Tree-Encoded Bitmaps

by Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, Alfons Kemper (SIGMOD 2020)

Presentation by **Anna Bolotina**

Seminar on Advances in Database Research

December 5, 2024

Paris Lodron University of Salzburg

Introduction: Bitmap Index

Picture source: Harald Lang. *Query Processing on Modern Hardware*. PhD thesis, TU Munich, 2023.

Base data

TID	A
0	1
1	3
2	3
3	7
4	4
5	7
6	4
7	1

⇨

Bitmap Index

		1	3	4	7
TID		B_0	B_1	B_2	B_3
0		**1**	0	0	0
1		0	**1**	0	0
2		0	**1**	0	0
3		0	0	0	**1**
4		0	0	**1**	0
5		0	0	0	**1**
6		0	0	**1**	0
7		**1**	0	0	0

Figure 4.1: A basic bitmap index.

- Questions:
  - How to efficiently **compress** bitmaps?
  - How to perform efficient **logical operations** (e.g., intersection) over bitmaps?

Introduction: Bitmap Index

Picture source: Harald Lang. *Query Processing on Modern Hardware*. PhD thesis, TU Munich, 2023.

Base data

TID	A
0	1
1	3
2	3
3	7
4	4
5	7
6	4
7	1

⇨

Bitmap Index

	1	3	4	7
TID	B_0	B_1	B_2	B_3
0	**1**	0	0	0
1	0	**1**	0	0
2	0	**1**	0	0
3	0	0	0	**1**
4	0	0	**1**	0
5	0	0	0	**1**
6	0	0	**1**	0
7	**1**	0	0	0

Figure 4.1: A basic bitmap index.

- Questions:
  - How to efficiently **compress** bitmaps?
  - How to perform efficient **logical operations** (e.g., intersection) over bitmaps?

  Tree-encoded bitmaps offer logarithmic random access time ⇒ faster logical operations.

Introduction: Tree-Encoded Bitmaps (TEB)

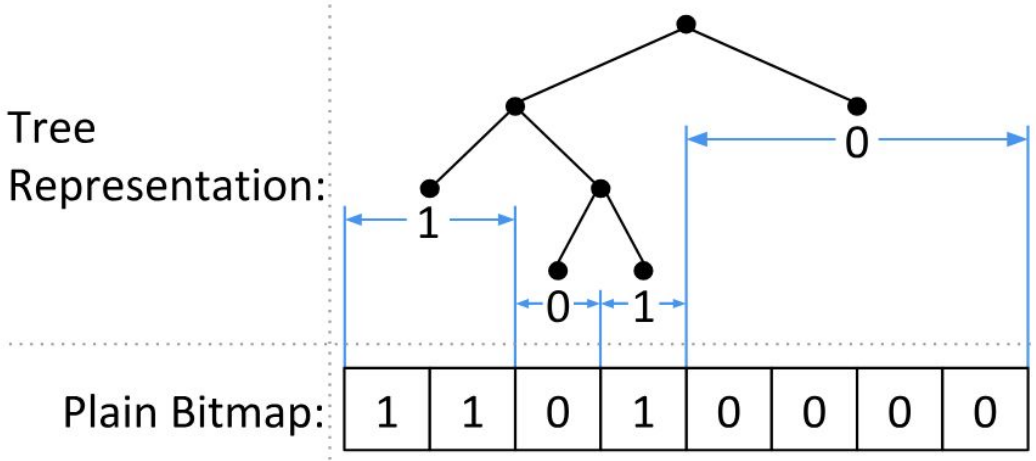


Figure 1: The key idea is to represent bitmaps as full binary trees. Longer runs are mapped to tree nodes closer to the root, and vice versa.

Introduction: TEB vs. Roaring Bitmaps

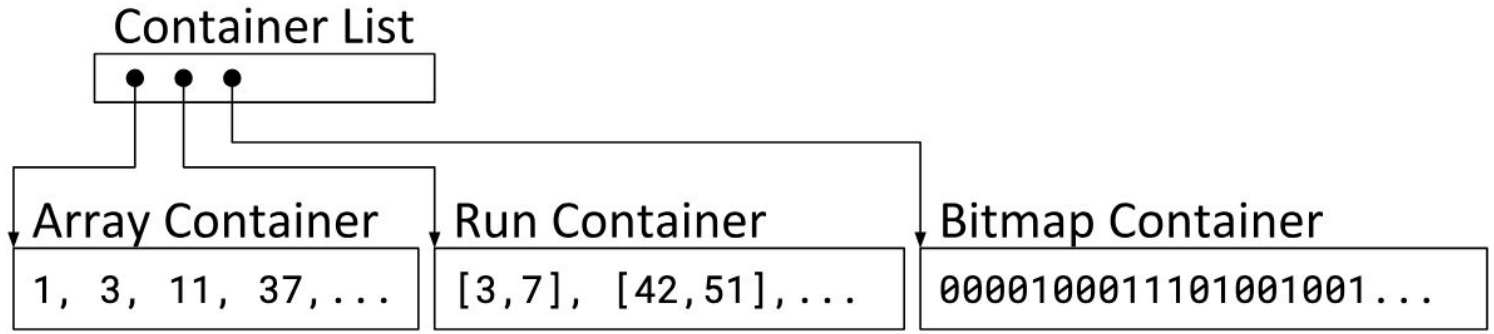


Figure 2: Roaring partitions the bitmap and stores each partition using the best suitable container type.

Notation

- n denotes the length of a bitmap.
- Two metrics characterize data distribution in individual bitmaps:
  - **Bit density** d—the fraction of bits set to 1, $0 \leq d \leq 1$.The total number of set bits is therefore $d \cdot n$.
  - **Clustering factor** f, $1 \leq f \leq n$—the degree of clustering of the 1-bits in a bitmap, i.e., how likely a 1-bit is followed by another 1-bit.

    Formally, f is defined as the **average length** of the 1-runs in a bitmap.
    Example: The bitmap $01110010 \Rightarrow d = 0.5$ and $f = 2$.

Notation

- n denotes the length of a bitmap.
- Two metrics characterize data distribution in individual bitmaps:
  - **Bit density** d—the fraction of bits set to 1, $0 \leq d \leq 1$.The total number of set bits is therefore $d \cdot n$.
  - **Clustering factor** f, $1 \leq f \leq n$—the degree of clustering of the 1-bits in a bitmap, i.e., how likely a 1-bit is followed by another 1-bit.

    Formally, f is defined as the **average length** of the 1-runs in a bitmap.
    Example: The bitmap $01110010 \Rightarrow d = 0.5$ and $f = 2$.
  - Restrictions on d and f:
    - The clustering factor cannot exceed the total number of bits set: $f \leq d \cdot n$.
    - When the bit density exceeds 50%, the smallest possible value for f increases as well. In the general case, the **smallest** possible clustering is $\max(1, d/(1-d))$.

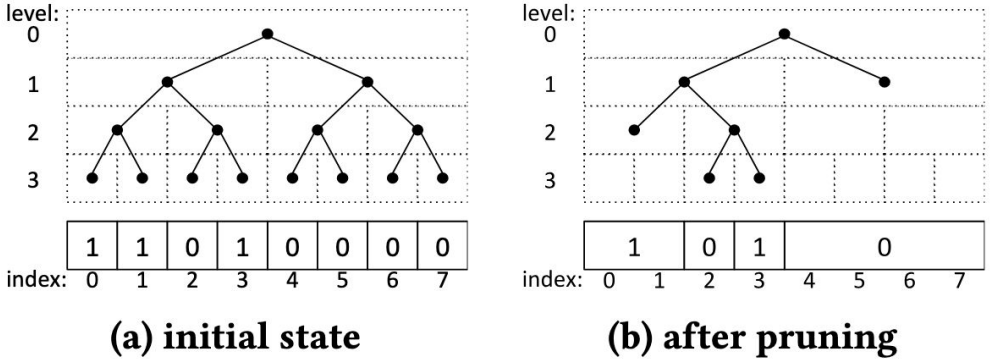
      Example: The bitmap $11010101 \Rightarrow d = 0.625$ and $f = 1.25$.

Tree-Encoded Bitmaps (TEB)

- Compression
- Encoding (basic version)
- Optimizations
  - Implicit Tree Nodes
  - Implicit Labels

Compression

- A TEB is constructed in two phases.
- In the first phase, a **perfect binary tree** is established on top of a given bitmap.
- Only leaf nodes (**labels**) carry a payload.
- A label can either be a 0-bit or a 1-bit.
- In the second phase, the binary tree is **pruned bottom-up** (lossless compression).
- Initially $2n-1$ nodes (and n labels), assuming n is a power of two.



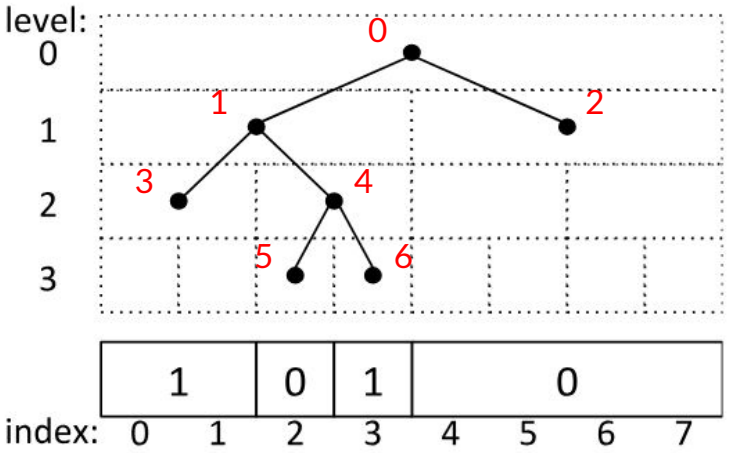
(a) initial state (b) after pruning

Figure 3: A bitmap represented as a binary tree. Initially, each leaf node is assigned a single bit (label). Sibling leaf nodes with identical labels are then pruned and the label is assigned to their parent. After pruning, the prior parent node becomes a leaf and represents multiple consecutive bits, a 0-run or a 1-run.

- ⇒ **Space consumption:** Initially, and in worst case, is $3n-1$ bits. (The tree is encoded using 1 bit per node.)

Encoding (1/4)

- TEB employs a **level-order binary marked** representation, which requires one bit per tree node.
- Differentiates between the tree data structure used during compression and the **encoded** tree that is eventually stored in a TEB.
- To encode the pruned tree structure, traverses it in **breadth-first left-to-right order** (or level-order).
- For each visited node, a single bit is emitted:
  - A 1-bit for **inner nodes**,
  - A 0-bit for **leaf nodes**.



$$T = 1100100 \qquad L = 0101$$

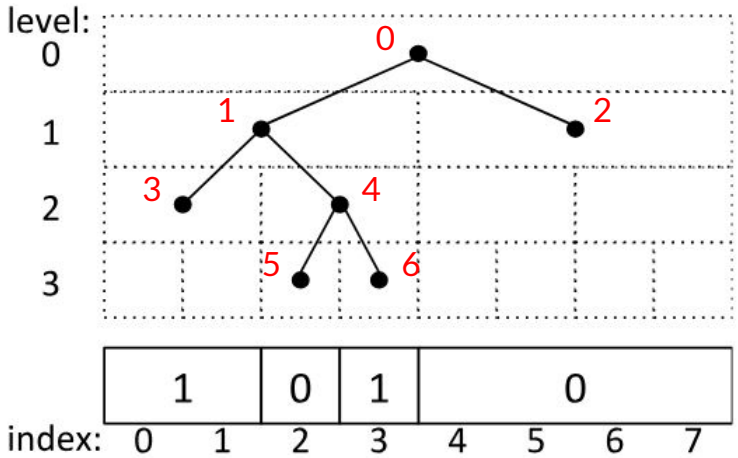
Encoding (2/4)**:** Child Nodes, Rank (Example)

Child nodes of a given tree node i:

$$\text{left-child}(i) := \text{right-child}(i) - 1$$

$$\text{right-child}(i) := 2 \cdot \text{rank}(i)$$

where $\text{rank}(i)$ is the number of 1-bits (inner nodes) in T within the range $[0, i]$.



$i = 1$

$T = 1100100$

$\text{rank}(1) = 2$

$\text{right-child}(1) = 4$

Encoding (3/4): Pre-computing the Rank

- TEB pre-computes the rank on 512-bit block granularity and stores the values in an auxiliary integer array.
- That results in a 6.25% increased memory footprint.
- The rank is computed as

$$\text{rank}(i) := R[\lfloor i/512 \rfloor] + \text{popcount}(T, \lfloor i/512 \rfloor \cdot 512, i)$$

where R refers to the array with the pre-computed values at block level and popcount counts the 1-bits in the last block up to index i.

Encoding (4/4): Label (Example)

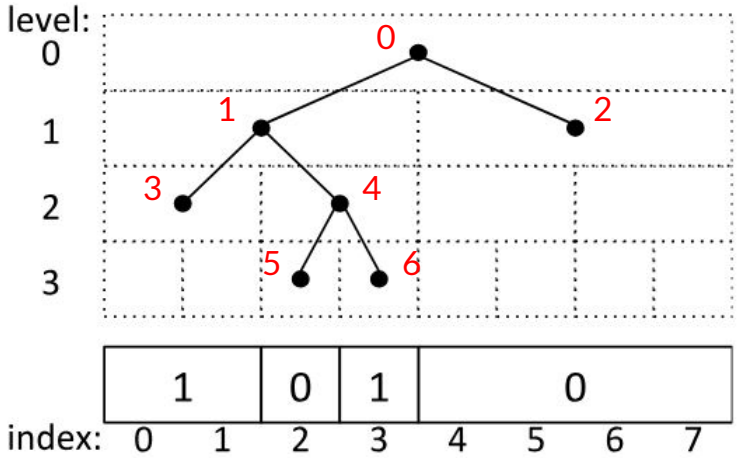
A label is accessed as follows:

$$\text{label}(i) := L[i - \text{rank}(i)]$$

$i = 5$

$T = 1100100 \quad L = 0101$

$\text{label}(5) = L[5 - 3] = L[2] = 0$



Size Comparison

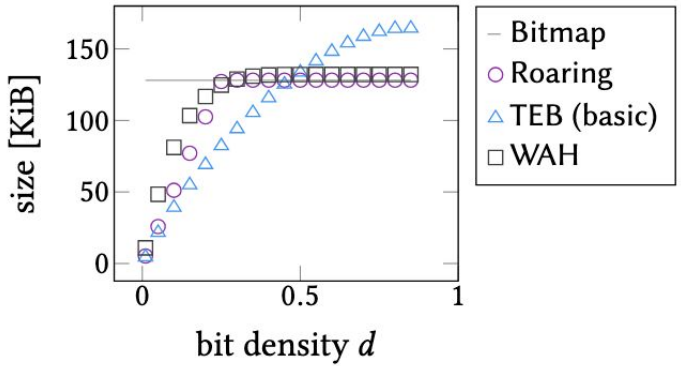


Figure 4: Size comparison for varying bit densities and a fixed clustering factor of 8.

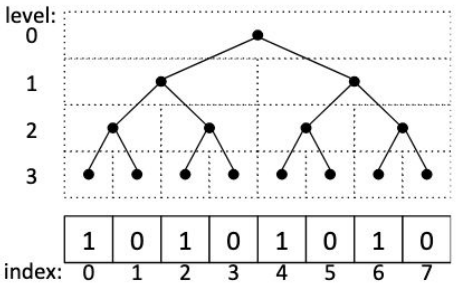


Figure 5: In worst case, the tree cannot be pruned (compressed) and the resulting TEB consumes approximately three times the space of the original bitmap.

Optimizations: Implicit Tree Nodes and Labels

1. **Implicit Tree Nodes:** Omitting leading 1-bits and trailing 0-bits of T.

The worst case of basic TEB: Only the n label bits remain:

$$T = \underbrace{1111111}_{\text{leading 1-bits}}\underbrace{00000000}_{\text{trailing 0-bits}}, \quad L = 10101010$$

2. **Implicit Labels:** Omitting leading 0-bits and trailing 0-bits of L—similar to implicit tree nodes .

Implications

1. The encoded tree structure T is an optional part of the physical TEB data structure, as the entire tree may be implicit.
2. The **space minimal TEB** instance does not necessarily contain a fully pruned tree.

(a) $3 \cdot 1.0625 + 4 = 7.1875$ bits

(b) $2 \cdot 1.0625 + 5 = 7.125$ bits

where 1.0625 is the space consumption factor of the rank helper structure.

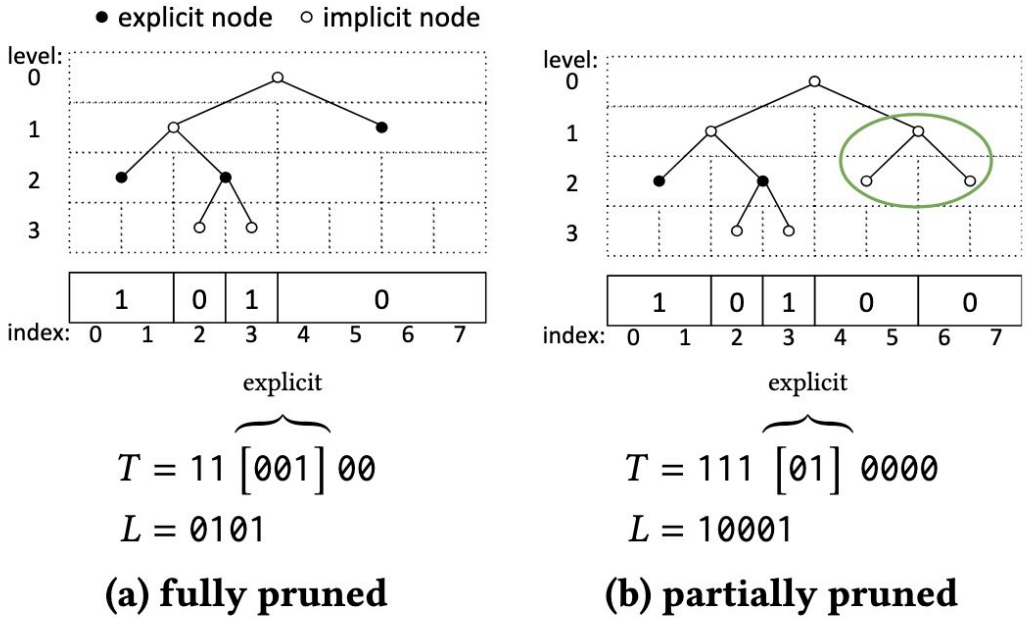


Figure 6: Two different tree representations of the bitmap 11010000. The fully pruned tree (a) occupies more space than the partially pruned tree (b), as more tree nodes need to be stored explicitly.

Basic TEB vs. Space Optimized TEB

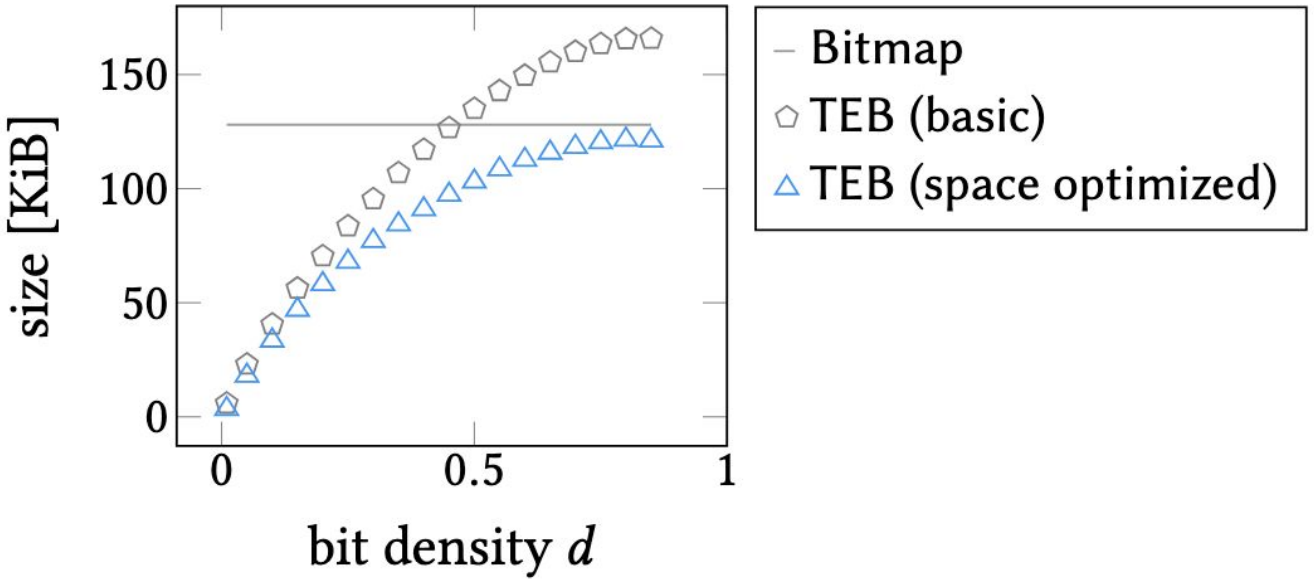


Figure 7: Size comparison of basic and space optimized TEBs using a clustering factor of 8.

Analysis (1/2)

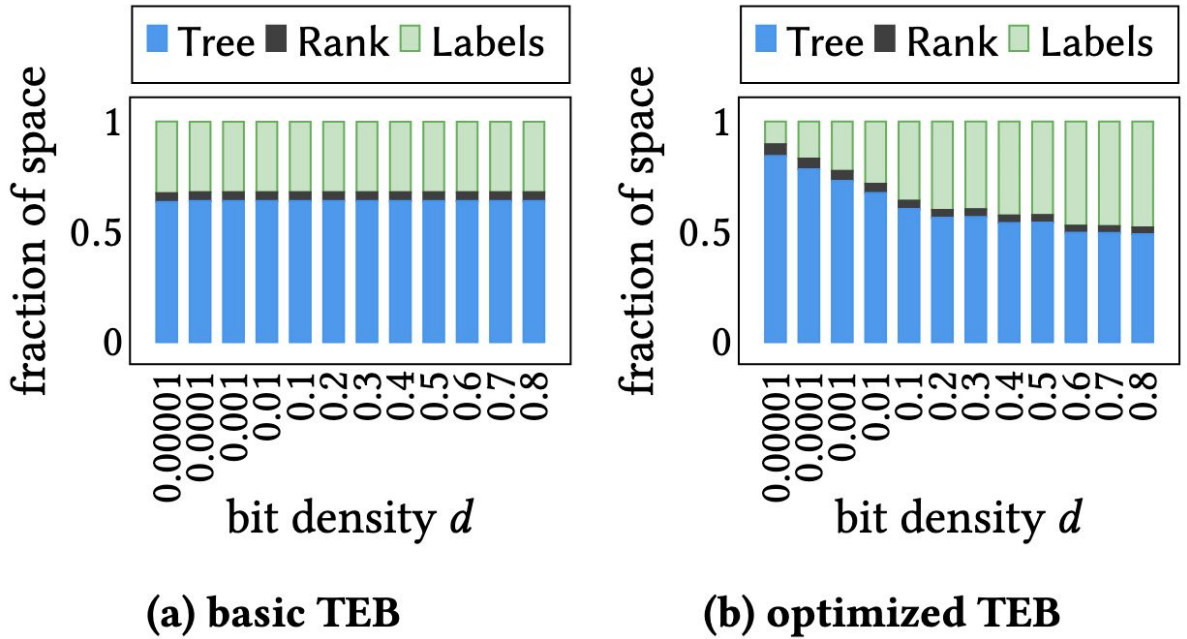


Figure 8: The fraction of space occupied by the tree, the rank helper structure, and the labels.

Analysis (2/2)

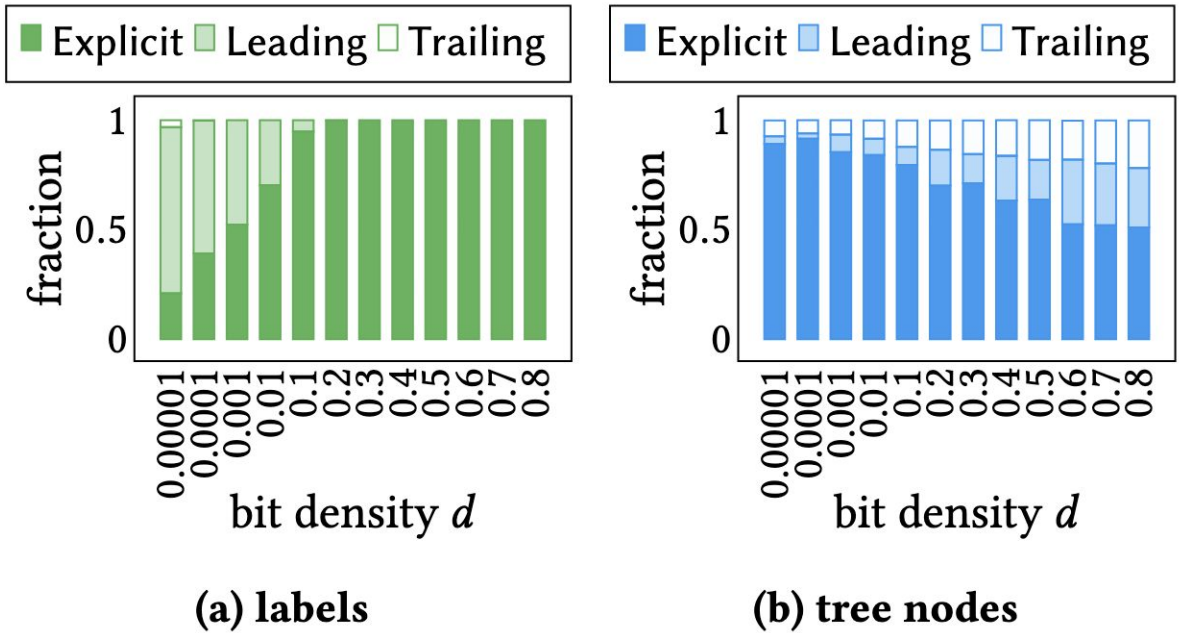
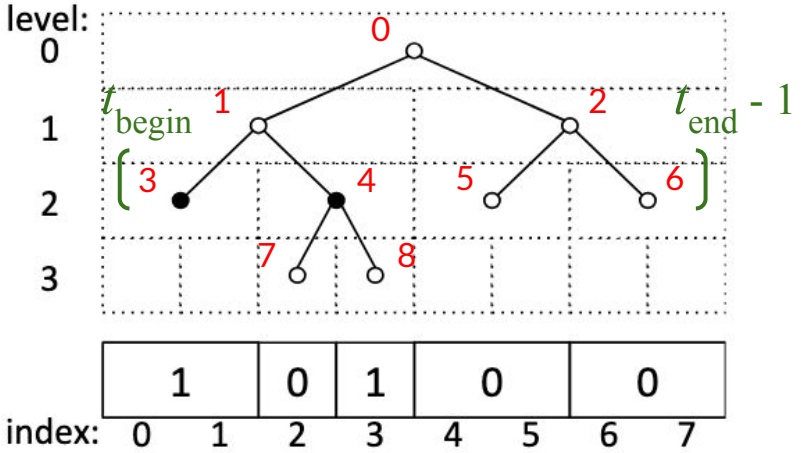


Figure 9: The fraction of explicitly stored labels (a) and tree nodes (b).

Perfect Levels



- If the number of the upper **perfect levels** is known, these levels of the tree can be logically cut off (no need to compute ranks), and only the remaining sub-trees need to be considered.
- The number of perfect levels:

$$u := \lfloor \log_2(c+1) \rfloor + 1$$

- The node IDs of the last perfect level are within the range $[t_{begin}, t_{end})$, with

$$t_{begin} := 2^{u-1} - 1$$
$$t_{end} := 2^{u} - 1$$

- Each of these nodes (the sub-trees rooted at these nodes) span a range of length $2^{\log_2(n)-u-1}$ in the original bitmap. (⇒ Uniform partitioning scheme.)

- c is the number of implicit inner nodes.

Example: $u = 3$

$$t_{begin} = 2^{3-1} - 1 = 3$$
$$t_{end} = 2^{3} - 1 = 7$$

Operations

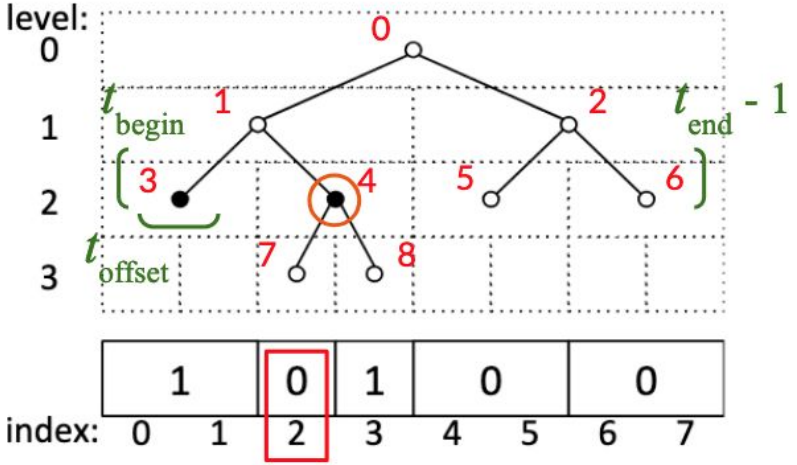
- **Point Lookup:**
  - Logarithmic time.
- **1-Run Iterator:**
  - Iterates over 1-runs.
  - Linear time (needs to visit all leaves).
  - But has **skip support**: fast-forwarding the iterator to a desired position (using point lookup)—in logarithmic time.
  - Therefore, more efficient for tree intersection.
  - Logical operations (e.g., AND iterator) are implemented on top of it.
  - Relies on the rank primitive to traverse the tree.
- **Scan Iterator:**
  - Iterates over all leaf nodes.
  - Linear time.
  - But offers higher read throughput, as it (i) decodes the tree in batches and (ii) does not rely on the rank primitive.
  - Therefore, more efficient for tree decompression.

Point Lookup

Algorithm 1: Point lookup

Input : The bit index k to test
Returns : `true` if the k^{th} bit is set, `false` otherwise

// Determine the tree node at the last perfect level.
$t_{\text{offset}} \leftarrow k >> (tree\_height - perfect\_levels - 1)$
$+ 1$
$i \leftarrow t_{\text{begin}} + t_{\text{offset}}$
$j \leftarrow tree\_height - 1 - perfect\_levels - 1$
// Navigate downwards until a leaf node is observed.
while i is an inner node **do**
    $direction \leftarrow$ extract j^{th} bit from k
    $i \leftarrow$ left-child(i) + $direction$
    $j \leftarrow j - 1$
end
return label(i)



$k = 2$

$tree\_height = 3$
$perfect\_levels = 3$
$t_{\text{begin}} = 2^{3-1} - 1 = 3$
$t_{\text{end}} = 2^3 - 1 = 7$
$[t_{\text{begin}}, t_{\text{end}})$

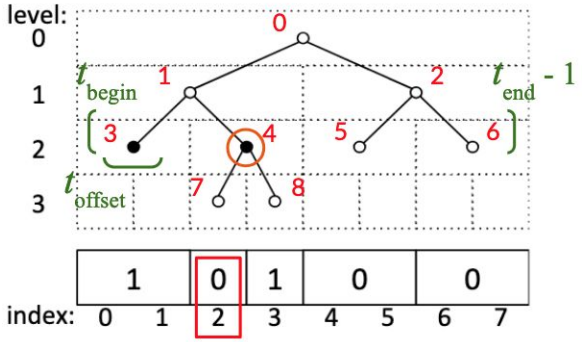
Point Lookup: Example 1

Algorithm 1: Point lookup

Input : The bit index k to test
Returns : `true` if the k^{th} bit is set, `false` otherwise

// Determine the tree node at the last perfect level.
$t_{\text{offset}} \leftarrow k >> (tree\_height - perfect\_levels - 1)$
$i \leftarrow t_{\text{begin}} + t_{\text{offset}}$ $+ 1$
$j \leftarrow tree\_height - 1 - perfect\_levels - 1$
// Navigate downwards until a leaf node is observed.
while i is an inner node **do**
  $direction \leftarrow$ extract j^{th} bit from k
  $i \leftarrow$ left-child(i) + $direction$
  $j \leftarrow j - 1$
end
return label(i)

$tree\_height = 3$
$perfect\_levels = 3$
$t_{\text{begin}} = 2^{3-1} - 1 = 3$
$t_{\text{end}} = 2^3 - 1 = 7$
$[t_{\text{begin}}, t_{\text{end}})$



$k = 2_{10} = 10_2$

$t_{\text{offset}} = 10_2 >> (3 - 3 + 1) = 1$
$i_0 = 3 + 1 = 4$
$j_0 = 3 - 3 = 0$

$i_0 = 4$ is an inner node.
$direction$ = extract(0, 10_2) = 0
i_1 = left-child(4) + 0 = 7
label(7) = 0

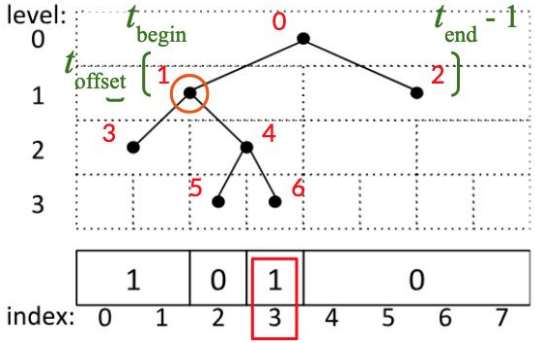
Point Lookup: Example 2

Algorithm 1: Point lookup

Input : The bit index k to test
Returns : `true` if the k^{th} bit is set, `false` otherwise

// Determine the tree node at the last perfect level.
$t_{offset} \leftarrow k >> (tree\_height - perfect\_levels - 1)$
$i \leftarrow t_{begin} + t_{offset}$ + 1
$j \leftarrow tree\_height - 1 - perfect\_levels - 1$
// Navigate downwards until a leaf node is observed.
while i is an inner node **do**
  $direction \leftarrow$ extract j^{th} bit from k
  $i \leftarrow$ left-child(i) + $direction$
  $j \leftarrow j - 1$
end
return label(i)

$tree\_height = 3$
$perfect\_levels = 2$
$t_{begin} = 2^{2-1} - 1 = 1$
$t_{end} = 2^2 - 1 = 3$
$[t_{begin}, t_{end})$



$k = 3_{10} = 11_2$

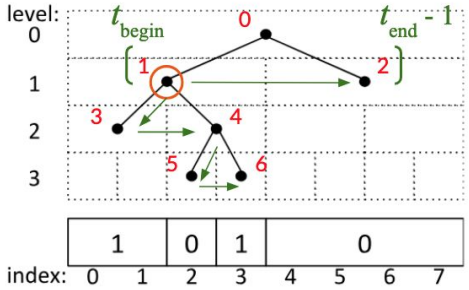
$t_{offset} = 11_2 >> (3 - 2 + 1) = 0$
$i_0 = 1 + 0 = 1$
$j_0 = 3 - 2 = 1$

$i_0 = 1$ is an inner node.
$direction_0 = \text{extract}(1, 11_2) = 1$
$i_1 = \text{left-child}(1) + 1 = 4$
$j_1 = 1 - 1 = 0$

$i_1 = 4$ is an inner node.
$direction_1 = \text{extract}(0, 11_2) = 1$
$i_1 = \text{left-child}(4) + 1 = 6$
label(6) = 1

Run Iterator

(⇒ Efficient Tree Intersection)



- A 1-run is represented as <*begin*, *end*>, pointing to the positions of the first and one past the last 1-bit.
- Maintains a **path** variable that encodes the path (as the sequence of directions 0 and 1, starting with 1) from the root to the current node.
- A **stack** populated during downward navigation is used to navigate upwards—faster than computing the counterpart to rank.
- **Perfect levels** of the tree are skipped.
- **Skip** (fast-forwarding) support.

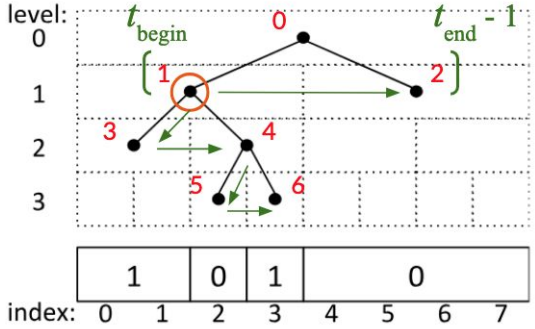
Algorithm 2: Forward the iterator to the next 1-run.

```
while t < t_end do
    while stack is not empty do
        // Pop tree node i and its path p from the stack.
        ⟨i, p⟩ ← stack.pop()
        while i is an inner node do
            // Push right child on stack and go to left child.
            i ← left-child(i)
            p ← p << 1
            stack.push( ⟨i + 1, p | 1⟩ )
        end
        // Reached a leaf node.
        if label(i) = 0 then continue
        // Found a 1-run. Update the iterator state.
        level ← sizeof(p) · 8 − 1 − lzcount(p)
        begin ← (p ⊕ (1 << level)) << (tree_height − level)
        end ← begin + (n >> level)
        return
    end
    t ← t + 1
    p ← (t − t_begin) | (1 << (perfect_levels − 1))
    stack.push( ⟨t, p⟩ )
end
begin ← end ← n    // Reached the end.
return
```

$tree\_height = 3$
$perfect\_levels = 2$

$t_0 = t_{begin} = 1$
$p_0 = 10_2$
$stack = \langle t_0, p_0 \rangle$

$\langle i_{00}, p_{00} \rangle = \langle 1, 10_2 \rangle$
$i_{000} = \text{left-child}(1) = 3$
$p_{000} = 10_2 << 1 = 100_2$
$stack = \langle 4, 101_2 \rangle$
// right-child(1)
$i_{000} = 3$ is a leaf node
$\wedge$ label(3) = 1.
$level_0 = 2$
$begin_0$
$= (100_2 \oplus (1 << 2)) << (3 - 2)$
$= 0 << 1 = 0$
$end_0 = 0 + (8 >> 2) = 1$



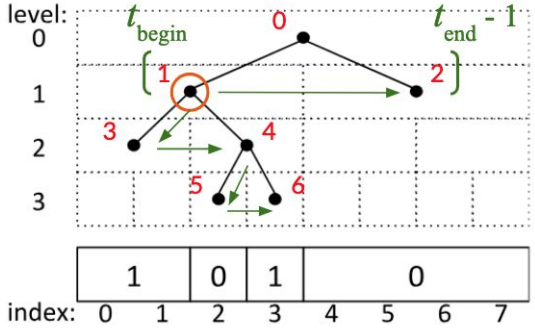
$\langle i_{01}, p_{01} \rangle = \langle 4, 101_2 \rangle$
$i_{010} = \text{left-child}(4) = 5$
$p_{010} = 101_2 << 1 = 1010_2$
$stack = \langle 6, 1011_2 \rangle$
// right-child(4)
$i_{010} = 5$ is a leaf node
$\wedge$ label(5) = 0
$\Rightarrow$ **continue**.

…

Algorithm 2: Forward the iterator to the next 1-run.

```
while t < t_end do
    while stack is not empty do
        // Pop tree node i and its path p from the stack.
        ⟨i, p⟩ ← stack.pop()
        while i is an inner node do
            // Push right child on stack and go to left child.
            i ← left-child(i)
            p ← p << 1
            stack.push( ⟨i + 1, p | 1⟩ )
        end
        // Reached a leaf node.
        if label(i) = 0 then continue
        // Found a 1-run. Update the iterator state.
        level ← sizeof(p) · 8 − 1 − lzcount(p)
        begin ← (p ⊕ (1 << level)) << (tree_height − level)
        end ← begin + (n >> level)
        return
    end
    t ← t + 1
    p ← (t − t_begin) | (1 << (perfect_levels − 1))
    stack.push( ⟨t, p⟩ )
end
begin ← end ← n    // Reached the end.
return
```

tree_height = 3
perfect_levels = 2



…

$<i_{02}, p_{02}> = <6, 1011_2>$
$i_{02} = 6$ is a leaf node
$\wedge$ label(6) = 1.
$level_1 = 3$
$begin_1$
$= (1011_2 \oplus (1 << 3)) << (3 - 3)$
$= 11_2 = 3$
$end_1 = 3 + (8 >> 3) = 3$

$t_1 = t_0 + 1 = 1 + 1 = 2$
$p_1 = (2 - 1) \mid 1 << (2 - 1)$
$= 1_2 \mid 10_2 = 11_2$
$stack = <2, 11_2>$

$<i_{10}, p_{10}> = <2, 11_2>$
$i_{01} = 2$ is a leaf node
$\wedge$ label(2) = 0
$\Rightarrow$ **continue**.
$t_1 = t_{end} \Rightarrow begin_2 = end_2 = 8$
// end

Algorithm 2: Forward the iterator to the next 1-run.

```
while t < t_end do
    while stack is not empty do
        // Pop tree node i and its path p from the stack.
        ⟨i, p⟩ ← stack.pop()
        while i is an inner node do
            // Push right child on stack and go to left child.
            i ← left-child(i)
            p ← p << 1
            stack.push( ⟨i + 1, p | 1⟩ )
        end
        // Reached a leaf node.
        if label(i) = 0 then continue
        // Found a 1-run. Update the iterator state.
        level ← sizeof(p) · 8 − 1 − lzcount(p)
        begin ← (p ⊕ (1 << level)) << (tree_height − level)
        end ← begin + (n >> level)
        return
    end
    t ← t + 1
    p ← (t − t_begin) | (1 << (perfect_levels − 1))
    stack.push( ⟨t, p⟩ )
end
begin ← end ← n    // Reached the end.
return
```

Fast-Forwarding Run Iterator (Example 1)

Check whether the destination node is within the current sub-tree:

$h = 3$

$u = 3$

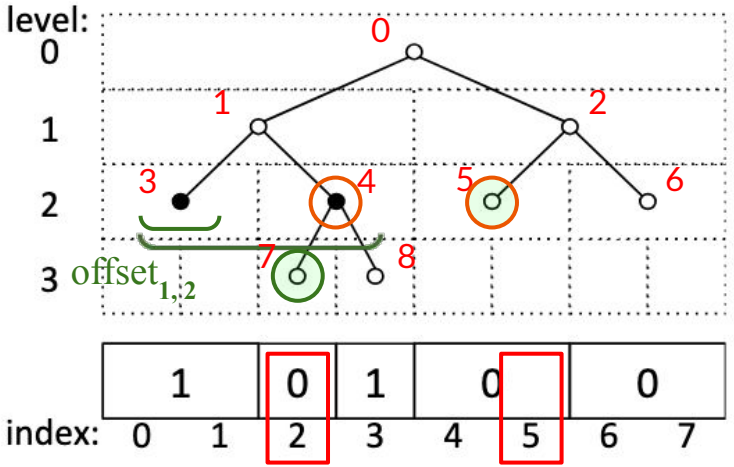
$$pos >> (h - u + 1) \; != \; to\_pos >> (h - u + 1)$$

(offset_1 and offset_2)

$$pos = 2_{10} = 10_2$$

$$to\_pos = 5_{10} = 101_2$$

$$1 \; != \; 2$$



Fast-Forwarding Run Iterator (Example 1)

Check whether the destination node is within the current sub-tree:

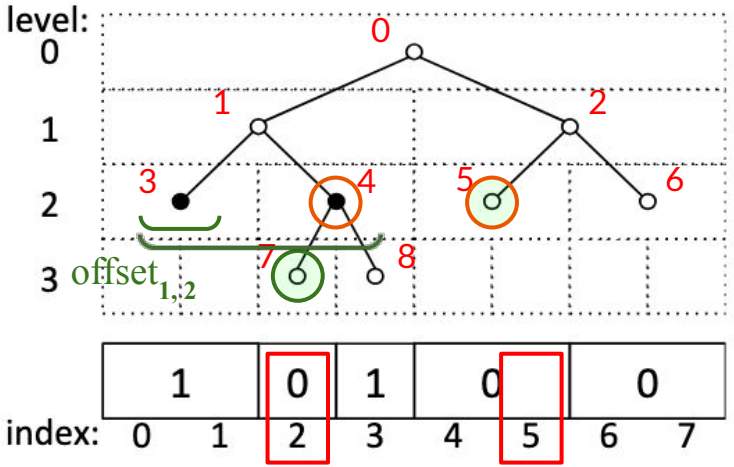
$h = 3$

$u = 3$

$$pos >> (h - u + 1) \; != \; to\_pos >> (h - u + 1)$$

(offset_1 = $pos >> (h - u + 1)$, offset_2 = $to\_pos >> (h - u + 1)$)

$$pos = 2_{10} = 10_2$$
$$to\_pos = 5_{10} = 101_2$$
$$1 \; != \; 2$$



- The destination node is outside of the current sub-tree, therefore:
  - Go to the corresponding node at the last perfect level and navigate downwards to the desired position.

Fast-Forwarding Run Iterator (Example 2)

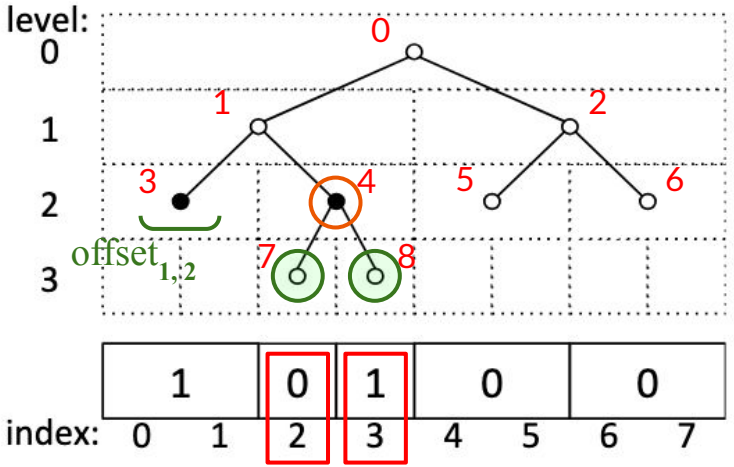
Check whether the destination node is within the current sub-tree:

$$h = 3$$
$$u = 3$$

$$pos >> (h - u + 1) \; != \; to\_pos >> (h - u + 1)$$

(offset_1 = $pos >> (h - u + 1)$, offset_2 = $to\_pos >> (h - u + 1)$)

$$pos = 2$$
$$to\_pos = 3$$
$$1 == 1$$



The rank primitive is significantly more costly than accessing the stack.

Therefore, a downward step is approximately 9 × more expensive than an upward step.

Fast-Forwarding Run Iterator (Example 2)

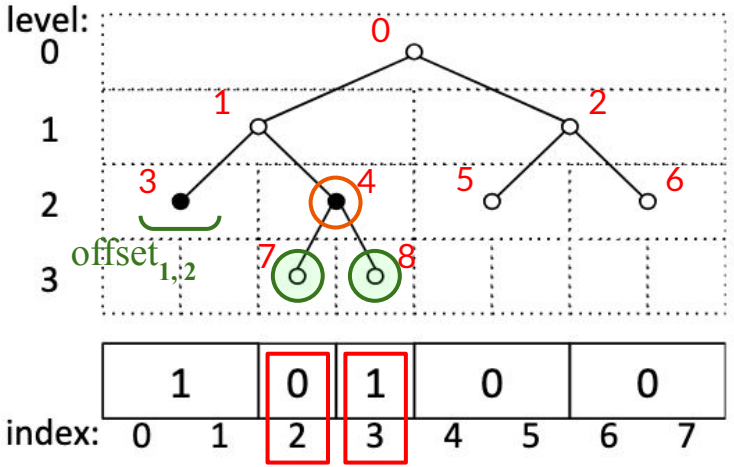
Check whether the destination node is within the current sub-tree:

$$h = 3$$
$$u = 3$$

$$pos >> (h - u + 1) \; != \; to\_pos >> (h - u + 1)$$

(offset_1 = $pos >> (h - u + 1)$, offset_2 = $to\_pos >> (h - u + 1)$)

$$pos = 2$$
$$to\_pos = 3$$
$$1 == 1$$



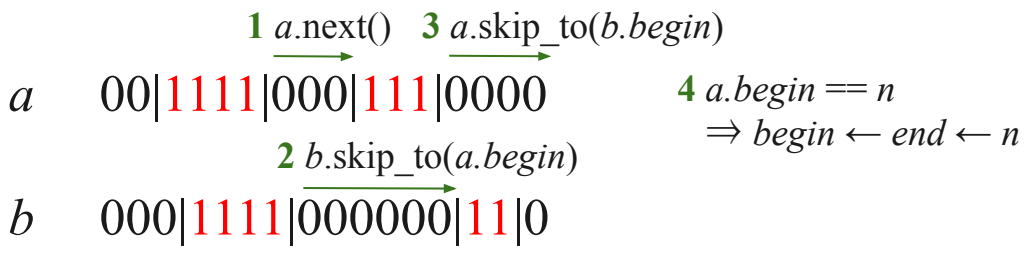
The rank primitive is significantly more costly than accessing the stack.

Therefore, a downward step is approximately 9 × more expensive than an upward step.

- The destination node is within the current sub-tree, therefore:
  - Determine the common ancestor node;
  - Estimate the navigational costs for navigating upwards and downwards;
  - Pick the cheaper path.

Logical Operations:

AND Iterator (Example)

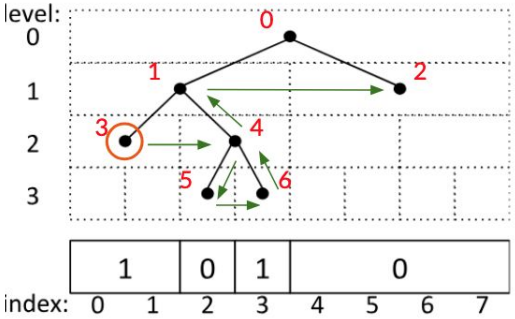


Algorithm 4: Next function of the AND iterator.

```
Input: Run iterators a and b.
while !(a.begin != n || b.begin != n) do
    begin_max ← max(a.begin, b.begin)
    end_min ← min(a.end, b.end)
    overlap ← begin_max < end_min
    if overlap then
        if a.end ≤ b.end then a.next()
        if b.end ≤ a.end then b.next()
        begin ← begin_max   // Update the iterator state.
        end ← end_min
        return
    else
        if a.end ≤ b.end then a.skip_to(b.begin)
        else b.skip_to(a.begin)
    end
end
begin ← end ← n   // Reached the end.
```

Tree Scan

(Efficient Tree Decompression)

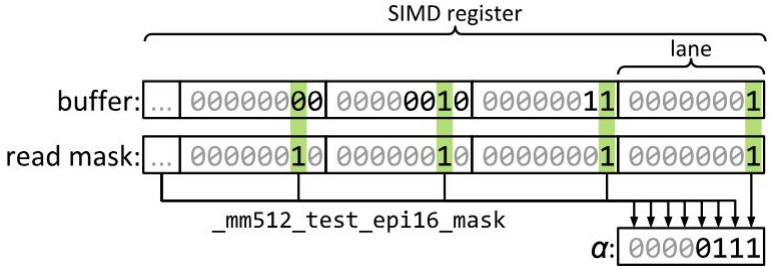


Algorithm 3: Tree scan

```
p // The current path. Initially points to the leftmost leaf.
do
    // Produce an output, if the label of the current node is 1.
    ...
    // Walk upwards until a left child is found.
    up_steps ← tzcount(~p)
    last ← level(p) + 1
    p ← p >> up_steps
    p ← p | 1    // Go to the right sibling.
    first ← level(p)
    increment the iterators t_first to t_last and update α
    // Walk downwards to the leftmost leaf in that sub-tree.
    down_steps ← tzcount(~(α >> level(p)))
    p ← p << down_steps
while not done
```

- The key idea:
  - to have **multiple lightweight bit iterators** for the encoded structure T, one per tree level: t_l with $0 \le l < h$;
  - scan the bit sequence T in parallel.
- The bits in $\alpha := b_{h-1} \ldots b_1 b_0$ are populated with $b_l = *t_l$, where $*$ denotes the dereference operator.
- A path variable p initially points to the leftmost leaf node.
- Efficient iteration over all leaf nodes in left-to-right order using α and p.

- Bit iterators implementation using the AVX-512 SIMD instruction set (the picture) ...



Tree Scan: Implementation

- The bit iterators are implemented using the AVX-512 SIMD instruction set.
  - A 512-bit SIMD register is used to buffer the tree structure.
  - The register is interpreted as **32 × 16-bit** integers, i.e., the register is split into **32 lanes**.
  - Thereby, each SIMD lane corresponds to a **tree level**—up to 16 bits of the level are loaded up in the buffer.
  - A second SIMD register used as **read mask** represents the state of all (up to) 32 lightweight bit iterators—a **single bit** is set within **each lane**, and the position of that bit represents the **current read position** in the corresponding buffer lane.
- **Offsets** of each tree level are stored in the TEB metadata.

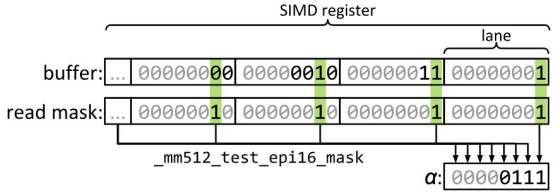
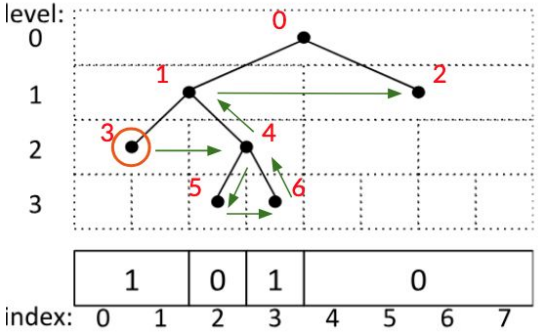


Figure 10: AVX-512 allows for the instantiation of up to 32 lightweight bit iterators (one for each tree level) using only two SIMD registers: The first is used to buffer the encoded tree level by level and the second represents the iterators' read positions.

$T = 1 \mid 10 \mid 01 \mid 00$

$p_0 = 100_2$
// NOT p_0
$\sim p_0 = 011_2$ // →
$p_1 = 1010_2$
$\text{level}(p_1) = 2$
$T = 1 \mid 10 \mid 0\underline{1} \mid 00$
$\alpha_1 = 0111_2$
$\sim(\alpha_1 >> 2) = 10_2$ // ↓



Algorithm 3: Tree scan

```
p // The current path. Initially points to the leftmost leaf.
do
    // Produce an output, if the label of the current node is 1.
    ...
    // Walk upwards until a left child is found.
    up_steps ← tzcount(~p)
    last ← level(p) + 1
    p ← p >> up_steps
    p ← p | 1    // Go to the right sibling.
    first ← level(p)
    increment the iterators t_first to t_last and update α
    // Walk downwards to the leftmost leaf in that sub-tree.
    down_steps ← tzcount(~(α >> level(p)))
    p ← p << down_steps
while not done
```

$p_1 = 1010_2$
$\sim p_1 = 0101_2$ // →
$p_2 = 1011_2$
$\text{level}(p_2) = 3$
$T = 1 \mid 10 \mid 01 \mid 0\underline{0}$
$\alpha_2 = 0111_2$
$\sim(\alpha_2 >> 3) = 1_2$

$p_2 = 1011_2$
$\sim p_2 = 0100_2$ // ↑→
$p_3 = 11_2$
$\text{level}(p_3) = 1$
$T = 1 \mid 1\underline{0} \mid 0\underline{1} \mid 0\underline{0}$
$\alpha_3 = 0101_2$
$\sim(\alpha_3 >> 3) = 1_2$

Experimental Analysis

- Real-world data.
- Synthetic data:
  - **Uniform** random bitmaps.
  - **Clustered** random bitmaps: generated using a two-state Markov process.

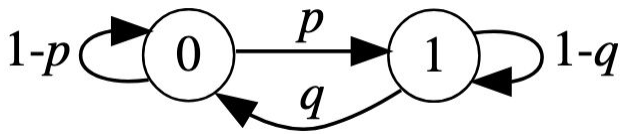
Real-World Data

	WAH	EWAH	Concise	Roaring	**TEB**
Census Income	3.4	3.3	2.9	2.6	**2.1**
Census Income (sorted)	0.66	0.64	0.55	0.6	**0.36**
Census 1881	34.4	33.8	25.6	15.1	**12.6**
Census 1881 (sorted)	3.0	2.9	2.5	2.1	**1.5**
Weather	6.8	6.7	5.9	5.4	**4.2**
Weather (sorted)	0.55	0.54	0.43	0.34	**0.26**
WikiLeaks	11.1	10.9	10.2	5.9	**5.4**
WikiLeaks (sorted)	2.9	2.7	2.2	**1.7**	1.7

Table 1: Space usage in bits per attribute value.

Synthetic Data

- **Uniform** random bitmaps are random bitmaps where each bit is set with probability d.
- **Clustered** random bitmaps are generated using a two-state Markov process



with the transition probabilities p and q set to

$$p := \frac{d}{(1-d)\cdot f}, \text{ and } q := \frac{1}{f}$$

with $0 < d < 1$ and $1 \leq f \leq n$.
(The restrictions as before apply: $\max(1, d/(1-d)) \leq f \leq d \cdot n$.)

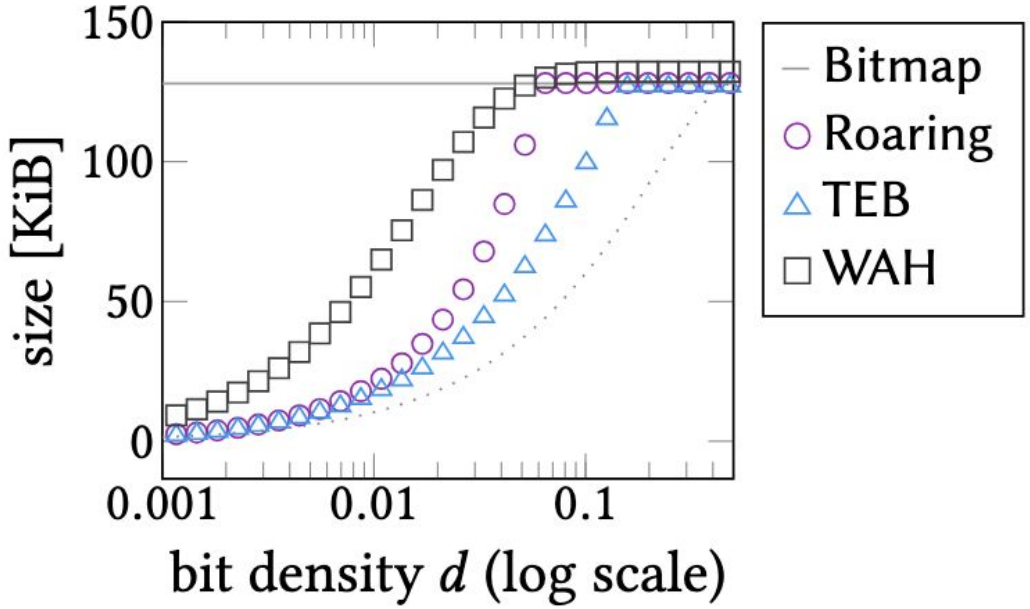
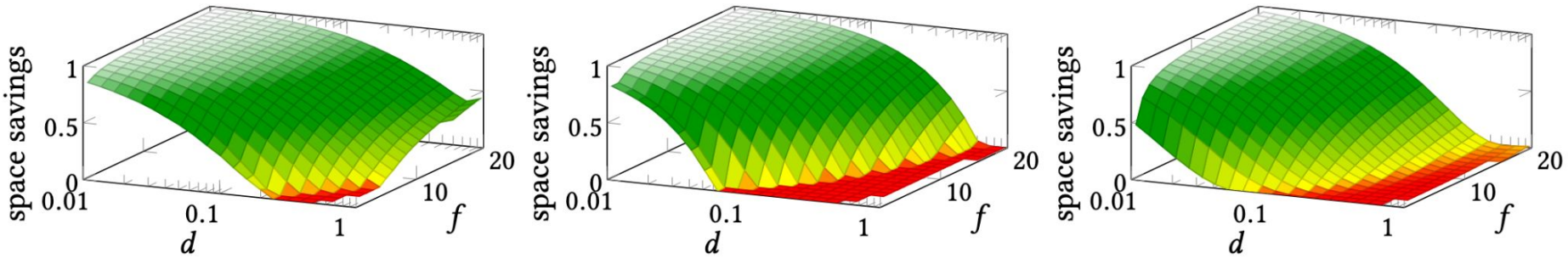


Figure 11: Size of uniform random bitmaps with varying bit densities. The dotted line refers to the information theoretic minimum.



(a) TEB (b) Roaring (c) WAH

Figure 13: Space savings $\left(1 - \frac{\text{compressed size}}{\text{uncompressed size}}\right)$ for varying d and f.

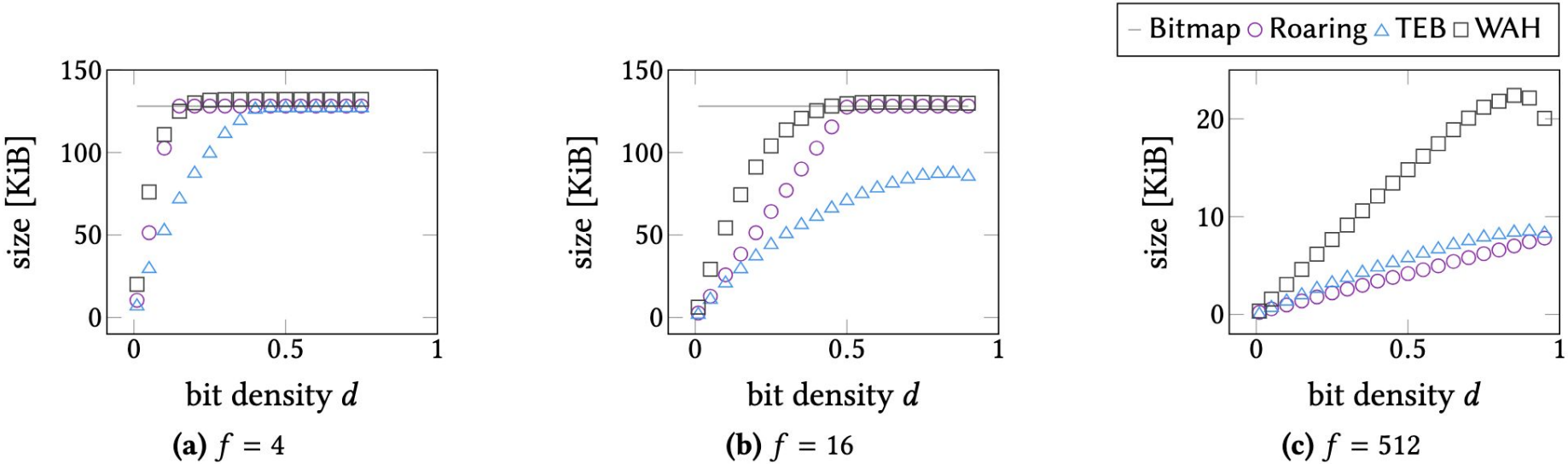


Figure 14: Compressed bitmap size for varying bit densities and fixed clustering factors.

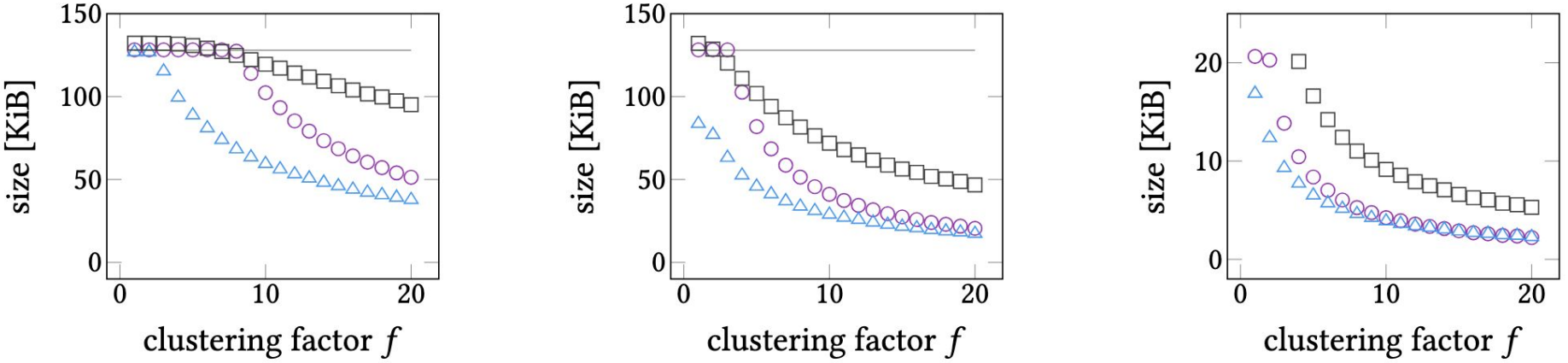


Figure 15: Compressed bitmap size for varying clustering factors and fixed bit densities.

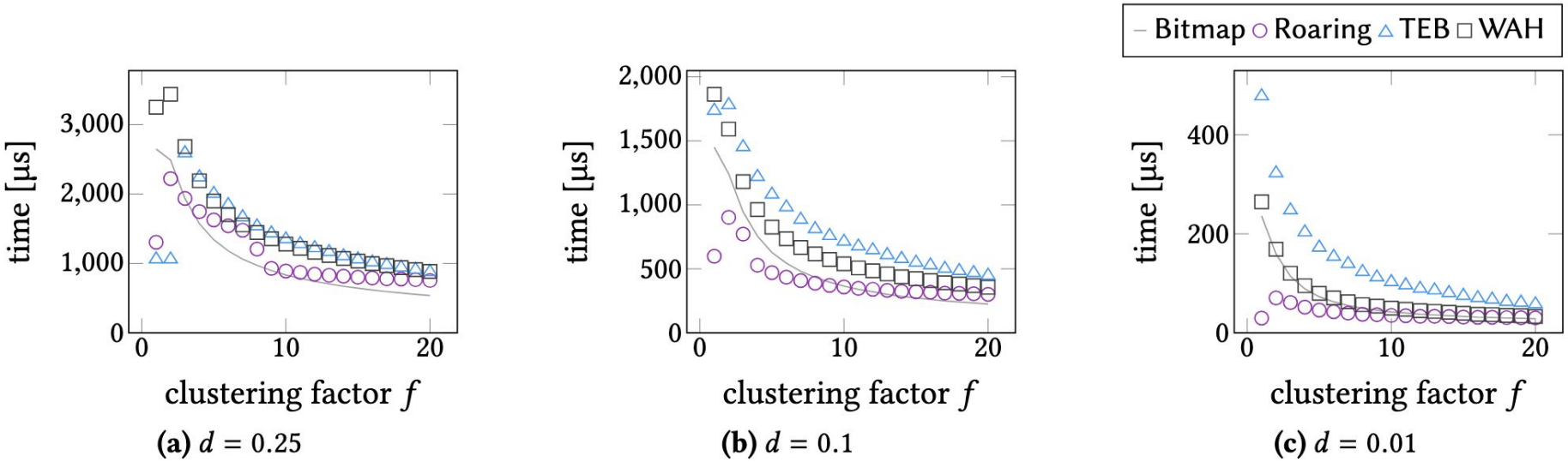
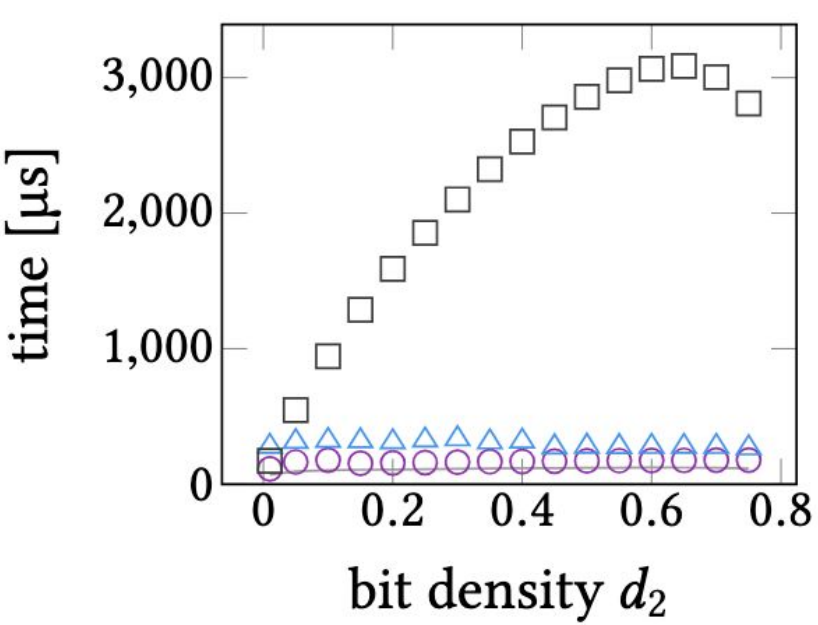
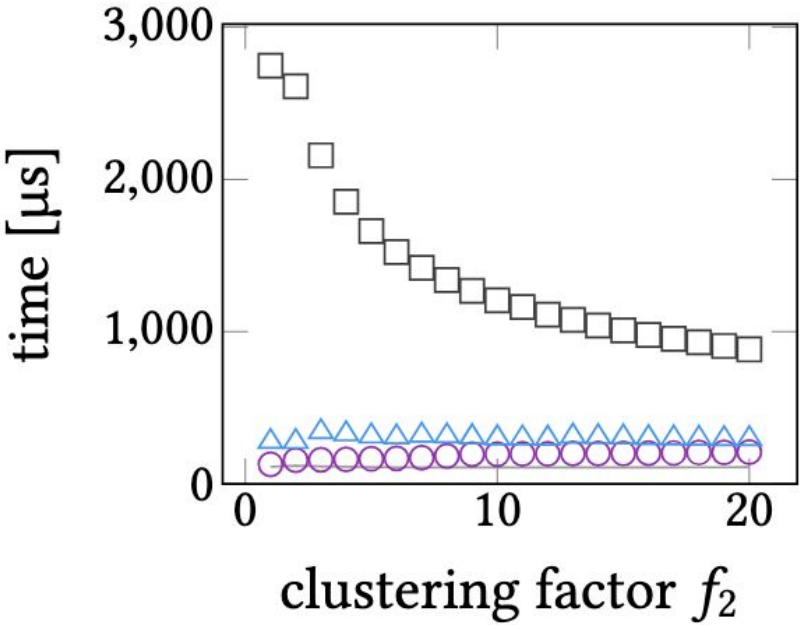


Figure 16: Read performance for varying clustering factors and fixed bit densities.



(a) $d_1 = 0.01, f_1 = 8, f_2 = 4$



(b) $d_1 = 0.01, f_1 = 8, d_2 = 0.25$

Figure 17: Intersection performance.

Differential Updates

(using Roaring as a differential data structure)

Compression method	avg. time per update [ns]	
	non-partitioned	partitioned
TEB	599	218
Roaring	480* / 574	121* / 216
WAH	17634	794

* using the in-memory layout (non serialized)

Table 3: The average time to apply an update.