

Succinct Range Filters

By Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo

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, such as range counts. SuRF is based on a new data structure called the *Fast Succinct Trie* (FST) that matches the performance of state-of-the-art order-preserving indexes, while consuming only 10 bits per trie node—a space close to the minimum required by information theory. Our experiments show that SuRF speeds up range queries in a widely used database storage engine by up to 5×.

1. INTRODUCTION

Write-optimized log-structured merge (LSM) trees¹⁶ are popular low-level storage engines for general-purpose databases that provide fast writes^{1, 14, 18} and ingest-abundant DBMSs such as time-series databases.^{5, 17} One of their main challenges for fast query processing is that items could reside in different immutable files (SSTables) from all levels. Item retrieval in these systems may therefore incur multiple expensive disk I/Os.^{16, 18}

Many LSM tree-based systems use Bloom filters to “guard” on-disk files to reduce the number of unnecessary I/Os^{2, 3, 17, 18}; they read an on-disk file only when the associated in-memory Bloom filter indicates that the query item may exist in the file. Bloom filters are a good match for this task. First, Bloom filters are fast and small enough to reside in memory. Second, Bloom filters answer approximate membership tests with “one-sided” errors—if the querying item is a member, the filter is guaranteed to return true; otherwise, the filter will likely return false, but may incur a false positive.

Although Bloom filters are useful for single-key lookups (“Is key 50 in the SSTable?”), they cannot handle range queries (“Are there keys between 40 and 60 in the SSTable?”). With only Bloom filters, an LSM tree-based storage engine still needs to read additional disk blocks for range queries. Alternatively, one could maintain an auxiliary index, such as a B+Tree, to accelerate range queries, but the memory cost would be significant. To partly address the high I/O cost of range queries, LSM tree-based designs often use *prefix Bloom filters* to optimize certain fixed-prefix queries (e.g., “where email starts with com.foo@”),^{2, 11, 17} despite their inflexibility for more general range queries.

To address these limitations, we present the **Succinct Range Filter** (SuRF), a fast and compact data structure 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 data structure called the *Fast Succinct Trie* (FST). It performs comparably to or better than state-of-the-art uncompressed index structures 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 microbenchmarks and as a Bloom filter replacement in RocksDB—a widely-used database storage engine.² Our experiments on a 100GB time-series dataset show that replacing the Bloom filters with SuRFs of the same filter size reduces I/O. This speeds up closed-range queries (i.e., with an upper bound) by up to 5× compared to the original implementation, with a modest cost on the worst-case point query throughput due to slightly higher false positive rate. One can eliminate this performance gap by increasing the size of SuRFs by a few bits per key.

2. FAST SUCCINCT TRIES

The core data structure in SuRF is the *Fast Succinct Trie* (FST). FST is a space-efficient, static trie that answers point and range queries. FST is 4–15× faster than earlier succinct tries,^{6, 12} achieving performance comparable to or better than the state-of-the-art pointer-based indexes.^{8, 15, 19}

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 lower levels of the trie using the succinct **LOUDS-Sparse** scheme to guarantee the overall space-efficiency of the data structure. Here, LOUDS stands for the *Level-Ordered Unary Degree Sequence*.¹³ By contrast, we encode the upper levels using a fast bitmap-based encoding scheme, called **LOUDS-Dense**, in which a child node search requires only one array lookup, choosing performance over space.

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).

The original version of this paper is entitled “SuRF: Practical Range Query Filtering with Fast Succinct Tries” and was published in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data* (Houston, TX, USA).

2.1 LOUDS-Sparse

LOUDS-Sparse encodes the trie nodes in the level order using four byte-/bit-sequences, as shown in the lower half of Figure 1.

The first byte-sequence, *S-Labels*, records all the branching labels for each trie node. As an example, the second node at level 2 in Figure 1 has three branches. *S-Labels* includes its 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 $0xFF^1$ at the beginning of the node. For example, in Figure 1, the second node at level 3 has ‘fas’ as its incoming prefix. As ‘fas’ itself is also a key stored in the trie, 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 subtree) or terminates (i.e., points to a value). Taking the rightmost node at level 2 in Figure 1 as an example, because the branch labeled *i* points to a subtree, the corresponding bit in *S-HasChild* is set. The branch labeled *y*, however, points to a value, and its *S-HasChild* bit is cleared.

The third bit-sequence (*S-LOUDS*) 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 1, the second node at level 2 has three branches and is encoded as 100 in *S-LOUDS*.

The final byte-sequence (*D-Values*) stores the fixed-length values (e.g., pointers) mapped by the keys. The values are concatenated in the level order—same as the three bitmaps.

Tree navigation relies on the fast rank & select primitives. Given a bit-vector, $rank(i)$ counts the number of 1s up to position *i*, while $select(i)$ returns the position of the *i*th 1. Modern rank & select implementations such as as^{21} achieve *constant time* by using lookup tables to store a sampling of precomputed results so that they only need to count between the samples. We denote $rank/select$ over bit-sequence *bs* on position *pos* to be $rank/select(bs, pos)$.

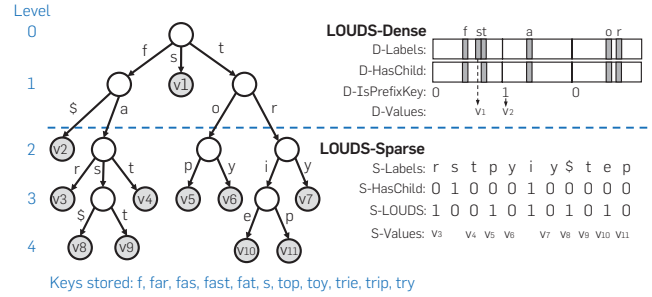
Let *pos* be the current bit position in *S-Labels*. Assume that $S-HasChild[pos] = 1$, indicating that the branch at *pos* points to a child node. To move to the child node, we first compute the child node’s rank in the overall level-ordered node list: $r = rank(S-HasChild, pos) + 1$. Because every node only has its first bit set in *S-LOUDS*, we can use $select(S-LOUDS, r)$ to find the position of that child node.

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

Given $S-HasChild[pos] = 0$, to access the associated value, we compute its index in *S-Values*. Because every cleared bit in *S-HasChild* has a value, there are $pos - rank(S-HasChild, pos)$ values before *pos*.

¹ If a node has a single branching label $0xFF$, it must be the real $0xFF$ byte (otherwise, the node will not exist in the trie).

Figure 1. An example fast succinct trie. The upper and lower levels of the trie are encoded using LOUDS-Dense and LOUDS-Sparse, respectively. “\$” represents the character whose ASCII number is $0xFF$. It is used to indicate the situation where the prefix leading to a node is also a valid key.



2.2 LOUDS-Dense

As shown in the top half of Figure 1, LOUDS-Dense encodes each trie node using three bitmaps of size 256 and a byte-sequence to hold the values. The encoding follows the level order.

The first bitmap (*D-Labels*) records the branching labels for each node. Specifically, the *i*th bit in the bitmap ($0 \leq i \leq 255$) indicates whether the node has a branch with label *i*. For example, the root node in Figure 1 has three outgoing branches labeled *f*, *s*, and *t*. The *D-Labels* bitmap thus sets the 102nd (*f*), 115th (*s*), and 116th (*t*) bits and clears the rest.

The second bitmap (*D-HasChild*) indicates whether a branch points to a subtree or terminates (i.e., points to the value or the branch does not exist). Taking the root node in Figure 1 as an example, the *f* and the *t* branches continue with subtrees, 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 to indicate whether the prefix that leads to the node is also a valid key. The same case is handled by the special byte $0xFF$ in LOUDS-Sparse.

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

Tree navigation in LOUDS-Dense also uses the rank & select primitives. Given a position *pos* in *D-Labels*, to move to the child node: $256 \times rank(D-HasChild, pos)$; to move to the parent: $select(D-HasChild, \lfloor pos/256 \rfloor)$; to access the value: $rank(D-Labels, pos) - rank(D-HasChild, pos) + rank(D-IsPrefixKey, \lfloor pos/256 \rfloor) - 1$.

LOUDS-Dense is faster than LOUDS-Sparse because (1) label search within a node requires only one lookup in the bitmap rather than a binary search and (2) move to child only computes a rank in one bit-vector instead of a rank and a select on different bit-vectors.

2.3 FST and operations

FST 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. By default, we

keep the size ratio R between LOUDS-Dense and LOUDS-Sparse to be less than 1:64 in favor of the space-efficiency provided by LOUDS-Sparse.

FST supports four 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 $k \geq \text{key}$.
- **MoveToNext(iter)**: Move the iterator to the next key.
- **Count(lowKey, highKey)**: Return the number of keys contained in the range $(\text{lowKey}, \text{highKey})$.

A point query (i.e., *ExactKeySearch*) in FST 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 label sequence for the target key byte. If the key byte does not exist, terminate and return NULL. Otherwise, check the corresponding bit in the *HasChild* bit-sequence. If the bit is set, 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.

LowerBound uses a high-level algorithm similar to the point query implementation. Instead of an exact match, the algorithm searches the current node's label sequence for the smallest label that is greater than or equal to the search byte of that level. The algorithm may recursively move up to the parent node if the search hits node boundaries. Once such label *L* is found, the algorithm moves iterator to the left-most key in the subtree rooted at *L*.

We include per-level cursors in the iterator to record a trace from root to leaf (i.e., the per-level positions in the label sequence) for the current key. Using the cursors, range scans (*MoveToNext*) in FST are implemented efficiently. Each level cursor is initialized once through a “move-to-child” call from its upper-level cursor. After that, scan operations at this level only involve cursor movement, which is cache-friendly and fast. Our evaluation shows that range queries in FST are even faster than pointer-based tries.

For *Count*, the algorithm first performs *MoveToNext* on both boundaries and obtains two iterators. It extends each iterator down the trie and sets the cursor at each level to the position of the smallest leaf key that is greater than the current key, until the two iterators meet or reach the maximum trie height. The algorithm then counts the number of leaf nodes at each level between the two iterators by computing the difference of their ranks on the *D-HasChild/S-HasChild* bit-vector. The sum of those counts is returned.

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

2.4 Space analysis

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 set. 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 (9.44 bits when $k = 256$ in our case).⁷

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). Although the space taken by LOUDS-Sparse is close to the information-theoretic lower 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 size ratio R to ensure that it does not affect the overall space efficiency of FST. 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. As 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

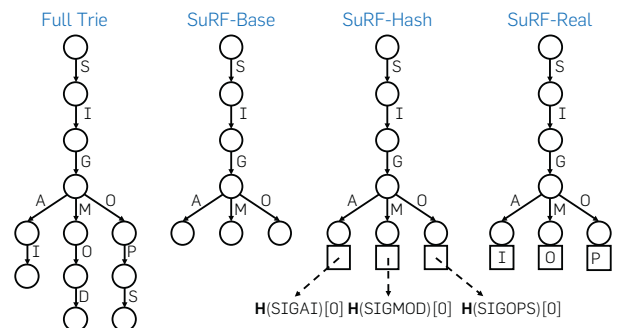
In building SuRF using FST, our goal was to balance a low false positive rate with the memory required by the filter. The key idea is to use a truncated trie, that is, to remove lower levels of the trie and replace them with suffix bits extracted from the key. We introduce three variations of SuRF. We 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. Figure 2

Figure 2. SuRF variations. Deriving SuRF variations from a full trie.



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. 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). A false positive in SuRF-Base occurs when the prefix of the nonexistent query key coincides with a stored key prefix. For example, in Figure 2, 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 here to allow SuRF to better distinguish between key prefixes.

3.2 SuRF with hashed key suffixes

As shown in Figure 2, 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). Experiments in Section 4.2 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 2 shows an example when $n = 8$. SuRF-Real includes the next character for each key ('I', 'O', 'P') to improve the distinguishability of the keys: for example, querying 'SIGMETRICS' no longer causes a false positive. Unlike in SuRF-Hash, both point and range queries benefit from the real suffix bits to reduce false positives. For point queries, the real suffix bits are used the same way as the hashed suffix bits. For range queries (e.g., move to the next key $> K$), when reaching a leaf node, SuRF-Real compares the stored suffix bits s to key bits k_s of the query key at the corresponding position.

If $k_s \leq s$, the iterator points to the current key; otherwise, it advances to the next key in the trie.

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

4. SURF MICROBENCHMARKS

In this section, we evaluate SuRF using in-memory micro-benchmarks to provide a comprehensive understanding of the filter's strengths and weaknesses.

The underlying data structure FST is evaluated separately in our original paper.²⁰ We found that compared to state-of-the-art pointer-based indexes such as B+tree⁸ and the Adaptive Radix Tree (ART),¹⁵ FST matches their performance while being an order-of-magnitude smaller. We also compared FST against other succinct trie alternatives^{6, 12} and showed that FST is 4–15× faster and is also smaller than these previous solutions.

4.1 Experiment setup

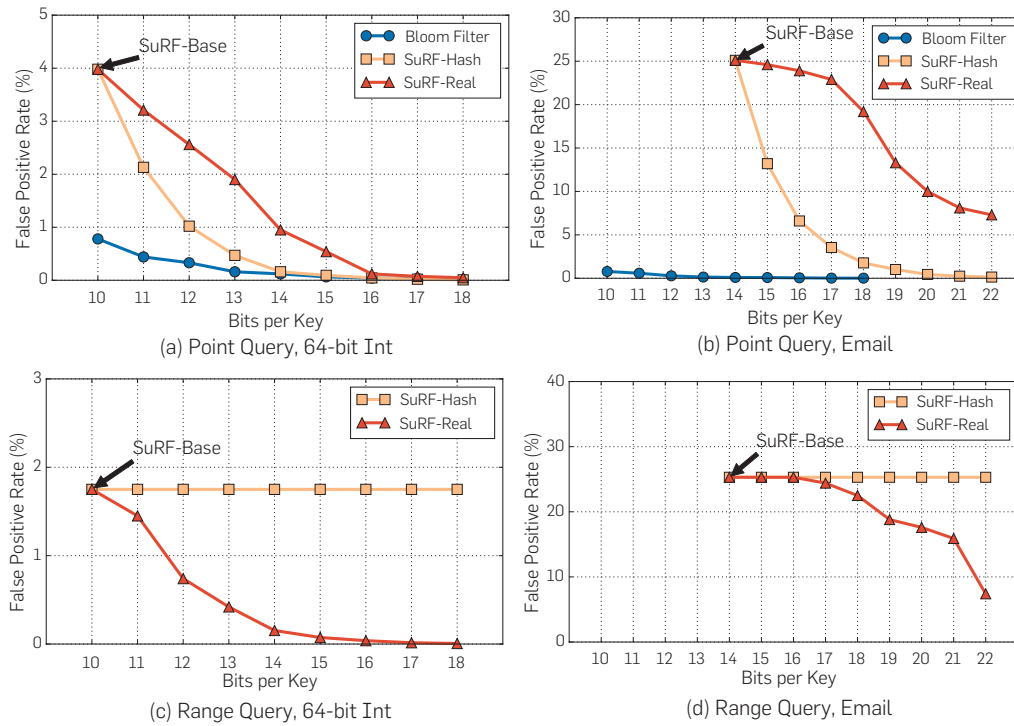
We use the YCSB⁹ workloads C and E to generate point and range queries. We test two representative key types: 64-bit random integers generated by YCSB and email addresses (host reversed, e.g., "com.domain@foo") drawn from a real-world dataset (average length = 22 bytes, max length = 129 bytes).

The 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. For 64-bit random integer keys, the range query is $[K + 2^{37}, K + 2^{38}]$ where 46% of the queries return true. For email keys, the range query is $[K, K \text{ (with last byte ++)}]$ (e.g., [org.acm@sigmod, org.acm@sigmoel]) where 52% of the queries return true.

4.2 False positive rate

Figure 3 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 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

Figure 3. SuRF false positive rate. False positive rate comparison between SuRF variants and the Bloom filter (lower is better).

information. The shape of the SuRF-Real curves in the email key workloads (i.e., the latter four suffix bits are more effective in recognizing false positives than the earlier four) 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 dataset are very similar (i.e., the key distribution is dense). Two email keys often differ by the last byte, or one may be a prefix of the other. If one of the keys is represented in the filter and the other key is not, querying the missing key on SuRF-Base is likely to produce false positives. The high FPR for SuRF-Base is significantly lowered by adding suffix bits, as shown in the figures.

4.3 Performance

Figure 4 shows the throughput comparison. The SuRF variants operate at a speed comparable to the Bloom filter for the 64-bit integer keyworkloads, thanks to the hybrid encodings 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 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. 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).

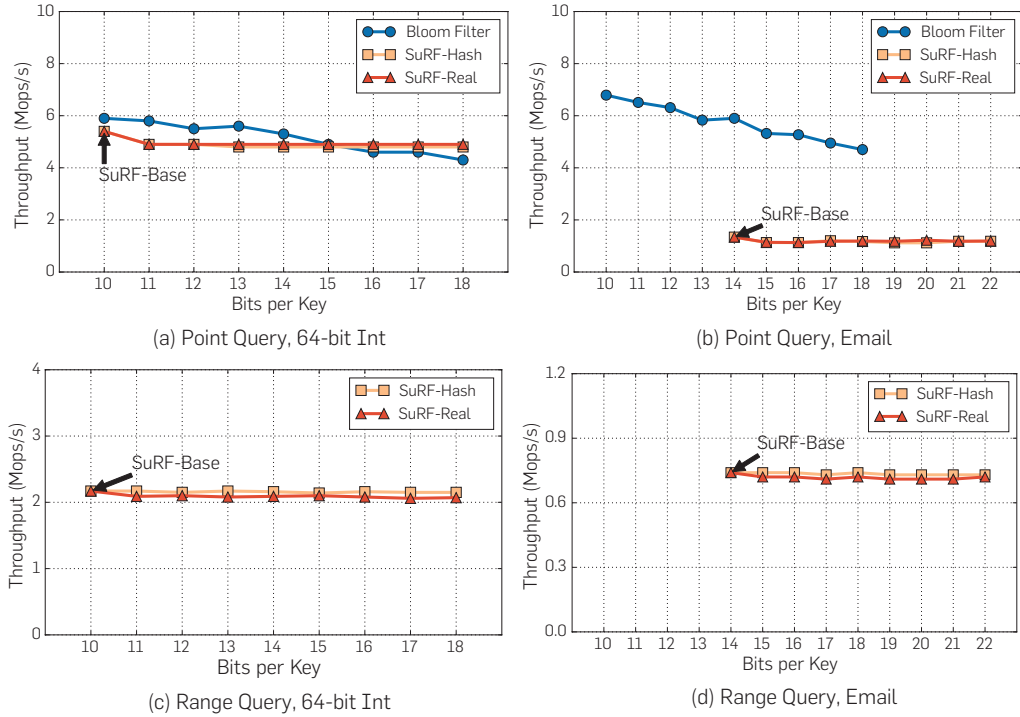
Some high-level takeaways from the experiments are as follows: (1) SuRF can perform range filtering while the Bloom filter cannot. (2) If the target application only needs point query filtering with moderate FPR requirements, the Bloom filter is usually a better choice than SuRF. (3) For point queries, 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. When the MemTable is full (e.g., exceeds 4MB), the engine sorts it and then converts it into an SSTable at 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 the block 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. The keys are globally sorted across SSTables for each level ≥ 1 . This property ensures that an entry lookup reads at most one SSTable per level for levels ≥ 1 .

We modified RocksDB's point (*Get*) and range (*Seek*) query implementations to use SuRF. For *Get(key)*, RocksDB uses SuRF exactly like the Bloom filter where at each level, it locates the candidate SSTable(s) and block(s) via the block indexes. For each candidate SSTable, RocksDB queries the

Figure 4. SuRF performance. Performance comparison between SuRF variants and the Bloom filter (higher is better).



in-memory filter first and fetches the SSTable block only if the filter result is positive.

To implement *Seek(lk, hk)*, RocksDB first collects the candidate SSTables at all levels by searching for *lk* 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 finds the global smallest key $K \geq lk$ among those candidate keys. If $K \leq hk$, the query succeeds; otherwise, the query returns empty.

With SuRFs, however, instead of fetching the actual blocks, RocksDB obtains the candidate key for each SSTable by performing a *LowerBound* query on its SuRF to avoid the one I/O per SSTable. If the query succeeds, RocksDB fetches exactly one block from the selected SSTable that contains the global minimum K . If the query returns empty, no I/O is involved. Because SuRF only stores key prefixes, the system must perform additional checks to break ties and to prevent false positives. The additional checks are described in our original paper.²⁰ Despite those potential checks, using SuRF in RocksDB reduces the average I/Os per *Seek(lk, hk)* query.

5.1 Evaluation setup

Time-series databases often use RocksDB or similar LSM-tree designs as their storage engine.^{5,17} We thus create a synthetic RocksDB benchmark to model a time-series dataset generated from distributed sensors for our 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 1KB long. The occurrence of each event detected by each sensor follows a Poisson distribution with an expected frequency of one every 0.2 s. Each sensor operates for 10K seconds and records $\sim 50K$ events. The starting timestamp for each sensor is randomly generated within the first 0.2 s. The total size of the raw records is approximately 100GB.

Our testing framework supports the following queries:

- **Point Query:** Given a timestamp and a sensor ID, return the record if there is an event.
- **Range 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 480GB SSD. We configured² RocksDB according to Facebook's recommendations.^{4,11} The resulting RocksDB instance has four levels and uses 52GB of disk space.

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 1 million uniformly-distributed point queries to existing keys so that every SSTable is touched roughly 1000 times, and the block indexes and

² Block cache size = 1 B; OS page cache $\leq 3GB$.

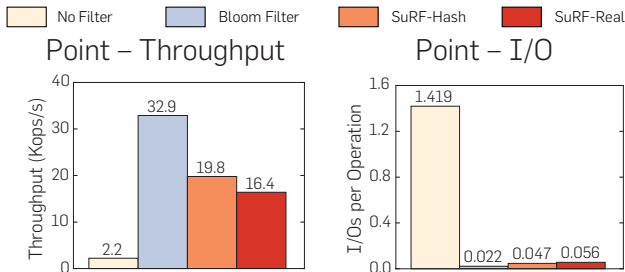
filters are cached. After the warm-up, both RocksDB's block cache and the OS page cache are full. We then execute 50k application queries, recording the DBMS's end-to-end throughput and I/O counts. The query keys (for range queries, the starting keys) are randomly generated: a random timestamp within the operated time range + a randomly picked sensor ID. The reported numbers are the average of three runs.

5.2 Point query results

Figure 5 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 5, which means that the entire Level 1 and approximately half of Level 2 are likely cached. This is representative of typical RocksDB configurations where the last two levels are not cached in memory.¹⁰

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

Figure 5. Point queries. RocksDB point query evaluation under different filter configurations.



5.3 Range query results

The main benefit of using SuRF is speeding up range queries. Figure 6 shows the throughput and I/O count for range queries. On the x-axis, we control the percentage 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 $\lambda \ln\left(\frac{1}{P}\right)$. For example, for 50%, the range size is 69310 ns.

As shown in Figure 6, 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 because the real suffix bits help reduce false positives at the range boundaries.

6. CONCLUSION


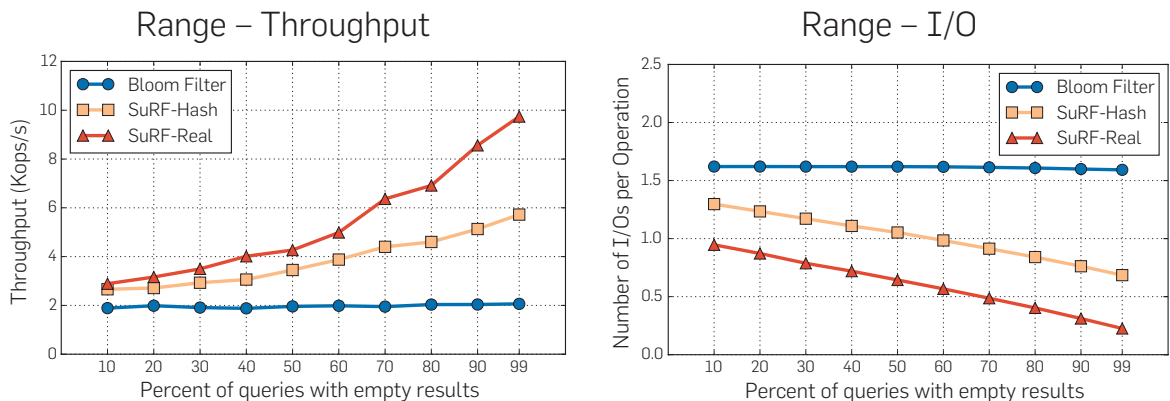
This paper introduces the SuRF filter structure, which supports approximate membership tests for single keys and ranges. SuRF is built upon a new succinct data structure, called the Fast Succinct Trie (FST), that requires only 10 bits per node to encode the trie. FST is engineered to have performance equivalent to state-of-the-art pointer-based indexes. SuRF is memory efficient, and its space and 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>. 

Figure 6. Range queries. RocksDB range query evaluation under different filter configurations and range sizes.



References

1. Facebook MyRocks. <http://myrocks.io/>.
2. Facebook RocksDB. <http://rocksdb.org/>.
3. Google LevelDB. <https://github.com/google/leveldb>.
4. RocksDB Tuning Guide. <https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide>.
5. The InfluxDB storage engine and the time-structured merge tree (TSM). https://docs.influxdata.com/influxdb/v1.0/concepts/storage_engine/.
6. tx-trie 0.18-Succinct Trie Implementation. <https://github.com/hillbig/tx-trie>, 2010.
7. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S. Representing trees of higher degree. *Algorithmica* 4, 43 (2005), 275–292.
8. Bingmann, T. STX B+tree C++ Template Classes. <http://idlebox.net/2007/stx-btree/>, 2008.
9. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of SOCC'10* (2010), ACM, 143–154.
10. Dong, S. Personal communication, 2017. 2017-08-28.
11. Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M. Optimizing space amplification in RocksDB. In *Proceedings of CIDR'17*, Volume 3 (2017), 3.
12. Grossi, R., Ottaviano, G. Fast compressed tries through path decompositions. *J. Exp. Algorithm.* 3–4, 19 (2015).
13. Jacobson, G. Space-efficient static trees and graphs. In *Foundations of Computer Science* (1989), IEEE, 549–554.
14. Lakshman, A., Malik, P., Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* 2, 44 (2010), 35–40.
15. Leis, V., Kemper, A., Neumann, T. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of ICDE'13* (2013), IEEE, 38–49.
16. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Inform.* 4, 33 (1996), 351–385.
17. Rhea, S., Wang, E., Wong, E., Atkins, E., Storer, N. LittleTable: A time-series database and its uses. In *Proceedings of SIGMOD'17* (2017), ACM, 125–138.
18. Sears, R., Ramakrishnan, R. bLSM: A general purpose log structured merge tree. In *Proceedings of SIGMOD'12* (2012), ACM, 217–228.
19. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of SIGMOD'16* (2016), ACM, 1567–1581.
20. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of SIGMOD'18* (2018), ACM, 323–336.

rank and select structures on uncompressed bit sequences. In *Proceedings of SEA'13* (2013), Springer, 151–163.

Huanchen Zhang, Hyeontaek Lim, David G. Andersen, and Andrew Pavlo ([huanche1, hl, dga, pavlo]@cs.cmu.edu), Carnegie Mellon University, Pittsburgh, PA, USA.

Viktor Leis (viktor.leis@uni-jena.de), Friedrich Schiller University, Jena, Germany.

Michael Kaminsky (kaminsky@cs.cmu.edu) BrdgAI, Pittsburgh, PA, USA.

Kimberly Keeton (kimberly.keeton@hpe.com), Hewlett Packard Labs, Palo Alto, CA, USA.

© 2021 ACM 0001-0782/21/4 \$15.00

Semantic Web for the Working Ontologist

Effective Modeling for Linked Data, RDFS, and OWL

**Dean Allemang
James Hendler
Fabien Gandon**

THIRD EDITION

ISBN: 978-1-4503-7617-4

DOI: 10.1145/3382097

<http://books.acm.org>

<http://store.morganclaypool.com/acm>



ACM BOOKS
Collection II