SuRF: Practical Range Query Filtering with Fast Succinct Tries

October 11, 2024

1 Introduction

This paper presents the **Succinct Range Filter** (SuRF), a fast and compact filter that provides exact-match filtering, range filtering, and approximate range counts. Like Bloom filters, SuRF guarantees one-sided errors for point and range membership tests. SuRF can trade between false positive rate and memory consumption, and this trade-off is tunable for point and range queries semi-independently. SuRF is built upon a new space-efficient (succinct) data structure called the **Fast Succinct Trie** (FST). It performs comparably to or better than state-of-the-art uncompressed index structures (B+tree, ART) 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.

2 Fast Succinct Tries

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.

2.1 Background

A tree representation is "succinct" if the space taken by the representation is close to the information-theoretic lower bound (suppose the information-theoretic lower bound is L bits. "close" means $L + O(1)$, $L + o(L)$ or $L + O(L)$), 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 $\{0, 1, \dots, k-1\}$. Since there are $\binom{kn+1}{n}/kn+1$ n-node tries of degree k, the information-theoretic lower bound is approximately $n(k \log_2 k - (k-1) \log_2 (k-1))$ bits.

An ordinal tree is a rooted tree where each node can have an arbitrary number of children in order. Jacobson introduced Level-Ordered Unary Degree Sequence (**LOUDS**) to encode an ordinal tree.

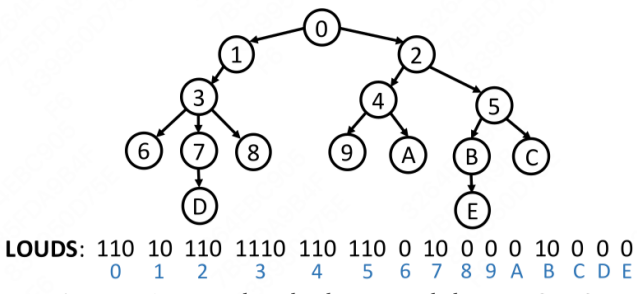

Figure 1: An example of ordinal tree encoded using LOUDS

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. Modern rank & select implementations achieve constant time by using look-up tables (LUTs) to store a sampling of precomputed results.

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

2.2 LOUDS-Dense

LOUDS-Dense encodes each trie node using three bitmaps of size 256 (because the node fanout is 256) and a byte-sequence for the values as shwon in the top half of Figure 2.2.

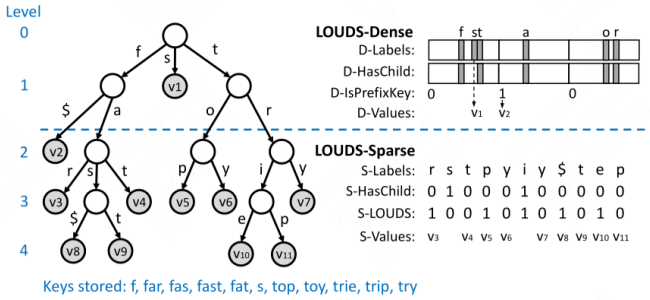

Figure 2: LOUDS-DS Encoded Trie – The $ symbol represents the character whose ASCII number is `0xFF`. It is used to indicate the situation where a prefix string leading to a node is also a valid key.

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.

The second bitmap (**D-HasChild**) indicates whether a branch points to a sub-trie or terminates.

The third bitmap (**D-IsPrefixKey**) include only one bit per node. The bit indicates whether the prefix that leads to the node is also a valid key.

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.

One LOUDS-Dense has only one bitmap of each type.

To get a taste:

```
bool LoudsDense::lookupKey(const std::string &key,
                           position_t &out_node_num) const {
  position_t node_num = 0;
  position_t pos = 0;
  for (level_t level = 0; level < height_; level++) {
    pos = (node_num * kNodeFanout);
    if (level >= key.length()) { // if run out of searchKey bytes
      if (prefixkey_indicator_bits_->readBit(
              node_num)) // if the prefix is also a key
        return suffixes_->checkEquality(getSuffixPos(pos, true), key,
                                        level + 1);
      else
```

```
13          return false;
14      }
15      pos += (label_t)key[level];
16
17      // child_indicator_bitmaps_->prefetch(pos);
18
19      if (!label_bitmaps_->readBit(pos)) // if key byte does not exist
20        return false;
21
22      if (!child_indicator_bitmaps_->readBit(pos)) // if trie branch
        ↪  terminates
23        return suffixes_->checkEquality(getSuffixPos(pos, false), key,
          ↪  level + 1);
24
25      node_num = getChildNodeNum(pos);
26    }
27    // search will continue in LoudsSparse
28    out_node_num = node_num;
29    return true;
30  }
```

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 $\texttt{D-HasChild}[pos] = 1$,

- $\texttt{D-ChildNodePos}(pos) = 256 \times rank_1(\texttt{D-HasChild}, pos)$ computes the bit position of the first child node.
- $\texttt{D-ParentNodePos}(pos) = 256 \times select_1(\texttt{D-HasChild}, \lfloor pos/256 \rfloor)$ computes the bit position of the parent node.
- $\texttt{D-ValuePos}(pos) = rank_1(\texttt{D-Labels}, pos) - rank_1(\texttt{D-HasChild}, pos) + rank_1(\texttt{D-IsPrefixKey}, \lfloor pos/256 \rfloor) - 1$ gives the lookup position. just to find the location of the pointer.

2.3 LOUDS-Sparse

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. We denote the case where the prefix leading to a node is also a value key using the special byte `0xFF` at the beginning of the node.

The second bit sequence **S-HasChild** includes one bit for each byte in **S-Labels** to indicate whether a child branch continues or terminates.

The third bit-sequence **S-LOUDS** also includes one bit for each byte in **S-Labels** denoting node boundaries: if a label is the first in a node, its **S-LOUDS** bit is set.

The final byte-sequence **S-Values** is the same as D-Values.

- to move down, $\texttt{S-ChildNodePos}(pos) = select_1(\texttt{S-LOUDS}, rank_1(\texttt{S-HasChild}, pos)+1)$
- to move up, $\texttt{S-ParentNodePos}(pos) = select_1(\texttt{S-HasChild}, rank_1(\texttt{S-LOUDS}, pos)-1)$
- to access a value, $\texttt{S-ValuePos}(pos) = pos - rank_1(\texttt{S-HasChild}, pos) - 1$

2.4 LOUDS-DS and Operations

We maintain a size ratio R between LOUDS-Sparse and LOUDS-Dense to determine the dividing point among levels. Suppose the trie has H levels. Let $\texttt{LOUDS-Dense-Size}(l)$, $0 \leq l \leq H$ denote the size of LOUDS-Dense-encoded levels up to l (non-inclusive). Let $\texttt{LOUDS-Sparse-Size}(l)$, represent the size of LOUDS-Sparse encoded levels from l (inclusive) to H. The **cutoff** level is defined as the largest l such that .

$$\texttt{LOUDS-Dense-Size}(l) \times R \leq \texttt{LOUDS-Sparse-Size}(l).$$

Reducing R leads to more levels, favoring performance over space. We use $R = 64$ as the default.

LOUDS-DS supports three basic operations efficiently:

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

3 Problems

4 References

References