Succinct Range Filters

HUANCHEN ZHANG, Carnegie Mellon University
HYEONTAEK LIM, Carnegie Mellon University
VIKTOR LEIS, Friedrich Schiller University Jena
DAVID G. ANDERSEN, Carnegie Mellon University
MICHAEL KAMINSKY, BrdgAI
KIMBERLY KEETON, Hewlett Packard Labs
ANDREW PAVLO, Carnegie Mellon University

We present the *Succinct Range Filter* (SuRF), a fast and compact data structure for approximate membership tests. Unlike traditional Bloom filters, SuRF supports both single-key lookups and common range queries: open-range queries, closed-range queries, and range counts. SuRF is based on a new data structure called the *Fast Succinct Trie (FST)* that matches the point and range query performance of state-of-the-art order-preserving indexes, while consuming only 10 bits per trie node. The false-positive rates in SuRF for both point and range queries are tunable to satisfy different application needs. We evaluate SuRF in RocksDB as a replacement for its Bloom filters to reduce I/O by filtering requests before they access on-disk data structures. Our experiments on a 100-GB dataset show that replacing RocksDB's Bloom filters with SuRFs speeds up open-seek (without upper-bound) and closed-seek (with upper-bound) queries by up to 1.5× and 5× with a modest cost on the worst-case (all-missing) point query throughput due to slightly higher false-positive rate.

CCS Concepts: • **Information systems → Data access methods**;

Additional Key Words and Phrases: Range filter, succinct data structure, trie, LSM-tree

ACM Reference format:
Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Trans. Database Syst.* 45, 2, Article 5 (June 2020), 31 pages.
https://doi.org/10.1145/3375660

1 INTRODUCTION

Write-optimized log-structured merge (LSM) trees [52] are popular low-level storage engines for general-purpose databases that provide fast writes [5, 59] and ingest-abundant DBMSs such as time-series databases [11, 57]. One of their main challenges for fast query processing is that items could reside in immutable files (SSTables) from all levels [3, 43]. Item retrieval in an LSM

This work was supported by funding from U.S. National Science Foundation under award CCF-1535821 and the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS).
Authors' addresses: H. Zhang, H. Lim, D. G. Andersen, A. Pavlo, and M. Kaminsky, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213; emails: {huanche1, hl, dga, pavlo, kaminsky}@cs.cmu.edu; V. Leis, Friedrich Schiller University Jena, Fürstengraben 1, 07743 Jena, Germany; email: viktor.leis@uni-jena.de; K. Keeton, Hewlett Packard Labs, 6280 America Center Dr., San Jose, CA 95002; email: kimberly.keeton@hpe.com.

0362-5915/2020/06-ART5 $15.00
https://doi.org/10.1145/3375660

tree-based design may therefore incur multiple expensive disk I/Os [52, 59]. This challenge calls for in-memory data structures that can help locate query items.

Bloom filters [20] 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, then 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 [3, 6, 57, 59], 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@") [6, 28, 57], despite their inflexibility for more general range queries. The designers of RocksDB [6] have expressed a desire to have a more flexible data structure for this purpose [27]. 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.

This article presents 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 tradeoff is tunable for point and range queries semi-independently.

SuRF is built upon a new space-efficient (succinct) data structure called the *Fast Succinct Trie (FST)*. It performs comparably to or better than state-of-the-art uncompressed index structures (e.g., B+tree [18], ART [44]) for both integer and string workloads. FST consumes only 10 bits per trie node, which is close to the information-theoretic lower bound.

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 article) trades space for decreased false positives.

We evaluate SuRF via micro-benchmarks and as a Bloom filter replacement in RocksDB. 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.

This article makes three primary contributions. First, we designed and implemented the FST data structure whose space consumption is close to the minimum number of bits required by information theory yet has performance equivalent to uncompressed order-preserving indexes. Second, using FST, we built SuRF, an approximate membership test that supports both single-key and range queries. Finally, we replace the Bloom filters with size-matching SuRFs in RocksDB and show that it improves range query performance with a modest cost on the worst-case point query throughput due to a slightly higher false-positive rate.

This is an extended version of the ACM SIGMOD'18 publication titled "SuRF: Practical Range Query Filtering with Fast Succinct Tries" [68]. Major additions include the following: (1) more background on succinct data structures (Section 2.1), (2) a detailed description on LOUDS-DS encoding and its operations (Sections 2.2 to 2.4), (3) an experimental analysis on the tradeoffs between LOUDS-Dense and LOUDS-Sparse encodings (Section 4.1.4), (4) an evaluation on SuRF's build time (Section 4.2.3), and (5) a new section discussing SuRF's limitations, including an analysis on a worst-case scenario (Section 7).

2 FAST SUCCINCT TRIES

The core data structure in SuRF is the *Fast Succinct Trie (FST)*. FST is a space-efficient, static trie that answers point and range queries. FST is 4–15× faster than earlier succinct tries [1, 38], achieving performance comparable to or better than the state-of-the-art pointer-based indexes [18, 44, 67] (evaluated in Section 4.1).

FST's design is based on the observation that the upper levels of a trie comprise few nodes but incur many accesses. 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 (LOUDS-Dense) in which a child node search requires only one array lookup, choosing performance over space. We encode the lower levels using the space-efficient LOUDS-Sparse scheme, so that the overall size of the encoded trie is bounded.

Combining LOUDS-Dense and LOUDS-Sparse within the same data structure (we call the hybrid encoding scheme LOUDS-DS) 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 section, 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).

2.1 Background: Succinct Trees and LOUDS

Succinct data structures [40] are those that require, asymptotically, only the minimum number of bits required by information theory, while still answering queries efficiently without expensive decoding/decompressing operations. The space-efficiency is attractive to massive-scale data services, because it allows them to keep a larger portion of the data resident in smaller, but faster, memories (i.e., cache, DRAM). This leads to better performance and/or lower costs.

Trees are among the most successful examples of succinct data structures. Succinct trees have been studied for over two decades, and there is rich literature [14, 17, 37, 38, 40, 41, 46, 47, 54, 58] in theory and in practice. Although succinct trees work well in specific scenarios, such as in information retrieval and XML processing [50, 56], 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 and indexing.

There are two major reasons their use is limited. 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. Succinct data structures are highly optimized for memory consumption and read performance. According to the RUM conjecture [15], their update performance must be compromised. Techniques such as hybrid indexes [67], however, can largely ameliorate this problem by amortizing individual update cost through batching.

Second, existing implementations of succinct trees are at least an order of magnitude slower than their corresponding pointer-based uncompressed trees [14]. This slowdown is hard to justify

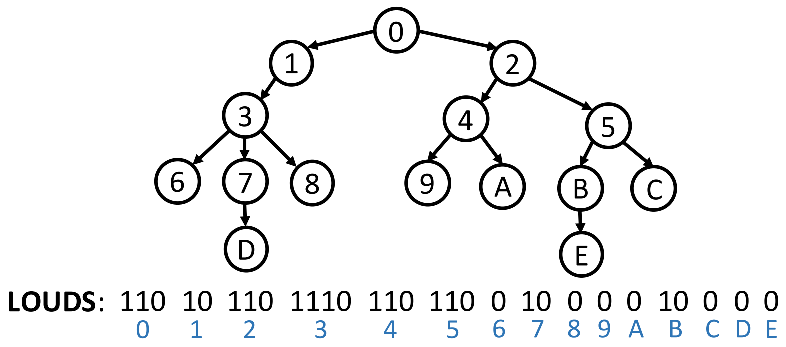

0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
LOUDS: 110 10 110 1110 110 110 0 10 0 0 0 10 0 0 0
0 1 2 3 4 5 6 7 8 9 A B C D E

Fig. 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.

for most systems despite the space advantage. The FST data structure presented below solves this problem. We show that through careful design and engineering, we can build succinct trees that are as fast as the state-of-the-art pointer-based trees, but with less memory.

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 [17].

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 toward succinct tries. Jacobson [40] pioneered research on succinct tree representations and introduced the *Level-Ordered Unary Degree Sequence* (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 1 has three children and is thus encoded as `'1110'`. Follow-up studies include LOUDS++ [54], 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 ith 1 ($select_0(i)$ selects 0's). The original Jacobson paper showed how to support RS operations in $O(\log n)$ bit-accesses [40]. Modern rank & select implementations [35, 51, 62, 69] achieve constant time by using look-up tables (LUTs) to store a sampling of precomputed results so that they only need to count between the samples. A state-of-the-art implementation is from Zhou et al. [69], 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 2.6.

[1]There are three ways to define "close" [10]. 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.

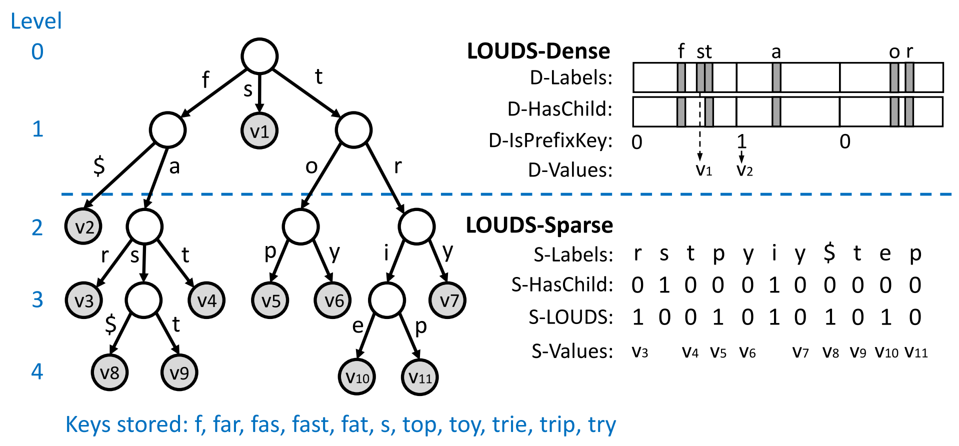

Level
0
1
2
3
4
f
s
t
v1
$
a
o
r
v2
r s t
p y
i y
v3 v4 v5 v6 v7
$ t
e p
v8 v9 v10 v11
LOUDS-Dense
D-Labels:
D-HasChild:
D-IsPrefixKey:
D-Values:
f st a o r
0 1 0
v1 v2
LOUDS-Sparse
S-Labels: r s t p y i y $ t e p
S-HasChild: 0 1 0 0 0 1 0 0 0 0 0
S-LOUDS: 1 0 0 1 0 1 0 1 0 1 0
S-Values: v3 v4 v5 v6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

Fig. 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.

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

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

2.2 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 2. The encoding follows level-order (i.e., breadth-first order).

The first bitmap (*D-Labels*) records the branching labels for each node. Specifically, the ith 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 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 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 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 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) 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, then 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

2.3 LOUDS-Sparse

As shown in the lower half of Figure 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 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 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 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**, however, 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 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 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: $r = rank_1(\textit{S-HasChild}, pos) + 1$. Because every node only has its first bit set in *S-LOUDS*, we can use $select_1(\textit{S-LOUDS}, r)$ to find the position of the *r*th node.

To move to the parent node, we first get the rank *r* of the current node by $r = rank_1(\textit{S-LOUDS}, pos)$, 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: $select_1(\textit{S-HasChild}, r-1)$.

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 - $rank_1(\textit{S-HasChild}, pos)$ values up to *pos* (non-inclusive).

To summarize:

- **S-ChildNodePos**(*pos*) = $select_1(\textit{S-LOUDS}, rank_1(\textit{S-HasChild}, pos) + 1)$
- **S-ParentNodePos**(*pos*) = $select_1(\textit{S-HasChild}, rank_1(\textit{S-LOUDS}, pos) - 1)$
- **S-ValuePos**(*pos*) = pos - $rank_1(\textit{S-HasChild}, pos)$

2.4 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.

A point query on LOUDS-DS works by first searching the LOUDS-Dense levels. If the search does not terminate, then 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, then 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), then compute the child node's starting position in the label sequence and continue to the next level. If the *HasChild* bit is not set, then 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, then 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, then the algorithm recursively moves up to the parent node and

ALGORITHM 1: LOUDS-DS ExactKeySearch

```
1: Variables
2:     $DenseHeight \leftarrow$ the height of the LOUDS-Dense encoded trie
3:     $DenseNodeCount \leftarrow$ total number of nodes in LOUDS-Dense levels
4:     $DenseChildCount \leftarrow$ total number of non-terminating branches LOUDS-Dense levels
5:
6: function LOOKUP($key$)
7:     $level \leftarrow 0, pos \leftarrow 0$
8:     while $level < DenseHeight$ do                                  ▷ First searching in LOUDS-Dense levels
9:         $nodeNum \leftarrow \lfloor pos/256 \rfloor$
10:        if $level \geq$ LEN($key$) then                                 ▷ If run out of search key bytes
11:            if D-IsPrefixKey[$nodeNum$] == 1 then                         ▷ If the current prefix is a key
12:                return D-Values[D-ValuePos($nodeNum \times 256$)]
13:            else
14:                return NULL
15:        $pos \leftarrow pos + key[level]$
16:        if D-Labels[$pos$] == 0 then                                              ▷ Search failed
17:            return NULL
18:        if D-HasChild[$pos$] == 0 then                                        ▷ reached leaf node
19:            return D-Values[D-ValuePos($pos$)]
20:        $pos \leftarrow$ D-ChildNodePos($pos$)                    ▷ Move to child node and continue search
21:        $level \leftarrow level + 1$
22:
23:    $pos \leftarrow$ S-ChildNodePos($nodeNum - DenseNodeCount$)                ▷ Transition to LOUDS-Sparse
24:
25:    while $level <$ LEN($key$) do                          ▷ Searching continues in LOUDS-Sparse levels
26:        if $key[level]$ does NOT exists in the label list of the current node (starting at $pos$) then
27:            return NULL
28:        if S-HasChild[$pos$] == 0 then                                        ▷ reached leaf node
29:            return S-Values[S-ValuePos($pos$)]
30:        $nodeNum \leftarrow$ S-ChildNodeNum($pos$) $+ DenseChildCount$
31:        $pos \leftarrow$ S-ChildNodePos($nodeNum - DenseNodeCount$)   ▷ Move to child node and continue search
32:        $level \leftarrow level + 1$
33:
34:    if S-Labels[$pos$] == 0xFF and S-HasChild[$pos$] == 0 then             ▷ If the search key is a “prefix key”
35:        return S-Values[S-ValuePos($pos$)]
```

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, then the algorithm searches for the left-most key in the subtrie rooted at **L**. If the iterator hits node boundary instead, then 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 4.1.1 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.

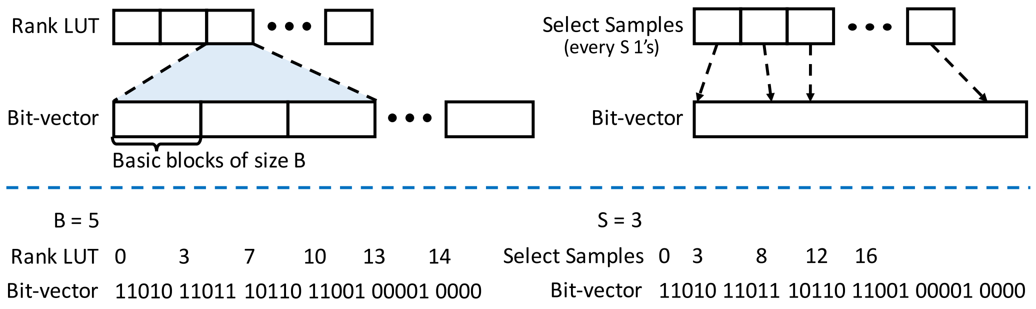

Rank LUT
Bit-vector
Basic blocks of size B
Select Samples
(every S 1's)
Bit-vector
B = 5
Rank LUT 0 3 7 10 13 14
Bit-vector 11010 11011 10110 11001 00001 0000
S = 3
Select Samples 0 3 8 12 16
Bit-vector 11010 11011 10110 11001 00001 0000

Fig. 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.

2.5 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 2.1, the information-theoretic lower bound (Z) for an n-node trie of degree 256 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 4.1.2).

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, then 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 bit vector *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.

2.6 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 4.1.3.

Rank. Figure 3 (left half) shows our lightweight rank structure. Instead of three levels of LUTs (look-up tables) as in Poppy [69], 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 precomputed rank of the start position of the block. For example, in Figure 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, $rank_1(i) = \text{LUT}[\lfloor i/B \rfloor] +$ (*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, 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 [69].

Select. The right half of Figure 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, 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, $select_1(i) = \text{LUT}[i/S]$ + (selecting the $(i - i/S \times S)$th set bit starting from position $\text{LUT}[i/S] + 1) + 1$. 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 $(8 - 8/3 \times 3) = 2$nd 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 4 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 SUCCINCT RANGE FILTERS

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 7.

3.1 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, 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.

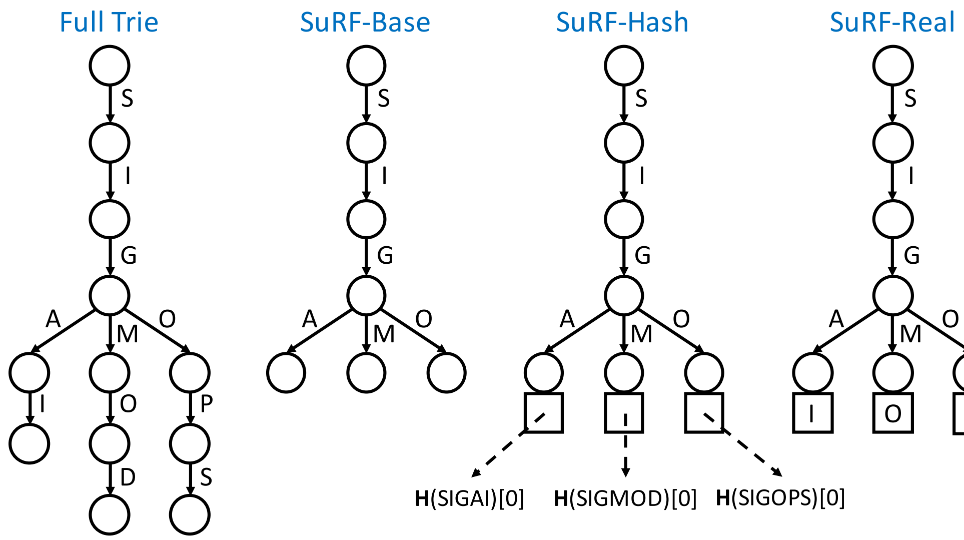


Fig. 4. SuRF Variations. An example of deriving SuRF variations from a full trie.

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 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 4.2. 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, 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, then query keys are likely of the same type. Our results in Section 4.2 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.

3.2 SuRF with Hashed Key Suffixes

As shown in Figure 4, 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 4.2.1 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.

3.3 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 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 $k_s \leq s$, then 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 tradeoff is that using real keys for suffix bits cannot provide as good FPR as using hashed bits, because the distribution correlation between the stored keys and the query keys weakens the distinguishability of the real suffix bits.

3.4 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.

3.5 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, then return false. If the search reaches a leaf, then 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 3.2 to 3.4.

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, then concatenate the real suffix bits to k'. It then compares k' to k. If $k' > k$, then 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.

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*, then 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 FST AND SURF MICROBENCHMARKS

In this section, we first evaluate SuRF and its underlying FST data structure using in-memory microbenchmarks to provide a comprehensive understanding of the filter's strengths and weaknesses. Section 6 creates an example application scenario and evaluates SuRF in RocksDB with end-to-end system measurements.

The Yahoo! Cloud Serving Benchmark (YCSB) [24] 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 with each having 10 physical cores and 20 hardware threads (with hyper-threading enabled), 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.

4.1 FST Evaluation

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 tradeoff. 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 2.6.

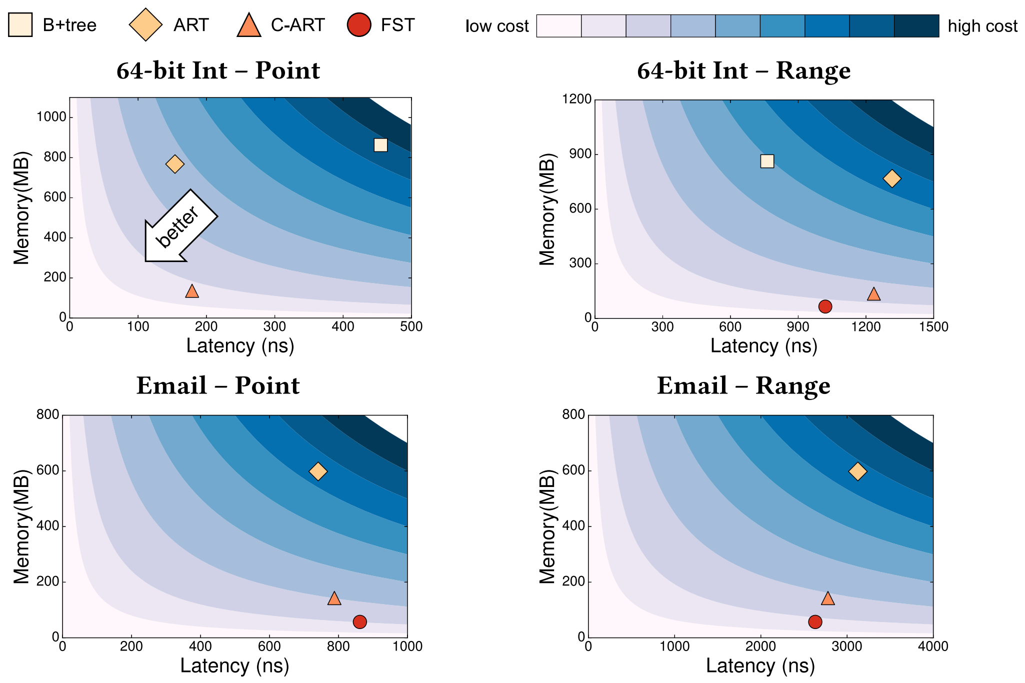


Fig. 5. FST vs. Pointer-based Indexes. Performance and memory comparisons between FST and state-of-the-art in-memory indexes. The blue equi-cost curves indicate a balanced performance-space tradeoff. Points on the same curve are considered "indifferent."

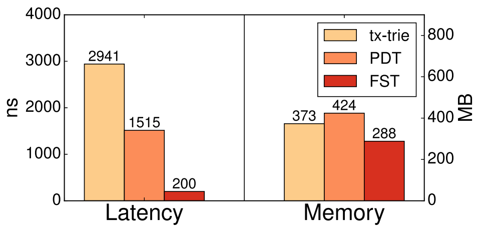
We begin each experiment by bulk-loading a sorted key list into the index. The list contains 50M entries for the integer keys and 25M entries for the email keys. We report the average throughput of 10M point or range queries on the index. The YCSB default range queries are short: most queries scan 50–100 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.

4.1.1 FST vs. Pointer-based Indexes. We examine the following index data structures in our testing framework:

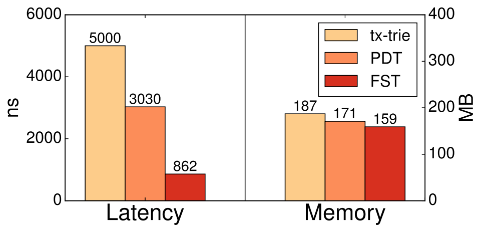
- **B+tree**: This is the most common index structure used in database systems. We use the fast STX B+tree [18] to compare against FST. The node size is set to 512 bytes for best in-memory performance. We tested only with fixed-length keys (i.e., 64-bit integers).
- **ART**: The Adaptive Radix Tree (ART) is a state-of-the-art index structure designed for in-memory databases [44]. 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 constructing a plain ART instance and converting it to a static version [67].

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

Figure 5 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



(a) 64-bit int



(b) Email

Fig. 6. 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).

FST is among the fastest choices in all cases while consuming less space. To better understand this tradeoff, 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 5 using cost function $C = PS$ ($r = 1$), assuming a balanced performance-space tradeoff. 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.

4.1.2 *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 [1]. Its design is similar to LOUDS-Sparse but without any optimizations from Section 2.6.
- **PDT**: The path-decomposed trie [38] is a state-of-the-art succinct trie implementation based on DFUDS (see Section 8). 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 6 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.

4.1.3 *Performance Breakdown.* We next analyze these performance measurements to better understand what makes FST fast. Figure 7 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 [69] 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 2.6.

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%.

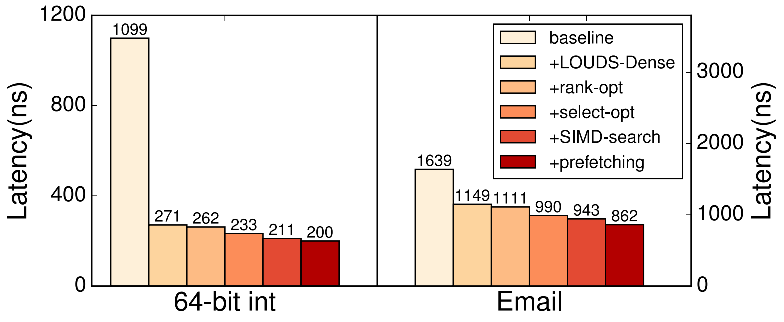


Fig. 7. FST Performance Breakdown. An evaluation on how much LOUDS-Dense and each of the other optimizations speed up FST.

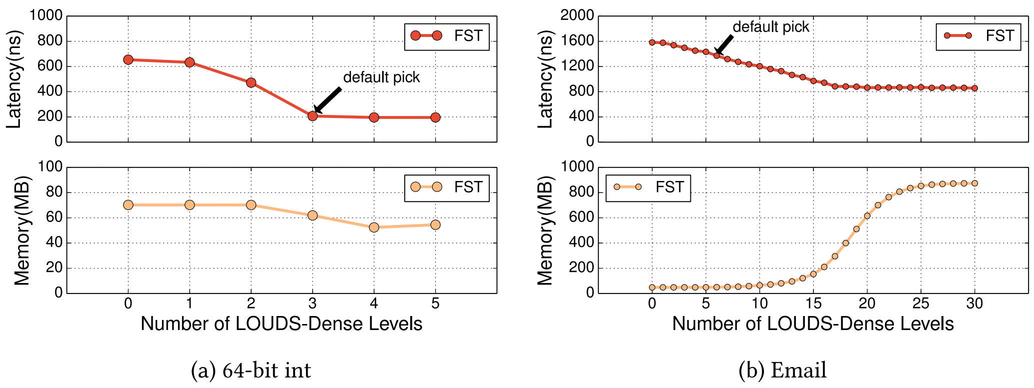


Fig. 8. Tradeoffs between LOUDS-Dense and LOUDS-Sparse. Performance and memory of FST as we increase the number of LOUDS-Dense levels.

4.1.4 *Tradeoffs between LOUDS-Dense and LOUDS-Sparse.* We next examine the performance and memory tradeoffs as we increase the number of LOUDS-Dense levels in FST (controlled by the *R* parameter as described in Section 2.4). Figure 8 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 2.5, 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.

4.2 SuRF Evaluation

The three most important metrics with which to evaluate SuRF are 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 $[K, K(\text{with last byte} ++)]$ (e.g., [org.acm@sigmod, org.acm@sigmoe]) where 52% of the queries return true. For the 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 [2].

4.2.1 False-positive Rate. We first study SuRF's FPR. FPR is the ratio of false positives to the sum of false positives and true negatives. Figure 9 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.

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 diminishing 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 dataset 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, then 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.

4.2.2 Performance. Figure 10 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 2.6. For email keys, the SuRF variants are slower than the Bloom filter because of the overhead of searching/traversing

[2]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.

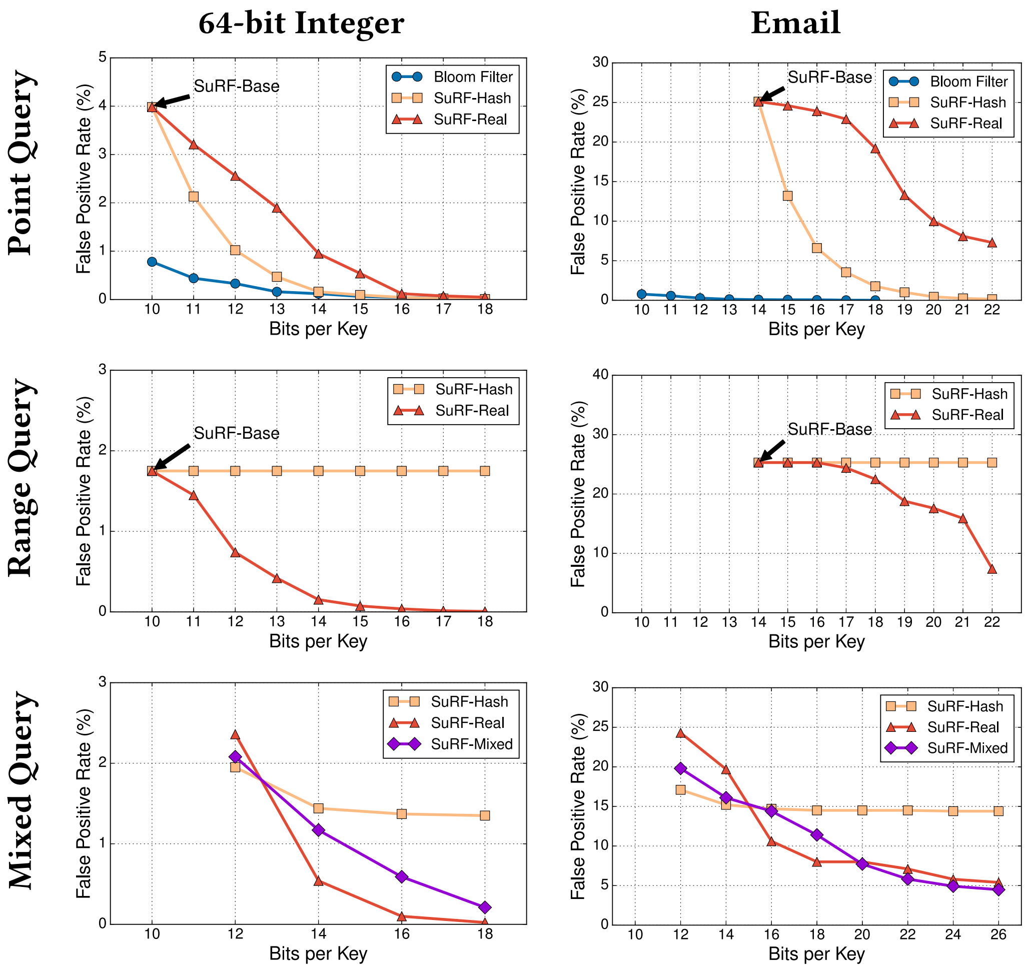


Fig. 9. SuRF False-positive Rate. False-positive rate comparison between SuRF variants and the Bloom filter (lower is better).

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 moving the iterator forward 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

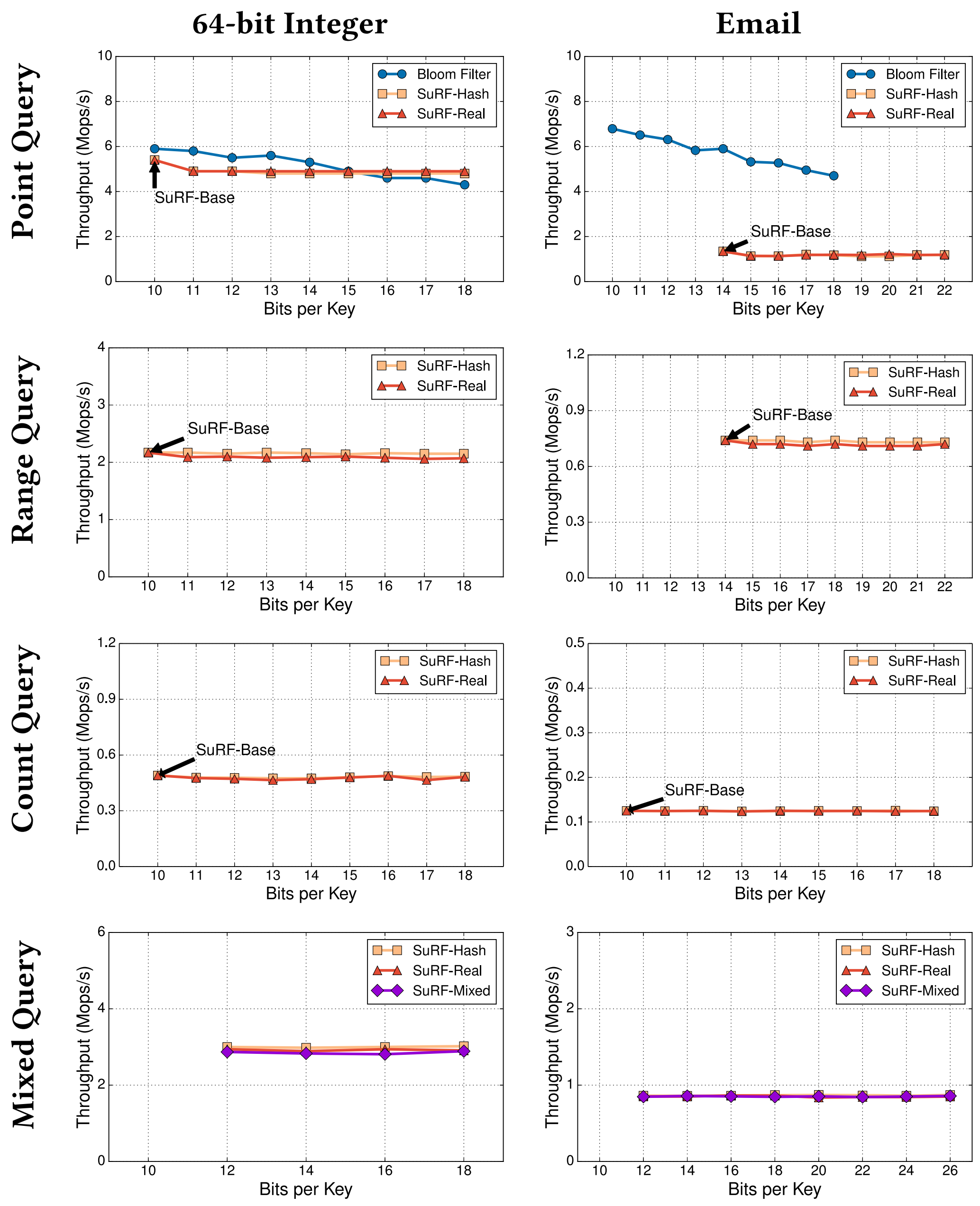


Fig. 10. SuRF Performance. Performance comparison between SuRF variants and the Bloom filter (higher is better).

FPR requirements, then 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; and (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.

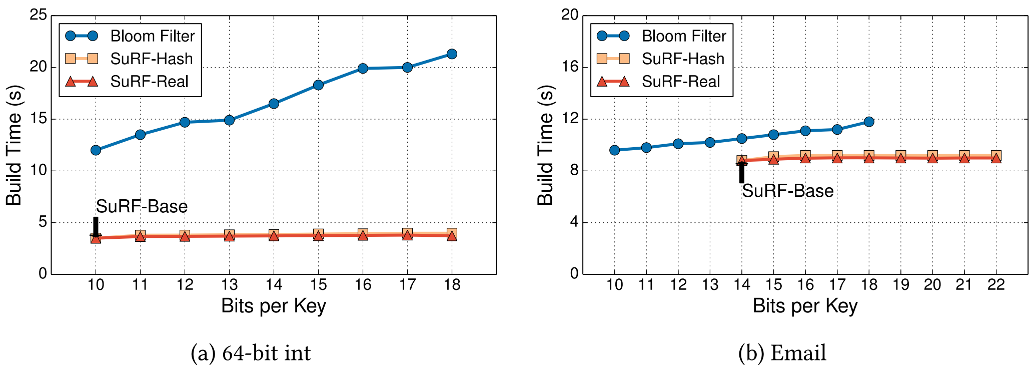


Fig. 11. SuRF Build Time. Build time comparison between SuRF variants and the Bloom filter.

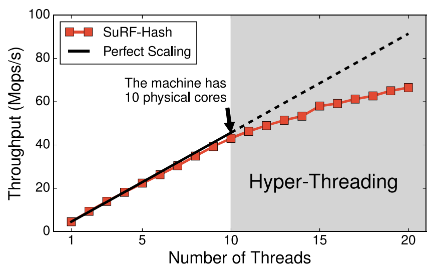


Fig. 12. SuRF Scalability. Point query performance as the number of threads increases.

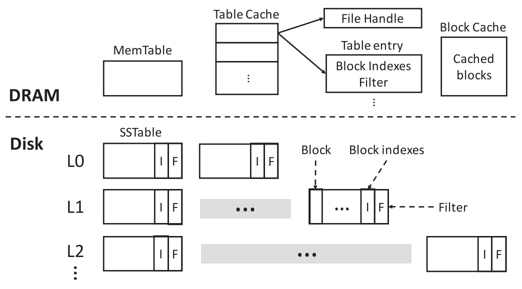


Fig. 13. RocksDB Architecture Overview. RocksDB is implemented based on the log-structured merge tree.

4.2.3 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 11, 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). However, 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.

4.2.4 Multi-threaded SuRF Experiments. 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 12 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 12, 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.

4.2.5 Comparing ARF and SuRF. The Adaptive Range filter (ARF) [13] introduced as part of Project Siberia [29] in Hekaton [26] is the state-of-the-art range filter. An ARF is a simple binary

Table 1. SuRF vs. ARF: Experimental Comparison between ARF and SuRF

	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

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 [9] 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 Section 4.2. 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 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 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 s. SuRF outperformed ARF mainly because ARF is not designed as a general-purpose range filter, but with specific application and scalability goals. Detailed reasons are discussed in Section 8.

Section 6 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. The next section describes the way we use SuRF in RocksDB.

5 EXAMPLE APPLICATION: ROCKSDB

We integrated SuRF with RocksDB as a replacement for its Bloom filter. Figure 13 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 ≥ 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

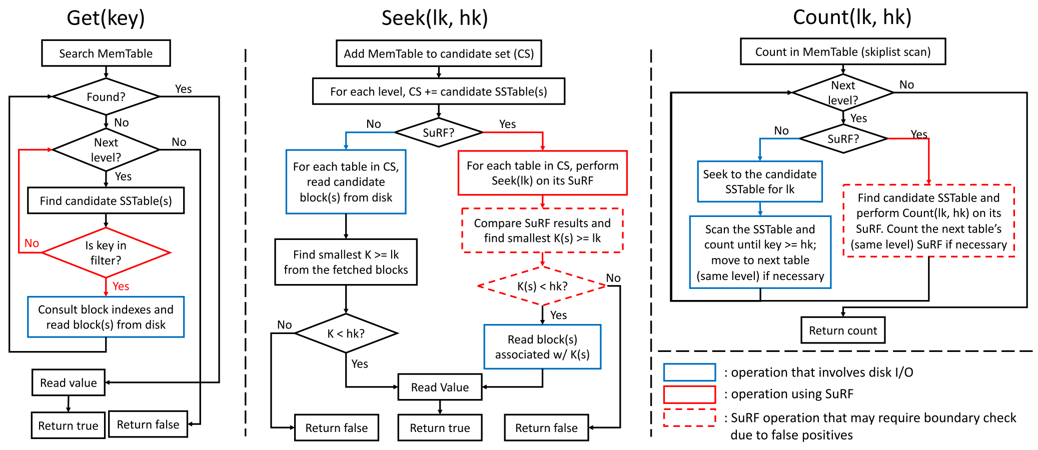

Get(key)
Search MemTable
Found?
Yes
No
Next level?
No
Yes
Find candidate SSTable(s)
No
Is key in filter?
Yes
Consult block indexes and read block(s) from disk
Read value
Return true
Return false
Seek(lk, hk)
Add MemTable to candidate set (CS)
For each level, CS += candidate SSTable(s)
No
SuRF?
Yes
For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

Fig. 14. RocksDB Query Execution Flowcharts. Execution paths for Get, Seek, and Count queries in RocksDB.

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 14 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, then 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, then 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 containing 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 $K \leq hk$), then RocksDB fetches exactly one block from the selected SSTable that contains the global minimum K. If the query fails (i.e., $K > hk$), then 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 3.5), 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, then 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 6.

To illustrate how SuRF s 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, then the counting is carried out by iterating in the filter(s) (refer to the *count* operation in SuRF). 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.

6 SYSTEM EVALUATION

Time-series databases often use RocksDB or similar LSM-tree designs for the storage engine. Examples are InfluxDB [11], QuasarDB [8], LittleTable [57], and Cassandra-based systems [7, 43]. 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 s. Each sensor operates for 10K s and records ~50K events. The starting timestamp for each sensor is randomly generated within the first 0.2 s. 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 earliest event after that time.

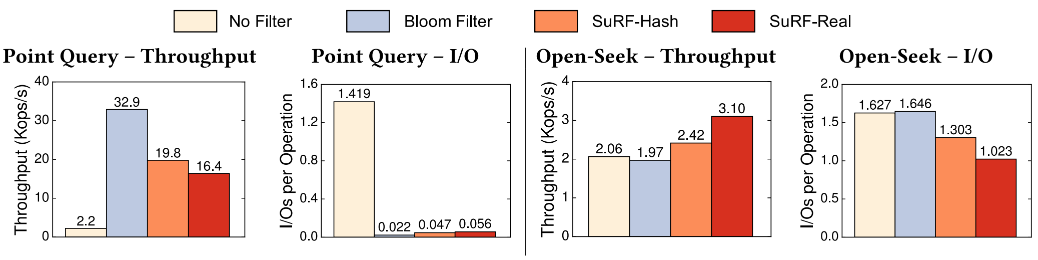


Fig. 15. Point and Open-Seek Queries. RocksDB point query and Open-Seek query evaluation under different filter configurations.

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

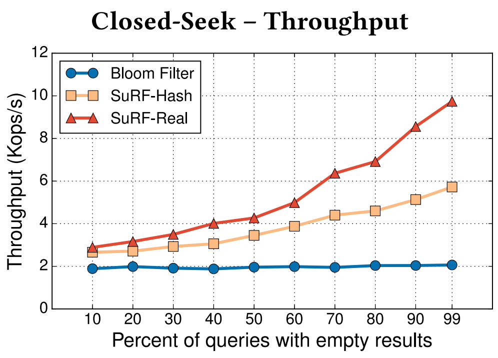
Our test machine has an Intel Core i7-6770HQ CPU, 32 GB RAM, and an Intel 540s 480 GB SSD. We use Snappy (RocksDB's default) for data compression. The resulting RocksDB instance has four levels (including Level 0) and uses 52 GB of disk space. We configured[3] RocksDB according Facebook's recommendations [28, 30].

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-distributed point queries to existing keys so that every SSTable is touched approximately 1,000 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 15 (left two figures) 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 (Levels 1, 2, and 3). Without filters, point queries incur approximately 1.5 I/Os per operation according to Figure 15, 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 [27].

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 4.2.1). SuRF-Real

[3]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`.



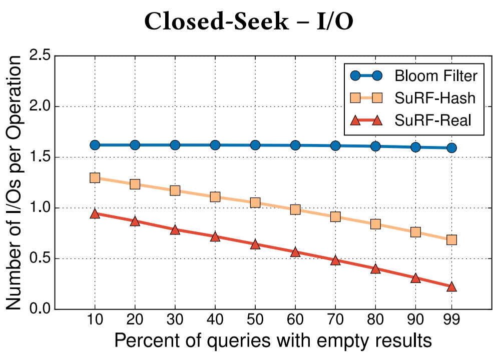


Fig. 16. Closed-Seek Queries. RocksDB Closed-Seek query evaluation under different filter configurations and range sizes.

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 15 (right two figures) shows that using SuRF-Real can speed up Open-Seek queries by 50%. SuRF-Real cannot improve further, because an Open-Seek query requires reading at least one SSTable block as described in Section 5, and that SSTable block read is likely to occur at the last level where the data blocks are not available in cache. In fact, the I/O figure (rightmost) shows that using SuRF-Real reduces the number of I/Os per operation to 1.023, which is close to the maximum I/O reduction for Open-Seeks.

Figure 16 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 14). 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.

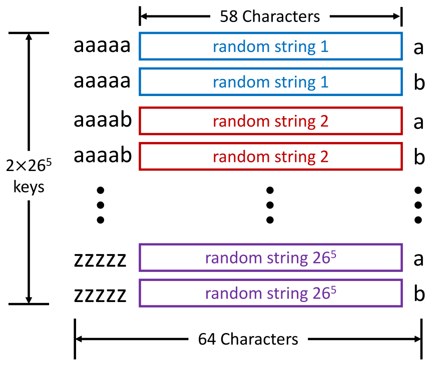


Fig. 17. Worst-case Dataset. A worst-case dataset for SuRF in terms of performance and space-efficiency.

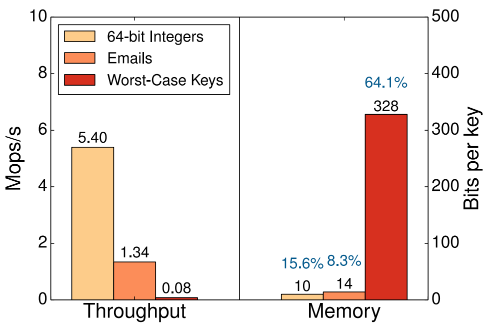


Fig. 18. Worst-case Evaluation. SuRF's throughput and memory consumption on a worst-case dataset. The percentage numbers on the right are the size ratios between SuRF and the raw keys for each dataset.

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.

7 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 17. 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 SuRF microbenchmarks in Section 4.2. 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 4.2) so that every query reaches a leaf node in SuRF (i.e., no early exit) to allow the worst-case performance.

Figure 18 shows the throughput and memory results. We include the numbers for 64-bit integers and emails obtained from Section 4.2.2 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. [36] 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 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 [23]. 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 of 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 5. SuRF is a natural fit for LSM tree designs: when compaction creates a new SSTable, simply rebuild its associated SuRF. For applications that require modifiable range filters, one can extend SuRF using a hybrid index [67]. 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. To support *delete*, SuRF can use 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*. Periodic garbage collection is needed to keep SuRF small. Supporting dynamic operations in SuRF is beyond the scope of this article, and we defer the problem to future work.

8 RELATED WORK

Alternatives to the FST for range filtering. The Bloom filter [20] and its major variants [21, 32, 53] are compact data structures designed for fast approximate membership tests. They are widely used in storage systems, especially LSM trees as described in the introduction, to reduce expensive disk I/O. Similar applications can be found in distributed systems to reduce network I/O [2, 60, 66]. The downside for Bloom filters, and other filters such as Quotient filters [16], Cuckoo filters [31] and Morton filters [22], 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 [18, 19, 44, 63] for the task, but the memory cost is high (see evaluation in Section 4.1). In practice, people often use prefix Bloom filters to help answer range-emptiness queries. For example, RocksDB [6], LevelDB [3], and LittleTable [57] 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) [13] was introduced as part of Project Siberia [29] in Hekaton [26] 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, then 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, however, assumes nothing about workloads. It can be used as a Bloom filter replacement but with range filtering ability, as shown in Section 6. 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 compare ARF and SuRF in Appendix 4.2.5.

LSM-trees. Many modern key-value stores adopt the log-structured merge tree (LSM-tree) design [52] for its high write throughput and low space amplification. Such systems include LevelDB [3], RocksDB [6], Cassandra [43, 61], HBase [4], WiredTiger [64], and cLSM [34] from Yahoo Labs. Monkey [25] 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 [28]. These optimizations fall under the RUM Conjecture [15]: 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 [65] improves read and write throughput over LevelDB for small key-value pairs, but it does not support range queries.

Why we chose LOUDS vs. alternatives. A second type of succinct tree representation, besides LOUDS, is based on "balanced parentheses" (BP) sequences [47]. 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 [33, 45, 48, 49]. The most recent "fully functional" representation [58] 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 [14]. 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 [17, 38, 55] are based on a third type of succinct tree representation that combines LOUDS and BP, called the Depth-First Unary Degree Sequence (DFUDS) [17]. 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 [38] provided a state-of-the-art succinct trie implementation based on DFUDS, which we compare against in Section 4.1.2.

Other systems applications of succinct data structures. Succinct [12] and follow-up work BlowFish [42] 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 [39] 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 dataset fits in DRAM.

9 CONCLUSION

This article introduces the SuRF filter structure, which supports approximate membership tests for single keys and ranges, and counting queries. SuRF is built upon a new succinct data structure, called the Fast Succinct Trie (FST), that requires only 10 bits per node to encode the trie. FST is engineered to have performance equivalent to state-of-the-art pointer-based indexes. SuRF is memory efficient, and its space/false-positive rates can be tuned by choosing different amounts of suffix bits to include. We have shown through extensive microbenchmarks both FST's advantages over the state of the art succinct tries, and where SuRF fits in the space/time tradeoff space. Replacing the Bloom filters with SuRFs of the same size in RocksDB, substantially reduced I/O and improved throughput for range queries with a modest cost on the worst-case point query throughput. We believe, therefore, that SuRF is a promising technique for optimizing future storage systems, and more. SuRF's source code is publicly available at https://github.com/efficient/SuRF.

REFERENCES

[1] 2010. tx-trie 0.18—Succinct Trie Implementation. Retrieved from https://github.com/hillbig/tx-trie.
[2] 2013. Squid Web Proxy Cache. Retrieved from http://www.squid-cache.org/.
[3] 2014. Google LevelDB. Retrieved from https://github.com/google/leveldb.
[4] 2015. Apache HBase. Retrieved from https://hbase.apache.org/.
[5] 2015. Facebook MyRocks. Retrieved from http://myrocks.io/.
[6] 2015. Facebook RocksDB. Retrieved from http://rocksdb.org/.
[7] 2015. KairosDB. Retrieved from https://kairosdb.github.io/.
[8] 2015. QuasarDB. Retrieved from https://en.wikipedia.org/wiki/Quasardb.
[9] 2016. ARF Implementation. Retrieved from https://github.com/carolinux/adaptive_range_filters.
[10] 2016. Succinct Data Structures. Retrieved from https://en.wikipedia.org/wiki/Succinct_data_structure.
[11] 2017. The InfluxDB Storage Engine and the Time-Structured Merge Tree (TSM). Retrieved from https://docs.influxdata.com/influxdb/v1.0/concepts/storage_engine/.
[12] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 337–350.
[13] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive range filters for cold data: Avoiding trips to siberia. *Proc. VLDB Endow.* 6, 14 (2013), 1714–1725.
[14] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. 2010. Succinct trees in practice. In *Proceedings of the Algorithm Engineering and Experiments Conference (ALENEX'10)*. 84–97.
[15] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing access methods: The RUM conjecture. In *Proceedings of the Extended Database Technology Conference (EDBT'16)*, Vol. 2016. 461–466.
[16] 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. 2012. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (2012), 1627–1637.
[17] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing trees of higher degree. *Algorithmica* 43, 4 (2005), 275–292.
[18] Timo Bingmann. 2008. STX B+ Tree C++ Template Classes. Retrieved from http://idlebox.net/2007/stx-btree/.
[19] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 521–534.
[20] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.
[21] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *Proceedings of the European Symposium on Algorithms*. Springer, 684–695.
[22] Alex D. Breslow and Nuwan S. Jayasena. 2018. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.* 11, 9 (2018), 1041–1055.
[23] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. 1978. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*. ACM, 59–65.
[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154.

[25] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 79–94.
[26] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1243–1254.
[27] Siying Dong. 2017. (personal communication, August 28, 2017).
[28] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing space amplification in rocksDB. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'17)*, Vol. 3. 3.
[29] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proc. VLDB Endow.* 7, 11 (2014), 931–942.
[30] Facebook. 2015. RocksDB Tuning Guide. Retrieved from https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.
[31] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. ACM, 75–88.
[32] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8, 3 (2000), 281–293.
[33] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. 2006. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.* 368, 3 (2006).
[34] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the 10th European Conference on Computer Systems*. ACM, 32.
[35] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Proceedings of the International Conference on Experimental Algorithms (WEA'05)*. 27–38.
[36] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2014. Approximate range emptiness in constant time and optimal space. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 769–775.
[37] Roberto Grossi and Giuseppe Ottaviano. 2013. Design of practical succinct data structures for large data collections. In *Proceedings of the Software Engineering Assembly (SEA'13)*.
[38] Roberto Grossi and Giuseppe Ottaviano. 2014. Fast compressed tries through path decompositions. *J. Exp. Algor.* 19 1, Article 1.8 (October 2014), 20 pages.
[39] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.* 35, 2 (2005), 378–407.
[40] Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Foundations of Computer Science*. IEEE, 549–554.
[41] Stelios Joannou and Rajeev Raman. 2012. Dynamizing succinct tree representations. In *International Symposium on Experimental Algorithms*. Springer, 224–235.
[42] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2016. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*.
[43] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operat. Syst. Rev.* 44, 2 (2010), 35–40.
[44] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. IEEE, 38–49.
[45] Hsueh-I. Lu and Chia-Chi Yeh. 2008. Balanced parentheses strike back. *Trans. Algor.* 4, 3 (2008), 28.
[46] Miguel Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. 2016. Practical compressed string dictionaries. *Inf. Syst.* 56 (2016), 73–108. http://repositorio.uchile.cl/bitstream/handle/2250/138288/Practical-compressed-string-dictionaries.pdf;sequence=1.
[47] J. Ian Munro and Venkatesh Raman. 2001. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.* 31, 3 (2001), 762–776.
[48] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. 2001. Space efficient suffix trees. *J. Algor.* 39, 2 (2001), 205–222.
[49] J. Ian Munro and S. Srinivasa Rao. 2004. Succinct representations of functions. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP'04)*.
[50] Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach*. Cambridge University Press.
[51] Gonzalo Navarro and Eliana Providel. 2012. Fast, small, simple rank/select on bitmaps. In *Proceedings of the Software Engineering Assembly (SEA'12)*. 295–306.
[52] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), 351–385.

[53] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash-and space-efficient bloom filters. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*. Springer, 108–121.
[54] Naila Rahman, Rajeev Raman, et al. 2006. Engineering the LOUDS succinct tree representation. In *Proceedings of the International Conference on Experimental Algorithms (WEA'06)*. 134–145.
[55] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *Trans. Algor.* 3, 4 (2007), 43.
[56] Rajeev Raman and S. Srinivasa Rao. 2013. Succinct representations of ordinal trees. In *Space-efficient Data Structures, Streams, and Algorithms*. Springer, 319–332.
[57] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. LittleTable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 125–138.
[58] Kunihiko Sadakane and Gonzalo Navarro. 2010. Fully-functional succinct trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*.
[59] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 217–228.
[60] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: An aid to network processing. *ACM SIGCOMM Comput. Commun. Rev.* 35, 4 (2005), 181–192.
[61] The Apache Software Foundation. 2015. Apache Cassandra. Retrieved from https://cassandra.apache.org/.
[62] Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08)*. 154–168.
[63] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 473–488.
[64] WiredTiger. 2014. WiredTiger. Retrieved from http://www.wiredtiger.com/.
[65] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. 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*. USENIX Association, 71–82.
[66] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. ACM, 313–324.
[67] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1567–1581.
[68] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 323–336.
[69] Dong Zhou, David G. Andersen, and Michael Kaminsky. 2013. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Proceedings of the International Symposium on Experimental Algorithms*. Springer, 151–163.

Received October 2018; revised October 2019; accepted October 2019