateja, Michael Kuchnik, Yixin Luo, Charles McGuffey, Jun Woo Park, Kai Ren, Dana Van Aken, Ziqi Wang, Jinliang Wei, Lin Xiao, Jason Yang, Michael Zhang, and Qing Zheng. I thank Deb Cavlovich, Angy Malloy, Karen Lindenfelser, and Joan Digney for their greatest administrative support throughout my Ph.D. study.

I also want to thank my good friends Zhuo Chen, Ziqiang Feng, Yan Gu, Wenlu Hu, Han Lai, Conglong Li, Danyang Li, Julian Shun, Yihan Sun, Junjue Wang, Xiaocheng Zhang and many others for making my life in Pittsburgh colorful.

I am especially grateful to my wife Yingjie Zhang who has always stood by my side, sharing my laughter and tears. My parents Suping Zhu and Jiming Zhang have made great sacrifices to help me get this far. I could not have asked for a better family. Finally, I am excited to have a wonderful new life coming and joining my journey. I dedicate this thesis to him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory has been a limiting resource since people first built computer systems. Such capacity constraint exists until today, and it seems to get worse. Although DRAM price dropped drastically in the last decade (i.e., the 2000s), the trend stops as Moore's Law fades: DRAM price has been relatively stable since 2013 for over six years [28]. Main-memory today is still a non-trivial capital cost when purchasing new equipment, and it incurs real operational costs in terms of power consumption. Studies have shown that memory accounts for 5% – 40% of the total power consumed by a database server [102, 110, 128]. Meanwhile, we observe a growing cost gap between memory and storage. For example, the price for solid-state drives (SSDs) keeps decreasing thanks to new technologies such as the introduction of 3D NAND in 2012 [16]. The $/GB ratio of DRAM versus SSDs increased from 10× in 2013 to 40× in 2018 [28, 41].

Together with the rapidly growing database sizes, database management systems (DBMSs) now operate with a lower memory to storage size ratio than before. Today, a typical mid-tier Amazon Elastic Compute Cloud (EC2) machine optimized for transactional workloads has roughly a 1:30 DRAM to SSD ratio [21]. DBMS developers in turn are changing how they implement their systems' architectures. For example, a major Internet company's engineering team assumes a 1:100 memory to storage ratio (instead of 1:10 a few years ago) to guide their future system designs [77].

On the other hand, modern online transaction processing (OLTP) applications demand that most if not all transactions complete in a short time (e.g., submillisecond) [155] – performance that is achievable only when the working set fits in memory. For instance, Alibaba's e-commerce platform maintains an average response time (i.e., transaction latency) of less than $0.5ms$ even during the Singles' Day Global Shopping Festival, where the database processes up to 70 million transactions per second [96]. Improving memory-efficiency in a DBMS, therefore, has two benefits. First, for a fixed working set size, reducing the required memory can save cost, both in capital and operating expenditures. Second, higher memory-efficiency allows the DBMS to keep more data resident in mem-

	Tuples	Primary Indexes	Secondary Indexes
TPC-C [156]	42.5%	33.5%	24.0%
Voter [45]	45.1%	54.9%	0%
Articles [12]	64.8%	22.6%	12.6%

Table 1.1: Index Memory Overhead – Percentage of the memory usage for tuples, primary indexes, and secondary indexes in H-Store [13] using the default indexes (DB size ≈ 10 GB).

ory, and a larger working set enables the system to achieve better performance with the same hardware.

To ensure fast query execution, applications often maintain many search trees (e.g., indexes and filters) in memory to minimize the number of I/Os on storage devices. But these search trees consume a large portion of the total memory available to the DBMS [59, 109, 168]. Table 1.1 shows the relative amount of storage used for indexes in several OLTP benchmarks deployed in H-Store main-memory DBMS [13]. We used the DBMS's internal statistics API to collect these measurements after running the workloads on a single node until the database size ≈ 10 GB. We found that indexes (i.e., B+trees in this case) consume up to 58% of the total database size for these benchmarks, which is commensurate with our experiences with real-world OLTP systems, where tuples are relatively small, and each table can have multiple indexes. Reducing the memory footprint of these search trees can lead to the aforementioned benefits: lower costs and larger working sets. However, simply getting rid of all or part of the search trees is suboptimal because they are crucial to query performance.

1.1 Existing Solutions: Performance vs. Space

Search trees in DBMSs mainly fall into two categories. The first is the B-tree/B+tree [69] family, including the more recent Cache Sensitive B+trees (CSB+trees) [148] and Bw-Trees [117, 160][1]. These trees store keys horizontally side-by-side in the leaf nodes and have good range query performance. The second category includes tries and radix trees [53, 60, 63, 83, 94, 105, 112, 124, 129]. They store keys vertically to allow prefix sharing. Recent memory-efficiency tries such as ART [112] and HOT [60] are proven to be faster than B+trees on modern hardware.

Existing search tree designs trade between performance and space. Performance-optimized search trees such as the Bw-Tree [117], the Adaptive Radix Tree (ART) [112], and Masstree [124] consume a large amount of memory, and they are a major factor in

[1] This category also includes skip lists [141]

the memory footprint of a database, as shown in Table 1.1. Although a performance-optimized search tree today can execute a point or short-range query in a few hundred nanoseconds – a latency that is equivalent to several DRAM accesses only, few compression techniques can reduce the search tree sizes significantly while maintaining their high-performance.

Most common compression techniques for search trees leverage general-purpose block compression algorithms such as LZ77 [27], Snappy [39], and LZ4 [26]. For example, InnoDB uses the zlib library [18] to compress its B+tree pages/nodes before they are written to disk. This approach works well for disk-based search trees because it minimizes data movement between disk and memory. For in-memory search trees, however, block compression algorithms impose too much computational overhead because the DBMS is unable to operate directly on the search tree data without having to decompress it first. With hundreds of nanoseconds, the fastest block compression algorithms can decompress only a few 4 KB memory pages [26].

Theoretically, we can only compress search trees to a certain degree before they start to lose the necessary information to answer queries correctly. This limit is called the information-theoretic lower bound. A data structure is called "succinct" if its size is close to the information-theoretic lower bound. Succinct trees [99] have been studied for over two decades, and there is rich literature [52, 57, 90, 91, 100, 126, 131, 143, 150] in theory and practice. A key advantage of succinct trees besides their space-efficiency is that they can answer queries directly from their compact representations without expensive decoding/decompressing operations.

Although succinct trees work well in specific scenarios, such as in information retrieval and XML processing [136, 144], their application in more general real-world systems is limited. To the best of our knowledge, none of the major databases and storage systems use succinct trees for data storage or indexing. There are two major reasons for their limited use. First, succinct trees are static. Inserting or updating an entry requires reconstructing a significant part of the structure. This is a fundamental limitation of all succinct data structures. Second, existing implementations of succinct trees are at least an order of magnitude slower than their corresponding pointer-based uncompressed trees [52]. This slowdown is hard to justify for most systems despite the space advantage.

1.2 A Pareto Improvement

As discussed in the section above, existing search tree designs typically focus on one of the optimization goals: performance or memory-efficiency. Performance-optimized trees use a lot of space to guarantee fast queries, while memory-optimized trees require

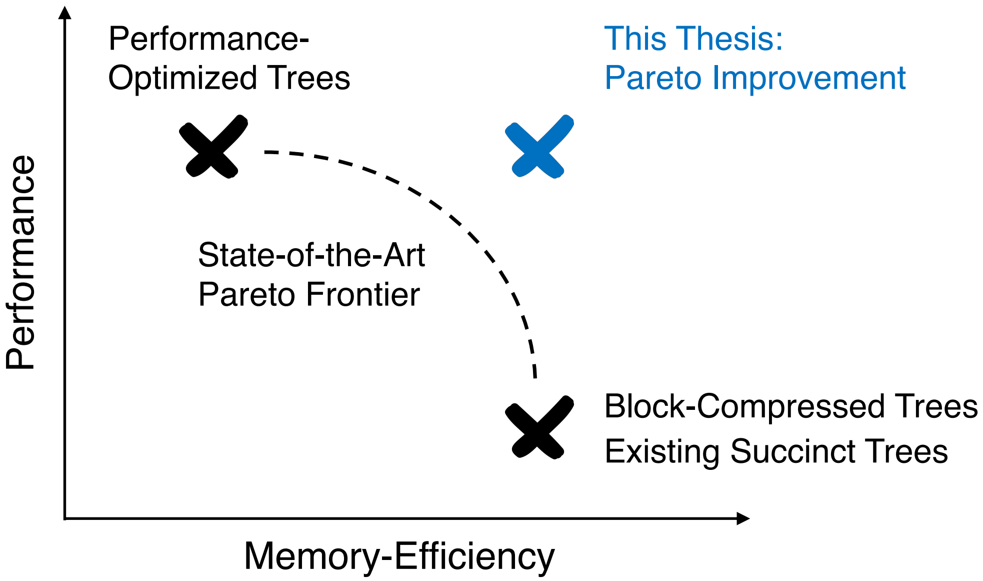


Figure 1.1: A Pareto Improvement – The goal of this thesis is to advance the state of the art in the performance-space trade-off when building in-memory search trees.

a lot of computation to achieve the level of memory-efficiency. State-of-the-art search trees make trade-offs by moving along the Pareto frontier connecting the two extremes as shown in Figure 1.1. In this dissertation, We ask the following question: *can we build search trees that are beyond the Pareto frontier in the performance-memory trade-off, that is, can we have the best of both worlds?* We answer this question in the affirmative by proposing a new recipe for constructing memory-efficient yet high-performance search trees for database applications. Figure 1.2 depicts our steps towards memory-efficiency.

Starting with a performance-optimized search tree, we first investigate techniques to compress the structure into a compact static/read-only tree (Chapters 2–4). We sacrifice temporarily the data structure's ability to efficiently support dynamic operations (e.g., inserts, updates) so that we can maximize compression. The *Dynamic-to-Static Rules* in Chapter 2 are a set of guidelines to help reduce two major sources of structural overhead in dynamic data structures: pre-allocated empty space and excessive pointers. We applied the rules to four widely-used index structures (B+tree, Masstree [124], Skip List [141], and ART [112]) and achieved 30 – 71% memory reduction depending on the workloads.

We then show that we can push the memory consumption of a search tree to the theoretical optimum without compromising its query performance. In Chapter 3, we introduce the *Fast Succinct Trie (FST)* that is only 6% larger than the minimum space required by information theory (i.e., the information-theoretic lower bound) while matching the query performance of the state-of-the-art uncompressed search trees. Compared to earlier succinct tries [3, 91], FST consumes even less space (10 bits per node) and is an order of magnitude faster. Based on FST, we build the *Succinct Range Filter (SuRF)* [169] in Chapter 4. Unlike traditional Bloom filters [62], SuRF supports approximate membership tests for both single-key and range queries. To the best of our knowledge, SuRF is the first data structure that is fast and small enough to solve the range filtering problem practically for general data processing systems. We applied SuRF to Facebook's RocksDB [37] and observed up to 5× speed up for range queries because of the unnecessary I/Os saved by the filters.

The next step in our recipe is to add support for dynamic operations back to the search trees with bounded and amortized cost in performance and space. We present a dual-stage architecture, called the *hybrid index* [168], in Chapter 5. A hybrid index is a single logical index but made of two physical search trees. The tree in the first stage ingests all incoming entries and is kept small for fast read and write operations. The index periodically migrates entries from the first stage to the second, which uses a compact and read-optimized data structure. A hybrid index guarantees memory-efficiency by storing the majority of the entries in the second (i.e., more compressed) stage. The hybrid index method completes our study on structural compression of search trees because it provides an efficient way to modify the memory-efficient but static data structures proposed in Chapters 2–4.

As the structural overhead of a search tree approaches the minimum, the keys stored in the tree becomes the dominating factor in its memory consumption. As a final step in the recipe, we present in Chapter 6 the *High-speed Order-Preserving Encoder (HOPE)* [170] that can compress arbitrary input keys effectively and efficiently while preserving their order. HOPE's approach is to identify common key patterns at a fine granularity and then exploit the entropy to achieve high compression rates with a small dictionary. HOPE includes six representative compression schemes that trade between compression rate and encoding performance, and its modularized design makes it easy to incorporate new algorithms. HOPE is an orthogonal approach that one can apply to any of the compressed search trees above to achieve additional space savings and performance gains. Our experiments show that using HOPE improves the search trees' query latency (up to 40% faster) and memory-efficiency (up to 30% smaller) simultaneously for most string key workloads, advancing the state-of-the-art Pareto frontier in the performance-memory trade-off to a new level, as shown in Figure 1.1.

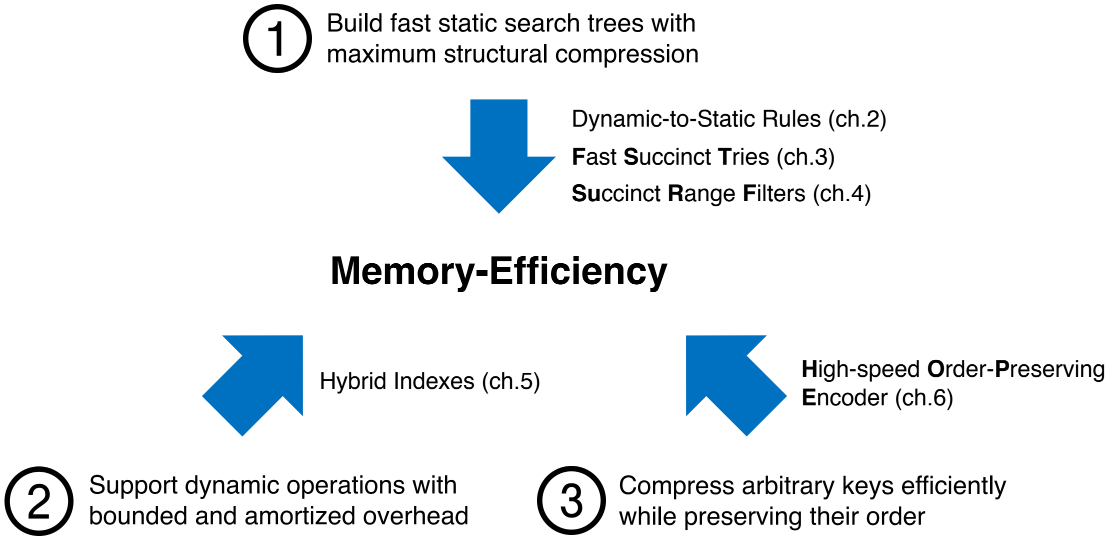

1 Build fast static search trees with maximum structural compression
Dynamic-to-Static Rules (ch.2)
Fast Succinct Tries (ch.3)
Succinct Range Filters (ch.4)
Memory-Efficiency
Hybrid Indexes (ch.5)
High-speed Order-Preserving Encoder (ch.6)
2 Support dynamic operations with bounded and amortized overhead
3 Compress arbitrary keys efficiently while preserving their order

Figure 1.2: Steps Towards Memory-Efficiency – The main contribution of this thesis is a new recipe for designing memory-efficiency yet high-performance search trees from existing solutions. The body of the thesis is organized according to these steps.

1.3 Thesis Statement and Contributions

This dissertation seeks to address the challenge of building compact yet fast in-memory search trees to allow efficient use of memory in data processing systems. We provide evidence to support the following statement:

Thesis: *Compressing in-memory search trees via efficient algorithms and careful engineering improves the performance and resource-efficiency of database management systems.*

A recurring theme in this dissertation is to make data structures in a DBMS memory-efficient without compromising (and in many cases, improving) query performance. The solutions provided in the thesis (outlined in Section 1.2) amalgamate *algorithmic innovations* that allow us to store, retrieve, and analyze data using fewer operations and resources, and system-aware *performance engineering* that allows us to exploit the underlying hardware capabilities better. We summarize the technical contributions of this thesis as follows:

- A set of guidelines (Dynamic-to-Static Rules) to help convert any dynamic search tree to a compact, immutable version (Chapter 2).
- Applications of the Dynamic-to-Static Rules to four different in-memory search trees to illustrate the effectiveness and generality of the method (Chapter 2).

- A new algorithm (LOUDS-DS) for trie representation that combines two encoding schemes to achieve high performance while remaining succinct (Chapter 3).
- A new succinct data structure (Fast Succinct Trie) that is as fast as the state-of-the-art performance-optimized search trees while being close to the minimal space defined by information theory (Chapter 3).
- The first practical and general-purpose data structure (SuRF) for range filtering, i.e., approximate membership test for ranges (Chapter 4).
- An application of SuRF to RocksDB that improves the system's range query performance by up to 5× (Chapter 4).
- A new dual-stage index architecture (Hybrid Index) that amortizes the cost of updating read-optimized data structures through ratio-bounded batching (Chapter 5).
- An application of hybrid indexes to H-Store that reduces the DBMS's index memory by up to 70% while achieving comparable query performance (Chapter 5).
- A new theoretical model to characterize the properties of dictionary encoding and to reason about order-preserving compression (Chapter 6).
- A new order-preserving key compressor (HOPE) for in-memory search trees, including six entropy encoding schemes that trade between compression rate and performance (Chapter 6).
- Applications of HOPE to five state-of-the-art search trees that achieve a Pareto improvement on performance and memory-efficiency (Chapter 6).

Chapter 2

Guidelines for Structural Compression: The Dynamic-to-Static Rules

In the first part of the thesis (Chapters 2–4), we consider the problem of building fast and memory-efficient static search trees. "Static" means that the data structure is optimized for read-only workloads. A dynamic operation (e.g., insert, update) will typically cause a significant part of the static structure to be reconstructed. The RUM conjecture in designing databases' access methods states that: "Read, update, memory – optimize two at the expense of the third" [54]. We found this conjecture applicable to data structure designs. We, therefore, sacrifice temporarily the performance of dynamic operations to achieve optimal space and read performance first. Later in Chapter 5, we study how to speed up modification queries on these static solutions with minimal overhead.

The Dynamic-to-Static Rules (D-to-S Rules) introduced in this chapter is our first attempt to structurally compress in-memory search trees. The crux of the D-to-S Rules is to exploit the fact that existing search trees such as B+trees and radix trees allocate extra space to support dynamic operations efficiently. We observe two major sources of such extra memory consumption 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 a dynamic data structure, the D-to-S Rules are:

- **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.

	Year	Supported Index Types
ALTIBASE [22]	1999	**B-tree/B+tree**, R-tree
H-Store [13]	2007	**B+tree**, hash index
HyPer [84]	2010	**Adaptive Radix Tree**, hash index
MSFT Hekaton [115]	2011	**Bw-tree**, hash index
MySQL (MEMORY) [31]	2005	**B-tree**, hash index
MemSQL [29]	2012	**skip list**, hash index
Peloton [33]	2017	**Bw-tree**
Redis [36]	2009	linked list, hash, skip list
SAP HANA [8]	2010	**B+tree/CPB+tree**
Silo [157]	2013	**Masstree**
SQLite [40]	2000	**B-tree**, R*-tree
TimesTen [42]	1995	**B-tree**, T-tree, hash index, bitmap
VoltDB [6]	2008	**Red-Black Tree**, hash index

Table 2.1: Index Types – The different types of index data structures supported by major commercial and academic in-memory OLTP DBMSs. The year corresponds to when the system was released or initially developed. The default index type for each DBMS is listed in **bold**.

- **Rule #3: Compression** – Compress parts of the data structure using a general-purpose block compression algorithm.

In the rest of this chapter, we explain each rule in detail through example applications to four different search trees. We then evaluate briefly the memory savings and performance impact of the static trees created by the rules in Section 2.5.

2.1 Example Data Structures

In order to pick the most representative search trees used in modern DBMSs, we examined twelve major commercial and academic in-memory OLTP DBMSs that were developed in the last two decades. Table 2.1 shows the index types supported by each DBMS and the year that they were released. We also include which index type is used as the default when the user does not specify any hints to the DBMS; that is, the data structure that the DBMS uses when an application invokes the CREATE INDEX command. According to the survey, we select B+tree, Masstree, Skip List, and Adaptive Radix Tree (ART) as example data structures to apply the D-to-S rules on. We briefly introduce these four search trees in this section.

B+tree: The B+tree is the most common index structure that is used in almost every OLTP DBMS [69]. It is a self-balancing search tree, usually with a large fanout. Al-

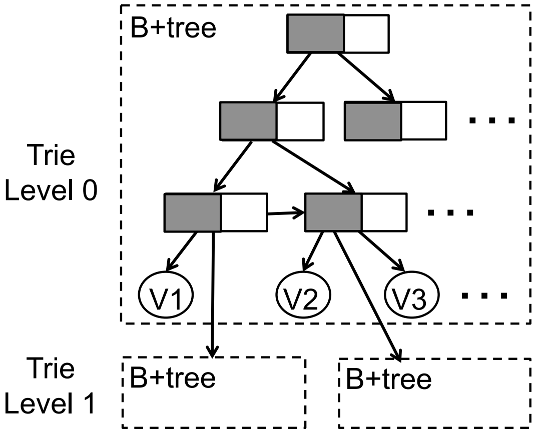


Figure 2.1: Masstree – Masstree adopts a multi-structure design, where a group of fixed-height B+trees conceptually forms a trie.

though originally designed for disk-oriented databases to minimize disk seeks, B+trees have maintained their prevalence in the main-memory DBMSs, as shown in Table 2.1. For our analysis, we use the STX B+tree [2] as the baseline implementation. We found in our experiments that a node size of 512 bytes performs best for in-memory operations.

Masstree: Masstree [124] is a high-performance key-value store that also supports range queries. It is used as index in main-memory databases such as SILO [157]. Masstree combines B+trees and tries to speed up key searches. The trie design makes the index particularly efficient in terms of both performance and space when handling keys with shared prefixes. As shown in Figure 2.1, the dashed rectangles in Masstree represent the individual B+trees that conceptually form a trie. The keys are divided into fixed-length 8-byte keyslices and are stored at each trie level. Unique key suffixes are stored separately in a structure called a *keybag*[1] Each B+tree leaf node has an associated keybag with a maximum capacity equal to the fanout of the B+tree. A value pointer in a leaf node can point to either a data record (when the corresponding keyslice is uniquely owned by a key) or a lower-level B+tree.

Skip List: The Skip List was introduced in 1990 as an alternative to balanced trees [141]. It has recently gained attention as a lock-free index for in-memory DBMSs [140]. The internal structure of the index is a linked hierarchy of subsequences that is designed to "skip" over fewer elements. The algorithms for insertion and deletion are designed to be simpler and potentially faster than equivalent operations in balanced trees. For our analysis, we use an implementation [15] of a variation of Skip List (called the *paged-deterministic Skip List* [133]) that resembles a B+tree.

Adaptive Radix Tree (ART): The Adaptive Radix Tree [112] is a fast and space-

[1] This structure is termed a *stringbag* in the Masstree implementation; we use *keybag* here for clarity.

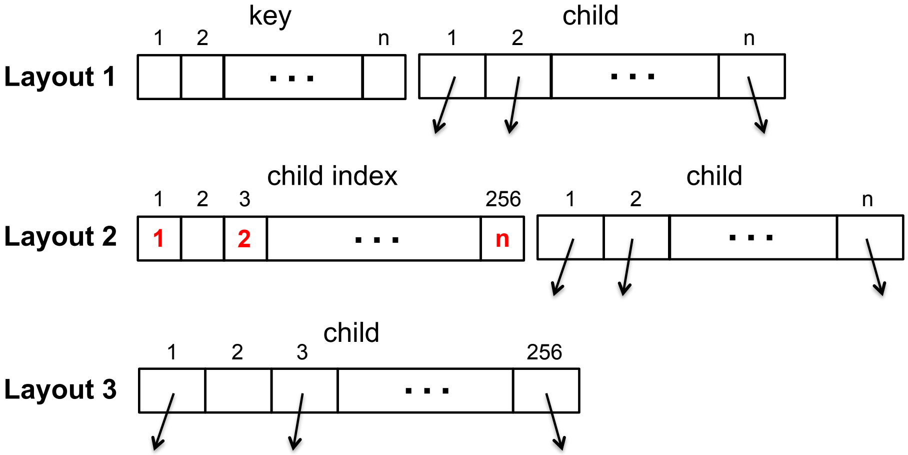


Figure 2.2: ART Node Layouts – Organization of the ART index nodes. In Layout 1, the key and child arrays have the same length and the child pointers are stored at the corresponding key positions. In Layout 2, the current key byte is used to index into the child array, which contains offsets/indexes 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 [112].

efficient data structure designed for in-memory databases. ART is a 256-way radix tree (i.e., each level represents one byte of the key). 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 (Node4, Node16, Node48, and Node256) with different layouts and capacities adaptively to achieve better memory-efficiency and better cache utilization. Figure 2.2 illustrates the three node layouts used in ART. Node4 and Node16 use the representation in Layout 1 with n=4, 16, respectively. Node48 uses Layout 2 (n=48), and Node256 uses Layout 3.

2.2 Rule #1: Compaction

The Compaction Rule seeks to generate a more efficient layout of a search tree'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 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 un-

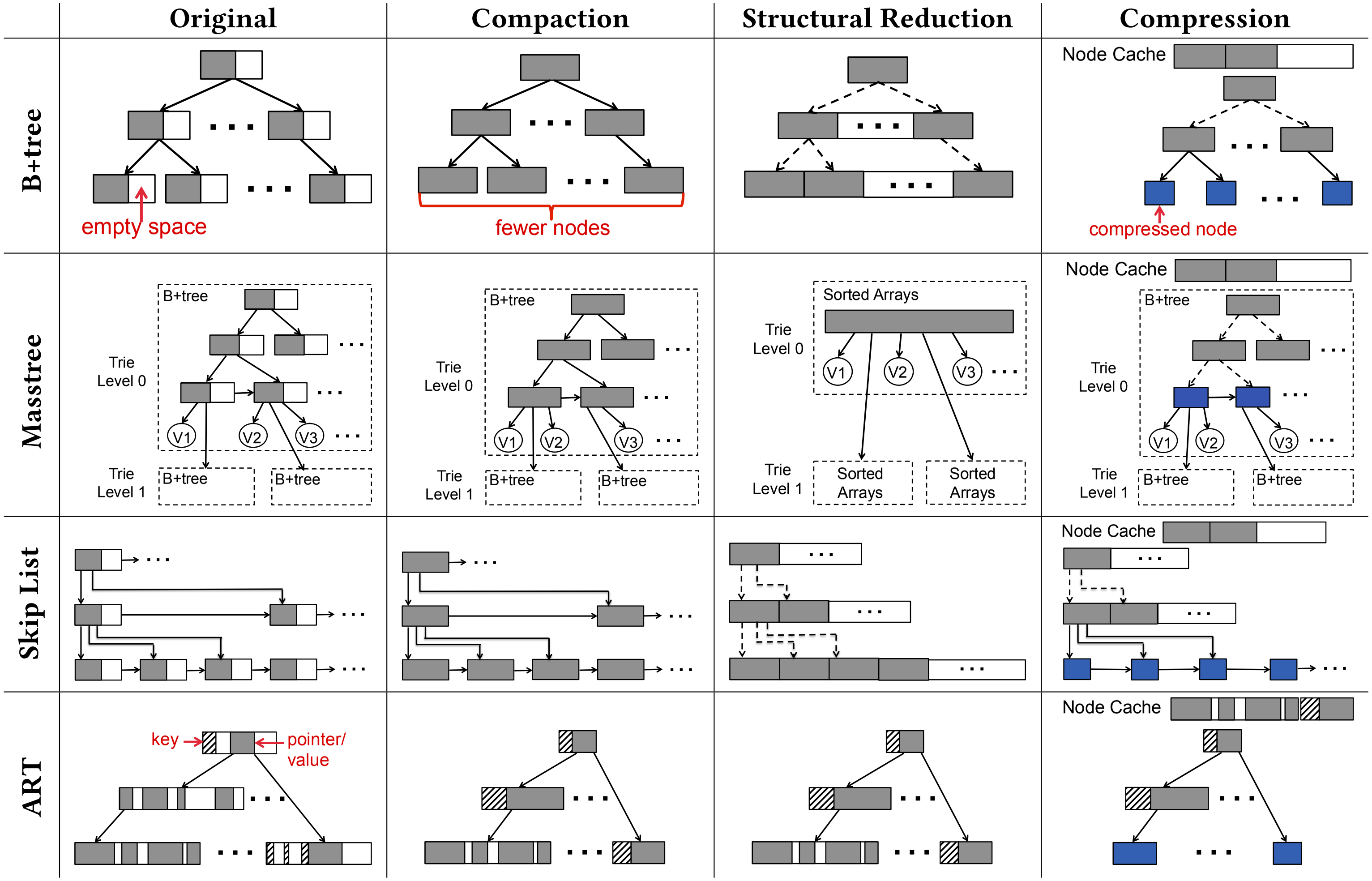


Figure 2.3: Examples of Applying the Dynamic-to-Static Rules – Solid arrows are pointers; dashed arrows indicate that the child node location is calculated rather than stored in the structure. After applying the Compaction Rule to the original dynamic data structure, we get the intermediate structure labeled "Compaction". We then applied the Structural Reduction Rule to the intermediate structure and obtain the more compact structure labeled "Reduction". The last structure in the figure shows the result of applying the Compression Rule, which is optional depending on workloads.

necessary since the data structure is static. The resulting data structure thus uses fewer memory blocks/nodes for the same entries.

As shown in Figure 2.3, a major source of memory waste in a B+tree, Masstree, Skip List, or ART is the empty space in each node. For example, the expected node occupancy of a B+tree is only 69% [164]. We observed similar occupancies in Masstree and Skip List. 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, Masstree, and Skip List, filling every node to 100% occupancy, as shown in Figure 2.3 (column "Compaction"), reduces space consumption by 31% on average without any structural changes to the search tree itself.

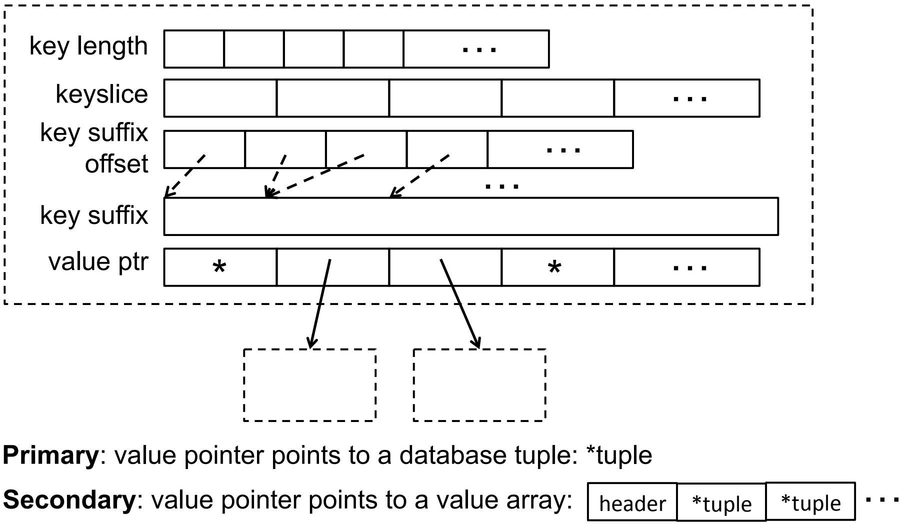
ART's prefix tree structure prevents us from filling the fixed-sized nodes to their full capacity. We instead customize the size of each node to ensure minimum slack space. This is possible because the content of each node is fixed and known when building the static structure. Specifically, let n denote 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 2.2 based on n. If $n \leq 227$, Layout 1 with array length n is used; otherwise, Layout 3 is used.

Because of the multi-structure design, compacting Masstree's memory blocks is a more complicated process: both its internal nodes and its dynamically-allocated keybags for suffixes require modification. We found that the original implementation of Masstree allocates memory for the keybags aggressively to avoid frequent resizing, which means that it wastes memory. Thus, for this rule, we instead only allocate the minimum memory space to store these suffixes.

For secondary indexes where a key can map to multiple values, the only additional change to the indexes is to remove duplicated entries by storing each key once followed by an array of its associated values.

2.3 Rule #2: Structural Reduction

The goal of the Structural Reduction 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



Primary: value pointer points to a database tuple: *tuple

Secondary: value pointer points to a value array:

Figure 2.4: Compact Masstree – The internal architecture of Masstree after applying the Compaction and Structural Reduction Rules.

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).

We applied this rule to our four indexes. The resulting data structures are shown in Figure 2.3 (column "Reduction"). We note that after the reduction, the nodes in B+tree, Masstree, and Skip List 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). 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 for this step.

There are additional opportunities for reducing the space overhead with this rule. For example, the internal nodes in the B+tree, Masstree, and Skip List 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 and Skip List. For Masstree, however, it is possible to do this without a significant performance penalty. This is because most of the trie nodes in Masstree are small and do not benefit from a B+tree structure. As a result, our compacted version of Masstree only stores the leaf nodes contiguously as an array in each trie node. To perform a look-up, it uses binary search over this array instead of a B+tree walk to find the appropriate entries. Our results show that performing a binary search is as fast as searching a B+tree in Masstree. We also note that this rule does not affect Masstree's overall trie, a distinguishing feature of Masstree compared to B+trees and Skip Lists.

We also need to deal with the Masstree keybags. In Figure 2.4, we provide a detailed structure of the compacted Masstree. We concatenate all the key suffixes within a trie node and stores them in a single byte array, along with an auxiliary offset array to mark their start locations. This reduces the structural overhead of maintaining multiple keybags for each trie node.

2.4 Rule #3: Compression

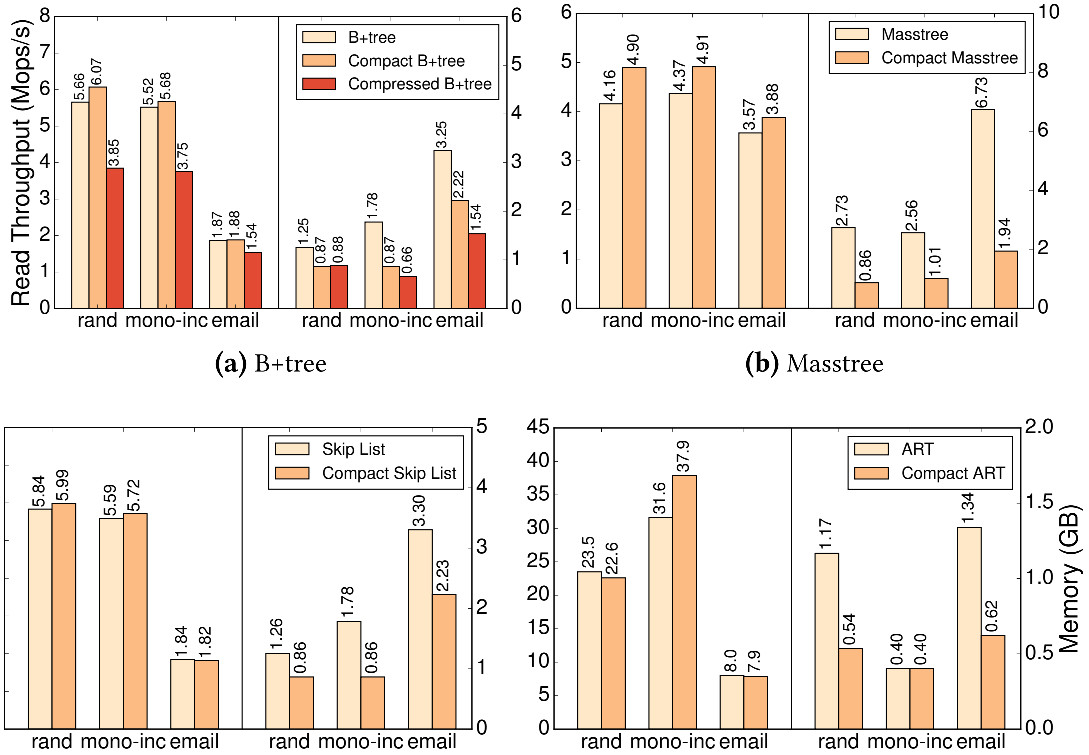
The final rule is to compress internal nodes or memory pages used in the index. For this step, we can use any general-purpose block compression algorithm. We choose the ones that are designed to have fast decompression methods in exchange for a lower compression rate, such as Snappy [39] or LZ4 [26]. Diagrams in Figure 2.3 (column "Compression") show how we apply the Compression Rule to B+tree, Masstree, Skip List, 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.

The Compression Rule is not always appropriate because of its performance overhead. Our results in Section 2.5 show that using general-purpose compression algorithms for in-memory data structures is expensive even with performance optimizations, such as the node cache. Furthermore, the compression ratio depends heavily on the workload. For many applications, the significant degradation in throughput may not justify the space savings; nevertheless, structural compression remains an option for environments with significant space constraints.

2.5 Evaluation

We compare the search trees created by the D-to-S Rules to the original data structures. *Compact* X represents the result after applying the Compaction and Structural Reduction Rules to the original structure X. while *Compressed* X means that the search tree is also compressed using Snappy [39] according to the Compression Rule. Here, X represents either B+tree, Masstree, Skip List, or ART. For the Compression Rule, we only implemented Compressed B+tree to verify that using block compression on search trees is not a desirable solution for improving the space-efficiency of main-memory OLTP databases (refer to Chapter 5 for an end-to-end system evaluation).

We run the experiments on a server equipped with 2×Intel® Xeon® E5-2680 v2 CPUs @ 2.80 GHz with 256 KB L2-cache, 26 MB L3-cache, and 4×32 GB DDR3 RAM. We used a



(a) B+tree

(b) Masstree

(c) Skip List

(d) ART

Figure 2.5: Compaction, Reduction, and Compression Evaluation – Read performance and memory 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).

set of YCSB-based microbenchmarks to mimic OLTP index workloads. The Yahoo! Cloud Serving Benchmark (YCSB) approximates typical large-scale cloud services [70]. We used its default workload **C** (*read-only*) with Zipfian distributions, which have skewed access patterns common to OLTP workloads. We tested three key types: 64-bit random integers, 64-bit monotonically increasing integers, and email addresses with an average length of 30 bytes. The random integer keys came directly from YCSB while the email keys were drawn from a large email collection. All values are 64-bit integers to represent tuple pointers.

The experiments in this section are single-threaded. We first insert 50 million entries into the search tree and then execute 10 million point queries. Throughput results in the bar charts are the number of operations divided by the execution time; memory consumption is measured at the end of each trial. All numbers reported are the average of three trials.

	Instructions	IPC	L1 Misses	L2 Misses
B+tree	4.9B	0.8	262M	160M
Masstree	5.4B	0.64	200M	174M
Skip List	4.5B	0.78	277M	164M
ART	2.1B	1.5	58M	26M

Table 2.2: Point Query Profiling – CPU-level profiling measurements for 10M point queries of random 64-bit integer keys for B+tree, Masstree, Skip List, and ART (B=billion, M=million).

As Figure 2.5 shows, the read throughput for the compact indexes is up to 20% higher in most cases 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. The only compact data structure that performs slightly worse is the Compact ART for random integer (4%) and email keys (1%). This is because unlike the other three compact indexes, Compact ART uses a slightly different organization for its internal nodes that causes a degradation in performance in exchange for a greater space saving (i.e., Layout 1 is slower than Layout 3 for look-ups – see Figure 2.2).

Figure 2.5 also shows that the compact indexes reduce the memory footprint by up to 71% (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. The Compact Masstree has the most space savings compared to the others because its internal structures (i.e., B+trees) are completely flattened into sorted arrays.

We also tested the Compression Rule on the B+tree. As shown in Figure 2.5a, 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.

We note that ART has higher point query performance than the other three index structures. To better understand this, we profiled the 10 million point queries of random 64-bit integer keys for the four original data structures using PAPI [32]. Table 2.2 shows the profiling results for total CPU instructions, instructions per cycle (IPC), L1 cache misses and L2 cache misses. We observe that ART not only requires fewer CPU instructions to perform the same load of point queries but also uses cache much more efficiently than the other three index structures. Results in other recent work [60, 160] confirm that trie-based indexes often outperform Btree-based ones for in-memory workloads. We, therefore, take a closer look at optimizing the memory-efficiency and performance of tries in the next chapter.

Chapter 3

Reducing Space to the Theoretical Limit: Fast Succinct Tries

In this chapter, we continue our investigation on compressing static search trees. We take the more performant trie indexes from the previous chapter and push its memory usage to the theoretical limit. We present the design and implementation of a new succinct data structure, called the Fast Succinct Trie (FST). FST is a space-efficient, static trie that answers point and range queries. FST consumes only 10 bits per trie node, which is close to the information-theoretic lower bound. FST is 4–15× faster than earlier succinct tries [3, 91], achieving performance comparable to or better than the state-of-the-art pointer-based indexes [2, 112, 168].

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 (i.e., **LOUDS-Dense**) in which a child node search requires only one array lookup, choosing performance over space. We encode the lower levels of the trie using the space-efficient succinct representation (i.e., **LOUDS-Sparse**) so that the overall size of the encoded trie is bounded.

Combining LOUDS-Dense and LOUDS-Sparse within the same data structure is key to achieving high performance while remaining succinct. 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). This performance improvement allows succinct tries to meet the requirements of a much wider range of real-world applications.

For the rest of the chapter, we assume that the trie maps the keys to fixed-length values. We also assume that the trie has a fanout of 256 (i.e., one byte per level).

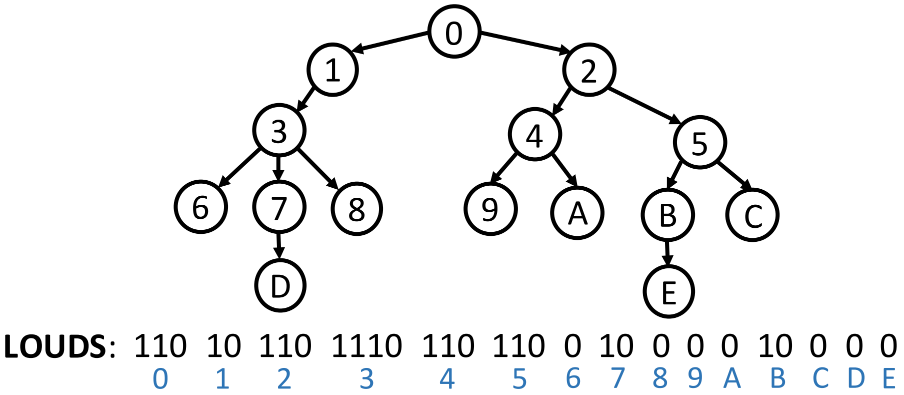

0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
LOUDS: 110 10 110 1110 110 110 0 10 0 0 0 10 0 0 0
0 1 2 3 4 5 6 7 8 9 A B C D E

Figure 3.1: Level-Ordered Unary Degree Sequence (LOUDS) – An example ordinal tree encoded using LOUDS. LOUDS traverses the nodes in a breadth-first order and encodes each node's degree using the unary code.

3.1 Background: Succinct Trees and LOUDS

A tree representation is "succinct" if the space taken by the representation is close[1] to the information-theoretic lower bound, which is the minimum number of bits needed to distinguish any object in a class. A class of size n requires at least $\log_2 n$ bits to encode each object. A trie of degree k is a rooted tree where each node can have at most k children with unique labels selected from set $\{0, 1, \ldots, k-1\}$. Since there are $\binom{kn+1}{n}/kn+1$ n-node tries of degree k, the information-theoretic lower bound is approximately $n(k\log_2 k - (k-1)\log_2(k-1))$ bits [57].

An ordinal tree is a rooted tree where each node can have an arbitrary number of children in order. Thus, succinctly encoding ordinal trees is a necessary step towards succinct tries. Jacobson [99] pioneered research on succinct tree representations and introduced the ***L**evel-**O**rdered **U**nary **D**egree **S**equence* (LOUDS) to encode an ordinal tree. As the name suggests, 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 3.1 has three children and is thus encoded as '1110'. Follow-up studies include LOUDS++ [143] which breaks the bit sequence into two parts that encode the runs of ones and zeros separately.

Navigating a tree encoded with LOUDS uses the rank & select primitives. Given a bit vector, $rank_1(i)$ counts the number of 1's up to position i ($rank_0(i)$ counts 0's), while $select_1(i)$ returns the position of the i-th 1 ($select_0(i)$ selects 0's). The original Jacobson paper showed how to support RS operations in $O(\log n)$ bit-accesses [99]. Modern

[1] There are three ways to define "close" [38]. Suppose the information-theoretic lower bound is L bits. A representation that uses L+$O(1)$, L+$o(L)$, and $O(L)$ bits is called *implicit*, *succinct*, and *compact*, respectively. All are considered succinct, in general.

rank & select implementations [87, 137, 159, 171] achieve constant time by using look-up tables (LUTs) to store a sampling of pre-computed results so that they only need to count between the samples. A state-of-the-art implementation is from Zhou et al. [171], who carefully sized the three levels of LUTs so that accessing all the LUTs incurs at most one cache miss. Their implementation adds only 3.5% space overhead to the original bit vector and is among the fastest rank & select structures available. In FST, we further optimized the rank & select structures according to the specific properties of our application to achieve better efficiency and simplicity, as described in Section 3.6.

With proper rank & select support, LOUDS performs tree navigation operations that are sufficient to implement the point and range queries required in FST in constant time. Assume that both node/child numbers and bit positions are zero-based:

- Position of the *i*-th node = $select_0(i) + 1$
- Position of the k-th child of the node started at *p* = $select_0(rank_1(p + k)) + 1$
- Position of the parent of the node started at *p* = $select_1(rank_0(p))$

3.2 LOUDS-Dense

LOUDS-Dense encodes each trie node using three bitmaps of size 256 (because the node fanout is 256) and a byte-sequence for the values as shown in the top half of Figure 3.2. The encoding follows the level order (i.e., the breadth-first order).

The first bitmap (*D-Labels*) records the branching labels for each node. Specifically, the i-th bit in the bitmap, where $0 \leq i \leq 255$, indicates whether the node has a branch with label i. For example, the root node in Figure 3.2 has three outgoing branches labeled **f**, **s**, and **t**. The *D-Labels* bitmap thus sets the 102nd (**f**), 115th (**s**) and 116th (**t**) bits and clears the rest.

The second bitmap (*D-HasChild*) indicates whether a branch points to a sub-trie or terminates (i.e., points to the value or the branch does not exist). Taking the root node in Figure 3.2 as an example, the **f** and the **t** branches continue with sub-tries while the **s** branch terminates with a value. In this case, the *D-HasChild* bitmap only sets the 102nd (**f**) and 116th (**t**) bits for the node. Note that the bits in *D-Labels* and *D-HasChild* have a one-to-one correspondence.

The third bitmap (*D-IsPrefixKey*) includes only one bit per node. The bit indicates whether the prefix that leads to the node is also a valid key. For example, in Figure 3.2, the first node at level 1 has **f** as its prefix. Meanwhile, ‘ f ’ is also a key stored in the trie. To denote this situation, the *D-IsPrefixKey* bit for this child node must be set.

The final byte-sequence (*D-Values*) stores the fixed-length values (e.g., pointers)

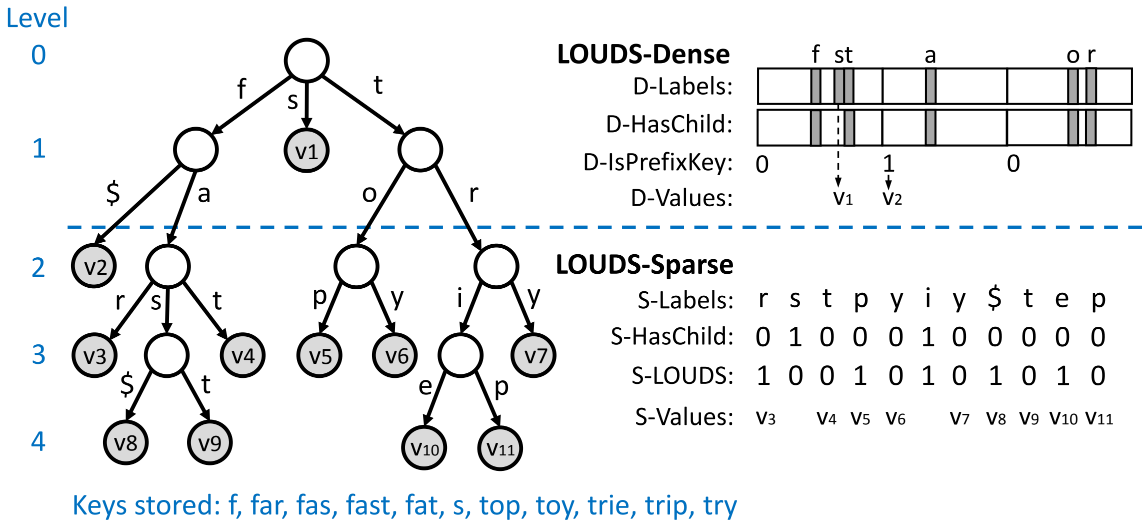

Level
0
1
2
3
4
f
s
t
v1
$
a
o
r
v2
r
s
t
p
y
i
y
v3
v4
v5
v6
v7
$
t
e
p
v8
v9
v10
v11
LOUDS-Dense
D-Labels:
D-HasChild:
D-IsPrefixKey:
D-Values:
f st
a
o r
0
1
0
v1
v2
LOUDS-Sparse
S-Labels: r s t p y i y $ t e p
S-HasChild: 0 1 0 0 0 1 0 0 0 0 0
S-LOUDS: 1 0 0 1 0 1 0 1 0 1 0
S-Values: v3 v4 v5 v6 v7 v8 v9 v10 v11
Keys stored: f, far, fas, fast, fat, s, top, toy, trie, trip, try

Figure 3.2: LOUDS-DS Encoded Trie – The upper levels of the trie are encoded using LOUDS-Dense, a bitmap-based scheme that is optimized for performance. The lower levels (which is the majority) are encoded using LOUDS-Sparse, a succinct representation that achieves near-optimal space. The \$ symbol represents the character whose ASCII number is 0xFF. It is used to indicate the situation where a prefix string leading to a node is also a valid key.

mapped by the keys. The values are concatenated in level order – same as the three bitmaps.

Tree navigation uses array lookups and rank & select operations. We denote $rank_1/select_1$ over bit sequence *bs* on position *pos* to be $rank_1/select_1$(*bs*, *pos*). Let *pos* be the current bit position in *D-Labels*. Assume that *D-HasChild*[*pos*] = 1, indicating that the branch at *pos* points to a child node (i.e., sub-trie). To move to the child node, we first compute its rank in the node list: $r = rank_1$(*D-HasChild*, *pos*). Since the child node is the rth node and each node has a fixed-size of 256 bits in *D-Labels*, the position of the child node is $256 \times r$.

To move up the trie to the parent node, we first get the rank of the current node: $r = \lfloor pos/256 \rfloor$. Since the current node is the rth node in the node list, its parent node must contain the rth set-bit in *D-HasChild*. Hence, the position of the parent node is $select_1$(*D-HasChild*, r).

If the branch at *pos* terminates (i.e., *D-HasChild*[*pos*] = 0), and we want to find out its associated value, we compute the rank of the value in *D-Values*. We first compute the total number of branches up to *pos*: $N_b = rank_1$(*D-Labels*, *pos*). Among those N_b branches, there are $N_c = rank_1$(*D-HasChild*, *pos*) non-terminating branches. Among those N_c non-terminating branches, there are $N_p = rank_1$(*D-IsPrefixKey*, $\lfloor pos/256 \rfloor$) branches who are both prefixes and valid keys (and thus have values); the rest $N_c - N_p$ branches do not have values associated. Hence, there are $N_b - (N_c - N_p)$ entries in *D-Values* up to *pos*.

To summarize:

- **D-ChildNodePos**(*pos*) = $256 \times rank_1$(*D-HasChild*, *pos*)
- **D-ParentNodePos**(*pos*) = $select_1$(*D-HasChild*, $\lfloor pos/256 \rfloor$)
- **D-ValuePos**(*pos*) = $rank_1$(*D-Labels*, *pos*) - $rank_1$(*D-HasChild*, *pos*) + $rank_1$(*D-IsPrefixKey*, $\lfloor pos/256 \rfloor$) - 1

3.3 LOUDS-Sparse

As shown in the lower half of Figure 3.2, LOUDS-Sparse encodes a trie node using four byte- or bit-sequences. The encoded nodes are then concatenated in level order.

The first byte-sequence, *S-Labels*, records all the branching labels for each trie node. As an example, the first non-value node at level 2 in Figure 3.2 has three branches. *S-Labels* includes their labels **r**, **s**, and **t** in order. We denote the case where the prefix leading to a node is also a valid key using the special byte `0xFF` at the beginning of the node (this case is handled by *D-IsPrefixKey* in LOUDS-Dense). For example, in Figure 3.2, the first non-value node at level 3 has ‘`fas`’ as its incoming prefix. Since ‘`fas`’ itself is also a stored key, the node adds `0xFF` to *S-Labels* as the first byte. Because the special byte always appears at the beginning of a node, it can be distinguished from the real `0xFF` label: if a node has a single branching label `0xFF`, it must be the real `0xFF` byte (otherwise the node will not exist in the trie); if a node has multiple branching labels, the special `0xFF` byte can only appear at the beginning while the real `0xFF` byte can only appear at the end.

The second bit-sequence (*S-HasChild*) includes one bit for each byte in *S-Labels* to indicate whether a child branch continues (i.e., points to a sub-trie) or terminates (i.e., points to a value). Taking the rightmost node at level 2 in Figure 3.2 as an example, because the branch labeled **i** points to a sub-trie, the corresponding bit in *S-HasChild* is set. The branch labeled **y**, on the other hand, points to a value, and its *S-HasChild* bit is cleared.

The third bit-sequence (*S-LOUDS*) also includes one bit for each byte in *S-Labels*. *S-LOUDS* denotes node boundaries: if a label is the first in a node, its *S-LOUDS* bit is set. Otherwise, the bit is cleared. For example, in Figure 3.2, the first non-value node at level 2 has three branches and is encoded as `100` in the *S-LOUDS* sequence. Note that the bits in *S-Labels*, *S-HasChild*, and *S-LOUDS* have a one-to-one correspondence.

The final byte-sequence (*S-Values*) is organized the same way as *D-Values* in LOUDS-Dense.

Tree navigation on LOUDS-Sparse is as follows. Given the current bit position *pos* and *S-HasChild*[*pos*] = 1, to move to the child node, we first compute the child node's rank in the level-ordered node list: $r = rank_1$(*S-HasChild*, *pos*) + 1. Because every node only has its first bit set in *S-LOUDS*, we can use $select_1$(*S-LOUDS*, r) to find the position of the rth node.

To move to the parent node, we first get the rank r of the current node by $r = rank_1$(*S-LOUDS*, *pos*) because the number of ones in *S-LOUDS* indicates the number of nodes. We then find the node that contains the $(r - 1)$th children: $select_1$(*S-HasChild*, $r - 1$).

Given *S-HasChild*[*pos*] = 0, to access the value associated with *pos*, we compute the rank of the value in *S-Values*. Because every clear-bit in *S-HasChild* has a value, there are *pos* - $rank_1$(*S-HasChild*, *pos*) values up to *pos* (non-inclusive).

To summarize:

- **S-ChildNodePos**(*nodeNum*) = $select_1$(*S-LOUDS*, $rank_1$(*S-HasChild*, *pos*) + 1)
- **S-ParentNodePos**(*pos*) = $select_1$(*S-HasChild*, $rank_1$(*S-LOUDS*, *pos*) - 1)
- **S-ValuePos**(*pos*) = *pos* - $rank_1$(*S-HasChild*, *pos*)

3.4 LOUDS-DS and Operations

LOUDS-DS is a hybrid trie in which the upper levels are encoded with LOUDS-Dense and the lower levels with LOUDS-Sparse. 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 maintain a size ratio R between LOUDS-Sparse and LOUDS-Dense to determine the dividing point among levels. Suppose the trie has H levels. Let *LOUDS-Dense-Size*(l), $0 \leq l \leq H$ denote the size of LOUDS-Dense-encoded levels up to l (non-inclusive). Let *LOUDS-Sparse-Size*(l), represent the size of LOUDS-Sparse encoded levels from l (inclusive) to H. The *cutoff* level is defined as the largest l such that *LOUDS-Dense-Size*$(l) \times R \leq$ *LOUDS-Sparse-Size*(l). Reducing R leads to more LOUDS-Dense levels, favoring performance over space. We use R=64 as the default so that LOUDS-Dense is less than 2% of the trie size but still covers the frequently-accessed top levels.

LOUDS-DS supports three basic operations efficiently:

- **ExactKeySearch**(*key*): Return the value of *key* if *key* exists (NULL otherwise).
- **LowerBound**(*key*): Return an iterator pointing to the key-value pair (k, v) where k is the smallest in lexicographical order satisfying $k \geq key$.
- **MoveToNext**(*iter*): Move the iterator to the next key-value.

Algorithm 1 LOUDS-DS Point Query

```
Variables
    $DenseHeight \leftarrow$ the height of the LOUDS-Dense encoded trie
    $DenseNodeCount \leftarrow$ total number of nodes in LOUDS-Dense levels
    $DenseChildCount \leftarrow$ total number of non-terminating branches LOUDS-Dense levels

function LOOKUP($key$)
    $level \leftarrow 0, pos \leftarrow 0$
    while $level < DenseHeight$ do                    ▷ First searching in LOUDS-Dense levels
        $nodeNum \leftarrow \lfloor pos/256 \rfloor$
        if $level \geq$ LEN($key$) then                    ▷ If run out of search key bytes
            if D-IsPrefixKey[$nodeNum$] == 1 then          ▷ If the current prefix is a key
                return D-Values[D-ValuePos($nodeNum \times 256$)]
            else
                return NULL
        $pos \leftarrow pos + key[level]$
        if D-Labels[$pos$] == 0 then                       ▷ Search failed
            return NULL
        if D-HasChild[$pos$] == 0 then                     ▷ reached leaf node
            return D-Values[D-ValuePos($pos$)]
        $pos \leftarrow$ D-ChildNodePos($pos$)          ▷ Move to child node and continue search
        $level \leftarrow level + 1$

    $pos \leftarrow$ S-ChildNodePos($nodeNum - DenseNodeCount$)          ▷ Transition to LOUDS-Sparse

    while $level <$ LEN($key$) do              ▷ Searching continues in LOUDS-Sparse levels
        if $key[level]$ does NOT exists in the label list of the current node (starting at $pos$) then
            return NULL
        if S-HasChild[$pos$] == 0 then                     ▷ reached leaf node
            return S-Values[S-ValuePos($pos$)]
        $nodeNum \leftarrow$ S-ChildNodeNum($pos$) $+DenseChildCount$
        $pos \leftarrow$ S-ChildNodePos($nodeNum - DenseNodeCount$)     ▷ Move to child node and continue search
        $level \leftarrow level + 1$

    if S-Labels[$pos$] == 0xFF and S-HasChild[$pos$] == 0 then     ▷ If the search key is a “prefix key”
        return S-Values[S-ValuePos($pos$)]
```

A point query on LOUDS-DS works by first searching the LOUDS-Dense levels. If the search does not terminate, it continues into the LOUDS-Sparse levels. The high-level searching steps at each level are similar regardless of the encoding mechanism: First, search the current node's range in the label sequence for the target key byte. If the key byte does not exist, terminate and return *NULL*. Otherwise, check the corresponding bit in the *HasChild* bit-sequence. If the bit is set (i.e., the branch is non-terminating), compute the child node's starting position in the label sequence and continue to the next level. If the *HasChild* bit is not set, return the corresponding value in the value sequence. We precompute two aggregate values based on the LOUDS-Dense levels: the node count and the number of *HasChild* bits set. Using these two values, LOUDS-Sparse can operate as if the entire trie is encoded with LOUDS-Sparse. Algorithm 1 shows the detailed steps.

LowerBound uses a high-level algorithm similar to the point query implementation. If the search byte does not exist in the label sequence of the current node, the algorithm looks for the smallest label that is greater than or equal to the search byte. If the search byte is greater than every label in the current node, the algorithm recursively moves up to the parent node and looks for the smallest label **L** that is greater than or equal to the previous search byte. Once label **L** is found, the algorithm then searches for the left-most key in the subtrie rooted at **L**.

For *MoveToNext*, the iterator starts at the current position in the label sequence and moves forward. If another valid label **L** is found within the node, the algorithm searches for the left-most key in the subtrie rooted at **L**. If the iterator hits node boundary instead, the algorithm recursively moves the iterator up to the parent node and repeat the "move-forward" process.

We include per-level cursors in the iterator to minimize the relatively expensive "move-to-parent" and "move-to-child" calls, which require rank & select operations. These cursors record a trace from root to leaf (i.e., the per-level positions in the label sequence) for the current key. Because of the level-order layout of LOUDS-DS, each level-cursor only moves sequentially without skipping items. With this property, range queries in LOUDS-DS are implemented efficiently. Each level-cursor is initialized once through a "move-to-child" call from its upper-level cursor. After that, range query operations at this level only involve cursor movement, which is cache-friendly and fast. Section 3.7.1 shows that range queries in FST are even faster than pointer-based tries.

Finally, LOUDS-DS can be built using a single scan over a sorted key-value list.

3.5 Space and Performance Analysis

Given an n-node trie, LOUDS-Sparse uses $8n$ bits for *S-Labels*, n bits for *S-HasChild*, and n bits for *S-LOUDS* – a total of $10n$ bits (plus auxiliary bits for rank & select). Referring to Section 3.1, the information-theoretic lower bound (Z) for an n-node trie of degree 256

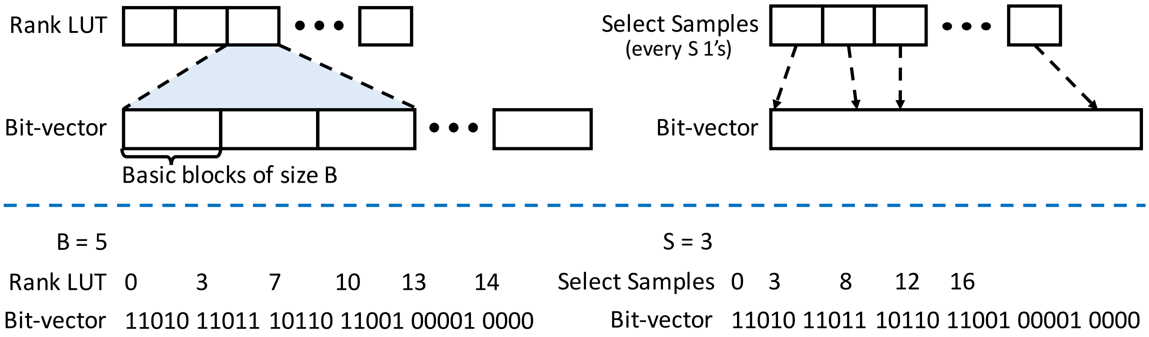

Rank LUT
Bit-vector
Basic blocks of size B
Select Samples
(every S 1's)
Bit-vector
B = 5
Rank LUT 0 3 7 10 13 14
Bit-vector 11010 11011 10110 11001 00001 0000
S = 3
Select Samples 0 3 8 12 16
Bit-vector 11010 11011 10110 11001 00001 0000

Figure 3.3: Rank and select structures in FST – Compared to a standard implementation, the customized single-level lookup table design with different sampling rates for LOUDS-Dense and LOUDS-Sparse speeds up the rank and select queries in FST.

is approximately $9.44n$ bits. Although the space taken by LOUDS-Sparse is close to the theoretical limit, technically, LOUDS-Sparse can only be categorized as *compact* rather than *succinct* in a finer-grained classification scheme because LOUDS-Sparse takes $O(Z)$ space (despite the small multiplier) instead of $Z + o(Z)$. In practice, however, FST is smaller than other succinct tries (see the evaluation in Section 3.7.2).

LOUDS-Dense's size is restricted by the ratio R to ensure that it does not affect the overall space-efficiency of LOUDS-DS. Notably, LOUDS-Dense does not always consume more space than LOUDS-Sparse: if a node's fanout is larger than 51, it takes fewer bits to represent the node using the former instead of the latter. Since such nodes are common in a trie's upper levels, adding LOUDS-Dense on top of LOUDS-Sparse often improves space-efficiency.

For point queries, searching at each LOUDS-Dense level requires two array lookups plus a rank operation on bitvector *D-HasChild*; searching at each LOUDS-Sparse level involves a label searching sub-routine plus a rank and a select operation on *S-HasChild* and *S-LOUDS*, respectively. The dominating operations are, therefore, the ranks and selects on all the bit vectors, as well as the label searching at LOUDS-Sparse levels. We next describe optimizations for these critical operations.

3.6 Optimizations

We focus on optimizing the three most critical operations: rank, select, and label search. Because all the bit-sequences in LOUDS-DS require either rank or select support, but not both, we gain the flexibility to optimize rank and select structures separately. We present a performance breakdown to show their effects in Section 3.7.3.

Rank. Figure 3.3 (left half) shows our lightweight rank structure. Instead of three levels of LUTs (look-up tables) as in Poppy [171], we include only a single level. The bit-vector is divided into fixed-length basic blocks of size B (bits). Each basic block owns a 32-bit entry in the rank LUT that stores the pre-computed rank of the start position of the block. For example, in Figure 3.3, the third entry in the rank LUT is 7, which is the total number of 1's in the first two blocks. Given a bit position i, $rank_1(i) =$ LUT[$\lfloor i/B \rfloor$] + ($popcount$ from bit ($\lfloor i/B \rfloor \times B$) to bit i), where $popcount$ is a built-in CPU instruction. For example, to compute $rank_1(12)$ in Figure 3.3, we first look up slot $\lfloor 12/5 \rfloor = 2$ in the rank LUT and get 7. We count the 1's in the remaining 3 bits (bit $\lfloor 12/5 \rfloor \times 5 = 10$ to bit $i = 12$) using the $popcount$ instruction and obtain 2. The final result is thus $7 + 2 = 9$.

We use different block sizes for LOUDS-Dense and LOUDS-Sparse. In LOUDS-Dense, we optimize for performance by setting B=64 so that at most one *popcount* is invoked in each rank query. Although such dense sampling incurs a 50% overhead for the bit-vector, it has little effect on overall space because the majority of the trie is encoded using LOUDS-Sparse, where we set B=512 so that a block fits in one cacheline. A 512-bit block requires only 6.25% additional space for the LUT while retaining high performance [171].

Select. The right half of Figure 3.3 shows our lightweight select structure. The select structure is a simple LUT (32 bits per item) that stores the precomputed answers for the sampled queries. For example, in Figure 3.3, because the sampling rate $S = 3$, the third entry in the LUT stores the position of the $3 \times 2 = 6$th (zero-based) set bit, which is 8. Given a bit position i, $select_1(i) =$ LUT[i/S] + (selecting the $(i - i/S \times S)$th set bit starting from position LUT[i/S] + 1) + 1. For example, to compute $select_1(8)$, we first look up slot $8/3 = 2$ in the LUT and get 8. We then select the $(8 - 8/3 \times 3) =$ 2nd set bit starting from position $LUT[8/3] + 1 = 9$ by binary-searching between position 9 and 12 using *popcount*. This select equals 1. The final result for $select_1(8)$ is thus $9 + 1 = 10$.

Sampling works well in our case because the only bit vector in LOUDS-DS that requires select support is *S-LOUDS*, which is quite dense (usually 17-34% of the bits are set) and has a relatively even distribution of the set bits (at least one set bit in every 256 bits). This means that the complexity of selecting the remaining bits after consulting the sampling answers is constant (i.e., needs to examine at most $256S$ bits) and is fast. The default sampling rate S is set to 64, which provides good query performance yet incurs only 9-17% space overhead locally (1-2% overall).

Label Search. Most succinct trie implementations search linearly for a label in a sequence. This is suboptimal, especially when the node fanout is large. Although a binary search improves performance, the fastest way is to use vector instructions. We use 128-bit SIMD instructions to perform the label search in LOUDS-Sparse. We first determine the node size by counting the consecutive 0's after the node's start position in the *S-LOUDS* bit-sequence. We then divide the labels within the node boundaries into 128-bit chunks, each containing 16 labels, and perform group equality checks. This search requires at most 16 SIMD equality checks using the 128 bit SIMD instructions. Our experiments in Section 3.7 show that more than 90% of the trie nodes have sizes less than eight, which means that the label search requires only a single SIMD equality check.

Prefetching. In our FST implementation, prefetching is most beneficial when invoked before switching to different bit/byte-sequences in LOUDS-DS. Because the sequences in LOUDS-DS have position correspondence, when the search position in one sequence is determined, the corresponding bits/bytes in other sequences are prefetched because they are likely to be accessed next.

3.7 Evaluation

In this section, we evaluate FST using in-memory microbenchmarks. The Yahoo! Cloud Serving Benchmark (YCSB) [70] is a workload generation tool that models large-scale cloud services. We use its default workloads C and E to generate point and range queries. We test two representative key types: 64-bit random integers generated by YCSB 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 machine on which we run the experiments has two Intel®Xeon®E5-2680v2 CPUs @ 2.80 GHz and 4×32 GB RAM. The experiments run on a single thread. We run each experiment three times and report the average result. We omit error bars because the variance is small.

We evaluate FST in three steps. First, we compare FST to three state-of-the-art pointer-based index structures. We use equi-cost curves to demonstrate FST's relative advantage in the performance-space trade-off. Second, we compare FST to two alternative succinct trie implementations. We show that FST is 4–15× faster while also using less memory. Finally, we present a performance breakdown of our optimization techniques described in Section 3.6.

We begin each experiment by bulk-loading a sorted key list into the index. The list contains 50M entries for the integer keys and 25M entries for the email keys. We report the average throughput of 10M point or range queries on the index. The YCSB default range queries are short: most queries scan 50–100 items, and the access patterns follow a Zipf distribution. The average query latency here refers to the reciprocal of throughput because our microbenchmark executes queries serially in a single thread. For all index types, the reported memory number excludes the space taken by the value pointers.

3.7.1 FST vs. Pointer-based Indexes

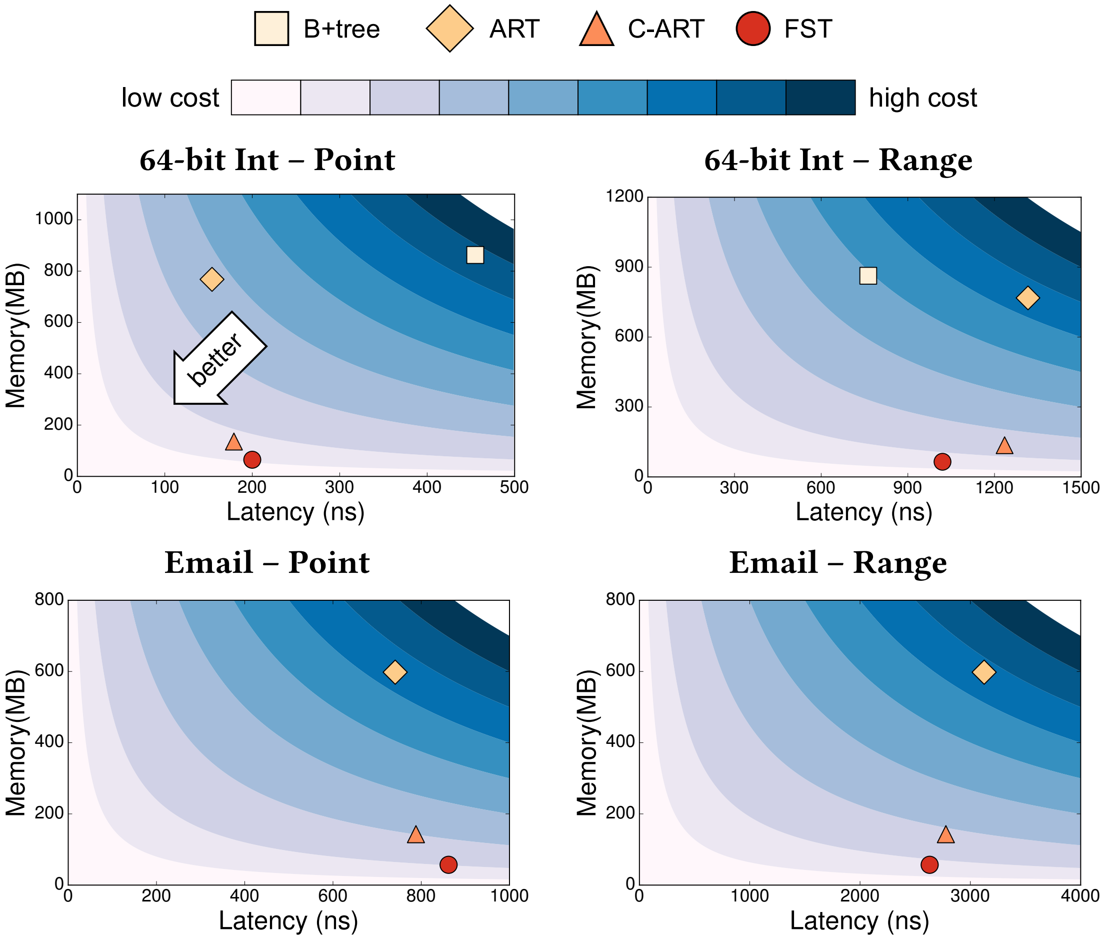


Figure 3.4: FST vs. Pointer-based Indexes – Performance and memory comparisons between FST and state-of-the-art in-memory indexes. The blue equi-cost curves indicate a balanced performance-space trade-off. Points on the same curve are considered "indifferent".

We examine the following index data structures in our testing framework:

- **B+tree**: This is the most common index structure used in database systems. We use the fast STX B+tree [2] to compare against FST. The node size is set to 512 bytes for best in-memory performance. We tested only with fixed-length keys (i.e., 64-bit integers).
- **ART**: The Adaptive Radix Tree (ART) is a state-of-the-art index structure designed for in-memory databases [112]. ART adaptively chooses from four different node layouts based on branching density to achieve better cache performance and space-efficiency.
- **C-ART**: We obtain a compact version of ART by converting a plain ART instance to

a static version according to the Compaction and Structural reduction rules discussed in Chapter 2.

We note that ART, C-ART, and FST are trie indexes and they store only unique key prefixes in this experiment.

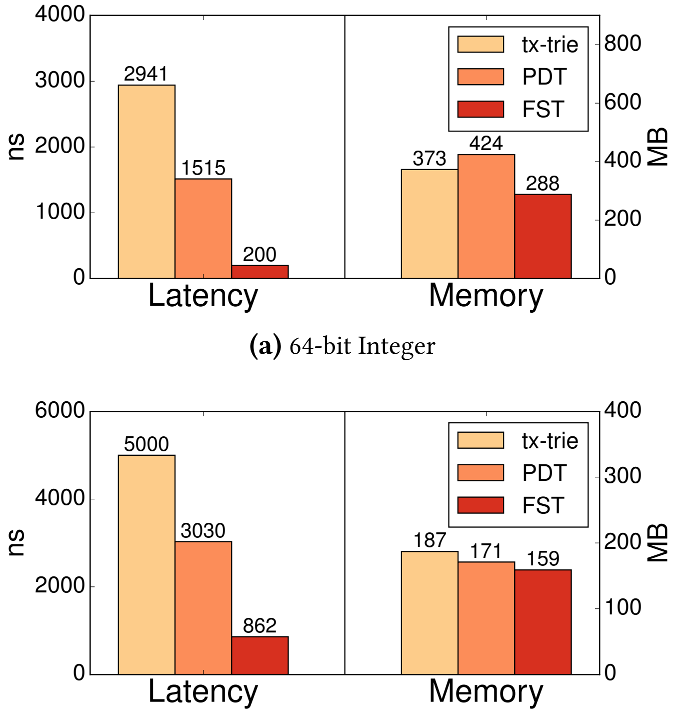
Figure 3.4 shows the comparison results. Each subfigure plots the locations of the four (three for email keys) indexes in the performance-space (latency vs. memory) map. We observe that FST is among the fastest choices in all cases while consuming less space. To better understand this trade-off, we define 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 3.4 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 the 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.

3.7.2 FST vs. Other Succinct Tries

We compare FST against the following alternatives:

- **tx-trie**: This is an open-source succinct trie implementation based on LOUDS [3]. Its design is similar to LOUDS-Sparse but without any optimizations from Section 3.6.
- **PDT**: The path-decomposed trie [91] is a state-of-the-art succinct trie implementation based on the Depth-First Unary Degree Sequence (DFUDS) [57]. PDT re-balances the trie using path-decomposition techniques to achieve latency and space reduction.

We evaluate the point query performance and memory for both integer and email key workloads. All three tries store the complete keys (i.e., including the unique suffixes). Figure 3.5 shows that FST is 6–15× faster than tx-trie, 4–8× faster than PDT, and is also smaller than both. Although tx-trie shares the LOUDS-Sparse design with FST, it is slower without the performance boost from LOUDS-Dense and other optimizations. We also notice that the performance gap between PDT and FST shrinks in the email workload because the keys have a larger variance in length and PDT's path decomposition helps rebalance the trie.



(a) 64-bit Integer

(b) Email

Figure 3.5: FST vs. Other Succinct Tries – Point query performance and memory comparisons between FST and two other state-of-the-art succinct trie implementations. All three tries store complete keys (i.e., no suffix truncation).

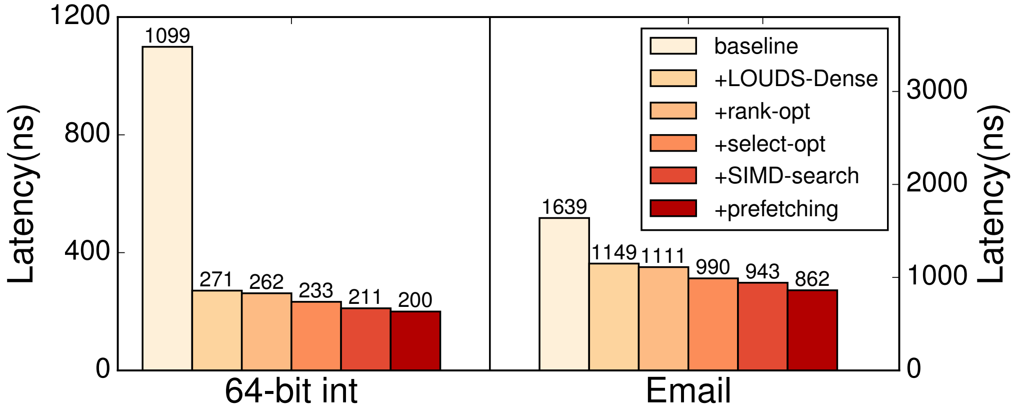
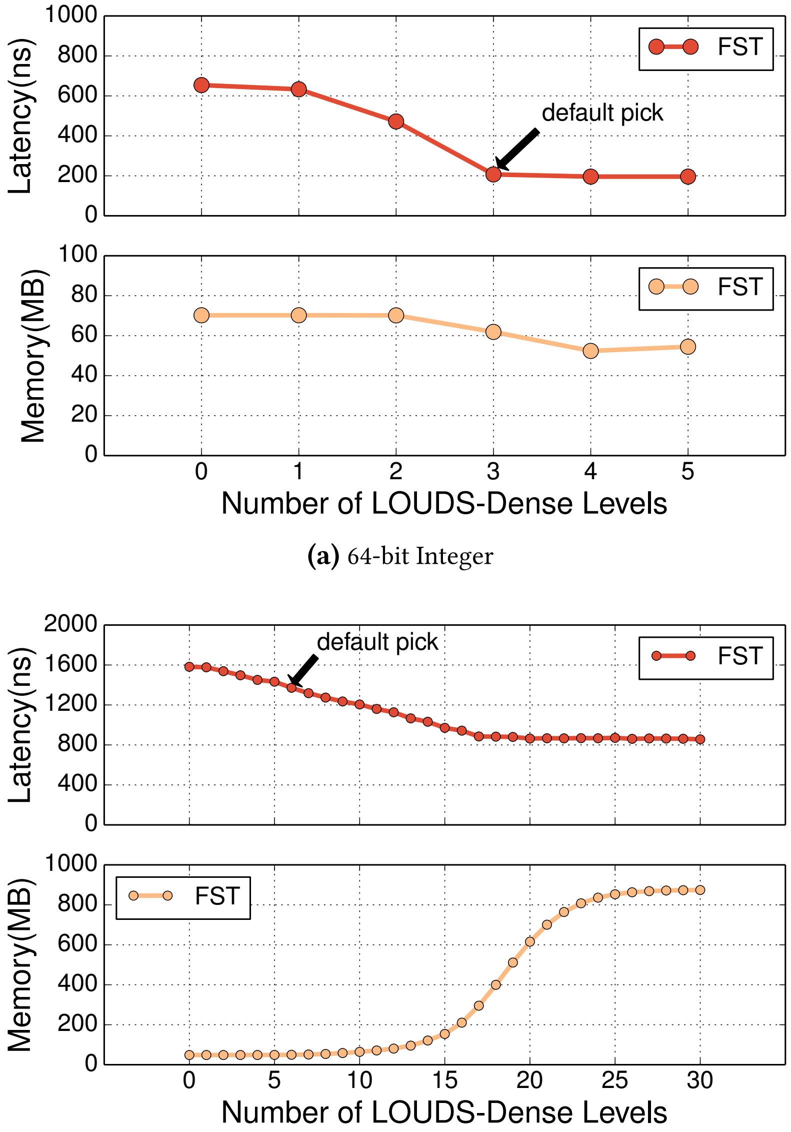


Figure 3.6: FST Performance Breakdown – An evaluation on how much LOUDS-Dense and each of the other optimizations speed up FST.



(a) 64-bit Integer

(b) Email

Figure 3.7: Trade-offs between LOUDS-Dense and LOUDS-Sparse – Performance and memory of FST as we increase the number of LOUDS-Dense levels.

3.7.3 Performance Breakdown

We then analyze these performance measurements to better understand what makes FST fast. Figure 3.6 shows a performance breakdown of point queries in both integer and email key workloads. Our baseline trie is encoded using only LOUDS-Sparse with Poppy [171] as the rank and select support. "+LOUDS-Dense" means that the upper-levels are encoded using LOUDS-Dense instead, and thus completes the LOUDS-DS design. “+rank-opt”, “+select-opt”, “+SIMD-search”, and “+prefetching” correspond to each of the optimizations described in Section 3.6.

We observe that the introduction of LOUDS-Dense to the upper-levels of FST provides a significant performance boost at a negligible space cost. The rest of the optimizations reduce the overall query latency by 3–12%.

3.7.4 Trade-offs between LOUDS-Dense and LOUDS-Sparse

We next examine the performance and memory trade-offs as we increase the number of LOUDS-Dense levels in FST (controlled by the R parameter as described in Section 3.4). Figure 3.7 shows the results for point queries in both 64-bit integer and email workloads. We observe that the query performance improves by up to 3× as we include more LOUDS-Dense levels in the trie. This is because searching in a LOUDS-Dense node requires only one bitmap lookup, which is more performant than searching in a LOUDS-Sparse node.

In terms of memory, we observe the opposite results in the two workloads. For the email workload, the memory used by FST grows as the number of LOUDS-Dense levels increases, because LOUDS-Dense sacrifices space for performance when the node fanout is low. For the integer workload, however, the LOUDS-Dense encoding is more space-efficient than the LOUDS-Sparse encoding. This is because the randomness of the integers creates trie nodes with large fanouts. As we have shown in the space analysis in Section 3.5, LOUDS-Dense takes fewer bits than LOUDS-Sparse to encode a node with a fanout greater than 51.

Although we observed a Pareto improvement on latency and memory by aggressively using LOUDS-Dense in the random integer workload, we believe that the LOUDS-Dense encoding should be restricted to the top levels in FST for other common workloads, where keys are less randomly distributed, to achieve a good performance-memory balance.

Chapter 4

Application: Succinct Range Filters

Write-optimized log-structured merge (LSM) trees [138] are popular low-level storage engines for general-purpose databases that provide fast writes [30, 151] and ingest-abundant DBMSs such as time-series databases [17, 149]. One of their main challenges for fast query processing is that items could reside in immutable files (SSTables) from all levels [5, 107]. Item retrieval in an LSM tree-based design may therefore incur multiple expensive disk I/Os [138, 151]. This challenge calls for in-memory data structures that can help locate query items.

Bloom filters [62] are a good match for this task. First, Bloom filters are fast and small enough to reside in memory. Second, Bloom filters answer approximate membership tests with "one-sided" errors—if the querying item is a member, the filter is guaranteed to return true; otherwise, the filter will likely return false, but may incur a false positive. Many LSM tree-based systems [5, 37, 149, 151], therefore, use in-memory Bloom filters to "guard" on-disk files to reduce the number of unnecessary I/Os: the system reads an on-disk file only when the corresponding Bloom filter indicates that a relevant item may exist in the file.

Although Bloom filters are useful for single-key lookups ("Is key 50 in the SSTable?"), they cannot handle range queries ("Are there keys between 40 and 60 in the SSTable?"). With only Bloom filters, an LSM tree-based storage engine still needs to read additional disk blocks for range queries. Alternatively, one could maintain an auxiliary index, such as a B+Tree, to accelerate range queries, but the memory cost is significant. To partly address the high I/O cost of range queries, LSM tree-based designs often use *prefix Bloom filters* to optimize certain fixed-prefix queries (e.g., "where email starts with com.foo@") [37, 78, 149], despite their inflexibility for more general range queries. The designers of RocksDB [37] have expressed a desire to have a more flexible data structure for this purpose [76]. A handful of approximate data structures, including the prefix Bloom filter, exist that can accelerate specific categories of range queries, but none is general purpose.

In this chapter, we present the **Succinct Range Filter** (SuRF), a fast and compact filter that provides exact-match filtering, range filtering, and approximate range counts. Like Bloom filters, SuRF guarantees one-sided errors for point and range membership tests. SuRF can trade between false positive rate and memory consumption, and this trade-off is tunable for point and range queries semi-independently.

SuRF is built upon the Fast Succinct Trie (FST) introduced in the previous chapter. The key insight in SuRF is to transform the FST into an approximate (range) membership filter by removing levels of the trie and replacing them with some number of suffix bits. The number of such bits (either from the key itself or from a hash of the key—as we discuss later in the chapter) trades space for decreased false positives.

We evaluate SuRF via microbenchmarks (Section 4.3) and as a Bloom filter replacement in RocksDB (Section 4.4). Our experiments on a 100 GB time-series dataset show that replacing the Bloom filters with SuRFs of the same filter size reduces I/O. This speeds up open-range queries (i.e., without upper-bound) by 1.5× and closed-range queries (i.e., with upper-bound) by up to 5× compared to the original implementation. For point queries, the worst-case workload is when none of the query keys exist in the dataset. In this case, RocksDB is up to 40% slower when using SuRFs because the SuRFs have higher false positive rates than the Bloom filters of the same size (0.2% vs. 0.1%). One can eliminate this performance gap by increasing the size of SuRFs by a few bits per key.

SuRF is open-sourced [20] and has been implemented/incorporated to the production systems of several major internet companies, while many more expressed interest in using SuRF because they can "see exactly where SuRF can fit in and benefit their systems," [quoted from a lead system developer from a storage company].

4.1 Design

In building SuRF using FST, our goal was to balance a low false positive rate with the memory required by the filter. The key idea is to use a truncated trie; that is, to remove lower levels of the trie and replace them with suffix bits extracted from the key (either the key itself or a hash of the key). We introduce four variations of SuRF. We describe their properties and how they guarantee one-sided errors. The current SuRF design is static, requiring a full rebuild to insert new keys. We discuss ways to handle updates in Section 4.5.

4.1.1 Basic SuRF

FST is a trie-based index structure that stores complete keys. As a filter, FST is 100% accurate; the downside, however, is that the full structure can be big. In many applications,

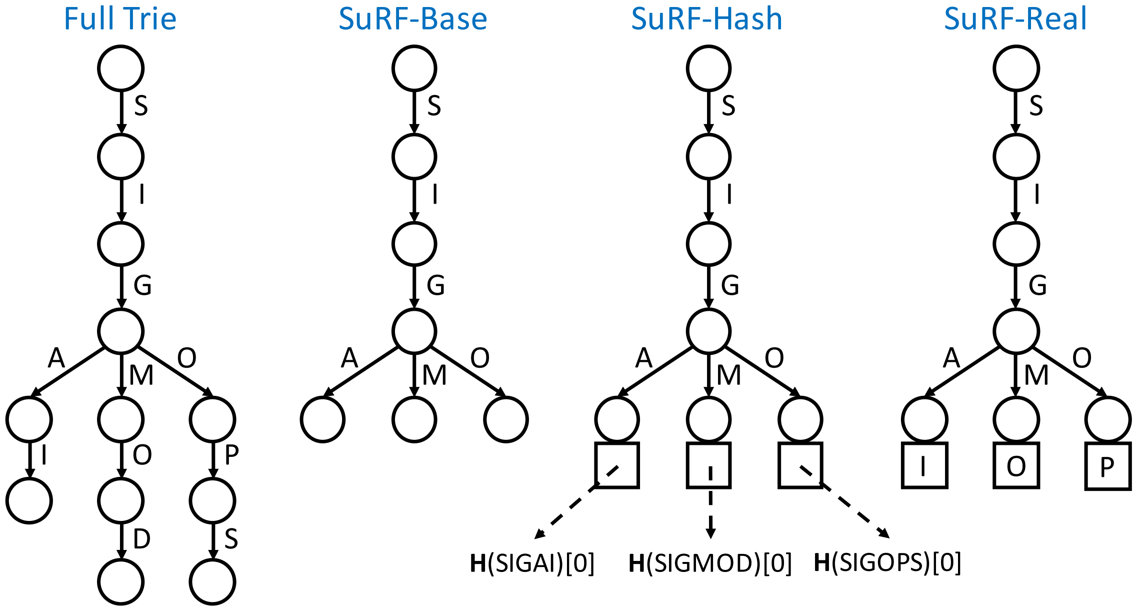


Figure 4.1: SuRF Variations – An example of deriving SuRF variations from a full trie.

filters must fit in memory to protect access to a data structure stored on slower storage. These applications cannot afford the space for complete keys, and thus must trade accuracy for space.

The basic version of SuRF (SuRF-Base) stores the minimum-length key prefixes such that it can uniquely identify each key. Specifically, SuRF-Base only stores an additional byte for each key beyond the shared prefixes. Figure 4.1 shows an example. Instead of storing the full keys (`SIGAI`, `SIGMOD`, `SIGOPS`), SuRF-Base truncates the full trie by including only the shared prefix (`SIG`) and one more byte for each key (`C`, `M`, `O`).

Pruning the trie in this way affects both filter space and accuracy. Unlike Bloom filters where the keys are hashed, the trie shape of SuRF-Base depends on the distribution of the stored keys. Hence, there is no theoretical upper-bound of the size of SuRF-Base. Empirically, however, SuRF-Base uses only 10 bits per key (BPK) for 64-bit random integers and 14 BPK for emails, as shown in Section 4.3. The intuition is that the trie built by SuRF-Base usually has an average fanout $F > 2$. When $F = 2$ (e.g., a full binary trie), there are twice as many nodes as keys. Because FST (LOUDS-Sparse to be precise) uses 10 bits to encode a trie node, the size of SuRF-Base is less than 20 BPK for $F > 2$.

Filter accuracy is measured by the false positive rate (FPR), defined as $\frac{FP}{FP+TN}$, where *FP* is the number of false positives and *TN* is the number of true negatives. 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 4.1, querying key `SIGMETRICS` will cause a false positive in SuRF-Base. FPR in SuRF-Base depends on the distributions of the stored and queried keys. Ideally, if the two distributions are independent, SuRF-Base's FPR is bounded by $N \cdot 256^{-H_{min}}$, where N is the number of stored keys and H_{min} is the minimum leaf height (i.e., the smallest depth among all the leaf nodes). To show this bound, Suppose we have a key s stored in SuRF-Base, with its leaf node L at height H (i.e., H bytes of s are stored in the trie). Given a querying key q, because we assumed that the byte distribution in q is independent of that in s, the probability that q reaches node L is 256^{-H}. Because q and s can be arbitrarily long, the probability that q and s have the same remaining suffix approaches 0. The stored key s, therefore, has a probability of 256^{-H} to lead query q to a false positive. Since there are N stored keys, the false positive rate for a query is $N \cdot 256^{-H}$. Note that this analysis assumes that the byte distributions in q and s are independent. In practice, however, query keys are almost always correlated to the stored keys. For example, if a SuRF-Base stores email addresses, query keys are likely of the same type. Our results in Section 4.3 show that SuRF-Base incurs a 4% FPR for integer keys and a 25% FPR for email keys. To improve FPR, we include three forms of key suffixes described below to allow SuRF to better distinguish between the stored key prefixes.

4.1.2 SuRF with Hashed Key Suffixes

As shown in Figure 4.1, 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). Even if the point query FPR of SuRF-Base is 100%, just 7 hash bits per key in SuRF-Hash provide a $\frac{1}{2^7} \simeq 1\%$ point query FPR. Experiments in Section 4.3.1 show that SuRF-Hash requires only 2–4 hash bits to reach 1% FPR.

The extra bits in SuRF-Hash do not help range queries because they do not provide ordering information on keys.

4.1.3 SuRF with Real Key Suffixes

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 4.1 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 (e.g., move to the next key $> K$), 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.

4.1.4 SuRF with Mixed Key Suffixes

SuRF with mixed key suffixes (SuRF-Mixed) includes a combination of hashed and real key suffix bits. The suffix bits for the same key are stored consecutively so that both suffixes can be fetched by a single memory reference. The lengths for both suffix types are configurable. SuRF-Mixed provides the full tuning spectrum (SuRF-Hash and SuRF-Real are the two extremes) for mixed point and range query workloads.

4.1.5 Operations

We summarize how SuRF's basic operations are implemented using FST. The key is to guarantee one-sided error (no false negatives).

***build**(keyList)*: Construct the filter given a list of keys. Suffix bits are stored in the FST's value vectors: *D-Values* and *S-Values*.

result = ***lookup**(k)*: Point membership test on *k*. Return true if *k* may exist (could be false positive); false guarantees non-existence. This operation first searches for k in the FST. If the search terminates without reaching a leaf, return false. If the search reaches a leaf, return true in SuRF-Base. In other SuRF variants, fetch the stored key suffix k_s of the leaf node and perform an equality check against the suffix bits extracted from k according to the suffix type as described in Sections 4.1.2–4.1.4.

iter, fp_flag = ***moveToNext**(k)*: Return an iterator pointing to the smallest key $\geq k$. Set *fp_flag* when the pointed key is a prefix of k to indicate the possibility of a false positive. This operation first performs a *LowerBound* search on the FST to reach a leaf node and get the stored key k'. If SuRF-Real or SuRF-Mixed is used, concatenate the real suffix bits to k'. It then compares k' to k. If $k' > k$, return the current iterator and set *fp_flag* to false; if k' is a prefix of k, return the current iterator and set *fp_flag* to true; if $k' < k$ and k' is not a prefix of k, advance the iterator (*iter*++) and set *fp_flag* to false.

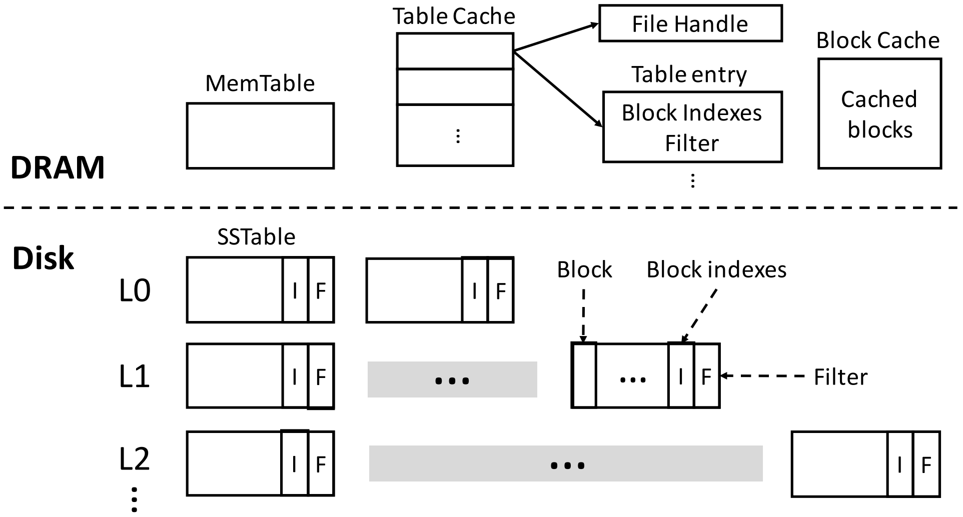


Figure 4.2: An overview of RocksDB architecture – RocksDB is implemented based on the log-structured merge tree.

result = ***lookupRange****(lowKey, highKey)*: Range membership test on (*lowKey*, *highKey*). Return true if there may exist keys within the range; false guarantees non-existence. This operation first invokes **moveToNext**(*lowKey*) and obtain an iterator. It then compares the key k pointed to by the iterator to *highKey*. If $k <$ *highKey*, return false. Otherwise, return true. A false positive could happen if k is a prefix of *highKey*.

count = ***count****(lowKey, highKey)*: Return the number of keys contained in the range (*lowKey*, *highKey*). This operation first performs **moveToNext** on both boundary keys and obtain two iterators. We extend each iterator down the trie to find the position of the smallest leaf key that is greater than the iterator key for each level, until the two iterators move to the same position or reach the maximum trie height. The operation then counts the number of leaf nodes at each level between the two iterators by computing the difference of their ranks on the FST's *D-HasChild*/*S-HasChild* bitvector. The sum of those counts is returned. False positives (over-counting) can happen at the boundaries when the first/last key included in the count is a prefix of *lowKey*/*highKey*. The count operation, therefore, can at most over-count by two.

4.2 Example Application: RocksDB

We integrated SuRF with RocksDB as a replacement for its Bloom filter. Figure 4.2 illustrates RocksDB's log-structured merge tree architecture. Incoming writes go into the MemTable and are appended to a log file (omitted) for persistence. When the MemTable is full (e.g., exceeds 4 MB), the engine sorts it and then converts it to an SSTable that becomes part of level 0. An SSTable contains sorted key-value pairs and is divided into fixed-length blocks matching the smallest disk access units. To locate blocks, RocksDB stores the "restarting point" (a string that is $\geq$ the last key in the current block and $<$ the first key in the next block) for each block as a fence index.

When the size of a level hits a threshold, RocksDB selects an SSTable at this level and merges it into the next-level SSTables that have overlapping key ranges. This process is called compaction. Except for level 0, all SSTables at the same level have disjoint key ranges. In other words, the keys are globally sorted for each level ≥ 1. Combined with a global table cache, this property ensures that an entry lookup reads at most one SSTable per level for levels ≥ 1.

To facilitate searching and to reduce I/Os, RocksDB includes two types of buffer caches: the table cache and the block cache. The table cache contains meta-data about opened SSTables while the block cache contains recently accessed SSTable blocks. Blocks are also cached implicitly by the OS page cache. When compression is turned on, the OS page cache contains compressed blocks, while the block cache always stores uncompressed blocks.

We modified RocksDB's point (*Get*) and range (*Seek*, *Next*) query implementations to use SuRF. SuRF also supports functionality beyond filtering. We implemented a new approximate *Count* query that returns the number of entries in a key range. We note that the query may over-count the deletion and modification entries in an LSM-tree, because it cannot distinguish update/delete records from insert records.

Figure 4.3 shows the execution paths for *Get*, *Seek*, and *Count* queries in RocksDB. *Next*'s core algorithm is similar to *Seek*. We use colors to highlight the potential I/O reduction by using filters. Operations in blue boxes can trigger I/O if the requesting block(s) are not cached. Filter operations are in red boxes. If the box is dashed, checks (by fetching the actual keys from SSTables) for boundary keys might be necessary due to false positives.

For *Get(key)*, SuRF is used exactly like the Bloom filter. Specifically, RocksDB searches level by level. 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, if a filter is available, 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 examines each candidate SSTable and fetches the block con-

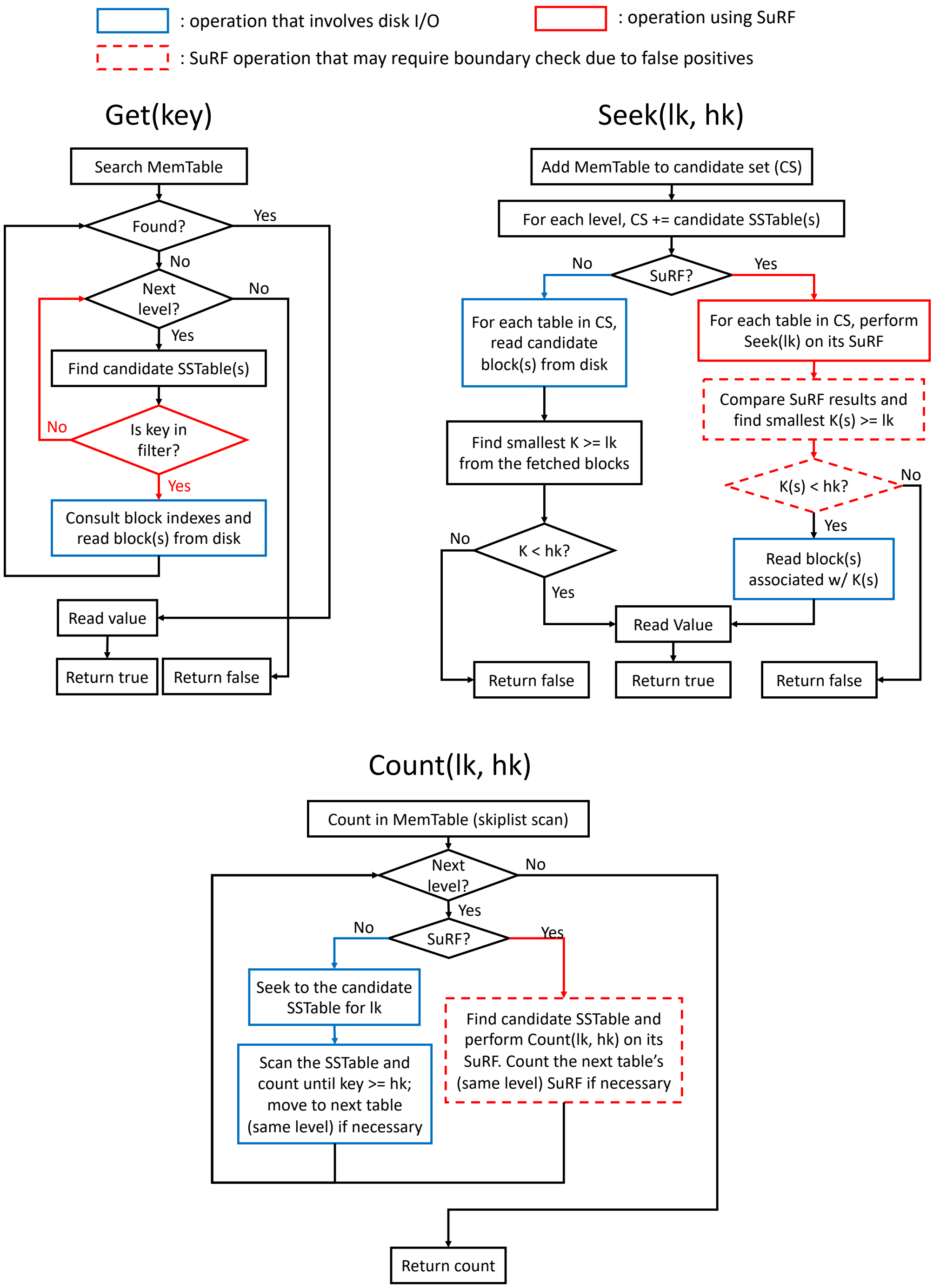

: operation that involves disk I/O
: operation using SuRF
: SuRF operation that may require boundary check due to false positives
Get(key)
Search MemTable
Found?
Yes
No
Next level?
No
Yes
Find candidate SSTable(s)
No
Is key in filter?
Yes
Consult block indexes and read block(s) from disk
Read value
Return true
Return false
Seek(lk, hk)
Add MemTable to candidate set (CS)
For each level, CS += candidate SSTable(s)
No
SuRF?
Yes
For each table in CS, read candidate block(s) from disk
For each table in CS, perform Seek(lk) on its SuRF
Compare SuRF results and find smallest K(s) >= lk
Find smallest K >= lk from the fetched blocks
K(s) < hk?
No
Yes
No
K < hk?
Yes
Read block(s) associated w/ K(s)
Read Value
Return false
Return true
Return false
Count(lk, hk)
Count in MemTable (skiplist scan)
Next level?
No
Yes
No
SuRF?
Yes
Seek to the candidate SSTable for lk
Find candidate SSTable and perform Count(lk, hk) on its SuRF. Count the next table's (same level) SuRF if necessary
Scan the SSTable and count until key >= hk; move to next table (same level) if necessary
Return count

Figure 4.3: RocksDB Query Execution Flowcharts – Execution paths for Get, Seek, and Count queries in RocksDB.

taining the smallest key that is $\geq$ *lk*. RocksDB then compares the candidate keys and finds the global smallest key $K \geq$ *lk*. For an Open Seek, the query succeeds and returns the iterators (at least one per level). For a Closed Seek, however, RocksDB performs an extra check against the *hk*: if $K \leq hk$, the query succeeds; otherwise the query returns an invalid iterator.

With SuRFs, however, instead of fetching the actual blocks, RocksDB can obtain the candidate key for each SSTable by performing a *moveToNext(lk)* operation on the SSTable's SuRF to avoid the I/O. If the query succeeds (i.e., Open Seek or $K \leq hk$), RocksDB fetches exactly one block from the selected SSTable that contains the global minimum K. If the query fails (i.e., $K > hk$), no I/O is involved. Because SuRF's *moveToNext* operation returns only a key prefix K_p, three additional checks are required to guarantee correctness. First, if the *moveToNext* operation sets the *fp_flag* (refer to Section 4.1.5), RocksDB must fetch the complete key K from the SSTable block to determine whether $K \geq lk$. Second, if K_p is a prefix of *hk*, the complete key K is also needed to verify $K \leq hk$. Third, multiple key prefixes could tie for the smallest. In this case, RocksDB must fetch their corresponding complete keys from the SSTable blocks to find the globally smallest. Despite the three potential additional checks, using SuRF in RocksDB reduces the average I/Os per *Seek(lk, hk)* query, as shown in Section 4.4.

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 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 probabilities. The number of I/Os is likely to drop from three to one.

Next(hk) is similar to *Seek(lk, hk)*, but the iterator at each level is already initialized. RocksDB increments the iterator (at some level) pointing to the current key, and then repeat the “find the global smallest” algorithm as in *Seek*.

For *Count(lk, hk)*, RocksDB first performs a *Seek* on *lk* to initialize the iterators and then counts the number of items between *lk* and *hk* at each level. Without SuRF, the DBMS computes the count by scanning the blocks in SSTable(s) until the key exceeds the upper bound. If SuRFs are available, the counting is performed in memory by calling SuRF's *count* operation. As in *Seek*, similar boundary key checks are required to avoid the off-by-one error. Instead of scanning disk blocks, *Count* using SuRFs requires at most two disk I/Os (one possible I/O for each boundary) per level. The cumulative count is then returned.

4.3 Microbenchmarks

In this section, we first evaluate SuRF using in-memory microbenchmarks to provide a comprehensive understanding of the filter's strengths and weaknesses. Section 4.4 creates an example application scenario and evaluates SuRF in RocksDB with end-to-end system measurements.

We use the YCSB [70] default workloads C and E to generate point and range queries. We test two representative key types: 64-bit random integers generated by YCSB 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 machine on which we run the experiments has two Intel®Xeon®E5-2680v2 CPUs @ 2.80 GHz with each having 10 physical cores and 20 hardware threads (with hyper-threading enabled), and 4×32 GB RAM. We run each experiment three times and report the average result.

The three most important metrics with which to evaluate SuRF are false positive rate (FPR), performance, and space. The datasets are 100M 64-bit random integer keys and 25M email keys. In the experiments, we first construct the filter under test using half of the dataset selected at random. We then execute 10M point, range, mixed (50% point and 50% range, interleaved), or count queries on the filter. The querying keys (K) are drawn from the *entire* dataset according to YCSB workload C so that roughly 50% of the queries return false. We tested two query access patterns: uniform and Zipf distribution. We show only the Zipf distribution results because the observations from both patterns are similar. For 64-bit random integer keys, the range query is $[K + 2^{37}, K + 2^{38}]$ where 46% of the queries return true. For email keys, the range query is $[K, K$(with last byte ++)$]$ (e.g., [org.acm@sigmod, org.acm@sigmoe]) where 52% of the queries return true. For count queries, we draw the lower and upper bounds from the dataset randomly so that most of them count long ranges. We use the Bloom filter implementation from RocksDB[1].

4.3.1 False Positive Rate

We first study SuRF's false positive rate (FPR). FPR is the ratio of false positives to the sum of false positives and true negatives. Figure 4.4 shows the FPR comparison between SuRF variants and the Bloom filter by varying the size of the filters. The Bloom filter only appears in point queries. Note that SuRF-Base consumes 14 (instead of 10) bits per key for the email key workloads. This is because email keys share longer prefixes, which increases the number of internal nodes in SuRF (Recall that a SuRF node is encoded using 10 bits). SuRF-Mixed is configured to have an equal number of hashed and real suffix bits.

[1] Because RocksDB's Bloom filter is not designed to hold millions of items, we replaced its 32-bit Murmur hash algorithm with a 64-bit Murmur hash; without this change, the false positive rate is worse than the theoretical expectation.

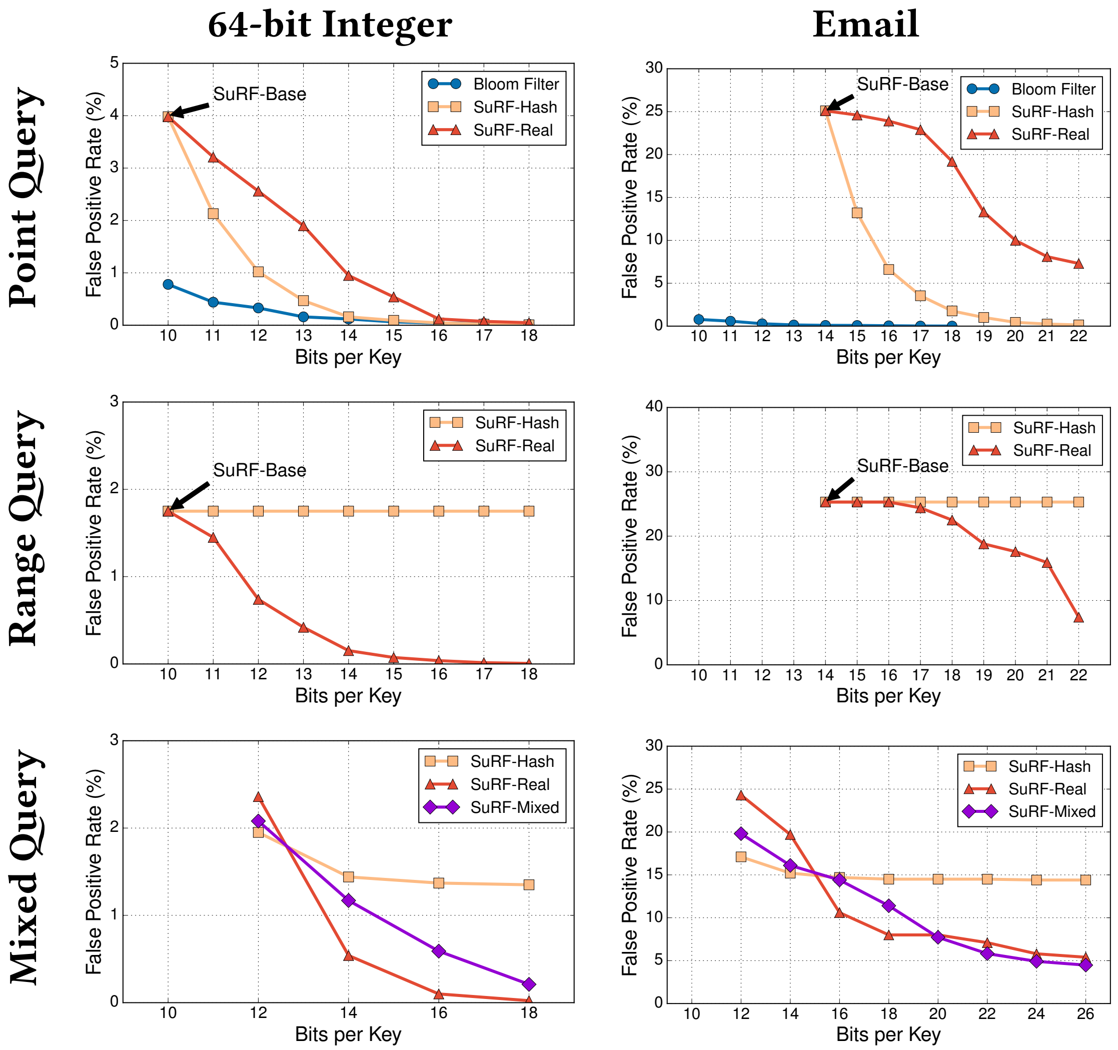


Figure 4.4: SuRF False Positive Rate – False positive rate comparison between SuRF variants and the Bloom filter (lower is better).

For point queries, the Bloom filter has lower FPR than the same-sized SuRF variants in most cases, although SuRF-Hash catches up quickly as the number of bits per key increases because every hash bit added cuts the FPR by half. Real suffix bits in SuRF-Real are generally less effective than hash bits for point queries. For range queries, only SuRF-Real benefits from increasing filter size because the hash suffixes in SuRF-Hash do not provide ordering information. The shape of the SuRF-Real curves in the email key workloads (i.e., the latter 4 suffix bits are more effective in recognizing false positives than the earlier 4) is because of ASCII encoding of characters.

For mixed queries, increasing the number of suffix bits in SuRF-Hash yields dimin-

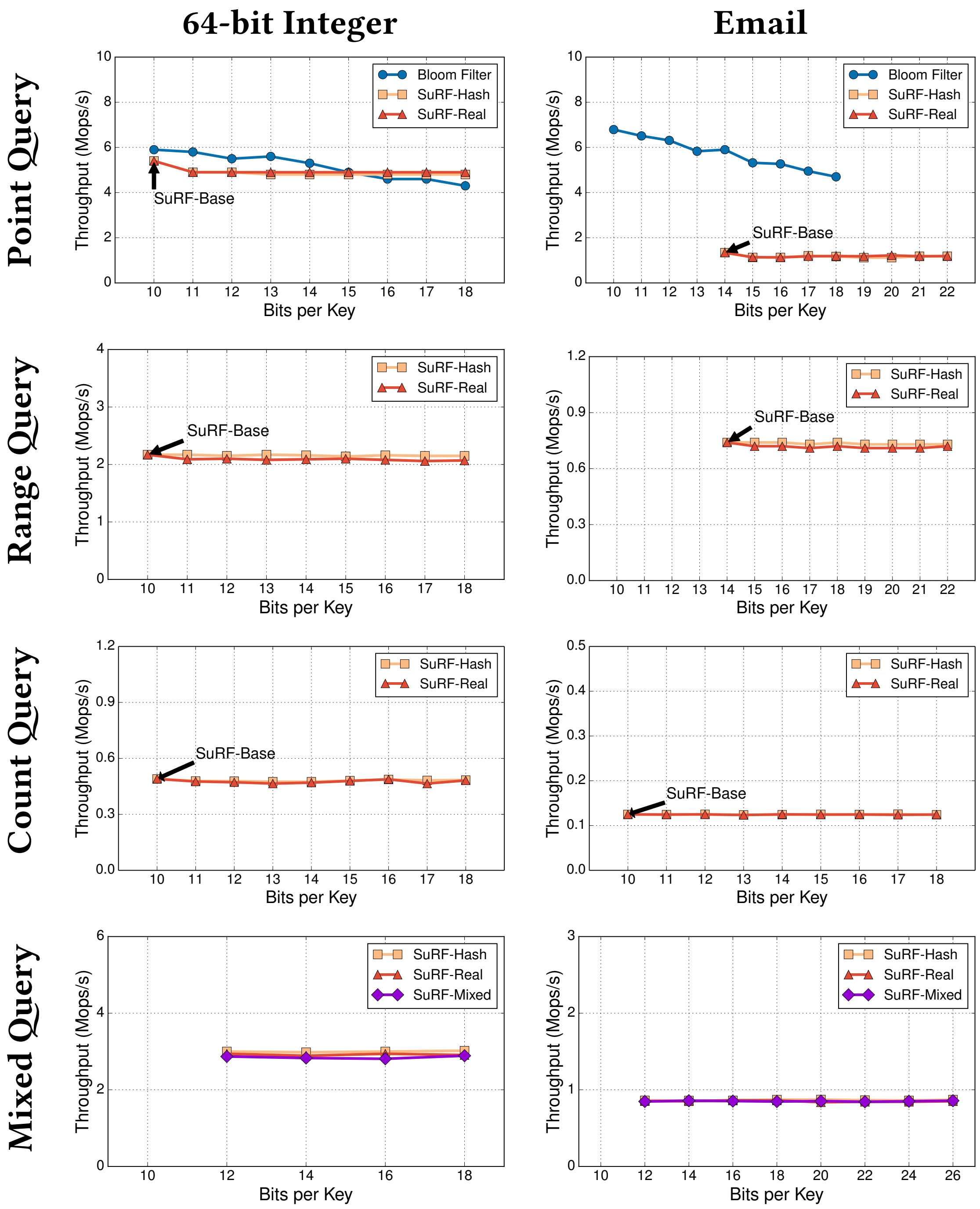


Figure 4.5: SuRF Performance – Performance comparison between SuRF variants and the Bloom filter (higher is better).

ishing returns in FPR because they do not help the range queries. SuRF-Mixed (with an equal number of hashed and real suffix bits) can improve FPR over SuRF-Real for some suffix length configurations. In fact, SuRF-Real is one extreme in SuRF-Mixed's tuning spectrum. This shows that tuning the ratio between the length of the hash suffix and that of the real suffix can improve SuRF's FPR in mixed point and range query workloads.

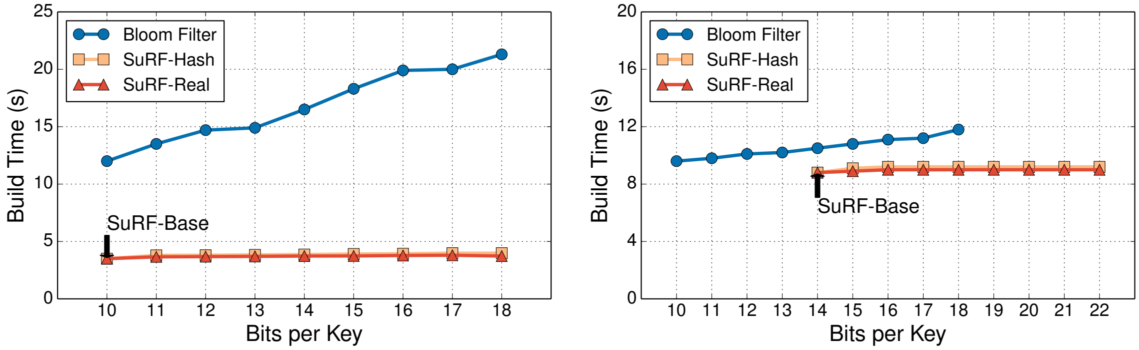
We also observe that SuRF variants have higher FPRs for the email key workloads. This is because the email keys in the data set are very similar (i.e., the key distribution is dense). Two email keys often differ by the last byte, or one may be a prefix of the other. If one of the keys is represented in the filter and the other key is not, querying the missing key on SuRF-Base is likely to produce false positives. The high FPR for SuRF-Base is significantly lowered by adding suffix bits, as shown in the figures.

4.3.2 Performance

Figure 4.5 shows the throughput comparison. The SuRF variants operate at a speed comparable to the Bloom filter for the 64-bit integer key workloads, thanks to the LOUDS-DS design and other performance optimizations mentioned in Section 3.6. For email keys, the SuRF variants are slower than the Bloom filter because of the overhead of searching/traversing the long prefixes in the trie. The Bloom filter's throughput decreases as the number of bits per key gets larger because larger Bloom filters require more hash probes. The throughput of the SuRF variants does not suffer from increasing the number of suffix bits because as long as the suffix length is less than 64 bits, checking with the suffix bits only involves one memory access and one integer comparison. The (slight) performance drop in the figures when adding the first suffix bit (i.e., from 10 to 11 for integer keys, and from 14 to 15 for email keys) demonstrates the overhead of the extra memory access to fetch the suffix bits.

Range queries in SuRF are slower than point queries because every query needs to reach a leaf node (no early exit). Count queries are also slower because such a query requires managing iterators at both ends and counting the leaf nodes between them at each trie level. Nevertheless, count queries in SuRF are much faster than those in previous trie implementations where they count by advancing the iterator one entry at a time.

Some high-level takeaways from the experiments: (1) SuRF can perform range filtering while the Bloom filter cannot; (2) If the target application only needs point query filtering with moderate FPR requirements, the Bloom filter is usually a better choice than SuRF; (3) For point queries, SuRF-Hash can provide similar theoretical guarantees on FPR as the Bloom filter, while the FPR for SuRF-Real depends on the key distribution; (4) To tune SuRF-Mixed for mixed point and range queries, one should start from SuRF-Real because real suffix bits benefit both query types and then gradually replace them with hash suffixes until the FPR is optimal.



(a) 64-bit Integer

(b) Email

Figure 4.6: SuRF Build Time – Build time comparison between SuRF variants and the Bloom filter (lower is better).

4.3.3 Build Time

We also measure the construction time of each filter in the above experiments. Recall that a filter stores half of the corresponding dataset (i.e., 50M 64-bit integer keys or 12.5M email keys) where the keys are sorted. As shown in Figure 4.6, building a SuRF is faster than building a Bloom filter. This is because a SuRF can be built in a single scan of the sorted input keys and it only involves sequential memory accesses during construction. Building a Bloom filter, however, requires multiple random writes per key. Therefore, building a SuRF has better cache performance. We also note that Bloom filters take longer to build as the number of bits per key increases because larger Bloom filters require more hash probes (and thus more random memory accesses). On the other hand, the number of suffix bits in SuRF affects little on the build time because extracting the suffix bits from a key only involves a memory read that is very likely a cache hit.

4.3.4 Scalability

In this experiment, we verify that SuRFs are scalable on multi-core systems. We repeat the SuRF experiments above by varying the number of threads. Figure 4.7 shows the aggregate point query throughput for 64-bit integer keys as the number of threads increases. We omit other scalability graphs because they show similar results. As shown in Figure 4.7, SuRF scales almost perfectly when disabling hyper-threading (only a bit off due to cache contention). Even with hyper-threading, SuRF's throughput keeps increasing without any performance collapse. This result is expected because SuRF is a read-only data structure and is completely lock-free, experiencing little contention with many concurrent threads.

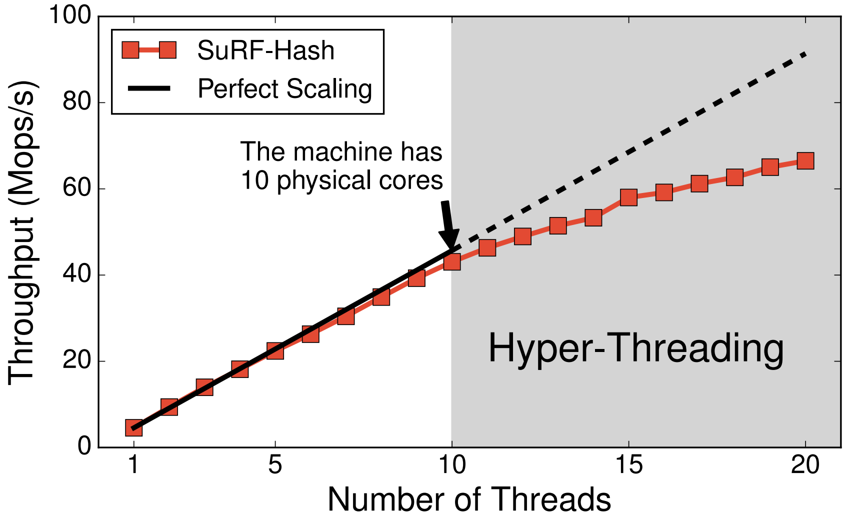


Figure 4.7: SuRF Scalability – Point query performance as the number of threads increases.

4.3.5 Comparing ARF and SuRF

The Adaptive Range filter (ARF) [49] introduced as part of Project Siberia [79] in Hekaton [74] is the state-of-the-art range filter. An ARF is a simple binary tree that covers the entire key space (e.g., for 64-bit integer keys, the root node represents range [0, 2^{64}-1] and its children represent [0, 2^{63}-1] and [2^{63}, 2^{64}-1]). Each leaf node indicates whether there may be any key or absolutely no key in its range. Using an ARF involves three steps: building a perfect trie, training with sample queries to determine which nodes to include in an ARF, and then encoding the trained ARF into a bit sequence in breadth-first order that is static.

In this experiment, we compare SuRF against ARF. We integrate the ARF implementation published by the paper authors [11] into our test framework. We set the space limit to 7 MB for ARF and use a 4-bit real suffix for SuRF so that both filters consume 14 bits per key. We use the same YCSB-based range query workloads described at the beginning of this section (Section 4.3). However, we scale down the dataset by 10× because ARF requires a large amount of memory for training. Specifically, the dataset contains 10M 64-bit integer keys (ARF can only support fixed-length keys up to 64 bits). We randomly select 5M keys from the dataset and insert them into the filter. The workload includes 10M Zipf-distributed range queries whose range size is 2^{40}, which makes roughly 50% of the queries return false. For ARF, we use 20% (i.e., 2M) of the queries for training and the rest for evaluation.

Table 4.1 compares the performance and resource use of ARF and SuRF. For query processing, SuRF is 20× faster and 12× more accurate than ARF, even though their final

	ARF	SuRF	Improvement
Bits per Key (held constant)	14	14	-
Range Query Throughput (Mops/s)	0.16	3.3	20×
False Positive Rate (%)	25.7	2.2	12×
Build Time (s)	118	1.2	98×
Build Mem (GB)	26	0.02	1300×
Training Time (s)	117	N/A	N/A
Training Throughput (Mops/s)	0.02	N/A	N/A

Table 4.1: SuRF vs. ARF – Experimental comparison between ARF and SuRF.

filter size is the same. Moreover, ARF demands a large amount of resources for building and training: its peak memory use is 26 GB and the building + training time is around 4 minutes, even though the final filter size is only 7 MB. In contrast, building SuRF only uses 0.02 GB of memory and finishes in 1.2 seconds. SuRF outperformed ARF mainly because ARF is not designed as a general-purpose range filter, but with specific application and scalability goals. We discuss the detailed reasons in Chapter 7.

The next section shows the evaluation of SuRF in the context of an end-to-end real-world application (i.e., RocksDB), where SuRF speeds up both point and range queries by saving I/Os.

4.4 System Evaluation

Time-series databases often use RocksDB or similar LSM-tree designs for the storage engine. Examples are InfluxDB [17], QuasarDB[34], LittleTable [149] and Cassandra-based systems [9, 107]. We thus create a synthetic RocksDB benchmark to model a time-series dataset generated from distributed sensors and use this for end-to-end performance measurements. We simulated 2K sensors to record events. The key for each event is a 128-bit value comprised of a 64-bit timestamp followed by a 64-bit sensor ID. The associated value in the record is 1 KB long. The occurrence of each event detected by each sensor follows a Poisson distribution with an expected frequency of one every 0.2 seconds. Each sensor operates for 10K seconds and records ~50K events. The starting timestamp for each sensor is randomly generated within the first 0.2 seconds. The total size of the raw records is approximately 100 GB.

Our testing framework supports the following database queries:

- **Point Query**: Given a timestamp and a sensor ID, return the record if there is an event.
- **Open-Seek Query**: Given a starting timestamp, return an iterator pointing to the

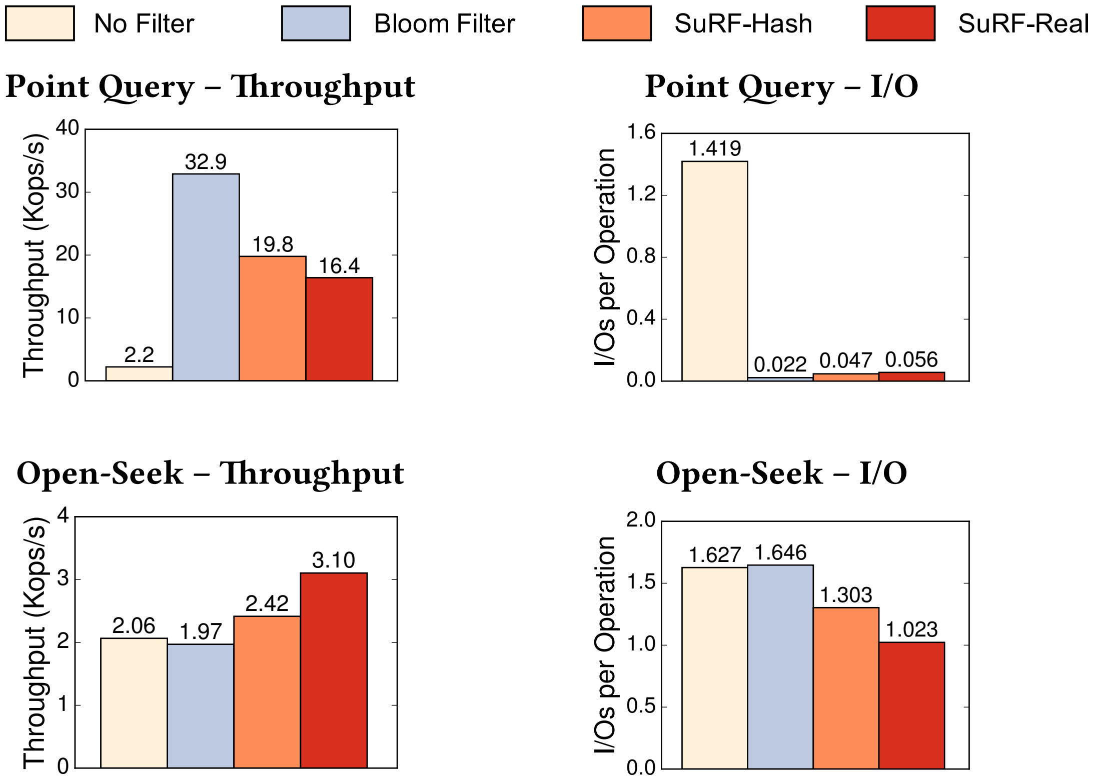


Figure 4.8: Point and Open-Seek Queries – RocksDB point query and Open-Seek query evaluation under different filter configurations.

earliest event after that time.

- **Closed-Seek Query**: Given a time range, determine whether any events happened during that time period. If yes, return an iterator pointing to the earliest event in the range.

Our test machine has an Intel®Core™i7-6770HQ CPU, 32 GB RAM, and an Intel®540s 480 GB SSD. We use Snappy (RocksDB's default) for data compression. The resulting RocksDB instance has four levels (including Level 0) and uses 52 GB of disk space. We configured[2] RocksDB according Facebook's recommendations [10, 78].

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. The equivalent-sized SuRF-Hash and SuRF-Real include a 4-bit suffix per key. We first warm the cache with 1M uniformly-

[2] 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`.

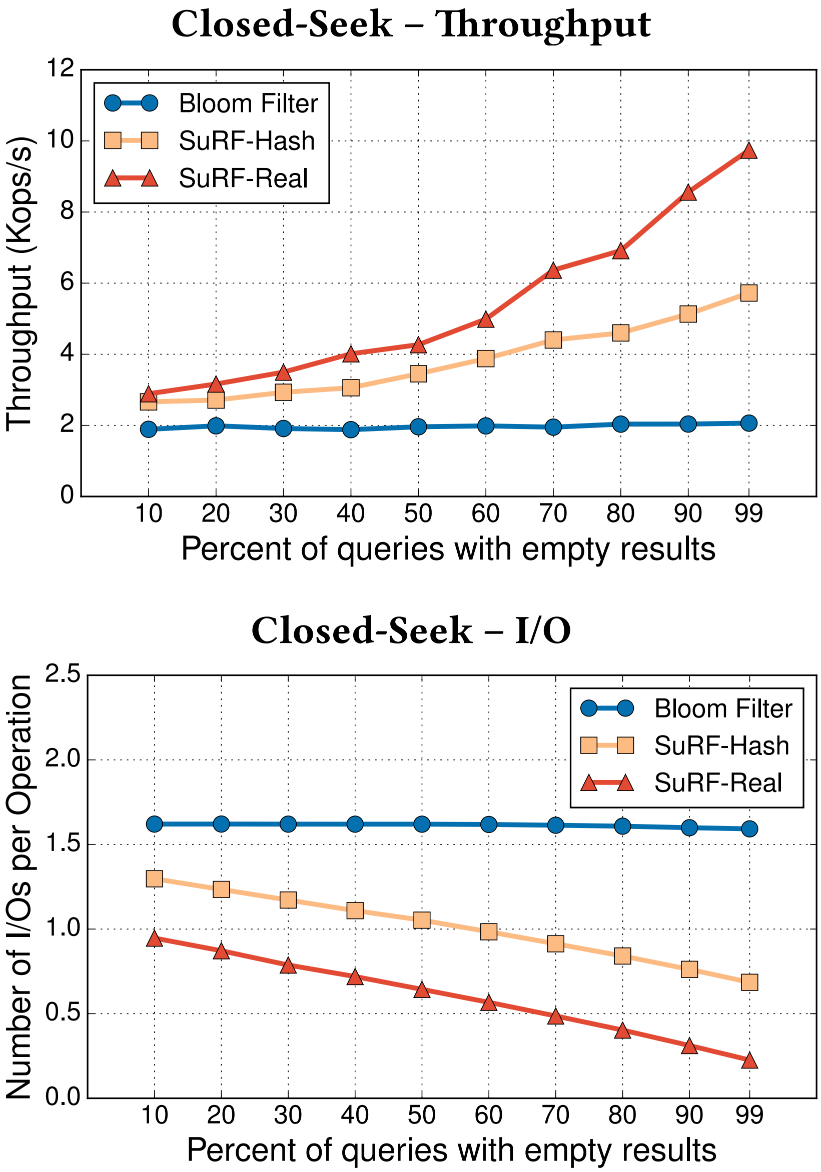


Figure 4.9: Closed-Seek Queries – RocksDB Closed-Seek query evaluation under different filter configurations and range sizes.

distributed point queries to existing keys so that every SSTable is touched approximately 1000 times and the block indexes and filters are cached. After the warm-up, both RocksDB's block cache and the OS page cache are full. We then execute 50K application queries, recording the end-to-end throughput and I/O counts. We compute the DBMS's throughput by dividing query counts by execution time, while I/O counts are read from system statistics before and after the execution. The query keys (for range queries, the starting keys) are randomly generated. The reported numbers are the average of three runs. Even though RocksDB supports prefix Bloom filters, we exclude them in our evaluation because they do not offer benefits over Bloom filters in this scenario: (1) range queries using arbitrary integers do not have pre-determined key prefixes, which makes it hard to generate such prefixes, and (2) even if key prefixes could be determined, prefix Bloom filters always return false positives for point lookups on absent keys sharing the same prefix with any present key, incurring high false positive rates.

Figure 4.8 (the first row) shows the result for point queries. Because the query keys are randomly generated, almost all queries return false. The query performance is dominated by the I/O count: they are inversely proportional. Excluding Level 0, each point query is expected to access three SSTables, one from each level (Level 1, 2, 3). Without filters, point queries incur approximately 1.5 I/Os per operation according to Figure 4.8, which means that the entire Level 1 and approximately half of Level 2 are likely cached. This agrees with the typical RocksDB application setting where the last two levels are not cached in memory [76].

Using filters in point queries reduces I/O because they prevent unnecessary block retrieval. Using SuRF-Hash or SuRF-Real is slower than using the Bloom filter because the 4-bit suffix does not reduce false positives as low as the Bloom filter configuration (refer to Section 4.3.1). SuRF-Real provides similar benefits as SuRF-Hash because the key distribution is sparse. One can shrink or eliminate the performance gap between Bloom filters and SuRFs by adding a few more suffix bits per key to the SuRFs.

The main benefit of using SuRF is accelerating range queries. Figure 4.8 (the second row) 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 as described in Section 4.2, 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 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 4.9 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. The Poisson distribution of events from all sensors has an expected frequency of one per $\lambda = 10^5$ ns. For an interval with length R, the probability that the range contains no event is given by $e^{-R/\lambda}$. Therefore, for a target percentage (P) of Closed-Seek queries with empty results, we set range size to $\lambda \ln(\frac{1}{P})$. For example, for 50%, the range size is 69310 ns.

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× when 99% of the queries return empty. Again, I/O count dominates performance. 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; RocksDB performs a read to the SSTable block only when the minimum key returned by the filters at each level falls into the querying range. 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.

To continue scanning after *Seek*, the DBMS calls *Next* and advances the iterator. We do not observe performance improvements for *Next* when using SuRF because the relevant SSTable blocks are already loaded in memory. Hence, SuRF mostly helps short range queries. As the range gets larger, the filtering benefit is amortized.

The RocksDB API does not support approximate queries. We measured the performance of approximate count queries using a simple prototype in LevelDB, finding that the speedup from using SuRF is similar to the speedup for Closed-Seek queries. (This result is expected based upon the execution paths in Figure 4.3). We believe it an interesting element of future work to integrate approximate counts (which are exact for static datasets) into RocksDB or another system more explicitly designed for approximate queries.

As a final remark, we evaluated RocksDB in a setting where the memory vs. storage budget is generous. The DBMS will benefit more from SuRF under a tighter memory constraint and a larger dataset.

4.5 The Theory-Practice Gaps

In this section, we discuss the theory-practice gaps between SuRF and an ideal range filter. The discussion includes a worst-case workload analysis on SuRF. Although we show that SuRF lacks certain theoretic guarantees, SuRF is still practical for many common applications. The discussion also suggests future directions in building a more powerful and efficient range filter.

The first theory-practice gap is that SuRF's performance and space-efficiency are workload-dependent. To illustrate this point, we constructed one of the worst-case datasets for SuRF in terms of performance and space-efficiency, as shown in Figure 4.10. In this dataset, we restrict the alphabet to be the lower case letters. Each key is 64 characters long, including a 5-character prefix, followed by a 58-character random string and 1-character suffix. The prefixes cover all possible 5-character combinations, with each combination appearing twice. The pair of keys that share the same prefix has the same random string followed but differs in the last byte. This way of constructing keys is unfriendly to SuRF because it maximizes the trie height (i.e., hurts performance) and minimizes the internal node sharing (i.e., hurts space-efficiency).

We evaluate SuRF on the above worst-case dataset. The experiment is similar to the

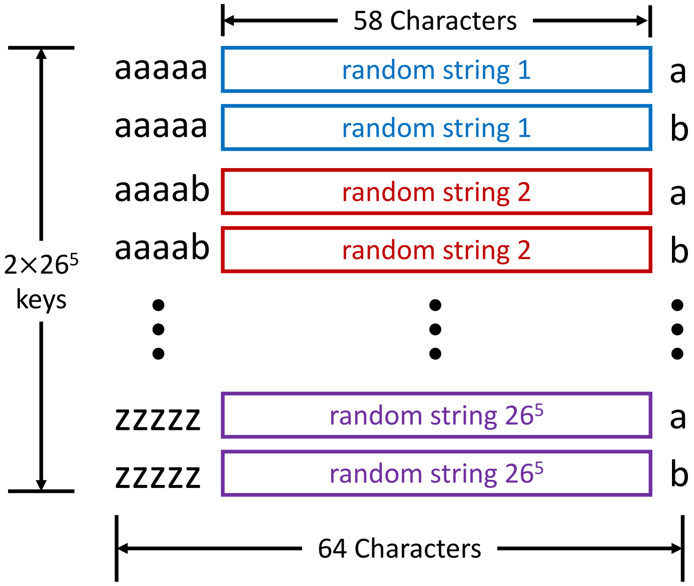


Figure 4.10: Worst-case Dataset – A worst-case dataset for SuRF in terms of performance and space-efficiency.

SuRF microbenchmarks in Section 4.3. Specifically, we insert all the keys in the dataset into SuRF-Base and then execute 10M point queries generated by YCSB workload C. Note that we store the entire dataset in SuRF (instead of 50% as in Section 4.3) so that every query reaches a leaf node in SuRF (i.e., no early exit) to allow the worst-case performance.

Figure 4.11 shows the throughput and memory results. We include the numbers for 64-bit integers and emails obtained from Section 4.3.2 for comparison. SuRF's performance is greatly compromised in the worst-case scenario because every query must traverse down 64 levels in the trie, causing a large number of cache misses. In terms of memory consumption, SuRF in the worst-case scenario takes 328 bits on average to encode each key, consuming memory that is equivalent to 64.1% of the dataset size. This is because our designed keys maximize the number of internal nodes in SuRF but minimize prefix sharing (i.e., each 58-character random string is only shared by two keys). Meanwhile, we have no suffix in the trie to truncate to save space. In other words, the SuRF that we built in the experiment is perfectly accurate (i.e., no false positives) because we stored every byte in each key.

The second theory-practice gap is that SuRF does not guarantee a theoretical false positive rate for range queries based on the number of bits used, despite that it achieves good empirical results. Goswami et al. [88] studied the theory aspect of the approximate range emptiness (i.e., range filtering) problem. They proved that any data structure that can answer approximate range emptiness queries has the worst-case space lower bound

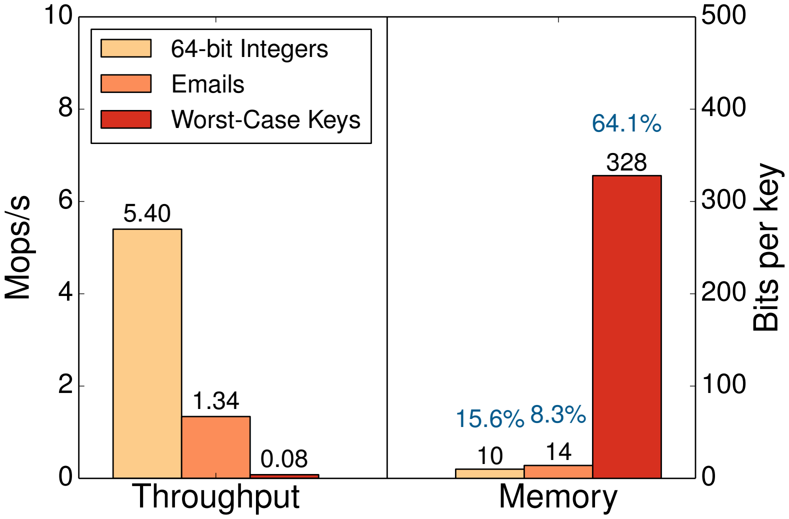


Figure 4.11: Worst-case Evaluation – SuRF's throughput and memory consumption on a worst-case dataset. The percentage numbers on the right are the size ratios between SuRF and the raw keys for each dataset.

of $\Omega(n \lg(L/\epsilon)) - O(n)$ bits, where n represents the number of items, L denotes the maximum interval length for range queries (in SuRF, L equals to the size of the key space), and ϵ is the false positive rate. In fact, this bound shows that there does not exist a "magic" data structure that can solve the range filtering problem by using only $n \lg(1/\epsilon) + O(n)$ bits as in Bloom filters [66]. In other words, even an "optimal" solution must use, in the worst case, close to the same number of bits needed to store the original data, truncated to the point where keys can be sufficiently differentiated from each other. In practice, on many datasets, however, SuRF provides a useful tradeoff between space and false positives. There is no contradiction here: SuRF 's succinct encoding helps it approach the lower bound in the worst case, and its trie structure practically compresses shared key prefixes when they exist.

Finally, the current version of SuRF only targets *static* use cases such as the log-structured merge tree described in Section 4.2. SuRF is a natural fit for LSM tree designs: when compaction creates a new SSTable, simply rebuild its associated SuRF. To create a deletable filter, we can introduce an additional "tombstone" bit-array with one bit per key to indicate whether the key has been deleted or not. With the tombstone bit-array, the cost of a *delete* in SuRF is almost the same as that of a *lookup*. For applications that require modifiable range filters, one can extend SuRF using a hybrid index [168]: A small dynamic trie sits in front of the SuRF and absorbs all *insert*s and *update*s; batch merges periodically rebuild the SuRF, amortizing the cost of individual modifications. We discuss the hybrid index architecture in detail in the next chapter.

Chapter 5

Supporting Dynamic Operations Efficiently: The Hybrid Index

So far in this thesis, we have focused on techniques to compress read-optimized (i.e., static) data structures. We showed in Chapters 2–4 that we can achieve optimal space while retaining high performance for static search trees. Although these static structures such as SuRF are useful in many practical scenarios, they lack efficient support for dynamic operations: An insert or update typically causes a rebuild of a significant part of the data structure. Such high modification cost limits the use of static trees in many database applications, especially for online transaction processing (OLTP) workloads where the ingestion rate is high. In this chapter, we relax this constraint by introducing techniques to speed up inserts and updates on static search trees with bounded and amortized cost in performance and space.

We present *hybrid index*, a dual-stage index architecture that can amortize the cost of modifying compact static data structures. Hybrid index is mainly designed for in-memory OLTP databases. It maintains a small dynamic "hot" store to absorb writes and a more compact, but read-only store to hold the bulk of index entries. Merge between the stages is triggered periodically and can be performed efficiently. Hybrid index leverages the skewed access pattern typically found in OLTP workloads. This skew manifests itself with respect to item popularity [72, 116]: certain data items are accessed more often than others and thus are more likely to be accessed again in the near future. This observation has been used extensively to move cold data from memory to block-based storage [72, 79, 154], and to store data efficiently by compressing the cold data in a main-memory database [84]. Unlike prior work [67, 108, 119, 152, 161], our design offers low latency and high throughput for the point queries and short-range scans that typify the OLTP workloads used with main-memory databases [106, 155].

A hybrid index explored unifying multiple underlying physical data structures, each with different optimization focuses, to construct a single logical entity. Our approach

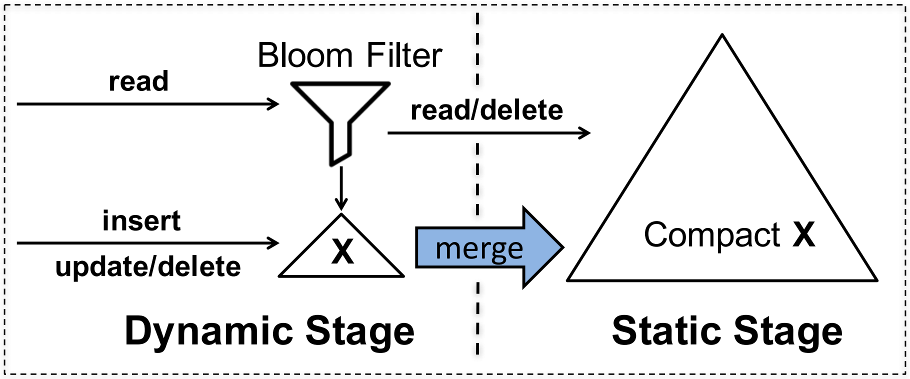


Figure 5.1: Dual-Stage Hybrid Index Architecture – All writes to the index first go into the dynamic stage. As the size of the dynamic stage grows, it periodically merges older entries to the static stage. For a read request, it searches the dynamic stage and the static stage in sequence.

differs from the log-structured merge trees (LSM-trees) [138] in several ways. First, log-structured engines are storage management systems that leverage the storage hierarchy while a hybrid index is an index data structure that resides only in memory. Such difference greatly influences a number of design decisions. For example, unlike LSM-trees, hybrid indexes avoid having too many stages/levels (unless the workload is extremely skewed) because the additional stages cause the worst-case read latency to increase proportionally to the number of stages. Furthermore, log-structured engines focus on speeding up writes while hybrid indexes target at saving memory space.

5.1 The Dual-Stage Architecture

As shown in Figure 5.1, 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 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 and migrates aged entries from its dynamic stage to the static stage which uses a more space-efficient data structure 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 (e.g., point queries, range queries) 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 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.

To facilitate using the dual-stage architecture to build hybrid indexes, we provide the following Dual-Stage Transformation steps for converting any order-preserving index structure to a corresponding hybrid index:

- **Step 1:** Select an order-preserving index structure (**X**) that supports dynamic operations efficiently for the dynamic stage.
- **Step 2:** Design a compact, read-optimized version of **X** for the static stage.
- **Step 3:** Provide a routine that can efficiently merge entries from **X** to compact **X**.
- **Step 4:** Place **X** and compact **X** in the dual-stage architecture as shown in Figure 5.1.

We note that these steps are a manual process. That is, a DBMS developer would need to convert the index to its static version. Automatically transforming any arbitrary data structure is outside the scope of this thesis.

An ideal data structure for the static stage must have three properties: First, it must be memory-efficient (i.e., have low space overhead per entry). Second, it must have good read performance for both point queries and range queries. This is particularly important for primary indexes where guaranteeing key uniqueness requires checking the static stage for every insert. Third, the data structure must support merging from the dynamic stage efficiently. This not only means that the merge process is fast, but also that the temporary memory use is low.

We have discussed techniques to accomplish Step 2 in the previous chapters. In this chapter, we choose to use the compacted search trees developed in Chapter 2 as example data structures for the static stage. These data structures (i.e., Compact B+tree, Compact Masstree, Compact Skip List, and Compact ART) were developed by applying the Dynamic-to-Static Rules to existing search trees, and they are good candidates for the static stage. First, they are more memory-efficient than the dynamic stage's indexes. Second, the data structures preserve the "essence" of the original indexes (i.e., they do not change the core structural designs fundamentally). This is important because applications sometimes choose certain index structures for certain workload patterns. For example, one may want to use a trie-based data structure to efficiently handle variable-length keys that have common prefixes. After applying the Dynamic-to-Static rules, a static trie is still a trie. Moreover, the similarity between the original and the compact structures enables an efficient merge routine to be implemented and performed without significant space overhead.

5.2 Merge

We focus on Step 3 of the Dual-Stage Transformation in this section: 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. There are trade-offs between blocking and non-blocking merge algorithms. Blocking algorithms are faster but hurt tail latency while non-blocking algorithms execute more smoothly but affect more queries because of locking and latching. Implementing non-blocking merge algorithms is out of the scope of this thesis, and we briefly discuss a proposed solution in Chapter 8.

The results in Section 5.3.3 show that our 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. Section 5.2.1 describes the merge algorithm. Section 5.2.2 discusses two important runtime questions: (1) what data to merge from one stage to the next; and (2) when to perform this merge.

5.2.1 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. As shown in the first part of this thesis (i.e., Chapters 2–4), 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.

We now briefly introduce the algorithms for merging B+tree, Masstree, Skip List, and ART to their compacted variations. The steps for merging B+tree to Compact B+tree is straightforward. They first merge the new items from the dynamic stage to the leaf-node arrays using the in-place merge sort algorithm described above. Then, the algorithm rebuilds the internal nodes level by level bottom up. The internal nodes are constructed based on the merged leaf nodes so that the balancing properties of the structures are maintained. Skip List merging uses a similar algorithm.

Merging Masstree and ART to their compacted versions uses recursive algorithms. When merging two trie nodes, the algorithms (depth-first) recursively create new merging tasks when two child nodes (or leaves/suffixes) require further merging. Figure 5.2 shows the pseudo-code for merging Masstree to Compact Masstree. The algorithm is a combination of merging sorted arrays and merging tries. We define three merge tasks that serve as building blocks for the merge process: merge two trie nodes, insert an item into a trie node, and create a trie node to hold two items. Note that the "==" sign between items in the pseudo-code means that they have equivalent keyslices.

The initial task is to merge the root nodes of the two tries, as shown in the *merge_nodes(root_m, root_n)* function in Figure 5.2. Merging any two trie nodes, including the root nodes, involves merging the sorted arrays of keys within the nodes. Conceptually, the algorithm proceeds as in a typical merge sort, except that it recursively creates new merging tasks. The merge process ends once the root node merge completes.

Merging ART to Compact ART adopts a slightly more complex recursive algorithm. Instead of checking the key suffixes directly within the node (as in Masstree), ART has to

```
merge_nodes(node m, n, parent):
  //merge the sorted arrays together
  merge_arrays(m, n)
  link n to parent

merge_arrays(node m, n):
  //2 running cursors: x for m, y for n
  for item x in m and item y in n:
    if x == y:                //equal keyslice
      recursively invoke:
      case 1: both x and y have child:
        merge_nodes(x.child, y.child, n)
      case 2: x has child, y has suffix:
        add_item(y.suffix, x.child, n)
      case 3: y has child, x has suffix:
        add_item(x.suffix, y.child, n)
      case 4: x.suffix != y.suffix:
        create_node(x.suffix, y.suffix, n)
    else
      move min(x, y) to new position in n
```

```
add_item(item x, node n, parent):
  //insert item x to the sorted arrays in n
  insert_one(x, n)
  link n to parent

insert_one(item x, node n):
  if x == (any item y in n):
    recursively invoke:
    case 1: y has child:
      add_item(x.suffix, y,child, n)
    case 2: y has suffix:
      create_node(x.suffix, y.suffix, n)
  else
    insert x to appropriate position in n
```

```
create_node(item x, y, node parent):
  //create a new node to hold x and y
  n = new_node(x, y)
  if x == y:
    create_node(x.suffix, y.suffix, n)
  link n to parent
```

Figure 5.2: Algorithm of merging Masstree to Compact Masstree – A recursive algorithm that combines trie traversal and merge sort.

load the full keys from the records and extract the suffixes based on the current trie depth. The two optimizations (lazy expansion and pass compression) in ART further complicates the algorithm because child nodes of the same parent can be at different levels.

5.2.2 Merge Strategy

In this section, we discuss two important design decisions: (1) what to merge, and (2) when to merge.

What to Merge: On every merge operation, the system must decide which entries to move from the dynamic stage to the static stage. Strategy one, called merge-all, 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 during a merge makes room for the incoming entries and alleviates the merge pressure as much as possible. An alternative strategy, merge-cold, tracks key popularity and selectively merges the cold entries to the static stage.

The two strategies interpret the role of the dynamic stage differently. Merge-all treats the dynamic stage as a write buffer that absorbs new records, amortizing the cost of bulk insert into the static stage. Merge-cold, however, treats the dynamic stage as a write-back cache that holds the most recently accessed entries. Merge-cold represents a tunable spectrum of design choices depending on how hot and cold are defined, of which merge-all is one extreme.

The advantage of merge-cold is that it creates "shortcuts" for accessing hot entries. However, it makes two trade-offs. First, it typically leads to higher merge frequency because keeping hot entries renders the dynamic stage unable to absorb as many new records before hitting the merge threshold again. The merge itself will also be slower because it must consider the keys' hot/cold status. Second, merge-cold imposes additional overhead for tracking an entry's access history during normal operations.

Although merge-cold may work better in some cases, given the insert-intensive workload patterns of OLTP applications, we consider merge-all to be the more general and more suitable approach. We compensate for the disadvantage of merge-all (i.e., some older yet hot tuples reside in the static stage and accessing them requires searching both stages in order) by adding a Bloom filter atop the dynamic stage as described in Section 5.1.

When to Merge: The second design decision is what event triggers the merge process to run. One strategy to use is a ratio-based trigger: merge occurs whenever the size ratio between the dynamic and the static stages reaches a threshold. An alternative strategy is to have a constant trigger that fires whenever the size of the dynamic stage reaches a constant 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. The side effect is that the average size of the dynamic stage gets larger over time, resulting in an increasingly longer average search time in the dynamic stage.

A constant trigger works well for read-intensive workloads because it bounds the size of the dynamic stage ensuring fast look-ups. For write-intensive workloads, however, this strategy leads to higher overhead because it keeps a constant merge frequency even though merging becomes more expensive over time. We found that a constant trigger is not suitable for OLTP workloads due to too frequent merges. We perform a sensitivity analysis of the ratio-based merge strategy in Section 5.3.3. Although auto-tuning is another option, it is beyond the scope of this thesis.

5.3 Microbenchmark

For our evaluation, we created five hybrid indexes using the Dual-Stage Transformation steps proposed in Section 5.1. We use X to represent either B+tree, Masstree, Skip List, or ART. *Hybrid-Compact* (or simply *Hybrid*) X means that the static stage uses Compact X, i.e., the structures developed by applying the Compaction and Structural Reduction Rules to X as shown in Chapter 2. *Hybrid-Compressed* means that the static stage structure is also compressed using Snappy [39] according to the Compression Rule.

We evaluate hybrid indexes in two steps. In this section, we evaluate the hybrid index as stand-alone key-value data structure using YCSB-based microbenchmarks. We first show the separate impact on performance and space of a hybrid index's building blocks. We then compare each hybrid index to its original structure to show the performance trade-offs made by adopting a hybrid approach for better space-efficiency. We did not use an existing DBMS for this section because we did not want to taint our measurement with features that are not relevant to the evaluation.

In Section 5.4, we evaluate hybrid indexes inside H-Store, a horizontally partitioned in-memory OLTP database management system. We replace the default B+tree indexes with the corresponding transformed hybrid indexes and evaluate the entire DBMS end-to-end.

5.3.1 Experiment Setup & Benchmarks

We used a server with the following configuration in our evaluation:

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 (used only in Section 5.4)

We used a set of YCSB-based microbenchmarks to mimic OLTP index workloads [70]. We used its default workloads **A** (*read/write*, 50/50), **C** (*read-only*), and **E** (*scan/ insert*, 95/5) with Zipf distributions, which have skewed access patterns common to OLTP workloads. The initialization phase in each workload was also measured and reported as the *insert-only* workload. For each workload, we tested three key types: 64-bit random integers, 64-bit monotonically increasing integers, and email addresses with an average length of 30 bytes. The random integer keys came directly from YCSB while the email keys were drawn from a large email collection. All values are 64-bit integers to represent tuple pointers. To summarize:

Workloads: insert-only, read-only, read/write, scan/insert

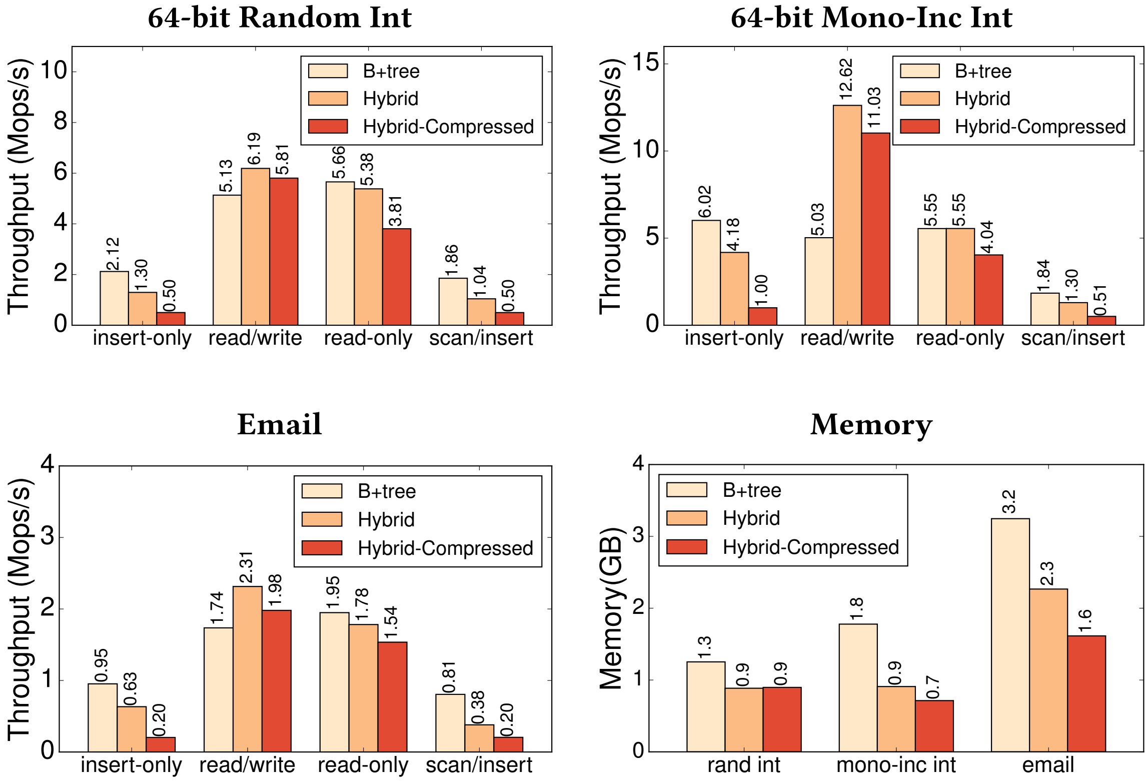


Figure 5.3: Hybrid B+tree vs. Original B+tree – Throughput and memory measurements for B+tree, Hybrid B+tree, and Hybrid-Compressed B+tree on different YCSB-based workloads and key types. The data structures are used as primary key (i.e., unique) indexes. Note that the figures have different Y-axis scales.

Key Types: 64-bit random int, 64-bit mono-inc int, email

Value: 64-bit integer (tuple pointers)

All experiments in this section are single-threaded without any network activity. We first insert 50 million entries into the index. We then execute 10 million key-value queries according to the workload and measure the execution time and index memory. Throughput results are the number of operations divided by the execution time; memory consumption is measured at the end of each trial. We report the average measurements from three independent trials.

5.3.2 Hybrid Indexes vs. Originals

We compare the hybrid indexes to their corresponding original structures to show the trade-offs of adopting a hybrid approach. We conducted separate experiments using the

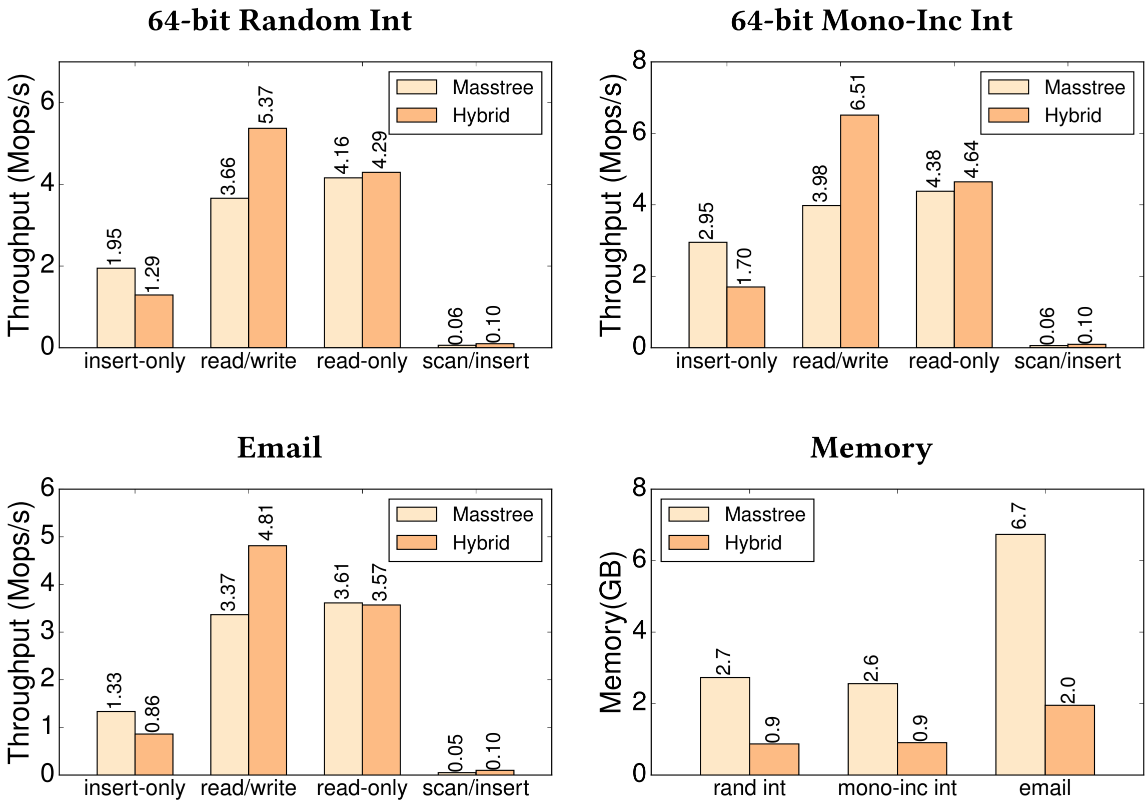


Figure 5.4: Hybrid Masstree vs. Original Masstree – Throughput and memory measurements for Masstree and Hybrid Masstree on different YCSB-based workloads and key types. The data structures are used as primary key (i.e., unique) indexes.

data structures as both primary key (i.e., unique) and secondary key (i.e., non-unique) indexes. We present the primary key index evaluation in this section. Results for secondary key indexes are in Section 5.3.5.

Figures 5.3–5.6 shows the throughput and memory consumption for hybrid indexes used as primary key indexes. The main takeaway is that all of the hybrid indexes provide comparable throughputs (faster in some workloads, slower in others) to their original structures while consuming 30–70% 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: One disadvantage of a hybrid index is that it requires periodic merges. As shown in Figures 5.3–5.6, all hybrid indexes are slower than their original structures under the insert-only workloads since they have the highest merge demand. The merging, however, is not the main reason for the performance degradation. Instead, it is because a hybrid index must check both the dynamic and static stages on every insert to verify

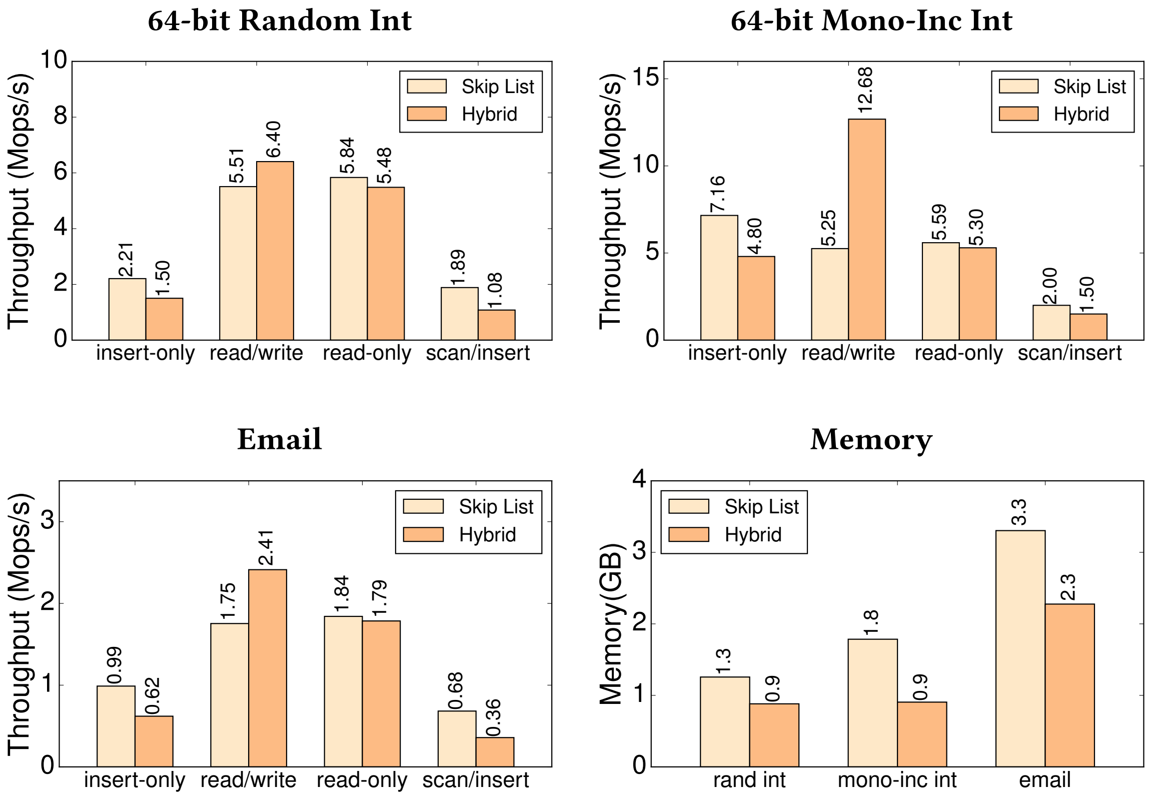


Figure 5.5: Hybrid Skip List vs. Original Skip List – Throughput and memory measurements for Skip List and Hybrid Skip List on different YCSB-based workloads and key types. The data structures are used as primary key (i.e., unique) indexes.

that a key does not already exist in either location. Such key uniqueness check causes about a 30% insert throughput drop. For the Hybrid-Compressed B+tree, however, merge remains the primary overhead because of the decompression costs.

Read/Write: Despite having to check for uniqueness in two locations on inserts, the hybrid indexes' dual-stage architecture is better at handling skewed updates. The results for this workload show that all of the hybrid indexes outperform their original structures for all key types because they store newly updated entries in the smaller (and therefore more cache-friendly) dynamic stage.

Read-only: We compared the point-query performance between the dynamic and static stage data structures in Section 2.5. When we put these structures together in a single hybrid index, the overall point-query performance is only slightly slower than the static stage alone because a query may have to check both data structures. We, therefore, use a Bloom filter in front of the dynamic stage to ensure that most reads only search one of the stages. We evaluate the impact of this filter later in Section 5.3.4.

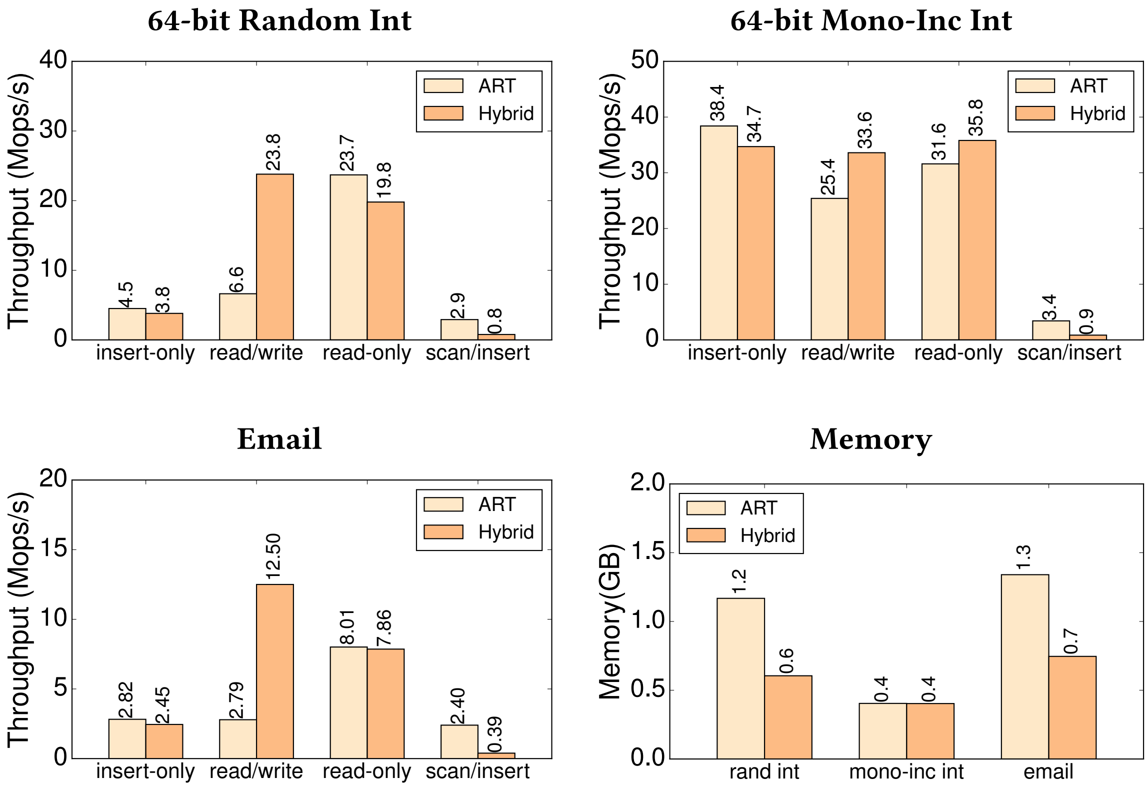


Figure 5.6: Hybrid ART vs. Original ART – Throughput and memory measurements for ART and Hybrid ART on different YCSB-based workloads and key types. The data structures are used as primary key (i.e., unique) indexes.

Scan/Insert: This last workload shows that the hybrid indexes have lower throughput for range queries. This is expected because their dual-stage design requires comparing keys from both the dynamic and static stages to determine the "next" entry when advancing the iterator. This comparison operation is particularly inefficient for Hybrid ART because the data structure does not store the full keys in the leaf nodes. Therefore, performing full-key comparison requires fetching the keys from the records first. We also note that range query results are less optimized in Masstree because it does not provide the same iterator API that the other index implementations support. We do not believe, however, that there is anything inherent to Masstree's design that would make it significantly better or worse than the other data structures for this workload.

Memory: All of the hybrid indexes use significantly less memory than their original data structures. An interesting finding is that although the random and mono-inc integer key datasets are of the same size, the B+tree and Skip List use more space to store the mono-inc integer keys. This is because the key insertion pattern of mono-inc integers produces B+tree nodes that are only 50% full (instead of 69% on average). The paged-

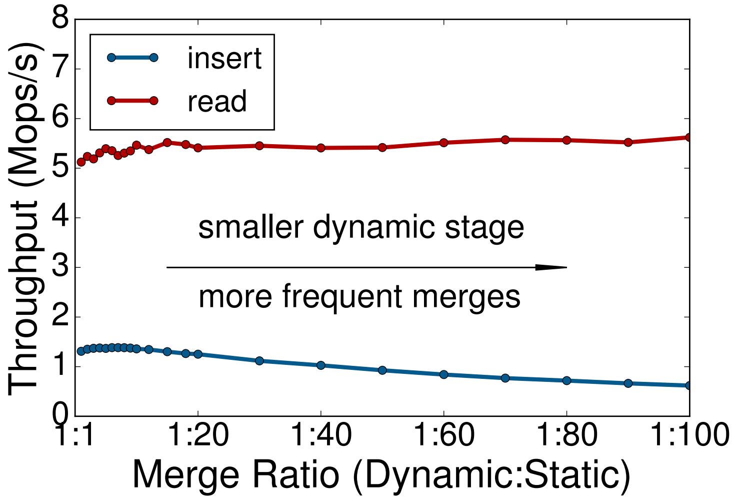


Figure 5.7: Merge Ratio – A sensitivity analysis of hybrid index's ratio-based merge strategy. The index used in this analysis is Hybrid B+tree.

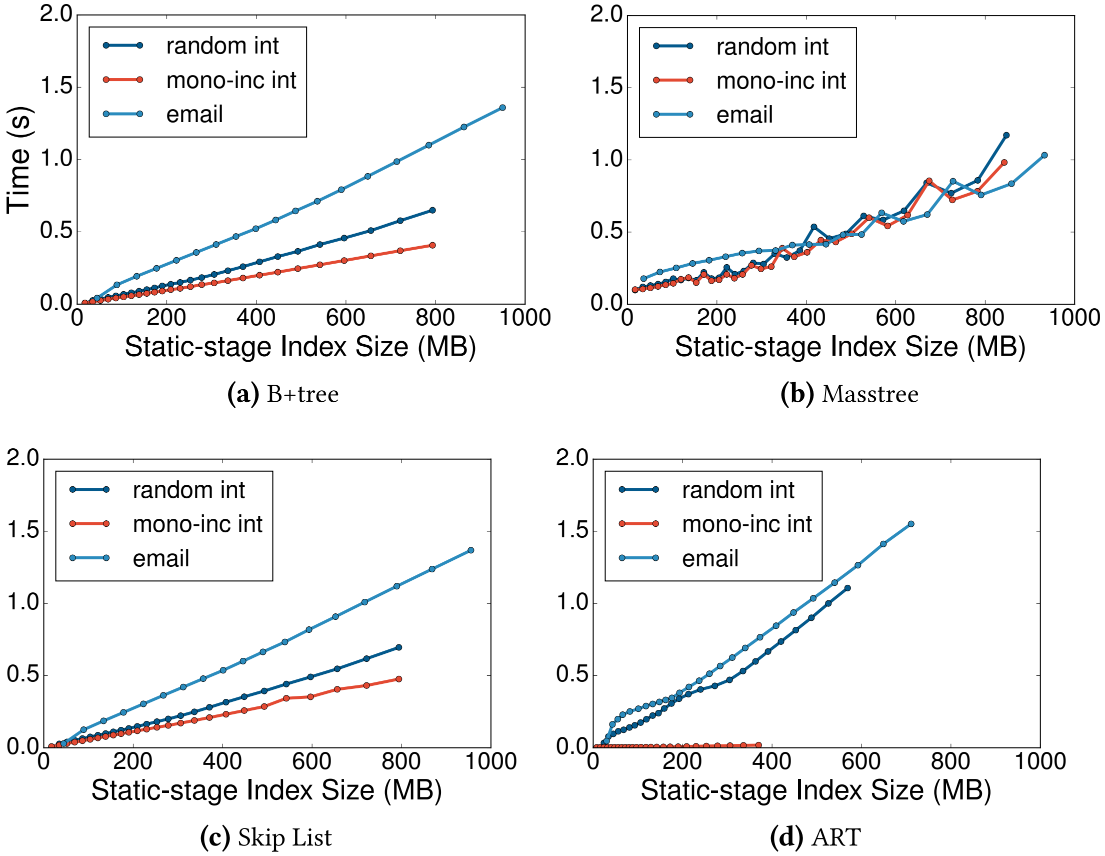
deterministic Skip List that we used has a similar hierarchical structure as the B+tree and thus has a similar node occupancy. ART, however, uses less space to store mono-inc keys than the random keys because of prefix compression, which also reduces memory for the email keys.

5.3.3 Merge Strategies & Overhead

We next zoom in and evaluate the merge process that moves data from the dynamic stage to the static stage at runtime. We concluded in Section 5.2.2 that ratio-based triggers are more suitable for OLTP applications because it automatically adjusts merge frequency according to the index size. Thus, we first show a sensitivity analysis of the ratio-based merge strategy

To determine a good default merge ratio that balances read and write throughput, we use the *insert-only* workload followed by the *read-only* workload with 64-bit integer keys to test ratios ranging from 1 to 100. For each ratio setting, we adjust the number of entries inserted so that the dynamic stage is about 50% "full" right before the read-only workload starts. We measure the average throughput of the hybrid index for the insert-only and read-only phases separately for each ratio. We only show the results for Hybrid B+tree because they are sufficient to demonstrate the relationship between the read/write throughput and merge ratio.

The results in Figure 5.7 show that a larger merge ratio leads to slightly higher read throughput and lower write throughput. A larger ratio keeps the dynamic stage smaller,



(a) B+tree

(b) Masstree

(c) Skip List

(d) ART

Figure 5.8: Merge Overhead – Absolute merge time given the static-stage index size. Dynamic-stage index size = $\frac{1}{10}$ static-stage index size.

thereby speeding up traversals in the dynamic stage. But it also triggers merges more frequently, which reduces the write throughput. As the merge ratio increases, the write throughput decreases more quickly than the read throughput increases. Since OLTP workloads are generally write-intensive, they benefit more from a relatively small ratio. Based on the analysis, we choose 10 as the default merge ratio for all hybrid indexes in the subsequent experiments in this chapter.

Using the default merge strategy, we next measure the cost of the merge process. We used the *insert-only* workload in this experiment because it generates the highest merge demand. For all four hybrid indexes and all three key types, we recorded the absolute time for every triggered merge operation along with the static-stage index size at the time of the execution to measure the merge speed. Note that the size of the dynamic stage is always 1/10 of that of the static stage at merge.

Figure 5.8 illustrates how the merge time changes with the size of the static stage of

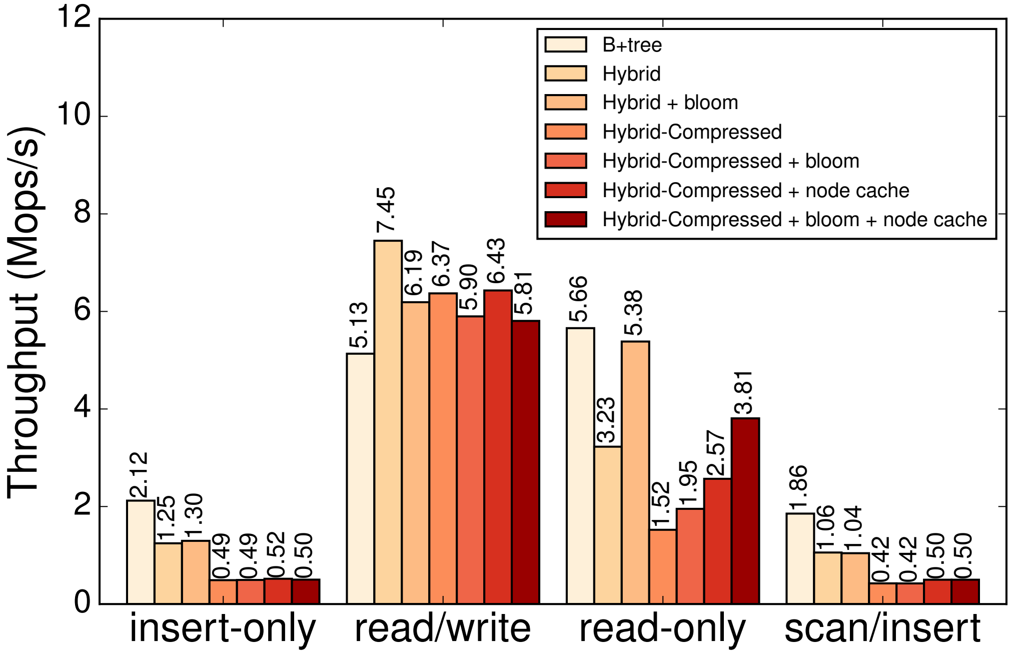
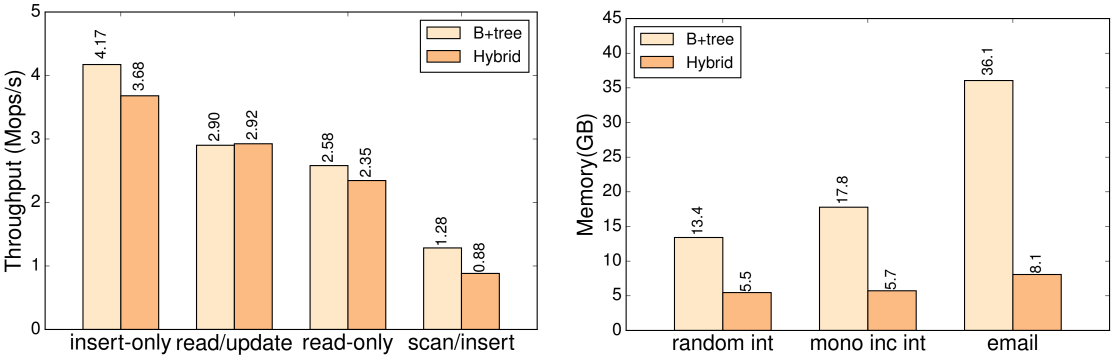


Figure 5.9: Auxiliary Structures – This figure is an extended version of the (B+tree, 64-bit random int) experiment in Figure 5.3 that shows the effects of the Bloom filter and the node cache separately in the hybrid index architecture.

the indexes. In general, the time to perform a merge increases linearly with the size of the index. Such linear growth is inevitable because of the fundamental limitations of merging sorted arrays. But merging occurs less frequently as the index size increases because it takes longer to accumulate enough new entries to reach the merge ratio threshold again. As such, the amortized cost of merging remains constant over time. We also observe an interesting exception when running Hybrid ART using mono-inc integer keys. As Figure 5.8d shows, the merge time (red line) is much lower than the other key types. This is because the Hybrid ART does not store nodes at the same level contiguously in an array in the same manner as the other data structures. Hence, the merge process for ART with mono-inc integers only needs to create and rebuild a few number of nodes to complete the merge, which is faster than re-adjusting the entire array.

5.3.4 Auxiliary Structures

We show the effects of two auxiliary structures presented in the hybrid index architecture: the Bloom filter (see Figure 5.1) and the node cache (see Figure 2.3). We extend the (B+tree, 64-bit random int) experiment in Figure 5.3 by making the inclusion of Bloom filter and node cache controlled variables to show their effects on performance separately.



(a) Throughput **(b)** Memory

Figure 5.10: Hybrid Index vs. Original (Secondary Indexes) – Throughput and memory measurements for different YCSB workloads using 64-bit random integer keys when the data structures are used as secondary (i.e., non-unique) indexes. The data set contains 10 values for each unique key.

Figure 5.9 presents the results. For all variants of the hybrid index, the read-only throughput improves significantly when adding the Bloom filter; similarly, adding a node cache also improves throughput over the same index variant without a node cache. In addition, Bloom filter and node cache improve read performance without noticeable overhead for other non-read-only workloads.

5.3.5 Secondary Indexes Evaluation

Lastly, we extend Section 5.3.2 by providing the experiment results for hybrid indexes used as secondary indexes. The experiment setup is described in Section 5.3.1. We insert ten values (instead of one, as in primary indexes) for each unique key. Because we implement multi-value support for all indexes in the same way, we only show the result for Hybrid B+tree in the 64-bit random integer key case as a representative to demonstrate the differences between using hybrid indexes as primary and secondary indexes.

As shown in Figure 5.10, the secondary index results are consistent with the primary index findings with several exceptions. First, the insert throughput gap between the original and Hybrid B+tree shrinks because secondary indexes do not require a key-uniqueness check for an insert, which is the main reason for the slowdown in the primary index case. Second, Hybrid B+tree loses its large throughput advantage in the read/write (i.e., update-heavy) workload case because it handles these value updates in-place rather than inserting new entries into the dynamic stage (as for primary indexes). In-place updates prevent the same key from appearing in both stages with different sets of values, which would require a hybrid index to search both stages to construct a complete value

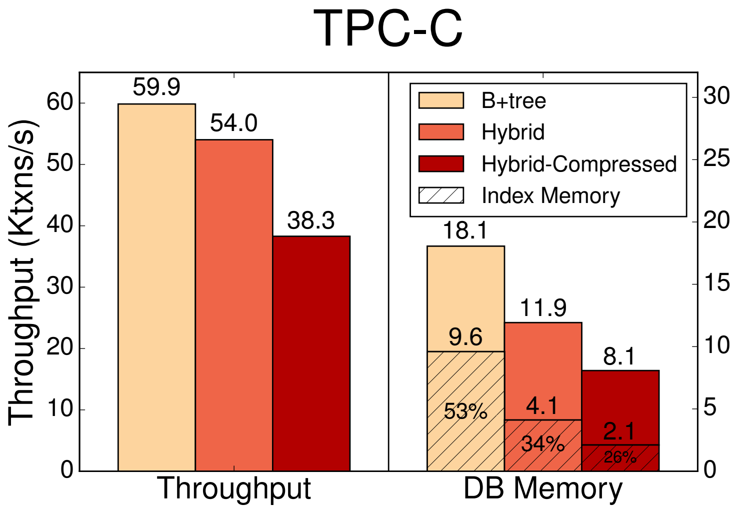


Figure 5.11: In-Memory Workload (TPC-C) – Throughput and memory measurements of the H-Store DBMS using the default B+tree, Hybrid, and Hybrid-Compressed B+tree when running the TPC-C workload that fit entirely in memory. The system runs for 6 min in each trial.

list for a key. Third, the memory savings of Hybrid B+tree are more significant in the secondary index case because the original B+tree stores duplicate keys while Compact B+tree does not.

5.4 Full DBMS Evaluation

This section shows the effects of integrating hybrid indexes into the in-memory H-Store OLTP DBMS [13, 101]. The latest version of H-Store uses B+tree as its default index data structure. We show that switching 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. We omit the evaluation of the other hybrid data structures because they provide similar benefits.

5.4.1 H-Store Overview

H-Store is a distributed, row-oriented DBMS that supports serializable execution of transactions over main memory partitions [101]. It is optimized for the efficient execution of workloads that contain transactions invoked as pre-defined stored procedures. Client applications initiate transactions by sending the procedure name and input parameters to any node in the cluster. Each partition is assigned a single-threaded execution engine that is responsible for executing transactions and queries for that partition. A partition

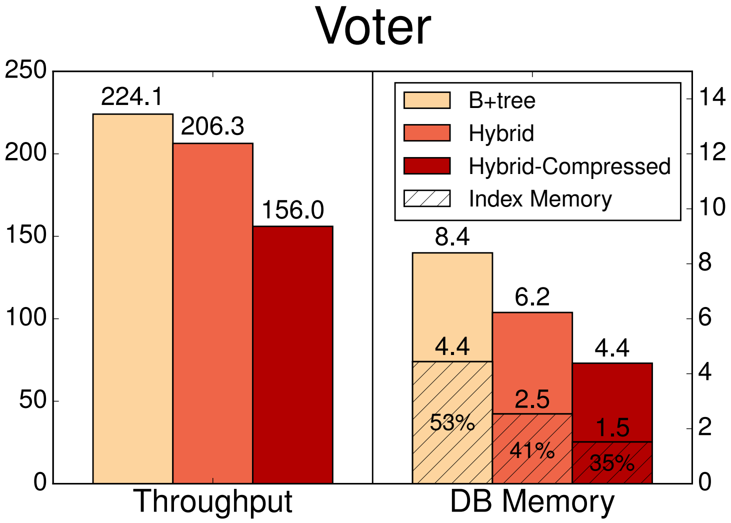


Figure 5.12: In-Memory Workload (Voter) – Throughput and memory measurements of the H-Store DBMS using the default B+tree, Hybrid, and Hybrid-Compressed B+tree when running the Voter workload that fit entirely in memory. The system runs for 6 min in each trial.

is protected by a single lock managed by its coordinator that is granted to transactions one-at-a-time based on the order of their arrival timestamp.

Anti-caching is a memory-oriented DBMS design that allows the system to manage databases that are larger than the amount of memory available without incurring the performance penalty of a disk-oriented system [72]. When the amount of in-memory data exceeds a user-defined threshold, the DBMS moves data to disk to free up space for new data. To do this, the system dynamically constructs blocks of the coldest tuples and writes them asynchronously to the anti-cache on disk. The DBMS maintains in-memory "tombstones" for each evicted tuple. When a running transaction attempts to access an evicted tuple through its tombstone, the DBMS aborts that transaction and fetches the tuple from the anti-cache without blocking other transactions. Once the data that the transaction needs is in memory, the system restarts the transaction.

5.4.2 Benchmarks

We use H-Store's built-in benchmarking framework to execute three workloads:

TPC-C: The TPC-C benchmark is the current industry standard for evaluating the performance of OLTP systems [156]. Its five stored procedures simulate a warehouse-centric order processing application. Approximately 88% of the transactions executed in TPC-C modify the database. We configure the database to contain eight warehouses and 100,000 items.

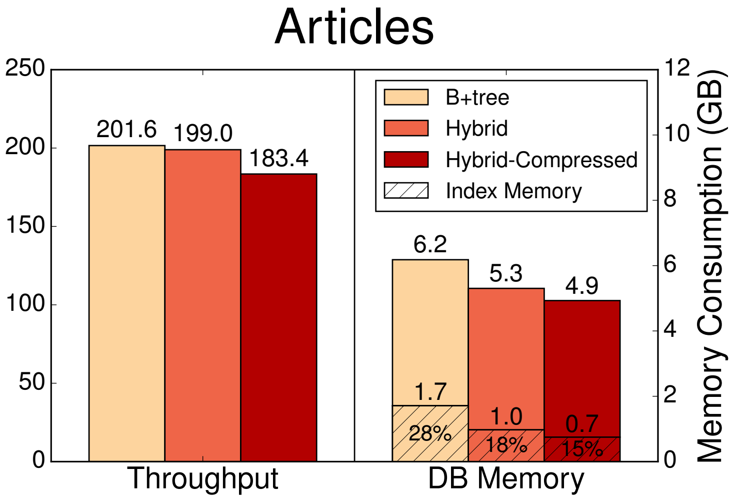


Figure 5.13: In-Memory Workload (Articles) – Throughput and memory measurements of the H-Store DBMS using the default B+tree, Hybrid, and Hybrid-Compressed B+tree when running the Articles workload that fit entirely in memory. The system runs for 6 min in each trial.

Voter: This benchmark simulates a phone-based election application. It is designed to saturate the DBMS with many short-lived transactions that all update a small number of records. There are a fixed number of contestants in the database. The workload is mostly transactions that update the total number of votes for a particular contestant. The DBMS records the number of votes made by each user based on their phone number; each user is only allowed to vote a fixed number of times.

Articles: This workload models an on-line news website where users submit content, such as text posts or links, and then other users post comments to them. All transactions involve a small number of tuples that are retrieved using either primary key or secondary indexes. We design and scale the benchmark so that the transactions coincide roughly with a week of Reddit's [35] traffic.

5.4.3 In-Memory Workloads

We first show that using hybrid indexes helps H-Store save a significant amount of memory. We ran the aforementioned three DBMS benchmarks on H-Store (anti-caching disabled) with three different index types: (1) B+tree, (2) Hybrid B+tree, and (3) Hybrid-Compressed B+tree. Each 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 deployed eight clients on the same machine using another eight cores on the other socket to exclude network factors. We compared throughput, index memory consumption, and total database memory consumption between the three index types.

	B+tree	Hybrid	Hybrid-Compressed
50%-tile	10 ms	10 ms	11 ms
99%-tile	50 ms	52 ms	83 ms
MAX	115 ms	611 ms	1981 ms

Table 5.1: TPC-C Latency Measurements – Transaction latencies of H-Store using the default B+tree, Hybrid B+tree, and Hybrid-Compressed B+tree as indexes for the TPC-C workload (same experiment as in Figure 5.11).

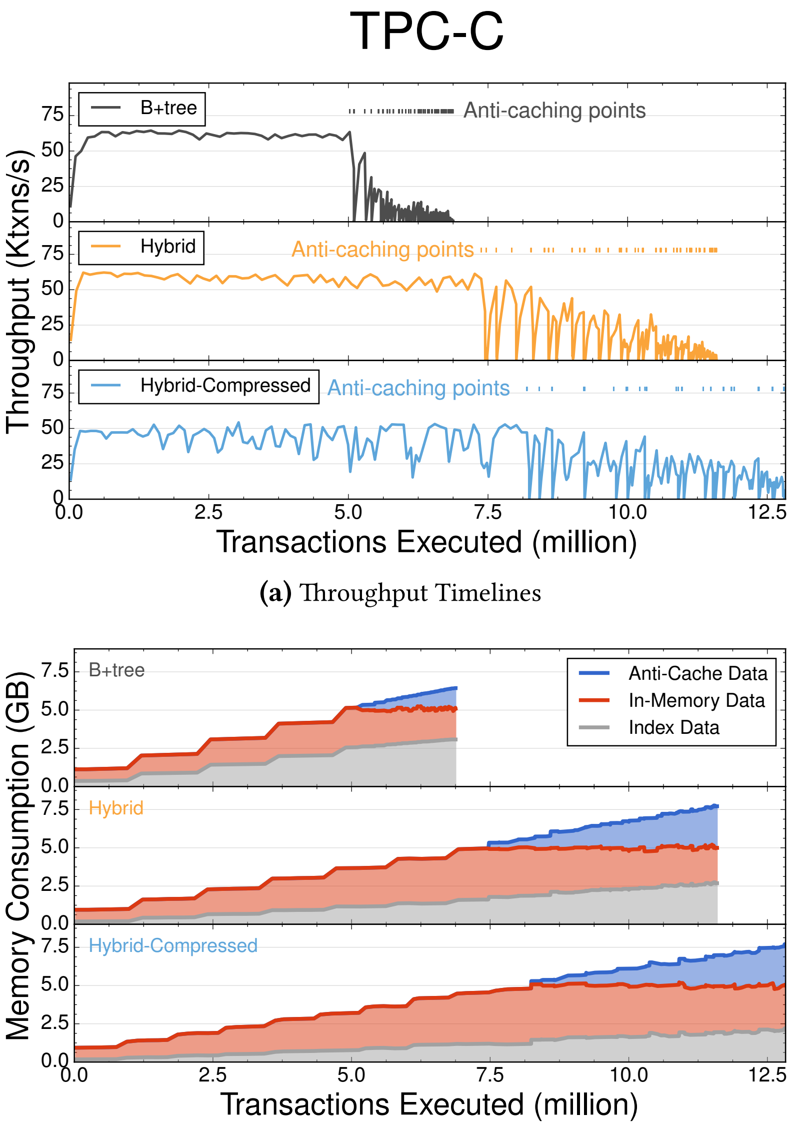
Figures 5.11–5.13 show the results. The throughput results are the average throughputs during the execution time (warm-up period excluded); memory consumption is measured at the end of each benchmark. We repeated each benchmark three times and compute the average for the final results.

As shown in Figures 5.11–5.13, both Hybrid and Hybrid-Compressed B+tree have a smaller memory footprint than the original B+tree: by 40–55% and 50–65%, respectively. The memory savings for the entire database depend on the relative size of indexes to the database. Hybrid indexes favor workloads with small tuples, as in TPC-C and Voter, so the index memory savings translate into significant savings at the database level.

Hybrid B+tree incurs a 1–10% average throughput drop compared to the original, which is fairly small considering the memory savings. Hybrid-Compressed B+tree, however, sacrifices throughput more significantly to reap its additional memory savings. These two hybrid indexes offer a throughput-memory tradeoff that may depend on the application's requirements.

The results in Figures 5.11–5.13 are consistent with our findings in the microbenchmark evaluation (Section 5.3). The throughput drops associated with hybrid indexes are more noticeable in the TPC-C (10%) and Voter (8%) benchmarks because they are insert-intensive and contain a large fraction of primary indexes. Referring to the *insert-only* workloads in Figures 5.3–5.6, we see that hybrid indexes are slower when used as primary indexes because of the key-uniqueness check. The Articles benchmark, however, is more read-intensive. Since hybrid indexes provide comparable or better read throughput, the throughput drop in Figure 5.13 is small (1%).

Table 5.1 lists the 50%-tile, 99%-tile, and MAX latency numbers for the TPC-C benchmark. Hybrid indexes have little effect on 50%-tile and 99%-tile latencies. For example, the difference in 99% latency between Hybrid B+tree and the original is almost negligible. The MAX latencies, however, increase when switching to hybrid indexes because our current merge algorithm is blocking. But the infrequency of merge means that the latency penalty only shows up when looking at MAX.



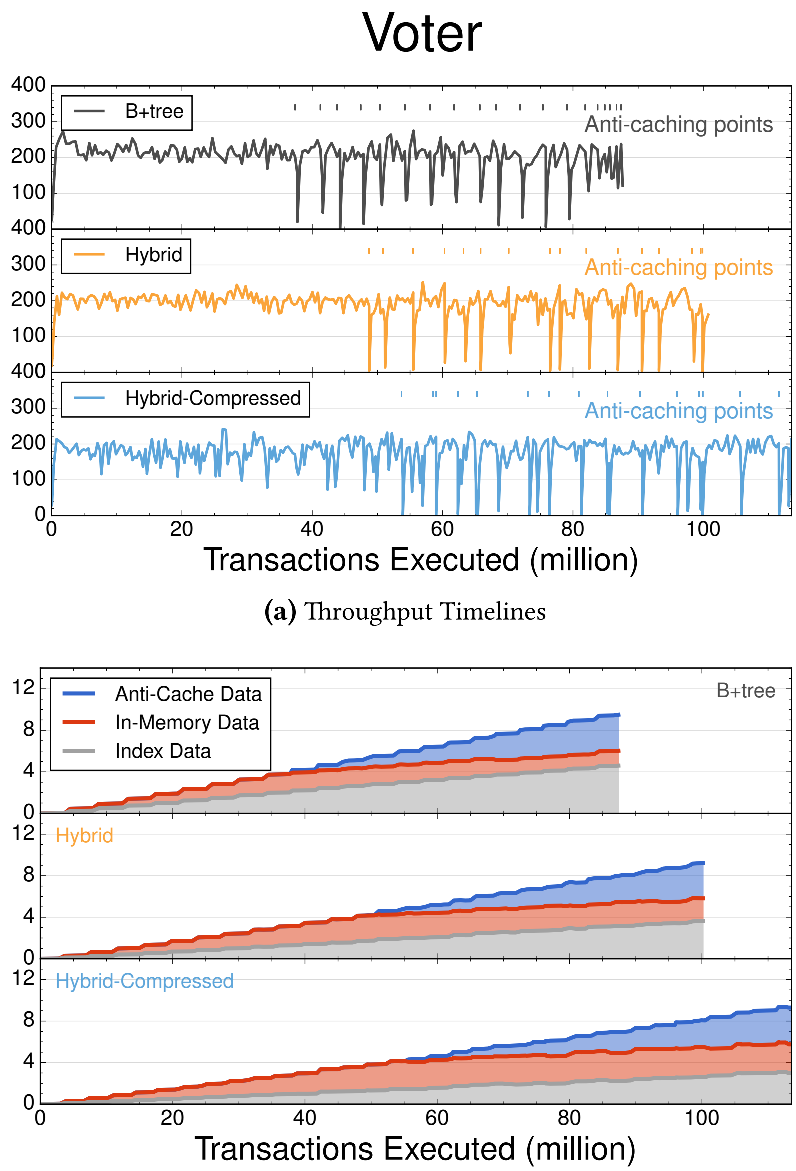
(a) Throughput Timelines

(b) Memory Breakdown

Figure 5.14: Larger-than-Memory Workload (TPC-C) – Throughput and memory measurements of the H-Store DBMS using B+tree, Hybrid, and Hybrid-Compressed B+tree as index structures when running the TPC-C workload that is 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. The system runs 12 minutes in each benchmark trial.

5.4.4 Larger-than-Memory Workloads

The previous section shows the savings from using hybrid indexes when the entire database fits in memory. Here, we show that hybrid indexes can further help H-Store

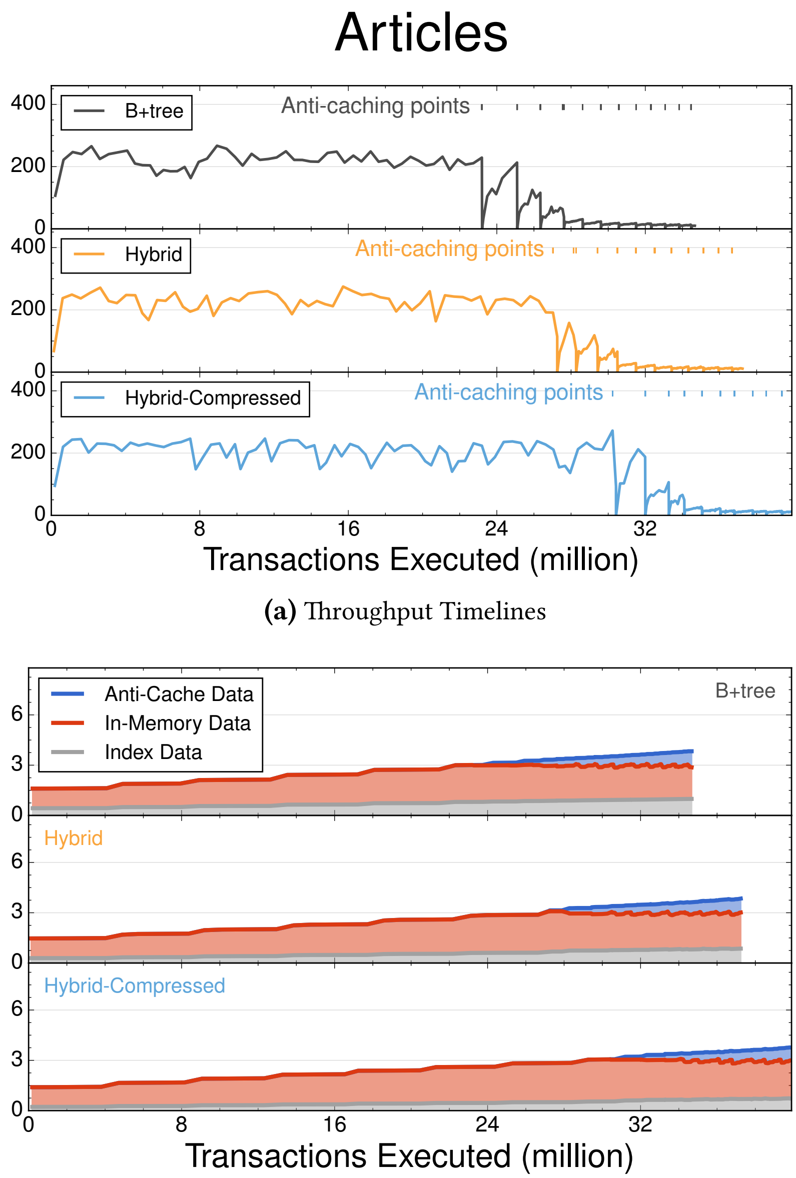


(a) Throughput Timelines

(b) Memory Breakdown

Figure 5.15: Larger-than-Memory Workload (Voter) – Throughput and memory measurements of the H-Store DBMS using B+tree, Hybrid, and Hybrid-Compressed B+tree as index structures when running the Voter workload that is larger than the amount of memory available to the system. The system runs 12 minutes in each benchmark trial.

with anti-caching enabled 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



(a) Throughput Timelines

(b) Memory Breakdown

Figure 5.16: Larger-than-Memory Workload (Articles) – Throughput and memory measurements of the H-Store DBMS using B+tree, Hybrid, and Hybrid-Compressed B+tree as index structures when running the Articles workload that is larger than the amount of memory available to the system. The system runs 12 minutes in each benchmark trial.

sustain a higher throughput because fewer queries must retrieve tuples from disk.

We ran TPC-C, Voter, and Articles on H-Store with anti-caching enabled for all three index configurations: B+tree, Hybrid B+tree, and Hybrid-Compressed B+tree. Each benchmark executes for 12 minutes after the initial load. We used the same client-server configurations as in Section 5.4.3. We set the anti-caching eviction threshold to be 5 GB for TPC-C and Voter, 3 GB for Articles so that the DBMS starts anti-caching in the middle of the execution. 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. Figures 5.14–5.16 show the experiment results; note that we use the **total number of transactions executed** on the x-axis rather than time.

Using hybrid indexes, H-Store with anti-caching executes more transactions than the original B+tree index during the same 12-minute run. We note that the B+tree and Hybrid B+tree configurations cannot execute the Voter benchmark for the entire 12 minutes because the DBMS runs out of memory to hold the indexes: only the database tuples can be paged out to disk.

Two features contribute to H-Store's improved capacity when using hybrid indexes. 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.

H-Store's throughput when using anti-caching depends largely on whether the workload reads evicted tuples [72]. TPC-C is an insert-heavy workload that mostly reads new data. Thus, TPC-C's throughput decreases relatively slowly as the tuples are evicted to disk. Voter never reads evicted data, so the throughput remains constant. Articles, however, is relatively read-intensive and occasionally queries cold data. These reads impact throughput during anti-caching, especially at the end of the run when a significant number of tuples have been evicted. The throughput fluctuations for hybrid indexes (especially Hybrid-Compressed indexes) before anti-caching are due to index merging. After anti-caching starts, the large throughput fluctuations are because of the anti-caching evictions since the current version of anti-caching is a blocking process: all transactions are blocked until the eviction completes.

Chapter 6

Compressing Input Keys: The High-Speed Order-Preserving Encoder

As we reduce the structural overhead of in-memory search trees towards the theoretical minimum, the actual keys stored in the trees start to dominate the space. In the final piece of this thesis, we address this problem by proposing an orthogonal approach to compress the individual input keys before inserting them into the search trees. Key compression is important for reducing index memory because real-world databases contain many variable-length string attributes [130] whose size often dominates the data structure's internal overheads. A common application of string compression is in columnar DBMSs [47], which often use dictionary compression to replace string values in a column with fixed-length integers. Traditional dictionary compression, however, does not work for in-memory search trees (e.g., OLTP indexes) for two reasons. First, the DBMS must continually grow its dictionary as new keys arrive. Second, key compression in a search tree must be order-preserving to support range queries properly.

We, therefore, present the **High-speed Order-Preserving Encoder** (HOPE), a dictionary-based key compressor for in-memory search trees (e.g., B+trees, tries). HOPE includes six entropy encoding schemes that trade between compression rate and encoding performance. When the DBMS creates a tree-based index/filter, HOPE samples the initial bulk-loaded keys and counts the frequencies of the byte patterns specified by a scheme. It uses these statistics to generate dictionary symbols that comply with our theoretical model to preserve key ordering. HOPE then encodes the symbols using either fixed-length codes or optimal order-preserving prefix codes. A key insight in HOPE is its emphasis on encoding speed (rather than decoding) because our target search tree queries need not reconstruct the original keys.

To evaluate HOPE, we applied it to five in-memory search trees: SuRF [169], ART [112], HOT [60], B+tree [43], and Prefix B+tree [55]. Our experimental results show that HOPE improves their latency by up to 40% and reduces their memory consumption

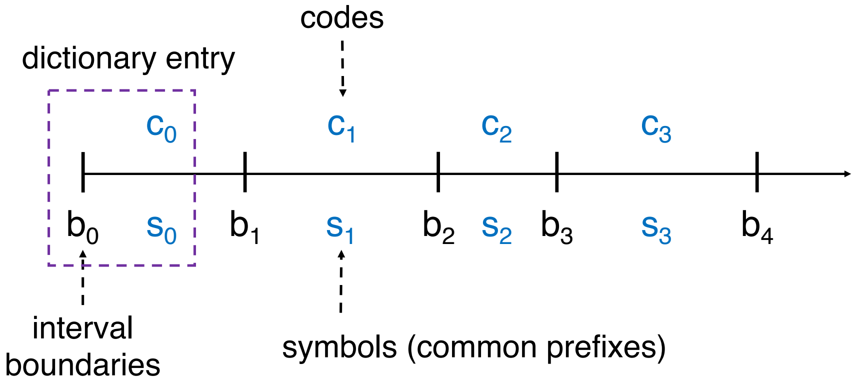


Figure 6.1: String Axis Model – The symbols are divided into connected intervals in lexicographical order. Strings in the same interval share a common prefix (s_i) that maps to code (c_i).

by up to 30%. HOPE improves both performance and memory use at the same time for most string key workloads.

6.1 Compression Model

Different dictionary encoding schemes, ranging from Huffman encoding [97] to the ALM-based compressor [50], provide different capabilities and guarantees. For example, some can encode arbitrary input strings while others preserve order. In this section, we introduce a unified model, called the string axis model, to characterize the properties of a dictionary encoding scheme. This model is inspired by the ALM string parsing algorithm [51], which solves the order-preserving problem for dictionary-based string compression. Using the string axis model, we can construct a wide range of dictionary-based compression schemes that can serve our target application (i.e., key compression for in-memory search trees). We divide qualified schemes into four categories, each making different trade-offs. We then briefly describe six representative compression schemes supported by HOPE.

6.1.1 The String Axis Model

As shown in Figure 6.1, a *string axis* lays out all possible source strings on a single axis in lexicographical order. We can model a dictionary encoding scheme using this representation and highlight three important properties: (1) completeness, (2) unique decodability, and (3) order-preserving.

Let Σ denote the source string alphabet. Σ^* is the set of all possible finite-length

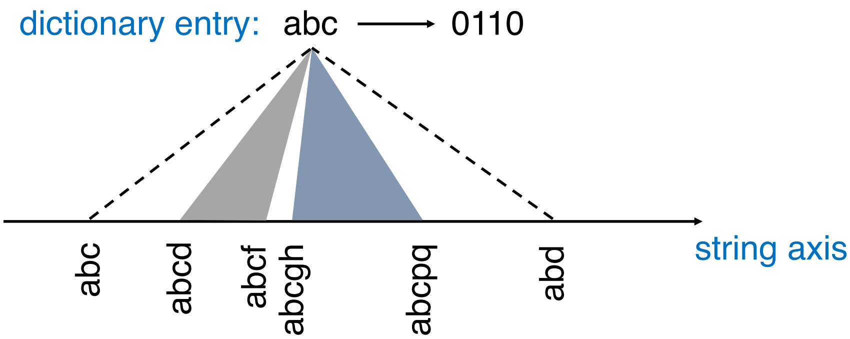


Figure 6.2: Dictionary Entry Example – All sub-intervals of [abc, abd) are valid mappings for dictionary entry abc ⟶ 0110.

strings over Σ. Similarly, let X denote the code alphabet and X^* be the code space. Typically, Σ is the set of all characters, and $X = \{0, 1\}$. A dictionary D maps a subset of the source strings S to the set of codes C:

$$D : S \to C, S \in \Sigma^*, C \in X^*$$

On the string axis, a dictionary entry $s_i \to c_i$ is mapped to an interval I_i, where s_i is a prefix of all strings within I_i. The choice of I_i is not unique. For example, as shown in Figure 6.2, both [abcd, abcf) and [abcgh, abcpq) are valid mappings for dictionary entry abc→0110. In fact, any sub-interval of [abc, abd) is a valid mapping in this example. If a source string src falls into the interval I_i, then a dictionary lookup on src returns the corresponding dictionary entry $s_i \to c_i$.

We can model the dictionary encoding method as a recursive process. Given a source string src, one can lookup src in the dictionary and obtain an entry $(s \to c) \in D, s \in S, c \in C$, such that s is a prefix of src, i.e., $src = s \cdot src_{suffix}$, where "$\cdot$" is the concatenation operation. We then replace s with c in src and repeat the process[1] using src_{suffix}.

To guarantee that encoding always makes progress, we must ensure that every dictionary lookup is successful. This means that for any src, there must exist a dictionary entry $s \to c$ such that $len(s) > 0$ and s is a prefix of src. In other words, we must consume some prefix from the source string at every lookup. We call this property **dictionary completeness**. Existing dictionary compression schemes for DBMSs are usually not complete because they only assign codes to the string values already seen by the DBMS. These schemes cannot encode arbitrary strings unless they grow the dictionary, but growing to accommodate new entries may require the DBMS to re-encode the entire

[1] One can use a different dictionary at every step. For performance reasons, we consider a single dictionary throughout the process in this chapter.

corpus [61]. In the string axis model, a dictionary is complete if and only if the union of all the intervals (i.e., $\bigcup I_i$) covers the entire string axis.

A dictionary encoding $Enc : \Sigma^* \to X^*$ is **uniquely decodable** if Enc is an *injection* (i.e., there is a one-to-one mapping from every element of Σ^* to an element in X^*). To guarantee unique decodability, we must ensure that (1) there is only one way to encode a source string and (2) every encoded result is unique. Under our string axis model, these requirements are equivalent to (1) all intervals I_i's are disjoint and (2) the set of codes C used in the dictionary are uniquely decodable (we only consider prefix codes here).

With these requirements, we can use the string axis model to construct a dictionary that is both complete and uniquely decodable. As shown in Figure 6.1, for a given dictionary size of n entries, we first divide the string axis into n consecutive intervals $I_0, I_1, \ldots, I_{n-1}$, where the max-length common prefix s_i of all strings in I_i is not empty (i.e., $len(s_i) > 0$) for each interval. We use $b_0, b_1, \ldots, b_{n-1}, b_n$ to denote interval boundaries. That is, $I_i = [b_i, b_{i+1})$ for $i = 0, 1, \ldots, n-1$. We then assign a set of uniquely decodable codes $c_0, c_1, \ldots, c_{n-1}$ to the intervals. Our dictionary is thus $s_i \to c_i, i = 0, 1, \ldots, n-1$. A dictionary lookup maps the source string src to a single interval I_i, where $b_i < src < b_{i+1}$.

We can achieve the **order-preserving** property on top of unique decodability by assigning monotonically increasing codes $c_0 < c_1 < \ldots < c_{n-1}$ to the intervals. This is easy to prove. Suppose there are two source strings (src_1, src_2), where $src_1 < src_2$. If src_1 and src_2 belong to the same interval I_i in the dictionary, they must share common prefix s_i. Replacing s_i with c_i in each string does not affect their relative ordering. If src_1 and src_2 map to different intervals I_i and I_j, then $Enc(src_1) = c_i \cdot Enc(src_{1_{suffix}})$, $Enc(src_2) = c_j \cdot Enc(src_{2_{suffix}})$. Since $src_1 < src_2$, I_i must preceed I_j on the string axis. That means $c_i < c_j$. Because c_i's are prefix codes, $c_i \cdot Enc(src_{1_{suffix}}) < c_j \cdot Enc(src_{2_{suffix}})$ regardless of what the suffixes are.

For encoding search tree keys, we prefer schemes that are complete and order-preserving; unique decodability is implied by the latter property. Completeness allows the scheme to encode arbitrary keys, while order-preserving guarantees that the search tree supports meaningful range queries on the encoded keys. For search tree applications that do not require unique decodability, a lossy compression scheme might be acceptable (or even preferable). Exploring lossy compression is out of the scope of this thesis, and we defer it to future work.

6.1.2 Exploiting Entropy

For a dictionary encoding scheme to reduce the size of the corpus, its emitted codes must be shorter than the source strings. Given a complete, order-preserving dictionary

$D : s_i \to c_i, i = 0, 1, \ldots, n-1$, let p_i denote the probability that a dictionary entry is accessed at each step during the encoding of an arbitrary source string. Because the dictionary is complete and uniquely decodable (implied by order-preserving), $\sum_{i=0}^{n-1} p_i = 1$. The dictionary encoding scheme achieves the best compression when the following compression rate is maximized:

$$CPR = \frac{\sum_{i=0}^{n-1} len(s_i)p_i}{\sum_{i=0}^{n-1} len(c_i)p_i}$$

According to the string axis model, we can characterize a dictionary encoding scheme in two parts: (1) how to divide intervals and (2) what code to assign to each interval. Interval division determines the symbol lengths ($len(s_i)$) and the access probability distribution (p_i) in a dictionary. Code assignment exploits the entropy in p_i's by using shorter codes (c_i) for more frequently-accessed intervals.

We consider two interval division strategies: fixed-length intervals and variable-length intervals. For code assignment, we consider two types of prefix codes: fixed-length codes and optimal variable-length codes. We, therefore, divide all complete and order-preserving dictionary encoding schemes into four categories, as shown in Figure 6.3.

Fixed-length Interval, Fixed-length Code (FIFC): This is the baseline scheme because ASCII encodes characters in this way. We do not consider this category for compression.

Fixed-length Interval, Variable-length Code (FIVC): This category is the classic Hu-Tucker encoding [95]. If order-preserving is not required, both Huffman encoding [97] and arithmetic encoding [162] also belong to this category[2]. Although intervals have a fixed length, access probabilities are not evenly distributed among the intervals. Using optimal (prefix) codes, thus, maximizes the compression rate.

Variable-length Interval, Fixed-length Code (VIFC): This category is represented by the ALM string compression algorithm proposed by Antoshenkov [50] Because the code lengths are fixed (i.e., $len(c_i) = L$), $CPR = \frac{1}{L}\sum_{i=0}^{n-1} len(s_i)p_i$. ALM applied the "equalizing" heuristic of letting $len(s_0)p_0 = len(s_1)p_1 = \cdots = len(s_n)p_n$ to try to achieve optimal compression (i.e., maximize CPR). We note that the example in Figure 6.3 has two intervals with the same dictionary symbol. This is allowed because only one of the intervals will contain a specific source string, which uniquely determines the result of a dictionary lookup. Also, by using variable-length intervals, we no longer have the "concatenation property" for the encoded results (e.g., $Code(ab) \neq Code(a) \cdot Code(b)$). This property, however, is not a requirement for our target application.

Variable-length Interval, Variable-length Code (VIVC): To the best of our knowledge, this category is unexplored by previous work. Although Antoshenkov suggests

[2] Arithmetic encoding does not operate the same way as a typical dictionary encoder. But its underlying principle matches this category.

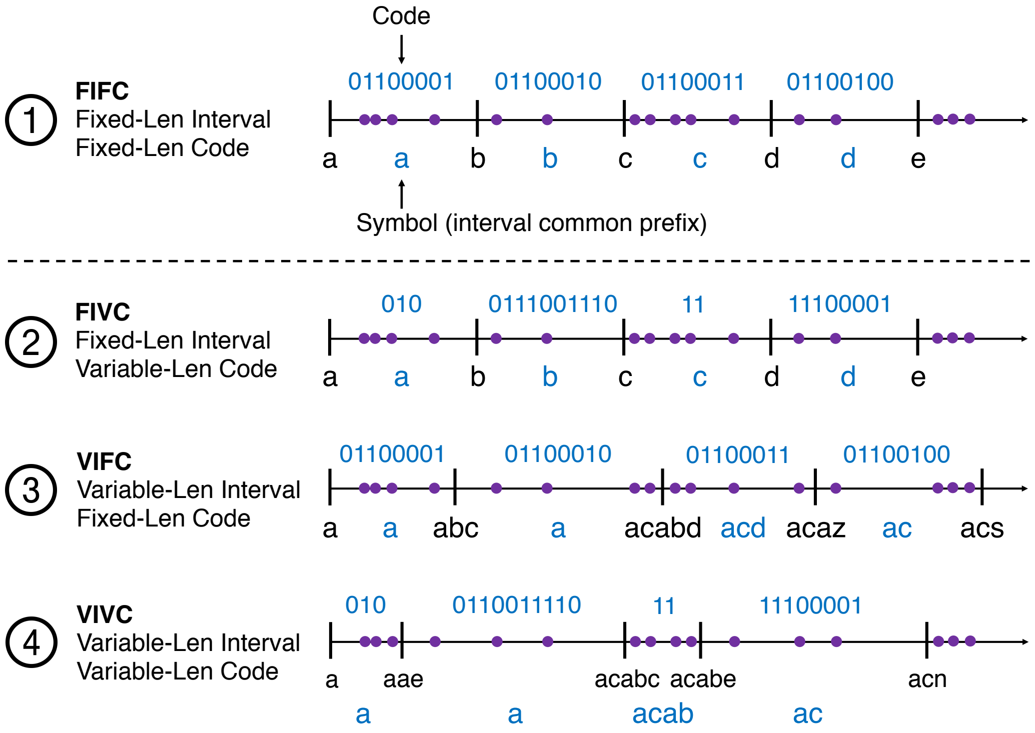


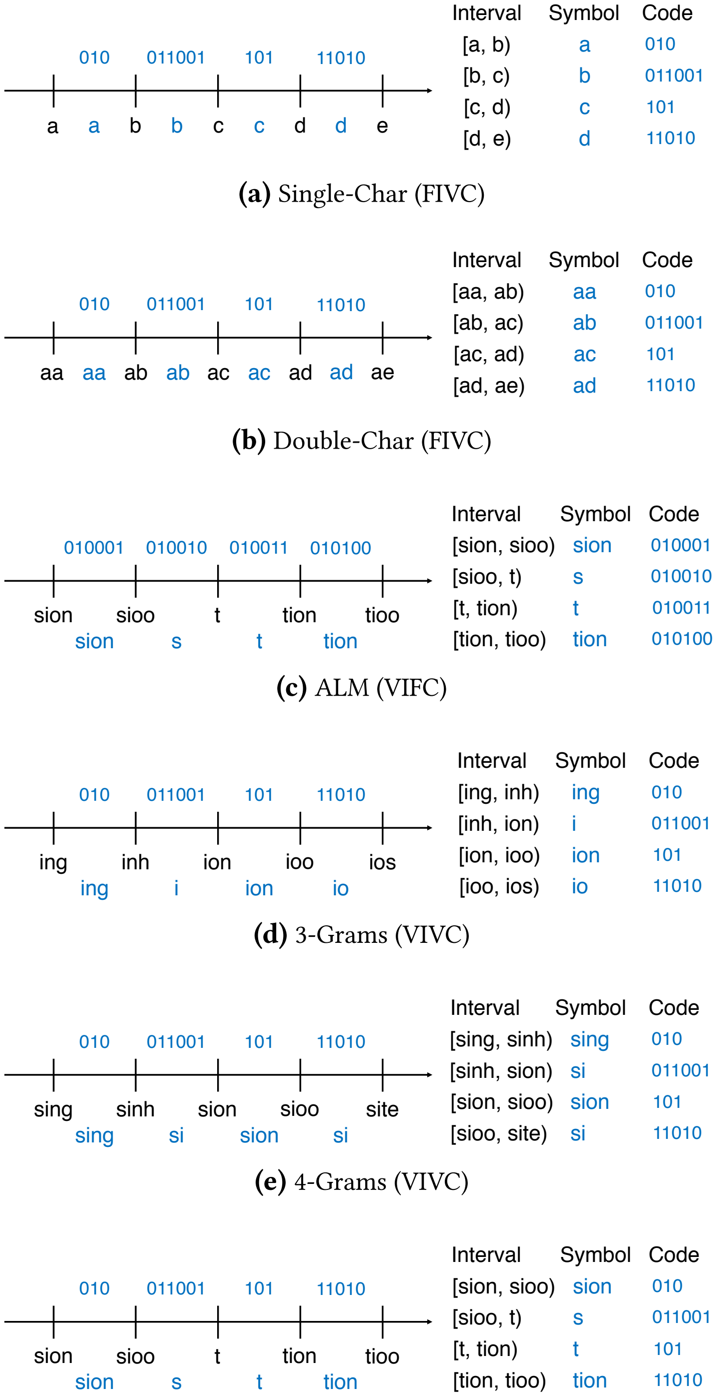
Figure 6.3: Compression Models – Four categories of complete and order-preserving dictionary encoding schemes.

that ALM could benefit from a supplementary variable-length code [50], it is neither implemented nor evaluated. VIVC has the most flexibility in building dictionaries (one can view FIFC, FIVC, and VIFC as special cases of VIVC), and it can potentially lead to optimal compression rate. We describe the VIVC schemes in HOPE in Section 6.1.3.

Although VIVC schemes can have higher compression rates than the other schemes, both fixed-length intervals and fixed-length codes have performance advantages over their variable-length counterparts. Fixed-length intervals create smaller and faster dictionary structures, while fixed-length codes are more efficient to decode. Our objective is to find the best trade-off between compression rate and encoding performance for in-memory search tree keys.

6.1.3 Compression Schemes

Based on the above dictionary encoding models, we next introduce six compression schemes implemented in HOPE. We select these schemes from the three viable categories (FIVC, VIFC, and VIVC). Each scheme makes different trade-offs between compression rate and encoding performance. We first describe them at a high level and then provide



Interval	Symbol	Code
[a, b)	a	010
[b, c)	b	011001
[c, d)	c	101
[d, e)	d	11010

(a) Single-Char (FIVC)

Interval	Symbol	Code
[aa, ab)	aa	010
[ab, ac)	ab	011001
[ac, ad)	ac	101
[ad, ae)	ad	11010

(b) Double-Char (FIVC)

Interval	Symbol	Code
[sion, sioo)	sion	010001
[sioo, t)	s	010010
[t, tion)	t	010011
[tion, tioo)	tion	010100

(c) ALM (VIFC)

Interval	Symbol	Code
[ing, inh)	ing	010
[inh, ion)	i	011001
[ion, ioo)	ion	101
[ioo, ios)	io	11010

(d) 3-Grams (VIVC)

Interval	Symbol	Code
[sing, sinh)	sing	010
[sinh, sion)	si	011001
[sion, sioo)	sion	101
[sioo, site)	si	11010

(e) 4-Grams (VIVC)

Interval	Symbol	Code
[sion, sioo)	sion	010
[sioo, t)	s	011001
[t, tion)	t	101
[tion, tioo)	tion	11010

(f) ALM-Improved (VIVC)

Figure 6.4: Compression Schemes – Example dictionary segments.

their implementation details in Section 6.2.

Single-Char is the FIVC compression algorithm used in Huffman encoding and arithmetic encoding. The fixed-length intervals have consecutive single characters as the boundaries (e.g., $[\mathtt{a}, \mathtt{b})$, $[\mathtt{b}, \mathtt{c})$). The dictionary symbols are 8-bit ASCII characters, and the dictionary has a fixed 256 entries. The codes assigned to the symbols are Hu-Tucker codes. Hu-Tucker codes are optimal order-preserving prefix codes (they are essentially order-preserving Huffman codes). Figure 6.4a shows an example dictionary segment.

Double-Char is a FIVC compression algorithm that is similar to Single-Char, except that the interval boundaries are consecutive double characters (e.g., $[\mathtt{aa}, \mathtt{ab})$, $[\mathtt{ab}, \mathtt{ac})$). To make the dictionary complete, we introduce a terminator character $\varnothing$ that is smaller than ASCII characters to fill the interval gaps between $[\mathtt{a`\backslash 255'}, \mathtt{b})$ and $[\mathtt{b`\backslash 0'}, \mathtt{b`\backslash 1'})$, for example, with the interval $[\mathtt{b}\varnothing, \mathtt{b`\backslash 0'})$. Figure 6.4b shows an example dictionary. This scheme should achieve better compression than Single-Char because it exploits the first-order entropy of the source strings instead of the zeroth-order entropy.

ALM is a state-of-the-art VIFC string compression algorithm. To determine the interval boundaries from a set of sample source strings (e.g., initial keys for an index), ALM first selects substring patterns that are long and frequent. Specifically, for a substring pattern s, it computes $len(s) \times freq(s)$, where $freq(s)$ represents the number of occurrence of s in the sample set. ALM includes s in its dictionary if the product is greater than a threshold W. It then creates one or more intervals for each gap between the adjacent selected symbols. The goal of the algorithm is to make the above product (i.e., length of the common prefix $\times$ access frequency) for each interval as equal as possible. The detailed algorithm is described in [50].

ALM uses monotonically increasing fixed-length codes. Figure 6.4c shows an example dictionary segment. The dictionary size for ALM depends on the threshold W. One must binary search on W's to obtain a desired dictionary size.

3-Grams is a VIVC compression algorithm where the interval boundaries are 3-character strings. Given a set of sample source strings and a dictionary size limit n, the scheme first selects the top $n/2$ most frequent 3-character patterns and adds them to the dictionary. For each interval gap between the selected 3-character patterns, 3-Grams creates a dictionary entry to cover the gap. For example, in Figure 6.4d, "$\mathtt{ing}$" and "$\mathtt{ion}$" are selected frequent patterns from the first step. "$\mathtt{ing}$" and "$\mathtt{ion}$" represent intervals $[\mathtt{ing}, \mathtt{inh})$ and $[\mathtt{ion}, \mathtt{ioo})$ on the string axis. Their gap interval $[\mathtt{inh}, \mathtt{ion})$ is also included as a dictionary entry. 3-Grams uses Hu-Tucker codes.

4-Grams is a VIVC compression algorithm similar to 3-Grams with 4-character string boundaries. Figure 6.4e shows an example. Compared to 3-Grams, 4-Grams exploits higher-order entropy; but whether it provides a better compression rate over 3-Grams depends on the dictionary size.

ALM-Improved improves the ALM scheme in two ways. First, as shown in Figure 6.4f, we replace the fixed-length codes in ALM with Hu-Tucker codes because we observe access skew among the intervals despite ALM's "equalizing" algorithm. Second, the original ALM counts the frequency for every substring (of any length) in the sample set, which is slow and memory-consuming. In ALM-Improved, we simplify the process by only collecting statistics for substrings that are suffixes of the sample source strings. Our evaluation in Section 6.4 shows that using Hu-Tucker codes improves ALM's compression rate while counting the frequencies of string suffixes reduces ALM's build time without compromising the compression rate.

6.2 HOPE

We now present the design and implementation of HOPE. There are two goals in HOPE's architecture. First, HOPE must minimize its performance overhead so that it does not negate the benefits of storing shorter keys. Second, HOPE must be extensible. From our discussion in Section 6.1, there are many choices in constructing an order-preserving dictionary encoding scheme. Although we support six representative schemes in the current version of HOPE, one could, for example, devise better heuristics in generating dictionary entries to achieve a higher compression rate, or invent more efficient dictionary data structures to further reduce encoding latency. HOPE can be easily extended to include such improvements through its modularized design.

6.2.1 Overview

As shown in Figure 6.5, HOPE executes in two phases (i.e., Build, Encode) and has four modules: (1) **Symbol Selector**, (2) **Code Assigner**, (3) **Dictionary**, and (4) **Encoder**. A DBMS provides HOPE with a list of sample keys from the search tree. HOPE then produces a Dictionary and an Encoder as its output. We note that the size and representativeness of the sampled key list only affect the compression rate. The correctness of HOPE's compression algorithm is guaranteed by the dictionary completeness and order-preserving properties discussed in Section 6.1.1. In other words, any HOPE dictionary can both encode arbitrary input keys and preserve the original key ordering.

In the first step of the build phase, the Symbol Selector counts the frequencies of the specified string patterns in the sampled key list and then divides the string axis into intervals based on the heuristics given by the target compression scheme. The Symbol Selector generates three outputs for each interval: (1) dictionary symbol (i.e., the common prefix of the interval), (2) interval boundaries, and (3) probability that a source string falls in that interval.

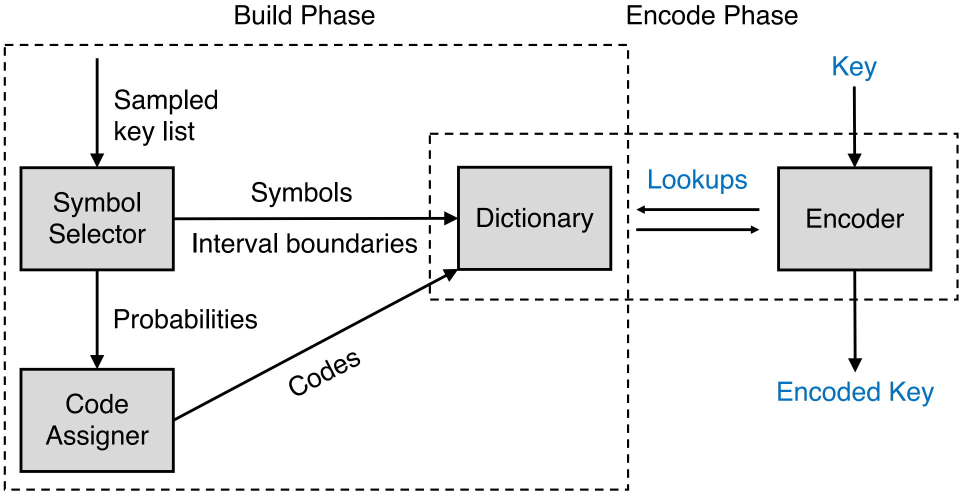


Figure 6.5: The HOPE Framework – An overview of HOPE's modules and their interactions with each other in the two phases.

The framework then gives the symbols and interval boundaries to the Dictionary module. Meanwhile, it sends the probabilities to the Code Assigner to generate codes for the dictionary symbols. If the scheme uses fixed-length codes, the Code Assigner only considers the dictionary size. If the scheme uses variable-length codes, the Code Assigner examines the probability distribution to generate optimal order-preserving prefix codes (i.e., Hu-Tucker codes).

When the Dictionary module receives the symbols, the interval boundaries, and the codes, it selects an appropriate and fast dictionary data structure to store the mappings. The string lengths of the interval boundaries inform the decision; available data structures range from fixed-length arrays to general-purpose tries. The dictionary size is a tunable parameter for VIFC and VIVC schemes. Using a larger dictionary trades performance for a better compression rate.

The encode phase uses only the Dictionary and Encoder modules. On receiving an uncompressed key, the Encoder performs multiple lookups in the dictionary. Each lookup translates a part of the original key to some code as described in Section 6.1.1. The Encoder then concatenates the codes in order and outputs the encoded result. This encoding process is sequential for variable-length interval schemes (i.e., VIFC and VIVC) because the remaining source string to be encoded depends on the results of earlier dictionary lookups.

We next describe the implementation details for each module. Building a decoder and its corresponding dictionary is optional because our target query workload for search trees does not require reconstructing the original keys.

Scheme	Symbol Selector	Code Assigner	Dictionary	Encoder
Single-Char	Single-Char	Hu-Tucker	Array	Fast Encoder
Double-Char	Double-Char			
ALM	ALM	Fixed-Length	ART-Based	
3-Grams	3-Grams	Hu-Tucker	Bitmap-Trie	
4-Grams	4-Grams			
ALM-Improved	ALM-Improved		ART-Based	

Table 6.1: Module Implementations – The configuration of HOPE's six compression schemes.

6.2.2 Implementation

HOPE users can create new compression schemes by combining different module implementations. HOPE currently supports the six compression schemes described in Section 6.1.3. For Symbol Selector and Code Assigner, the goal is to generate a dictionary that leads to the maximum compression rate. We no longer need these two modules after the build phase. We spend extra effort optimizing the Dictionary and Encoder modules because they are on the critical path of every search tree query.

Symbol Selector: It first counts the occurrences of substring patterns in the sampled keys using a hash table. For example, 3-Grams collects frequency statistics for all three-character substrings. The ALM, however, considers substrings of all lengths. For Single-Char and Double-Char, the interval boundaries are implied because they are fixed-length-interval schemes (i.e., FIVC). For the remaining schemes, the Symbol Selectors divide intervals using the algorithms described in Section 6.1.3: first identify the most frequent symbols and then fill the gaps with new intervals.

The ALM and ALM-Improved Symbol Selectors require an extra *blending* step before their interval-division algorithms. This is because the selected variable-length substrings may not satisfy the *prefix property* (i.e., a substring can be a prefix of another substring). For example, "`sig`" and "`sigmod`" may both appear in the frequency list, but the interval-division algorithm cannot select both of them because the two intervals on the string axis are not disjoint: "`sigmod`" is a sub-interval of "`sig`". A blending algorithm redistributes the occurrence count of a prefix symbol to its longest extension in the frequency list [50]. We implement this blending algorithm in HOPE using a trie data structure.

After the Symbol Selector decides the intervals, it performs a test encoding of the keys in the sample list using the intervals as if the code for each interval has been assigned. The purpose of this step is to obtain the probability that a source string (or its remaining suffix after certain encoding steps) falls into each interval so that the Code Assigner can generate codes based on those probabilities to maximize compression. For variable-length-interval schemes, the probabilities are weighted by the symbol lengths of the intervals.

Code Assigner: Assume that the Code Assigner receives N probabilities from the Symbol Selector. To assign fixed-length codes, the Code Assigner outputs monotonically increasing integers $0, 1, 2, \cdots, N - 1$, each using $\lceil \log_2 N \rceil$ bits. For variable-length codes, HOPE uses the Hu-Tucker algorithm to generate optimal order-preserving prefix codes. One could use an alternative method, such as *Range Encoding* [125] (i.e., the integer version of Arithmetic Encoding). Range Encoding, however, requires more bits than Hu-Tucker to ensure that codes are exactly on range boundaries to guarantee order-preserving.

The Hu-Tucker algorithm works in four steps. First, it creates a leaf node for each probability received from the Symbol Selector and then lists the leaf nodes in interval order. Second, it selects the two least-frequent nodes and merges them to create a new internal node. This new node takes the place of the existing left node. Unlike the Huffman algorithm, Hu-Tucker allows two nodes to merge only if there are no leaf nodes between them. This is where the algorithm guarantees order. After constructing this probability tree, it computes the depth of each leaf node to derive the lengths of the codes. Finally, the algorithm constructs a tree by adding these leaf nodes level-by-level starting from the deepest and then connecting adjacent nodes at the same level in pairs. HOPE uses this Huffman-tree-like structure to extract the final codes. Our Hu-Tucker implementation in the Code Assigner uses an improved algorithm that runs in $\mathcal{O}(N^2)$ time [165].

Dictionary: A dictionary in HOPE maps an interval (and its symbol) to a code. Because the intervals are connected and disjoint, the dictionary needs to store only the left boundary of each interval as the key. A key lookup in the dictionary then is a “greater than or equal to” index query to the underlying data structure. For the values, we store only the codes along with the lengths of the symbols to determine the number of characters from the source string that we have consumed at each step.

We implemented three dictionary data structures in HOPE. The first is an *array* for the Single-Char and Double-Char schemes. Each dictionary entry includes an 8-bit integer to record the code length and a 32-bit integer to store the code. The dictionary symbols and the interval left boundaries are implied by the array offsets. For example, the 97^{th} entry in Single-Char has the symbol a, while the 24770^{th} entry in Double-Char corresponds to the symbol aa[3]. A lookup in an array-based dictionary is fast because it requires only a

[3] $24770 = 96 \times (256+1) + 97 + 1$. The $+1$’s are because of the terminator character $\varnothing$. See Section 6.1.3.

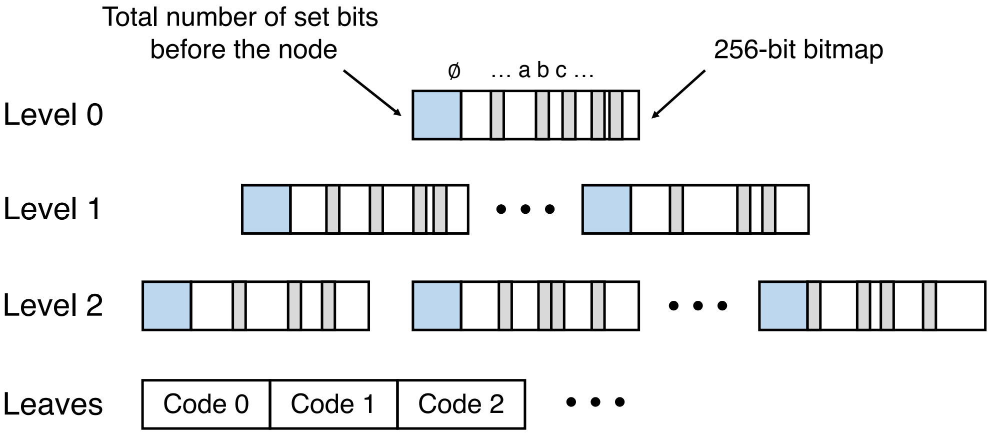


Figure 6.6: 3-Grams Bitmap-Trie Dictionary – Each node consists of a 256-bit bitmap and a counter. The former records the branches of the node and the latter represents the total number of set bits in the bitmaps of all the preceding nodes.

single memory access and the array fits in CPU cache.

The second dictionary data structure in HOPE is a *bitmap-trie* used by the 3-Grams and 4-Grams schemes. Figure 6.6 depicts the structure of a three-level bitmap-trie for 3-Grams. The nodes are stored in an array in breadth-first order. Each node consists of a 32-bit integer and a 256-bit bitmap. The bitmap records all the branches of the node. For example, if the node has a branch labeled a, the 97^{th} bit in the bitmap is set. The integer at the front stores the total number of set bits in the bitmaps of all the preceding nodes. Since the stored interval boundaries can be shorter than three characters, the data structure borrows the most significant bit from the 32-bit integer to denote the termination character $\varnothing$. In other words, $\varnothing$ is the first bit of the 257-bit bitmap in a node.

Given a node (n, bitmap) where n is the count of the preceding set bits, its child node pointed by label l is at position $n + \mathit{popcount}(\mathit{bitmap}, l)$[4] in the node array. Our evaluation shows that looking up a bitmap-trie is 2.3× faster than binary-searching the dictionary entries because it require fewer memory probes and has better cache performance.

Finally, we use an ART-based dictionary to serve the ALM and ALM-Improved schemes. ART is a radix tree that supports variable-length keys [112]. We modified three aspects of ART to make it more suitable as a dictionary. First, we added support for *prefix keys* in ART. This is necessary because both abc and abcd, for example, can be valid interval boundaries stored in a dictionary. We also disabled ART's *optimistic common prefix skipping* that compresses paths on single-branch nodes by storing only the first

[4] The POPCOUNT CPU instruction counts the set bits in a bit-vector. The function $\mathit{popcount}(\mathit{bitmap}, l)$ counts the set bits up to position l in *bitmap*.

few bytes. If a corresponding segment of a query key matches the stored bytes during a lookup, ART assumes that the key segment also matches the rest of the common prefix (a final key verification happens against the full tuple). HOPE's ART-based dictionary, however, stores the full common prefix for each node since it cannot assume that there is a tuple with the original key. Lastly, we modified the ART's leaf nodes to store the dictionary entries instead of tuple pointers.

Encoder: HOPE looks up the source string in the dictionary to find an interval that contains the string. The dictionary returns the symbol length L and the code C. HOPE then concatenates C to the result buffer and removes the prefix of length L that matches the symbol from the source string. It repeats this process on the remaining string until it is empty.

To make the non-byte-aligned code concatenation fast, HOPE stores codes in 64-bit integer buffers. It adds a new code to the result buffer in three steps: (1) left-shift the result buffer to make room for the new code; (2) write the new code to the buffer using a bit-wise OR instruction; (3) split the new code if it spans two 64-bit integers. This procedure costs only a few CPU cycles per code concatenation.

When encoding a batch of sorted keys, the Encoder optimizes the algorithm by first dividing the batch into blocks, where each block contains a fixed number of keys. The Encoder then encodes the common prefix of the keys within a block only once, avoiding redundant work. When the batch size is two, we call this optimization pair-encoding. Compared to encoding keys individually, pair-encoding reduces key compression overhead for the range queries in a search tree. We evaluate batch encoding in Section 6.4.4.

6.3 Integration

Integrating HOPE in a DBMS is a straightforward process because we designed it to be a standalone library that is independent of the target search tree data structure and with zero external dependencies.

When the DBMS creates a new search tree, HOPE samples the initial bulk-loaded keys to construct the dictionary (i.e., the build phase). Once HOPE creates the Dictionary and Encoder modules, every query for that tree, including the initial bulk-inserts, must go through the Encoder first to compress the keys. If the search tree is initially empty, HOPE samples keys as the DBMS inserts them into the tree. It then rebuilds the search tree using the compressed keys once it sees enough samples. We use a small sample size because it guarantees fast tree rebuild, and it does not compromise the compression rate, as shown in Section 6.4.1.

We typically invoke HOPE's Build Phase only once because switching dictionaries causes the search tree to rebuild, which is particularly expensive for large trees. Our assumption is that the value distribution in a database column is relatively stable, especially at the substring level. For example, "`@gmail.com`" is likely a common pattern for emails. Because HOPE exploits common patterns at relatively fine granularity, its dictionary remains effective in compressing keys over time. We evaluated HOPE under a dramatic key distribution change in Section 6.4.5 and observed a compression rate decreases as expected, with simpler schemes such as Single-Char less affected. Even if a dramatic change in the key distribution happens, HOPE is not required to rebuild immediately because it still guarantees query correctness. The system can schedule the reconstruction during maintenance to regain the compression rate.

We applied HOPE to five in-memory search trees used in today's DBMSs:

- **SuRF**: The Succinct Range Filter [169] is a trie-based data structure that performs approximate membership tests for ranges. SuRF uses succinct data structures [38] to achieve an extremely small memory footprint.
- **ART**: The Adaptive Radix Tree [112, 113] is the default index structure for HyPer [103]. ART adaptively selects variable-sized node layouts based on fanouts to save space and to improve cache performance.
- **HOT**: The Height Optimized Trie [60] is a fast and memory-efficient index structure. HOT guarantees high node fanouts by combining nodes across trie levels.
- **B+tree**: We use the cache-optimized TLX B+tree [43] (formerly known as STX). TLX B+tree stores variable-length strings outside the node using reference pointers. The default node size is 256 bytes, making a fanout of 16 (8-byte key pointer and 8-byte value pointer per slot).
- **Prefix B+tree**: A Prefix B+tree [55] optimizes a plain B+tree by applying prefix and suffix truncation to the nodes [89]. A B+tree node with prefix truncation stores the common prefix of its keys only once. During a leaf node split, suffix truncation allows the parent node to choose the shortest string qualified as a separator key. We implemented both techniques on a state-of-the-art B+tree [14, 114] other than TLX B+tree for better experimental robustness.

HOPE provides the most benefit to search trees that store the full keys. Many tree indexes for in-memory DBMSs, such as ART and HOT, only store partial keys to help the DBMS find the record IDs. They then verify the results against the full keys after fetching the records because the step is as cheap as accessing index nodes. To understand HOPE's interaction with these different search trees, we arrange them in Figure 6.7 according to how large a part of the keys they store. The B+tree is at one extreme where the data structure stores full keys. At the other extreme sits the T-Tree [111] (or simply a sorted list of record IDs) where no keys appear in the data structure. Prefix B+tree, SuRF, ART, and HOT fall in the middle. HOPE is more effective towards the B+tree side, especially

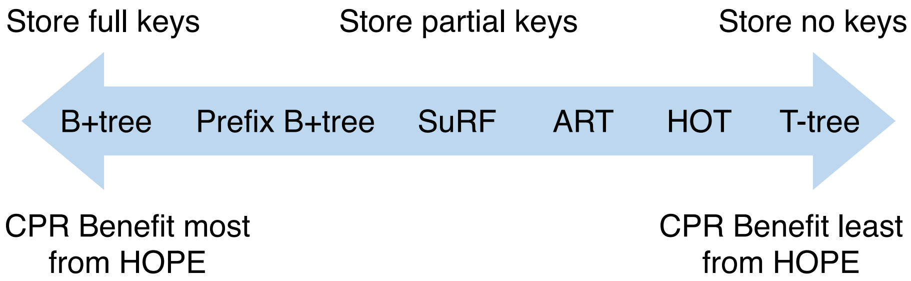


Figure 6.7: Search Tree on Key Storage – B+tree, Prefix B+tree, SuRF, ART, HOT, and T-Tree get decreasing benefits from HOPE, especially in terms of compression rate (CPR), as the completeness of key storage goes down.

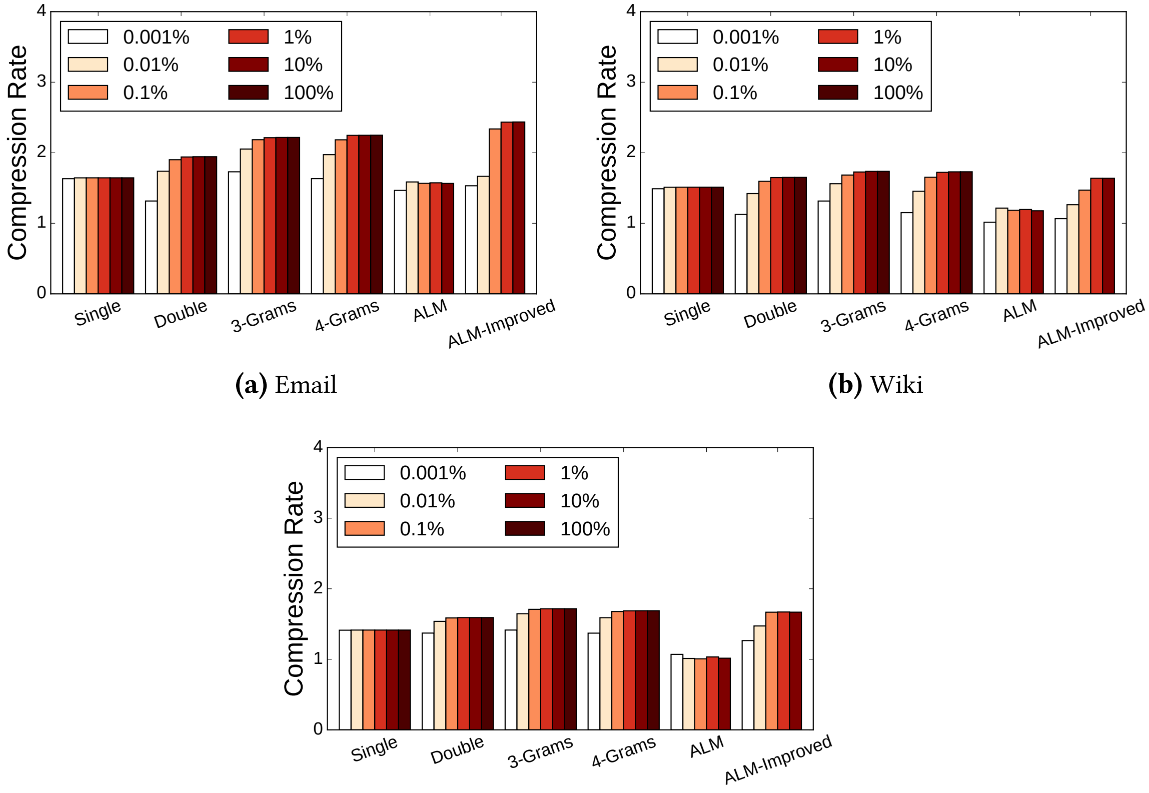
in terms of compression rate. The query latency improvement is the difference between the speedup due to shorter keys and the overhead of key compression. For the B+tree family, shorter keys means larger fanouts and faster string comparisons. Although tries only store partial keys, HOPE improves their performance by reducing the tree height. The rest of this section analyzes the latency reduction of using HOPE on a trie.

Let l denote the average key length and *cpr* denote the compression rate (i.e., uncompressed length / compressed length). The average height of the original *trie* is h. We use t_{trie} to denote the time needed to walk one level (i.e., one character) down the *trie*, and t_{encode} to denote the time needed to compress one character in HOPE.

The average point query latency in the original *trie* is $h \times t_{trie}$, while this latency in the compressed *trie* is $l \times t_{encode} + \frac{h}{cpr} \times t_{trie}$, where $l \times t_{encode}$ represents the encoding overhead. Therefore, the percentage of latency reduction is:

$$\frac{h \times t_{trie} - (l \times t_{encode} + \frac{h}{cpr} \times t_{trie})}{h \times t_{trie}} = 1 - \frac{1}{cpr} - \frac{l \times t_{encode}}{h \times t_{trie}}$$

If the expression > 0, we improve performance. For example, when evaluating SuRF on the email workload in Section 6.5, the average key length l is 21.2 bytes. The original SuRF has the average trie height $h = 18.2$, and has an average point query latency of $1.46\mu s$. t_{trie} is, thus, $\frac{1.46\mu s}{18.2} = 80.2ns$. Our evaluation in Section 6.4 shows that HOPE's Double-Char scheme achieves a compression rate $cpr = 1.94$ and an encoding latency per character $t_{encode} = 6.9ns$. Hence, we estimate that by using Double-Char on SuRF, we can reduce the point query latency by $1 - \frac{1}{1.94} - \frac{21.2 \times 6.9}{18.2 \times 80.2} = 38\%$. The real latency reduction is usually higher (41% in this case as shown in Section 6.5) because smaller tries also improve the cache performance.



(a) Email

(b) Wiki

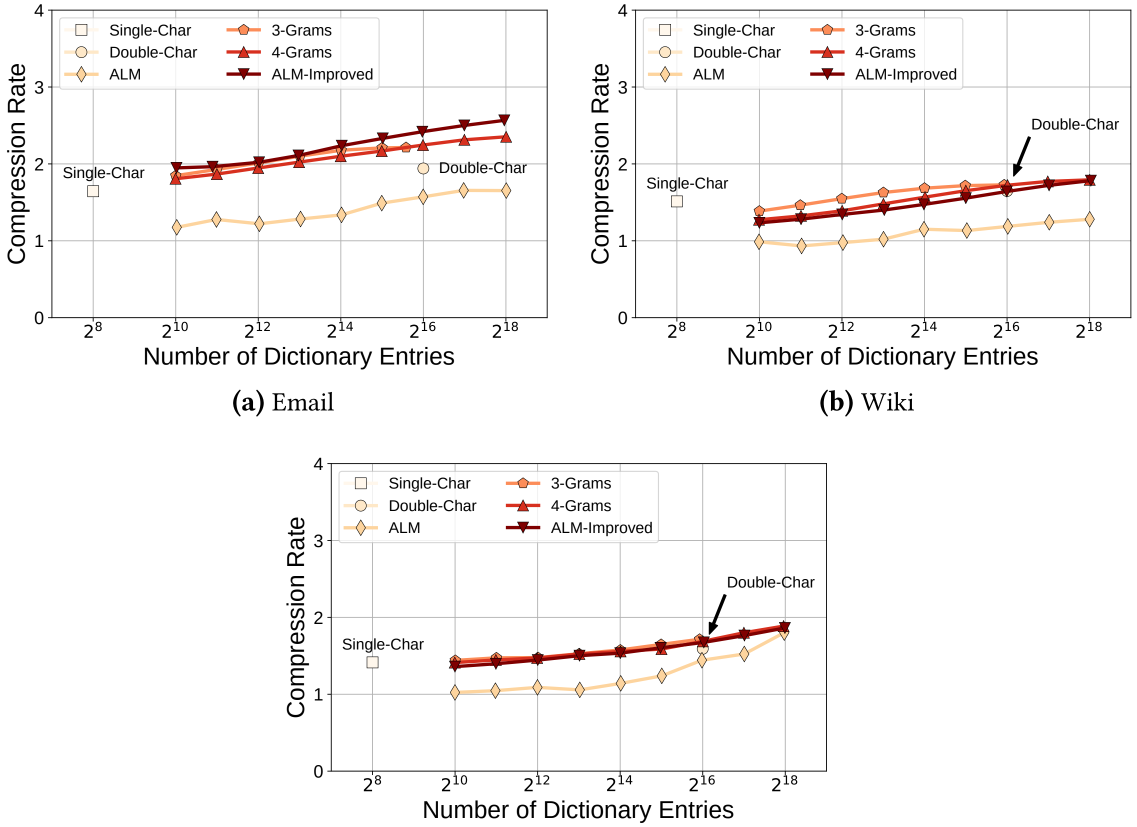
(c) URL

Figure 6.8: Sample Size Sensitivity Test – Compression rate measured under varying sample sizes for all schemes in HOPE. The dictionary size limit is set to 2^{16} (64K) entries.

6.4 HOPE Microbenchmarks

We evaluate HOPE in the next two sections. We first analyze the trade-offs between compression rate and compression overhead of different schemes in HOPE. These microbenchmarks help explain the end-to-end measurements on HOPE-integrated search trees in Section 6.5.

We run our experiments using a machine equipped with two Intel® Xeon® E5-2630v4 CPUs (2.20GHz, 32 KB L1, 256 KB L2, 25.6 MB L3) and 8×16 GB DDR4 RAM. In each experiment, we randomly shuffle the target dataset before each trial. We then select 1% of the entries from the shuffled dataset and use that as the sampled keys for HOPE. Our sensitivity test in Section 6.4.1 shows that 1% is large enough for all schemes to reach their maximum compression rates. We repeat each trial three times and report the average result.



(a) Email

(b) Wiki

(c) URL

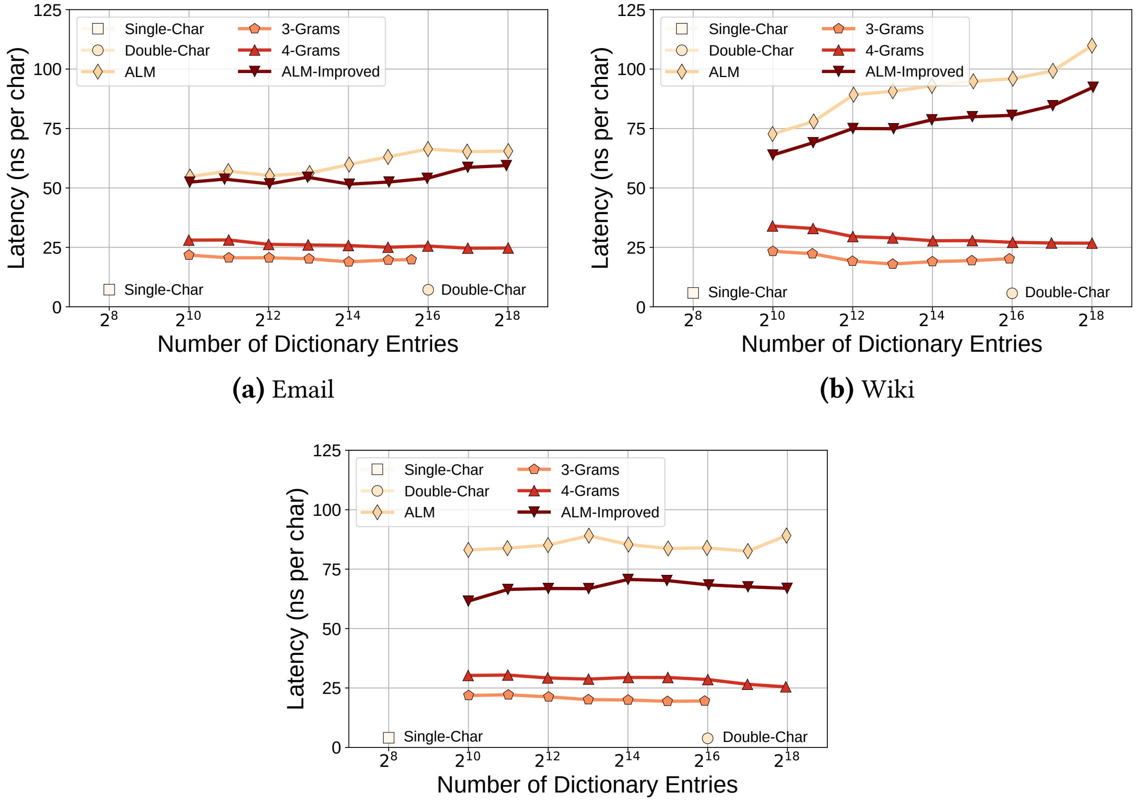
Figure 6.9: Microbenchmarks (CPR) – Compression rate measurements of HOPE's six schemes on the different datasets.

We use the following datasets for all our experiments:

- **Email:** 25 million email addresses (host reversed – e.g., "`com.gmail@foo`") with an average length of 22 bytes.
- **Wiki:** 14 million article titles from the English version of Wikipedia with an average length of 21 bytes [46].
- **URL:** 25 million URLs from a 2007 web crawl with an average length of 104 bytes [44].

6.4.1 Sample Size Sensitivity Test

We first perform a sensitivity test on how the size of the sampled key list affects HOPE's compression rate. We use the three datasets (i.e., Email, Wiki, and URL) introduced above. We first randomly shuffle the dataset and then select the first $x\%$ of the entries as the



(a) Email

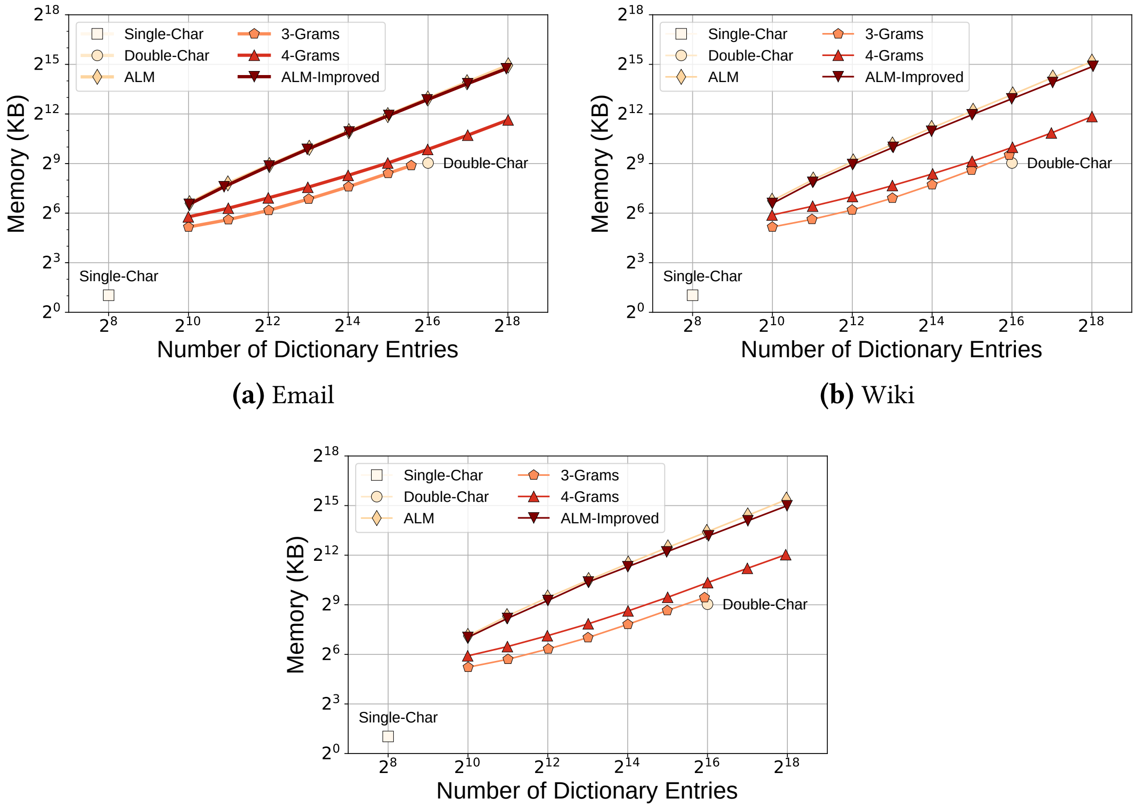
(b) Wiki

(c) URL

Figure 6.10: Microbenchmarks (Latency) – Compression latency measurements of HOPE's six schemes on the different datasets.

sampled keys for HOPE. We set x to 0.001, 0.01, 0.1, 1, 10, and 100, which translates to 250, 2.5K, 25K, 250K, 2.5M, and 25M samples for the Email and URL datasets, and 140, 1.4K, 14K, 140K, 1.4M, and 14M samples for the Wiki dataset. We measure the compression rate for each scheme in HOPE for each x. We set the dictionary size limit to 2^{16} (64K) entries. Note that for $x = 0.001, 0.01$, schemes such as 3-Grams do not have enough samples to construct the dictionary of the limit size.

Figure 6.8 shows the results. Note that for $x = 100$, the numbers are missing for ALM and ALM-Improved because the experiments did not finish in a reasonable amount of time due to their complex symbol select algorithms. From the figures, we observe that a sample size of 1% of the dataset (i.e., 250K for Email and URL, 140K for Wiki) is large enough for all schemes to reach their maximum compression rates. 1% is thus the default sample size percentage used in all experiments. We also notice that the compression rates for schemes that exploit higher-order entropies are more sensitive to the sample size because these schemes require more context information to achieve better compression. As



(a) Email

(b) Wiki

(c) URL

Figure 6.11: Microbenchmarks (Memory) – Dictionary memory of HOPE's six schemes on the different datasets.

a general guideline, a sample size between 10K and 100K is good enough for all schemes, and we can use a much smaller sample for simpler schemes such as Single-Char.

6.4.2 Performance & Efficacy

In this microbenchmark, we evaluate the runtime performance and compression efficacy of HOPE's six built-in schemes listed in Table 6.1. HOPE compresses the keys one-at-a-time with a single thread. We vary the number of dictionary entries in each trial and measure three facets per scheme: (1) the compression rate, (2) the average encoding latency per character, and (3) the size of the dictionary. We compute the compression rate as the uncompressed dataset size divided by the compressed dataset size. We obtain the average encoding latency per character by dividing the execution time by the total number of bytes in the uncompressed dataset.

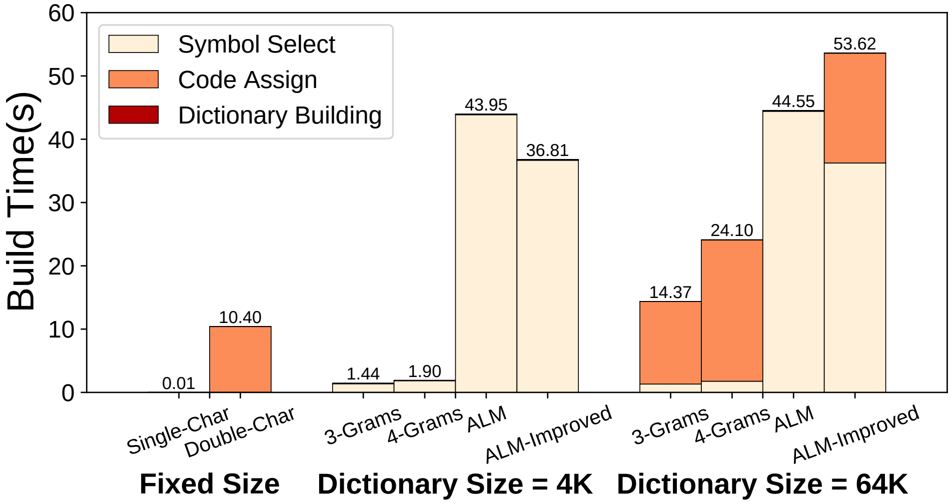


Figure 6.12: Dictionary Build Time – A breakdown of the time it takes for HOPE to build dictionaries on a 1% sample of email keys.

Figures 6.9–6.11 show the experiment results. We vary the number of dictionary entries on the x-axis (log scaled). The Single-Char and Double-Char schemes have fixed dictionary sizes of 2^8 and 2^{16}, respectively. The 3-Grams dictionary cannot grow to 2^{18} because there are not enough unique three-character patterns in the sampled keys.

Compression Rate: Figure 6.9 shows that the VIVC schemes (3-Grams, 4-Grams, ALM-Improved) have better compression rates than the others. This is because VIVC schemes exploit the source strings' higher-order entropies to optimize both interval division and code assignment at the same time. In particular, ALM-Improved compresses the keys more than the original ALM because it uses a better pattern extraction algorithm and the the Hu-Tucker codes. These Hu-Tucker codes improve compression in ALM-Improved because they leverage the remaining skew in the dictionary entries' access probabilities. ALM tries to equalize these weighted probabilities but our improved version has better efficacy. We also note that a larger dictionary produces a better compression rate for the variable-length interval schemes.

Encoding Latency: The latency results in Figure 6.10 demonstrate that the simpler schemes have lower encoding latency. This is expected because the latency depends largely on the dictionary data structures. Single-Char and Double-Char are the fastest because they use array dictionaries that are small enough to fit in the CPU's L2 cache. Our specialized bitmap-tries used by 3-Grams and 4-Grams are faster than the general ART-based dictionaries used by ALM and ALM-Improved because (1) the bitmap speeds up in-node label search; and (2) the succinct design (without pointers) improves cache performance.

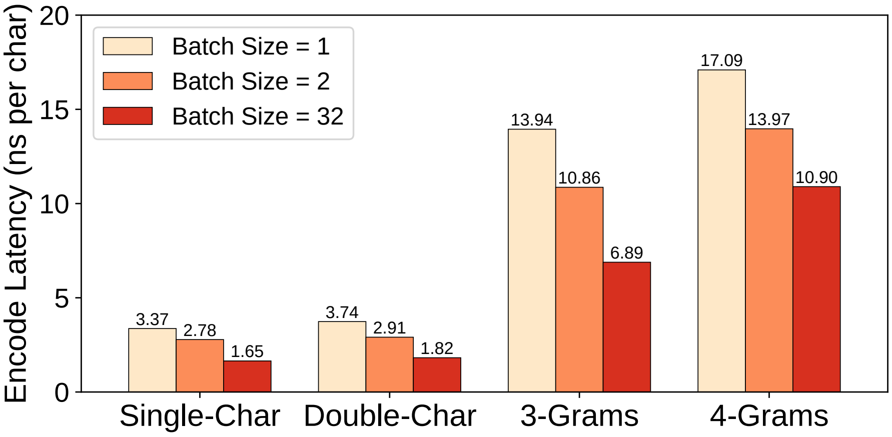


Figure 6.13: Batch Encoding – Encoding latency measured under varying batch sizes on a pre-sorted 1% sample of email keys. The dictionary size is 2^{16} (64K) for 3-Grams and 4-Grams.

The figures also show that latency is stable (and even decrease slightly) in all workloads for 3-Grams and 4-Grams as their dictionary sizes increase. This is interesting because the cost of performing a lookup in the dictionary increases as the dictionary grows in size. The larger dictionaries, however, achieve higher compression rates such that it reduces lookups: larger dictionaries have shorter intervals on the string axis, and shorter intervals usually have longer common prefixes (i.e., dictionary symbols). Thus, HOPE consumes more bytes from the source string at each lookup with larger dictionaries, counteracting the higher per-lookup cost.

Dictionary Memory: Figure 6.11 shows that the dictionary sizes for the variable-length schemes grow linearly as the number of dictionary entries increases. Even so, for most dictionaries, the total tree plus dictionary size is still much smaller than the size of the corresponding uncompressed search tree. These measurements also show that our bitmap-tries for 3-Grams and 4-Grams are up to an order of magnitude smaller than the ART-based dictionaries for all the datasets. The 3-Grams bitmap-trie is only $1.4\times$ larger than Double-Char's fixed-length array of the same size.

Discussion: Schemes that compress more are slower, except that the original ALM is strictly worse than the other schemes in both dimensions. The latency gaps between schemes are generally larger than the compression rate gaps. We evaluate this trade-off in Section 6.5 by applying the HOPE schemes to in-memory search trees.

6.4.3 Dictionary Build Time

We next measure how long HOPE takes to construct the dictionary using each of the six compression schemes. We record the time HOPE spends in the modules from Section 6.2.2

when building a dictionary: (1) Symbol Selector, (2) Code Assigner, and (3) Dictionary. The last step is the time required to populate the dictionary from the key samples. We present only the Email dataset for this experiment; the results for the other datasets produce similar results and thus we omit them. For the variable-length-interval schemes, we perform the experiments using two dictionary sizes (2^{12}, 2^{16}).

Figure 6.12 shows the time breakdown of building the dictionary in each scheme. First, the Symbol Selector dominants the cost for ALM and ALM-Improved because these schemes collect statistics for substrings of all lengths, which has a super-linear cost relative to the number of keys. For the other schemes, the Symbol Selector's time grows linearly with the number of keys. Second, the time used by the Code Assigner rises dramatically as the dictionary size increases because the Hu-Tucker algorithm has quadratic time complexity. Finally, the Dictionary build time is negligible compared to the Symbol Selector and Code Assigner modules.

6.4.4 Batch Encoding

We also evaluate the batching optimization described in Section 6.2.2. In this experiment, we sort the email dataset and then encode the keys with varying batch sizes (1, 2, 32). As shown in Figure 6.13, batch encoding significantly improves encoding performance because it encodes the common prefix of a batch only once to avoid redundant work. ALM and ALM-Improved schemes do not benefit from batch encoding. Because these schemes have dictionary symbols of arbitrary lengths, we cannot determine a priori a common prefix that is aligned with the dictionary symbols for a batch without encoding them.

6.4.5 Updates and Key Distribution Changes

As discussed in Sections 6.2 and 6.3, HOPE can support key updates without modifying the dictionary because the completeness and order-preserving properties of the String Axis Model (refer to Section 6.1.1) guarantee that any HOPE dictionary can encode arbitrary input keys while preserving the original key ordering. However, a dramatic change in the key distribution may hurt HOPE's compression rate.

To simulate a sudden key distribution change, we divide our email dataset into two subsets (roughly the same size): Email-A and Email-B. Email-A contains all the `Gmail` and `Yahoo` accounts while Email-B has the rest, including accounts from `Outlook`, `Hotmail`, and so on. In the experiments, we build two dictionaries (i.e., Dict-A and Dict-B) using samples from Email-A and Email-B, respectively for each compression scheme in HOPE. We use the different dictionaries to compress the keys in the different datasets and then measure the compression rates.

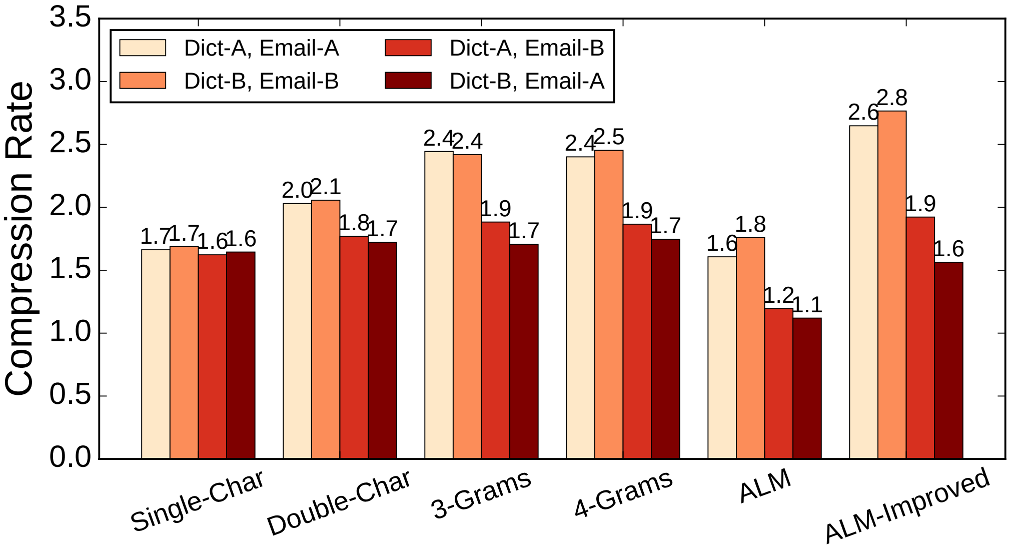


Figure 6.14: Key Distribution Changes – Compression rate measurements under stable key distributions and sudden key pattern changes.

Figure 6.14 shows the results. "Dict-A, Email-A" and "Dict-B, Email-B" represent cases where key distributions are stable, while "Dict-A, Email-B" and "Dict-B, Email-A" simulate dramatic changes in the key patterns. From the figure, we can see that HOPE's compression rate decreases in the "Dict-A, Email-B" and "Dict-B, Email-A" cases. This result is expected because the dictionary built based on earlier samples cannot capture the new common patterns in the new distribution for better compression. We also observe that simpler schemes (i.e., schemes that exploit lower-order entropy) such as Single-Char are less affected by the workload changes. We note that a compression rate drop does not mean that we must rebuild the HOPE-integrated search tree immediately because HOPE still guarantees query correctness. A system can monitor HOPE's compression rate to detect a key distribution change and then schedule an index rebuild to recover the compression rate if necessary.

6.5 Search Tree Evaluation

To experimentally evaluate the benefits and trade-offs of applying HOPE to in-memory search trees, we integrated HOPE into five data structures: SuRF, ART, HOT, B+tree, and Prefix B+tree (as described in Section 6.3). Based on the microbenchmark results in Section 6.4, we evaluate six HOPE configurations for each search tree: (1) Single-Char, (2) Double-Char, (3) 3-Grams with 64K (2^{16}) dictionary entries, (4) 4-Grams with 64K dictionary entries, (5) ALM-Improved with 4K (2^{12}) dictionary entries, and (6) ALM-Improved

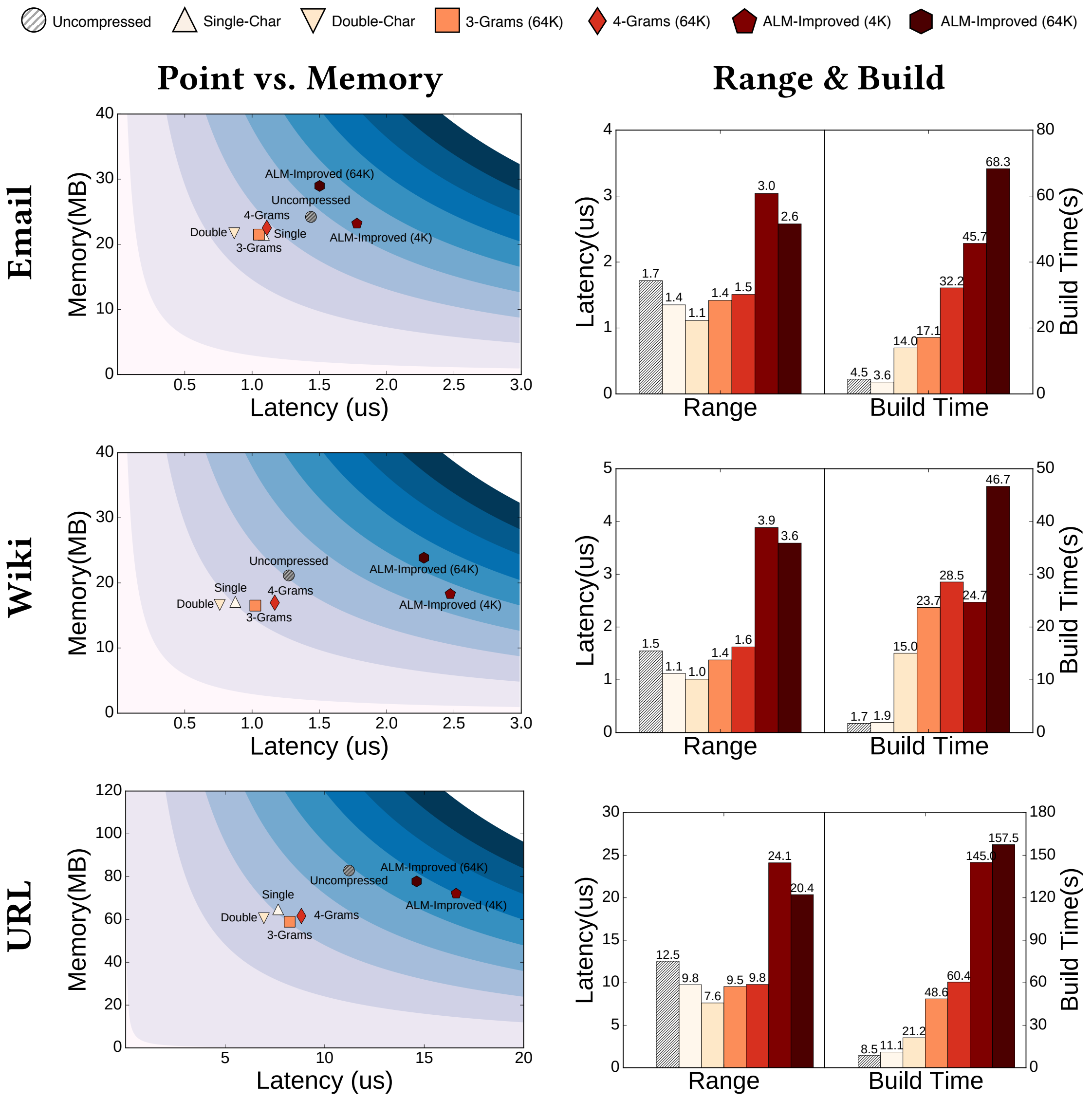
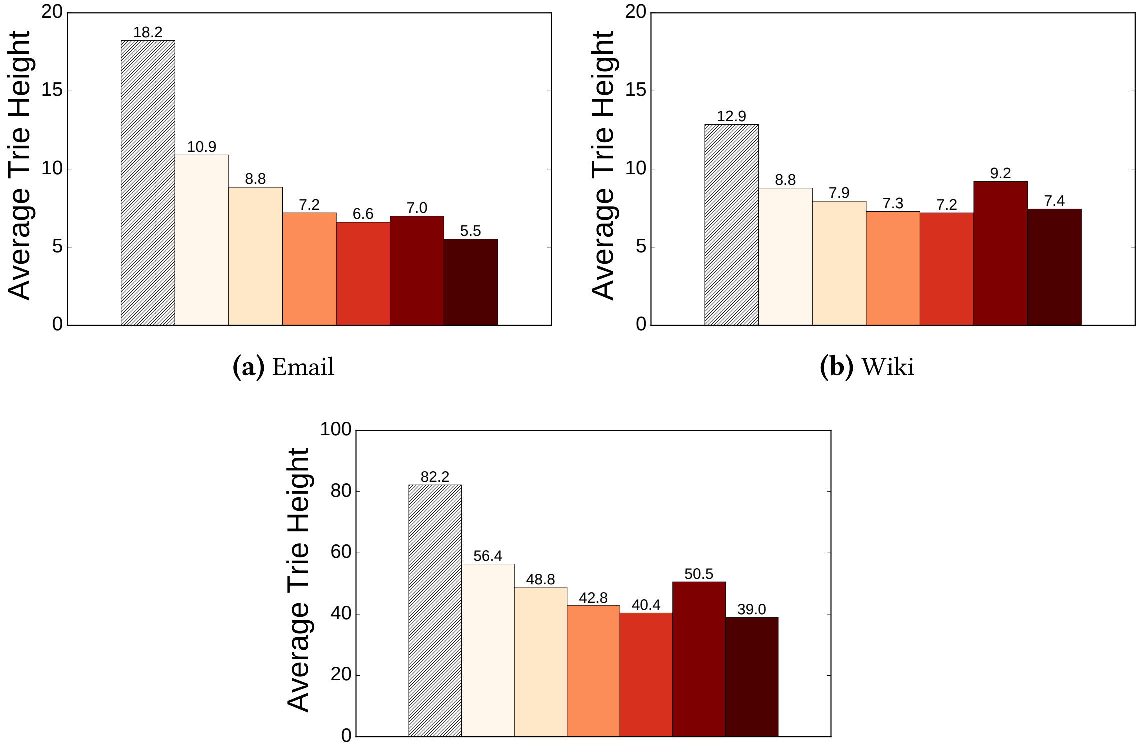


Figure 6.15: SuRF YCSB Evaluation – Runtime measurements for executing YCSB workloads on HOPE-optimized SuRF with three datasets.

with 64K dictionary entries. We include the original uncompressed search trees as baselines (labeled as "Uncompressed"). We choose 64K for 3-Grams, 4-Grams, and ALM-Improved so that they have the same dictionary size as Double-Char. We evaluate an additional ALM-Improved configuration with 4K dictionary size because it has a similar dictionary memory as Double-Char, 3-Grams (64K), and 4-Grams (64K). We exclude the original ALM scheme because it is always worse than the others.



(a) Email

(b) Wiki

(c) URL

Figure 6.16: SuRF Trie Height – the average height of each leaf node after loading all keys

6.5.1 Workload

We use the YCSB-based [70] index-benchmark framework proposed in the Hybrid Index [168] and later used by HOT [60] and SuRF [169]. We use the YCSB workloads **C** and **E** with a Zipf distribution to generate **point** and **range** queries. Point queries are the same for all trees. Each range query for ART, HOT, B+tree, and Prefix B+tree is a start key followed by a scan length. Because SuRF is a filter, its range query is a start key and end key pair, where the end key is a copy of the start key with the last character increased by one (e.g., [“`com.gmail@foo`”, “`com.gmail@fop`”]). We replace the original YCSB keys with the keys in our email, wiki and URL datasets. We create one-to-one mappings between the YCSB keys and our keys during the replacement to preserve the Zipf distribution. We omit the results for other YCSB query distributions (e.g., uniform) because they demonstrate similar performance gains/losses as in the Zipf case when applying HOPE to the search trees.

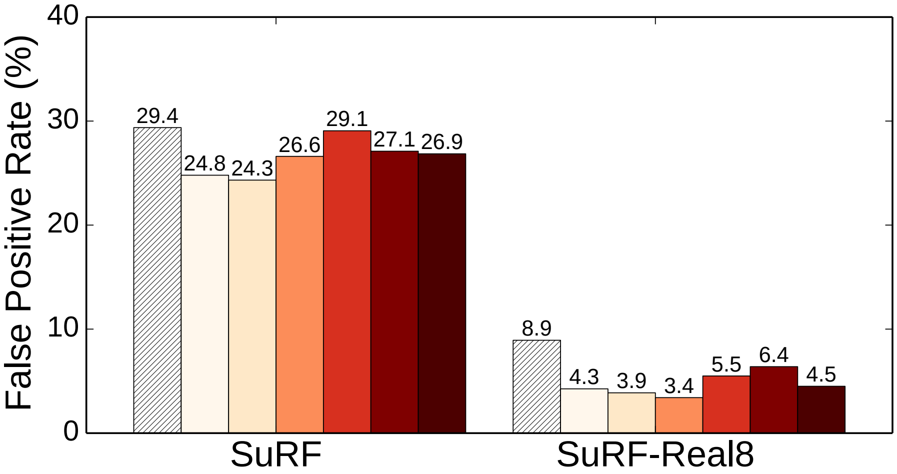


Figure 6.17: SuRF False Positive Rate – Point queries on email keys. SuRF-Real8 means it uses 8-bit real suffixes.

6.5.2 YCSB Evaluation

We start each experiment with the building phase using the first 1% of the dataset's keys. Next, in the loading phase, we insert the keys one-by-one into the tree (except for SuRF because it only supports batch loading). Finally, we execute 10M queries on the compressed keys with a single thread using a combination of point and range queries according to the workload. We obtain the point, range, and insert query latencies by dividing the corresponding execution time by the number of queries. We measure memory consumption (HOPE size included) after the loading phase.

The results for the benchmarks are shown in Figures 6.15–6.21. We first summarize the high-level observations and then discuss the results in more detail for each tree.

High-Level Observations: First, in most cases, multiple schemes in HOPE provide a *Pareto improvement* to the search tree's performance and memory-efficiency. Second, the simpler FIVC schemes, especially Double-Char, stand out to provide the best trade-off between query latency and memory-efficiency for the search trees. Third, more sophisticated VIVC schemes produce the lowest search tree memory in some cases. We believe that compared to Double-Char, however, their small additional memory reduction does not justify the significant performance loss in general.

SuRF: The heatmaps in the first column of Figure 6.15 show the point query latency vs. memory trade-offs made by SuRFs with different HOPE configurations. We define a cost function $C = L \times M$, where L represents latency, and M represents memory. This cost function assumes a balanced performance-memory trade-off. We draw the equi-cost curves (as heatmaps) where points on the same curve have the same cost.

HOPE reduces SuRF's query latencies by up to 41% in all workloads with Single-Char, Double-Char, 3-Grams, and 4-Grams encoders. This is because compressed keys generate shorter tries, as shown in Figure 6.16. According to our analysis in Section 6.3, the performance gained by fewer levels in the trie outweighs the key encoding overhead. Although SuRF with ALM-Improved (64K) has the lowest trie height, it suffers high query latency because encoding is slow for ALM-Improved schemes (refer to Figure 6.10).

Although the six HOPE schemes under test achieve compression rates of 1.5–2.5× in the microbenchmarks, they only provide $\sim$30% memory savings to SuRF. The reason is that compressing keys only reduces the number of internal nodes in a trie (i.e., shorter paths to the leaf nodes). The number of leaf nodes, which is often the majority of the storage cost, stays the same. SuRF with ALM-Improved (64K) consumes more memory than others because of its large dictionary.

The results for SuRF with ALM-Improved (4K) are interesting. For email keys, Section 6.4.2 showed that ALM-Improved (4K) achieves a better compression rate than Double-Char with a similar-sized dictionary. When we integrate this scheme into SuRF, however, the memory saving is smaller than Double-Char even though it produces a shorter trie. Although this seems counterintuitive, it is because ALM-Improved allows dictionary symbols to have arbitrary lengths and it favors long symbols. Encoding long symbols one-at-a-time can prevent prefix sharing. As an example, ALM-Improved may treat the keys "`com.gmail@c`" and "`com.gmail@s`" as two separate symbols and thus have completely different codes.

All schemes, except for Single-Char, add computational overhead in building SuRF. The dictionary build time grows quadratically with the number of entries because of the Hu-Tucker algorithm. One can reduce this overhead by shrinking the dictionary size, but this diminishes performance and memory-efficiency gains.

Finally, the HOPE-optimized SuRF achieves lower false positive rate under the same suffix-bit configurations, as shown in Figure 6.17. This is because each bit in the compressed keys carries more information and is, thus, more distinguishable than a bit in the uncompressed keys.

ART, HOT: Figures 6.18 and 6.19 show that HOPE improves ART and HOT's performance and memory-efficiency for similar reasons as for SuRF because they are also trie-based data structures. Compared to SuRF, however, the amount of improvement for ART and HOT is less. This is for two reasons. First, ART and HOT include a 64-bit value pointer for each key, which dilutes the memory savings from the key compression. More importantly, as described in Section 6.2.2 and Section 6.3, ART and HOT only store partial keys using *optimistic common prefix skipping* (OCPS). HOT is more optimistic than ART as it only stores the *branching points* in a trie (i.e., the minimum-length partial keys needed to uniquely map a key to a value pointer). Although OCPS can incur false positives, the DBMS will verify the match when it retrieves the tuple. Therefore, since ART

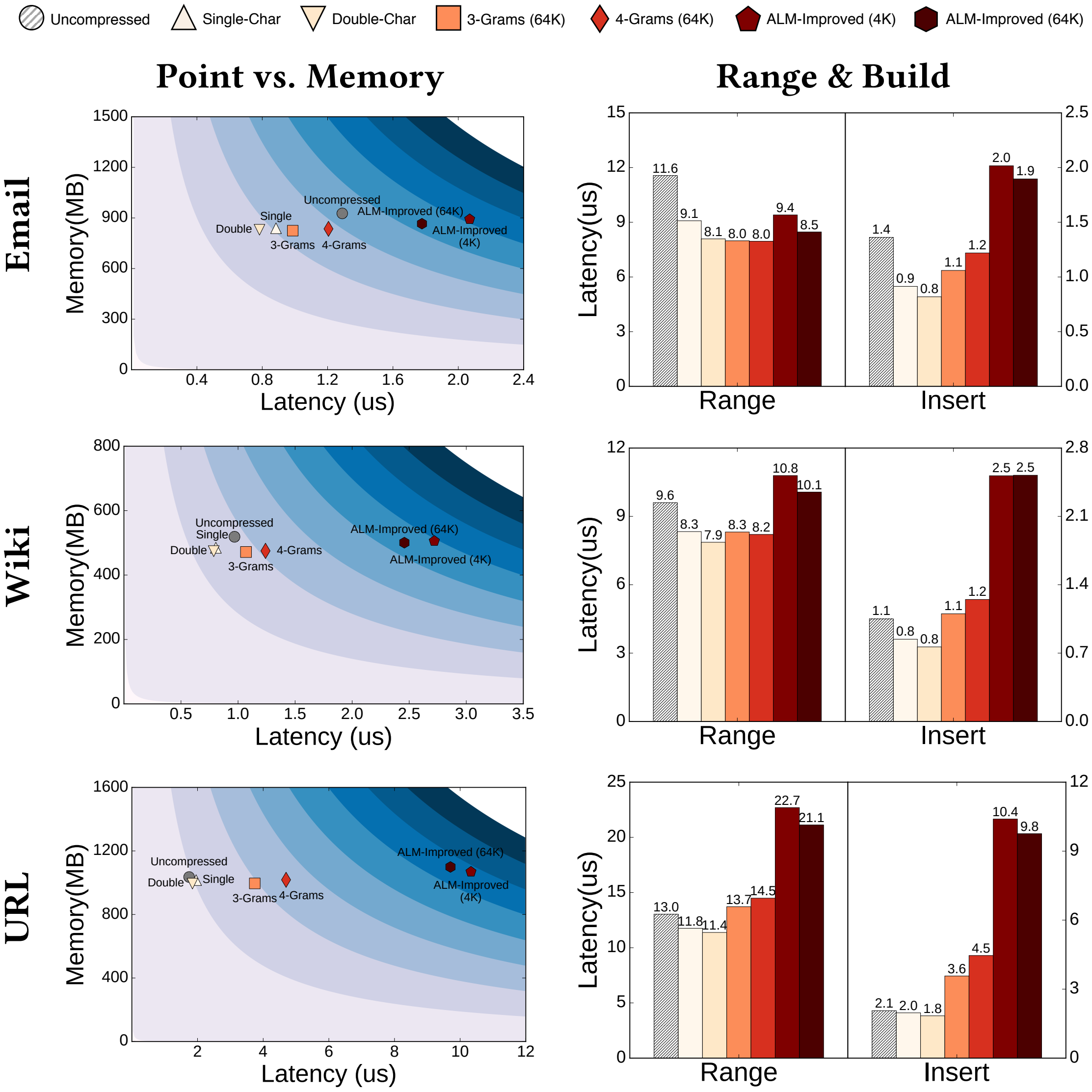


Figure 6.18: ART YCSB Evaluation – Runtime measurements for executing YCSB workloads on HOPE-optimized ART with three datasets.

and HOT store partial keys, they do not take full advantage of key compression. The portion of the URL keys skipped by ART is large because they share long prefixes. Nevertheless, our results show that HOPE still provides some benefit and thus is worth using in both of these data structures.

B+tree, Prefix B+tree: The results in Figures 6.20 and 6.21 show that HOPE is beneficial to search trees beyond tries. Because the TLX B+tree uses reference pointers to store variable-length string keys outside of each node, compressing the keys does not change

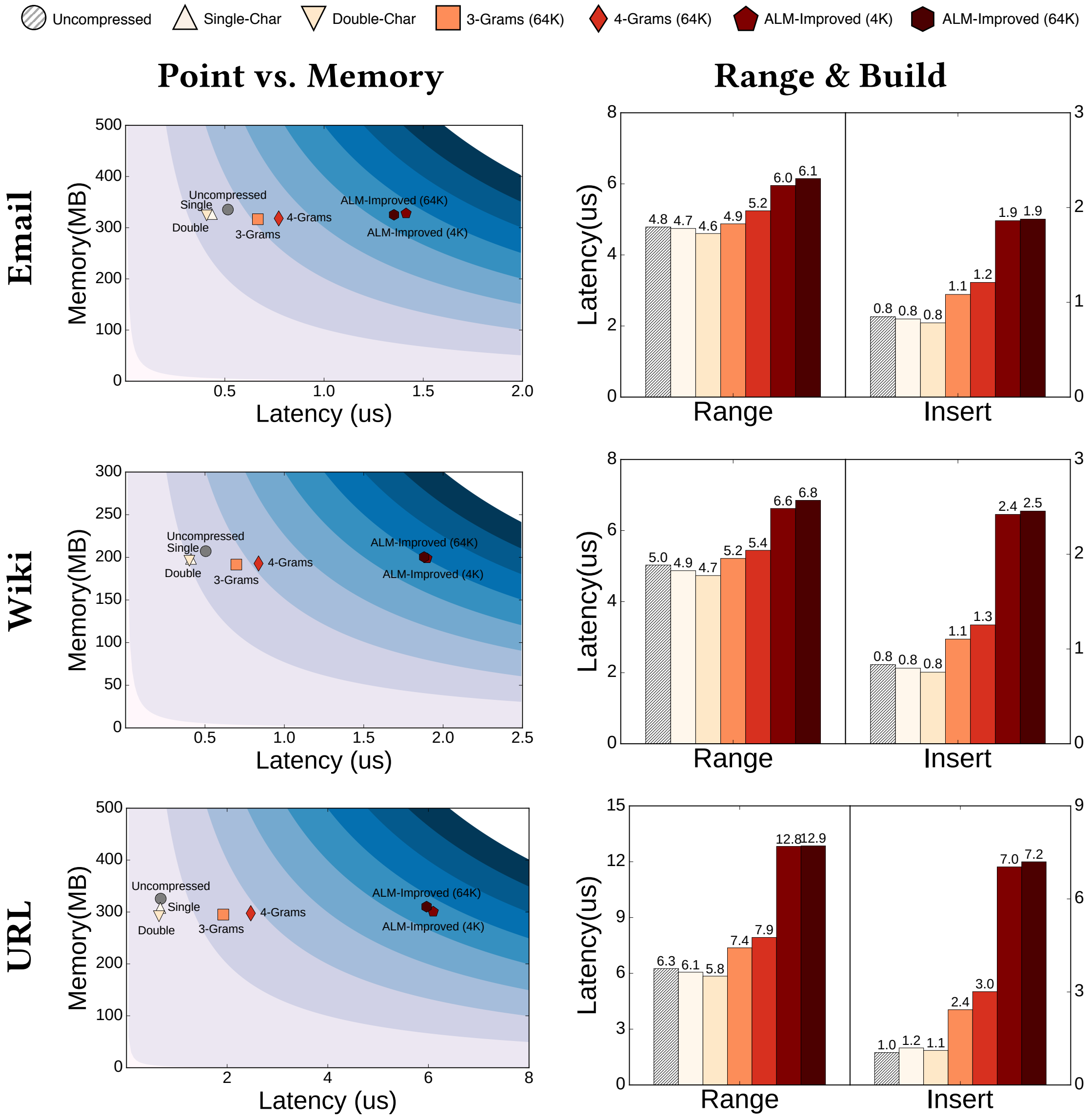


Figure 6.19: HOT YCSB Evaluation – Runtime measurements for executing YCSB workloads on HOPE-optimized HOT with three datasets.

the tree structure. In addition to memory savings, the more lightweight HOPE schemes (Single-Char and Double-Char) also improve the B+tree's query performance because of faster string comparisons and better cache locality. To validate this assumption, we reran the point-query workload on email keys and used `cachegrind` [23] to measure cache misses. We found that Double-Char on TLX B+tree reduces the L1 and last-level cache misses by 34% and 41%, respectively.

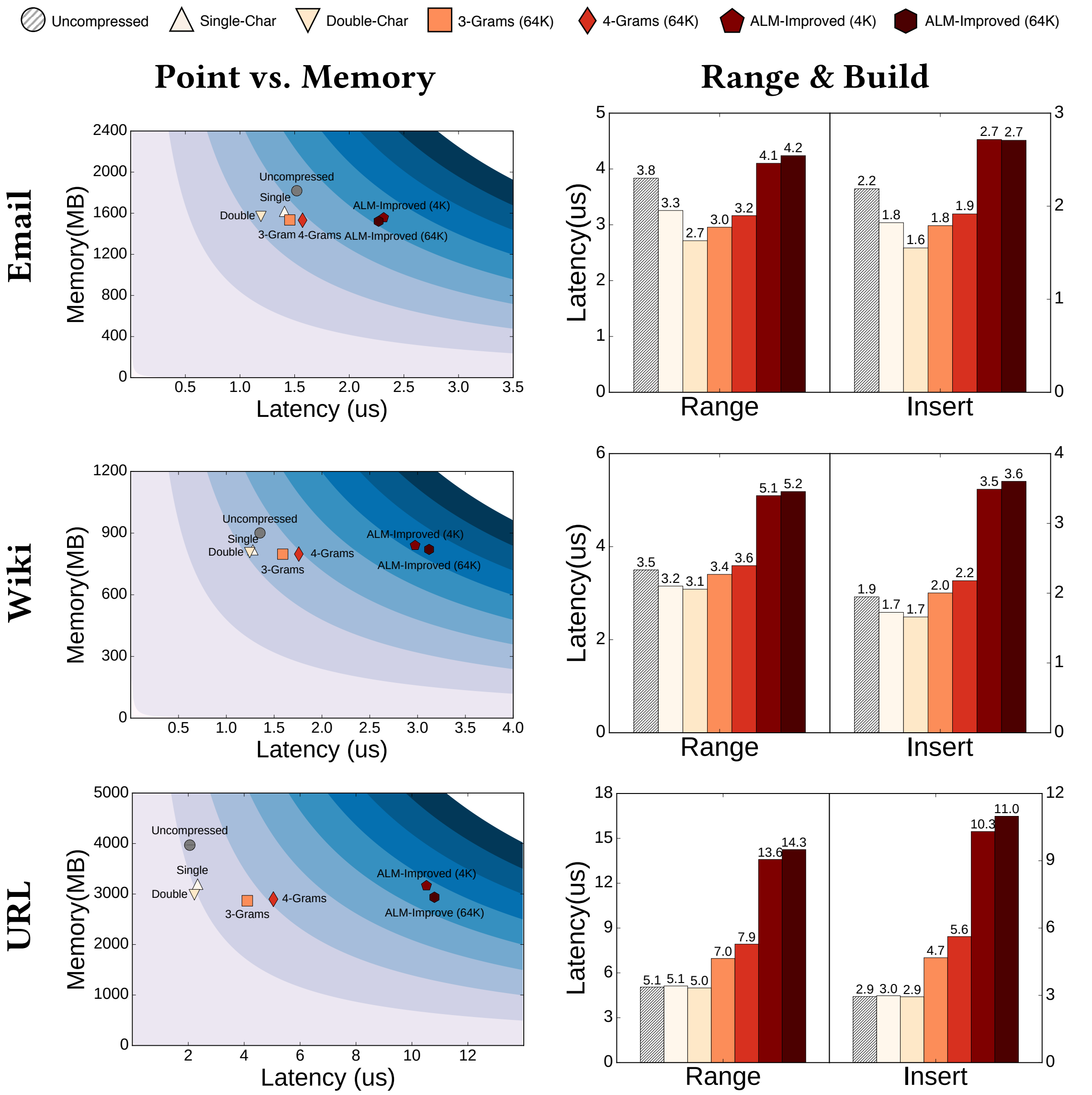


Figure 6.20: B+tree YCSB Evaluation – Runtime measurements for executing YCSB workloads on HOPE-optimized B+tree with three datasets.

Compared to plain B+trees, we observe smaller memory saving percentages when using HOPE on Prefix B+trees. This is because prefix compression reduces the storage size for the keys, and thus making the structural components of the B+tree (e.g., pointers) relatively larger. Although HOPE provides similar compression rates when applied to a Prefix B+tree compared to a plain B+tree, the percentages of space reduction brought by HOPE-compressed keys in a Prefix B+tree is smaller with respect to the entire data

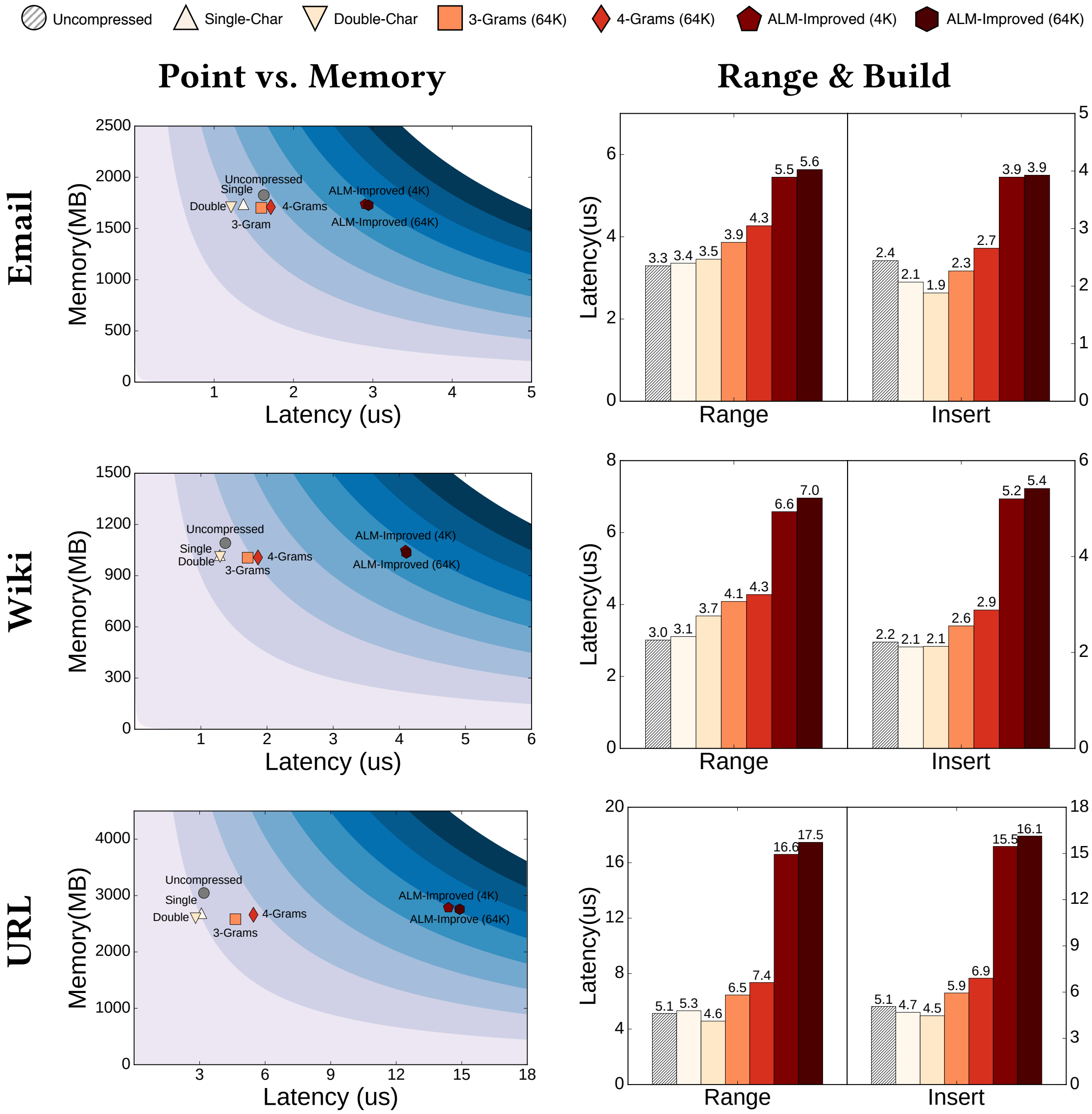


Figure 6.21: Prefix B+tree YCSB Evaluation – Runtime measurements for executing YCSB workloads on HOPE-optimized Prefix B+tree with three datasets.

structure size.

As a final remark, HOPE still improves the performance and memory for highly-compressed trees such as SuRF. It shows that HOPE is orthogonal to many other compression techniques and can benefit a wide range of data structures.

Chapter 7

Related Work

The overhead of managing disk-resident data has given rise to a new class of OLTP DBMSs that store the entire database in main memory [73, 74, 82, 155]. These systems outperform traditional disk-oriented DBMSs because they eschew the legacy components that manage data stored on slow, block-based storage [93]. 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 [154]. 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.

Indexes are a major factor in the memory footprint of a database. OLTP applications often maintain several indexes per table to ensure that queries execute quickly. This is important in applications that interact with users and other external systems where transactions must complete in milliseconds or less [155]. These indexes consume a significant fraction of the total memory used by a database. Designing memory-efficient indexes is thus important for improving database performance and reducing costs. Achieving space-efficient indexes is, however, non-trivial because there are trade-offs between function, performance, and space. For example, hash tables are fast and potentially more space-efficient than tree-based data structures, but they do not support range queries, which prevents them from being ubiquitous. We now discuss prior work related to the concepts and techniques introduced in this thesis.

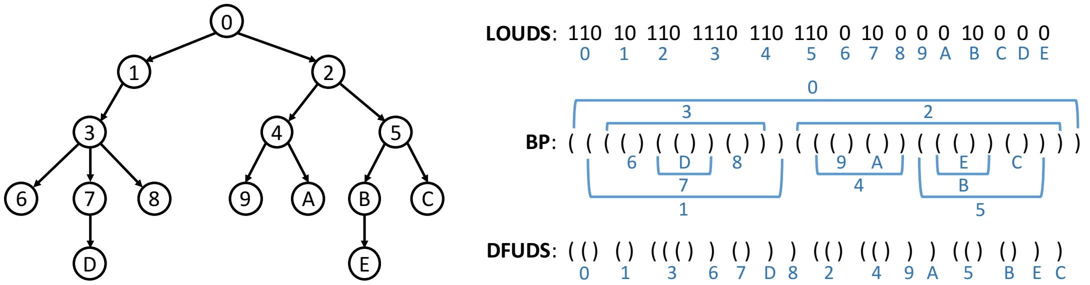

0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
LOUDS: 110 10 110 1110 110 110 0 10 0 0 0 10 0 0 0
0 1 2 3 4 5 6 7 8 9 A B C D E
BP: (((( ) (( )) ( ))) ((( ) ( )) ((( )) ( ))))
0
3
2
6
D
8
7
1
9
A
4
E
C
B
5
DFUDS: (( ) ( ) ((( )) ( ))) (( ) (( ))) (( ) ( )))
0 1 3 6 7 D 8 2 4 9 A 5 B E C

Figure 7.1: Succinct Tree Representations – An example ordinal tree encoded using three major succinct representations: LOUDS, BP, and DFUDS.

7.1 Succinct Tree Representations

Besides the Level-Ordered Unary Degree Sequence (LOUDS) discussed in Chapter 3, a second type of succinct tree representation is based on "balanced parentheses" (BP) sequences [131]. As shown in Figure 7.1, BP traverses the nodes in depth-first order and appends to the sequence an opening parenthesis when a node is first visited and a closing parenthesis after the entire subtree is covered. The BP representations support a wider range of tree operations in constant time than LOUDS [85, 121, 132, 134]. The most recent "fully functional" representation [150] reduces the conceptual and implementational complexity of BP by reducing the tree operations to primitives that give the difference between the opening and closing parentheses at a certain position. Although BP supports more functions, it is more complex and is slower than LOUDS for the simple "move-to-child" and "more-to-parent" navigations that are essential to FST [52]. Moreover, developing an efficient trie representation from BP is difficult because child labels of the same node have poor locality.

Many state-of-the-art succinct tries [57, 91, 145] are based on a third type of succinct tree representation that combines LOUDS and BP, called the Depth-First Unary Degree Sequence (DFUDS) [57]. As shown in Figure 7.1, It uses the same unary encoding as in LOUDS, but traverses the tree in depth-first order as in BP. DFUDS offers a middle ground between fast operations and additional functions, and is popular for building general succinct tries. Grossi and Ottaviano [91] provided a state-of-the-art succinct trie implementation based on DFUDS, which we compare against in Section 3.7.2.

We choose to use LOUDS in FST and SuRF because it is fast and easy to implement compared to the alternatives. With proper rank and select support, LOUDS can perform "move to i-th child" and "move to parent" in constant time, and they are sufficient to implement the point and range queries in our system indexing use case. As Arroyuelo et al. [52] reported, LOUDS outperforms all the other succinct tree representations in the above two operations. Although LOUDS lacks efficient support for many other operations such as subtree size and level ancestor [136], its good performance and moderate complexity in the simple "parent-child" navigations fit our needs.

Succinct [48] and follow-up work BlowFish [104] are among the few attempts in systems research to use succinct data structures extensively in a general distributed data store. They store datasets using compressed suffix arrays [92] and achieve significant space savings. Compared to other non-compressed systems, Succinct and BlowFish achieve better query performance mainly through keeping more data resident in DRAM. FST can provide similar benefits when used in larger-than-DRAM workloads. In addition, FST does not slow down the system even when the entire data set fits in DRAM.

7.2 Range Filtering

The Bloom filter [62] and its major variants [64, 81, 142] are compact data structures designed for fast approximate membership tests. They are widely used in storage systems, especially LSM trees, to reduce expensive disk I/O. Similar applications can be found in distributed systems to reduce network I/O [4, 153, 166]. The downside for Bloom filters, and other filters such as Quotient filters [56], Cuckoo filters [80] and Morton filters [65], however, is that they cannot handle range queries because their hashing does not preserve key order. One could build state-of-the-art tree indexes [43, 60, 112, 160] for the task, but the memory cost is high (see evaluation in Section 3.7). In practice, people often use prefix Bloom filters to help answer range-emptiness queries. For example, RocksDB [37], LevelDB [5], and LittleTable [149] store pre-defined key prefixes in Bloom filters so that they can identify an empty-result query if they do not find a matching prefix in the filters. Compared to SuRFs, this approach, however, has worse filtering ability and less flexibility. It also requires additional space to support both point and range queries.

Adaptive Range Filter (ARF) [49] was introduced as part of Project Siberia [79] in Hekaton [74] to guard cold data. ARF differs from SuRF in that it targets different applications and scalability goals. First, ARF behaves more like a cache than a general-purpose filter. Training an ARF requires knowledge about prior queries. An ARF instance performs well on the particular query pattern for which it was trained. If the query pattern changes, ARF requires a rebuild (i.e., decode, re-train, and encode) to remain effective. ARF works well in the setting of Project Siberia, but its workload assumptions limit its effectiveness as a general range filter. SuRF, on the other hand, assumes nothing about workloads. It can be used as a Bloom filter replacement but with range filtering ability. In addition, ARF's binary tree design makes it difficult to accommodate variable-length string keys because a split key that evenly divides a parent node' key space into its children nodes' key space is not well defined in the variable-length string key space. In contrast, SuRF natively supports variable-length string keys with its trie design. Finally, ARF performs a linear scan over the entire level when traversing down the tree. Linear lookup complexity prevents ARF from scaling; the authors suggest embedding many small ARFs into the existing B-tree index in the hot store of Hekaton, but lookups within individual ARFs still require linear scans. SuRF avoids linear scans by navigating its internal tree structure with rank & select operations. We compared ARF and SuRF in Section 4.3.5.

7.3 Log-Structured Storage

Many modern key-value stores adopt the log-structured merge tree (LSM-tree) design [138] for its high write throughput and low space amplification. Such systems include LevelDB [5], RocksDB [37], Cassandra [7, 107], HBase [24], WiredTiger [1], OctopusDB [75], LHAM [135], and cLSM [86] from Yahoo Labs. Monkey [71] explores the LSM-tree design space and provides a tuning model for LSM-trees to achieve the Pareto optimum between update and lookup speeds given a certain main memory budget. The RocksDB team published a series of optimizations (including the prefix Bloom filter) to reduce the space amplification while retaining acceptable performance [78]. These optimizations fall under the RUM Conjecture [54]: for read, update, and memory, one can only optimize two at the cost of the third. The design of FST also falls under the RUM Conjecture because it trades update efficiency for fast read and small space. LSM-trie [163] improves read and write throughput over LevelDB for small key-value pairs, but it does not support range queries.

SILT is a flash-based key-value store that achieves high performance with a small memory footprint by using a multi-level storage hierarchy with different data structures [119]. The first level is a log-structured store that supports fast writes. The second level is a transitional hash table to perform buffering. The final level is a compressed trie structure. Hybrid indexes borrow from this design, but unlike SILT, a hybrid index does not use a log-structured storage tier because maximizing the number of sequential writes is not a high priority for in-memory databases. Hybrid indexes also avoid SILT's heavyweight compression because of the large performance overhead. Similar systems include Anvil, a modular framework for database backends to allow flexible combinations of the underlying key-value stores to maximize their benefits [123].

7.4 Hybrid Index and Other Compression Techniques for Main-memory Databases

A common way to reduce the size of B+trees is to compress their nodes before they are written to disk using a general-purpose compression algorithm (e.g., LZMA) [25].

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. To the best of our knowledge, the only compressed main-memory indexes are for OLAP systems, such as bitmap [67] and columnar [108] indexes. These techniques, however, are inappropriate for the write-heavy workload mixtures and small-footprint queries of OLTP applications [155]. As we show in Sections 2.5 and 5.3, compressed indexes perform poorly due to the overhead of decompressing an entire block to access a small number of tuples.

An important aspect of these previous approaches is that the indexes treat all of the data in the underlying table equally. That is, they assume that the application will execute queries that access all of the table's tuples in the same manner, either in terms of frequency (i.e., how many times it will be accessed or modified in the future) or use (i.e., whether it will be used most in point versus range queries). This assumption is incorrect for many OLTP applications. For example, a new tuple is likely to be accessed more often by an application soon after it was added to the database, often through a point query on the index. But as the tuple ages, its access frequency decreases. Later, the only time it is accessed is through summarization or aggregation queries.

One could handle this scenario through multiple partial indexes on the same keys in a table that use different data structures. There are several problems with this approach beyond just the additional cost of maintaining more indexes—foremost is that developers might need to modify their application so that each tuple specifies what index it should be stored in at runtime. This information is necessary because some attributes, such as a tuple's creation timestamp, may not accurately represent how likely it will be accessed in the future. Second, the DBMS's query optimizer might not be able to infer what index to use for a query since a particular tuple's index depends on this identifying value. If a complex query accesses tuples from multiple partial indexes that each has a portion of the table's data, then the system will need to retrieve data from multiple sources for that query operator. This type of query execution is not possible in today's DBMSs, so the system would likely fall back to scanning the table sequentially.

We, therefore, argue that a better approach is to use a single logical hybrid index that is composed of multiple data structures. This approach gives the system more fine-grained control over data storage without requiring changes to the application. To the rest of the DBMS, a hybrid index looks like any other, supporting a conventional interface and API. Previous work such as LSM-trees showed the effectiveness of using multiple physical data structures or building blocks to construct a higher-level logical entity. Applying these ideas to database indexes is a natural fit, especially for in-memory OLTP systems. In these applications, transactions' access patterns vary over time with respect to age and use. Index entries for new tuples go into a fast, write-friendly data structure since they are more likely to be queried again in the near future. Over time, the tuples become colder and their access patterns change, usually from frequent modification to occasional read [72]. Aged tuples thus eventually migrate to a more read-friendly and more compact data structure to save space [152].

Several DBMSs use compressed indexes to reduce the amount of data that is read from disk during query execution. There has been considerable work on space-efficient indexes for OLAP workloads to improve the performance of long-running queries that access large segments of the database [67, 161]. SQL Server's columnar indexes use a combination of dictionary-based, value-based, and run-length encoding to compress the column store indexes [108]. MySQL's InnoDB storage engine has the ability to compress B-tree pages when they are written to disk [25]. To amortize compression and decompression overhead, InnoDB keeps a modification log within each B-tree page to buffer incoming changes to the page. This approach differs from hybrid indexes, which focus on structural reduction rather than data compression. Because hybrid indexes target in-memory databases and their concomitant performance objectives, data compression is prohibitive in most cases.

Other in-memory databases save space by focusing on the tuple stores rather than the index structures. One example is SAP's HANA hybrid DBMS [82, 152]. In HANA, all new data is first inserted into a row-major store engine that is optimized for OLTP workloads. Over time, the system migrates tuples to dictionary-compressed, in-memory columnar store that is optimized for OLAP queries. This approach is also used in HyPer [84]. Hybrid indexes take a similar approach to migrate cold data from the write-optimized index to the compact, read-only index. Both these techniques are orthogonal to hybrid indexes. A DBMS can use hybrid indexes while still moving data out to these compressed data stores.

Other work seeks to reduce the database's storage footprint by exploiting the access patterns of OLTP workloads to evict cold tuples from memory. These approaches differ in how they determine what to evict and the mechanism they use to move data. The anti-caching architecture in H-Store uses an LRU to track how often tuples are accessed and then migrates cold data to an auxiliary, on-disk data store [72]. Although the tuple data is removed from memory, the DBMS still has to keep all of the index keys in-memory. A similar approach was proposed for VoltDB (the commercial implementation of H-Store) where the database relies on the OS's virtual memory mechanism to move cold pages out to disk [154]. The Siberia Project for Microsoft's Hekaton categorizes hot/cold tuples based on sampling their access history [116] and can also migrate data out to an on-disk data store [79]. Hekaton still uses a disk-backed index, so cold pages are swapped out to disk as needed using SQL Server's buffer pool manager and the remaining in-memory index data is not compressed. Hybrid indexes do not rely on any tracking information to guide the merging process since it may not be available in every DBMS. It is future work to determine whether such access history may further improve hybrid indexes' performance.

The Dynamic-to-Static Rules are inspired by work from Bentley and Saxe [58]. In their paper, they propose general methods for converting static structures to dynamic structures; their goal is to provide a systematic method for designing new, performance-optimized dynamic data structures. In Chapter 2, we use a different starting point, a dynamic data structure, and propose rules for creating a static version; furthermore, our focus is on creating space-optimized rather than performance-optimized variants.

Dynamic materialized views materialize only a selective subset of tuples in the view based on tuple access frequencies to save space and maintenance costs [172]. Similarly, database cracking constructs self-organizing, discriminative indexes according to the data access patterns [98]. Hybrid indexes leverage the same workload adaptivity by maintaining fast access paths for the newly inserted/updated entries to save memory and improve performance.

7.5 Key Compression in Search Trees

Existing compression techniques for search trees leverage general-purpose block compression algorithms such as LZ77 [27], Snappy [39], and LZ4 [26]. Block compression algorithms, however, are too slow for in-memory search trees: query latencies for in-memory B+trees and tries range from 100s of nanoseconds to a few microseconds, while the fastest block compression algorithms can decompress only a few 4 KB memory pages in that time [26]. Recent work has addressed this size problem through new data structures [60, 127, 168, 169]. Compressing input keys using HOPE is an orthogonal approach that one can apply to any of the above search tree categories to achieve additional space savings and performance gains.

One could apply existing field/table-wise compression schemes to search tree keys. Whole-key dictionary compression is the most popular scheme used in DBMSs today. It replaces the values in a column with smaller fixed-length codes using a dictionary. Indexes and filters, therefore, could take advantage of those existing dictionaries for key compression. There are several problems with this approach. First, the dictionary compression must be order-preserving to allow range queries on search trees. Order-preserving dictionaries, however, are difficult to maintain with changing value domains [120], which is often the case for string keys in OLTP applications. Second, the latency of encoding a key is similar to that of querying the actual indexes/filters because most order-preserving dictionaries use the same kind of search trees themselves [61]. Finally, dictionary compression only works well for columns with low/moderate cardinalities. If most values are unique, then the larger dictionary negates the size reduction in the actual fields.

Existing order-preserving frequency-based compression schemes, including the one used in DB2 BLU [147] and padded encoding [118], exploit the column value distribution skew by assigning smaller codes to more frequent values. Variable-length codes, however, are inefficient to locate, decode, and process in parallel. DB2 BLU, thus, only uses up to a few different code sizes per column and stores the codes of the same size together to speed up queries. Padded encoding, on the other hand, pads the variable-length codes with zeros at the end so that all codes are of the same length (i.e., the maximum length in the variable-length codes) to facilitate scan queries. DB2 BLU and padded encoding are designed for column stores where most queries are reads, and updates are often in batches. Both designs still use the whole-key dictionary compression discussed above and therefore, cannot encode new values without extending the dictionary, which can cause expensive re-encodes of the column. HOPE, however, can encode arbitrary input values using the same dictionary while preserving their ordering. Such property is desirable for write-intensive OLTP indexes.

HOPE focuses on compressing string keys. Numeric keys are already small and can be further compressed using techniques, such as null suppression and delta encoding [47]. Prefix compression is a common technique used in B+trees, where each node only stores the common prefix of its keys once [55]. Prefix compression can achieve at most the same level of reduction as a radix tree. Suffix truncation is another common technique where nodes skip the suffixes after the keys are uniquely identified in the tree [60, 112, 169]. Suffix truncation is a lossy scheme, and it trades a higher false positive rate for better memory-efficiency.

Prior studies considered entropy encoding schemes, such as Huffman [97] and arithmetic coding [162], too slow for columnar data compression because their variable-length codes are slow to decode [47, 59, 61, 68, 120, 146]. For example, DB2 BLU only uses up to a few different code sizes [147]. This concern does not apply to search trees, because non-covering index and filter queries do not reconstruct the original keys[1]. In addition, entropy encoding schemes produce high compression rates even with small dictionaries because they exploit common patterns at a fine granularity.

Antoshenkov et al. [50, 51] proposed an order-preserving string compressor with a string parsing algorithm (ALM) to guarantee the order of the encoded results. We introduced our string axis model in Section 6.1, which is inspired by the ALM method but is more general: The ALM compressor belongs to a specific category in our compression model.

[1] The search tree can, of course, recover the original keys if needed: entropy encoding is lossless, unlike suffix truncation.

Chapter 8

Conclusion and Future Work

In this dissertation, we presented three steps towards memory-efficient search trees for database management systems. In the first step, we target at building fast static search trees to approach theoretically optimal compression. In this step, we first developed the *Dynamic-to-Static Rules* to serve as the high-level guidelines for identifying and reducing structural memory overhead in existing search trees. We then introduced the *Fast Succinct Trie* (*FST*) that consumes space close to the information-theoretic lower bound but achieves query performance comparable to the state-of-the-art solutions. Using FST, we built the *Succinct Range Filter* (*SuRF*) that solves the range filtering problem practically in real databases. In the second step, we introduced the *Hybrid Index* architecture that can support inserts and updates on static search trees efficiently with bounded and amortized cost in performance and memory. In the final step, we focused on compressing the keys stored in a search tree instead of the tree structure by building the *High-speed Order-Preserving Encoder* (*HOPE*) for search tree keys that achieves high compression rates and performance while preserving the ordering of arbitrary input keys. These three steps together form a practical recipe for achieving memory-efficiency in search trees and in databases.

We briefly discuss several directions to extend the work presented in this dissertation. As we discussed in Chapter 4, the current version of SuRF has two major limitations that prevent it from being used in a wider range of applications First, SuRF is static. Inserts and updates will cause a significant part of the data structure to rebuild. Second, SuRF lacks a theoretical guarantee on the false positive rate for range queries, despite its good empirical performance. Addressing these limitations should be the focus of the next-generation range filters.

The rise of main-memory (and NVM) databases running on multicore machines has motivated research in developing highly concurrent indexes to support simultaneous reads and writes at scale. However, existing concurrent indexes are memory-consuming. One possible solution is to extend the hybrid index architecture (Chapter 5) to support concurrent operations. Recall that a hybrid index includes the dynamic stage and a static stage. The main challenge in building a concurrent hybrid index is to design an efficient non-blocking algorithm to periodically merge the two stages. We briefly describe a proposed non-blocking merge algorithm here.

We first add a temporary intermediate stage between the dynamic and the static stages. When a merge is triggered, the current dynamic stage freezes and becomes the read-only intermediate stage. Meanwhile, a new empty dynamic stage is created to continue receiving all the writes. The writes are thus independent of the merge process. The problem reduces to merging two read-only indexes (i.e., from the intermediate stage to the static stage) without blocking access to any item. A naive solution is to create an entire copy of the static stage and then perform the merge on the copy (i.e., a full copy-on-write). This is undesirable because it doubles the memory use during a merge. We can, however, merge the two structures *incrementally* by performing atomic updates at the subtree level in the static stage. It is valid to have a partially-merged static stage (i.e., with a subset of the new items merged in) in this case because read requests for the to-be-merged items will hit in the intermediate stage instead of in the final static stage.

Applying the compression techniques in HOPE to further parts of the DBMS is another potential extension to this thesis. For example, because HOPE exploits entropy at the substring level, table columns that store the same type of information could share a HOPE dictionary while still achieving good compression rates. Encoding columns using the same dictionary could speed up joins by allowing them to operate on compressed data directly, avoiding the cost of decoding and re-encoding the joining columns.

Two trends have emerged in recent database designs. The first trend is to offload data-intensive jobs to specialized hardware such as GPUs and FPGAs to achieve better performance and energy efficiency. The problem with today's hardware accelerators, however, is that they can only benefit a subset of queries in an ad-hoc manner. The architecture community is experimenting with new hardware, such as the Configurable Spatial Accelerator (CSA) [19] from Intel, that can directly map and execute compiler-generated dataflow graphs with a short (e.g., a few microseconds) board-reconfiguration time. Unlike using today's FPGAs, DBMSs can determine what tasks to accelerate on-the-fly by directly loading the compiled programs to the board. If such reconfigurable dataflow units co-exist with CPUs on the same Xeon-like die, it will eliminate the PCIe bottleneck and the cache-coherence problem faced by today's FPGAs. That means shorter-running queries, including those in OLTP applications, can also benefit from the hardware acceleration. For search tree indexes, this architectural change might bring us opportunities to hard-code the branching keys and comparison logic of heavily-queried indexes to the board so that index lookups can happen at bare-metal speed [167].

The second trend is to use machine learning to automate systems' configuration and performance tuning. Recent work [122, 139, 158] studied using machine learning to predict simple workload patterns and to tune certain database knobs. An interesting problem is to figure out how to automatically select and tune the index data structures in a DBMS. One challenge is that many index data structures today are difficult to tune: they are not designed to facilitate systems to make trade-offs (e.g., between performance, memory, and accuracy). Therefore, we must first create a spectrum of index structures that are "tuning-friendly", and then use machine learning and program synthesis techniques to automatically obtain optimized configurations of those structures to achieve maximized DBMS performance. The goal is to allow the long tail of database applications to benefit from the latest advanced data structures, such as the ones introduced in this dissertation, with a modest engineering expense.

Bibliography

[1] WiredTiger. http://wiredtiger.com. 118

[2] Stx b+ tree c++ template classes. http://idlebox.net/2007/stx-btree/, 2008. 11, 21, 32

[3] tx-trie 0.18 – succinct trie implementation. https://github.com/hillbig/tx-trie, 2010. 5, 21, 33

[4] Squid web proxy cache. http://www.squid-cache.org/, 2013. 117

[5] Google LevelDB. https://github.com/google/leveldb, 2014. 37, 117, 118

[6] Best practices for index optimization in voltdb. https://www.voltdb.com/blog/2014/06/09/best-practices-index-optimization-voltdb/, 2014. 10

[7] Apache Cassandra. https://cassandra.apache.org/, 2015. 118

[8] SAP Hana SQL and system view reference – indexes system view. https://help.sap.com/viewer/4fe29514fd584807ac9f2a04f6754767/2.0.03/en-US/20a7044375191014a939f50ae14306f7.html?q=index, 2015. 10

[9] Kairosdb. https://kairosdb.github.io/, 2015. 52

[10] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2015. 53

[11] Adaptive Range Filter implementation. https://github.com/carolinux/adaptive_range_filters, 2016. 51

[12] Articles benchmark. https://github.com/apavlo/h-store/tree/release-2016-06/src/benchmarks/edu/brown/benchmark/articles, 2016. 2

[13] H-Store. http://hstore.cs.brown.edu, 2016. xix, 2, 10, 75

[14] OLC B+tree. https://github.com/wangziqi2016/index-microbench/blob/master/BTreeOLC, 2016. 97

[15] Cache-Optimized Concurrent Skip list. http://sourceforge.net/projects/skiplist/files/Templatized%20C%2B%2B%20Version/, 2016. 11

[16] The history of 3D NAND flash memory. https://www.nvmdurance.com/history-of-3d-nand-flash-memory/, 2017. 1

[17] The InfluxDB storage engine and the time-structured merge tree (tsm). https://docs.influxdata.com/influxdb/v1.0/concepts/storage_engine/, 2017. 37, 52

[18] Zlib. http://www.zlib.net/, 2017. 3

[19] Configurable spatial accelerator (csa) – intel. https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator, 2018. 124

[20] SuRF: First practical and general-purpose range filter. https://github.com/efficient/SuRF, 2018. 38

[21] Amazon EC2 I3 instances. https://aws.amazon.com/ec2/instance-types/i3/, 2019. 1

[22] ALTIBASE index documentation. http://aid.altibase.com/display/migfromora/Index, 2019. 10

[23] Cachegrind. http://valgrind.org/docs/manual/cg-manual.html, 2019. 112

[24] Apache HBase. https://hbase.apache.org/, 2019. 118

[25] MySQL v8.0 – How compression works for innodb tables. https://dev.mysql.com/doc/refman/8.0/en/innodb-compression-internals.html, 2019. 118, 120

[26] LZ4. https://lz4.github.io/lz4, 2019. 3, 16, 121

[27] LZ77 and LZ78. https://en.wikipedia.org/wiki/LZ77_and_LZ78, 2019. 3, 121

[28] Memory Prices (1957-2019). https://www.jcmit.net/memoryprice.htm, 2019. 1

[29] MemSQL documentation. http://docs.memsql.com/latest/concepts/indexes/, 2019. 10

[30] Facebook MyRocks. http://myrocks.io/, 2019. 37

[31] MySQL memory storage engine. http://dev.mysql.com/doc/refman/5.7/en/memory-storage-engine.html, 2019. 10

[32] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/index.html, 2019. 18

[33] Peloton: The Self-Driving Database Management System. https://pelotondb.io/, 2019. 10

[34] QuasarDB. https://www.quasardb.net/, 2019. 52

[35] Reddit. http://www.reddit.com, 2019. 77

[36] Redis index. http://redis.io/topics/indexes, 2019. 10

[37] RocksDB: A persistent key-value store for fast storage environment. https://rocksdb.org/, 2019. 5, 37, 117, 118

[38] Succinct data structures. https://en.wikipedia.org/wiki/Succinct_data_structure, 2019. 22, 97

[39] Snappy. https://github.com/google/snappy, 2019. 3, 16, 66, 121

[40] SQLite Documentation. https://www.sqlite.org/docs.html, 2019. 10

[41] Flash Memory and SSD Prices (2003-2019). https://jcmit.net/flashprice.htm, 2019. 1

[42] Timesten in-memory database operations guide. https://docs.oracle.com/database/121/TTOPR/perform.htm#TTOPR411, 2019. 10

[43] TLX B+tree (formerly STX B+tree). https://github.com/tlx/tlx, 2019. 83, 97, 117

[44] URL dataset. http://law.di.unimi.it/webdata/uk-2007-05/, 2019. 100

[45] Voter benchmark. https://github.com/VoltDB/voltdb/tree/master/examples/voter, 2019. 2

[46] English Wikipedia article title dataset. https://dumps.wikimedia.org/enwiki/20190701/enwiki-20190701-all-titles-in-ns0.gz, 2019. 100

[47] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM International Conference on Management of Data (SIGMOD'06)*, pages 671–682. ACM, 2006. 83, 122

[48] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 337–350, 2015. 117

[49] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment*, 6(14): 1714–1725, 2013. 51, 117

[50] Gennady Antoshenkov. Dictionary-based order-preserving string compression. *The VLDB Journal*, 6(1):26–39, 1997. 84, 87, 88, 90, 93, 122

[51] Gennady Antoshenkov, David Lomet, and James Murray. Order preserving string compression. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE'96)*, pages 655–663. IEEE, 1996. 84, 122

[52] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Proceedings of the 2019 Algorithm Engineering and Experiments (ALENEX'10)*, pages 84–97, 2010. 3, 116

[53] Nikolas Askitis and Ranjan Sinha. Hat-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 97–105. Australian Computer Society, Inc., 2007. 2

[54] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing access methods: The rum conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT'16)*, volume 2016, pages 461–466, 2016. 9, 118

[55] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Transactions on Database*

Systems (TODS), 2(1):11–26, 1977. 83, 97, 122

[56] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012. 117

[57] 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, 22, 33, 116

[58] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980. 121

[59] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George Mihaila, Kenneth Ross, Sherman Lau, Cathy McArthur, Zoltan Toth, and Reza Sherkat. Efficient index compression in db2 luw. *Proceedings of the VLDB Endowment*, 2(2): 1462–1473, 2009. 2, 122

[60] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 521–534. ACM, 2018. 2, 18, 83, 97, 108, 117, 121, 122

[61] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 International Conference on Management of Data (SIGMOD'09)*, pages 283–296. ACM, 2009. 86, 121, 122

[62] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. 5, 37, 117

[63] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. Efficient in-memory indexing with generalized prefix trees. *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2011. 2

[64] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, pages 684–695. Springer, 2006. 117

[65] Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018. 117

[66] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing (STOC'78)*, pages 59–65. ACM, 1978. 58

[67] Chee-Yong Chan and Yannis E Ioannidis. Bitmap index design and evaluation. In *ACM SIGMOD Record*, volume 27, pages 355–366. ACM, 1998. 59, 119, 120

[68] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *ACM SIGMOD Record*, 30(2):271–282, 2001. 122

[69] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979. 2, 10

[70] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143–154. ACM, 2010. 17, 31, 46, 66, 108

[71] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, pages 79–94. ACM, 2017. 118

[72] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment*, 6(14):1942–1953, 2013. 59, 76, 82, 120

[73] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984. 115

[74] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 International Conference on Management of Data (SIGMOD'13)*, pages 1243–1254. ACM, 2013. 51, 115, 117

[75] Jens Dittrich and Alekh Jindal. Towards a one size fits all database architecture. In *Proceedings of the 2011 Conference on Innovative Data Systems Research (CIDR'11)*, pages 195–198, 2011. 118

[76] Siying Dong. personal communication, 2017. 2017-08-28. 37, 55

[77] Siying Dong. personal communication, 2018. 2018-12-26. 1

[78] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in RocksDB. In *Proceedings of the 2017 Conference on Innovative Data Systems Research (CIDR'17)*, volume 3, page 3, 2017. 37, 53, 118

[79] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment*, 7(11):931–942, 2014. 51, 59, 117, 120

[80] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT'14)*, pages 75–88. ACM, 2014. 117

[81] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000. 117

[82] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012. 115, 120

[83] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. 2

[84] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid OLTP & OLAP databases. *Proceedings of the VLDB Endowment*, 5 (11):1424–1435, 2012. 10, 59, 120

[85] Richard F Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368 (3), 2006. 116

[86] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*, page 32. ACM, 2015. 118

[87] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38, 2005. 23

[88] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space. In *Proceedings of the twenty-sixth annual ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*, pages 769–775. SIAM, 2014. 57

[89] Goetz Graefe et al. Modern b-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011. 97

[90] Roberto Grossi and Giuseppe Ottaviano. Design of practical succinct data structures for large data collections. In *Proceedings of the 2013 International Symposium on Experimental Algorithms (SEA'13)*, pages 5–17. Springer, 2013. 3

[91] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *Journal of Experimental Algorithmics (JEA)*, 19:3–4, 2015. 3, 5, 21, 33, 116

[92] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Comput-*

ing, 35(2):378–407, 2005. 117

[93] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 International Conference on Management of Data (SIGMOD'08)*, pages 981–992. ACM, 2008. 115

[94] Steffen Heinz, Justin Zobel, and Hugh E Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, 20(2): 192–223, 2002. 2

[95] Te C Hu and Alan C Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971. 87

[96] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, pages 651–665. ACM, 2019. 1

[97] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 84, 87, 122

[98] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database cracking. In *Proceedings of the 2007 Conference on Innovative Data Systems Research (CIDR'07)*, volume 7, pages 68–78, 2007. 121

[99] Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science*, pages 549–554. IEEE, 1989. 3, 22

[100] Stelios Joannou and Rajeev Raman. Dynamizing succinct tree representations. In *Proceedings of the 2012 International Symposium on Experimental Algorithms (SEA'12)*, pages 224–235. Springer, 2012. 3

[101] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008. 75

[102] Alexey Karyakin and Kenneth Salem. An analysis of memory power consumption in database systems. In *Proceedings of the 13th International Workshop on Data Management on New Hardware (DaMoN'17)*, page 2. ACM, 2017. 1

[103] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE'11)*, pages 195–206. IEEE, 2011. 97

[104] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-

performance tradeoff in data stores. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 485–500, 2016. 117

[105] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. Kiss-tree: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN'12)*, pages 16–23. ACM, 2012. 2

[106] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011. 59

[107] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. 37, 52, 118

[108] Per-Åke Larson, Cipri Clinciu, Eric N Hanson, Artem Oks, Susan L Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL server column store indexes. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD'11)*, pages 1177–1184. ACM, 2011. 59, 119, 120

[109] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Suleyman S Demirsoy, and Kai-Uwe Sattler. Fast & strong: The case of compressed string dictionaries on modern cpus. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*, page 4. ACM, 2019. 2

[110] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12), 2003. 1

[111] Tobin J Lehman and Michael J Carey. A study of index structures for main memory database management systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1985. 97

[112] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*, pages 38–49. IEEE, 2013. xiii, 2, 4, 11, 12, 21, 32, 83, 95, 97, 117, 122

[113] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)*, page 3. ACM, 2016. 97

[114] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019. 97

[115] Justin Levandoski, David Lomet, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. Indexing on modern hardware: Hekaton and beyond. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD'14)*, pages 717–720. ACM, 2014. 10

[116] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*, pages 26–37. IEEE, 2013. 59, 120

[117] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*, pages 302–313. IEEE, 2013. 2

[118] Yinan Li, Craig Chasseur, and Jignesh M Patel. A padded encoding scheme to accelerate scans by leveraging skew. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD'15)*, pages 1509–1524. ACM, 2015. 122

[119] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 1–13. ACM, 2011. 59, 118

[120] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. Mostly order preserving dictionaries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE'19)*. IEEE, 2019. 121, 122

[121] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms (TALG)*, 4(3):28, 2008. 116

[122] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 631–645. ACM, 2018. 124

[123] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with anvil. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'11)*, pages 147–160. ACM, 2009. 118

[124] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European conference on Computer Systems (EuroSys'12)*, pages 183–196. ACM, 2012. 2, 4, 11

[125] G Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference, Southampton, 1979*, pages 24–27, 1979. 94

[126] Miguel Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016. 3

[127] Markus Mäsker, Tim Süß, Lars Nagel, Lingfang Zeng, and André Brinkmann. Hyperion: Building the largest in-memory search tree. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, pages 1207–1222. ACM, 2019. 121

[128] Justin Meza, Mehul A Shah, Parthasarathy Ranganathan, Mike Fitzner, and Judson Veazey. Tracking the power in an enterprise decision support system. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 261–266. ACM, 2009. 1

[129] Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968. 2

[130] Ingo Müller, Cornelius Ratsch, Franz Faerber, et al. Adaptive string dictionary compression in in-memory column-store database systems. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*, pages 283–294, 2014. 83

[131] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. 3, 116

[132] J Ian Munro and S Srinivasa Rao. Succinct representations of functions. In *International Colloquium on Automata, Languages, and Programming (ICALP'04)*, pages 1006–1015. Springer, 2004. 116

[133] J Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375. SIAM, 1992. 11

[134] J Ian Munro, Venkatesh Raman, and S Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001. 116

[135] Peter Muth, Patrick O'Neil, Achim Pick, and Gerhard Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, 2000. 118

[136] Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016. 3, 117

[137] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proceedings of the 2012 International Symposium on Experimental Algorithms (SEA'12)*, pages 295–306. Springer, 2012. 23

[138] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. 37, 60, 118

[139] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving

database management systems. In *CIDR*, volume 4, page 1, 2017. 124

[140] Adam Prout. The story behind memsql's skiplist indexes. https://www.memsql.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/, 2014. 11

[141] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. 2, 4, 11

[142] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient bloom filters. In *Proceedings of the 2007 International Workshop on Experimental and Efficient Algorithms (WEA'07)*, pages 108–121. Springer, 2007. 117

[143] Naila Rahman, Rajeev Raman, et al. Engineering the louds succinct tree representation. In *Proceedings of the 2006 International Workshop on Experimental and Efficient Algorithms (WEA'06)*, pages 134–145. Springer, 2006. 3, 22

[144] Rajeev Raman and S Srinivasa Rao. Succinct representations of ordinal trees. In *Space-efficient data structures, streams, and algorithms*, pages 319–332. Springer, 2013. 3

[145] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43, 2007. 116

[146] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd international conference on Very large databases (VLDB'06)*, pages 858–869. VLDB Endowment, 2006. 122

[147] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013. 122

[148] Jun Rao and Kenneth A Ross. Making B+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000. 2

[149] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable: a time-series database and its uses. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD'17)*, pages 125–138. ACM, 2017. 37, 52, 117

[150] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms (SODA'10)*, pages 134–149. SIAM, 2010. 3, 116

[151] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD'12)*, pages 217–228. ACM, 2012. 37

[152] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and

Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD'12)*, pages 731–742. ACM, 2012. 59, 120

[153] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005. 117

[154] Radu Stoica and Anastasia Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN'13)*, page 7. ACM, 2013. 59, 115, 120

[155] 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 *Proceedings of the 33rd international conference on very large databases (VLDB'07)*, pages 1150–1160. VLDB Endowment, 2007. 1, 59, 115, 119

[156] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11). http://www.tpc.org/tpcc/, February 2010. 2, 76

[157] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 18–32. ACM, 2013. 10, 11

[158] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD'17)*, pages 1009–1024. ACM, 2017. 124

[159] Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proceedings of the 2008 International Workshop on Experimental and Efficient Algorithms (WEA'08)*, pages 154–168. Springer, 2008. 23

[160] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 473–488. ACM, 2018. 2, 18, 117

[161] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *ACM Sigmod Record*, 29 (3):55–67, 2000. 59, 120

[162] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. 87, 122

[163] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC'15)*, pages 71–82. USENIX As-

sociation, 2015. 118

[164] Andrew Chi-Chih Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978. 14

[165] JM Yohe. Hu-tucker minimum redundancy alphabetic coding method [z] (algorithm 428). *Commun. ACM*, 15(5):360–362, 1972. 94

[166] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies (CoNEXT'09)*, pages 313–324. ACM, 2009. 117

[167] Huanchen Zhang. The end of the x86 dominance in databases? *CIDR'19*, 2019. 124

[168] 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'16)*, pages 1567–1581. ACM, 2016. 2, 5, 21, 58, 108, 121

[169] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 323–336. ACM, 2018. 5, 83, 97, 108, 121, 122

[170] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)*. ACM, 2020. (to appear). 5

[171] Dong Zhou, David G Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Proceedings of the 2013 International Symposium on Experimental Algorithms (SEA'13)*, pages 151–163. Springer, 2013. 23, 30, 35

[172] Jingren Zhou, Per-Ake Larson, Jonathan Goldstein, and Luping Ding. Dynamic materialized views. In *2007 IEEE 23rd International Conference on Data Engineering (ICDE'07)*, pages 526–535. IEEE, 2007. 121