Succinct Trie Indexes Made Practical

Huanchen Zhang

David G. Andersen, Michael Kaminsky, Andrew Pavlo, Kimberly Keeton
DRAM price won’t fall forever
Memory-efficient data structures are helpful

Smaller data structures

More data resident in faster memory

Better performance + lower costs
The limit: information-theoretic lower bound (ITLB)

The minimum # of bits required to distinguish any object in a class

$|S| = n \Rightarrow \log_2 n \text{ bits}$
The limit: information-theoretic lower bound (ITLB)

The minimum # of bits required to distinguish any object in a class

\[ |S| = n \]

\[ \log_2 n \text{ bits} \]

In-node trie of degree \( k \)

\[ = \binom{kn+1}{n}/kn + 1 \]

\[ n(\log_2 k - (k - 1)\log_2 (k - 1)) \text{ bits} \]
The limit: information-theoretic lower bound (ITLB)

The minimum # of bits required to distinguish any object in a class

$|S| = n \quad \implies \quad \log_2 n$ bits

$\text{In-node trie of degree } k \quad \implies \quad n(\log_2 k - (k - 1)\log_2(k - 1))$ bits

$= \binom{kn+1}{n}/kn + 1 \quad \implies \quad 9.44n$ bits

$\text{for } k = 16 \quad \implies \quad 256$ bits
The limit: information-theoretic lower bound (ITLB)

The minimum # of bits required to distinguish any object in a class

\[ |S| = n \]

\[ \text{In-node trie of degree } k \]

\[ = \binom{kn+1}{n}/kn + 1 \]

\[ \downarrow \]

256

\[ \downarrow \]

log₂n bits

\[ \downarrow \]

\[ n(\log_2 k - (k - 1)\log_2(k - 1)) \] bits

\[ = 9.44n \]

FST = 10n
Succinct Data Structures

Use # of bits close to ITLB

Suppose ITLB = L bits

Implicit: \( L + O(1) \)

Succinct: \( L + o(L) \)

Compact: \( O(L) \)

FST
Why aren’t succinct data structures popular?

- Read-only
- Log-structured design
- Slow
- Complex
Existing succinct tries are slow

- 50M 64-bit integer keys

**Lookup Latency**

- ART: 3
- tx-trie: 0.5
- PDT: 1.5

**Memory** including key suffixes

- ART: 0.5
- tx-trie: 0.5
- PDT: 0
Fast Succinct Trie (FST) is fast **and** small

50M 64-bit integer keys

**Lookup Latency**

<table>
<thead>
<tr>
<th>ART</th>
<th>tx-trie</th>
<th>PDT</th>
<th>FST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Memory** including key suffixes

<table>
<thead>
<tr>
<th>ART</th>
<th>tx-trie</th>
<th>PDT</th>
<th>FST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Encoding Mechanism
3 ways to succinctly encode ordinal trees

Ordinal tree: a rooted tree where each node can have an arbitrary # of children in order
3 ways to succinctly encode ordinal trees

In-node ordinal tree $C_n = \frac{1}{n+1} \binom{2n}{n} \approx 2n$ bits
3 ways to succinctly encode ordinal trees

1. **LOUDS**: level-ordered unary degree sequence

![Diagram of a tree with levels and degrees marked]
3 ways to succinctly encode ordinal trees

1. LOUDS: 110 10 110 1110 110 110 0 10 0 0 0 10 0 0 0

Diagram:

```
  0
 /\  \
1  2
 /\  \
3  4 5
 /\  \\
6  7 8 / 9 A B C
 / \ / \ / \ / \
D  10 0  10 0  10 0
   0   0   0
```
3 ways to succinctly encode ordinal trees

BP: balanced parenthesis
3 ways to succinctly encode ordinal trees

BP: (((()(()(()())))((()())(()(()()))))
3 ways to succinctly encode ordinal trees

BP: ( (( ( ) ( ( ) ) ( ) ) ) ( ( ( ) ( ) ) ( ( ( ) ) ( ) ) ) )

Diagram:

- Level 0: 0
- Level 1: 1, 2
- Level 2: 3, 4, 5
- Level 3: 6, 7, 8, 9, A, B, C
- Level 4: D, E
3 ways to succinctly encode ordinal trees

DFUDS: depth-first unary degree sequence
3 ways to succinctly encode ordinal trees

DFUDS: ( ( ( ) ( ) ( ( ( ) ) ( ) ) ) ( ( ) ( ( ) ) ) ( ( ) ( ) ) )
3 ways to succinctly encode ordinal trees

DFUDS: ( ( ) ( ) ( ( ( ) ) ( ) ) ) ( ( ) ( ( ) ) ) ( ( ) ( ) ) ( ( ) ( ) )

0 1 3 6 7 D 8 2 4 9 A 5 B E C
LOUDS-Sparse: succinctly encode tries

L: fst$s a or rs t y p i y$tep
HC: 101011101000100000
S: 100101010010101010
V: v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11
LOUDS-Sparse: succinctly encode tries

Why LOUDS?
1. Fast tree nav.
2. Good label locality
3. Easy implementation
### Rank & select on bit-vectors

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**bv:** 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0

**rank(bv, i) =**  
# of 1’s in bv up to position i

**select(bv, i) =**  
position of the ith 1 in bv

**Examples:**
rank(bv, 7) = 4
select(bv, 7) = 14
Compute rank & select in constant time

The classic algorithm for computing rank

$\text{bv}$
Compute rank & select in constant time

The classic algorithm for computing rank

super block = $lg^2 n$ bits

$bv$
Compute rank & select in constant time

The classic algorithm for computing rank

- super block = $\log^2{n}$ bits
- basic block = $\frac{1}{2}\log{n}$ bits
Compute rank & select in constant time

The classic algorithm for computing rank

super block = $\log^2 n$ bits

basic block = $\frac{1}{2} \log n$ bits

per super block

cumulative rank
Compute rank & select in constant time

The classic algorithm for computing rank

- Super block = $\log^2 n$ bits
- Basic block = $\frac{1}{2} \log n$ bits

Cumulative rank per super block
Rank in super block per basic block
Compute rank & select in constant time

The classic algorithm for computing rank

- Super block = $lg^2 n$ bits
- Basic block = $\frac{1}{2} lgn$ bits

bv

- Per super block cumulative rank
- Per basic block rank in super block
- Within super block all possible queries
Compute rank & select in constant time

The classic algorithm for computing rank

- Super block = \( \log^2 n \) bits
- Basic block = \( \frac{1}{2} \log n \) bits

Flowchart:
- Per super block cumulative rank
- Per basic block rank in super block
- Within super block all possible queries
Compute rank & select in constant time

The classic algorithm for computing rank

- Super block = $\log^2 n$ bits
- Basic block = $\frac{1}{2} \log n$ bits

- Per super block cumulative rank
- Per basic block rank in super block
- Within super block all possible queries

\[
\left\lfloor \frac{i}{\log^2 n} \right\rfloor \quad \left\lfloor \frac{i}{\frac{1}{2} \log n} \right\rfloor \quad \left\lfloor \frac{i}{\frac{1}{2} \log n} \right\rfloor
\]
Compute rank & select in constant time

The classic algorithm for computing rank

super block = $lg^2 n$ bits

basic block = $\frac{1}{2} lg n$ bits

per super block
  cumulative rank

$\left\lfloor \frac{i}{lg^2 n} \right\rfloor$

per basic block
  rank in super block

$\left\lfloor \frac{i}{\frac{1}{2} lg n} \right\rfloor$

within super block
  all possible queries

remaining bits

remaining bits
Compute rank & select in constant time

The classic algorithm for computing rank

- **super block** = \(lg^2n\) bits
- **basic block** = \(\frac{1}{2}lgn\) bits

Per super block:
- cumulative rank

Per basic block:
- rank in super block

Within super block:
- all possible queries

\(\left\lfloor \frac{i}{lg^2n} \right\rfloor\) → \(\left\lfloor \frac{i}{lgn} \right\rfloor\) → remaining bits → O(1) time
Compute rank & select in constant time

The classic algorithm for computing rank

super block = \( lg^2 n \) bits

basic block = \( \frac{1}{2} lg n \) bits

per super block cumulative rank

per basic block rank in super block

within super block all possible queries

space: \( O\left(\frac{n}{\lg n}\right) \) + \( O\left(\frac{n}{\lg n} \lg \lg n\right) \) + \( O(\sqrt{n} \lg n \lg \lg n) \equiv o(n) \)

O(1) time
Compute rank & select in constant time

The classic algorithm for computing rank

- super block = $lg^2 n$ bits
- basic block = $\frac{1}{2} lgn$ bits

\[
\begin{align*}
&\left\lceil \frac{i}{lg^2 n} \right\rceil \\
&\left\lceil \frac{i}{lgn} \right\rceil \\
&\text{remaining bits}
\end{align*}
\]

- per super block cumulative rank
- per basic block rank in super block
- within super block all possible queries

space: $O\left(\frac{n}{lgn}\right) + O\left(\frac{n}{lgn} lglgn\right) + O\left(\sqrt{n} lgn lglgn\right) = o(n)$

Select is similar but trickier, often based on rank structures
Tree navigation relies on rank & select

child(i) = select(S, rank(HC, i)+1)

parent(i) = select(S, rank(S, i)-1)

value(i) = i - rank(HC, i)
Tree navigation relies on rank & select

child(i) = select(S, rank(HC, i)+1)
parent(i) = select(S, rank(S, i)-1)
value(i) = i - rank(HC, i)
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1) \\
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]

\[L: \text{fst}$\text{a}$\text{ors}\text{ty}$\text{pi}$\text{y}\text{step}
\]

\[HC: 10101110100010000000
\]

\[S: 10010101001010101010
\]

\[V: v_1 \ v_2 \ v_3 \ v_4 v_5 v_6 \ v_7 v_8 v_9 v_{10} v_{11}
\]
Tree navigation relies on rank & select

$$\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1)$$

$$\text{value}(i) = i - \text{rank}(HC, i)$$
Tree navigation relies on rank & select

\[
\text{L: } f s t a o r r s t y p i y s t e p \\
\text{HC: } 101011101000100000 \\
\text{S: } 10010101001010101010 \\
\text{V: } v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} \\
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1) \\
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[ \text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1) \]

\[ \text{value}(i) = i - \text{rank}(HC, i) \]
Tree navigation relies on rank & select

child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[ \text{child}(i) = \text{select}(\text{S}, \text{rank}(\text{HC}, i)+1) \]

\[ \text{value}(i) = i - \text{rank}(\text{HC}, i) \]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1) \\
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i)+1) \\
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1)
\]

\[
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

$L: \text{first}\ s\ a\ o\ r\ s\ t\ y\ i\ p\ y\ $ Step

$HC: 101011101000100000$

$S: 10010101001010101010$

$V: v_1 \ v_2 \ v_3 \ v_4 v_5 v_6 \ v_7 v_8 v_9 v_{10} v_{11}$

child($i$) = select($S$, rank($HC$, $i$)+1)

value($i$) = $i$ - rank($HC$, $i$)

0 5 10 15

16 7
Tree navigation relies on rank & select

\[
\text{child}(i) = \text{select}(S, \text{rank}(HC, i) + 1) \\
\text{value}(i) = i - \text{rank}(HC, i)
\]
Tree navigation relies on rank & select

child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)
Performance Optimization
LOUDS-Dense: optimize for the common case

Frequently visited

Divided by size ratio

Majority of nodes
LOUDS-Dense: optimize for the common case

Fast

Divided by size ratio

Space-efficient
LOUDS-Dense: optimize for the common case

Fast

L: $\text{fst}\,a\,or$

HC: 1010111

S: 1001010

V: $v_1\ v_2$

Space-efficient
LOUDS-Dense: optimize for the common case

Fast

L:

HC:

IsPrefixKey:

V:

Space-efficient
LOUDS-DS: best of both worlds

Space-efficient
Observations: rank & select on LOUDS-DS

1. Either rank or select is required, but not both

2. Space taken by auxiliary structures is small

3. The bit vector \( (S) \) that requires select is dense
Optimizing rank structure for LOUDS-DS

- rank LUT
- bit-vector
- basic block

size = 512 bits for LOUDS-Sparse
size = 64 bits for LOUDS-Dense

within basic block: use popcount instruction
Optimizing rank structure for LOUDS-DS

select samples
(every x 1’s)

bit-vector
Optimizing label search algorithm using SIMD

L: abcdeghijklmnopqrstuvwxyzfst...

S: 1000 ... 100...
Optimizing label search algorithm using SIMD

128-bit SIMD

L: \textbf{abcdefghijklmnopqrstuvwxyzfst} \ldots

S: 1000 \ldots 100 \ldots

node boundary
Optimizing label search algorithm using SIMD

L: abcdeghijklmnopqrstuvwxyz fst ...  

S: 1000 ... 100 ...  

node boundary
Using prefetching to hide memory latency

L: fst$ a o r r s t y p i y$ t e p

HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0

V: \( v_1 \) \( v_2 \) \( v_3 \) \( v_4 v_5 v_6 \) \( v_7 v_8 v_9 v_{10} v_{11} \)
What makes FST fast

50M 64-bit integer keys

Lookup Latency

ns

baseline  LOUDS-Dense  rank opt  select opt  SIMD search  prefetch
Where is FST in the performance-space trade-off

50M 64-bit integer keys

Cost function: \( C = PrS \)

- \( r > 1 \) favors performance
- \( r < 1 \) favors space

\( r = 1 \)

Latency(ns) vs. Memory(MB)

- B+tree
- ART
- C-ART
- FST
Conclusion

1. LOUDS-DS
   - A hybrid approach to succinctly encode tries

2. FST
   - Matches performance of pointer-based tries
   - 4 – 15x faster than existing succinct tries
   - Consumes only close-to-optimal space