Databend

Succinct Trie

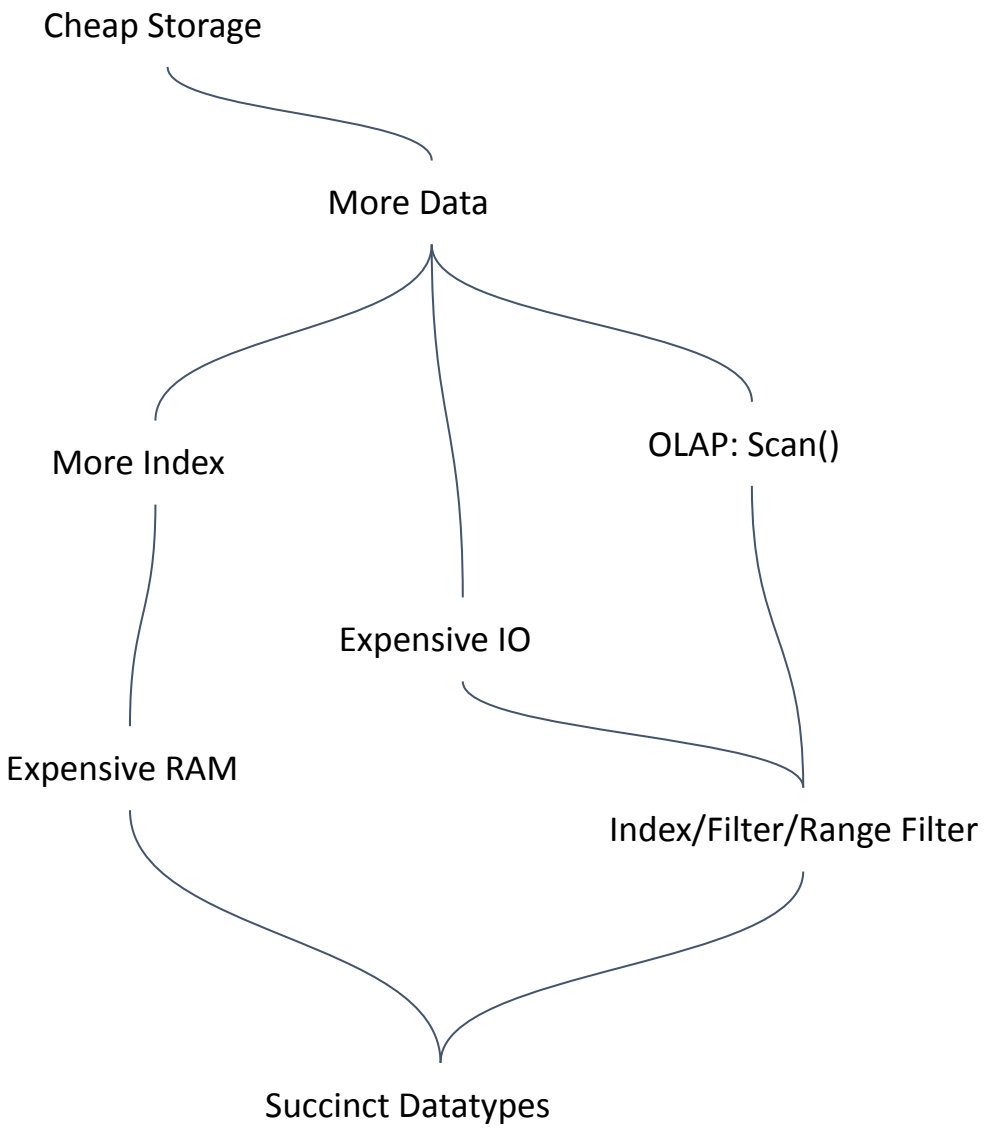
by drmingdrmer @ databend

Databend

1. Why
2. Succinct-Trie
3. Optimization
4. Application

Background & challenges

- 💰 Cost Driven.
- 💰 To index more data with less RAM.
- 🔥 Succinct Datatypes become hot.



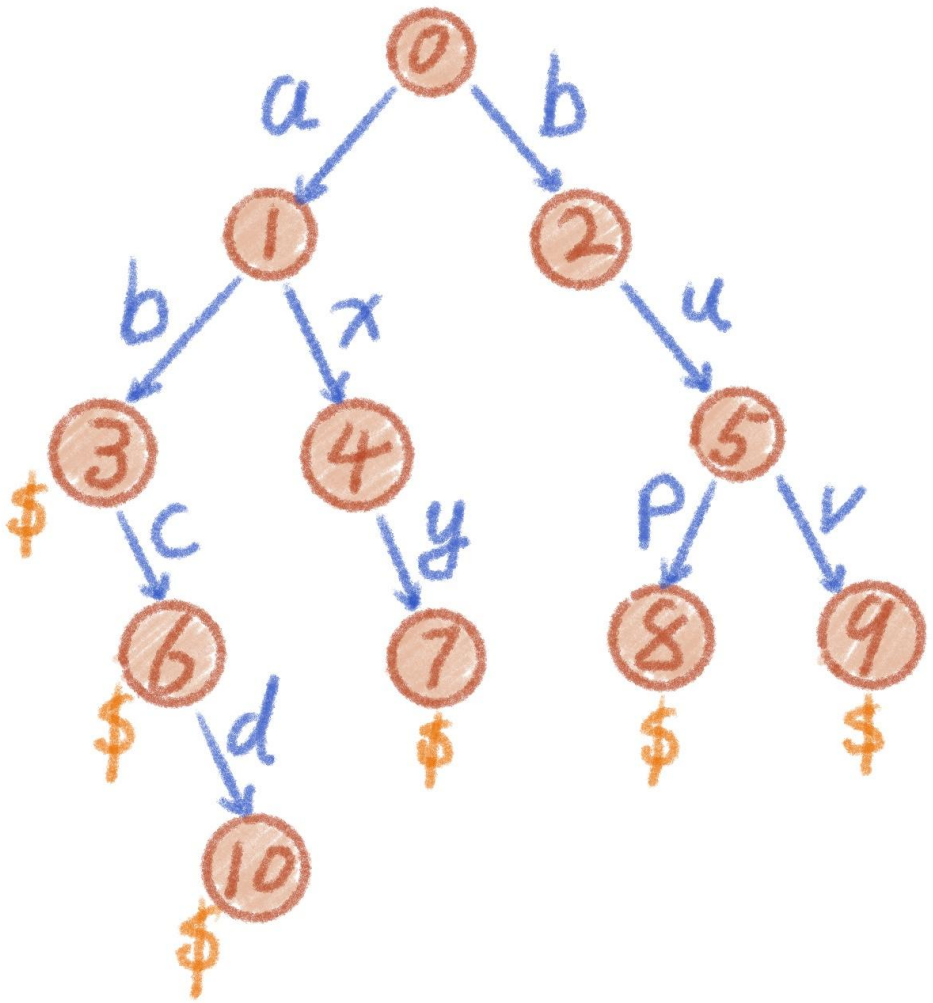
Trie: structure

Store (Sorted) Strings in a Trie:
i.e., every key is a path in a tree:

ab
ab**c**
abc**d**
a**xy**
bup
bu**v**

Storage: 10/18

- Nodes: 0, 1, … 10
- Labels: <byte>
- Leaf: 3, 6, 7, 8, 9, 10



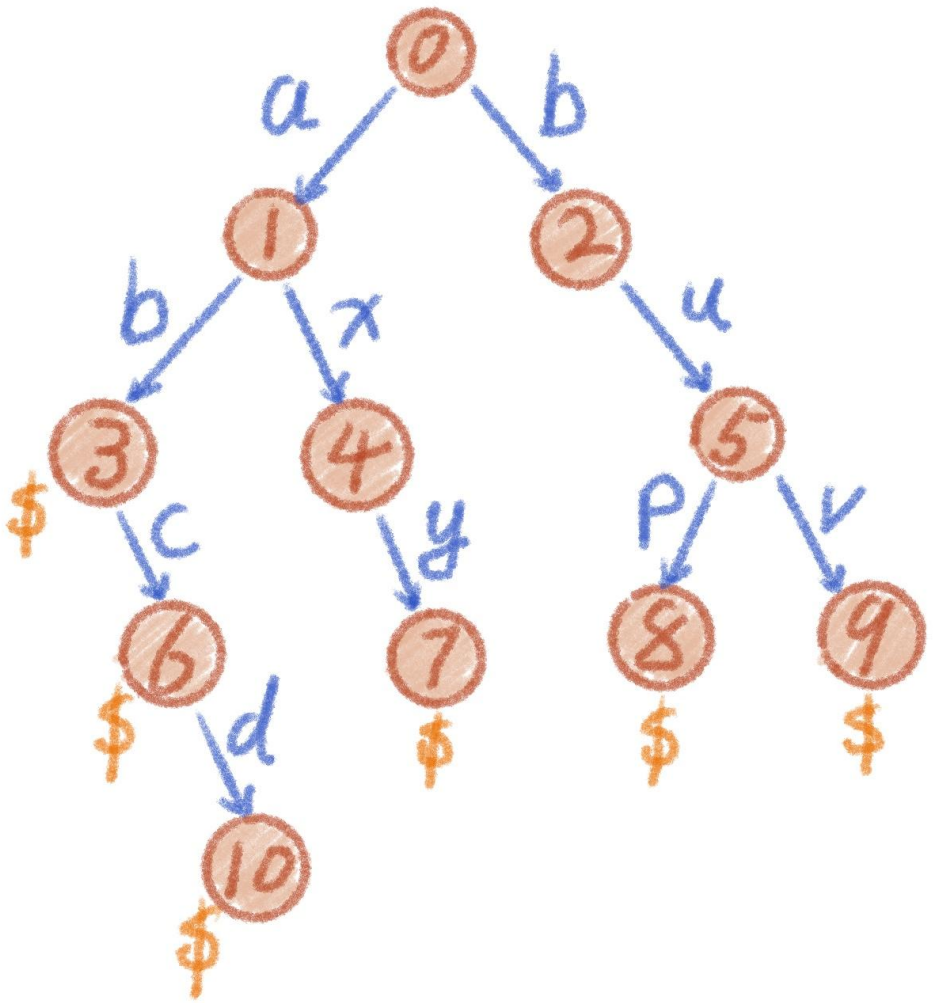
Trie: data type


```
struct Node {
    labels: Vec<u8>,
    children: Vec<Box<Node>>,
    is_leaf: bool
}

# Lookup:

let node = root;
for label in key {
    node = node.walk_along_label(label)?;
}
return Ok(node.is_leaf);
```

✅ Prefix compression
❌ One pointer per node(8 byte/node).
❌ One pointer to labels(8 byte/node).

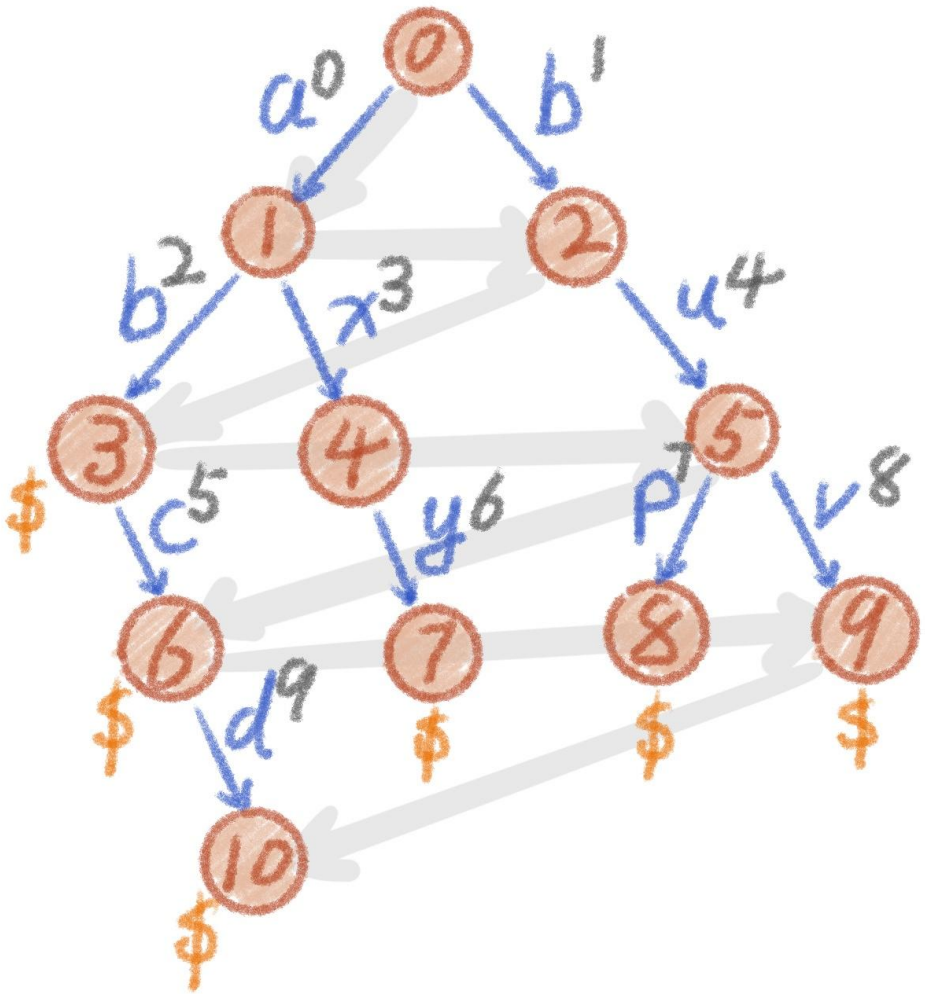


Succinct-Trie: static trie

`node = node.walk_along_label(label)` requires:
1. Find node labels;
2. Find label target node.

```
0  1  2 3 4 5  6 789A // node id
01 23 4 5 6 78 9      // label index
ab bx u c y pv d      // 📦 labels
001001010101001011111 // 📦 bitmap
```

- bitmap:
  `0` indicates a label.
  `1` indicates the end of a node.



Succinct-Trie: op1: walk-to-sub-node

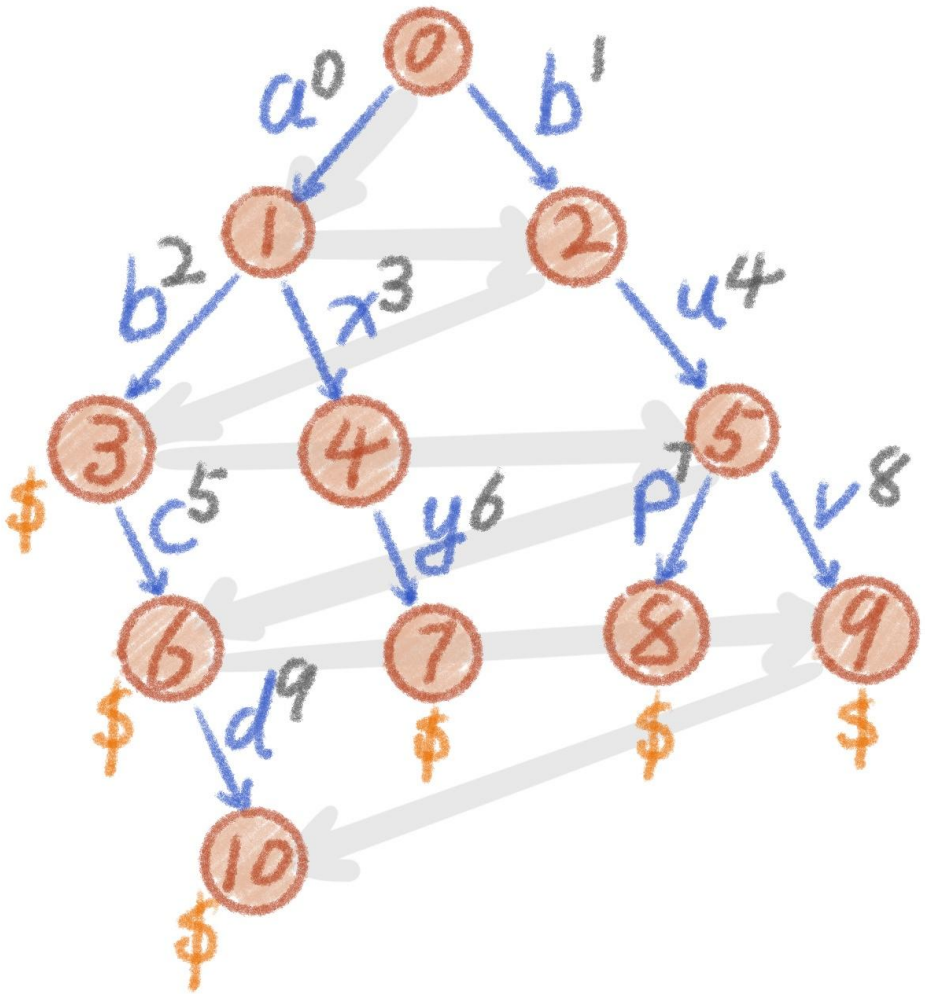
Find sub node by label:

e.g.: (node-5, label "p") -> node-8

```
0  1  2 3 4 5  6 789A // node id
01 23 4 5 6 78 9      // label index
ab bx u c y pv d      // 📦 labels
0010010101010010111 11 // 📦 bitmap
            ^bm_i=12
```

node_id = label_index + 1

node_id = bitmap.count_zeros(0..=bm_i)
count_ones = population count



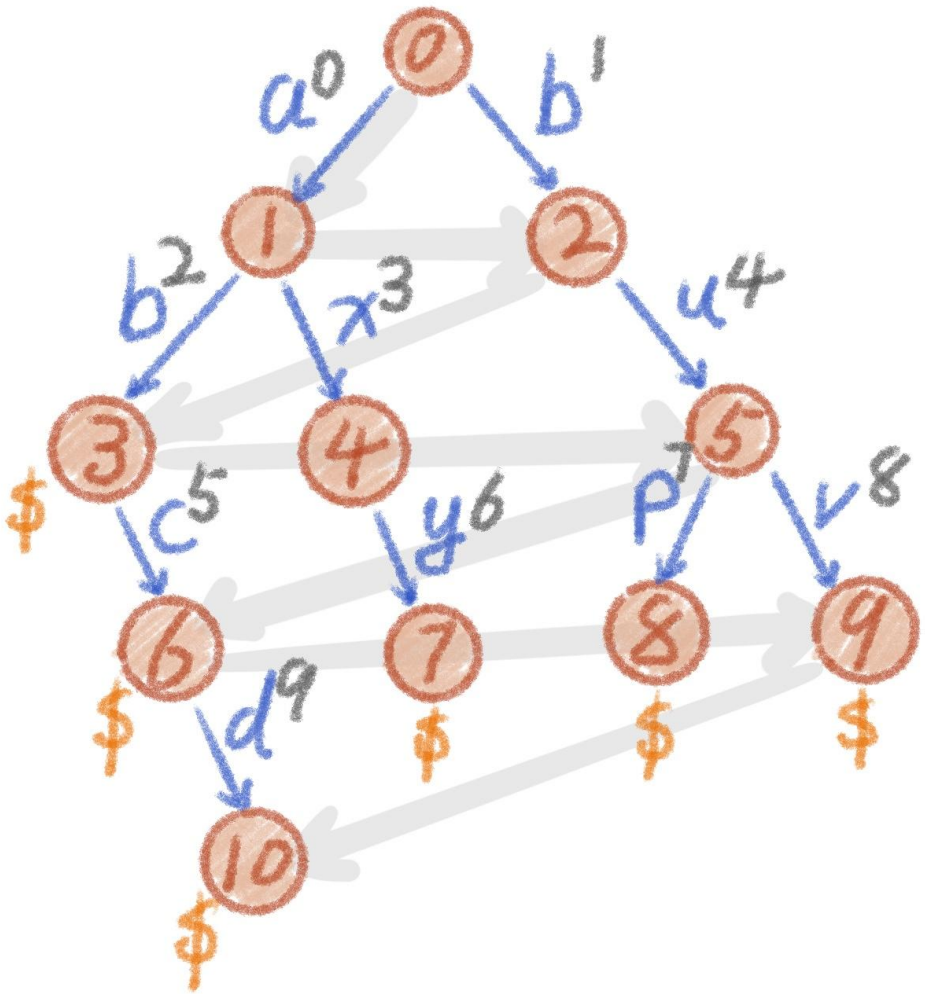
Succinct-Trie: op2: find-labels

Find labels by node id:

e.g.: node-5 -> bm_i: 12

```
0  1  2 3 4 5  6 789A // node id
01 23 4 5 6 78 9      // label index
ab bx u c y pv d      // 📦 labels
0010010101010010111 1 // 📦 bitmap
            ^bm_i=12
```

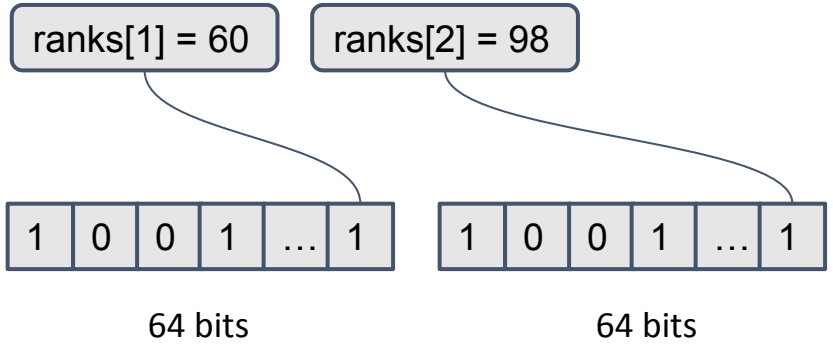
bm_i = bitmap.find_ith_one(node_id-1)+1



Succinct-Trie: ~ O(1) count_ones() & find_ith_one()


```
# count_ones() // AKA rank()

1. ranks[i] = count_ones(0..=i*64)
```

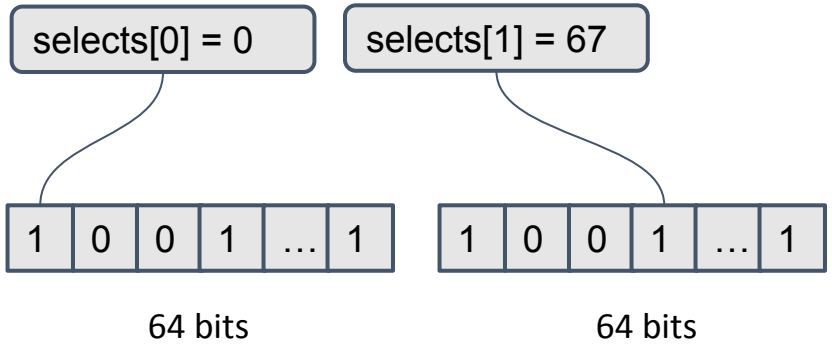



```
2. POPCNT — Return the Count of Number of Bits
Set to 1

let mask = (1 << (i&63)) - 1;
rank_index[(i / 64)]
  + (bitmap[i & 63] & mask).count_ones();
```

```
# find_ith_one() // AKA select()

1. selects[i]: count_ones(0..selects[i]) == i
```



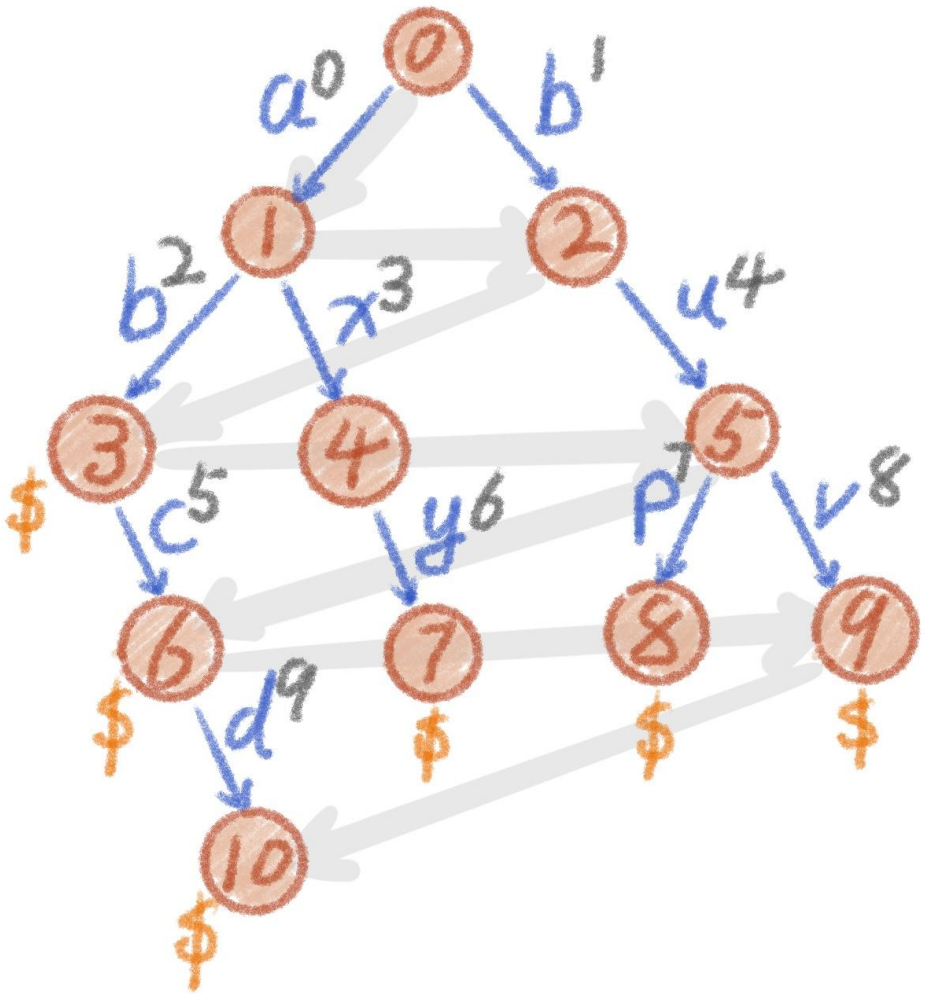

```
2. Lookup table

000010001

lookup8[0]=4
lookup8[1]=7
lookup8[2]=8
```

Succinct-Trie: leaves


```
0  1  2 3 4 5  6 789A // node id
ab bx u c y pv d      //  labels
00100101010100101111 //  bitmap
0  0  0 1 0 0  1 1111 // is_leaf
```



Succinct-Trie: storage


```
0  1  2 3 4 5  6 789A // node id
01 23 4 5 6 78 9      // label index
ab bx u c y pv d      // 📦 labels
0010010101010010111 1 // 📦 bitmap
0  0  0 1 0 0  1 1111 // 📦 is_leaf
```

Succinct-Trie: query


```rust
pub fn has(&self, key: &[u8]) -> bool {
    let mut node_id: i32 = 0;
    let mut label_bitmap_index: i32 = 0;

    for c in key {
        loop { // linear search for a matching label
            if get_bit(&self.label_bitmap.words, label_bitmap_index as usize) != 0 {
                return false;
            }

            if self.labels[(label_bitmap_index - node_id) as usize] == *c {
                break;
            }

            label_bitmap_index += 1;
        }

        // go to next level
        node_id = self.label_bitmap.count_zeros(label_bitmap_index + 1).0;
        label_bitmap_index = self.label_bitmap.select_ith_one(node_id - 1) + 1;
    }

    get_bit(&self.leaves, node_id as usize) != 0
}
```

Succinct-Trie: perf: unoptimized

200,000 web words
~2480 KB
Load pattern: zipf(s=1.5)

Data	Engine	Size(KB)	Size/original	ns/op
200kweb2	bsearch	5890	267%	229
200kweb2	succinct.Set	1258	57%	356
200kweb2	btree	12191	553%	483

```
A
Aani
Aaron
Aaronic
Aaronical
Aaronite
Aaronitic
Aaru
Ab
Ababdeh
Ababua
Abadite
Abama
Abanic
Abantes
Abarambo
Abaris
Abasgi
Abassin
Abatua
Abba
Abbadide
Abbasside
Abbie
Abby
Abderian
Abderite
Abdiel
Abdominales
Abe
…
Abel
Abelia
Abelian
Abelicea
Abelite
Abelmoschus
Abelonian
Abencerrages
Aberdeen
Aberdonian
Aberia
Abhorson
Abie
Abies
Abietineae
Abiezer
Abigail
Abipon
Abitibi
Abkhas
Abkhasian
Ablepharus
Abnaki
Abner
Abo
Abobra
Abongo
Abraham
Abrahamic
Abrahamidae
Abrahamite
Abrahamitic
Abram
Abramis
Abranchiata
Abrocoma
Abroma
Abronia
Abrus
Absalom
Absaroka
Absi
Absyrtus
Abu
Abundantia
Abuta
Abutilon
Abyssinian
Acacia
Acacian
Academic
Academus
Acadia
Acadian
Acadie
Acaena
Acalepha
Acalephae
Acalypha
Acalypterae
```

Succinct-Trie: optimization

1-Bitmap root:
Transform the topmost levels to a bitmap(65536).

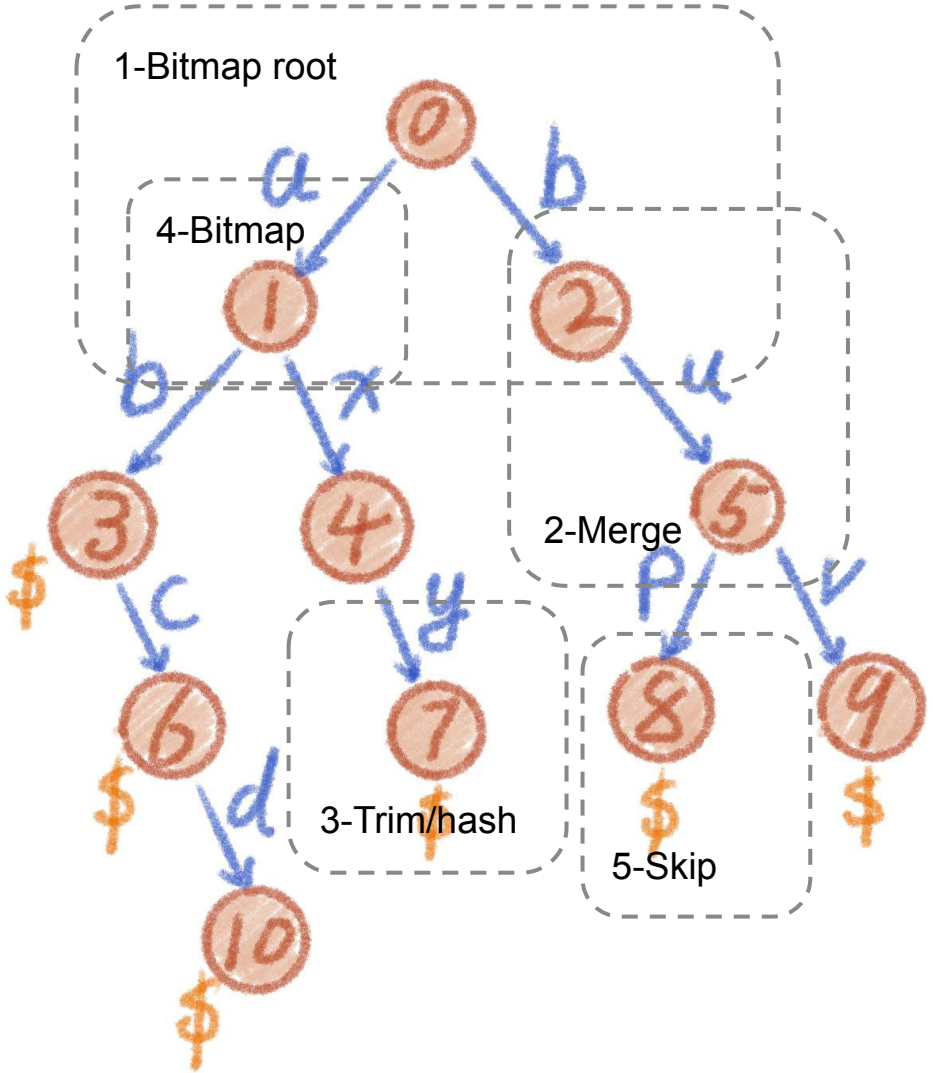
2-Merge:
Merge single-label nodes.

3-Trim/hash: FP up
- Remove bottom single-label leaves.
- Replace a `tail` with hash(tail).

4-Bitmap:
Replace inner nodes with bitmap(16/256).

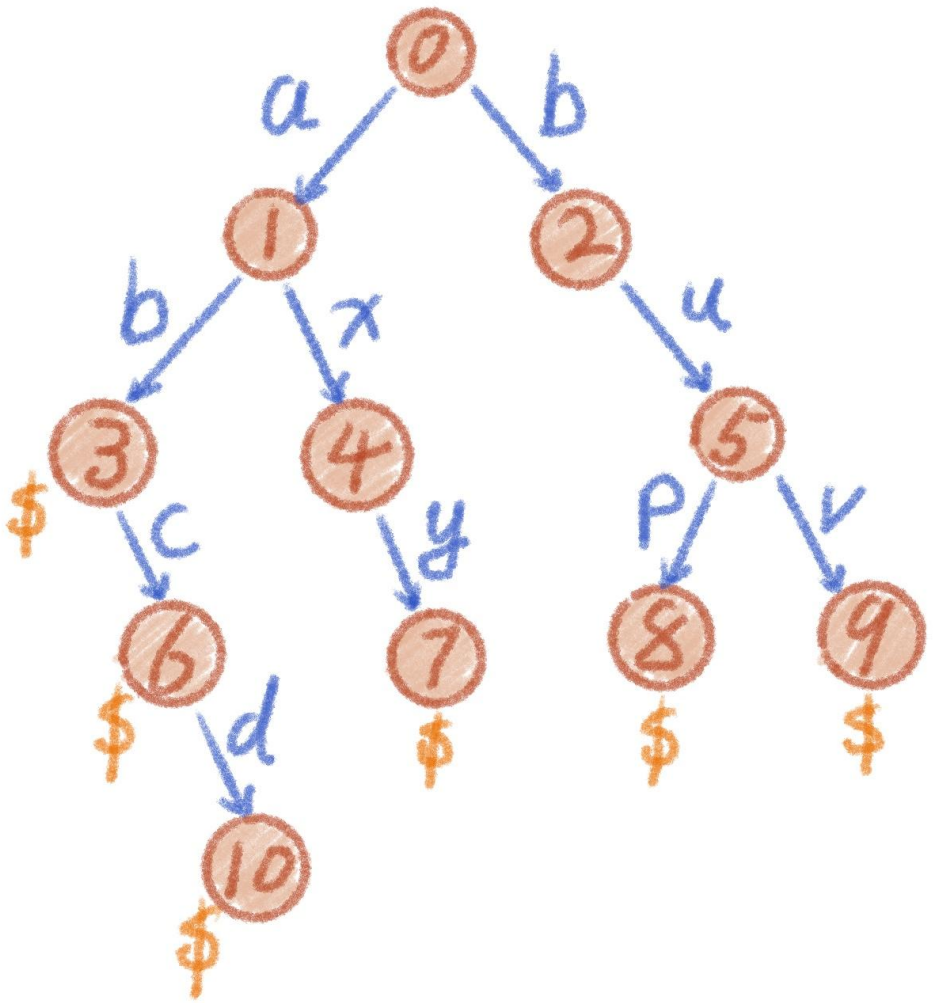
5-Skip: FP up
ab ~~abc~~ ~~abcd~~ axy ~~bup~~ buv

6-Transparent bitmap compression.
"000100100" -> "01001"
- has to hold: consistent `count_ones()`.



Succinct-Trie: characteristics

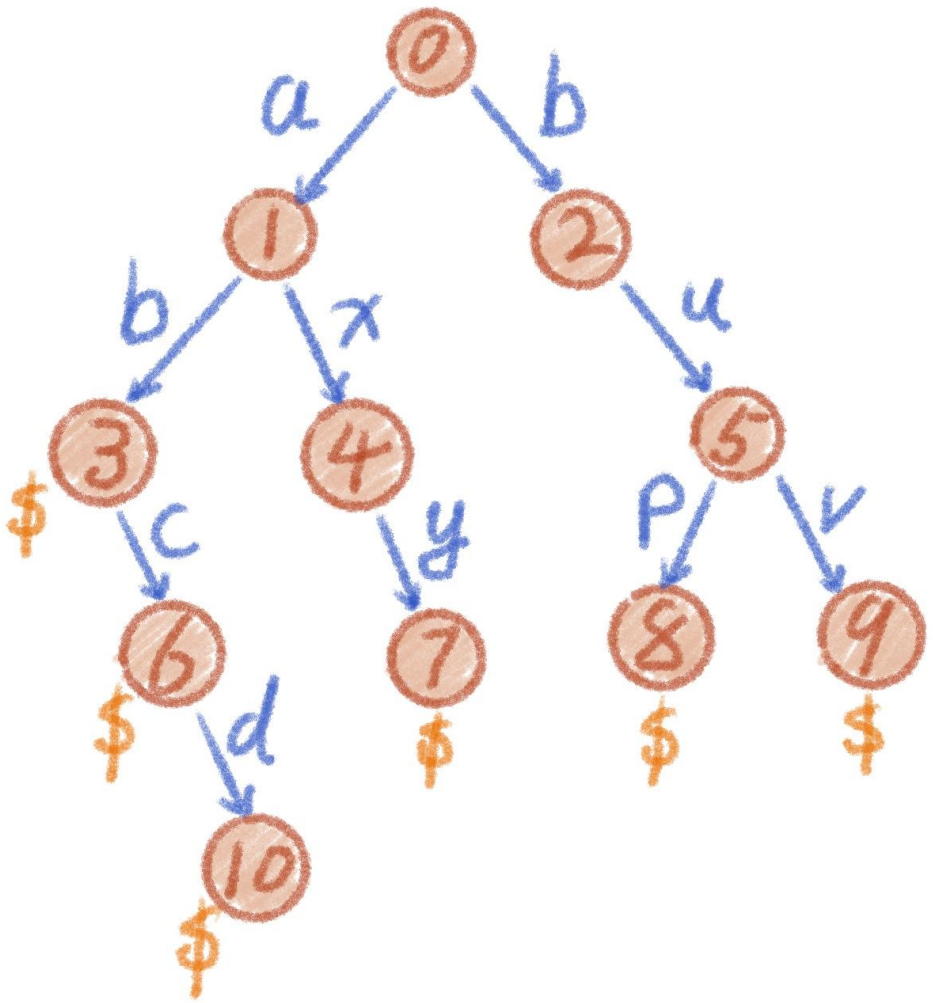
- ❌ Update
- ✅ Range query
- ✅ Point query
- 🤔 Cache friendly
- 🤔 Vectorization
- ✅ Large data set
- 🤔 Small data set
- 🤔 Performance: ~ bsearch,tree
- ❌ Performance: 0.2X hash_map



Succinct-Trie: applications

Index of KV
- In a private project:
  ~ 4.5 bit/key
  ~ 200 ns/get

Filter
- SuRF:
  ~ million keys
  ~ 10 bit/key
  ~ 200 ns/get



Succinct-Trie: refs

Source code(rs):
https://github.com/drmingdrmer/succinct-rs

SuRF(cpp): tail hash, root node is bitmap(65536), SIMD
https://www.cs.cmu.edu/~pavlo/papers/mod601-zhangA-hm.pdf
https://github.com/efficient/SuRF

slim(go): bitmap nodes
https://github.com/openacid/slim

Databend

Thank You \ && Q&A