

Prune Early, Intersect Fast: PRISM, a Prefix-based Radix Integer Set with Morphing Nodes

Adrian Riedl
TUM
Munich, Germany
adrian.riedl@in.tum.de

Thomas Neumann
TUM
Munich, Germany
neumann@in.tum.de

ABSTRACT

Intersecting sets of integers is a fundamental operation in search engines and graph analytics. Yet most approaches treat integers as opaque values and compare them one by one, ignoring that every integer set partitions the key space into occupied and empty ranges. PRISM exploits this: it decomposes 64-bit integers into fixed-width prefix segments organized in a trie, where each node uses a bitmap to record which sub-ranges are occupied. Ranges absent from at least one operand are pruned without examining individual elements, fixed-width leaf segments map directly onto SIMD registers, and the structure extends naturally to k -way intersection via simultaneous descent. Morphing-node layouts adjust each node's size by occupancy, reducing memory usage by up to $8\times$ on sparse data while minimally affecting throughput on dense data. We also evaluate PRISM on three real-world case studies: the Star Schema Benchmark, a TPC-H trigram index, and inverted list intersection on ClueWeb09. On all three, PRISM is the fastest method for intersection-heavy workloads, ahead of compressed-bitmap methods and more than an order of magnitude faster than general-purpose ordered and hash-based containers.

PVLDB Reference Format:

Adrian Riedl and Thomas Neumann. Prune Early, Intersect Fast: PRISM, a Prefix-based Radix Integer Set with Morphing Nodes. PVLDB, 20(1): 53 - 65, 2026.

doi:10.14778/3845598.3845603

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://gitlab.db.in.tum.de/AdrianRiedl/intersection-vldb27>.

1 INTRODUCTION

Set intersection is a common primitive in information retrieval and graph processing: search engines intersect posting lists to resolve queries, and graph analytics compute common neighbors for triangle counting and link prediction.

Given its importance, researchers have studied set intersection extensively. Classical approaches, e.g. merge-based intersection of sorted arrays [3–6, 12, 16] and hash-based probing [13], remain widely deployed for their predictable performance. More recently,

Prefix-based pruning for set intersection

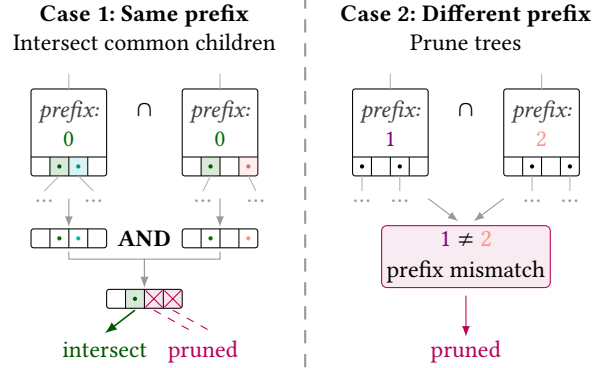


Figure 1: PRISM organizes integer sets in a trie over fixed-width prefix segments, where each node stores a prefix and a bitmap recording which child subtrees are present. Intersection traverses both tries top-down: where two nodes share a prefix, it descends into their common children; where the prefixes differ, it prunes the entire subtree without inspecting its elements.

compressed bitmap representations such as EWAH [21] and ROARING [9, 22] have been proposed, offering favorable compression ratios and efficient bitwise operations. However, these structures incur overhead from run-length decoding or container-type dispatch during intersection, and their performance degrades on skewed workloads that frequently arise in practice.

These approaches treat each element as an opaque key, so their representations reveal nothing about which regions of the key space a set occupies. Yet every set partitions the key space the same way, regardless of how identifiers are assigned. Some ranges hold elements and others hold none. A representation that records occupancy per prefix-defined subtree can therefore decide a whole subtree at once. Intersection prunes a subtree that is absent from any operand, and union, difference, and symmetric difference copy or discard a subtree that belongs to one operand alone, without descending into it. Sorted arrays compare across disjoint ranges instead. Hash sets discard the structure that would reveal those ranges, and compressed bitmaps often traverse containers that contribute nothing to the result. All three revisit individual elements that a coarser, subtree-level view would let them skip. We therefore propose a structure that records which prefix-defined subtrees of an integer set are present and which are absent. Set operations then become a coordinated traversal that resolves whole subtrees before examining individual elements.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 20, No. 1 ISSN 2150-8097. doi:10.14778/3845598.3845603

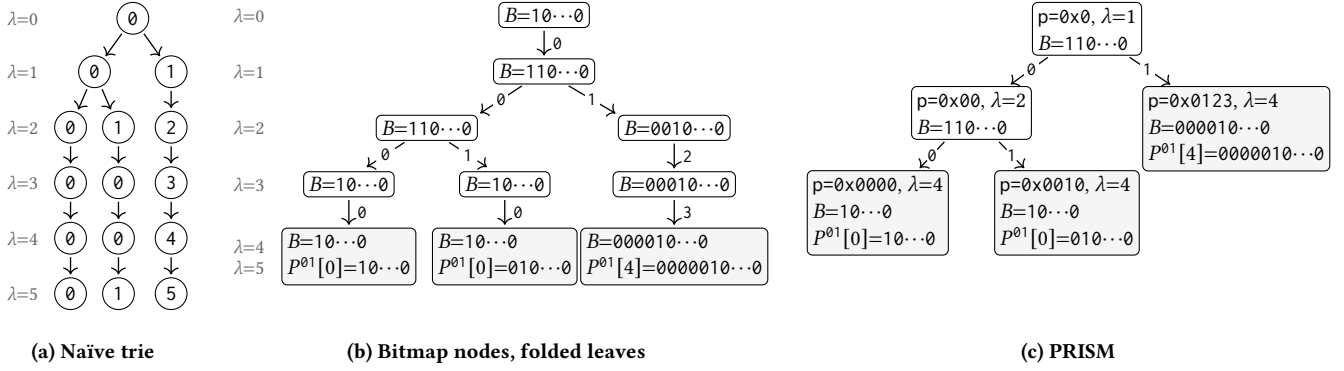


Figure 2: From a naïve trie to PRISM on three 24-bit keys (0x000000, 0x001001, 0x012345; $b=4$, $\ell=6$). (a) One node per segment. (b) Each node stores a presence bitmap B instead of a child list, and the last two segments fold into a single leaf (B for $\lambda=4$, P^{01} for $\lambda=5$). (c) Prefix compression collapses one-child chains into one node tagged with prefix p and divergence level λ .

In this paper, we present PRISM, an integer set structure that realizes this idea. PRISM decomposes 64-bit integers into fixed-width segments organized in a trie whose internal nodes encode child presence via bitmaps. Intersection then reduces to a synchronized top-down traversal in which a single bitwise AND of two bitmaps at each level identifies the shared children and prunes all disjoint branches from further consideration. Figure 1 illustrates the two characteristic cases: when two nodes share a prefix, intersection descends into their common children and prunes the rest; when the prefixes differ, the entire branch is pruned, yielding an empty result. Because each node covers a contiguous subset of the key space, a single prefix mismatch at any level suffices to skip the entire subtree beneath it. At the leaves, the fixed-width layout of the payload bitmaps aligns with SIMD register widths, so the final segment of the intersection executes as a block of vectorized bitwise operations. The same traversal pattern extends to k inputs by replacing the pairwise AND with a k -ary conjunction, identifying common children in one pass and avoiding the intermediate materialization inherent in repeated binary intersection. To accommodate workloads with highly variable density, PRISM further supports morphing-node layouts that size each node to its actual occupancy, reducing memory consumption on sparse inputs with minor impact on dense throughput. Beyond intersection, the same trie structure supports union and other set operations through analogous top-down traversals.

We make the following contributions:

- (1) We introduce PRISM, a trie-based integer set that encodes per-node child occupancy as bitmaps and uses adaptive prefix compression to reduce set operations to synchronized bitwise traversals (Section 2).
- (2) We present a unified set-operation framework for PRISM, including a k -way intersection with no intermediate materialization that prunes entire subtrees by exploiting the trie’s hierarchical structure (Section 3).
- (3) We reuse the bitmap in our set representation as a uniform node interface, so that a single set of SIMD routines can operate on all node sizes without the traversal logic dispatching on node type. Node sizes follow occupancy, as in

ART [19], and memory consumption drops by up to 8× on sparse data (Sections 4 and 5.2.1).

- (4) We evaluate PRISM on synthetic data against strong baselines from every major algorithmic category (merge-based, hierarchical, hash-based, and compressed bitmaps), showing it is competitive across distributions and overlap coefficients and the best performer in many cases (Section 5).
- (5) We study three real-world cases: query filters on the Star Schema Benchmark, a TPC-H trigram index, and an inverted index on ClueWeb09, where PRISM is the fastest method on the intersection-heavy workloads (Section 6).

2 A PREFIX-BASED RADIX INTEGER SET

Traditional set implementations such as hash sets, balanced trees, and sorted arrays compare elements pairwise, requiring work proportional to the cardinality of the operands regardless of value distribution. By treating integers as opaque keys, these approaches fail to exploit the fact that subsets of values may share or differ in entire prefixes.

PRISM addresses this limitation by decomposing each integer into fixed-width segments and inserting it into a prefix trie whose nodes encode child occupancy as bitmaps. Each node represents a contiguous range of values that share a common prefix, reducing set operations to synchronized traversals with bitwise operations at every level. Intersection prunes entire subtrees when bitmaps indicate disjoint ranges, while union, difference, and symmetric difference copy subtrees exclusive to one operand without descending into them. Concretely, PRISM combines the following four principles:

1. Bit-Level Partitioning. Each n -bit integer is split into $\ell = \lceil n/b \rceil$ segments of b bits, consumed from most to least significant along any root-to-leaf path. If n is not a multiple of b , the root segment ($\lambda = 0$) carries the remaining $n \bmod b$ bits; all other segments are exactly b bits wide. Figure 2a shows a naïve trie storing three 24-bit values across six 4-bit segments.

2. Bitmap Encoding. To locate occupied children in constant time and reduce set operations to bitwise primitives, PRISM maintains a presence bitmap B of 2^b bits per node, where $B[i] = 1$ if and only if the subtree contains a value whose segment at the current level

equals i . A naïve trie node, in contrast, either lists the occupied segment values explicitly or marks each child slot individually, both of which require a scan to locate occupied children. In Figure 2b, $B=10 \dots 0$ at the root indicates that the segment value 0 is occupied.

3. Leaf Folding. To eliminate a wasteful pointer indirection at the deepest level, PRISM folds the final level into its parent: each occupied slot at level $\lambda = \ell - 2$ stores a 2^b -bit bitmap encoding the present values of segment $\ell - 1$ in place of a child pointer. A naïve trie, in contrast, encodes the final segment values in dedicated leaf nodes at level $\lambda = \ell - 1$ (cf. Figure 2a); a direct bitmap encoding would replace these with 2^b -bit bitmaps reached through pointers from level $\lambda = \ell - 2$, paying one pointer dereference for a payload no larger than the pointer itself. For example, the rightmost leaf in Figure 2b encodes the value $0x012345$ via $B[4] = 1$ at $\lambda = 4$ together with bit 5 set in the bitmap stored at $P^{01}[4]$.

4. Adaptive Prefix Compression. To avoid wasting space on chains of one-child nodes, PRISM allows nodes to span multiple segments by adding a prefix p and a level λ to each node: p records the bit pattern shared by all values in the subtree, and λ identifies the first segment at which those values diverge. An insertion that introduces a new divergence splits the path by creating an internal node at the divergence point. This preserves the invariant that every internal node has at least two occupied children, bounding both tree depth and memory footprint. A fixed-depth trie of $\ell - 1$ levels, in contrast, wastes space whenever an internal node has a single occupied child, since the corresponding path degenerates into a chain of one-child nodes. As shown in Figure 2c, the first two nodes of the naïve trie collapse into a single node with prefix $0x0$ and level $\lambda = 1$, since all values share the top segment. Its right child further collapses to a single node with prefix $0x0123$ at $\lambda = 4$, because the third value has no sibling that shares any subsequent segment.

Node Representation. Each node v represents a subset of the stored set through four components:

Prefix p is the longest segment-aligned bit pattern shared by all values in v 's subtree.

Level $\lambda \in \{0, \dots, \ell - 2\}$ is the segment at which the values in the subset begin to diverge; equivalently, p spans the first λ segments.

Presence bitmap B is a 2^b -bit vector recording occupancy at level λ , with $B[i] = 1$ if and only if segment value i is occupied.

Payload P holds up to 2^b entries aligned with B , defined whenever $B[i] = 1$. The form of $P[i]$ depends on the node type:

Internal ($\lambda < \ell - 2$): $P[i] = P^\rightarrow[i]$ is a pointer to the child responsible for segment value i .

Leaf ($\lambda = \ell - 2$): $P[i] = P^{01}[i]$ is a 2^b -bit bitmap encoding the values of the final segment $\ell - 1$ present in the subtree.

Space Complexity. For a set S of cardinality N , the space is $O(N)$: adaptive compression keeps at least two occupied children per internal node, so the trie holds fewer than N internal nodes, each of constant size. Without it, the bound would be $O(N \cdot \ell)$, since a value could require one node per level. The actual space falls below this bound by an amount that depends on how the values are distributed. Clustered values share long prefixes, so multiple levels collapse into single nodes and both tree depth and node count drop. Uniformly distributed values give a balanced tree in which each path spans multiple levels, so degenerate linear chains do not occur.

In dense regions, leaf bitmaps encode up to 2^b values in 2^b bits, which is $O(1)$ space per value.

Parameter Selection. Our examples use $b = 4$ for clarity, but the implementation uses $b = 6$ for hardware efficiency. With $b = 6$, both the presence bitmap B and the payload array P contain $2^6 = 64$ entries. The presence bitmap fits exactly in a single 64-bit word, so bitwise operations and hardware instructions such as population count (POPCNT) and bit scanning (BSF/BSR) apply directly. Both payload arrays occupy 64 entries, so they have identical memory layouts and cache line footprints. Internal nodes store 64 child pointers, and leaf nodes store 64 bitmaps in the same space, making transitions between the two node representations uniform. AVX-512 SIMD instructions process eight 64-bit values at once, so leaf-level bitmaps can be operated on in vectorized form. The structure parameters therefore match the hardware: 64-bit words, 64-byte cache lines, and SIMD register widths. For 64-bit integers, $b = 6$ gives $\ell = \lceil 64/6 \rceil = 11$ segments, so a root-to-leaf path visits at most $\ell - 1 = 10$ nodes.

3 SET OPERATIONS

PRISM encodes subset information hierarchically, and set operations use that structure to decide which subtrees can contribute to the result. Intersection directly discards subtrees with no shared prefix. Union, difference, and symmetric difference handle a subtree that belongs to only one operand without inspecting its individual elements. We traverse the input tries synchronously, identify common prefixes and shared subtrees at each level, and prune large portions of the search space without examining elements directly. This gives substantial speedups over generic set-operation algorithms, which have no such structural information.

3.1 k -Way Set Intersection

Intersecting multiple sets arises naturally in database workloads, for instance when evaluating conjunctive filter predicates or probing trigram indexes where each trigram produces a candidate set. A naïve approach applies binary intersection repeatedly, $((S_1 \cap S_2) \cap S_3) \cap \dots$, materializing $O(k)$ intermediate structures and requiring $O(k)$ separate tree traversals. Worse, one can construct cases in which any ordering of binary intersection yields non-empty intermediate results even when the final intersection is empty [28]. The bitmap-based node representation of PRISM avoids both problems by allowing intersection to proceed recursively from the root over all k inputs simultaneously: at each level, a single k -ary bitwise conjunction across all node bitmaps identifies exactly the children shared by every operand, restricting subsequent traversal to common subtrees and pruning every branch absent from any operand without further inspection. Recursion terminates at the leaf level, and result nodes are assembled bottom-up, yielding the output set without intermediate materialization (Listing 1).

3.1.1 Algorithm Description. Given k input sets $\{S_1, \dots, S_k\}$, described by their roots $\{v_1, \dots, v_k\}$, PRISM intersects them by traversing all k structures jointly, combining prefix-based pruning to skip entire subtrees irrelevant to the result, level-misalignment handling, and level-wise intersection (cf. Listing 1).

❶ **Prefix-Based Pruning.** Since each node covers only the subset of values sharing a common prefix, nodes whose prefixes

```

1 function INTERSECT( $v_1, v_2, \dots, v_k$ )
2   if  $\exists i: v_i = \text{null}$  // Return  $\emptyset$  if any input is empty
3     return null
4    $M \leftarrow \text{prefixMask}(v_1.\lambda) \wedge \dots \wedge \text{prefixMask}(v_k.\lambda)$  ❶
5    $p \leftarrow v_1.p \wedge M$ 
6   for each  $i$  in  $2 \dots k$  do // Prefix-Based Pruning
7     if  $(v_i.p \wedge M) \neq p$ 
8       return null
9   if  $\exists i, j: v_i.\lambda \neq v_j.\lambda$  // Level Misalignment
10     $v_D \leftarrow \arg \max_{v_i} v_i.\lambda$  ❷
11    for each  $v_s$  with  $v_s.\lambda < v_D.\lambda$  do
12       $v_s \leftarrow \text{descendInto}(v_s, v_D)$ 
13      if  $v_s = \text{null}$ 
14        return null
15    return INTERSECT( $v_1, \dots, v_k$ )
16   else // Level-wise Intersection
17      $B_{\text{common}} \leftarrow v_1.B \wedge \dots \wedge v_k.B$  ❸
18     if  $B_{\text{common}} = 0$ 
19       return null
20      $N \leftarrow \text{createNode}(v_1.\lambda, p)$ 
21     if isLeaf( $v_1$ )
22       for each set bit  $i$  in  $B_{\text{common}}$  do ❸.1
23          $N.P^{01}[i] \leftarrow v_1.P^{01}[i] \wedge \dots \wedge v_k.P^{01}[i]$ 
24          $N.B[i] \leftarrow N.P^{01}[i] \neq 0$ 
25     else
26       for each set bit  $i$  in  $B_{\text{common}}$  do ❸.2
27          $N.P^{-}[i] \leftarrow \text{INTERSECT}(v_1.P^{-}[i], \dots, v_k.P^{-}[i])$ 
28          $N.B[i] \leftarrow N.P^{-}[i] \neq \text{null}$ 
29   return  $N.B = 0 ? \text{null} : \text{collapseIfSingleton}(N)$ 

```

Listing 1: k -way intersection algorithm returning another set

differ must root disjoint subtrees and can be pruned immediately. To test compatibility across all k operands, we compute the combined alignment mask

$$M \leftarrow \bigwedge_{i=1}^k \text{prefixMask}(v_i.\lambda),$$

where $\text{prefixMask}(\lambda)$ produces the bitmask covering the segments $[0, \lambda)$. This mask retains only the bits that are part of the prefix in every node, so if any $(v_i.p \wedge M)$ differs from $(v_1.p \wedge M)$, the subtrees rooted at $v_1, \dots, v_k$ are mutually disjoint and recursion terminates immediately.

❷ **Level Misalignment.** When the operand nodes share the same prefix but reside at different levels, the deeper nodes cover subsets of the shallower ones, and we must first align them to a common level before intersecting. Let $v_D = \arg \max_{v_i} v_i.\lambda$ denote the deepest (most specific) node. For each shallower (less specific) node v_s with $v_s.\lambda < v_D.\lambda$, we recursively descend into that node following the prefix bits of v_D until we reach at least v_D 's level (descendInto). If the descent reaches a missing entry, no value in v_s 's subtree matches v_D 's prefix, and the intersection is empty. Otherwise the descent path is unique, though prefix compression may carry it past v_D 's level; the recursive call re-checks alignment and descends again if needed.

❸ **Level-wise Intersection.** When all k nodes share compatible prefixes and reside at the same level, their subtrees cover the same subset of the universe, so their intersection reduces to independently intersecting each k -tuple of corresponding children. To identify which children exist in every subtree, we compute the

bitwise conjunction of all node bitmaps,

$$B_{\text{common}} = \bigwedge_{i=1}^k v_i.B,$$

and recurse only on positions $\mathcal{B} = \{j \mid B_{\text{common}}[j] = 1\}$, reducing the branching factor from the maximum of 2^b down to $|\mathcal{B}|$. The intersection then proceeds differently depending on whether the nodes are internal or leaf nodes:

❸.1 **Leaf Node Intersection:** At the leaf level, each payload entry is a bitmap over the final segment, so intersection reduces to a single k -ary bitwise AND per position $i \in \mathcal{B}$:

$$N.P^{01}[i] = \bigwedge_{j=1}^k v_j.P^{01}[i].$$

Positions that yield an empty bitmap contribute no values to the result, so the presence bitmap is updated as (cf. line 24):

$$N.B[i] = \begin{cases} 1 & \text{if } N.P^{01}[i] \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

❸.2 **Internal Node Intersection:** For internal nodes, each position $i \in \mathcal{B}$ may produce a non-empty child subtree, so the procedure recurses on the corresponding k -tuple of children. As with leaf nodes, the presence bitmap reflects only positions whose recursive call returned a non-empty result (cf. line 28):

$$N.B[i] = \begin{cases} 1 & \text{if } N.P^{-}[i] \neq \text{null} \\ 0 & \text{otherwise.} \end{cases}$$

Since PRISM requires every internal node to have at least two children, the algorithm restores this invariant before returning by pruning empty subtrees and collapsing a node left with a single child ($\text{collapseIfSingleton}$). This keeps the result compact and well-formed, improving space efficiency and query performance for subsequent operations.

3.1.2 Intersection Example. Figure 3 illustrates the execution trace of the algorithm on the binary case ($k = 2$); the general k -way case proceeds analogously with k simultaneous descents. The input comprises two PRISM indexes L and R , both sharing the root prefix $0x1$ at level $\lambda = 1$. First, the algorithm executes the prefix matching check (❶) and computes $B_{\text{common}} = 11$, indicating that both child positions require recursive processing (❸) (cf. Figure 3a). Based on this result, the algorithm then explores two parallel branches corresponding to the set bits in B_{common} (❸.2).

For the first branch ($B_{\text{common}}[0] = 1$, cf. Figure 3b), the algorithm detects a level misalignment: L 's child resides at $\lambda = 4$ while R 's child resides at $\lambda = 2$. Following (❷), the algorithm identifies the shallower node ($\lambda = 2$) and descends to the appropriate child determined by the deeper node's prefix ($\lambda = 4$), reaching the target node at the common level. The subsequent recursive call detects a prefix mismatch between $0x1000$ and $0x1001$, triggering immediate subtree pruning (cf. Figure 3d).

For the second branch ($B_{\text{common}}[1] = 1$, cf. Figure 3c), both children are leaf nodes at $\lambda = 4$ with matching prefixes. The algorithm applies leaf-level intersection (❸.1), computing the bitwise conjunction of the payload bitmaps to produce the final result in Figure 3e.

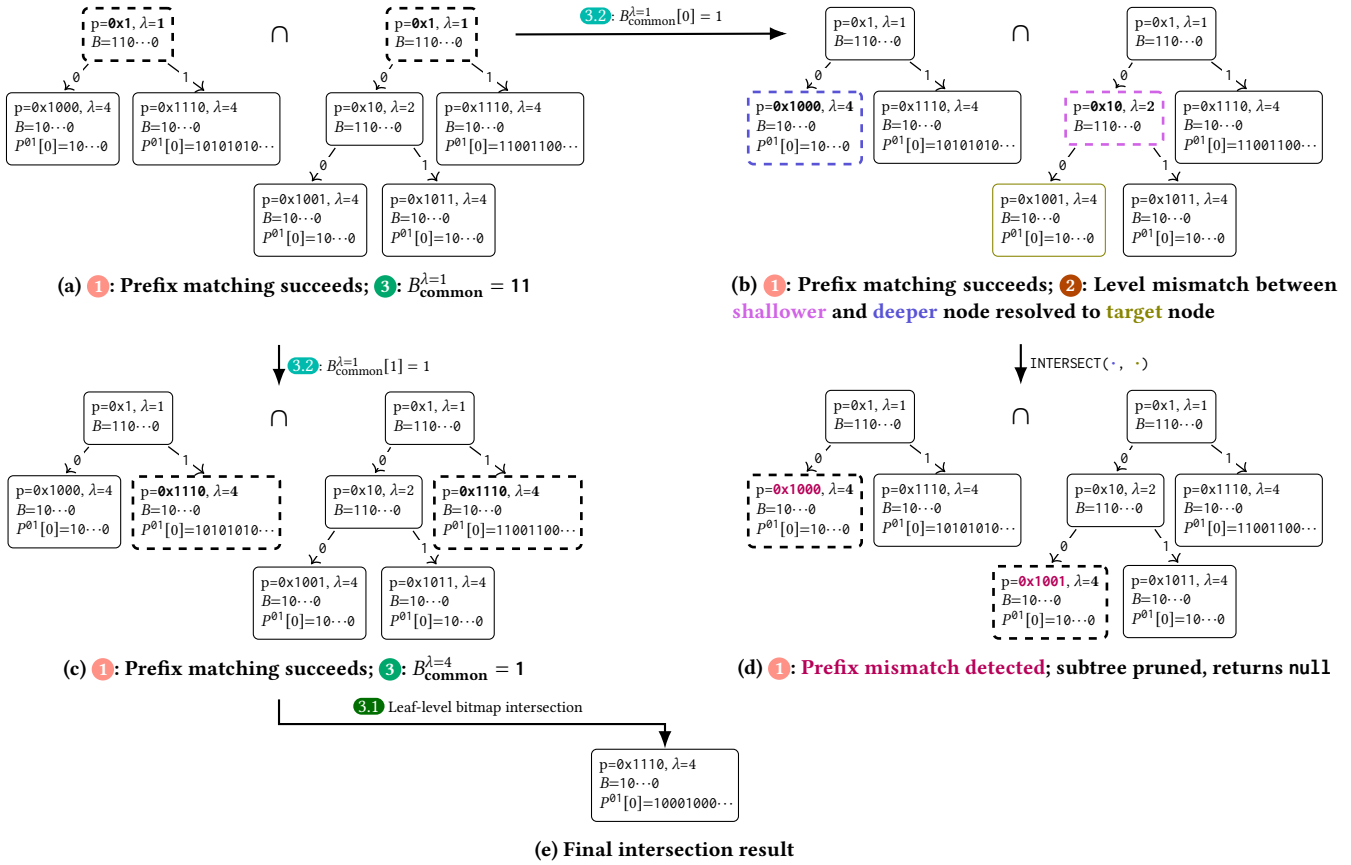


Figure 3: Execution trace of the intersection algorithm demonstrating prefix-based pruning and bitmap conjunction. Path (a)→(b)→(d) terminates due to prefix mismatch; path (a)→(c)→(e) completes via leaf-level intersection. Boldface and dashed borders mark active nodes at each recursion step; annotations reference algorithm phases from Listing 1.

3.2 Other Set Operations

The trie structure of PRISM extends naturally to set union ($\cup$), difference ($\setminus$), and symmetric difference (Δ). Where intersection prunes disjoint subtrees entirely, these operations handle them according to their specific semantics: copying, merging, or discarding the subtree. Listing 2 presents a generalized framework that unifies all three operations over PRISM. The procedure traverses the trie recursively, applying operation-specific logic at each level according to four principal cases.

Handling Empty Sets (Listing 2, lines 2–7). The algorithm first handles the degenerate cases where one or both operands are null. If both are null, so is the result. If only L is null, union and symmetric difference clone R , while difference returns null: L is empty, so the difference is empty. If only R is null, all three clone L .

Disjoint Prefixes (Listing 2, lines 8–11). Before descending, the algorithm checks whether the prefixes overlap. For disjoint operands, union and symmetric difference combine the two subtrees via `mergeDisjoint`, whereas set difference reduces to a clone of L , since none of L ’s elements are removed.

Level Misalignment (Listing 2, lines 12–29). If $L.\lambda \neq R.\lambda$, the shallower (less specific) node S must descend to the level of the deeper (more specific) node D . The algorithm extracts index i from D ’s prefix at level $S.\lambda$ to locate D among S ’s children. If this slot is unoccupied ($S.B[i] = 0$), the operands do not overlap there: union and symmetric difference clone S and attach a clone of D at position i (line 18), whereas set difference returns a clone of L , since no element of L is removed (line 19). If slot i is occupied, the algorithm processes S ’s children: at position i , it recurses on the corresponding child and D (lines 21–23); at all other positions, it clones the child for union and symmetric difference, or for set difference only if $S = L$ (lines 24–25). For each inspected position, $N.P^{\rightarrow}[j]$ is updated with the result c and $N.B[j]$ records whether c is non-null (cf. line 28).

Level-wise Processing (Listing 2, lines 30–48). If $L.\lambda = R.\lambda$, both nodes lie at the same level and their children are processed jointly. First, the algorithm computes the shared child positions via bitwise AND on their presence bitmaps. For these positions, it either applies the operation-specific bitwise primitive to the payload bitmaps when the nodes are at leaf level (cf. apply in Table 1), or recurses on the corresponding child pointers. A position is marked

```

1 function SET_OP(op)(L, R)
2   if L = null and R = null // Handling Empty Sets
3     return null
4   if L = null
5     return (op ∈ {∪, Δ}) ? clone(R) : null
6   if R = null
7     return clone(L)
8   // Disjoint Prefixes
9   M ← prefixMask(L.λ) ∧ prefixMask(R.λ)
10  if (L.p ∧ M) ≠ (R.p ∧ M)
11    return op ∈ {∪, Δ} ? mergeDisjoint(L, R) : clone(L)
12  if L.λ ≠ R.λ // Level Misalignment
13    (S, D) ← L.λ < R.λ ? (L, R) : (R, L)
14    N ← createNode(S.λ, S.p)
15    leftShallow ← (S = L)
16    i ← extractBits(D.p, S.λ)
17    if S.B[i] = 0 // no overlap at i
18      return op ∈ {∪, Δ} ? (clone(S) with clone(D) at i)
19                        : clone(L)
20  for each set bit j in S.B do // overlap: descend
21    if j = i
22      c ← SET_OP(op)(leftShallow ? (S.P→[j], D)
23                    : (D, S.P→[j]))
24    else if op ∈ {∪, Δ} or (op = \ and leftShallow)
25      c ← clone(S.P→[j])
26    else continue
27    N.P→[j] ← c
28    N.B[j] ← c ≠ null
29  return N.B = 0 ? null : collapseIfSingleton(N)
30  // Level-wise Processing
31  N ← createNode(L.λ, L.p ∧ M)
32  Bcommon ← L.B ∧ R.B
33  for each set bit i in Bcommon do // Handle common
34    if isLeaf(L)
35      N.P01[i] ← apply(L.P01[i], R.P01[i], op)
36      N.B[i] ← N.P01[i] ≠ 0
37    else
38      N.P→[i] ← SET_OP(op)(L.P→[i], R.P→[i])
39      N.B[i] ← N.P→[i] ≠ null
40  // Handle left-only children (all three operations)
41  for each set bit i in L.B ∧ ¬R.B do
42    N.P[i] ← clone(isLeaf(L) ? L.P01[i] : L.P→[i])
43    N.B[i] ← 1
44  if op ∈ {∪, Δ} // Handle right-only children
45    for each set bit i in R.B ∧ ¬L.B do
46      N.P[i] ← clone(isLeaf(R) ? R.P01[i] : R.P→[i])
47      N.B[i] ← 1
48  return N.B = 0 ? null : collapseIfSingleton(N)

```

Listing 2: Set operations $\cup$, $\setminus$, and Δ

in the result node’s presence bitmap $N.B$ only if it contributes: a non-zero bitmap at leaf level or a non-null child (cf. line 36 and line 39). It then handles positions unique to each operand: union ($\cup$) and symmetric difference (Δ) clone children from both sides (lines 40–43 and lines 44–47), whereas difference ($\setminus$) clones only children unique to L (lines 40–43). The presence bitmap $N.B$ is updated to mark these children (cf. line 43 and line 47).

To reduce cloning cost, we implement copy-on-write (COW): each node maintains a reference count, incremented on shared copy and decremented on release. Nodes are duplicated only when modified during recursive descent. This avoids unnecessary duplication when large subtrees remain unchanged, as is common in $\cup$, $\setminus$, and Δ .

Table 1: Set operation-specific bitwise primitive.

	$\cup$	$\setminus$	Δ
apply(x, y, op)	$x \vee y$	$x \wedge \neg y$	$x \oplus y$

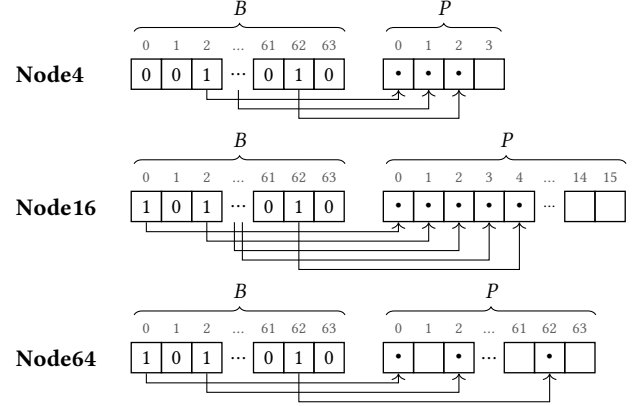


Figure 4: Memory layout of the morphing nodes. Each node stores a 64-bit bitmap B indicating present children and a payload array P . Node4 and Node16 pack payload entries compactly (arrows show bitmap-to-array mapping), while Node64 uses direct indexing. Bullets mark occupied slots.

4 MORPHING NODES

Workloads spread occupancy unevenly: some nodes are dense, some are sparse, and no single layout serves both. Storing all 2^b payload entries per node wastes memory on sparse nodes, while a uniformly compact layout slows access to dense ones. Sizing containers to their load is established practice: ART [19] grows radix nodes through capacity classes and ROARING [9, 22] switches container representations as a partition fills. PRISM uses three node types (Figure 4): Node4 and Node16 store 4 or 16 children in compact arrays; Node64 stores up to 64 by direct indexing.

Unlike ART and ROARING, which couple their operations to the container representation, PRISM exposes a node-type-agnostic interface. Every node type carries the same fixed-width 64-bit presence bitmap B , where bit i records whether child i is present. The operations that detect shared children and prune the remaining subtrees read B alone, independent of node type.

4.1 Node Structure and Access Patterns

Each node carries a prefix p , a level indicator λ , and a 64-bit presence bitmap B . With this, any node, regardless of its type, answers whether child i is present with a single bit test on B , which is all that detecting shared children and pruning subtrees requires. The types differ only in how B maps to the payload array P ; Figure 4 shows this mapping and Listing 3 shows how it is realized efficiently using available instructions. Growth selects among them by observed density, recovering memory at sparse nodes without altering the algorithms.

```

1 function ACCESS_CHILD(v, i)
2   if v is Node4 or Node16
3     pos ← popcount(v.B ∧ (2i - 1)) // Compressed index
4     return v.P[pos]
5   else // v is Node64
6     return v.P[i] // Direct indexing

```

Listing 3: Morphing-node child access

```

1 function INSERT_NODE(v, i, c)
2   m ← 1 << i // Bit for position i
3   if v is Node4 or Node16
4     n ← popcount(v.B) // Current child count
5     pos ← popcount(v.B ∧ (m - 1)) // Compressed index
6     present ← (v.B ∧ m) ≠ 0
7     if n < capacity(v) or present
8       upsert(v.P, pos, c, present) // Shift-insert or
          overwrite
9     v.B ← v.B ∨ m
10  else
11    v' ← getNextAndCopy(v) // Grow to next size class
12    return INSERT_NODE(v', i, c)
13  else // v is Node64
14    v.B ← v.B ∨ m
15    v.P[i] ← c // Direct write
16  return v

```

Listing 4: Morphing-node insertion

Node4 and Node16 store children contiguously in compressed form. To locate child i , we map its bitmap position to a compressed index: $pos = \text{popcount}(B \wedge (2^i - 1))$. This counts the set bits before position i in $O(1)$ time using a hardware population count instruction (Listing 3, lines 3–4), keeping only occupied children in memory.

Node64 drops the indirection: child i sits at position i in P (Listing 3, line 6). Access is direct, at the cost of allocating all 64 slots regardless of occupancy.

4.2 Dynamic Node Growth

Insertion keeps each node sized to its load by growing it only when capacity is reached. Listing 4 gives the procedure, branching on node type.

For Node4 and Node16, we locate the compressed index pos with $\text{popcount}(B \wedge (m - 1))$, where $m = 2^i$ is the bit for position i , and test whether child i is already present. If the node has room or the child is already present, `upsert` writes c at pos , shifting later entries for a new child or overwriting an existing one, and sets bit i in B (lines 7–9). Otherwise the node is full: we get a node from the next size class, copy the existing children, and reinsert recursively (lines 11–12). A node undergoes at most two such transitions over its lifetime (Node4 to Node16 to Node64), so growth adds only amortized constant cost per insertion. Node64 needs no indirection: it sets bit i in B and writes c to $P[i]$ (lines 14–15).

Growth optimization. The Node16→Node64 transition expands compressed children into direct-indexed positions. Where available, AVX-512 accelerates this: each 8-bit bitmap segment is processed with masked loads and the `vpexpandq` instruction.

4.3 SIMD Vectorization

SIMD processes many bitmap positions per instruction, and PRISM exposes three opportunities to exploit this across all node types. First, leaf set operations are bitwise over many bitmaps; the fixed-width P^{61} layout maps to wide registers, so AVX-512 handles eight 64-bit bitmaps per instruction. Second, synchronized traversals yield a common bitmap B_{common} that doubles as a mask for vectorized loads and stores, touching only positions with $B_{\text{common}}[i] = 1$. Third, vectorized comparisons mark the non-zero result positions to emit, cutting branch mispredictions in inner loops.

Realizing all three on every node type is not automatic: Node4 and Node16 store children in compressed form, breaking the contiguous loads the opportunities above assume. PRISM keeps a single kernel across layouts by reconciling the compressed form with masked memory operations:

- (1) `mask_expandloadu` spreads compressed children into vector lanes according to the presence bitmap
- (2) positions absent in B_{common} are zeroed
- (3) bitwise operations (AND/OR/XOR) run across the register
- (4) `mask_compressstoreu` writes results back in compressed form

When a node grows to Node64, storage is contiguous and the masked path gives way to direct AVX-512 access: eight bitmaps load, operate, and store in one pass. Throughout, PRISM uses AVX-512 at full register width and falls back to scalar processing for sparser sets.

5 EVALUATION

We evaluate PRISM against strong baselines from every algorithmic category surveyed in Section 7, covering sorted and SIMD merge-based intersection, hierarchical representations, hash-based methods, and compressed bitmaps. For PRISM, we consider both PRISM64, which uses only Node64 layouts, and the full morphing-node design.

Our primary competitors are two 64-bit adaptations of ROARING [9, 22]¹: ROARING (C++) partitions keys by their upper 32 bits via a `std::map` and stores the lower halves in the standard 32-bit Roaring structure; ROARING (C) uses a 48/16-bit split, indexing the upper 48 bits through an ART [19] that maps to Roaring containers for the remaining 16-bit suffixes.

For SIMD merge-based intersection, we include SIMD GALLOPING [20], BMISS [17], and QFILTER [15], which exploit vectorized comparisons to intersect sorted sequences. For hierarchical representations, we include HERO [31] and the balanced-tree baseline `STD::SET`, a red-black tree. We exclude the compressed binary tries (CBT) [2] because its implementation targets 32-bit keys and returns incorrect results on our 64-bit inputs. For hash-based methods, we include FESIA [32] and `TSL::ORDERED_SET` (TSL)². For compressed bitmaps, we retain ROARING and EWAH [21]³, which encode sets as run-length compressed bit vectors.

Experimental Setup. Experiments run on two platforms spanning x86-64 and ARM; Table 2 lists hardware and compiler details. On AMD Ryzen 7950X, we enable all vector extensions up to AVX-512,

¹<https://github.com/RoaringBitmap/CRoaring> (v4.6.1, accessed 2026-06-30)

²<https://github.com/Tessil/ordered-map> (v1.2.0, accessed 2026-06-30)

³<https://github.com/lemire/EWAHBoolArray> (v0.8.0, accessed 2026-06-30)

Table 2: Experimental platforms.

AMD Ryzen 7950X	AVX-512, Cache: 32K / 1M / 64M ^s 64GB DDR5 GCC 14.2 and Clang 20.1
Apple M3 Ultra	NEON, Cache: 128K ^p / 64M ^u 256GB LPDDR5 Clang 21.2

^sShared L3. ^pPer P-core (64K per E-core). ^uShared by P-cores (4M by E-cores).

implement both AVX2 and AVX-512 kernels for the SIMD merge-based methods, and report the faster one. The Apple M3 Ultra has no hand-vectorized kernels, so we omit the SIMD merge-based methods there. All builds use `-O3`, with `-march=native` on x86-64 and `-march=armv8-a+simd` on ARM. Each experiment runs 100 times on a single thread and reports average throughput. Due to memory limitations, we run TSL only up to set sizes of 10^5 .

Data Distributions. We use four synthetic distributions of 64-bit integers:

Dense: the contiguous range $[1, n]$.

Grid: each byte of each key drawn uniformly from $[1, 14]$ [25].

Pointer: clustered keys simulating tuple identifiers, combining a random base from $\{i \cdot 2^{16} : i \in [0, 2^{20}]\}$ with a random offset from $[0, 2^{12}]$.

Sparse: uniform random samples from $[0, 2^{64}]$.

Overlap Coefficient. To quantify similarity between two sets A and B , we use the overlap coefficient

$$\text{overlap}(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)}.$$

For set operations, the first operand is drawn from one of the distributions above; the second is constructed by sampling a fraction of the first operand’s elements equal to the target overlap coefficient, then filling the remaining positions with uniform random 64-bit integers. Thus, at low overlap, the second operand is dominated by uniform keys.

5.1 Set Operations

As our data structure is primarily designed to optimize set operations, we evaluate them across distributions and overlap coefficients on the machines described above.

5.1.1 Intersection. Figure 5 reports the fastest algorithm per size combination, broken down by distribution and overlap coefficient on AMD Ryzen (AVX-512) and Apple M3 Ultra (no SIMD kernels). For ROARING, we consider both the C++ and C interfaces and both processing orders, reporting the best-performing one per configuration.

When the two sets are mostly disjoint, PRISM nearly always outperforms the competitors: its hierarchical layout enables aggressive early termination when subtrees share no keys. This is most pronounced on dense distributions, where large contiguous regions can be skipped entirely, and also holds across the grid and pointer distributions on both platforms. On Apple M3 Ultra, PRISM and ROARING achieve comparable throughput at zero overlap.

With increasing overlap, PRISM mostly dominates on the dense and grid distributions. However, on the dense distribution, once the first operand exceeds 10^4 elements, ROARING begins to outperform PRISM. At this scale, at least one operand consists entirely of bitmap containers, enabling ROARING’s specialized bitwise AND routines at minimal overhead. PRISM supports analogous node-level bitwise operations, but its trie layout requires loading and intersecting substantially more nodes.

At higher overlap, on the sparse and pointer distributions, PRISM falls behind the SIMD merge-based approaches. Both distributions spread elements across many nodes without forming large contiguous regions, so PRISM cannot exploit subtree pruning and must instead traverse and allocate result nodes individually; by construction, the pointer distribution behaves like a sparse one over a smaller universe. SIMD merge-based algorithms avoid this overhead entirely: since they operate on flat, contiguous, sorted arrays, they reduce the intersection to a tight scan with no pointer chasing or node allocation, which is cheaper than walking and rebuilding PRISM’s trie node by node. On Apple M3 Ultra, the SIMD merge-based methods are unavailable, and hash-based approaches take the lead instead, avoiding the same per-node tree overhead through $O(1)$ lookups. Skewed workloads flip this: when one side is much smaller than the other, PRISM wins.

5.1.2 Other Set Operations. Figure 6 reports the speedup of PRISM over `STD::SET` and ROARING on set operations at overlap coefficients of 0%, 50%, and 100%. For the dense, grid, and pointer distributions, PRISM outperforms both competitors on nearly all operations at low overlap coefficients; these speedups mainly come from PRISM’s COW semantics, which let it share whole disjoint subtrees with the result instead of traversing and materializing them. This optimization fades as the operands begin to overlap: they then occupy the same nodes and differ only in which elements they mark, so PRISM must traverse and merge those nodes individually. Speedups relative to ROARING therefore shrink as the overlap coefficient grows. The effect is most pronounced for the dense distribution with identical sets larger than 10^4 elements, where PRISM’s trie layout requires merging substantially more nodes, increasing cache pressure and reducing throughput. For the sparse distribution, the speedup is smaller: PRISM performs on par with `STD::SET` and is roughly $10\times$ faster than ROARING, as PRISM and ROARING must traverse and merge more nodes or containers.

5.2 Insertion Performance and Threshold-based Intersection

Beyond set operations, two more aspects matter for integer set structures: insertion performance and threshold-based intersection.

5.2.1 Insertion Performance. Figure 7 reports insertion throughput for ordered and unordered sequences and per-element memory. We include only structures that support elementwise insertion without needing the whole set at once. Where available, we use bulk insertion interfaces, which cache the last-accessed container or node to favor ordered and dense workloads.

Ordered Insertion. On the dense distribution, PRISM64 comes closest to EWAH, which still leads through run-length encoding; on grid, it matches or exceeds EWAH up to 10^5 elements before

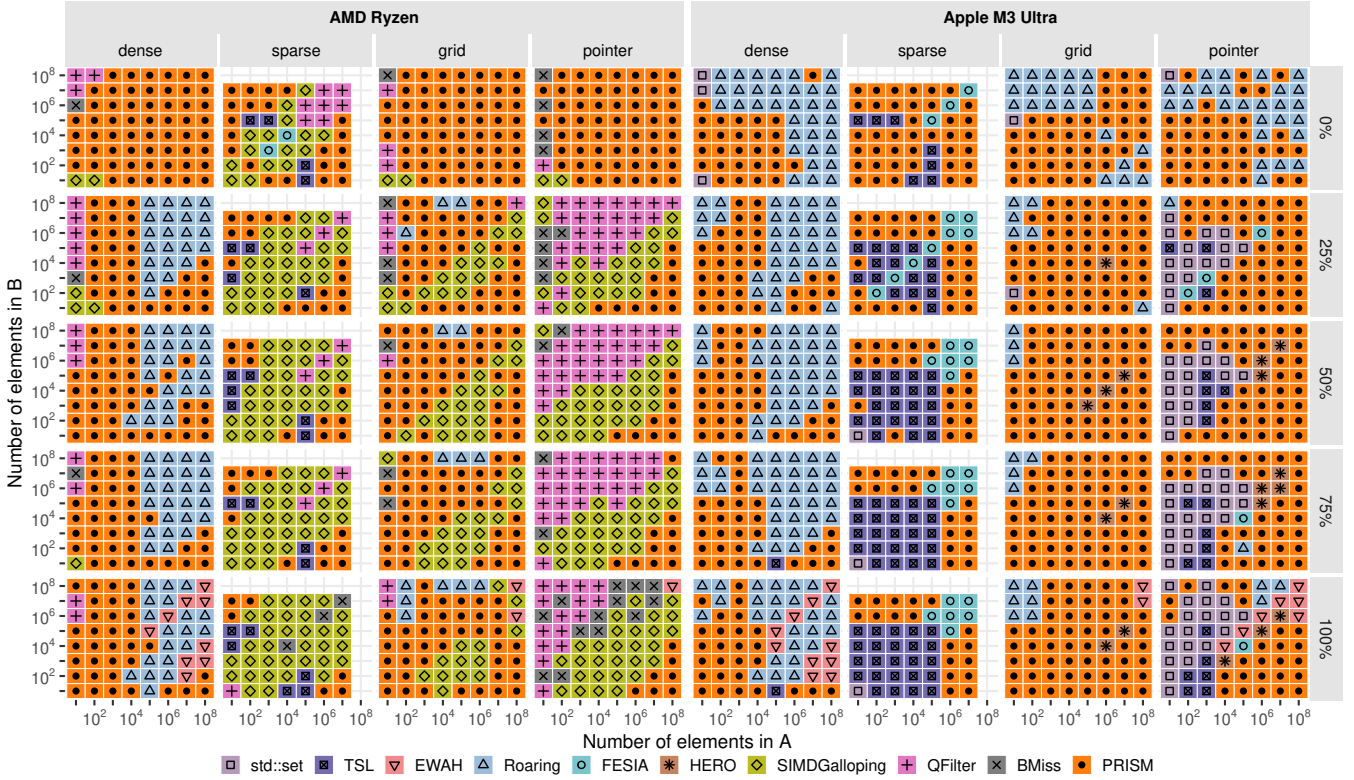


Figure 5: Best-performing intersection algorithm by distribution and overlap coefficient, on AMD Ryzen and Apple M3 Ultra (PRISM64 is omitted). We exclude SIMD merge-based methods on Apple M3 Ultra, which we run without hand-vectorized kernels. Sparse distributions are not measured for sets of 10^8 elements.

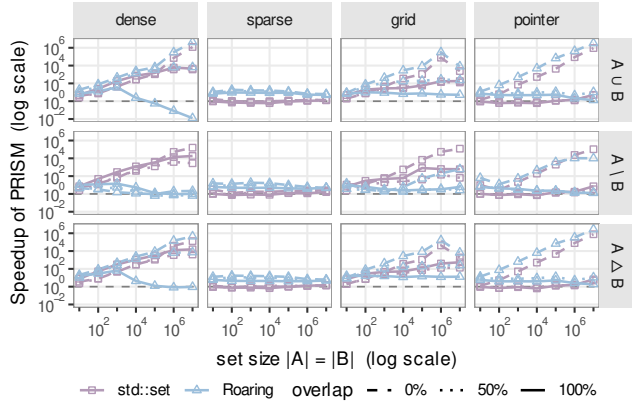


Figure 6: Speedup of PRISM over `std::set` and Roaring on set operations across distributions and overlap coefficients for equally sized sets on AMD Ryzen.

falling behind at larger cardinalities. PRISM trails PRISM64 due to morphing overhead, but still matches or beats both ROARING variants, which pay for dynamic container expansion. ROARING (C) is compiler-sensitive: GCC roughly halves its throughput versus

ROARING (C++), while Clang reaches parity. For the pointer distribution, PRISM and PRISM64 land between `std::set` and ROARING. On the sparse distribution, PRISM stays the fastest compressed structure, ahead of both ROARING variants and EWAH, since its morphing nodes avoid over-allocation for isolated keys; PRISM64 degrades at scale and falls behind the ROARING variants; among all structures, only `std::set` is faster than PRISM here.

Unordered Insertion. We exclude EWAH here, as it requires sorted input. On dense and grid distributions, PRISM64 achieves the highest throughput: most keys fit within a single Node64 up to 10^3 elements on dense and 10^2 elements on grid, and throughput peaks before further node allocations bring it down. PRISM pays for morphing-node growth, and both ROARING variants stay ahead of `std::set` by up to 2× in the mid-range. On pointer and sparse distributions, `std::set` leads up to 10^3 elements but degrades at larger cardinalities due to balanced-tree maintenance; PRISM64 and PRISM then take over. ROARING (C++) shows the lowest throughput, since dispersed keys force separate container allocations. On the pointer distribution, throughput recovers slightly past 10^6 elements once sorted-array containers convert to bitmap containers.

Memory Consumption. The right panel of Figure 7 reports per-element memory after all insertions. EWAH has one of the smallest footprints on the dense, grid, and pointer distributions through

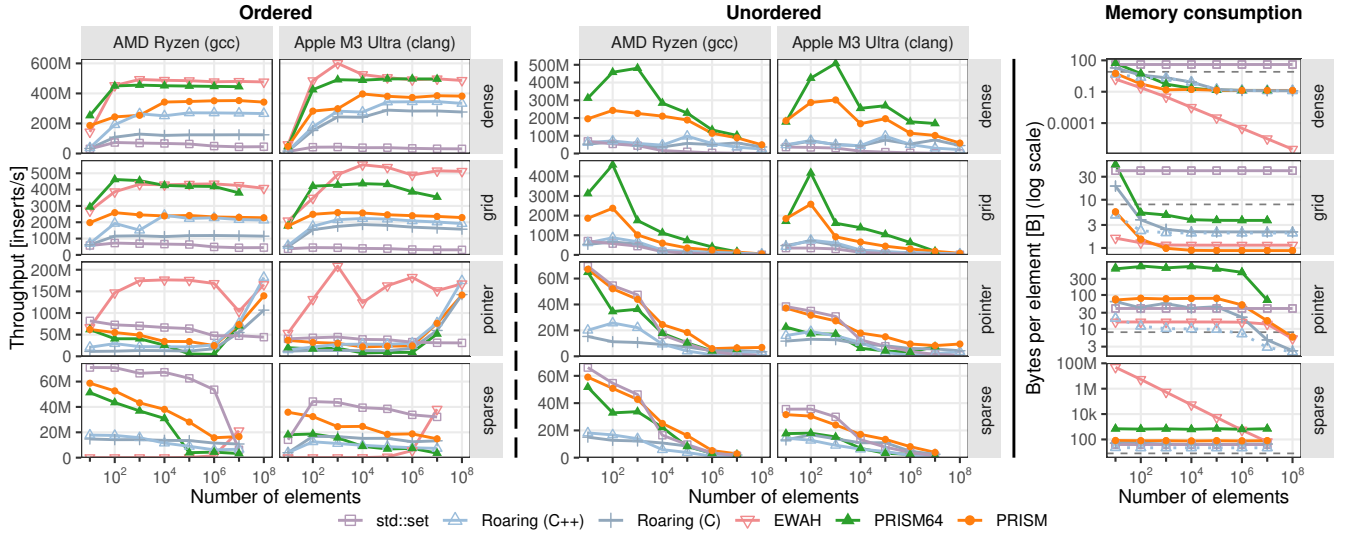


Figure 7: Insertion throughput (left: ordered; middle: unordered) and per-element memory (right) across platforms and distributions. The dashed line marks the size of the uncompressed data (8 bytes per element). ROARING is dotted, as its reported footprint counts only the serialized bitset.

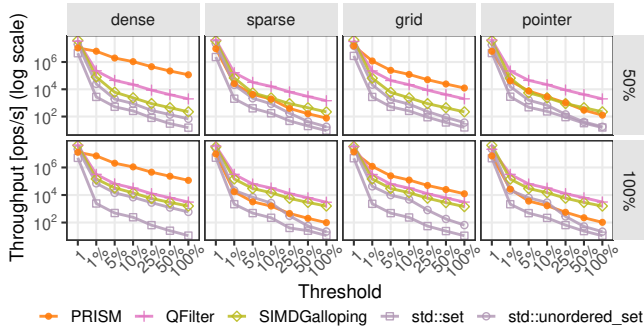


Figure 8: Threshold-based intersection throughput on AMD Ryzen for two sets of one million elements each, across distributions, overlap coefficients, and thresholds.

run-length compression, but is largest on sparse data. On the dense distribution, every structure except `STD::SET` and `EWAH` converges to a comparable per-element cost. The grid, pointer, and sparse distributions show the advantage of morphing nodes: `PRISM` uses about 8× less memory per element than `PRISM64` on pointer and sparse, and about 4× on grid. Both `ROARING` footprints are underestimates, reflecting only the serialized bitset size rather than the full in-memory representation.

5.2.2 Threshold-based Intersection. In some scenarios, one does not need the full intersection of two sets, only whether its size exceeds a given threshold. Figure 8 reports threshold-based intersection throughput on AMD Ryzen for two sets of one million elements each, across distributions, overlap coefficients, and thresholds. We set the threshold to a single element (a test for non-empty intersection) or to 1%, 5%, 10%, 25%, 50%, or 100% of the actual

intersection size. Throughput drops as the threshold grows, since a larger fraction of the intersection must be computed before termination. The overlap coefficient barely affects throughput: `PRISM` leads on the dense and grid distributions and trails `QFILTER` and `SIMD GALLOPING` on the pointer and sparse distributions. Those two distributions scatter elements across more nodes, so traversal issues more memory accesses.

6 REAL-WORLD CASE STUDIES

We evaluate three real-world scenarios: conjunctive and disjunctive queries on the Star Schema Benchmark (SSB), a simulated trigram-based inverted index on TPC-H, and inverted-list intersection on the ClueWeb09 dataset.

We compare against representatives of each major category: the ordered container `STD::SET`; the hash-based `TSL` and `FESIA` [32]; the hierarchical `CBT` [2], whose implementation is limited to 32-bit elements but applicable here since all elements fall within that range, and `HERO` [31]; the compressed bitmaps `EWAH` [21] and `ROARING` (C and C++ interfaces) [9, 22]; and the SIMD-accelerated `BMISS` [17], `QFILTER` [15], and `SIMD GALLOPING` [20]. Hash-based competitors iterate over the smallest set and probe the others. `ROARING` keeps its native container order here, since processing smallest-first slows it down. Measurements exclude preparation time such as building or sorting.

6.1 Case Study: Filtering on SSB

SSB joins a central fact table (`LINEORDER`) with four dimension tables. On our AMD machine, we evaluate the filtering portion of queries Q1.1, Q2.1, Q3.4, and Q4.1 [23, 29] at scale factors 1, 10, and 100. Q1.1 and Q2.1 reduce to the intersection of three and two sets, respectively, while Q3.4 and Q4.1 add disjunctive predicates and

Table 3: Query throughput (ops/s) on SSB across scale factors. Best result per column (excluding PRISM64) is bold; PRISM64 is PRISM without morphing, isolating its effect.

Algorithm	Q1.1			Q2.1		
	1	10	100	1	10	100
PRISM	13.7M	11.8M	4.1M	21.9M	5.0M	396K
PRISM64	15.7M	13.2M	5.2M	27.9M	6.2M	441K
ROARING (C)	1.9M	1.9M	36K	5.1M	638K	206K
ROARING (C++)	2.3M	2.1M	739K	2.0M	367K	91K
CBT	198K	187K	150K	206K	115K	23K
FESIA	220	23	1	313K	72K	19K
HERO	14K	615	24	858K	183K	40K
EWAH	5.6M	5.4M	1.5M	1.2M	287K	68K
STD::SET	202K	212K	174K	693K	55K	3K
TSL	549K	591K	382K	1.6M	118K	4K
BMiss (AVX2)	1.4M	1.3M	1.0M	3.1M	330K	30K
QFILTER (AVX-512)	1.6M	1.5M	1.2M	4.4M	438K	42K
SIMD GALLOPING (v1)	2.1M	2.0M	1.5M	3.6M	391K	38K

Algorithm	Q3.4			Q4.1		
	1	10	100	1	10	100
PRISM	2.9M	279K	26K	507K	87K	52K
PRISM64	2.9M	343K	32K	510K	121K	52K
ROARING (C)	1.2M	332K	39K	406K	99K	57K
ROARING (C++)	1.7M	328K	31K	341K	73K	30K
FESIA	489K	42K	1K	328	73	37
HERO	543K	56K	4K	1K	314	184
EWAH	465K	49K	4K	130K	30K	14K
STD::SET	131K	14K	1K	349	75	34
TSL	733K	74K	3K	1K	330	143

thus also require set union. Table 3 reports the throughput for each query across all scale factors.

On the pure intersection queries Q1.1 and Q2.1, PRISM and PRISM64 achieve the highest throughput at every scale factor, ahead of ROARING and EWAH by 2–8× in most configurations and ahead of the general-purpose baselines STD::SET and TSL by more than an order of magnitude. PRISM64 stays ahead of PRISM throughout, by up to about 1.3× with no consistent trend across scale factors.

On the union queries Q3.4 and Q4.1, PRISM leads at scale factor 1, but trails ROARING (C) at 10 and 100, where the bitmap merge wins once nearly every node must be combined. We attribute this to the structure of the SSB data: the elements are drawn from a small universe, so they share long common prefixes and differ only in their low-order bits. The union must therefore merge nearly every node, precluding the COW shortcut and increasing the number of accessed nodes, which raises cache pressure and lowers throughput. CBT, BMiss, QFILTER, and SIMD GALLOPING do not support set union and are therefore omitted.

6.2 Case Study: TPC-H Trigram Index

We simulate an inverted index by building a trigram index⁴ on the `part.name` column of the TPC-H dataset⁵. Each part name is decomposed into contiguous three-character sequences, and a map from trigrams to part-identifier sets is constructed. To evaluate a query string, we decompose it into trigrams and perform a k -way intersection over the corresponding sets, returning all parts whose

⁴<https://www.postgresql.org/docs/current/pgtrgm.html> (accessed 2026-06-30)

⁵<https://www.tpc.org/tpch/> (accessed 2026-06-30)

Table 4: Throughput (ops/s) for TPC-H trigram index queries across scale factors. Best result per column (excluding PRISM64) is bold; PRISM64 is PRISM without morphing, isolating its effect.

Algorithm	chocolate			firebrick			salmon		
	1	10	100	1	10	100	1	10	100
PRISM	120K	12K	1K	120K	12K	1K	152K	17K	2K
PRISM64	104K	11K	1K	106K	11K	1K	161K	16K	2K
ROARING (C)	50K	5K	469	89K	9K	872	53K	5K	533
ROARING (C++)	13K	1K	146	11K	1K	124	49K	5K	451
CBT	9K	701	68	9K	704	67	13K	1K	98
FESIA	407	34	3	443	35	3	798	63	5
HERO	2K	152	13	2K	154	13	5K	309	25
EWAH	20K	2K	192	20K	2K	192	39K	4K	391
STD::SET	415	26	2	457	32	2	628	35	3
TSL	611	50	4	660	51	5	1K	94	8
BMiss (AVX2)	11K	714	62	15K	2K	118	13K	591	56
QFILTER (AVX-512)	10K	733	64	13K	1K	105	12K	675	64
SIMD GALLOPING (v1)	13K	1K	89	15K	2K	113	27K	1K	117

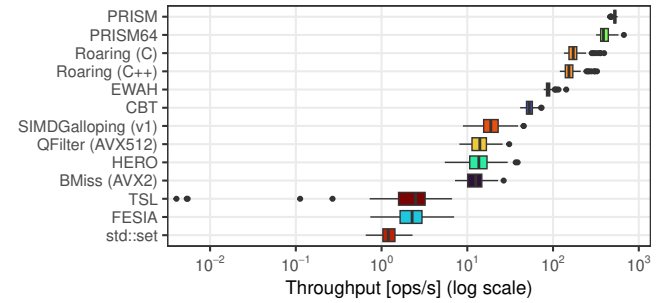


Figure 9: Intersection throughput on ClueWeb09 posting lists (AMD Ryzen). Each point intersects 4 lists randomly drawn from the 200 largest; boxes summarize 100 draws.

names contain every query trigram. Table 4 reports throughput across three representative query strings.

PRISM and PRISM64 achieve the highest throughput across all configurations. ROARING (C) and ROARING (C++) exhibit data-dependent performance: while they achieve comparable throughput on the salmon query, ROARING (C) is 8× faster than ROARING (C++) on the firebrick query. Throughput scales nearly linearly with the inverse of the scale factor for all structures, confirming that the bottleneck is proportional to set size. EWAH, the SIMD merge-based methods, and CBT form the next tier, trailing PRISM by roughly 4 to 13×, while HERO, FESIA, STD::SET, and TSL are the slowest, an order of magnitude or more behind.

6.3 Case Study: Inverted Lists on ClueWeb09

To cover a broader range of real-world workloads, we evaluate inverted-list intersection on the ClueWeb09 dataset⁶. We download a pre-built index⁷, select the 200 largest posting lists, and intersect 4 randomly chosen lists at a time, repeating the draw 100 times. These lists range from 8.2 to 35.5 million entries, with a median of

⁶<https://lemurproject.org/> (accessed 2026-06-30)

⁷<http://searchivarius.org/personal/data-sets/clueweb09gap> (accessed 2026-06-30)

11.6 million and a mean of 13.1 million. As Figure 9 shows, PRISM reaches the highest throughput. It is faster than both ROARING variants and faster than PRISM64 on most intersections.

7 BACKGROUND AND RELATED WORK

Efficient set intersection over fixed-size integers is a building block of query processing in databases, information retrieval, and data analytics. We first survey direct competitors, organized by representation: sorted arrays, hierarchical representations, hash-based methods, and compressed bitmaps. We also discuss the set containment join problem, in which set intersection appears as a subroutine.

Sorted Arrays and Merge-Based Intersection. Sorted arrays are the standard integer set representation, intersected by linear merge or by searching one operand in the other. Classical work reduces comparisons via galloping search and adaptive merge algorithms [3–6, 12, 16]. More recent work targets instruction throughput with SIMD, which compares multiple elements per cycle: BMISS [17], SIMDGALLOPING [20], and QFILTER [15].

Hierarchical Representations. Treaps, balanced trees, and skip lists give logarithmic search with efficient updates [7, 14, 24, 26]. Tries go further, since shared prefixes let structural similarity between sets do algorithmic work directly. Chen and Shen [10] encode multiple sets into a trie and reduce intersection to interval comparison and binary search. Closest to our work are two trie-based approaches: Arroyuelo and Castillo [2] represent integer sets as compressed binary tries and intersect them via synchronized depth-first traversal. Subtrees that cannot contribute to the result are pruned along the way. Each node branches on a single bit, so navigation moves one level per bit and relies on rank over succinct bit vectors rather than bitmap arithmetic. Descent and membership are separate primitives, and the design favors space over throughput. PRISM instead exposes every node through one fixed-width bitmap, so a single word-parallel AND covers several levels at once, where a binary trie can only descend one bit at a time. HERO [31] combines a hierarchical structure with bitmap summaries for graph workloads. It partitions each adjacency list into fixed-size segments across a fixed number of levels and encodes segment presence at each level as a bitmap. Together, these bitmaps form a *set intersection bitmap tree* (SIB-tree) that filters non-overlapping segments early. Its levels are heterogeneous, its depth is fixed rather than data-dependent, and all processing is scalar. PRISM differs on each point: every node at every depth exposes the same fixed-width bitmap, so one AND-and-recurse step handles pruning, navigation, and processing at once; adaptive prefix compression makes depth data-dependent; and the uniform layout lets SIMD operate both within and across levels.

Hash-Based Methods. When ordered access is not required, hash tables make a different trade-off. The standard approach builds a hash table over the smaller set and probes it with the larger set; expected time is $O(n)$ with $m \leq n$, regardless of value distribution. Ding and König [13] hash-partition sets into cache-friendly blocks, enabling word-parallel comparisons and reducing probe overhead. The trade-off is losing sorted order and the structural properties that hierarchical methods rely on. FESIA [32] sits between bitmap, hash-based, and merge-based methods. It hash-partitions each set into

segments and summarizes each segment with a compact bitmap signature. Intersection then runs in two phases: a bitwise AND of the signature arrays does Bloom-style filtering and prunes non-overlapping segments without touching the payloads; a SIMD kernel then intersects whatever survives via all pairwise comparisons within a segment.

Compressed Bitmaps. An alternative strategy encodes a set $S \subseteq [0, U)$ as a length- U bit vector. Intersection then reduces to word-parallel AND operations. However, raw bitmaps are impractical for large sparse universes. Run-length compression schemes such as BBC [1], WAH [30], EWAH [21], and CONCISE [11] address this by encoding runs of identical bits while preserving bitwise operations on the compressed representation. ROARING bitmaps [9] take a different approach and combine per-chunk container selection with SIMD-optimized operations [22]. A broad experimental study by Wang et al. [29] confirms ROARING as the fastest bitmap-based intersection method, while inverted lists are generally more space-efficient. Where ROARING dispatches on container type at every chunk, PRISM presents one *node-type-agnostic* bitmap interface.

Set Containment Joins. Set intersection is also part of set containment joins, which join tables on set-valued attributes under a subset constraint. Existing solutions [8, 18] build a trie over one relation and an inverted index over the other, then intersect the posting lists retrieved by trie traversal to produce the join output. This trie-over-inverted-file design was first proposed for single-relation containment search [27] and later adapted to joins. PRISM does not address containment joins directly, but its k -way intersection can accelerate the posting-list step.

8 CONCLUSION

We presented PRISM, a prefix-based integer set that exploits the hierarchical structure of 64-bit keys for efficient set operations. PRISM organizes keys into a trie of fixed-width segments, with nodes recording child occupancy as bitmaps. Every set operation then becomes a single synchronized top-down traversal. Node prefixes and bitmaps alone identify shared and disjoint subtrees, so we never examine individual elements, and we vectorize leaf-level processing with SIMD. The operations differ only in how they treat these subtrees: intersection prunes disjoint ones and extends to the k -way case by simultaneous descent; union, difference, and symmetric difference use copy-on-write sharing to handle subtrees exclusive to one operand. Morphing-node layouts size each node to its occupancy and compress sparse regions by up to $8\times$, at minor cost to dense throughput. PRISM is the fastest method we evaluated for pairwise intersection. It leads at low overlap on most distributions and stays competitive where SIMD merge-based, bitmap, and hash-based methods take over. Three real-world case studies support these gains: conjunctive and disjunctive queries on SSB, a k -way trigram-index workload on TPC-H, and inverted-list intersection on ClueWeb09. Across all three, PRISM leads on intersection-heavy workloads, and it remains competitive on union queries on SSB at larger scales. These results show that organizing integer data hierarchically exposes shared and disjoint regions, and that the structure alone is enough to identify them and prune redundant computation.

REFERENCES

- [1] Gennady Antoshenkov. 1995. Byte-aligned bitmap compression. In *Proceedings DCC '95 Data Compression Conference*. IEEE Computer Society, 476.
- [2] Diego Arroyuelo and Juan Pablo Castillo. 2023. Trie-Compressed Adaptive Set Intersection. In *CPM (LIPIcs)*, Vol. 259. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:19.
- [3] Ricardo A. Baeza-Yates. 2004. A Fast Set Intersection Algorithm for Sorted Sequences. In *CPM (Lecture Notes in Computer Science)*, Vol. 3109. Springer, 400–408.
- [4] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. 2006. Faster Adaptive Set Intersections for Text Searching. In *WEA (Lecture Notes in Computer Science)*, Vol. 4007. Springer, 146–157.
- [5] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. 2009. An experimental investigation of set intersection algorithms for text searching. *ACM J. Exp. Algorithmics* 14, Article 7 (2009), 24 pages.
- [6] Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An Almost Optimal Algorithm for Unbounded Searching. *Inf. Process. Lett.* 5, 3 (1976), 82–87.
- [7] Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *SPAA*. ACM, 16–26.
- [8] Panagiotis Bouras, Nikos Mamoulis, Shen Ge, and Manolis Terrovitis. 2016. Set containment join revisited. *Knowl. Inf. Syst.* 49, 1 (2016), 375–402.
- [9] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with Roaring bitmaps. *Softw. Pract. Exp.* 46, 5 (2016), 709–719.
- [10] Yangjun Chen and Weixin Shen. 2016. An efficient method to evaluate intersections on big data sets. *Theor. Comput. Sci.* 647 (2016), 1–21.
- [11] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inf. Process. Lett.* 110, 16 (2010), 644–650.
- [12] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. 2000. Adaptive set intersections, unions, and differences. In *SODA*. ACM/SIAM, 743–752.
- [13] Bolin Ding and Arnd Christian König. 2011. Fast Set Intersection in Memory. *Proc. VLDB Endow.* 4, 4 (2011), 255–266.
- [14] Leonidas J. Guibas and Robert Sedgwick. 1978. A Dichromatic Framework for Balanced Trees. In *FOCS*. IEEE Computer Society, 8–21.
- [15] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *SIGMOD Conference*. ACM, 1587–1602.
- [16] Frank K. Hwang and Shen Lin. 1972. A Simple Algorithm for Merging Two Disjoint Linearly-Ordered Sets. *SIAM J. Comput.* 1, 1 (1972), 31–39.
- [17] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions. *Proc. VLDB Endow.* 8, 3 (2014), 293–304.
- [18] Ravindranath Jampani and Vikram Pudi. 2005. Using Prefix-Trees for Efficiently Computing Set Joins. In *DASFAA (Lecture Notes in Computer Science)*, Vol. 3453. Springer, 761–772.
- [19] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE Computer Society, 38–49.
- [20] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exp.* 46, 6 (2016), 723–749.
- [21] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.* 69, 1 (2010), 3–28.
- [22] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2018. Roaring bitmaps: Implementation of an optimized software library. *Softw. Pract. Exp.* 48, 4 (2018), 867–895.
- [23] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC (Lecture Notes in Computer Science)*, Vol. 5895. Springer, 237–252.
- [24] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.
- [25] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (2015), 96–107.
- [26] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. *Algorithmica* 16, 4/5 (1996), 464–497.
- [27] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos K. Sellis. 2006. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*. ACM, 728–737.
- [28] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. OpenProceedings.org, 96–106.
- [29] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *SIGMOD Conference*. ACM, 993–1008.
- [30] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (2006), 1–38.
- [31] Boyu Yang, Weiguo Zheng, Xiang Lian, Yuzheng Cai, and X. Sean Wang. 2024. HERO: A Hierarchical Set Partitioning and Join Framework for Speeding up the Set Intersection Over Graphs. *Proc. ACM Manag. Data* 2, 1 (2024), 29:1–29:25.
- [32] Jiyuan Zhang, Yi Lu, Daniele G. Spampinato, and Franz Franchetti. 2020. FESIA: A Fast and SIMD-Efficient Set Intersection Approach on Modern CPUs. In *ICDE*. IEEE, 1465–1476.