Trie-Compressed Adaptive Set Intersection

Diego Arroyuelo
Departamento de Informática, Universidad Técnica Federico Santa María, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Juan Pablo Castillo
Departamento de Informática, Universidad Técnica Federico Santa María, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Abstract

We introduce space- and time-efficient algorithms and data structures for the offline set intersection problem. We show that a sorted integer set $S \subseteq [0..u)$ of n elements can be represented using compressed space while supporting k-way intersections in adaptive $O(k\delta \lg(u/\delta))$ time, δ being the alternation measure introduced by Barbay and Kenyon. Our experimental results suggest that our approaches are competitive in practice, outperforming the most efficient alternatives (Partitioned Elias-Fano indexes, Roaring Bitmaps, and Recursive Universe Partitioning (RUP)) in several scenarios, offering in general relevant space-time trade-offs.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Design and analysis of algorithms; Theory of computation → Data structures and algorithms for data management; Information systems → Information retrieval query processing

Keywords and phrases Set intersection problem, Adaptive Algorithms, Compressed and compact data structures

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.1

Supplementary Material *Software*: `https://github.com/jpcastillog/compressed-binary-tries` archived at `swh:1:dir:4ec1b85ad1d97fa10c648a3ad1ad2366c0cafb5c`

Funding This work was funded by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile (both authors).

Acknowledgements We thank Gonzalo Navarro, Cristian Riveros, Adrián Gómez-Brandón, and Francesco Tosoni for enlightening comments, suggestions, and discussions about this work. We also thank the anonymous reviewers whose thorough reviews helped us to improve this paper.

1 Introduction

Sets are one of the most fundamental mathematical concepts related to the storage of data. Operations such as set intersections, unions, and differences are key for querying them. E.g., the use of logical `AND` and `OR` operators in web search engines translate into intersections and unions, respectively. Representing sets to support their basic operations efficiently has been a major concern since many decades ago [4]. In several applications, such as query processing in information retrieval (IR) [15] and database management systems (DBMS) [23], sets are known in advance to queries, hence data structures can be built to speed up query processing. With this motivation, in this paper we focus on the following problem.

THE *OFFLINE SET INTERSECTION PROBLEM*, **OSIP**

Input: A family $\mathcal{S} = \{S_1, \ldots, S_N\}$ of N sorted integer sets over universe $[0..u)$, with $|S_i| = n_i$.

Task : To preprocess family $\mathcal{S}$ to efficiently support query instances of the form $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, which ask to compute $\mathcal{I}(\mathcal{Q}) = \bigcap_{i \in \mathcal{Q}} S_i$.

© Diego Arroyuelo and Juan Pablo Castillo;
licensed under Creative Commons License CC-BY 4.0
34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).
Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 1; pp. 1:1–1:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We assume $u = 2^k$ in this paper, for $k \geq 0$. Unless explicitly otherwise stated, we also assume $\lg x = \lceil \lg_2 x \rceil$ and $\lg 0 = 0$. Typical applications of this problem include the efficient support of join operations in DBMS [23, 51], query processing using inverted indexes in IR [15, 52], and computational biology [33], among others. Building a data structure to speed up intersections, however, increases the space usage. Today, data-intensive applications encourage not only time- but also space-efficient solutions [7]. Being able to process big datasets entirely in main memory is the main motivation. Compact, succinct, and compressed data structures are important to achieve this [41]. We study here compressed data structures to efficiently support the OSIP. We assume the word RAM model of computation with word size $w = \Theta(\lg u)$. Arithmetic, logic, and bitwise operations, as well as accesses to w-bit memory cells, take $O(1)$ time.

The literature on this problem is vast. For the online version of the problem, where sets to be intersected are given at query time – so there is no time to preprocess them – algorithms like the ones by Baeza-Yates [9], Demaine et al. [20], and Barbay and Kenyon [12] are among the most efficient and well-known approaches. In particular, the two latter algorithms are adaptive, meaning that they are able to perform faster on "easier" query instances. The algorithm by Barbay and Kenyon runs in optimal $O(\delta \sum_{i \in \mathcal{Q}} \lg(n_i/\delta))$ time, where δ is the so-called alternation measure that quantifies the query difficulty [12]. The algorithm by Demaine et al. [20] has running time $O(k\delta \lg(n/\delta))$, for $n = \sum_{i \in \mathcal{Q}} n_i$, which is optimal when $\max_{i \in \mathcal{Q}} \{\lg n_i\} = O(\min_{i \in \mathcal{Q}} \{\lg n_i\})$ [11]. These algorithms require sets to be stored in plain form, e.g. using a sorted array or a B-tree [20], requiring $\Theta(mw)$ bits of space, for $m = \sum_{i=1}^{N} n_i$. This can be excessive when dealing with large databases.

For the OSIP, we have the extensive literature on inverted indexes [52, 55, 15, 46], whose main focus is on practical space-efficient set representations supporting intersections. Approaches like Optimized PForDelta [53], Roaring Bitmaps [36], SIMD-BP128 [35], and Recursive Universe Partitioning [45] shine in practical scenarios, yet without appealing theoretical guarantees of space usage and intersection computation time. Another relevant approach on these lines is Partitioned Elias-Fano (PEF) [43], able to exploit the distribution and clustering of set elements to improve space usage. Barbay and Kenyon's algorithm can be implemented on PEF, taking $O(\delta \sum_{i \in \mathcal{Q}} \lg(u/n_i))$ time. Regarding space usage, there is no known bound (although it performs well in practice). On a more theoretical track, Bille et al. [14] introduce a data structure that uses $O(mw)$ bits of space and supports intersections in $O(n \lg^2(w)/w + k|\mathcal{I}(\mathcal{Q})|)$ time. Cohen and Porat [18] data structure also uses $O(mw)$ bits of space and allows one to compute the intersection between any two sets in $\mathcal{S}$ in $O(\sqrt{N|\mathcal{I}(\mathcal{Q})|} + \mathcal{I}(\mathcal{Q})|)$ time. Besides using linear space, this approach only works for pair-wise intersections (and is hard to efficiently extend to multiway intersections). Finally, Ding and Konig [21] introduce a data structure able to compute intersections in $O(n/\sqrt{w} + k|\mathcal{I}(\mathcal{Q})|)$ expected time, and uses linear $O(m)$ space. The space can be improved in practice to use about 1.88 times the space of an Elias γ/δ compressed inverted index [21], yet with no theoretical guarantees. Later, Gagie et al. [26] showed that wavelet trees [28] can support intersections in $O(k\delta \lg(u/\delta))$ time, using *uncompressed* $mw(1 + o(1))$ bits of space.

In this paper we show that $O(k\delta \lg(u/\delta))$ intersection time using compressed space is possible. In particular, (1) in Section 3 we revisit a classic (and neglected) algorithm by Trabb-Pardo [50] (former Knuth's student) to prove that its running time is actually $O(k\delta \lg(u/\delta))$ – so it is likely the first adaptive intersection algorithm that ever existed; (2) in Section 4 we show that Trabb-Pardo's algorithm can be implemented in compressed space, yielding an adaptive and compressed set intersection algorithm; (3) in Section 5 we show how to exploit the presence of runs of successive elements, typical in some applications [8],

to formally improve both space usage of input sets and the intersection computation time by introducing an intersection algorithm that runs in time $O(k\xi \lg(u/\xi))$, where $\xi \leq \delta$ is an adaptability measure we introduce; and (4) in Sections 6 and 7 we implement our proposals and show preliminary experimental results that indicate that our approaches are appealing not only in theory, but also in practice, outperforming the most competitive state-of-the-art approaches in some practical inverted-index datasets we use in our tests. Overall, we conclude that both theoretical guarantees and practicality can be achieved with a single approach, which is a step forward in bridging the gap between theory and practice in this important line of research.

2 Preliminaries and Related Work

2.1 Operations rank and select

The following operations on a sorted integer set S are of interest:
- $\mathsf{rank}(S, x)$: for $x \in [0..u)$, yields $|\{y \in S,\ y \leq x\}|$.
- $\mathsf{select}(S, j)$: for $1 \leq j \leq |S|$, yields $x \in S$ s.t. $\mathsf{rank}(S, x) = j$.

A set S can be alternatively described using its *characteristic bit vector* (cbv, for short) $C_S[0..u)$, such that $C_S[x] = \mathbf{1}$ if $x \in S$, $C_S[x] = \mathbf{0}$ otherwise. On a cbv C_S we define:
- $C_S.\mathsf{rank}_\mathbf{1}(x)$: for $x \in [0..u)$, yields the number of $\mathbf{1}$s in $C_S[0..x]$.
- $C_S.\mathsf{select}_\mathbf{1}(k)$: for $1 \leq j \leq |S|$, yields the smallest position $0 \leq x < u$ s.t. $C_S.\mathsf{rank}_\mathbf{1}(x) = j$.

  Notice that $\mathsf{rank}(S, x) \equiv C_S.\mathsf{rank}_\mathbf{1}(x)$ and $\mathsf{select}(S, j) \equiv C_S.\mathsf{select}_\mathbf{1}(j)$.

2.2 Set Compression Measures

A *compression measure* quantifies the amount of bits needed to encode data using a particular compression model. For an integer universe $U = [0..u)$, let $C^{(n)} \subseteq 2^U$, $n \in U$, denote the class of all sets $S \subseteq U$ such that $|S| = n$. We assume $S = \{x_1, \ldots, x_n\}$, for $0 \leq x_1 \leq \cdots \leq x_n < u$. As $|C^{(n)}| = \binom{u}{n}$, in the worst case one needs at least $\mathcal{B}(n, u) = \lceil \lg \binom{u}{n} \rceil$ bits to encode a set $S \in C^{(n)}$. If $n \ll u$, $\mathcal{B}(n, u) = n \lg(u/n) + n \lg e - O(\lg u)$ bits (using Stirling for $n!$). Notice $\mathcal{B}(n, u)$ is a worst-case lower bound: some sets in $C^{(n)}$ can be encoded using less bits, as we shall see.

2.2.1 The $\mathtt{gap}(S)$ Compression Measure

Let us denote $g_1 = x_1$ and, for $i = 2, \ldots, n$, $g_i = x_i - x_{i-1} - 1$. Thus, in the gap model we have $C_S[0..u) = \mathbf{0}^{g_1}\mathbf{1}\mathbf{0}^{g_2}\mathbf{1}\cdots\mathbf{0}^{g_n}\mathbf{1}$ (assuming wlog that C_S ends with $\mathbf{1}$). Then, we define $\mathtt{gap}(S) = \sum_{i=1}^{n} (\lfloor \lg g_i \rfloor + 1)$, as the amount of bits required to represent S provided we encode the sequence of gaps $\mathcal{G} = \langle g_1, \ldots, g_n \rangle$, using $\lfloor \lg g_i \rfloor + 1$ bits per gap. Although this measure is not achievable, it exploits the variation in the gaps between consecutive set elements: the closer the elements, the smallest this measure is. It holds that $\mathtt{gap}(S) \leq n \lg \frac{u}{n}$, with equality only when $g_i = \frac{u}{n}$ (for $i = 1, \ldots, n$). This is a measure traditionally used in applications like inverted-index compression in information retrieval [15] and databases [52].

2.2.2 The $\mathtt{rle}(S)$ Compression Measure

When set elements tend to be clustered into runs of successive elements, a (usually) better way to model its cbv is $C_S[0..u) = \mathbf{0}^{z_1}\mathbf{1}^{\ell_1}\mathbf{0}^{z_2}\mathbf{1}^{\ell_2}\cdots\mathbf{0}^{z_r}\mathbf{1}^{\ell_r}$, where the sequences $\mathcal{Z} = \langle z_1, \ldots, z_r \rangle$ and $\mathcal{O} = \langle \ell_1, \ldots, \ell_r \rangle$ are the lengths of the alternating $\mathbf{0}/\mathbf{1}$-runs in C_S (assume wlog that C_S begins with $\mathbf{0}$ and ends with $\mathbf{1}$). Then, $\mathtt{rle}(S) = \sum_{i=1}^{r} (\lfloor \lg(z_i - 1) \rfloor + 1) + \sum_{i=1}^{r} (\lfloor \lg(\ell_i - 1) \rfloor + 1)$. Unfortunately, $\mathtt{gap}(S)$ and $\mathtt{rle}(S)$ are not comparable measures. If $n < u/2$, it holds that $\mathtt{rle}(S) < \mathcal{B}(n, u) + n + O(1)$ [24].

2.2.3 The $\texttt{trie}(S)$ Compression Measure

Let us consider now representing a set $S \in C^{(n)}$ using a binary trie denoted $\mathsf{bintrie}(S)$, where the $\ell = \lceil \lg u \rceil$-bit binary encoding of every element is added. Each internal node in $\mathsf{bintrie}(S)$ has two children, the left one corresponding to bit **0** and the right one to bit **1**. The external nodes of $\mathsf{bintrie}(S)$ have no children, as usual. In our case, we distinguish two kinds of external nodes. A *void external node* is one whose depth is either $d < \ell$, or alternatively $d = \ell$ yet it represents no element in S. A *valid external node* (or, simply, *external node*, or alternatively a *leaf*), on the other hand, is one whose depth is exactly ℓ and corresponds to an element in S. Thus, $\mathsf{bintrie}(S)$ has $|S|$ valid external nodes, all at depth ℓ. For a leaf v corresponding to element $x_i \in S$, the root-to-v path is hence labeled with the binary encoding of x_i. This approach has been used for representing sets since at least the late 70s by Trabb-Pardo [50]. Consider the example sets $S_1 = \{1, 3, 7, 8, 9, 11, 12\}$, and $S_2 = \{2, 5, 7, 12, 15\}$ over universe $[0..16)$, that we shall use as running examples. Figure 1 shows the corresponding tries $\mathsf{bintrie}(S_1)$ and $\mathsf{bintrie}(S_2)$, with external nodes shown as squares and void external nodes with dotted lines. Interestingly, the following compression

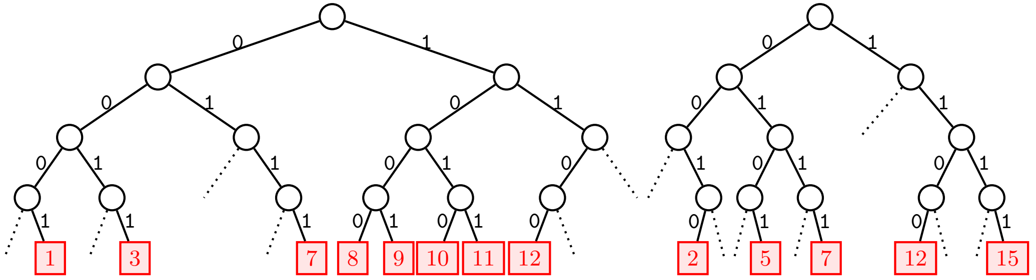

Figure 1 Binary tries $\mathsf{bintrie}(S_1)$ and $\mathsf{bintrie}(S_2)$ encoding sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$. Square nodes at depth 4 in the tries correspond to set elements, whereas dotted lines indicate void external nodes.

measure can be derived from this representation [29]. Given two bit strings x and y of ℓ bits each, let $x \ominus y$ denote the bit string obtained after removing the longest common prefix among x and y from x. For instance, for $x = \texttt{0110100}$ and $y = \texttt{0111011}$, we have $x \ominus y = \texttt{0100}$. The prefix omission method by Klein and Shapira [31] represents a sorted set S as a binary sequence $\mathcal{T} = \langle x_1; x_2 \ominus x_1; \ldots; x_n \ominus x_{n-1} \rangle$. If we denote $|x_i \ominus x_{i-1}|$ the length of bit string $x_i \ominus x_{i-1}$, then the whole sequence uses

$$\texttt{trie}(S) = |x_1| + \sum_{i=2}^{n} |x_i \ominus x_{i-1}|.$$

It turns out that $\texttt{trie}(S)$ is the number of edges in $\mathsf{bintrie}(S)$ [29]. Notice that $\texttt{trie}(S)$ decreases as longer trie paths are shared among set elements: consider two integers x and y, the trie represents their longest common prefix just once (then saving space), and then represents both $x \ominus y$ and $y \ominus x$. Extreme cases are as follows: (1) All set elements form a single run of consecutive elements, which maximizes the number of trie edges shared among set elements, hence minimizing the space usage; and (2) The n elements are uniformly distributed within $[0..u)$ (i.e., the gap between successive elements is $g_i = u/n$), which minimizes the number of trie edges shared among elements, and hence maximizes space usage. Notice this is similar to the case that maximizes the $\texttt{gap}(S)$ measure.

▶ **Definition 1.** *We say that a node v in* $\mathsf{bintrie}(S)$ covers *all leaves that descend from it. In such a case, we call v a* cover node *of the corresponding leaves.*

The following lemma summarizes several results that shall be important for our work:

▶ **Lemma 2** ([26], Lemmas 1–5). *For* bintrie(S)*, the following results hold:*

1. *Any contiguous range of L leaves in* bintrie(S) *is covered by* $\mathrm{O}(\lg L)$ *nodes.*
2. *Any set of r nodes in* bintrie(S) *has* $\mathrm{O}(r \lg \frac{u}{r})$ *ancestors.*
3. *Any set of r nodes in* bintrie(S) *minimally covering a contiguous range of leaves in the trie has* $\mathrm{O}(r + \lg u)$ *ancestors.*
4. *Any set of r nodes in* bintrie(S) *minimally covering L contiguous leaves has* $\mathrm{O}(\lg u + r \lg \frac{L}{r})$ *ancestors.*

▶ **Definition 3.** *Given a set $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$, let $S + a$, for $a \in [0..u)$, denote a shifted version of S: $S + a = \{(x_1 + a) \bmod u, (x_2 + a) \bmod u, \ldots, (x_n + a) \bmod u\}$.*

The following result is relevant for our proposal:

▶ **Lemma 4** ([29], Section 2). *Given a set $S \subseteq [0..u)$ of n elements, it holds that:*

1. $\mathtt{trie}(S) \leq \min\{2\mathtt{gap}(S), n \lg(u/n) + 2n - 2\}$.
2. $\exists a \in [0..u)$*, such that* $\mathtt{trie}(S + a) \leq \mathtt{gap}(S) + 2n - 2$.
3. $\mathtt{trie}(S + a) \leq \mathtt{gap}(S) + 2n - 2$ *on average over all values of* $a \in [0..u)$.

2.3 Adaptive Set Intersection Algorithms

An adaptive algorithm is one whose running time is a function not only of the instance size (as usual), but also of a difficulty measure of the instance. In this way, "easy" instances are solved faster than "difficult" ones, allowing for a more refined analysis than typical worst-case approaches. For the set intersection problem, algorithms by Demaine, López-Ortiz, and Munro [20] and by Barbay and Kenyon [12] are the most important adaptive approaches. To analyze adaptive intersection algorithms, Demaine et al. and Barbay and Kenyon agree in that any algorithm that computes $\mathcal{I}(\mathcal{Q})$ must show a certificate [12] or proof [20] to prove that the intersection is correct. That is, that any element in $\mathcal{I}(\mathcal{Q})$ belongs to the k sets $S_{i_1}, \ldots, S_{i_k}$, and no element in the intersection has been left out of the result. Then, the analysis determines the size of a certificate (or proof) and the time it takes to compute them. In particular, Barbay and Kenyon [12] *partition certificates* are defined as follows.

▶ **Definition 5.** *Given a query $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, a* partition certificate *is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P}_{\mathtt{BK}}(\mathcal{Q}) = \{I_1, I_2, \ldots, I_p\}$, such that:*

1. $\forall x \in \mathcal{I}(\mathcal{Q}), [x..x] \in \mathcal{P}_{\mathtt{BK}}(\mathcal{Q})$*;*
2. $\forall x \notin \mathcal{I}(\mathcal{Q}), \exists I_j \in \mathcal{P}_{\mathtt{BK}}(\mathcal{Q}), x \in I_j \ \wedge \ \exists q \in \mathcal{Q}, S_q \cap I_j = \emptyset$.

For a given query $\mathcal{Q}$, several valid partition certificates could be given. However, we are interested in the smallest partition certificate of $\mathcal{Q}$, as it takes the least time to be computed.

▶ **Definition 6.** *For a given query instance $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, let δ denote the size of the smallest partition certificate of $\mathcal{Q}$.*

Measure δ is known as the *alternation* of the query instance [12], measuring its difficulty. Notice $|\mathcal{I}(\mathcal{Q})| \leq \delta$ holds. Figure 2 shows the smallest partition certificate (of size $\delta = 8$) for sets S_1 and S_2 of our running example. Barbay and Kenyon [11, 12] proved a lower bound of $\Omega(\delta \sum_{i \in \mathcal{Q}} \lg(n_i/\delta))$ comparisons for the set intersection problem. They also gave and optimal intersection algorithm running in $\mathrm{O}(\delta \sum_{i \in \mathcal{Q}} \lg(n_i/\delta))$ time.

S_1 :	1		3		7	8 9 10 11	12	
S_2 :		2		5	7		12	15

Figure 2 Vertical lines show the smallest partition certificate $\mathcal{P} = \{[0..1], [2..2], [3..4], [5..6], [7..7], [8..11], [12..12], [13..15]\}$ of size $\delta = 8$ of the universe $[0..16)$ for the intersection of sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$.

3 Trie Intersection Certificates: A Revisit to Trabb-Pardo Algorithm

In this section we revisit an old divide-and-conquer intersection algorithm by Trabb-Pardo [50], not only to review it but also to prove an adaptive bound on its running time. Algorithm 1 shows the pseudocode. Given a query instance $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the algorithm must

Algorithm 1 `TP-Intersection`(sets $S_1, \ldots, S_k$; universe $[L..R)$).

Result: The set intersection $S_1 \cap \cdots \cap S_k$

1 **begin**
  `// Base cases`
2   **for** $i \leftarrow 1$ to k **do**
3   **if** $S_i = \varnothing$ **then**
4   **return** $\varnothing$
5   **if** $L = R$ **then**
6   **return** $\{L\}$ `// Universe of size 1, all sets are the same singleton`
7   **else**
    `// Divide`
8     $M \leftarrow \lfloor (R + L)/2 \rfloor$
9   **for** $i \leftarrow 1$ to k **do**
10      $S_{i,l} \leftarrow \{x \in S_i \mid x \in [L..M)\}$
11      $S_{i,r} \leftarrow \{x \in S_i \mid x \in [M..R)\}$
    `// Conquer`
12    $R_1 \leftarrow$ `TP-Intersection`$(S_{1,l}, \ldots, S_{k,l}, [L..M))$
13    $R_2 \leftarrow$ `TP-Intersection`$(S_{1,r}, \ldots, S_{k,r}, [M..R))$
    `// Combine`
14   **return** $R_1 \cup R_2$ `// Disjoint set union`

be invoked as `TP-Intersection`$(S_{i_1}, \ldots, S_{i_k}, [0..u))$. The main idea is to divide the universe into two halves, to then split each set according to this universe division. This differs from, e.g., Baeza-Yates's algorithm [9, 10], which splits according to the median of one of the sets. The *Divide* steps (lines 10 and 11) can be implemented using binary search. At the first level of recursion, the most-significant bit of every element in sets $S_{i,l}$ is `0`, as they belong to the left half of the universe. Similarly, for $S_{i,r}$ the most-significant bit is `1` as all elements belong to the right half. At each node of the recursion tree, the current universe is divided into two halves, to then recurse on the sets split accordingly.

As sets are known in advance to queries and set splits carried out by Algorithm 1 depend just on the universe, the Divide step of Algorithm 1 can be implemented efficiently by using a suitable set representation that not only stores the set values, but also precomputes the set splits carried out recursively by the algorithm. Trabb-Pardo proposes to represent each $S_i \in \mathcal{S}$ using $\mathsf{bintrie}(S_i)$, mimicking the way set S_i is recursively split by Algorithm 1. The left child of the root represents all elements whose most-significant bit is `0`, i.e., elements in set $S_{i,l}$ of Algorithm 1 (line 10) in the first level of recursion; similarly for $S_{i,r}$, containing

all elements in S_i whose most-significant bit is 1. To simulate the recursive execution of Algorithm 1 on the binary tries, one must carry out a DFS traversal in synchronization on all tries involved in the query, following the same path in all of them and stopping (and backtracking if needed) as soon as we reach a dead end in one of the tries (which correspond to dotted lines in Figure 1), or we reach a leaf node in all the tries (in which case we have found an element belonging to the intersection). In this way, (1) we stop as soon as we detect a universe interval that does not have any element in the intersection, and (2) we find the relevant elements when we arrive at the leaves.

To analyze Algorithm 1, we introduce the concept of *trie intersection certificates*, denoted $\mathsf{cert}(\mathcal{Q})$, as an alternative to existing certificates [20, 12]. Figure 3 shows a possible $\mathsf{cert}(\mathcal{Q})$ for the intersection of S_1 and S_2 from Figure 1. Let $\mathtt{path}(v)$ denote the binary string labelling the root-to-v path, and $\mathtt{depth}(v) = |\mathtt{path}(v)|$. For a query $\mathcal{Q}$, a binary trie $\mathsf{cert}(\mathcal{Q})$ is a

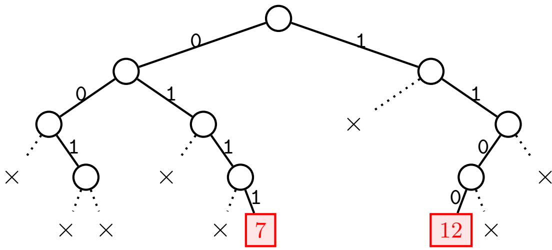

Figure 3 Trie certificate for the intersection $\{1, 3, 7, 8, 9, 10, 11, 12\} \cap \{2, 5, 7, 12, 15\}$. This trie shows the nodes that must be checked to determine that the result is, in this case, $\{7, 12\}$. This corresponds to the intersection of the binary tries representing these sets.

trie partition certificate if: (1) for any internal node v of $\mathsf{cert}(\mathcal{Q})$ such that $\mathtt{path}(v) = b$, there exists an internal node v_i with $\mathtt{path}(v_i) = b$ in every $\mathsf{bintrie}(S_i)$, $i \in \mathcal{Q}$; (2) for any void external node v with $\mathtt{depth}(v) = d \leq \ell = \lceil \lg u \rceil$ and $\mathtt{path}(v) = b$, there exists at least a set S_i ($i \in \mathcal{Q}$) such that there is no node v_i with $\mathtt{path}(v_i) = b$ in $\mathsf{bintrie}(S_i)$. So, the universe interval $[\mathsf{dec}(b \cdot \mathbf{0}^{\ell-d})..\mathsf{dec}(b \cdot \mathbf{1}^{\ell-d})]$ has no element in the intersection, where $\mathsf{dec}(x)$ denotes the decimal representation of a binary string x. We call them *fail nodes*, shown as "$\times$" in Figure 3; and (3) for any valid external node v with $\mathtt{depth}(v) = \ell$ corresponding to $\mathtt{path}(v) = b$, there exists a valid external node v_i with $\mathtt{path}(v_i) = b$ in every $\mathsf{bintrie}(S_i)$, $i \in \mathcal{Q}$. We call them *success nodes* as they correspond to elements in $\mathcal{I}(\mathcal{Q})$.

Notice that $\mathsf{cert}(\mathcal{Q})$ is the trie obtained by intersecting $\mathsf{bintrie}(S_i)$, for all $i \in \mathcal{Q}$, and that it is the smallest trie that allows us to prove the correctness of the intersection. For instance, Figure 3 shows $\mathsf{cert}(\mathcal{Q})$ for a given query. Interestingly, the recursion tree of Algorithm 1 is exactly $\mathsf{cert}(\mathcal{Q})$, as the algorithm stops as soon as one arrives at a fail node. The external nodes of $\mathsf{cert}(\mathcal{Q})$ cover the universe $[0..u)$ with intervals, similar to Barbay and Kenyon partition certificates, as stated by the following definition.

▶ **Definition 7.** *Given a query $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, its* trie partition certificate *is a partition of the universe $[0..u)$ into a set of intervals, we say that $\mathsf{cert}(\mathcal{Q})$ induces the following partition of $[0..u)$ into a set of intervals that we call* trie partition certificate*:*

$$\mathcal{P}_{\mathtt{AC}}(\mathcal{Q}) = \bigcup_{l \in \mathcal{E}(\mathsf{cert}(\mathcal{Q}))} \left\{ [\mathsf{dec}(\mathtt{path}(l) \cdot \mathbf{0}^{\ell - \mathtt{depth}(l)})..\mathsf{dec}(\mathtt{path}(l) \cdot \mathbf{1}^{\ell - \mathtt{depth}(l)})] \right\},$$

where $\mathcal{E}(\mathsf{cert}(\mathcal{Q}))$ denotes the set of external nodes of $\mathsf{cert}(\mathcal{Q})$.

For instance, the trie certificate of Figure 3 induces the following trie partition certificate of the universe $[0..16)$: $\{[0..1], [2..2], [3..3], [4..5], [6..6], [7..7], [8..11], [12..12], [13..13], [14..15]\}$.

Since $\mathsf{cert}(\mathcal{Q})$ is the recursion tree of Algorithm 1, its running time is $\mathrm{O}(k|\mathsf{cert}(\mathcal{Q})|)$. As in the worst-case one must traverse completely all tries $\mathsf{bintrie}(S_i)$, $i \in \mathcal{Q}$, we have:

$$k|\mathsf{cert}(\mathcal{Q})| \leq \sum_{i\in\mathcal{Q}} \mathtt{trie}(S_i) \leq \sum_{i\in\mathcal{Q}} n_i \lg \frac{u}{n_i} + 2n_i - 2,$$

where the last bound is from Lemma 4 (1). Next we prove an adaptive bound for $k|\mathsf{cert}(\mathcal{Q})|$.

▶ **Theorem 8.** *Given a query instance $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$ with alternation measure δ and over sets with universe $[0..u)$, algorithm `TP-Intersection` computes $\mathcal{I}(\mathcal{Q}) = \cap_{i\in\mathcal{Q}} S_i$ in time $\mathrm{O}(k\delta \lg(u/\delta))$.*

Proof. Consider a smallest partition certificate $\mathcal{P}_{\mathtt{BK}}(\mathcal{Q}) = \{I_1, \ldots, I_\delta\}$ of universe $[0..u)$, such that $|I_i| = L_i$ for $i = 1, \ldots, \delta$. Let us think now of the worst-case smallest $\mathsf{cert}(\mathcal{Q})$ we could have, by covering the δ intervals in $\mathcal{P}_{\mathtt{BK}}(\mathcal{Q})$ with as many external nodes of $\mathsf{cert}(\mathcal{Q})$ as possible. For any $I_j \in \mathcal{P}_{\mathtt{BK}}(\mathcal{Q})$ formed by elements not in $\mathcal{I}(\mathcal{Q})$, there exists a set of external fail nodes in $\mathsf{cert}(\mathcal{Q})$ that cover I_j. This is because when traversing the tries $\mathsf{bintrie}(S_i)$ in coordination, $i \in \mathcal{Q}$, the algorithm stops as long as one gets into one of the cover nodes of I_j, since it does not belong to at least one of the tries. According to Lemma 2 (1), a contiguous range of L leaves (corresponding to the values in I_j) can be covered with up to $\mathrm{O}(\lg L)$ nodes. Thus, in the worst-case, $\mathsf{cert}(\mathcal{Q})$ has $\mathrm{O}(\sum_{i=1}^{\delta} \lg L_i)$ external nodes that overall cover $[0..u)$. Now, recall that the external nodes of $\mathsf{cert}(\mathcal{Q})$ cover the contiguous range of leaves corresponding to $[0..u)$. Hence, according to Lemma 2 (3), these external nodes have $\mathrm{O}(\sum_{i=1}^{\delta} \lg L_i + \lg u)$ ancestors, so overall $\mathsf{cert}(\mathcal{Q})$ has $\mathrm{O}(\sum_{i=1}^{\delta} \lg L_i + \lg u)$ nodes. The sum is maximized when $L_i = u/\delta$, for all $1 \leq i \leq \delta$, hence $\mathsf{cert}(\mathcal{Q})$ has $\mathrm{O}(\delta \lg (u/\delta))$ nodes. The result follows from the fact that for each node in $\mathsf{cert}(\mathcal{Q})$ the algorithm runs in time $\mathrm{O}(k)$. ◀

4 Compressed Intersectable Sets

We devise next a space-efficient representation of $\mathsf{bintrie}(S)$, for a set $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$ of n elements such that $0 \leq x_1 < \cdots < x_n < u$. This representation will also allow for efficient intersections, supporting Trabb-Pardo's [50] algorithm.

We represent $\mathsf{bintrie}(S)$ level-wise [30]. Let $B_1[1..2l_1], \ldots, B_\ell[1..2l_\ell]$ be bit vectors such that B_i represents the l_i nodes at level i of $\mathsf{bintrie}(S)$ ($1 \leq i \leq \ell$), from left to right. Each node is encoded using 2 bits, indicating the presence (using bit **1**) or absence (bit **0**) of the left and right children, respectively. In this way, the feasible codewords for trie nodes are **01**, **10**, and **11**, whereas **00** is not a valid codeword. The codewords of all nodes at level $i \geq 1$ in the trie are concatenated from left to right to form B_i. The j-th node at level i (from left to right) is stored at positions $2j - 1$ and $2j$. We say that $2j - 1$ is the position of such node in B_i.

Let p be the position in B_i corresponding to a node v at level i of $\mathsf{bintrie}(S)$. As the nodes are stored level-wise and from left to right, the number of **1**s before position p in B_i equals the number of nodes in level $i + 1$ that are before the child(ren) of node v. So, $2B_i.\mathsf{rank}_{\mathbf{1}}(p - 1) + 1$ yields the position of B_{i+i} where the first child of node v is. Figure 4 illustrates our representation.

The total number of **1**s in the bit vectors of our representation equals the number of edges in the trie. That is, there are $\mathtt{trie}(S)$ **1**s. Besides, the trie has $\mathtt{trie}(S) + 1$ internal nodes and leaves: n of them are leaves, so $\mathtt{trie}(S) - n + 1$ are internal. In our representation we only need to represent the internal trie nodes. As we encode each node using 2 bits, the total space usage for $B_1, \ldots, B_\ell$ is $2(\mathtt{trie}(S) - n + 1)$ bits. On top of them we use Clark's data structure [16] to support rank in $\mathrm{O}(1)$ time, adding $o(\mathtt{trie}(S))$ extra bits overall.

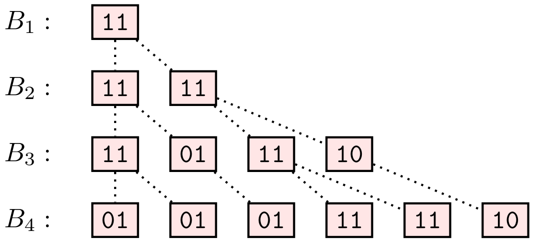

Figure 4 Level-wise bit vector representation of $\mathsf{bintrie}(S)$ for $S = \{1, 3, 7, 8, 9, 10, 11, 12\}$. Dotted lines are implicit, as they are computed using operation $\mathsf{rank}_\mathbf{1}$ on the bit vectors.

Given a query $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, we traverse $\mathsf{bintrie}(S_{i_1}), \ldots, \mathsf{bintrie}(S_{i_k})$ using a recursive DFS traversal as in Algorithm 1. Besides the query itself, our algorithm receives: (1) an integer value, *level*, indicating the current recursion level, and (2) integer values $r_1, \ldots, r_k$, indicating the current nodes in each trie, represented as the positions of these nodes within B_{level}. Algorithm 2 shows the pseudo-code of our adaptive and compressed algorithm to compute the compact representation for $\mathsf{bintrie}(\mathcal{I}(\mathcal{Q}))$ (denoted T_I in the pseudocode). The algorithm uses a binary variable s, initialized with **11**, which stores the bitwise-and of all current node codewords (line 4). So, $s =$ **00** means that recursion must stop, $s =$ **10** indicates to go down only to the left, $s =$ **01** just to the right, and $s =$ **11** to both children.

Lines 9–13 carry out the needed computation to go down to the left child. In particular, we compute the positions of the left-subtrie roots using $\mathsf{rank}_\mathbf{1}$ operation. Then, in line 13 we recursively go down to the left. The result of that recursion in stored in variable *lChild*, indicating with a **1** that the left recursion yielded a non-empty intersection, **0** otherwise. A similar procedure is carried out for the right child in lines 14–21. Line 17 determines whether we have already computed the $\mathsf{rank}_\mathbf{1}$s corresponding to the left child. If that is not the case, we compute them in line 18. In this way, we compute only one $\mathsf{rank}_\mathbf{1}$ operation per traversed node in the tries, which is important in practice. Just as for the left child, we store the result of the right-child recursion in variable *rChild* in line 21. Finally, in line 22 we determine whether the left and right recursions yielded an empty intersection or not. If both $lChild = rChild = 0$, the intersection was empty on both children, so we return **0**. Otherwise, we append *lChild* and *rChild* to $T_I.B_{level}$, as that is the codeword of the corresponding node in T_I. Note how we actually generate the output trie T_I in postorder, after we visited both children of the current nodes, despite the input is traversed in preorder. Thus, we write the output in time proportional to its size. Although the total running time is still proportional to $|\mathsf{cert}(\mathcal{Q})|$, this can save important time in practice.

Besides computing $\mathcal{I}(\mathcal{Q})$, a distinctive feature of our algorithm is that it also allows one to obtain the sequence $\langle \mathsf{rank}(S_{i_1}, x), \ldots, \mathsf{rank}(S_{i_k}, x)\rangle$, for all $x \in \mathcal{I}(\mathcal{Q})$, for free (in asymptotic terms). The idea is to compute $\langle \mathsf{bintrie}(S_{i_1}).B_\ell.\mathsf{rank}_\mathbf{1}(r_1), \ldots, \mathsf{bintrie}(S_{i_k}).B_\ell.\mathsf{rank}_\mathbf{1}(r_k)\rangle$ every time the recursion reaches level ℓ (i.e., just before the **return** of line 7 in Algorithm 2). Outputting this information is important for several applications, such as cases where set elements have satellite data associated to them. For an element $x_j \in S_i$, the associated data d_j is stored in an auxiliary array $D_i[1..n_i]$ such that $D[\mathsf{rank}(S_i, x_j)] = d_j$. Typical applications are inverted indexes in IR (where ranking information, such as frequencies, is associated to inverted list elements), and the Leapfrog Triejoin algorithm [51] (where at each step we must compute the intersection of sets, and for each element in the intersection we must go down following a pointer associated to it).

Algorithm 2 `AC-Intersection`(query $\mathcal{Q}$; roots $r_1, \ldots, r_k$; *level*).

```
   Result: The binary trie T_I representing I(Q) = ∩_{i∈Q} S_i
 1 begin
 2    s ← 11 // binary encoding
 3    for i ∈ Q do
 4        s ← s & (bintrie(S_i).B_level [r_i] · bintrie(S_i).B_level [r_i + 1])
 5    if level = ℓ then
 6        append s to T_I.B_ℓ
 7        return 1
 8    lChild ← 0; rChild ← 0
      // Go down to the left in the tries
 9    if s is 10 or 11 then
10        lRoots ← ∅
11        for i ∈ Q do
12            lRoots ← lRoots ∪ {2 × bintrie(S_i).B_level.rank_1(r_i − 1) + 1}
13        lChild ← AC-Intersection(Q, lRoots, level + 1)
      // Go down to the right in the tries
14    if s is 01 or 11 then
15        rRoots ← ∅
16        for i ∈ Q do
17            if s = 01 then
18                rRoots ← rRoots ∪ {2 × bintrie(S_i).B_level.rank_1(r_i − 1) + 2}
19            else
20                rRoots ← rRoots ∪ {lRoots_i + 2}
21        rChild ← AC-Intersection(Q, rRoots, level + 1)
      // Output written in postorder
22    if lChild ≠ 0 or rChild ≠ 0 then
23        append lChild · rChild to T_I.B_level
24        return 1
25    else
26        return 0
```

We have proved the following theorem:

▶ **Theorem 9.** *Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u)$. There exists a data structure able to represent each set S_i using $2(\texttt{trie}(S_i) - n_i + 1) + o(\texttt{trie}(S_i))$ bits, such that given a query $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(\mathcal{Q}) = \cap_{i \in \mathcal{Q}} S_i$ can be computed in $\mathrm{O}(k\delta \lg (u/\delta))$ time, where δ is the alternation measure of $\mathcal{Q}$. Besides, for every $x \in \mathcal{I}(\mathcal{Q})$, the data structure also allows one to obtain the sequence $\langle \mathsf{rank}(S_{i_1}, x), \ldots, \mathsf{rank}(S_{i_k}, x)\rangle$ asymptotically for free.*

5 Compressing Runs of Elements

Next, we exploit the presence of runs of successive elements in the input sets to reduce both the space usage of the binary trie representation, as well as intersection time. Runs tend to be captured by full subtrees in the corresponding binary tries. See, e.g., the full subtree whose leaves correspond to elements $8, 9, 10, 11$ in the binary trie of Figure 5. Let v be a $\mathsf{bintrie}(S)$ node whose subtree is full. Let $\texttt{depth}(v) = d$. If $b = \texttt{path}(v)$, the $2^{\ell-d}$ leaves

covered by v correspond to the integer interval $[\mathsf{dec}(b \cdot \mathbf{0}^{\ell-d})..\mathsf{dec}(b \cdot \mathbf{1}^{\ell-d})]$. So, the subtree of v can be removed, keeping just v, saving space and still being able to recover the removed elements.

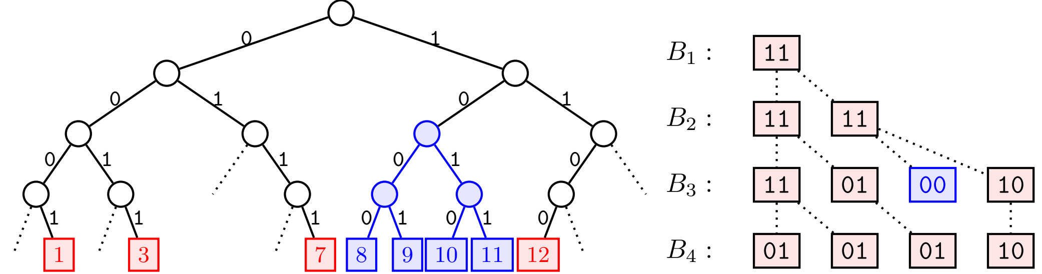

Figure 5 Left side, the binary trie representing set $\{1, 3, 7, 8, 9, 10, 11, 12\}$. Notice that the subtree whose leaves correspond to elements $8, 9, 10, 11$ is a full subtree. Right side, our compact representation removing full subtrees and encoding their roots with **00**.

▶ **Definition 10.** *Let $S \subseteq [0..u)$ be a set of n elements. We define $\texttt{rTrie}(S)$ as the number of edges in $\mathsf{bintrie}(S)$ after removing the maximal full subtrees.*

This immediately implies $\texttt{rTrie}(S) \leq \texttt{trie}(S) \leq 2\texttt{gap}(S)$, yet we can prove tighter bounds. Assume a set S with r runs of $\ell_1, \ldots, \ell_r$ successive elements each, respectively. The ℓ_i elements of a given run correspond to ℓ_i contiguous leaves in $\mathsf{bintrie}(S)$ which, according to Lemma 2 (item 1), are covered by at most $2\lfloor \lg(\ell_i/2) \rfloor$ nodes. This is a pessimistic case that removes the least edges, so we analyze it. Among the cover nodes, there are 2 whose subtrees have 0 edges, 2 whose subtrees have 2 edges, 2 whose subtrees have 6 edges, and so on. In general, for each $i = 1, \ldots, \lfloor \lg(\ell_i/2) \rfloor$, there are 2 cover nodes whose subtrees have $2^i - 2$ edges. If we remove them all, the total number of edges removed is $2\sum_{i=i}^{\lfloor \lg(\ell_i/2) \rfloor}(2^i - 2) \leq 2\ell_i - 4\lg \ell_i$. This removes the least edges belonging to full subtrees, so we can bound

$$\texttt{rTrie}(S) \leq \texttt{trie}(S) - \sum_{i=1}^{r}(2\ell_i - 4\lg \ell_i). \tag{1}$$

We can also prove the following bounds.

▶ **Lemma 11.** *Given a set $S \subseteq [0..u)$ of n elements, it holds that*

1. $\texttt{rTrie}(S) \leq 2 \cdot \min\{\texttt{rle}(S) + \sum_{i=1}^{r} \lg \ell_i,\ \texttt{gap}(S)\}$.
2. $\exists a \in [0..u)$, *such that* $\texttt{rTrie}(S + a) \leq \min\{\texttt{rle}(S) - \sum_{i=1}^{r} \ell_i + 3\sum_{i=1}^{r} \lg \ell_i,\ \texttt{gap}(S)\} + 2n - 2$.
3. $\texttt{rTrie}(S + a) \leq \min\{\texttt{rle}(S) - \sum_{i=1}^{r} \ell_i + 3\sum_{i=1}^{r} \lg \ell_i,\ \texttt{gap}(S)\} + 2n - 2$ *on average, assuming $a \in [0..u)$ is chosen uniformly at random.*

Proof. Since S has r runs of $\ell_1, \ldots, \ell_r$ elements, we can rewrite $\texttt{gap}(S) = \sum_{i=1}^{r}(\lfloor \lg(z_i - 1) \rfloor + 1) + \sum_{i=1}^{r}(\ell_i - 1)$. As $\texttt{rTrie}(S) \leq \texttt{trie}(S) \leq 2\texttt{gap}(S)$, and $\texttt{rTrie}(S) \leq \texttt{trie}(S) - \sum_{i=1}^{r}(2\ell_i - 4\lg \ell_i)$ (Equation 1), it holds that

$$\begin{aligned}
\texttt{rTrie}(S) &\leq \texttt{trie}(S) - \sum_{i=1}^{r} (2\ell_i - 4\lg \ell_i) \\
&\leq 2(\sum_{i=1}^{r} (\lfloor \lg (z_i - 1) \rfloor + 1) + \sum_{i=1}^{r} (\ell_i - 1)) - \sum_{i=1}^{r} (2\ell_i - 4\lg \ell_i) \\
&= 2\sum_{i=1}^{r} (\lfloor \lg (z_i - 1) \rfloor + 1) + 4\sum_{i=1}^{r} \lg \ell_i = 2(\texttt{rle}(S) + \sum_{i=1}^{r} \lg \ell_i),
\end{aligned}$$

proving item 1. Items 2 and 3 can be proved similarly from items 2 and 3 of Lemma 4. ◀

In our compact representation, we encode a full-subtree cover node using **00**. Recall that **00** is an invalid codeword, so we use it as a special mark. See Figure 5 for an illustration.

Given a query $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the procedure to compute $\mathcal{I}(\mathcal{Q})$ is similar to that of Algorithm 2. The only difference is that if in a given trie $\mathsf{bintrie}(S_i)$ we arrive at a node encoded **00**, every possible set element in the subtrie of the node belongs to S_i. In other words, the intersection within the current subtries is independent of S_i, so we can safely temporarily exclude $\mathsf{bintrie}(S_i)$ from the intersection and continue intersecting the remaining tries. To implement this idea, we keep boolean flags $f_1, \ldots, f_k$ such that f_j corresponds to $\mathsf{bintrie}(S_{i_j})$. The idea is that at each point during the synchronized DFS traversal, only tries whose flag is true participate in the intersection. Initially, we set $f_i \leftarrow \mathsf{true}$, for $1 \leq i \leq k$. If, during the intersection process, we arrive at a node encoded **00** in $\mathsf{bintrie}(S_i)$, we set $f_i \leftarrow \mathsf{false}$. When the recursion at a node encoded **00** in $\mathsf{bintrie}(S_i)$ finishes, we set $f_i \leftarrow \mathsf{true}$ again. If, at a given point, all tries have been temporarily excluded but one, let us say $\mathsf{bintrie}(S_j)$, we only need to traverse the current subtree in S_j, copying it verbatim to the output. If this subtree contains nodes encoded **00**, they will appear in the output. This way, the maximal runs of successive elements in the output will be covered by nodes encoded **00**. This fact is key for the adaptive running time of our algorithm, as we shall see below.

We analyze our algorithm introducing the following variant of partition certificates.

▶ **Definition 12.** *Given a query instance $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, a run-partition certificate for it is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P}^r_{\texttt{AC}}(\mathcal{Q}) = \{I_1, I_2, \ldots, I_p\}$, such that the following conditions hold:*

1. $\forall x \in \mathcal{I}(\mathcal{Q}), \exists I_j \in \mathcal{P}^r_{\texttt{AC}}(\mathcal{Q})$, *such that* $x \in I_j \wedge \mathcal{I}(\mathcal{Q}) \cap I_j = I_j$*;*
2. $\forall x \notin \mathcal{I}(\mathcal{Q}), \exists I_j \in \mathcal{P}^r_{\texttt{AC}}(\mathcal{Q})$, *such that* $x \in I_j \;\wedge\; \exists q \in \mathcal{Q}, S_q \cap I_j = \emptyset$.

Let ξ denote the size of the smallest run-partition certificate $\mathcal{P}^r_{\texttt{AC}}(\mathcal{Q})$ of $\mathcal{Q}$. We call ξ the run alternation measure.

Item 2 is the same as for Barbay and Kenyon's partition certificates, corresponding to intervals of elements not in $\mathcal{I}(\mathcal{Q})$. Item 1, on the other hand, corresponds to elements in $\mathcal{I}(\mathcal{Q})$ which, unlike Barbay and Kenyon certificates, are not necessarily covered by singletons: our definition allows one to cover a run of successive elements in $\mathcal{I}(\mathcal{Q})$ using a single interval. Clearly, $\xi \leq \delta$ holds. Besides, although $|\mathcal{I}(\mathcal{Q})| \leq \delta$ holds, in our case there can be query instances such that $\xi < |\mathcal{I}(\mathcal{Q})|$. Figure 6 illustrates our definition for an intersection of 4 sets on the universe $[0..15)$. Notice that $\xi = 5$, whereas $|\mathcal{I}(\mathcal{Q})| = 6$ and $\delta = 9$.

We must also introduce a fourth type of node to our trie certificate definition of Section 3. If for an internal node v of $\mathsf{cert}(\mathcal{Q})$ with $\texttt{path}(v) = b$, it holds that there is a node v_i with $\texttt{path}(v_i) = b$ in every $\mathsf{bintrie}(S_i)$, $i \in \mathcal{Q}$, and the subtries of all v_is is full, then v is called an *internal success node*. It is important to note that every interval I_j from item 1 of Definition 12 is covered only by internal success nodes. Also, internal success nodes only cover intervals from item 1 of Definition 12.

S_{i_1} :				7	8	9	10	11	12	13	14	15
S_{i_2} :		5	6	7	8	9	10	11	12	13	14	
S_{i_3} :	4	5	6	7	8	9		11	12	13	14	
S_{i_4} :					8	9	10	11	12	13	14	15

Figure 6 A query instance $\mathcal{Q} = \{S_{i_1}, S_{i_2}, S_{i_3}, S_{i_4}\}$ and its smallest run-partition certificate $\mathcal{P}^r_{\texttt{AC}}(\mathcal{Q}) = \{[0..7], [8..9], [10..10], [11..14], [15..15]\}$ of size $\xi = 5$.

Our main result is stated in the following theorem:

▶ **Theorem 13.** *Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u)$. There exists a data structure able to represent each set S_i using $2\texttt{rTrie}(S_i)(1 + o(\texttt{rTrie}(S_i)))$ bits, such that given a query $\mathcal{Q} = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(\mathcal{Q}) = \cap_{i \in \mathcal{Q}} S_i$ can be computed in $\mathrm{O}(k\xi \lg (u/\xi))$ time, where ξ is the run alternation measure of $\mathcal{Q}$.*

Proof. Consider the smallest run-partition certificate $\mathcal{P}^r_{\texttt{AC}}(\mathcal{Q}) = \{I_1, \ldots, I_\xi\}$ of universe $[0..u)$, such that $|I_i| = L_i$ for $i = 1, \ldots, \xi$. Let us cover these ξ intervals with as many nodes of the smallest $\mathsf{cert}(\mathcal{Q})$ as possible. As we already saw in the proof of Theorem 8, all intervals I_i such that $I_i \cap \mathcal{I}(\mathcal{Q}) = \emptyset$ are covered by at most $\mathrm{O}(\lg L_i)$ nodes in $\mathsf{cert}(\mathcal{Q})$. We now prove the same for intervals $I_j \subseteq \mathcal{I}(\mathcal{Q})$, which are covered by internal success nodes of $\mathsf{cert}(\mathcal{Q})$. The only thing to note is that our algorithm stops as soon as it arrives to an internal success node. As there can be $\mathrm{O}(\lg L_j)$ such cover nodes, universe $[0..u)$ can be covered by $\mathrm{O}(\lg u + \sum_{i=1}^{\xi} \lg L_i) = \mathrm{O}(\lg u + \sum_{i=1}^{\xi} \lg(u/\xi))$ nodes, hence $\mathsf{cert}(\mathcal{Q})$ has $\mathrm{O}(\xi \lg(u/\xi))$ nodes overall. The result follows from the fact that at each node the algorithm runs in time $\mathrm{O}(k)$. ◀

6 Implementation

We implemented bit vectors $B_1, \ldots, B_\ell$ in plain form using class `bit_vector<>` from the `sdsl` library [27]. We support $\mathsf{rank_1}$ on them using different data structures to obtain the following schemes. (`trie v`, `rTrie v`): the variants defined in Section 4 and 5, respectively, using `rank_support_v` for $\mathsf{rank_1}$. It uses $\sim$25% extra space on top of the bit vector, supporting $\mathsf{rank_1}$ in $\mathrm{O}(1)$ time. (`trie v5`, `rTrie v5`): use `rank_support_v5`, requiring $\sim$6.25% extra space on top of the bit vectors, supporting $\mathsf{rank_1}$ in $\mathrm{O}(1)$ time. This alternative is smaller, yet slower in practice. (`trie IL`, `rTrie IL`): use `rank_support_il`, aiming at reducing the number of cache misses to compute $\mathsf{rank_1}$. We use block size 512, requiring $\sim$12.5% extra space on top of the bit vectors, while supporting $\mathsf{rank_1}$ in $\mathrm{O}(1)$ time.

Most state-of-the-art alternatives we compare with do not support operation $\mathsf{rank}(S, x)$. So, to be fair, we do not store any $\mathsf{rank_1}$ data structure for the last-level bit vector B_ℓ. Recall that $\mathsf{rank}(S, x)$ is equivalent to a $\mathsf{rank_1}$ on the corresponding position of B_ℓ. We implemented Algorithm 2 on our compact trie data structures, following the descriptions from Sections 4 and 5 very closely. We implemented, however, two alternatives for representing the output: (1) the binary trie representation, and (2) the plain array representation. In our experiments we will use the latter, to be fair: all testes alternatives produce their outputs in plain form.

We also implemented a simple multithreaded version of our algorithm. Let t denote the number of available threads. Then, we define $c = \lfloor \lg t \rfloor$. Our algorithm proceeds as in Algorithm 2, generating a binary trie of height c (that we will call *top trie*), with at most t leaves. Then, we execute Algorithm 2 again, this time in parallel, with each thread

starting from a different leaf of the top trie. Each thread generates its own output in parallel, using our compact trie representation. Once all threads finish, we concatenate these tries to generate the final output. We just need to count, in parallel, how many nodes there are in each level of the trie. Then, we allocate a bit vector of the appropriate size for each level, where each thread will write its own part of the output in parallel. This simple approach does not guarantees load balancing among threads, however it works relatively well in practice.

Our source code and instructions to replicate our experiments are available at `https://github.com/jpcastillog/compressed-binary-tries`.

7 Experimental Results

We experimentally evaluate our approaches on a server with an i7 10700k CPU, 8 cores and 16 threads at 4.70 GHz, 32 GB of RAM (DDR4-3.6GHz) running in dual channel, and Ubuntu 20.04 LTS OS. Our implementation is developed in C++, compiled with `g++` 9.3.0 and optimization flags `-O3` and `-march=native`.

In our tests, we used families of sets corresponding to inverted indexes of three standard document collections: Gov2 [17], ClueWeb09 [1], and CC-News [38]. For Gov2 and ClueWeb09 collections, we used the freely-available inverted indexes and query logs by Daniel Lemire (see [34] for details), corresponding to the URL-sorted document enumeration [48] (which tends to yield runs of successive elements in the sets). The query log contains 20,000 random queries from the TREC million-query track (1MQ). Each query has at least 2 query terms. Also, each term is in the top-1M most frequently queried terms. For CC-News we use the freely-available inverted index by Mackenzie et al. [38] in *Common Index File Format* (CIFF) [37], as well as their query log of 9,666 queries. Table 1 shows a summary of statistics of the collections. In all cases, we only keep sets with at least 4,096 elements.

Table 1 Dataset summary and average space usage (in bits per integer, bpi) for different compression measures and baseline representations.

	Gov2	ClueWeb09	CC-News
# Lists	57,225	131,567	79,831
# Integers	5,509,206,378	14,895,136.282	18,415,151,585
u	25,205,179	50,220,423	43,495,426
$\lceil \lg u \rceil$	25	26	26
$\mathtt{gap}(S)$	2.25	3.25	3,70
$\mathtt{rle}(S)$	1.99	3.33	4,23
$\mathtt{trie}(S)$	3.48	4.56	5,18
$\mathtt{rTrie}(S)$	2.51	4.00	5,12
Elias γ	3.71	5.74	6.81
Elias δ	3.64	5.40	6.69
Fibonacci	3.90	5.35	6.09
Elias γ 128	4.07	6.10	7.05
Elias δ 128	4.00	5.77	7.17
Fibonacci 128	4.26	5.71	6.45
`rrr_vector<>`	11.82	19.94	11.29
`sd_vector<>`	8.45	8.52	7.17

As baseline, Table 1 also shows the average bit per integer (bpi) for different compression measures on our tested set collections. We also show the average bpi for different integer compression approaches, namely Elias γ and δ [22], Fibonacci [25], `rrr_vector<>` [47], and `sd_vector<>` [42], all of them from the `sdsl` library [27]. In particular, Elias γ, δ, and Fibonacci codes are known for yielding highly space-efficient set representations in IR indexing [15], hence they are a strong baselines for comparison. We show a plain version of them, as well as variants with blocks of 128 integers. The latter are needed to speed up decoding. However, these approaches are relatively slow to be decoded [15, See Table 6.9], and hence yield higher intersection times. On the other hand, `sd_vector<>` uses $n \lg (u/n) + 2n + o(n)$ bits to encode a set of n elements and universe $[0..u)$. Finally, `rrr_vector<>` uses $\mathcal{B}(n, u) + o(u)$ bits of space. As it can be seen, the $o(u)$-bit term yields a higher space usage.

Next, we compare our approaches with state-of-the-art set compression alternatives available at the project *Performant Indexes and Search for Academia* [1] (PISA) [39]:

- `IPC`: the Binary Interpolative Coding approach by Moffat et al. [40]. This is a highly space-efficient approach, with a relatively slow processing performance [15, 40].
- `PEF Opt`: the highly competitive approach by Ottaviano and Venturini [43].
- `OptPFD`: The Optimized PForDelta approach by Yan et al. [53].
- `SIMD-BP128`: The highly efficient approach by Lemire and Boytsov [35], aimed at decoding billions of integers per second using vectorization capabilities of modern processors.
- `Simple16`: The approach by Zhang at al. [54], a variant of the `Simple9` approach [5] that combines a relatively good space usage and an efficient intersection time.
- `VarintGB`: The approach used in Google and presented by Dean [19].
- `Varint-G8IU`: by Stepanov et al. [49], using SIMD instructions to speed-up set processing.

We also compared with the following approaches, available from their authors:

- `Roaring`: the compressed bitmap approach by Lemire et al. [36], widely used as indexing tool on several systems and platforms [3]. Roaring bitmaps are highly competitive, leveraging modern CPU hardware architectures. We use the code from the authors [2].
- `RUP`: The recent recursive universe partitioning approach by Pibiri [45], using also SIMD instructions to speed up processing. We use the code from the author [44].

Table 2 shows the average experimental intersection time (in milliseconds per query) and space usage (in bits per integer) for all the alternatives tested. Figure 7 (in the Appendix) shows the same results, using space vs. time plots. Our approaches introduce competitive trade-offs, as follows:

Results for Gov2: `rTrie` uses 1.166–1.329 times the space of `PEF`, the former being 1.549–2.442 times faster. `rTrie` uses 0.481–0.548 times the space of `Roaring`, the former being up to 1.415 times faster. Finally, `rTrie` uses 0.837–0.954 times the space of `RUP`, the former being up to 1.428 times faster.

Results for ClueWeb09: `rTrie` uses 1.188–1.361 times the space of `PEF`, the former being 2.117–3.316 times faster. Also, `rTrie` uses 0.551–0.631 times the space of `Roaring`, the former being 1.221–1.913 times faster. Finally, `rTrie` uses 0.823–0.943 times the space of `RUP`, the former being 1.391–2.178 times faster.

Results for CC-News: for this dataset, the resulting inverted lists have considerably less runs, hence the space usage of `trie` and `rTrie` are about the same. However, `trie` is faster than `rTrie`, as the code to handle runs introduces an overhead that does not pay

[1] `https://github.com/pisa-engine/pisa`

Table 2 Average intersection time (milliseconds per query) and space usage (in bits per integer) for all alternatives tested.

	Gov2		ClueWeb09		CC-News	
Data Structure	Space	Time	Space	Time	Space	Time
`IPC`	3.34	8.66	5.15	30.18	5.87	68.98
`Simple16`	4.65	2.44	6.72	8.66	6.88	19.74
`OptPFD`	4.07	2.15	6.28	7.79	6.50	11.80
`PEF Opt`	3.62	1.88	5.85	6.50	5.80	17.33
`VarintGB`	10.80	1.43	11.40	7.34	11.04	12.38
`Varint-G8IU`	9.97	1.38	10.55	5.25	10.24	12.09
`SIMD-BP128`	6.07	1.29	8.98	4.47	7.36	15.90
`Roaring`	8.77	1.09	12.62	3.75	9.86	5.56
`RUP`	5.04	1.10	8.44	4.27	8.41	5.44
`trie` (v5)	5.18	1.21	7.46	2.81	8.77	8.72
`trie` (IL)	5.41	1.06	7.83	2.42	9.30	7.46
`trie` (v)	5.85	0.77	8.50	1.64	9.99	5.21
`rTrie` (v5)	4.22	1.22	6.95	3.07	8.73	9.74
`rTrie` (IL)	4.42	1.10	7.31	2.62	9.16	8.13
`rTrie` (v)	4.81	0.77	7.96	1.96	9.95	6.09

off in this case. So, we will use `trie` to compare here. It uses 1.512–1.722 times the space of `PEF`, the former being 1.987–3.326 times faster. `trie` uses 0.889–1.013 times the space of `Roaring`, the former being up to 1.067 times faster. Finally, `trie` uses 1.043–1.188 times the space of `RUP`, the former being up to 1.044 times faster.

We can conclude that in all tested datasets, at least one of our trade-offs is the fastest and competitive in space usage, outperforming the highly-engineered ultra-efficient set compression techniques we tested.

8 Conclusions

Trie partition certificates, the main concept we introduced as an alternative to existing certificates by Demaine et al. [20] and Barbay and Kenyon [12], allowed us to introduce our main contributions. In particular, we were able to prove that Trabb-Pardo's intersection algorithm [50] works in $O(k\delta \lg (u/\delta))$ time, where δ is the alternation measure of the query instance [12]. Thus, Trabb-Pardo's intersection algorithm was likely the first adaptive intersection algorithm that ever existed, appearing about 22 years before Demaine et al.'s adaptive approach. The lack of analysis on this algorithm (the original author only analyzed his algorithm in the average case) might explain the lack of consideration regarding this algorithm, in particular in practice. Motivated by this result, we introduced compressed representations of integer sets preserving the running time of Trabb-Pardo's algorithm, and even improving it. Summarizing, our proposals: (1) use compressed space usage, (2) have adaptive intersection computation time, and (3) have highly competitive practical performance.

Multiple avenues for future research are open now. For instance, novel data structures supporting operation rank_1 have emerged recently [32]. These offer interesting trade-offs, using less space than then ones we used, with competitive operation times. Another interesting

line is that of alternative binary trie compact representations. E.g., a DFS representation [13] (rather than BFS, as the one used in this paper), which would potentially reduce the number of cache misses when traversing the tries. Finally, our representation would support dynamic sets (where insertion and deletion of elements are allowed) if we use dynamic binary tries [6].

References

1. The Lemur Project. `https://lemurproject.org/`. Accessed March 14, 2023.
2. Roaring bitmaps. `https://github.com/RoaringBitmap/CRoaring`. Accessed March 14, 2023.
3. Roaring bitmaps: A better compressed bitset. `https://roaringbitmap.org/`. Accessed March 14, 2023.
4. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
5. V. Ngoc Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.
6. D. Arroyuelo, P. Davoodi, and S. Rao Satti. Succinct dynamic cardinal trees. *Algorithmica*, 74(2):742–777, 2016.
7. D. Arroyuelo, J. Fuentes-Sepúlveda, and D. Seco. Three success stories about compact data structures. *Communications of the ACM*, 63(11):64–65, 2020.
8. D. Arroyuelo and R. Raman. Adaptive succinctness. *Algorithmica*, 84(3):694–718, 2022.
9. R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 400–408. Springer, 2004.
10. R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th International Conference on String Processing and Information Retrieval (SPIRE)*, LNCS 3772, pages 13–24. Springer, 2005.
11. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–399. ACM/SIAM, 2002.
12. J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transations on Algorithms*, 4(1):4:1–4:18, 2008. `doi:10.1145/1328911.1328915`.
13. David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. `doi:10.1007/s00453-004-1146-6`.
14. P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *Proc. 18th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 4835, pages 739–750. Springer, 2007.
15. S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.
16. D. Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, 1997.
17. C. Clarke, F. Scholer, and I. Soboroff. TREC terabyte track. `https://www-nlpir.nist.gov/projects/terabyte/`. Accessed March 14, 2023.
18. H. Cohen and E. Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010. `doi:10.1016/j.tcs.2010.06.002`.
19. J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proc. 2nd ACM International Conference on Web Search and Data Mining (WSDM'09)*, pages 1–1, 2009.
20. E. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752. ACM/SIAM, 2000.
21. B. Ding and A. König. Fast set intersection in memory. *Proc. VLDB Endowment*, 4(4):255–266, 2011. `doi:10.14778/1938545.1938550`.

22 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

23 R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 6th Edition.* Pearson, 2011.

24 L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.

25 A. S. Fraenkel and S. T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64(1):31–55, 1996. `doi:10.1016/0166-218X(93)00116-H`.

26 T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012.

27 S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

28 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. ACM/SIAM, 2003.

29 A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.

30 G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989. `doi:10.1109/SFCS.1989.63533`.

31 S. T. Klein and D. Shapira. Searching in compressed dictionaries. In *Proc. Data Compression Conference (DCC)*, page 142. IEEE Computer Society, 2002.

32 F. Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 13617, pages 257–272. Springer, 2022.

33 R. M. Layer and A. R. Quinlan. A parallel algorithm for n-way interval set intersection. *Proc. IEEE*, 105(3):542–551, 2017. `doi:10.1109/JPROC.2015.2461494`.

34 D. Lemire. Document identifier data set. `https://lemire.me/data/integercompression2014.html`. Accessed March 14, 2023.

35 D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

36 D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi Yan Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice & Experience*, 48(4):867–895, 2018. `doi:10.1002/spe.2560`.

37 J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman, and A. de Vries. Supporting interoperability between open-source search engines with the common index file format, 2020. `doi:10.48550/ARXIV.2003.08276`.

38 J. M. Mackenzie, R. Benham, M. Petri, J. R. Trippas, J. S. Culpepper, and A. Moffat. CC-News-En: A large english news corpus. In *Proc. 29th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 3077–3084. ACM, 2020.

39 A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: performant indexes and search for academia. In *Proc. of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–56, 2019.

40 A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.

41 G. Navarro. *Compact Data Structures – A Practical Approach.* Cambridge University Press, 2016.

42 D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

43 G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *Proc. of 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 273–282, 2014.

44 G. E. Pibiri. Sliced indices. `https://github.com/jermp/s_indexes`. Accessed March 14, 2023.

45 G. E. Pibiri. Fast and compact set intersection through recursive universe partitioning. In *Proc. Data Compression Conference (DCC)*, pages 293–302. IEEE, 2021.

46 G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):125:1–125:36, 2021. `doi:10.1145/3415148`.

47 R. Raman, V. Raman, and S. Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007.

48 F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of 29th European Conference on IR Research (ECIR)*, LNCS 4425, pages 101–112. Springer, 2007.

49 A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM'11)*, pages 317–326, 2011.

50 L. Trabb-Pardo. *Set Representation and Set Intersection.* PhD thesis, STAN-CS-78-681, Department of Computer Science, Stanford University, 1978. D. E. Knuth, advisor.

51 T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106. OpenProceedings.org, 2014.

52 I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Edition.* Morgan Kaufmann, 1999.

53 H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web (WWW)*, pages 401–410, 2009.

54 J. Zhang, X. Long, , and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. 17th International Conference on World Wide Web (WWW)*, pages 387–396, 2008.

55 J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006. `doi:10.1145/1132956.1132959`.

A Plots of Experimental Results

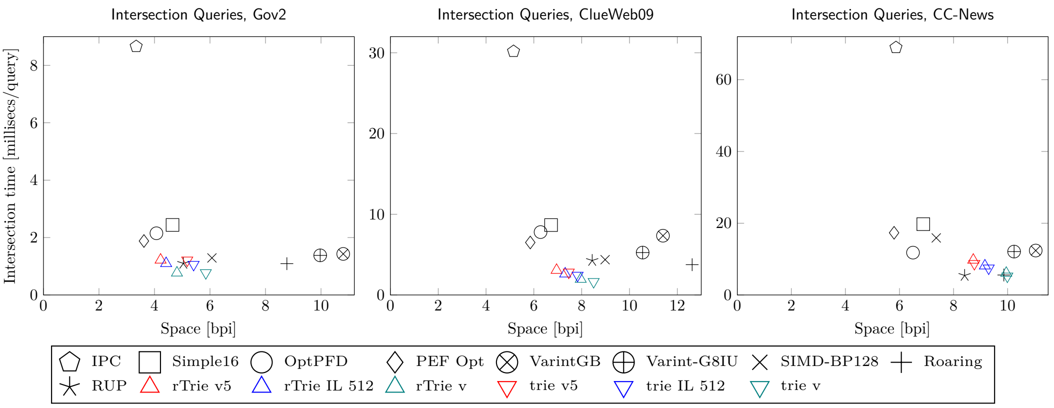

Figure 7 Space vs. time trade-off for all alternative tested on the 3 datasets.