SuRF: Practical Range Query Filtering with Fast Succinct Tries

Huanchen Zhang
Carnegie Mellon University
huanche1@cs.cmu.edu

Hyeontaek Lim
Carnegie Mellon University
hl@cs.cmu.edu

Viktor Leis
TU München
leis@in.tum.de

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Michael Kaminsky
Intel Labs
michael.e.kaminsky@intel.com

Kimberly Keeton
Hewlett Packard Enterprise
kimberly.keeton@hpe.com

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

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.

ACM Reference Format:
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 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3183713.3196931

1 INTRODUCTION

Write-optimized log-structured merge (LSM) trees [44] are popular low-level storage engines for general-purpose databases that provide fast writes [5, 50] and ingest-abundant DBMSs such as time-series databases [11, 48]. One of their main challenges for fast query processing is that items could reside in immutable files (SSTables) from all levels [3, 36]. Item retrieval in an LSM tree-based design may therefore incur multiple expensive disk I/Os [44, 50]. This challenge calls for in-memory data structures that can help locate query items. *Bloom filters* are a good match for this task [48, 50] because they are small enough to reside in memory, and they have only "one-sided" errors—if the key is present, then the Bloom filter returns true; if the key is absent, then the filter will likely return false, but might incur a false positive.

Although Bloom filters are useful for single-key lookups ("Is key 42 in the SSTable?"), they cannot handle range queries ("Are there keys between 42 and 1000 in the SSTable?"). With only Bloom filters, an LSM tree-based storage engine must read additional table blocks from disk for range queries. Alternatively, one could maintain an auxiliary index, such as a B+Tree, to support such range queries. The I/O cost of range queries is high enough that LSM tree-based designs often use *prefix Bloom filters* to optimize certain fixed-prefix queries (e.g., "where email starts with com.foo@") [6, 25, 48], 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 [24]. A handful of approximate data structures, including the prefix Bloom filter, exist that accelerate specific categories of range queries, but none is general purpose.

This paper 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 trade-off 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 [37]) 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 paper) 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 (without upper-bound) by 1.5× and closed-range queries (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 using SuRFs instead of Bloom filters because they have higher (0.2% vs. 0.1%) false positive rates. One can eliminate this performance gap by increasing the size of SuRFs by a few bits per key.

This paper makes three primary contributions. First, we describe in Section 2 our FST data structure whose space consumption is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196931

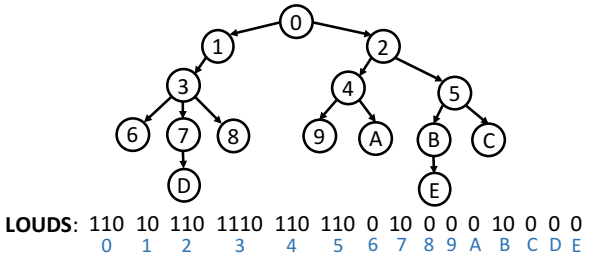


Figure 1: An example ordinal tree encoded using LOUDS

close to the minimum number of bits required by information theory yet has performance equivalent to uncompressed order-preserving indexes. Second, in Section 3 we describe how to use the FST to build 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.

2 FAST SUCCINCT TRIES

The core data structure in SuRF is the FST. It is a space-efficient, static trie that answers point and range queries. FST is 4–15× faster than earlier succinct tries using other tree representations [15, 17, 31, 32, 34, 39, 40, 46, 49], achieving performance comparable to or better than the state-of-the-art pointer-based indexes.

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.

The contribution of FST is two-fold. Although the algorithmic designs of LOUDS-Dense and LOUDS-Sparse are not new, the combination (we call the hybrid encoding scheme **LOUDS-DS**) within the same data structure is. As shown in Section 4.1, LOUDS-DS is key to achieving high performance while remaining succinct. Second, 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

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

[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.

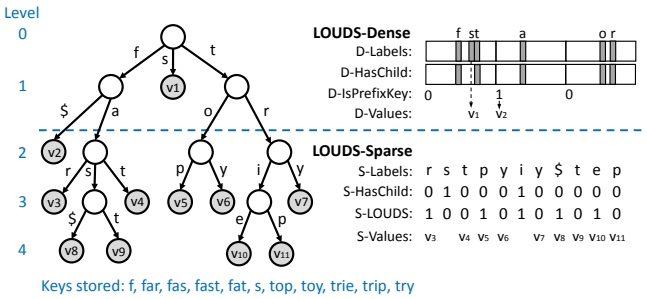


Figure 2: LOUDS-DS Encoded Trie – The $ symbol represents the character whose ASCII number is `0xFF`. It is used to indicate the situation where a prefix string leading to a node is also a valid key.

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 towards succinct tries.

Jacobson [34] pioneered research on succinct tree representations and introduced the ***L**evel-**O**rdered **U**nary **D**egree **S**equence* (LOUDS) to encode an ordinal tree. As the name suggests, LOUDS traverses the nodes in a breadth-first order and encodes each node's degree using the unary code. For example, node 3 in Figure 1 has three children and is thus encoded as `'1110'`.

Navigating a tree encoded with LOUDS uses the rank & select primitives. Given a bit vector, $rank_1(i)$ counts the number of 1's up to position i ($rank_0(i)$ counts 0's), while $select_1(i)$ returns the position of the i-th 1 ($select_0(i)$ selects 0's). Modern rank & select implementations [30, 43, 53, 58] achieve constant time by using look-up tables (LUTs) to store a sampling of precomputed results so that they only need to count between the samples.

With 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. We assume that both node/child numbers and bit positions are zero-based:

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

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 i-th bit in the bitmap, where $0 \le i \le 255$, 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 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.

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*. To traverse down the trie, given *pos* where *D-HasChild*[*pos*] = 1, **D-ChildNodePos**(*pos*) = 256 $\times rank_1$(*D-HasChild*, *pos*) computes the bit position of the first child node. To move up the trie, **D-ParentNodePos**(*pos*) = 256 $\times select_1$ (*D-HasChild*, $\lfloor pos/256 \rfloor$) computes the bit position of the parent node. To access values, given *pos* where *D-HasChild*[*pos*] = 0, **D-ValuePos**(*pos*) = $rank_1$(*D-Labels*, *pos*) - $rank_1$(*D-HasChild*, *pos*) + $rank_1$(*D-IsPrefixKey*, $\lfloor pos/256 \rfloor$)-1 gives the lookup position.

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.

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**, on the other hand, points to a value. 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.

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: to move down the trie, **S-ChildNodePos**(*pos*) = $select_1$(*S-LOUDS*, $rank_1$(*S-HasChild*, *pos*) + 1); to move up, **S-ParentNodePos**(*pos*) = $select_1$(*S-HasChild*, $rank_1$(*S-LOUDS*, *pos*) - 1); to access a value, **S-ValuePos** (*pos*) = *pos* - $rank_1$(*S-HasChild*, *pos*) - 1.

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 levels, favoring performance over space. We use R=64 as the default.

LOUDS-DS supports three basic operations efficiently:

- **ExactKeySearch**(*key*): Return the value of *key* if *key* exists (or 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, it continues into the LOUDS-Sparse levels. The high-level searching steps at each level are similar regardless of the encoding mechanism: First, search the current node's range in the label sequence for the target key byte. If the key byte does not exist, terminate and return *NULL*. Otherwise, check the corresponding bit in the *HasChild* bit-sequence. If the bit is 1 (i.e., the branch points to a child node), compute the child node's starting position in the label sequence and continue to the next level. Otherwise, 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.

Range queries use a high-level algorithm similar to the point query implementation. When performing LowerBound, instead of doing an exact search in the label sequence, the algorithm searches for the smallest label $\geq$ the target label. When moving to the next key, the cursor starts at the current leaf label position and moves forward. If another valid label **l** is found within the node, the algorithm finds the left-most leaf key in the subtree rooted at **l**. If the cursor hits node boundary instead, the algorithm moves the cursor up to the corresponding position in the parent node.

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,

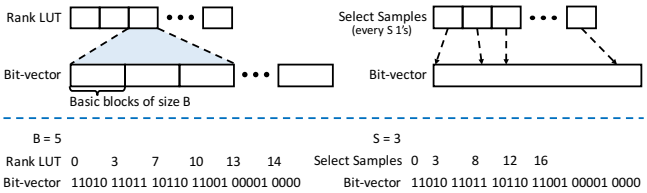


Figure 3: Rank and select structures in FST

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 shows that range queries in FST are even faster than pointer-based tries.

LOUDS-DS can be built using one scan over a key-value list.

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) is approximately $9.44n$ bits. Although the space taken by LOUDS-Sparse is close to the information-theoretic bound, technically, LOUDS-Sparse can only be categorized as *compact* rather than *succinct* in a finer 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 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, 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, rank and select on all bit vectors and label searching at LOUDS-Sparse levels. We next describe optimizations for these critical operations.

2.6 Optimizations

We focus on 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 [58], 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, $rank_1(i)$ = 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 [58].

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, $select_1(i)$ = LUT$[i/S]$ + (selecting the $(i - i/S \times S)$th set bit starting from position LUT$[i/S]$ + 1) + 1. For example, to compute $select_1(8)$, we first look up slot $8/3 = 2$ in the LUT and get 8. We then select the $(8 - 8/3 \times 3) = 2$nd set bit starting from position $LUT[8/3] + 1 = 9$) by binary searching in the third basic block using *popcount*. This select equals 1. The final result for $select_1(8)$ is $8 + 1 + 1 = 10$.

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 (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, relevant addresses in other sequences can be computed and prefetched for later use.

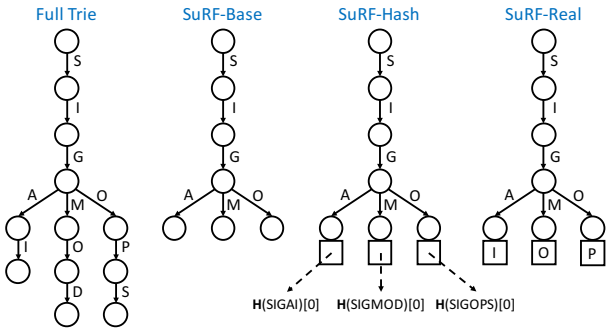


Figure 4: An example of deriving SuRF variations from a full trie.

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 Appendix A.

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.

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 than 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. In practice, however, query keys are almost always correlated to the stored keys. For example, if a SuRF-Base stores email addresses, query keys are likely of the same type. Our results in Section 4.2 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 $k_s \leq s$, the iterator points to the current key; otherwise, it advances to the next key in the trie.

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

3.4 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.

result = ***lookup****(k)*: Perform a point query on k. Returns true if k may exist (could be false positive); false guarantees non-existence. This operation first searches for k in the FST. If the search terminates without reaching a leaf, return false. If the search reaches a leaf, return true in SuRF-Base. In other SuRF variants, fetch the stored key suffix k_s of the leaf node and perform an equality check against the suffix bits extracted from k according to the suffix type as described in Sections 3.2 to 3.4.

iter, fp_flag = ***moveToNext****(k)*: Return an iterator pointing to the smallest key that is $\geq k$. The iterator supports retrieving the next and previous keys in the filter. This operation performs a *LowerBound* search on the FST to reach a leaf node. If an approximation occurs in the search (i.e., a key-byte at certain level does not exist in the trie and it has to move to the next valid label), then the function sets the *fp_flag* to false and returns the current iterator. Otherwise, the prefix of k matches that of a stored key (k') in the trie. SuRF-Base and SuRF-Hash do not have auxiliary suffix bits that can determine the order between k and k'; they have to set the *fp_flag* to true and return the iterator pointing to k'. SuRF-Real and SuRF-Mixed include the real suffix bits k'_r for k' to further compare to the corresponding real suffix bits k_r for k. If $k'_r > k_r$, *fp_flag* = false and return the current iterator; If $k'_r = k_r$, *fp_flag* = true and return the current iterator; If $k'_r < k_r$, *fp_flag* = false and return the advanced iterator (iter++).

count, low_fp_flag, high_fp_flag = ***count****(lowKey, highKey)*: Return the number of keys contained in [*lowKey*, *highKey*]. The *low_fp_flag* and the *high_fp_flag* indicate the possibility of over-counting each of the boundary keys (when the prefix of a boundary key matches a stored key, SuRF cannot decide whether the boundary key should be included in the count). This operation performs a *moveToNext* on *lowKey*, setting *low_fp_flag* if necessary. It then advances the iterator and increments the count until the key k' pointed to by the iterator is greater than or equal to *highKey*. If k' is a prefix of *highKey*, set the *high_fp_flag*. Finally, return the count.

4 FST & 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) [21] 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, 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 trade-off. Second, we compare FST to two alternative succinct trie implementations. We show that FST is 4–15× faster while also using less memory. Finally, we present a performance breakdown of our optimization techniques described in Section 2.6.

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

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 structures 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 [37]. 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 [57].

ART, C-ART, and FST store only unique key prefixes in this experiment as described in Section 3.1. 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 FST is among the fastest choices in all cases while consuming less space. To better understand this trade-off, we define a cost function $C = P^r S$, where P represents performance (latency), and S represents space (memory). The exponent r indicates the relative importance between P and S. $r > 1$ means that the application is performance critical, and $0 < r < 1$ suggests otherwise. We define an "indifference curve" as a set of points in the performance-space map that have the same cost. We draw the equi-cost curves in Figure 5 using cost function $C = PS$ ($r = 1$), assuming a balanced performance-space trade-off. We observe that FST has the lowest cost (i.e., is most efficient) in all cases. In order for the second place (C-ART) to have the same cost as FST in the first subfigure, for example, r needs to be 6.7 in the cost function, indicating an extreme preference for performance.

4.1.2 FST vs. Other Succinct Tries. We compare FST against the following alternatives:

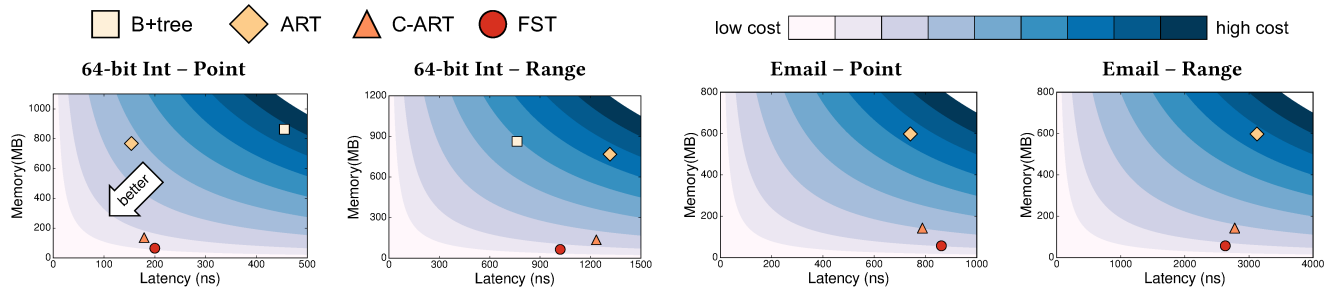
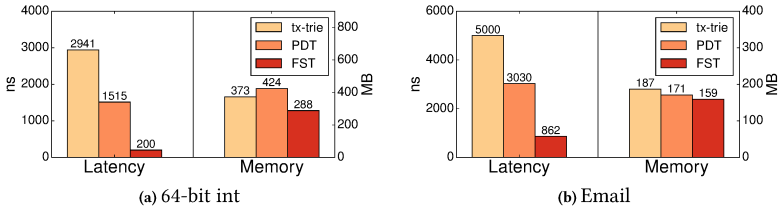


Figure 5: FST vs. Pointer-based Indexes



(a) 64-bit int (b) Email

Figure 6: FST vs. Other Succinct Tries

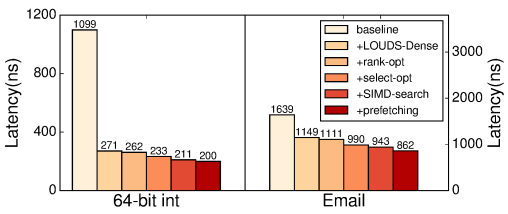


Figure 7: FST Performance Breakdown

- **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 [32] is a state-of-the-art succinct trie implementation based on DFUDS (see Section 7). PDT re-balances the trie using path-decomposition techniques and achieves good 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 [58] 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 FST is fast because of the LOUDS-DS design. The introduction of LOUDS-Dense to the upper-levels of the trie provides a significant performance boost at a negligible space cost. The rest of the optimizations reduce the overall query latency by 3–12%.

4.2 SuRF Evaluation

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

4.2.1 False Positive Rate. We first study SuRF's false positive rate (FPR). FPR is the ratio of false positives to the sum of false positives and true negatives. Figure 8 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

[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.

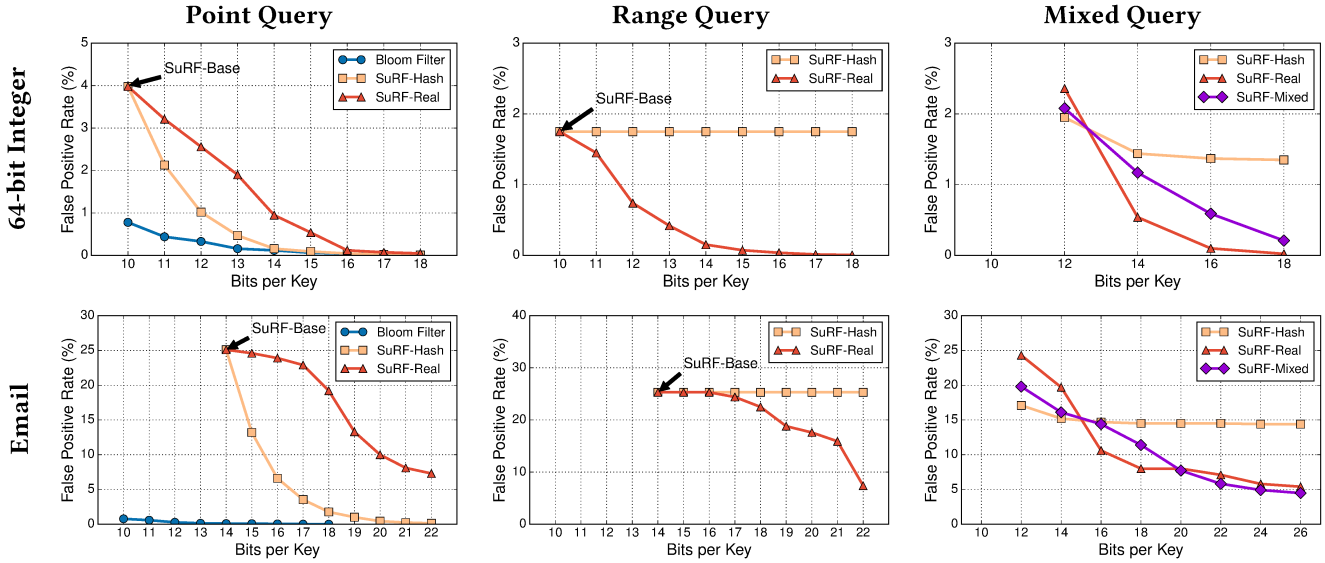


Figure 8: False positive rate comparison between SuRF variants and the Bloom filter (lower is better)

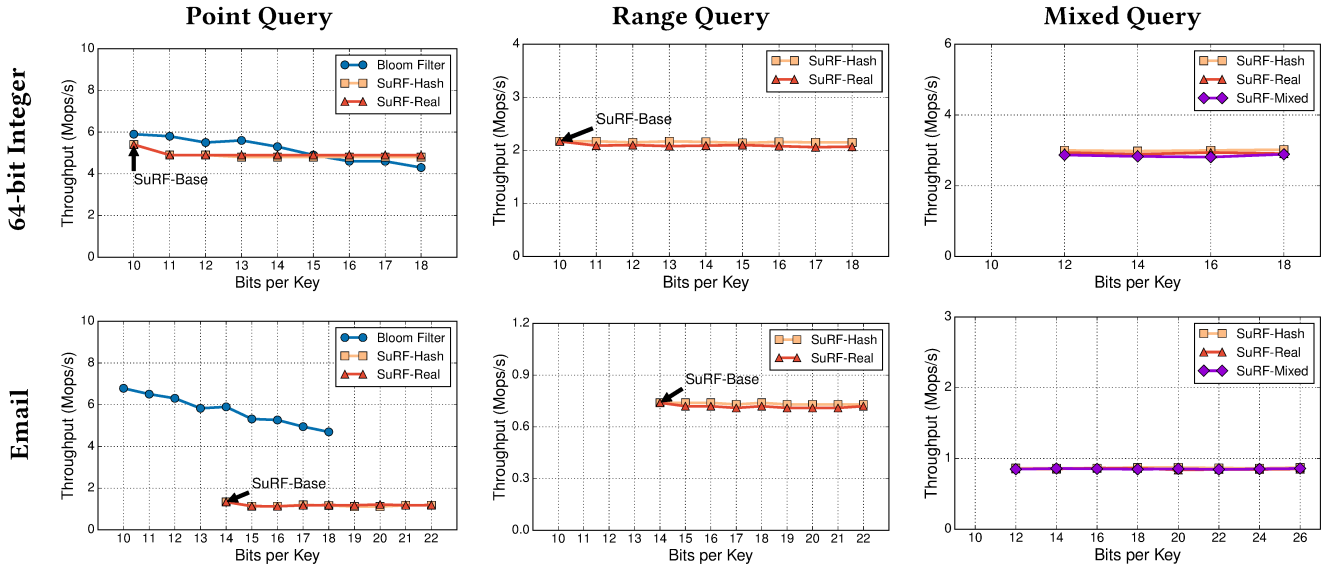


Figure 9: Performance comparison between SuRF variants and the Bloom filter (higher is better)

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

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

4.2.2 Performance. Figure 9 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 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

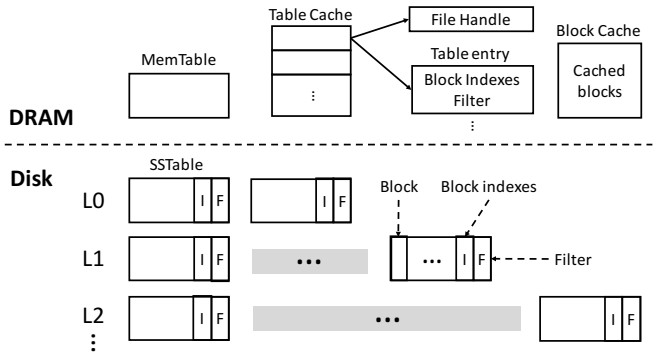


Figure 10: An overview of RocksDB architecture.

fetch the suffix bits. Range queries in SuRF are slower than point queries because every query needs to walk down to the bottom of the trie (no early exit). In addition, the construction speed of SuRF is comparable to that of the Bloom filter (not shown in figure) because they can be built in a single scan of the sorted input keys.

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

Section 6 shows the evaluation of SuRF in the context of an end-to-end 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 10 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 an index.

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

To facilitate searching and to reduce I/Os, RocksDB includes two types of buffer caches: the table cache and the block cache. The table cache contains meta-data about open 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. We also implemented a new approximate *Count* query that returns the number of entries in a key range. The approximate count may over-count the deletion and modification entries, because it cannot distinguish update/delete records from insert records. If the dataset is static, the count is accurate. This shows that SuRF supports functionality beyond filtering.

Figure 11 shows the execution paths for *Get*, *Seek*, and *Count* queries in RocksDB. *Next*'s core algorithm is similar to *Seek*. We use colors to highlight the potential I/O reduction by using filters. Operations in blue boxes can trigger I/O if the requesting block(s) are not cached. Filter operations are in red boxes. If the box is dashed, checks (by fetching the actual keys from SSTables) for boundary keys might be necessary due to false positives.

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

For *Seek(lk, hk)*, if *hk* (high key) is not specified, we call it an *Open Seek*. Otherwise, we call it a *Closed Seek*. To implement *Seek(lk, hk)*, RocksDB first collects the candidate SSTables from all levels by searching for *lk* (low key) in the block indexes.

Absent SuRFs, RocksDB examines each candidate SSTable and fetches the block containing the smallest key that is ≥ *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)* query on its SuRF to avoid the one I/O per SSTable. If the query succeeds (i.e., Open Seek or $K \leq hk$), RocksDB fetches exactly one block from the selected SSTable that contains the global minimum K. If the query fails (i.e., $K > hk$), no I/O is involved. Because SuRF's moveToNext query returns only a key prefix K_p, three additional checks are required to guarantee correctness. First, if the moveToNext query sets the false positive flag, RocksDB must fetch the complete key K from the SSTable block to determine whether $K \geq lk$. If not set, RocksDB fetches the next key after K. Second, if K_p is a prefix of *hk*, the complete key K is also needed to verify $K \leq hk$. If not, the current SSTable is skipped. 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 SuRFs benefit range queries, suppose a RocksDB instance has three levels (L_N, L_{N-1}, L_{N-2}) of SSTables on disk. L_N has an SSTable block containing keys 2000, 2011, 2020

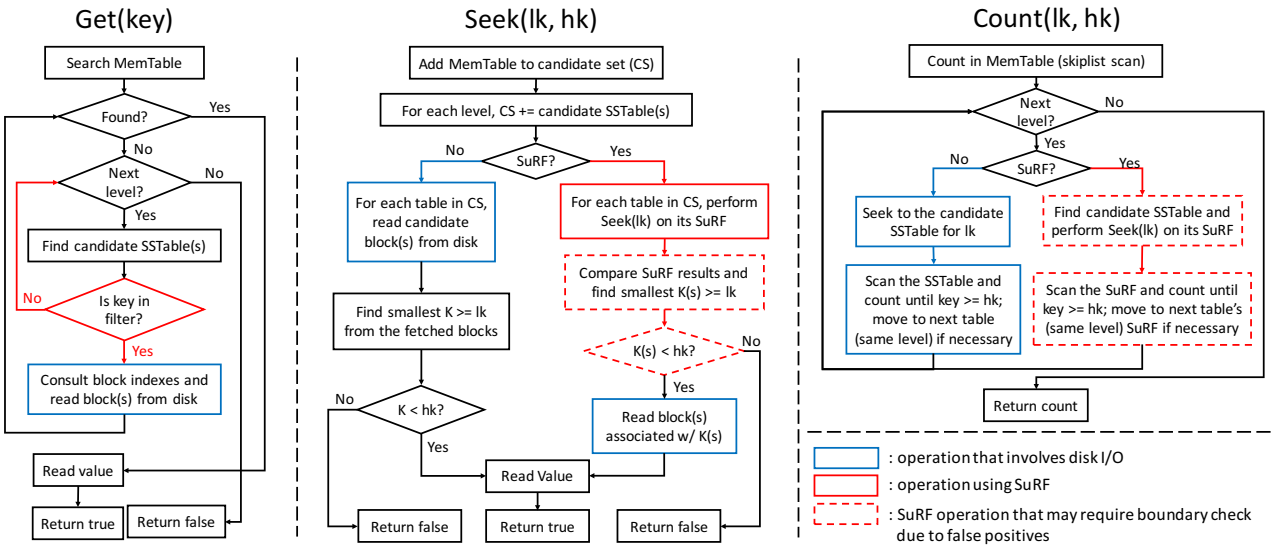


Figure 11: Execution paths for Get, Seek, and Count in RocksDB

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 probability. 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 must only increment the iterator pointing to the current key, and then repeat the "find the global smallest" algorithm as in *Seek*.

For *Count(lk, hk)*, RocksDB first performs a *Seek* on *lk* to initialize the iterators and then counts the number of items between *lk* and *hk* at each level. Without SuRF, the DBMS computes the count by scanning the blocks in SSTable(s) until the key exceeds the upper bound. If SuRFs are available, the counting is carried out by iterating in the filter(s). 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 [12], QuasarDB[8], LittleTable [48] and Cassandra-based systems [7, 36]. We thus create a synthetic RocksDB benchmark to model a time-series dataset generated from distributed sensors and use this for end-to-end performance measurements. We simulated 2K sensors to record events. The key for each event is a 128-bit value comprised of a 64-bit timestamp followed by a 64-bit sensor ID. The associated value in the record is 1 KB long. The occurrence of each event detected by each sensor follows a Poisson distribution with an expected frequency of one every 0.2 seconds. Each sensor operates for 10K seconds and records ~50K events. The starting timestamp for each sensor is randomly generated within the first 0.2 seconds. The total size of the raw records is approximately 100 GB.

Our testing framework supports the following database queries:

- **Point Query**: Given a timestamp and a sensor ID, return the record if there is an event.
- **Open-Seek Query**: Given a starting timestamp, return an iterator pointing to the earliest event after that time.
- **Closed-Seek Query**: Given a time range, determine whether any events happened during that time period. If yes, return an iterator pointing to the earliest event in the range.

Our test machine has an Intel Core i7-6770HQ CPU, 32 GB RAM, and an Intel 540s 480 GB SSD. We use Snappy (RocksDB's default) for data compression. The resulting RocksDB instance has four levels (including Level 0) and uses 52 GB of disk space. We configured[3] RocksDB according Facebook's recommendations [25, 26].

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 ~ 1000 times and the block indexes and filters are cached. After the warm-up, both RocksDB's block cache and the OS page cache are full. We then execute 50K application queries, recording the end-to-end throughput and I/O counts. We compute the DBMS's throughput by dividing query counts by execution time, while I/O counts are read from system statistics before and after the execution. The query keys (for range queries, the starting keys) are randomly generated. The reported numbers are the average of three runs. Even though RocksDB supports prefix Bloom filters, we exclude them in our evaluation because they do not offer benefits over Bloom filters in this scenario: (1) range queries using arbitrary integers do not have pre-determined key prefixes, which makes it

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

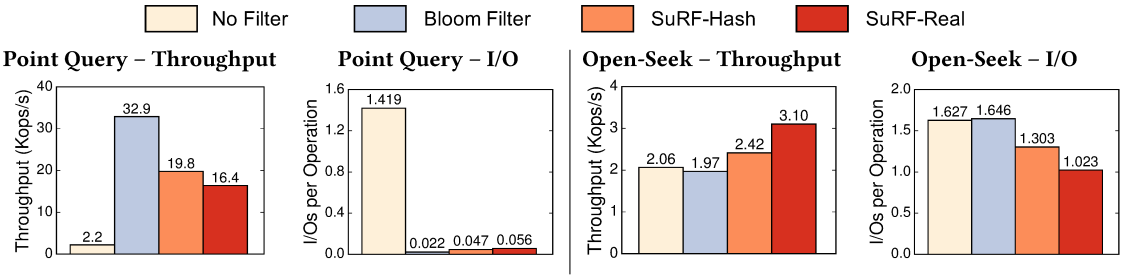


Figure 12: RocksDB point query and Open-Seek query evaluation under different filter configurations

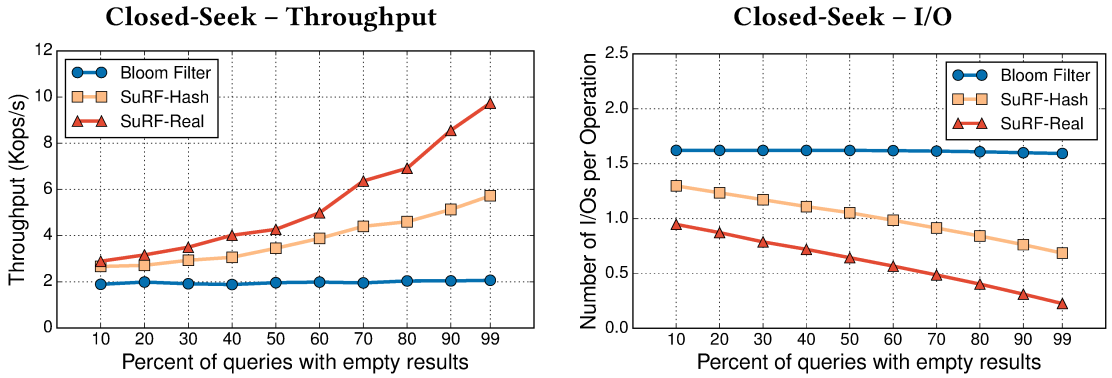


Figure 13: RocksDB Closed-Seek query evaluation under different filter configurations and range sizes

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 12 (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 (Level 1, 2, 3). Without filters, point queries incur approximately 1.5 I/Os per operation according to Figure 12, 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 [24].

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 provides similar benefit to SuRF-Hash because the key distribution is sparse.

The main benefit of using SuRF is speeding range queries. Figure 12 (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 13 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 containing that minimum key 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 11). We believe it an interesting element of future work to integrate approximate counts (which are exact for static datasets) into RocksDB or another system more explicitly designed for approximate queries.

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

7 RELATED WORK

Alternatives to the FST for range filtering. The Bloom filter [19] and its major variants [20, 27, 45] 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, 51, 56]. The downside for Bloom filters, however, is that they cannot handle range queries because their hashing does not preserve key order. In practice, people use prefix Bloom filters to help answer range-emptiness queries. For example, RocksDB [6], LevelDB [3], and LittleTable [48] 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) [14] was introduced as part of Project Siberia in Hekaton [23] to guard cold data. An ARF is a simple binary tree that covers the entire key space (e.g., for 64-bit integer keys, the root node represents range [0, 2^{64}-1] and its children represent [0, 2^{63}-1] and [2^{63}, 2^{64}-1]). Each leaf node indicates whether there may be any key or absolutely no key in its range. Using an ARF involves three steps: building a perfect trie, training with sample queries to determine which nodes to include in an ARF, and then encoding the trained ARF into a bit sequence in breadth-first order that is static. ARF differs from SuRF in that it targets different applications and scalability goals. First, ARF behaves more like a cache than a general-purpose filter. Training an ARF requires knowledge about prior queries. An ARF instance performs well on the particular query pattern for which it was trained. If the query pattern changes, ARF requires a rebuild (i.e., decode, re-train, and encode) to remain effective. ARF works well in the setting of Project Siberia, but its workload assumptions limit its effectiveness as general range filter. SuRF, on the other hand, assumes nothing about workloads. It can be used as a Bloom filter replacement but with range filtering ability, 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 in 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 B.

LSM-trees. Many modern key-value stores adopt the log-structured merge tree (LSM-tree) design [44] for its high write throughput and low space amplification. Such systems include LevelDB [3], RocksDB [6], Cassandra [36, 52], HBase [4], WiredTiger [54] and cLSM [29] from Yahoo Labs. Monkey [22] 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 [25]. These optimizations fall under the RUM Conjecture [16]: 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 [55] 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. Succinct tree representations typically use LOUDS, as FST does, or "balanced parentheses" (BP) sequences [40]. The BP representations support more tree operations in constant time [28, 38, 41, 42, 49], but are slower for the simple "parent-child" navigations that are needed for FST [15].

Many state-of-the-art succinct tries [17, 32, 47] 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 [32] 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 [13] and follow-up work BlowFish [35] are among the few attempts in systems research to use succinct data structures extensively in a general distributed data store. They store data sets using compressed suffix arrays [33] and achieve significant space savings. Compared to other non-compressed systems, Succinct and BlowFish achieve better query performance mainly through keeping more data resident in DRAM. FST can provide similar benefits when used in larger-than-DRAM workloads. In addition, FST does not slow down the system even when the entire data set fits in DRAM.

8 CONCLUSION

This paper 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.

REFERENCES

[1] 2010. tx-trie 0.18 – Succinct Trie Implementation. https://github.com/hillbig/tx-trie. (2010).
[2] 2013. Squid Web Proxy Cache. http://www.squid-cache.org/. (2013).
[3] 2014. Google LevelDB. https://github.com/google/leveldb. (2014).
[4] 2015. Apache HBase. https://hbase.apache.org/. (2015).
[5] 2015. Facebook MyRocks. http://myrocks.io/. (2015).
[6] 2015. Facebook RocksDB. http://rocksdb.org/. (2015).
[7] 2015. KairosDB. https://kairosdb.github.io/. (2015).
[8] 2015. QuasarDB. https://en.wikipedia.org/wiki/Quasardb. (2015).
[9] 2016. ARF Implementation. https://github.com/carolinux/adaptive_range_filters. (2016).
[10] 2016. Succinct Data Structures. https://en.wikipedia.org/wiki/Succinct_data_structure. (2016).
[11] 2017. InfluxData InfluxDB. https://www.influxdata.com/time-series-platform/influxdb/. (2017).
[12] 2017. The InfluxDB Storage Engine and the Time-Structured Merge Tree (TSM). https://docs.influxdata.com/influxdb/v1.0/concepts/storage_engine/. (2017).
[13] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *NSDI '15*. 337–350.
[14] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1714–1725.
[15] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. 2010. Succinct trees in practice. In *Proc. of ALENEX '10*. 84–97.
[16] 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 *EDBT*, Vol. 2016. 461–466.
[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. http://idlebox.net/2007/stx-btree/. (2008).
[19] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *CACM* 13, 7 (1970), 422–426.
[20] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *14th Annual European Symposium on Algorithms, LNCS 4168*. 684–695.
[21] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*.
[22] 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.
[23] 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*.
[24] Siying Dong. 2017. personal communication. (2017). 2017-08-28.
[25] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR*.
[26] Facebook. 2015. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. (2015).
[27] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 1998. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. ACM SIGCOMM*.
[28] Richard F Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. 2006. A simple optimal representation for balanced parentheses. *Theoretical Computer Science* 368, 3 (2006).
[29] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 32.
[30] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Proc. of WEA*. 27–38.
[31] Roberto Grossi and Giuseppe Ottaviano. 2013. Design of practical succinct data structures for large data collections. In *Proc. of SEA '13*.
[32] Roberto Grossi and Giuseppe Ottaviano. 2015. Fast compressed tries through path decompositions. *Journal of Experimental Algorithmics (JEA)* 19 (2015), 3–4.
[33] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SICOMP* 35, 2 (2005), 378–407.
[34] Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Foundations of Computer Science*. IEEE, 549–554.
[35] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2016. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proc of NSDI' 16*.
[36] A. Lakshman and P. Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating System Review* 44 (April 2010), 35–40.
[37] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *Proceedings of ICDE*.
[38] Hsueh-I Lu and Chia-Chi Yeh. 2008. Balanced parentheses strike back. *TALG* 4, 3 (2008), 28.
[39] Miguel Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. 2016. Practical compressed string dictionaries. *Information Systems* 56 (2016), 73–108.
[40] J Ian Munro and Venkatesh Raman. 2001. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.* 31, 3 (2001), 762–776.
[41] J Ian Munro, Venkatesh Raman, and S Srinivasa Rao. 2001. Space efficient suffix trees. *Journal of Algorithms* 39, 2 (2001), 205–222.
[42] J Ian Munro and S Srinivasa Rao. 2004. Succinct representations of functions. In *ICALP*.
[43] Gonzalo Navarro and Eliana Providel. 2012. Fast, small, simple rank/select on bitmaps. In *Proc. of SEA '12*. 295–306.
[44] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree. *Acta Inf.* 33, 4 (1996), 351–385.
[45] Felix Putze, Peter Sanders, and Singler Johannes. 2007. Cache-, Hash- and Space-Efficient Bloom Filters. In *Experimental Algorithms*. 108–121.
[46] Naila Rahman, Rajeev Raman, et al. 2006. Engineering the LOUDS succinct tree representation. In *Proc. of WEA '06*. 134–145.
[47] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *TALG* 3, 4 (2007), 43.
[48] 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.
[49] Kunihiko Sadakane and Gonzalo Navarro. 2010. Fully-functional succinct trees. In *SODA*.
[50] 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*.
[51] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *SIGCOMM*. 181–192.
[52] The Apache Software Foundation. 2015. Apache Cassandra. https://cassandra.apache.org/. (2015).
[53] Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *Proceedings of the 7th international conference on Experimental algorithms (WEA'08)*. 154–168.
[54] WiredTiger. 2014. WiredTiger. http://www.wiredtiger.com/. (2014).
[55] 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.
[56] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. CoNEXT*.
[57] 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 *SIGMOD*.
[58] Dong Zhou, David G. Andersen, and Michael Kaminsky. 2013. Space-Efficient, High-Performance Rank & Select Structures on Uncompressed Bit Sequences. In *Symposium on Experimental Algorithms*.

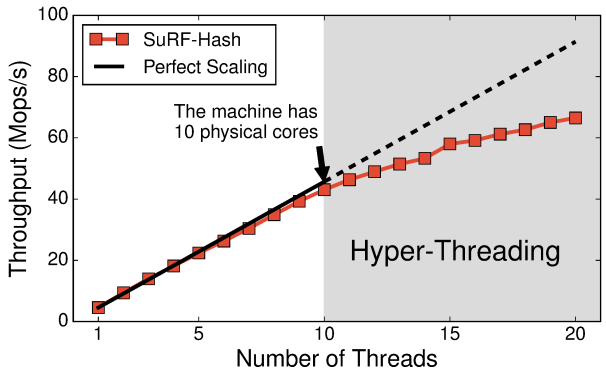


Figure 14: Point query performance of SuRF as the number of threads increases.

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

Table 1: Experimental comparison between ARF and SuRF.

APPENDIX

A EXTENDING SURF TO SUPPORT UPDATES

The current version of SuRF, as described in this paper, targets static use cases such as in a log-structured merge tree, as described in Section 5. For applications that require dynamic use of range filters, one can extend SuRF to support updates. To support *insert*, SuRF can leverage the hybrid index technique [57] to include a small dynamic trie in front of it to absorb all the writes and perform batch merges periodically so that the cost of individual inserts to SuRF is amortized. To support *delete*, SuRF can add a "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*. Periodical garbage collection is needed to keep SuRF small. Supporting updates in SuRF is beyond the scope of this paper, and we defer the problem to future work.

B COMPARING ARF AND SURF

This section extends our discussion on the Adaptive Range Filter (ARF) in Section 7 by providing an experimental evaluation. 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 YCSB workload C setting from 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 bit). 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 7 MB. In contrast, building SuRF only uses 0.02 GB of memory and finishes in 1.2 seconds.

C MULTI-THREADED SURF EXPERIMENTS

This section verifies that SuRFs are scalable on multi-core systems. We use the same YCSB-C-based workloads as in Section 4.2. The dataset includes 100M 64-bit integer keys. We construct SuRF using half of the keys (i.e., 50M) picked at random. We then execute 100M point queries generated by YCSB on the entire dataset, varying the number of threads. The queries are partitioned evenly across threads; for example, if there are 10 concurrent threads, each thread will execute 10M queries. The experiment runs on Intel Xeon E5-2680 v2 @ 2.80 GHz with 256 KB L2-cache, 25 MB L3-cache and 4×32 GB RAM. It has 10 physical cores and 20 hardware threads (with hyper-threading enabled).

Figure 14 shows the aggregate throughput as the number of threads increases. SuRF scales perfectly when using no hyper-threading (only a bit off due to cache contention). SuRF's throughput keeps increasing without any performance collapse, even with hyper-threading. This result is expected because SuRF is a read-only data structure and is completely lock-free, experiencing little contention with many concurrent threads. We omit a scalability graph for range queries that shows similar results.

D ACKNOWLEDGEMENTS

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.

This paper is dedicated to the memory of Leon Wrinkles.
May his tortured soul rest in peace.