# Succinct Range Filters

Huanchen Zhang[฿], Hyeontaek Lim[฿], Viktor Leis[€], David G. Andersen[฿],
Michael Kaminsky[$], Kimberly Keeton[£], Andrew Pavlo[฿]
[฿]Carnegie Mellon University, [€]Friedrich Schiller University Jena, [$]Intel Labs, [£]Hewlett Packard Labs
huanche1, hl, dga, pavlo@cs.cmu.edu, viktor.leis@uni-jena.de, michael.e.kaminsky@intel.com,
kimberly.keeton@hpe.com

## 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. 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.

## 1. INTRODUCTION

Write-optimized log-structured merge (LSM) trees [30] are popular low-level storage engines for general-purpose databases that provide fast writes [1, 34] and ingest-abundant DBMSs such as time-series databases [4, 32]. One of their main challenges for fast query processing is that items could reside in immutable files (SSTables) from all levels [3, 25]. Item retrieval in an LSM tree-based design may therefore incur multiple expensive disk I/Os [30, 34]. This challenge calls for in-memory data structures that can help locate query items. *Bloom filters* are a good match for this task [32, 34] 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@") [2, 20, 32], despite their inflexibility for more general range queries. The designers of RocksDB [2] have expressed a desire to have a more flexible data structure for this purpose [19]. 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 and range filtering. 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 [14], ART [26]) 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 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.

©ACM 2019 This is a minor revision of the paper entitled "SuRF: Practical Range Query Filtering with Fast Succinct Tries", published in SIGMOD'18, ISBN 978-1-4503-4703-7/18/06, June 10–15, 2018, Houston, TX, USA. DOI: https://doi.org/10.1145/3183713.3196931

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Copyright 2008 ACM 0001-0782/08/0X00 ...$5.00.

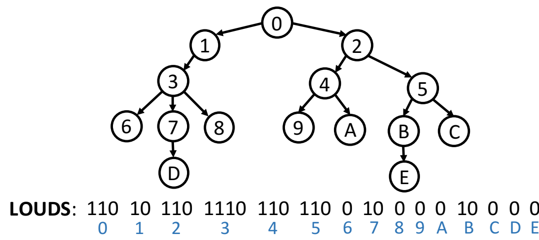



Figure 1: An example ordinal tree encoded using LOUDS

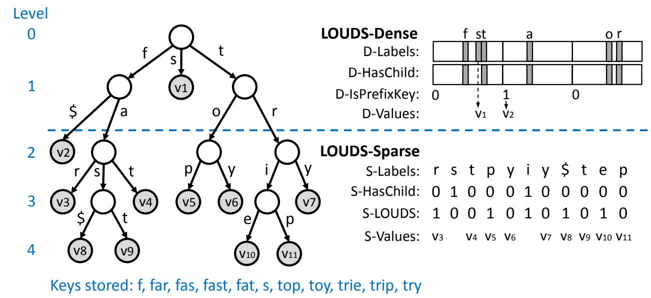



Figure 2: **LOUDS-DS Encoded Trie** – "$" represents the character whose ASCII number is `0xFF`. It is used to indicate the situation where a prefix string leading to a node is also a valid key.

## 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 [12, 13, 23, 24, 27, 28, 33], 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.

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 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 $\{0, 1, \ldots, k-1\}$. The information-theoretic lower bound of a trie of degree $k$ is approximately $n(k \log_2 k - (k-1)\log_2(k-1))$ bits [13].

Jacobson [24] 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 (i.e., a rooted tree where each node can have an arbitrary number of children in order). LOUDS traverses the nodes in a breadth-first order and encodes each node's degree using the unary code. For example, node 3 in Fig. 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 [22, 29, 36, 40] 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.

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

There are three ways to define "close" [9]. 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.

The first bitmap (*D-Labels*) records the branching labels for each node. Specifically, the $i$-th bit in the bitmap, where $0 \leq i \leq 255$, indicates whether the node has a branch with label $i$. For example, the root node in Fig. 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 Fig. 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 Fig. 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 ×$rank_1$(*D-HasChild*, *pos*) computes the bit position of the first child node. To move up the trie, **D-ParentNodePos**(*pos*) = $select_1$(*D-HasChild*, ⌊*pos*/256⌋) 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*, ⌊*pos/256*⌋)-1 gives the lookup position.

### 2.3 LOUDS-Sparse

As shown in the lower half of Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 2, the first non-value node at level 2 has three branches and is encoded as `100` in *S-LOUDS*.

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*).

## 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 use a size ratio $1 : R$ between LOUDS-Dense and LOUDS-Sparse to determine the dividing point among levels. Reducing $R$ leads to more LOUDS-Dense 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.

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, 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.

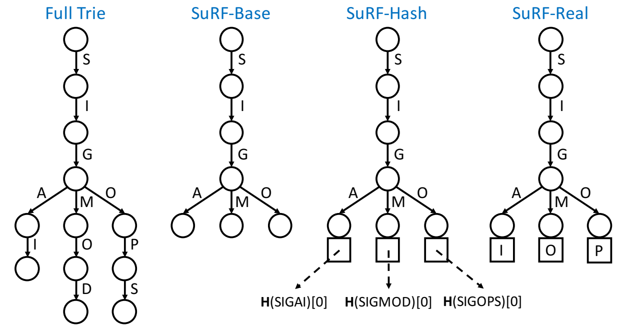



Figure 3: An example of deriving SuRF variations from a full trie.

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

## 2.5 Space 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 $Z + o(Z)$.

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 take more space than LOUDS-Sparse: if a node's fanout is larger than 51, it takes fewer bits to encode 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.

# 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.

## 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 guard 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. Fig. 3 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 $F > 2$: there are less than 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 Fig. 3, querying key `SIGMETRICS' will cause a false positive in SuRF-Base. FPR in SuRF-Base depends on the distributions of the stored and query keys. 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 two 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 Fig. 3, 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. Fig. 3 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 Operations

We summarize how to implement SuRF's basic operations 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 and 3.3.

*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 includes 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++).

# 4. FST & SuRF MICROBENCHMARKS

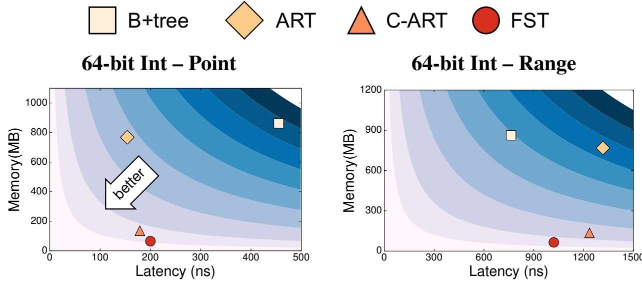
In this section, we first evaluate SuRF and its underlying FST data structure using in-memory microbenchmarks to provide a comprehensive understanding of the filter's strengths and weaknesses. We use the Yahoo! Cloud Serving Benchmark (YCSB) [17] workloads C and E to generate point and range queries. 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 omit error bars because the variance is small.

## 4.1 FST Evaluation

We compare FST to three state-of-the-art pointer-based indexes:

- **B+tree**: This is the most common index structure used in database systems. We use the fast STX B+tree [14] with node size 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 [26]. 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 [38].

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



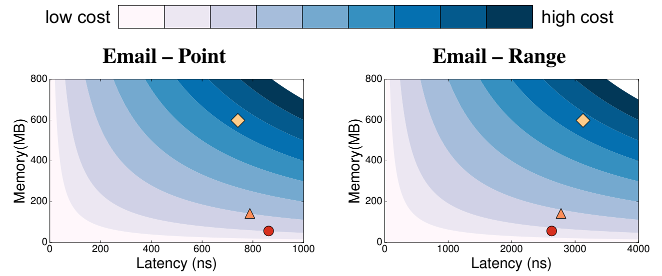



Figure 4: FST vs. Pointer-based Indexes

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.

ART, C-ART, and FST store only unique key prefixes in this experiment as described in Section 3.1. Fig. 4 shows the comparison results. Each subfigure plots the locations of the four (three for email keys) indexes in the performance-space (latency vs. memory) map. We observe that FST is among the fastest choices in all cases while consuming less space. To better understand this trade-off, we define a cost function $C = P^r S$, where $P$ represents performance (latency), and $S$ represents space (memory). The exponent $r$ indicates the relative importance between $P$ and $S$. $r > 1$ means that the application is performance critical, and $0 < r < 1$ suggests otherwise. We define an "indifference curve" as a set of points in the performance-space map that have the same cost. We draw the equi-cost curves in Fig. 4 using cost function $C = PS$ ($r = 1$), assuming a balanced performance-space trade-off. We observe that FST has the lowest cost (i.e., is 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.

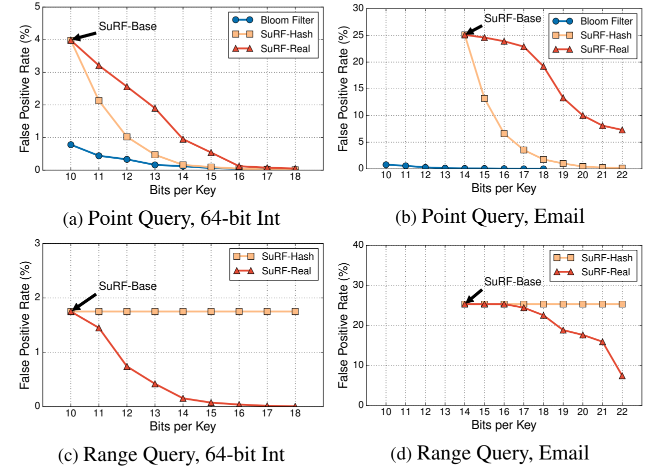
We also compared FST against other succinct trie alternatives (i.e., tx-trie [10] and the path-decomposed trie (PDT) [23]). Our results showed that FST is 6–15× faster than tx-trie, 4–8× faster than PDT, and is also smaller than both. Detailed evaluation is included in the original SIGMOD paper [39].

## 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 or range 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.

### 4.2.1 False Positive Rate

Fig. 5 shows the false positive rate (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




(a) Point Query, 64-bit Int
(b) Point Query, Email
(c) Range Query, 64-bit Int
(d) Range Query, Email

Figure 5: False positive rate comparison between SuRF variants and the Bloom filter (lower is better)

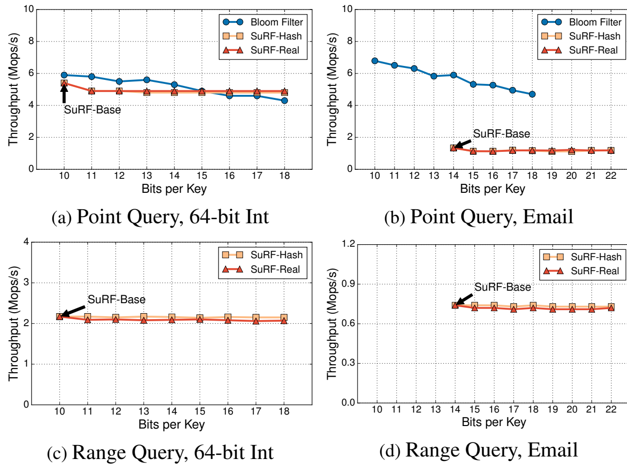
workloads. This is because email keys share longer prefixes, which increases the number of internal nodes in SuRF.

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.

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

Fig. 6 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 such as vectorized label search and memory prefetching. 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 fil-




(a) Point Query, 64-bit Int (b) Point Query, Email

(c) Range Query, 64-bit Int (d) Range Query, Email

Figure 6: Performance comparison between SuRF variants and the Bloom filter (higher is better)

ters 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 walk down to the bottom of the trie (no early exit).

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.

# 5. EXAMPLE APPLICATION: ROCKSDB

We integrated SuRF with RocksDB as a replacement for its Bloom filter. Incoming writes go into the RocksDB's MemTable and are appended to a log file for persistence. When the MemTable is full (e.g., exceeds 4 MB), the engine sorts it and then converts it to an SSTable that becomes part of level 0. An SSTable contains sorted key-value pairs and is divided into fixed-length blocks matching the smallest disk access units. To locate blocks, RocksDB stores the "restarting point" (a string that is $\geq$ the last key in the current block and $<$ the first key in the next block) for each block as 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 $\geq 1$. This property ensures that an entry lookup reads at most one SSTable per level for levels $\geq 1$.

We modified RocksDB's point (*Get*) and range (*Seek*, *Next*) query implementations to use SuRF. 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 $\geq$ *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)* 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.

To illustrate how SuRFs benefit range queries, suppose a RocksDB instance has three levels ($L_N$, $L_{N-1}$, $L_{N-2}$) of SSTables on disk. $L_N$ has an SSTable block containing keys 2000, 2011, 2020 with 2000 as the block index; $L_{N-1}$ has an SSTable block containing keys 2012, 2014, 2029 with 2012 as the block index; and $L_{N-2}$ has an SSTable block containing keys 2008, 2021, 2023 with 2008 as the block index. Consider the range query [2013, 2019]. Using only block indexes, RocksDB has to read all three blocks from disk to verify whether there are keys between 2013 and 2019. Using SuRFs eliminates the blocks in $L_N$ and $L_{N-2}$ because the filters for those SSTables will return false to query [2013, 2019] with high 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*.

# 6. SYSTEM EVALUATION

Time-series databases often use RocksDB or similar LSM-tree designs for the storage engine. Examples are InfluxDB [4], QuasarDB[6], LittleTable [32] and Cassandra-based systems [5, 25]. 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

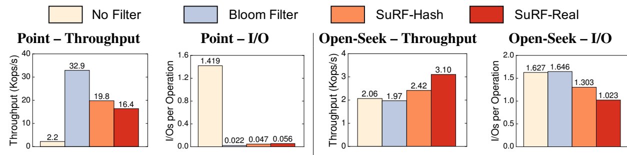



Figure 7: RocksDB point query and Open-Seek query evaluation under different filter configurations

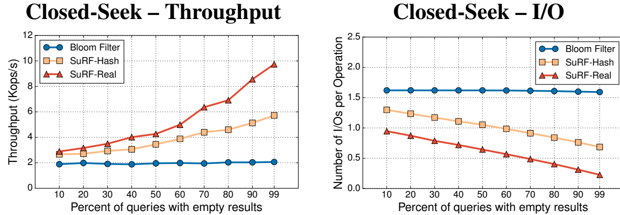



Figure 8: RocksDB Closed-Seek query evaluation under different filter configurations and range sizes

0.2 seconds. The total size of the raw records is approximately 100 GB.

Our testing framework supports the following 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 RocksDB according Facebook's recommendations [7, 20].

We create four instances of RocksDB with different filter options: no filter, Bloom filter, SuRF-Hash, and 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 $\sim$ 1000 times and the block indexes and filters are cached. After the warm-up, both RocksDB's block cache and the OS page cache are full. We then execute 50K application queries, recording the end-to-end throughput and I/O counts. We compute the DBMS's throughput by dividing query counts by execution time, while I/O counts are read from system statistics before and after the execution. The query keys (for range queries, the starting keys) are randomly generated. The reported numbers are the average of three runs. Even though RocksDB supports prefix Bloom filters, we exclude them in our evaluation because they do not offer benefits over Bloom filters in this scenario: (1) range queries using arbitrary integers do not have pre-determined key prefixes, which makes it hard to generate such prefixes, and (2) even if key prefixes could be determined, prefix Bloom filters always return false positives for point lookups on absent keys sharing the same prefix with any present key, incurring high false positive rates.

Fig. 7 (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 Fig. 7, 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 [19].

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. Fig. 7 (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.

Fig. 8 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$\times$ 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.

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

The Bloom filter [15] and its major variants [16, 21, 31] 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 [8, 35, 37]. 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,

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`.

RocksDB [2], LevelDB [3], and LittleTable [32] 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) [11] was introduced as part of Project Siberia in Hekaton [18] to guard cold data. An ARF is a simple encoded binary tree that covers the entire key space. 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. It requires knowledge about prior queries for training. SuRF, on the other hand, assumes nothing about workloads. 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's 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. SuRF avoids linear scans by navigating its internal tree structure with rank & select operations.

# 8. CONCLUSION

This paper introduces the SuRF filter structure, which supports approximate membership tests for single keys and ranges. 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. 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] Facebook MyRocks. http://myrocks.io/.

[2] Facebook RocksDB. http://rocksdb.org/.

[3] Google LevelDB. https://github.com/google/leveldb.

[4] The influxdb storage engine and the time-structured merge tree (tsm). https://docs.influxdata.com/influxdb/v1.0/concepts/storage_engine/.

[5] Kairosdb. https://kairosdb.github.io/.

[6] Quasardb. https://en.wikipedia.org/wiki/Quasardb.

[7] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[8] Squid Web Proxy Cache. http://www.squid-cache.org/.

[9] Succinct data structures. https://en.wikipedia.org/wiki/Succinct_data_structure.

[10] tx-trie 0.18 – succinct trie implementation. https://github.com/hillbig/tx-trie, 2010.

[11] K. Alexiou, D. Kossmann, and P.-Å. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.

[12] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proceedings of ALENEX '10*, pages 84–97, 2010.

[13] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[14] T. Bingmann. Stx b+ tree c++ template classes. http://idlebox.net/2007/stx-btree/, 2008.

[15] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[16] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of SOCC'10*, pages 143–154. ACM, 2010.

[18] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of SIGMOD'13*, pages 1243–1254. ACM, 2013.

[19] S. Dong. personal communication, 2017. 2017-08-28.

[20] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[21] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[22] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proceedings of WEA'05*, pages 27–38, 2005.

[23] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. *Journal of Experimental Algorithmics (JEA)*, 19:3–4, 2015.

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

[25] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[26] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE'13*, pages 38–49. IEEE, 2013.

[27] M. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016.

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

[29] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proceedings of SEA '12*, pages 295–306, 2012.

[30] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[31] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *Proceedings of WEA'07*, pages 108–121. Springer, 2007.

[32] S. Rhea, E. Wang, E. Wong, E. Atkins, and N. Storer. Littletable: a time-series database and its uses. In *Proceedings of SIGMOD'17*, pages 125–138. ACM, 2017.

[33] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA'10*, 2010.

[34] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of SIGMOD'12*, pages 217–228. ACM, 2012.

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

[36] S. Vigna. Broadword implementation of rank/select queries. In *Proceedings of WEA'08*, pages 154–168, 2008.

[37] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: Bloom filter forwarding architecture for large organizations. In *Proceedings of CoNEXT'09*, pages 313–324. ACM, 2009.

[38] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of SIGMOD'16*, pages 1567–1581. ACM, 2016.

[39] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: practical range query filtering with fast succinct tries. In *Proceedings of SIGMOD'18*, pages 323–336. ACM, 2018.

[40] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Proceedings of SEA '13*, pages 151–163. Springer, 2013.