Trie-Compressed Intersectable Sets

Diego Arroyuelo
Dept. of Informatics, Universidad Técnica Federico Santa María, Chile, & Millennium Institute for Foundational Research on Data, Chile
darroyue@inf.utfsm.cl

Juan Pablo Castillo
Dept. of Informatics, Universidad Técnica Federico Santa María, Chile, & Millennium Institute for Foundational Research on Data, Chile
juan.castillog@sansano.usm.cl

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.

PVLDB Reference Format:
Diego Arroyuelo and Juan Pablo Castillo. Trie-Compressed Intersectable Sets. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jpcastillog/compressed-binary-tries.

1 INTRODUCTION

Sets are one of the most fundamental mathematical concepts related to the storage of data. Operations such as intersections, unions, and differences are key for querying them. For instance, 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 [1]. In several applications, such as query processing in information retrieval [11] and database management systems [18], 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.

Definition 1.1 (The Offline Set Intersection Problem, ***OSIP****).* Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u)$. The OSIP consist in preprocessing this family to support query instances of the form $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, which ask to compute $\mathcal{I}(Q) = \bigcap_{i_j \in Q} S_{i_j}$.

Typical applications of this problem include the efficient support of join operations in databases [18, 52], query processing using inverted indexes in Information Retrieval (IR) [11, 55], and computational biology [30], among others. Building a data structure to speed up intersections, however, increases the space usage. Today, data-intensive applications compel not only for time-efficient solutions: space-efficient solutions allow one to maintain data structures of very large datasets entirely in main memory, thus avoiding accesses to slower secondary storage. Compact, succinct, and compressed data structures have been the solution to this problem in the last few decades [38]. 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 set intersection problem has been studied in depth, resulting in a plethora of algorithms and data structures that we briefly review in what follows. First, let us look at the online version of the problem, where sets to be intersected are given at query time, not before, so there is no time to preprocess them. Algorithms like the ones by Baeza-Yates [6], Demaine, Lopez-Ortiz, and Munro [14], and Barbay and Kenyon [9] are among the most efficient approaches. In particular, the two latter algorithms are adaptive, meaning that they are able to perform faster on "easier" query instances. In particular, Barbay and Kenyon's algorithm runs in optimal $O(\delta \sum_{i=1}^{k} \lg(n_i/\delta))$, where δ is the so-called alternation measure, which quantifies the query difficulty. The algorithm by Demaine et al. [14] has running time $O(k\delta \lg(n/\delta))$ (where $n = \sum_{i \in Q} n_i$), which is optimal when $\max_{i \in Q} \{\lg n_i\} = O(\min_{i \in Q} \{\lg n_i\})$ [8]. These algorithms require sets to be stored in plain form, in a sorted array of $\Theta(n_i \lg u)$ bits, $i = 1, \ldots, N$. This can be excessive when dealing with large databases. An interesting question is whether one can support the OSIP in time proportional to δ, while using compressed space.

On the offline version of the problem, we can cite the vast literature on inverted indexes [11, 43, 55, 58], where the main focus is on practical space-efficient set representations supporting intersections. Although there are highly-efficient approaches in practice within these lines, it is hard to find worst-case guarantees both in space usage and query time. One notorious exception is the data structure by Ding and Konig [15], which computes intersection in $O(n/\sqrt{w} + k|\mathcal{I}(Q)|)$ time, and uses linear space. According to Ding and Konig's experiments, the space can be improved to use bout 1.88 times the space of an Elias γ/δ compressed inverted index [15].

In this paper we revisit the classical set representation and intersection algorithm by Trabb-Pardo [50]. To represent a set S, Trabb-Pardo inserts the w-bit binary codes of each $x \in S$ into a binary trie data structure [22]. Intersection on this representation can be computed traversing all the tries corresponding to the query sets in coordination. We will show that an approach like this allows for

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

a space- and time-efficient implementation, with theoretical guarantees and competitive practical performance. Next, we enumerate our contributions.

1.1 Contributions

The main contributions of this paper are as follows:

- We introduce the concept of *trie certificate* of an instance of the intersection problem. This is analogous to the proofs by Demaine et al. [14] and partition certificates by Barbay and Kenyon [9], which were introduced to analyze their adaptive intersection algorithms. See Definition 3.1.
- We use trie certificates to show that the classical intersection algorithm by Trabb-Pardo [50] is actually an adaptive intersection algorithm. In particular, we show that its running time to compute $\mathcal{I}(Q)$ is $O(k\delta \lg(u/\delta))$, where δ is the alternation measure by Barbay and Kenyon [9]. See Theorem 3.2
- We introduce compact trie data structures that support the navigation operations needed to implement the intersection algorithm. We then show that these compact data structures yield a compressed representations of the sets, bounding their size by the trie entropy from Klein and Shapira [28] (and a variant we introduce in Definition 5.1). See Theorems 4.1 and 5.5.
- We 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 several scenarios. For instance, we show that our algorithm computes intersections 1.55–3.32 times faster than highly-competitive Partitioned Elias-Fano indexes [40], using 1.15–1.72 times their space. We also compare with Roaring Bitmaps [32], being 1.07–1.91 times faster while using about 0.48–1.01 times their space. Finally, we compared to RUP [41], being 1.04–2.18 times faster while using 0.82–1.19 times its space.

Overall, our work seems to be 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 of Interest

Besides intersections (and other set operations like unions and differences), there are several other operations one would want to support on an integer set. The following are among the most fundamental operations:

- $\mathsf{rank}(S, x)$: 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. $\mathsf{rank}(S, x) = j$.
- $\mathsf{predecessor}(S, x)$: for $x \in [0..u)$, yields $\max\{y \in S,\ y \leq x\}$.
- $\mathsf{successor}(S, x)$: for $x \in [0..u)$, yields $\min\{y \in S,\ y \geq x\}$.

A set S can be also described using its *characteristic bit vector* (cbv, for short) $C_S[0..u)$, such that $C_S[x_i] = \mathbf{1}$ if $x_i \in S$, $C_S[x_i] = \mathbf{0}$ otherwise. On the cbv C_S we define the following operations:

- $C_S.\mathsf{rank}_\mathbf{1}(x)$: 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 position $0 \leq x < u$ s.t. $C_S.\mathsf{rank}(x) = j$.

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* (or *entropy*) quantifies the amount of bits needed to encode data using a particular compression model. For an integer universe $U = [0..u)$, let $C^{(n)} \subseteq 2^U$ denote the class of all sets $S \subseteq U$ such that $|S| = n$. In this section, we assume $S = \{x_1, \ldots, x_n\}$, for $0 \leq x_1 \leq \cdots \leq x_n < u$. We review next typical compression measures for integer sets $S \in C^{(n)}$, that will be of interest for our work. For set S, let $C_S[0..u)$ denote its *characteristic bit vector* (cbv for short), such that $C_S[x] = \mathbf{1}$ iff $x \in S$, $C_S[x] = \mathbf{0}$ otherwise.

2.2.1 The Information-Theoretic Worst-Case Lower Bound. A worst-case lower bound on the number of bits needed to represent any $S \in C^{(n)}$ can be obtained using the following information-theoretic argument. As $|C^{(n)}| = \binom{u}{n}$, at least $\mathcal{B}(n, u) = \lceil \lg \binom{u}{n} \rceil$ bits are needed to differentiate a given $S \in C^{(n)}$ among all other possible sets. This bound is tight, as there are set representations achieving this bound [44, 46]. If $n \ll u$, $\mathcal{B}(n, u) \approx n \lg e + n \lg \frac{u}{n} - O(\lg u)$ bits (using the Stirling approximation of $n!$). This is just a worst-case lower bound: some sets in $C^{(n)}$ can be represented using less bits, as we will see next.

2.2.2 The $\mathsf{gap}(S)$ Compression Measure. A well-studied compression measure is $\mathsf{gap}(S)$, defined as follows. Let us denote $g_1 = x_1$ and, for $i = 2, \ldots, n$, define $g_i = x_i - x_{i-1} - 1$. Then, we define the $\mathsf{gap}(S)$ measure as follows:

$$\mathsf{gap}(S) = \sum_{i=1}^{n} \lfloor \lg g_i \rfloor + 1.$$

Note that $\mathsf{gap}(S)$ is 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. This measure exploits the variation in the gaps between consecutive set elements: the closer the elements, the smallest this measure is. The following lemma upper bounds $\mathsf{gap}(S)$:

Lemma 2.1. *Given a set $S \subseteq [0..u)$ of n elements, it holds that $\mathsf{gap}(S) \leq \mathcal{B}(n, u)$, with equality only when $g_i = \frac{u}{n}$ (for $i = 1, \ldots, n$).*

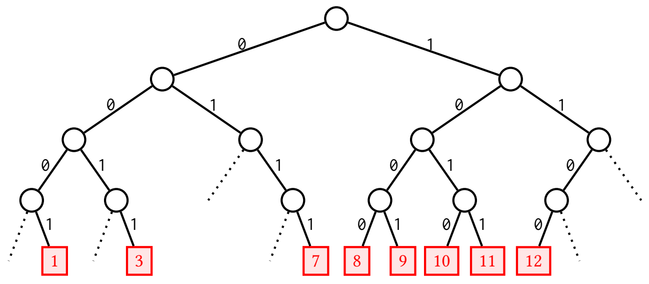
The $\mathsf{gap}(S)$ measure has been traditionally used in applications like inverted-index compression in information retrieval [11] and databases [55].

2.2.3 The $\mathtt{rle}(S)$ Compression Measure. Run-length encoding is a more appropriate model when set elements tend to be clustered into runs of consecutive elements. Following Arroyuelo and Raman [5], a *maximal run of successive elements* $R \subseteq S$ contains $|R| \geq 1$ elements $x_i, x_i + 1, \ldots, x_i + |R| - 1$, such that $x_i - 1 \notin S$ and $x_i + |R| \notin S$. Let $R_1, \ldots, R_r$ be the partition of set S into maximal runs of successive elements, such that $\forall x \in R_i, \forall y \in R_j$, $x < y \iff i < j$. Let $z_1, \ldots, z_r$ be defined as $z_1 = \min\{R_1\}$, and $z_i = \min\{R_i\} - \max\{R_{i-1}\} - 1$, and $\ell_1, \ldots, \ell_r$ be such that $\ell_i = |R_i|$, for $i = 1, \ldots, r$. According to these definitions, the cbv of S is denoted as $C_S[0..u) = \mathbf{0}^{z_1}\mathbf{1}^{\ell_1}\mathbf{0}^{z_2}\mathbf{1}^{\ell_2} \cdots \mathbf{0}^{z_r}\mathbf{1}^{\ell_r}$, and the set can be represented through the sequences $\mathcal{Z} = \langle z_1, \ldots, z_r \rangle$ and $\mathcal{O} = \langle \ell_1, \ldots, \ell_r \rangle$ of $2r$ lengths of the alternating 0/1-runs in C_S (assume wlog that C_S begins with $\mathbf{0}$ and ends with $\mathbf{1}$). Then:

$$\texttt{rle}(S) = \sum_{i=1}^{r} (\lfloor \lg (z_i - 1) \rfloor + 1) + \sum_{i=1}^{r} (\lfloor \lg (\ell_i - 1) \rfloor + 1).$$

Lemma 2.2 ([20]). *Given a set $S \subseteq [0..u)$ of $n < u/2$ elements, it holds that $\texttt{rle}(S) < \mathcal{B}(n,u) + n + O(1)$.*

2.2.4 The $\texttt{trie}(S)$ Compression Measure. Let us consider now representing a set $S \in C^{(n)}$ using a binary trie denoted bintrie(S), where the $\ell = \lceil \lg u \rceil$-bit binary encoding of every element is added. Hence, bintrie(S) has $|S|$ external nodes, all at depth ℓ. For an external trie node v corresponding to an 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], former Knuth's student. 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 bintrie(S_1) and bintrie(S_2). From this encoding, the following compression measure can be



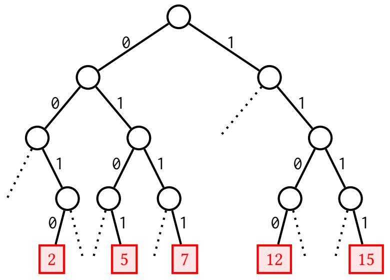


Figure 1: Binary tries bintrie(S_1) and bintrie(S_2) encoding sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$.

derived [26]. 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 [28] 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 bintrie(S) [26]. Notice that $\texttt{trie}(S)$ decreases as more 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 2.3. We say that a node v in 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 2.4 ([23]). *For* bintrie(S)*, the following results hold:*

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

Definition 2.5. 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 2.6 ([26]). *Given a set $S \subseteq [0..u)$ of n elements, it holds that:*

1. $\texttt{trie}(S) \le \min\{2\texttt{gap}(S), n \lg(u/n) + 2n\}$.
2. *$\exists a \in [0..u)$, such that* $\texttt{trie}(S + a) \le \texttt{gap}(S) + 2n - 2$.
3. *For any $a \in [0..u)$ chosen uniformly at random,* $\texttt{trie}(S{+}a) \le \texttt{gap}(S) + 2n - 2$ *on average.*

2.3 Set Intersection Algorithms

We review set intersection algorithms needed for our work.

2.3.1 Barbay and Kenyon Algorithm. Barbay and Kenyon intersection algorithm [9] is an *adaptive approach* that works on the (more general) comparison model. 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. Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the algorithm assumes the involved sets are represented using sorted arrays storing the $\lceil \lg u \rceil$-bit binary codes of the set elements. Algorithm 1 shows the pseudocode. The algorithm keeps an element x which is used to carry out searches in all sets. Initially, $x \leftarrow S_1[1]$ (i.e., the first element of set S_1). At each step, the algorithm looks for the successor of x in the next set (say, S_2), using doubling (or exponential) search [10]. If $x \in S_2$, we repeat the process with S_3. Otherwise, $x \leftarrow$ successor(S_2, x), and continue with S_3. It is also important to keep a finger f_i with every set S_i in the query. The idea is that f_i indicates the point where the previous search finished in S_i. So, the next search will start from f_i.

Algorithm 1: `BK-Intersection`(sets $S_1, \ldots, S_k$)

```
Result: The set intersection $I = S_1 \cap \cdots \cap S_k$
begin
    $I \leftarrow \emptyset$
    $x \leftarrow S_1[1]$
    $i \leftarrow 2$
    $occ \leftarrow 1$
    while $x \neq \infty$ do
        $y \leftarrow$ successor$(S_i, x)$
        if $x = y$ then
            $occ \leftarrow occ + 1$
        if $occ = k \vee x \neq y$ then
            if $occ = k$ then
                $I \leftarrow I \cup \{x\}$
            $x \leftarrow y$
            $occ \leftarrow 1$
        $i \leftarrow (i + 1) \bmod k$
    return $I$
```

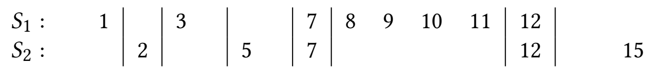


Figure 2: Vertical lines show the smallest partition certificate $\mathcal{P}$ = $\{[0..1], [2..2], [3..4], [5..6], [7..7], [8..11], [12..12], [13..15]\}$ of size $\delta = 8$ of the universe $[0..16)$ for the intersection of sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$.

Any algorithm that computes $\mathcal{I}(Q)$ must show a certificate [9] or proof [14] to show that the intersection is correct. That is, that any element in $\mathcal{I}(Q)$ belongs to the k sets $S_{i_1}, \ldots, S_{i_k}$, and that no element in the intersection has been left out of the result. Barbay and Kenyon [9] introduced the notion of *partition certificate* of a query Q, which can be defined as follows.

Definition 2.7. Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, its *partition certificate* is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P} = \{I_1, I_2, \ldots, I_{|\mathcal{P}|}\}$, such that:

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

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

Definition 2.8. For a given query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, let δ denote the size of the smallest partition certificate of Q.

This value δ is known as the *alternation measure* [9], and it (somehow) measures the difficulty of a given instance of the intersection problem. Notice $|\mathcal{I}(Q)| \leq \delta$ 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 [8, 9] proved a lower bound of $\Omega(\delta \sum_{i \in Q} \lg(n_i/\delta))$ comparisons for the set intersection problem. They also proved that `BK-Intersection` runs in $O(\delta \sum_{i \in Q} \lg(n_i/\delta))$ time, which is optimal.

2.3.2 Trabb-Pardo Algorithm. In his thesis, Trabb-Pardo [50] studied integer-set representations and their corresponding intersection algorithms, such as the divide-and-conquer approach shown in Algorithm 2. Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, $\mathcal{I}(Q)$

Algorithm 2: `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 $S_i = \emptyset$ then
4              return $\emptyset$
5      if $L = R$ then
6          return $\{L\}$  // Universe of size 1, all sets
                            are the same singleton
7      else
           // Divide
8          $M \leftarrow (R + L)/2$
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
```

is computed invoking `TP-Intersection`$(S_{i_1}, \ldots, S_{i_k}, [0..u))$. The main idea is to divide the universe into two halves, to then divide each set into two according to this universe division. This differs from, e.g., Baeza-Yates's algorithm [6, 7], which divides according to the median of one of the sets. The *Divide* steps (lines 10 and 11) can be implemented implicitly by binary searching for the successor of $(R + L)/2$, provided the sorted sets are represented using plain arrays. At the first level of recursion, notice that the most significant bit of every element in sets $S_{i,l}$ is `0` (i.e., belong to the left half of the universe), whereas for $S_{i,r}$ that bit is a 1 (i.e., they belong to the right half of the universe). At each node of the recursion tree, the current universe is divided into 2 halves, and then we recurse on the sets divided accordingly. The universe partition carried out by Trabb-Pardo's algorithm avoids several of the problems faced by Baeza-Yates' algorithm and allows us for a more efficient implementation with time guarantees, as we shall see.

Sets are known in advance, so the Divide step of Algorithm 2 can be implemented efficiently by precomputing the set divisions carried out recursively. This is because set divisions are carried out according to the universe, which is query independent. Algorithms that divide according to the median element [6, 7], on the other hand, have query dependent divisions, hence hard to be precomputed. Trabb-Pardo proposed to store the precomputed divisions using a binary trie representations of the sets (see Section 2.2.4). We store $\mathsf{bintrie}(S_i)$ for each $S_i \in \mathcal{S}$. Notice how $\mathsf{bintrie}(S_i)$ mimics the way set S_i is recursively divided by Algorithm 2. The left child of the root represents all elements whose most significant bit is `0`, that is, the elements in $S_{i,l}$ (see line 10) of Algorithm 2; similarly for $S_{i,r}$, which contains all elements in S_i whose most-significant bit is `1`.

Algorithm 2 is implemented on binary tries as follows. In particular, `TP-Intersection` induces 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 whose 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 a leaf node.

2.4 Compressed Set Representations

We review next some compressed set representations.

2.4.1 Integer Compression. A rather natural way to represent a set S is to use *universal codes* [55] for integers to represent either the sequence of gaps $\mathcal{G}$, the run lengths $\mathcal{Z}$ and $\mathcal{O}$, or the POM sequence $\mathcal{T}$. Elias γ and δ [17] are among the most know universal codes, as well as Fibonacci codes [21]. In particular, Elias δ encodes an integer x using $\lfloor \lg x \rfloor + 2\lfloor \lg(\lfloor \lg x \rfloor + 1)\rfloor$ bits. Hence, sequence $\mathcal{G}$ can be encoded using $\sum_{i=1}^{n} \lfloor \lg x \rfloor + 2\lfloor \lg(\lfloor \lg x \rfloor + 1)\rfloor = \mathsf{gap}(S)(1+o(1))$ bits. Similarly, one can achieve space close to $\mathtt{rle}(S)$ or $\mathtt{trie}(S)$ by using Elias δ of sequences $\mathcal{Z}$ and $\mathcal{O}$ or $\mathcal{T}$, respectively.

As we encode the gap sequence of the set, direct access to the set elements in no longer possible. To obtain element $x_i \in S$ we must decompress $g_1, \ldots, g_i$ to compute $x_i = \sum_{j=1}^{i} g_j$, which takes $O(i)$ time. To reduce this cost, the sequence of codes is divided into $\lceil n/B \rceil$ blocks of B elements each. We store an array $A[1..\lceil n/B \rceil]$ such that $A[i] = \sum_{j=1}^{i\lceil n/B \rceil} g_j$, adding $\lceil n/B \rceil \lg u$ extra bits. Now, decoding x_i takes $O(B)$ time, as only the block containing x_i must be accessed. This representation is typical in inverted index compression in IR [11, 55]. Intersections can be computed using `BK-Intersection`, using array A to skip unwanted blocks. The running time is increased by up to $O(\delta B)$. Besides, decoding a single integer encoded with Elias δ or Fibonacci is slow in practice —15–30 nsecs per decoded integer is usual [11]. Hence, alternative approaches are used in practice, such as VByte [54], Simple9 [2] (and optimized variants like Simple16 [57] and Simple18 [4]), and PForDelta [59] (and optimized variants like OptPFD [56]). These have efficient decoding time —e.g., less than 1 nsec/int on average is typical [11]—, yet their space usage is not guaranteed to achieve any compression measure, although they yield efficient space usage in practice.

Alternatively, Elias-Fano [16, 19, 39, 53] compression represents set elements directly, rather than their gaps, using $n \lg(u/n) + 2n + o(n)$ bits of space [39] (which is worst-case-optimal [1]) and supporting efficient access to set elements. In general, this representation uses more space than Elias γ and δ over the gap sequences (recall Lemma 2.1), however, e.g., Okanohara and Sadakane [39] support access to a set element x_i in $O(1)$ time. Hence, algorithm `BK-Intersection` can be implemented in $O(\delta \sum_{i=1}^{k} (\lg \frac{u}{n_i} + \lg \frac{n_i}{\delta}))$ $O(\delta \sum_{i=1}^{k} \lg \frac{u}{\delta})$. Recent variants of Elias-Fano are Partitioned Elias-Fano (PEF) by Ottaviano and Venturini [40] and Clustered Elias-Fano by Pibiri and Venturini [42]. These are highly competitive approaches in practice, taking advantage of the non-uniform distribution of set elements along the universe.

The approach of Ding and Konig is also worth to be mentioned [15]. They introduce a linear-space data structure that leverages the word RAM model of computation to compute intersections in time $O(n/\sqrt{w} + k|\mathcal{I}(Q)|)$, where $n = \sum_{i \in Q} n_i$. According to their experiments, this seems to be a practical approach. Another practical studies of set intersection algorithms are by Tsirogiannis et al. [51] and Kim et al. [27].

2.4.2 Compressed rank/select *Data Structures.* Compressed data structures supporting rank and select operations can be used to support successor (i.e., $\mathsf{successor}(S, x) \equiv \mathsf{select}(S, \mathsf{rank}(S, x-1)+1)$) and $S[i] \equiv \mathsf{select}(S, i)$, needed to implement `BK-Intersection`. We survey next the most efficient approaches on these lines.

Operations rank and select can be supported in $O(c)$ time using a combination of Raman, Raman, and Rao succinct representation [46] and Pătraşcu's succincter [44], using $\mathcal{B}(n, u) + O(u/\lg^c (u/c)) + O(u^{3/4} \mathrm{poly} \lg(u))$ bits, for any $c > 0$. Algorithm `BK-Intersection` on this representation takes $O(ck\delta)$ time. However, the big-oh term in the space usage depends on the universe size, which would introduce an excessive space usage if the universe is big. The approaches of Mäkinen and Navarro [35] and Sadakane and Grossi [47] can be used to reduce the term $\mathcal{B}(n, u)$ to $\mathsf{gap}(S)$, while retaining $O(c)$ time rank and select. Arroyuelo and Raman [5] were able to squeeze the space usage further, to achieve better entropy bounds. They support rank and select in $O(1)$ time while using $nH_0^{\mathrm{gap}}(S) + O(u(\lg\lg u)^2/\lg u)$ bits, or alternatively $rH_0^{\mathrm{run}}(S) + O(u(\lg\lg u)^2/\lg u)$ bits, where $H_0^{\mathrm{gap}}(S)$ and $H_0^{\mathrm{run}}(S)$ are the zero-order entropies of the gap and run-length distributions, respectively, and r is the number of runs of successive elements in S. Algorithm `BK-Intersection` takes $O(k\delta)$ time on this representation. However, again the $o(u)$ term still dominates for big universes.

To avoid the $o(u)$-bit dependence, Gupta et al. [26] data structure uses $\mathsf{gap}(S)(1+o(1))$ bits, supporting rank in $O(\mathsf{PT}(u, n, a) + \lg\lg n)$ time and select in $O(\lg\lg n)$ time, where $\mathsf{PT}(u, n, a)$ is Pătraşcu and Thorup [45] optimal bound for a universe of size u, n elements, and $a = \lg(\lg u/\lg^2 n)$ [5]. Hence, `BK-Intersection` takes $O(\delta \sum_{i=1}^{k} (\mathsf{PT}(u, n_i, a_i) + \lg\lg n_i))$ time on this representation, where $a_i = \lg(\lg u/\lg^2 n_i)$. Alternatively, one can use $\mathtt{rle}(S) + o(\mathtt{rle}(S))$ bits of space, with $O(\delta \sum_{i=1}^{k} (\mathsf{PT}(u, r_i, a_i) + (\lg\lg r_i)^2))$ intersection time [5], for $a_i = \lg(\lg(u - n_i)/\lg^2 r_i)$, where r_i is the number of runs of successive elements in set S_i, $i \in Q$. Finally, the data structures by Arroyuelo and Raman [5] use $(1+1/t)nH_0^{\mathrm{gap}}(S) + O(n/\lg u)$ bits or $(1+1/t)rH_0^{\mathrm{run}}(S) + O(n/\lg u)$ bits of space, for $t > 0$. `BK-Intersection` on them takes $O(\delta \sum_{i=1}^{k} (t + \mathsf{PT}(u, n_i, O(1))))$ time. Although they avoid the dependence on u, their space still depends on n. This would dominate when the n elements are strongly grouped into runs, blowing up the space usage.

Finally, it is worth noting that wavelet trees's [25] intersection algorithm by Gagie et al. [23] actually carries out Trabb-Pardo's

[1]Actually, this is almost worst-case optimal, as this is $\Theta(n)$ bits above the worst-case lower bound defined before in this paper.

approach, adapted to work on this particular data structure. They show that their algorithm works in $O(k\delta \lg(u/\delta))$ time, yet their space usage to represent a set would be $n \lg u$ bits, for a set of n elements.

3 TRIE CERTIFICATES

Next, we analyze the running time of `TP-Intersection` when implemented using binary tries. Trabb-Pardo carried out just an average-case analysis of his algorithm [50]. We introduce the concept of *trie certificate* to carry out an adaptive-case analysis.

We show that the recursion tree (which we denote $\text{cert}(Q)$) of the synchronized DFS traversal carried out by `TP-Intersection` can act as a certificate (or proof) for the intersection. Figure 3 shows the binary trie $\text{cert}(Q)$ for $S_1 \cap S_2$, for sets S_1 and S_2 from Figure 1. Notice that: (1) every internal node in $\text{cert}(Q)$ is a comparison

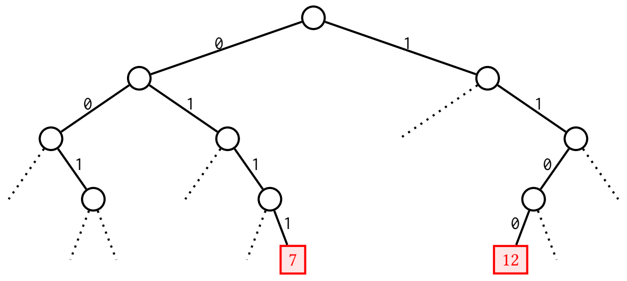


Figure 3: Trie certificate for the intersection $\{1,3,7,8,9,10,11,12\} \cap \{2,5,7,12,15\}$. This trie shows the nodes that must be checked to determine that the result is, in this case, $\{7,12\}$.

that cannot discard elements, so we must keep going down; (2) every external node at depth $d \leq \ell$ (which descend from an internal node using a dashed edge in Figure 3) is a comparison that allows us discard the corresponding universe interval of $2^{\ell-d}$ elements. For instance, the leftmost dashed edge in Figure 3 corresponds to interval $[0..1]$, whereas the rightmost corresponds to $[14..15]$. More formally, assume a binary string b of length $d-1 \leq \ell$ such that b belongs to all $\text{bintrie}(S_{i_1}), \ldots, \text{bintrie}(S_{i_k})$, and $b \cdot \mathbf{0}$ (or, alternatively, $b \cdot \mathbf{1}$) belongs to all $\text{bintrie}(S_i)$, $i \in Q$, except for at least one of the tries, say $\text{bintrie}(S_j)$. That means that the universe interval $[b \cdot \mathbf{0}^{\ell-d}..b \cdot \mathbf{1}^{\ell-d}]$ (with endpoints represented in binary) has no elements in the intersection, and hence we can safely stop at path $b \cdot \mathbf{0}$; and (3) every external node at depth ℓ is an element belonging to the intersection. In this way, the recursion tree is a certificate for the intersection.

Overall, notice that $\text{cert}(Q)$ covers the universe $[0..u)$ with intervals, indicating which universe elements belong to $\mathcal{I}(Q)$, and which ones do not. This is similar to the partition certificate delivered by Barbay and Kenyon's algorithm. For instance, the trie certificate of Figure 3 partitions the universe $[0..16)$ into $\{[0..1], [2..2], [3..3], [4..5], [6..6], [7..7], [8..11], [12..12], [13..13], [14..15]\}$.

Definition 3.1. Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, its *trie partition certificate* is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P} = \{I_1, I_2, \ldots, I_{|\mathcal{P}|}\}$, induced by the trie $\text{cert}(Q)$.

Let $|\text{cert}(Q)|$ denote the number of nodes of $\text{cert}(Q)$. Since $\text{cert}(Q)$ is the recursion tree of algorithm `TP-intersection`, the time spent by the algorithm is $O(k|\text{cert}(Q)|)$. To bound $k\text{cert}(Q)$ in the worst-case , notice that at most we must traverse all tries $\text{bintrie}(S_i)$, $i \in Q$, so we have:

$$k|\text{cert}(Q)| \leq \sum_{i \in Q} \texttt{trie}(S_i) \leq \sum_{i \in Q} n_i \lg \frac{u}{n_i}.$$

Next we bound $k|\text{cert}(Q)|$ in an adaptive way, to show that Algorithm 2 is actually an adaptive approach.

THEOREM 3.2. *Given a query instance $Q = \{i_1, \ldots, i_k\} \subset [1..N]$ with alternation measure δ,* `TP-Intersection` *computes $\mathcal{I}(Q)$ in time $O(k\delta \lg(u/\delta))$.*

PROOF. Consider the smallest partition certificate $\mathcal{P}$ of query Q, which partitions the universe into δ intervals. Let $L_1, \ldots, L_\delta$ be the size of each of the intervals $I_1, \ldots, I_\delta$ in $\mathcal{P}$. Let us consider the worst-case trie $\text{cert}(Q)$ we could have. This can be obtained by covering the δ intervals with trie nodes. The number of cover nodes equals the size of the worst-case trie we could have, and hence it is a bound on the time spent computing the intersection. Lets consider an interval I_j formed by elements not in $\mathcal{I}(Q)$. Notice that when traversing the tries $\text{bintrie}(S_i)$ in coordination, $i \in Q$, as long as one gets into one of the cover nodes of I_j, the algorithm stops at that node because it does not belong to at least one of the tries. According to Lemma 2.4 (1), a contiguous range of L leaves can be covered with up to $O(\lg L)$ nodes. Thus, in the worst-case, $\text{cert}(Q)$ has $O(\sum_{i=1}^{\delta} \lg L_i)$ external nodes that overall cover $[0..u)$. Now, according to Lemma 2.4 (3), these external nodes have $O(\sum_{i=1}^{\delta} \lg L_i + \lg u)$ ancestors, so overall $\text{cert}(Q)$ has $O(\sum_{i=1}^{\delta} \lg L_i + \lg u)$ nodes. The sum is maximized when $L_i = u/\delta$, for all $1 \leq i \leq \delta$, hence $\text{cert}(Q)$ has $O(\delta \lg (u/\delta))$ nodes. As for each node in $\text{cert}(Q)$ we must pay time $O(k)$, and the result follows. □

4 COMPRESSED INTERSECTABLE SETS

We devise next a space-efficient representation of $\text{bintrie}(S)$, 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.

4.1 A Space-Efficient $\text{bintrie}(S)$

We represent $\text{bintrie}(S)$ level-wise. Let $B_1[1..2l_1], \ldots, B_\ell[1..2l_\ell]$ be bit vectors such that B_i stores the l_i nodes at level i of $\text{bintrie}(S)$ ($1 \leq i \leq \ell$), from left to right. Each node is encoded using 2 bits, indicating the presence (using bit $\mathbf{1}$) or absence (bit $\mathbf{0}$) of the left and right children. In this way, the feasible node codes are $\mathbf{01}$, $\mathbf{10}$, and $\mathbf{11}$. Notice that $\mathbf{00}$ is not a valid node code: that would mean that the entire subtree of a node is empty, contradicting the fact that the path to which the node belongs has at least one element (as the trie only expands the paths that contain elements). The node codes 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 $2(j-1)+1$ and $2(j-1)+2$.

Let p be the position in B_i corresponding to a node v at level i of $\text{bintrie}(S)$. As the nodes are stored level-wise, the number of 1s before position p in B_i equals the number of nodes in B_{i+1} that are before the child(ren) of node v. So, $2 \cdot B_i.\mathsf{rank}_1(p-1)$ yields the position of B_{i+i} where the child(ren) of node v are. Figure 4 illustrates our representation.

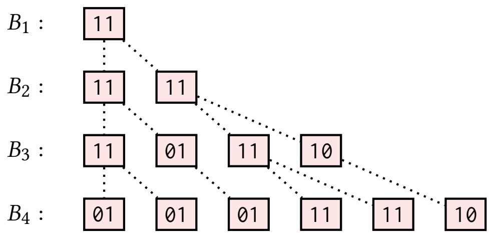


Figure 4: Level-wise bit vector representation of $\mathsf{bintrie}(S)$ **for** $S = \{1, 3, 7, 8, 9, 10, 11, 12\}$**. Dotted lines are just for clarity, as they are computed using operation** rank_1 **on the bit vectors.**

Notice that 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, there are $\mathtt{trie}(S) + 1$ nodes: n of them are external, 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 [12] to support rank and select in O(1) time, adding $o(\mathtt{trie}(S))$ extra bits.

4.2 Supporting rank and select

4.2.1 Operation $\mathsf{rank}(S, x)$*.* The main idea is to transform $\mathsf{rank}(S, x)$ into $B_\ell.\mathsf{rank}_1(p)$, for a given position p that we compute as follows. Basically, we use the binary code of x to go down from the root of $\mathsf{bintrie}(S)$. If $x \in S$, we will eventually reach the corresponding trie leaf at position p in B_ℓ. Then, $\mathsf{rank}(S, x) \equiv B_\ell.\mathsf{rank}_1(p)$, as said before. The case $x \notin S$ is more challenging. During the traversal, we keep the last trie node v_{last} along this path such that: (1) v_{last} is a branching node, and (2) the search path continues within the right child of v_{last}, and (2) the left child of v_{last} exists (i.e., v_{last} is encoded **11**). As $x \notin S$, we will eventually reach an internal trie node at level $l \geq 1$ that has no child corresponding to the l-th most-significant bit in the binary code of x. At this point, we must look for the leaf corresponding to $y = \mathsf{predecessor}(S, x)$. Notice y is the largest value stored within the left subtree of node v_{last} (i.e., the rightmost leaf in that subtree). Thus, we start from the left child of v_{last}, descending always to the right child. Upon reaching a node with no right child, we must go down to the left (which exists for sure), and then continue the process going to the right child again. Eventually, we will reach the corresponding trie leaf at position p in B_ℓ, and will be able to compute $\mathsf{rank}(S, x)$ as before.

4.2.2 Operation $\mathsf{select}(S, j)$*.* The j-th **1** in B_ℓ corresponds to the j-th element in S. So, to compute $\mathsf{select}(S, j)$ we must start from $i \leftarrow B_\ell.\mathsf{select}_1(j)$, and then go up to the parent to compute the binary representation of x_j. Given a position i in B_l, its parent can be computed as $B_{l-1}.\mathsf{select}_1(\lceil i/2 \rceil)$. When going to the parent of the current node v in the trie, we only need to know if v is the right or left child of its parent, as this allows us to know each bit in the binary representation of x_j. In our representation, the left child of a node always correspond to an odd position within the corresponding bit vectors, whereas the right child correspond to an even position.

4.3 Supporting Intersections

We assume that for each $S_i \in \mathcal{S}$, $\mathsf{bintrie}(S_i)$ has been represented using our space-efficient trie representation from the previous subsection (that we denote T_i). Given a query $Q = \{i_1, \ldots, i_k\} \subset [1..N]$, we traverse the tries $\mathsf{bintrie}(S_{i_1}), \ldots, \mathsf{bintrie}(S_{i_k})$ using the same recursive DFS traversal as in algorithm `TP-Intersection`. Besides the query itself, our algorithm receives: (1) an integer value, *level*, indicating the current level in the recursion, and (2) the integer values $r_1, \ldots, r_k$, indicating the current nodes in each trie. These are the positions of the current nodes in each trie within their corresponding bit vectors B_{level}. Algorithm 3 shows the pseudocode of our algorithm, which we call `AC-Intersection`, where AC stands for Adaptive and Compressed. `AC-Intersection` computes our compact representation for $\mathsf{bintrie}(\mathcal{I}(Q))$, which we denote T_I. The algorithm uses a variable s, initialized with **11**, which will store the bitwise-and of all current node codes (computed on line 4). In this way, $s = \mathbf{00}$ indicates that recursion must be stopped at this node, $s = \mathbf{10}$ indicates to go down only to the left child, $s = \mathbf{01}$ to go down just to the right child, and $s = \mathbf{11}$ indicates to go down on both children.

In lines 9–13 we carry out the needed computation to go down to the left child. We first determine whether we must go down to the left or not. In the former case, we compute the positions of the left-subtrie roots using rank operation. Then, on 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 children in lines 14–21. Line 17 determines whether we have already computed the ranks corresponding to the left child. If that is not the case, we compute them in line 18; otherwise we avoid them. In this way we ensure the computation of only one rank operation per traversed node in the tries. Although rank can be computed in constant time, this 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 $lChild = 0$ and $rChild = 0$, the intersection was empty on both children. In such a case we return **0** indicating this fact. Otherwise, we append the value of *lChild* and *rChild* to $T_I.B_{level}$, as that is just the encoding of the corresponding node in the output trie T_I. Note how we actually generate the output trie T_I in postorder, after we visited both children of the current nodes. This way, we write the output in time proportional to its size, saving time in practice.

Besides computing $\mathcal{I}(Q)$, a distinctive feature of our algorithm is that it also allows one to obtain for free the sequence $\langle \mathsf{rank}(S_{i_1}, x), \ldots, \mathsf{rank}(S_{i_k}, x)\rangle$, for all $x \in \mathcal{I}(Q)$. The idea is that every time we reach level ℓ of the tries, we compute $\langle T_1.B_\ell.\mathsf{rank}_1(r_1), \ldots, T_k.B_\ell.\mathsf{rank}_1(r_k)\rangle$, just before the **return** of line 7 in Algorithm 3. 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 array $D_i[1..n_i]$ such that $D[\mathsf{rank}(S_i, x_j)] = d_j$. Typical applications

Algorithm 3: AC-Intersection(query Q; roots $r_1, \ldots, r_k$; *level*)

```
Result: The binary trie $T_I$ representing $\mathcal{I}(Q) = \cap_{i \in Q} S_i$
begin
    $s \leftarrow \mathbf{11}$ // binary encoding
    for $i \in Q$ do
        $s \leftarrow s \,\&\, (T_i.B_{level}[r_i] \cdot T_i.B_{level}[r_i + 1])$
    if $level = \ell$ then
        append $s$ to $T_I.B_\ell$
        return 1
    $lChild \leftarrow \mathbf{0}$; $rChild \leftarrow \mathbf{0}$
    // Go down to the left in the tries
    if $s$ is 10 or 11 then
        $lRoots \leftarrow \emptyset$
        for $i \in Q$ do
            $lRoots \leftarrow lRoots \cup \{2 \times T_i.B_{level}.\mathsf{rank}_1(r_i)\}$
        $lChild \leftarrow$ AC-Intersection($Q, lRoots, level + 1$)
    // Go down to the right in the tries
    if $s$ is 01 or 11 then
        $rRoots \leftarrow \emptyset$
        for $i \in Q$ do
            if $s = \mathbf{01}$ then
                $rRoots \leftarrow rRoots \cup \{2 \times$
                    $T_i.B_{level}.\mathsf{rank}_1(r_i) + 1\}$
            else
                $rRoots \leftarrow rRoots \cup \{lRoots_i + 1\}$
        $rChild \leftarrow$ AC-Intersection($Q, rRoots, level + 1$)
    // Output written in postorder
    if $lChild \neq \mathbf{0}$ or $rChild \neq \mathbf{0}$ then
        append $lChild \cdot rChild$ to $T_I.B_{level}$
        return 1
    else
        return 0
```

are inverted indexes in IR (where ranking information, such as frequencies, is associated to inverted list elements), and the Leapfrog Triejoin algorithm [52] (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).

We have proved the following theorem:

Theorem 4.1. *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 $2(\mathtt{trie}(S_i) - n_i + 1) + o(\mathtt{trie}(S_i))$ bits, such that given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(Q) = \cap_{i \in Q} S_i$ can be computed in $O(k\delta \lg(u/\delta))$ time, where δ is the alternation measure of Q. Besides, for all $x \in \mathcal{I}(Q)$, 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 maximal runs of a set S to reduce the space usage of bintrie(S), as well as intersection time. Runs tend to form 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 bintrie(S) node whose subtree is full. Let depth(v) $= d$. If b denotes the binary string (of length d) corresponding to node v, the $2^{\ell-d}$ leaves covered by v correspond to the range of integers whose binary encodings are

$$b \cdot \mathbf{0}^{\ell-d}, b \cdot \mathbf{0}^{\ell-d-1}\mathbf{1}, b \cdot \mathbf{0}^{\ell-d-2}\mathbf{10}, \ldots, b \cdot \mathbf{1}^{\ell-d}.$$

So, the subtree of v can be removed, keeping just v, saving space and still being able to recover the removed elements.

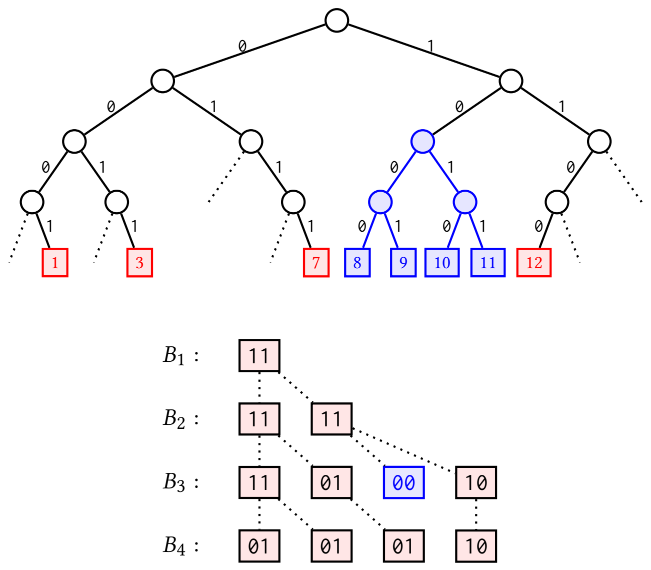


Figure 5: Above, the binary trie representing set $\{1, 3, 7, 8, 9, 10, 11, 12\}$. Notice that the subtree whose leaves correspond to elements 8, 9, 10, 11 is a full subtree. Below, our compact representation removing full subtrees and encoding their roots with 00.

Definition 5.1. Let $S \subseteq [0..u)$ be a set of n elements. We define $\mathtt{rTrie}(S)$ as the number of edges in bintrie(S) after removing the maximal full subtrees.

Notice this immediately implies $\mathtt{rTrie}(S) \leq \mathtt{trie}(S) \leq 2\mathtt{gap}(S)$. We can, however, prove tighter bounds. Assume that S can be partitioned into the maximal runs $R_1, \ldots, R_r$, each of $\ell_i = |R_i|$ elements. For a given R_i, notice that its elements correspond to ℓ_i contiguous leaves in bintrie(S). According to Lemma 2.4 (item 1), these R_i contiguous leaves are covered by at most $2\lfloor \lg(\ell_i/2) \rfloor$ cover nodes. This is the case that removes the least edges, so we analyze it. Among the cover nodes, there are 2 whose subtree has 0 edges, 2 whose subtree has 2 edges, 2 whose subtree has 6 edges, and so on. In general, for each $i = 1, \ldots, \lfloor \lg(\ell_i/2) \rfloor$, there are 2 cover nodes whose subtree has $2^i - 2$ edges. If we remove all these subtrees, the total number of nodes removed is $2\sum_{i=i}^{\lfloor \lg(\ell_i/2) \rfloor}(2^i - 2) = 2\ell_i - 4\lg \ell_i$. Since this is the case that removes the least edges belonging to full subtrees, we can bound

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

We can also prove the following bounds.

Lemma 5.2. *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. As S has maximal runs $R_1, \ldots, R_r$, each of $\ell_i = |R_i|$ elements, notice that we can rewrite

$$\texttt{gap}(S) = \sum_{i=1}^{r} (\lfloor \lg (z_i - 1) \rfloor + 1) + \sum_{i=1}^{r} (\ell_i - 1).$$

Since $\texttt{rTrie}(S) \leq \texttt{trie}(S) \leq 2\texttt{gap}(S)$, and $\texttt{rTrie}(S) \leq \texttt{trie}(S) - \sum_{i=1}^{r} (2\ell_i - 4\lg \ell_i)$ (Equation 1), it holds that

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

This proves item 1. The remaining items can be proved similarly from items 2 and 3 of Lemma 2.6. □

In our compact representation, we encode a cover node whose full subtree has been removed using **00**. Recall that **00** is an invalid node encoding, hence we use it now as a special mark. See Figure 5 for an illustration. It remains now to explain how to carry out operations on this representation.

5.1 rank(S, x)

We proceed mostly as explained for our original representation, traversing the path from the root to the leaf representing y = predecessor(S, x). However, this time there can be subtrees that have been removed as explained, hence rank(S, x) not necessarily corresponds to the rank up to the **1** we arrive at the last level B_ℓ: this rank only gives partial information to compute the operation. It remains to account for all removed **1**s corresponding to leaves of complete subtries. To do so, at each level of the trie we must regard the **00**s that lie to the left of the path we are following. Notice that every **00** at depth $1 \leq l \leq \ell$ corresponds to a full subtrie of $2^{\ell-l+1}$ leaves that have been removed from B_ℓ. To account for them, we keep a variable d during the traversal, initialized as $d \leftarrow 0$. At each level $l \geq 1$, being at position i of B_l, we carry out $d \leftarrow d + 2^{\ell-l+1} \cdot B_l.\text{rank}_{00}(i)$, where rank$_{00}$ is the number of nodes encoded **00** before position i in B_l. Notice this is different to the number of **00**s before position i, in our case we need to count the number of **00**s that are aligned with odd positions (and hence represent removed nodes).

5.2 Intersection

Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the procedure is similar to that of Algorithm 3. The only difference is that if in a given trie bintrie(S_i) we arrive at a node encoded **00**, we can temporarily exclude that trie from the intersection without affecting the result. The rationale is that every possible element below that node belongs to S_i, hence the intersection within that subtree is independent of S_i and we can temporarily exclude it. To implement this idea, we keep boolean flags $f_1, \ldots, f_k$ such that f_j corresponds to bintrie(S_{i_j}). Initially, we set $f_i \leftarrow$ true, for $1 \leq i \leq k$. If, during the intersection process, we arrive at a node encoded **00** in bintrie(S_i), we set $f_i \leftarrow$ false. The idea is that at each node visited during the recursive procedure, only the tries whose flag is true participate in the intersection. The remaining ones are within a full subtrie, so they are currently excluded. When the recursion at a node encoded **00** in bintrie(S_i) finishes, we set $f_i \leftarrow$ true again. If, at a given point, all tries have been temporarily excluded but one, let us say 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.

5.3 Running Time Analysis

Next we analyze the running time of our intersection algorithm on $\texttt{rTrie}(S)$ compressed sets. As we have seen, runs can be exploited to use less space. Now, we also prove that runs can be exploited to improve intersection computation time. The rationale is that if we intersect sets with runs of successive elements, very likely the output will have some runs too. We will show that our algorithm is able to provide a smaller partition certificate when there are runs in the output. Next, we redefine the concept of partition certificate [9], taking into account the existence of runs in the output.

Definition 5.3. Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, a run-partition certificate for it is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P} = \{I_1, I_2, \ldots, I_p\}$, such that the following conditions hold:

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

The second item is the same as Barbay and Kenyon's partition certificate, and correspond to intervals that cover the elements not in the intersection. The first item, on the other hand, correspond to the elements in the intersection. Unlike Barbay and Kenyon, in our partition certificate elements in the intersection are not necessarily covered by singletons: if there is a run of successive elements in the intersection, our definition allows for them to be covered by a single interval in the certificate.

Definition 5.4. For a given query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, let ξ denote the size of the smallest run-partition certificate of Q.

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 $Q = \{S_{i_1}, S_{i_2}, S_{i_3}, S_{i_4}\}$ and its smallest run-partition certificate $\mathcal{P} = \{[0..7], [8..9], [10..10], [11..14], [15..15]\}$ of size $\xi = 5$.

It is easy to see that $\xi \leq \delta$ holds. Besides, although $|\mathcal{I}(Q)| \leq \delta$ holds, in our case there can be query instances such that $\xi < |\mathcal{I}(Q)|$. Figure 6 illustrates our definition for an intersection of 4 sets on the universe $[0..15)$. Notice that $\xi = 5$ for this example, whereas $|\mathcal{I}(Q)| = 6$ and $\delta = 9$.

Our main result is stated in the following theorem:

Theorem 5.5. *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 $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(Q) = \cap_{i \in Q} S_i$ can be computed in $O(k\xi \lg(u/\delta))$ time, where ξ is the run alternation measure of Q.*

Proof. Consider the smallest partition certificate $\mathcal{P}$ of query Q, which partitions the universe $[0..u)$ into ξ intervals $I_1, \ldots, I_\xi$, of size $L_1, \ldots, L_\xi$, respectively. Let us cover the ξ intervals with trie nodes, in order to bound the number of nodes of $\text{cert}(Q)$ (and, hence, the running time). As we already saw in the proof of Theorem 3.2, all intervals I_i such that $I_i \cap \mathcal{I}(Q) = \emptyset$ are covered by at most $O(\lg L_i)$ nodes in $\text{cert}(Q)$. We now prove the same for intervals $I_j \subseteq \mathcal{I}(Q)$. The only thing to note is that our algorithm stops as soon as it arrives to a node covering successive elements in the output. As there can be $O(\lg L_j)$ such cover nodes, we can prove that $[0..u)$ can be covered by $O(\sum_{i=1}^{\xi} \lg L_i) = O(\sum_{i=1}^{\xi} \lg(u/\xi))\ \text{cert}(Q)$ nodes, hence $\text{cert}(Q)$ has $O(O(\xi \lg(u/\xi)))$ nodes overall. The result follows from the fact that at each node the time is $O(k)$. □

6 IMPLEMENTATION

6.1 Implementing the Tries

We implemented bit vectors $B_1, \ldots, B_\ell$ in plain form using class `bit_vector<>` from the `sdsl` library [24]. We support rank_1 on them using different data structures and obtain:

- **`trie v`, `rTrie v`:** the variants defined in Section 4 and 5, respectively, using `rank_support_v` for rank_1. It uses 25% extra space on top of the bit vector, and supports rank_1 in $O(1)$ time.
- **`trie v5`, `rTrie v5`:** use `rank_support_v5`, requiring 6.25% extra space on top of the bit vectors, and supporting rank_1 in $O(1)$ time. This alternative uses less space, yet it is slower in practice.
- **`trie IL`, `rTrie IL`:** use `rank_support_il`, aiming at reducing the number of cache misses to compute rank_1. We use block size 512, requiring 12.5% extra space on top of the bit vectors, while supporting rank_1 in constant time.

We do not store any rank_1 data structure for the last-level bit vector B_ℓ. So, we are not able to compute $\text{rank}(S, x)$ for a given set S (recall that this operation is equivalent to a rank_1 on the corresponding position of B_ℓ). This is in order to be fair, as most state-of-the-art alternatives we tested do not support this operation. Also, we do not use data structures for select_1 in our implementation, as we only test intersections.

6.2 Implementing the Intersection Algorithm

We implemented Algorithm 3 on our compressed trie representation, as well as the variant that eliminates subtrees that are full because of runs. We follow 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.

6.3 A Multithreaded Implementation

Our intersection algorithm can be implemented in a multithreading architecture quite straightforwardly. Let t denote the number of available threads. Then, we define $c = \lfloor \lg t \rfloor$. Hence, our algorithm proceeds as in Algorithm 3, generating a binary trie of height c (that we will call *top trie*), with at most t leaves. Then, we execute again Algorithm 3, this time in parallel with each thread starting from a different leaf of the top trie. If there are less than t leaves in the top trie, we go down until the t threads have been allocated. 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, and each thread will write its own part of the output in parallel. This is just a simple approach that does not guarantees load balancing among threads, however it works relatively well in practice.

6.4 Source Code Availability

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

7 EXPERIMENTAL RESULTS

Next, we experimentally evaluate our approaches.

7.1 Experimental Setup

Hardware and Software. All experiments were run on server with an i7 10700k CPU with 8 cores and 16 threads, with disable turbo boost running at 4.70 GHz in all cores and running Ubuntu 20.04 LTS operating system. We have 32GB of RAM(DDR4-3.6GHz) running in dual channel. Our implementation is developed in C++, compiled with g++ 9.3.0 and optimization flags `-O3` and `-march=native`.

Datasets and Queries. In our tests, we used family of sets corresponding to inverted indexes of three standard document collections: Gov2 [2], ClueWeb09 [3], and CC-News [34]. For Gov2 and

[2]https://www-nlpir.nist.gov/projects/terabyte/
[3]https://lemurproject.org/

ClueWeb09 collections, we used the freely-available inverted indexes and query log by D. Lemire [4], 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. [34] in a *Common Index File Format* (CIFF) [33], 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 inverted lists with length at least 4096.

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
gap(S)	2.25	3.25	3,70
rle(S)	1.99	3.33	4,23
trie(S)	3.48	4.56	5,18
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

Compression Measures and Baselines. 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 [17], Fibonacci [21], rrr_vector<> [46], and sd_vector<> [39], all of them from the sdsl library [24]. In particular, Elias γ, δ, and Fibonacci codes are know for yielding highly space-efficient set representations in IR [11], hence they are a strong baseline 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, it is well known that these codes are relatively slow to be decoded, 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 from the universe $[0..u)$, which is close to the worst-case upper bound $\mathcal{B}(n, u)$. 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 space usage higher than the remaining alternatives. These values will serve as a baseline to compare our results.

[4]See https://lemire.me/data/integercompression2014.html for download details.

Indexes Tested. We compare our proposal with state-of-the-art set compression approaches. In particular, we use the code available at the project *Performant Indexes and Search for Academia*[5] (PISA) [36] for the following approaches:

IPC: the Binary Interpolative Coding approach by Moffat et al. [37]. This is a highly space-efficient approach, with a relatively slow processing performance [11, 37].

PEF Opt: the highly competitive approach by Ottaviano and Venturini [40], which partitions the universe into non-uniform blocks and represents each block appropriately.

OptPFD: The Optimized PForDelta approach by Yan et al. [56].

SIMD-BP128: The highly efficient approach by Lemire and Boyts [31], aimed at decoding billions of integers per second using vectorization capabilities of modern processors.

Simple16: The approach by Zhang at al. [57], a variant of the Simple9 approach [2] that combines a relatively good space usage and an efficient intersection time.

Varint-G8IU: The approach by Stepanov et al. [49], using SIMD instructions to speed up set manipulation.

VarintGB: The approach presented by Dean [13].

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

Roaring: the compressed bitmap approach by Lemire et al. [32], which is widely used as an indexing tool on several systems and platforms [6]. Roaring bitmaps are highly competitive, taking full advantage of modern CPU hardware architectures. We use the C++ code from the authors [7]

RUP: The recent recursive universe partitioning approach by Pibiri [41], using also SIMD instructions to speed up processing. We use the code from the author [8].

7.2 Experimental Intersection Queries

Table 2 shows the average experimental intersection time and space usage (in bits per integer) for all the alternatives tested. Figure 7 shows the same results, using space vs. time plots. As it can be seen, our approaches introduce relevant trade-offs. We compare next with the most competitive approaches on the different datasets. 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. 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. For CC-News, the resulting inverted index has considerably less runs in the inverted lists, hence the space usage of trie and rTrie is about the same. However, trie is faster than rTrie, as the code to handle runs introduces an overhead that does not pay 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

[5]https://github.com/pisa-engine/pisa
[6]See, e.g., https://roaringbitmap.org/
[7]https://github.com/RoaringBitmap/CRoaring.
[8]https://github.com/jermp/s_indexes

Table 2: Average intersection time 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

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, outperforming the highly-engineered ultra-efficient set compression techniques we tested.

8 CONCLUSIONS

We conclude that Trabb-Pardo's intersection algorithm [50] implemented using compact binary tries yields a solution to the offline set intersection problem that is appealing both in theory and practice. Namely, our proposal has: (1) theoretical guarantees of compressed space usage, (2) adaptive intersection computation time, and (3) highly competitive practical performance. Regarding our experimental results, we show that our algorithm computes intersections 1.55–3.32 times faster than highly-competitive Partitioned Elias-Fano indexes [40], using 1.15–1.72 times their space. We also compare with Roaring Bitmaps [32], being 1.07–1.91 times faster while using about 0.48–1.01 times their space. Finally, we compared to RUP [41], being 1.04–2.18 times faster while using 0.82–1.19 times its space.

After this work, multiple avenues for future research are open. For instance, novel data structures supporting operations rank_1 and select_1 have emerged recently [29]. These offer interesting trade-offs, using much less space than then ones we used in our implementation, with competitive operation times. We think these can improve our space usage significantly, keeping our competitive query times. Another interesting line is that of alternative binary trie compact representations. E.g., a DFS representation (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 [3]

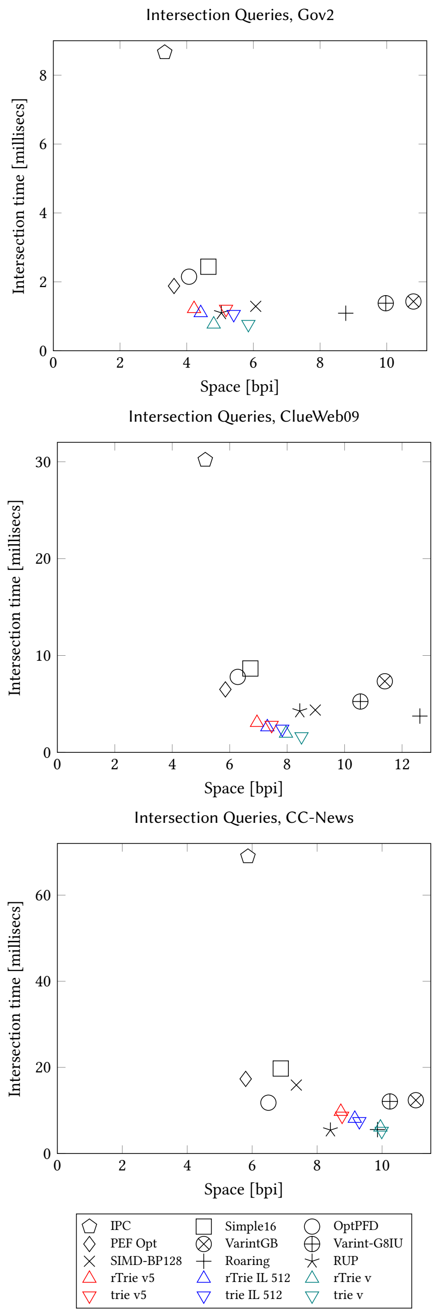


Figure 7: Space vs. time trade-off for all alternative tested on the 3 datasets.

ACKNOWLEDGMENTS

This work was funded by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile (both authors). We thank Gonzalo Navarro, Adrián Gómez-Brandón, and Francesco Tosoni for enlightening comments, suggestions, and discussions about this work.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

[2] Vo Ngoc Anh and Alistair Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Inf. Retr.* 8, 1 (2005), 151–166. https://doi.org/10.1023/B:INRT.0000048490.99518.5c

[3] Diego Arroyuelo, Pooya Davoodi, and Srinivasa Rao Satti. 2016. Succinct Dynamic Cardinal Trees. *Algorithmica* 74, 2 (2016), 742–777. https://doi.org/10.1007/s00453-015-9969-x

[4] D. Arroyuelo, M. Oyarzún, S. González, and V. Sepulveda. 2018. Hybrid compression of inverted lists for reordered document collections. *Information Processing & Management* (2018). In press.

[5] Diego Arroyuelo and Rajeev Raman. 2022. Adaptive Succinctness. *Algorithmica* 84, 3 (2022), 694–718. https://doi.org/10.1007/s00453-021-00872-1

[6] Ricardo A. Baeza-Yates. 2004. A Fast Set Intersection Algorithm for Sorted Sequences. In *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings (LNCS)*, Süleyman Cenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusöz (Eds.), Vol. 3109. Springer, 400–408.

[7] Ricardo A. Baeza-Yates and Alejandro Salinger. 2005. Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences. In *String Processing and Information Retrieval, 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005, Proceedings (LNCS)*, Mariano P. Consens and Gonzalo Navarro (Eds.), Vol. 3772. Springer, 13–24.

[8] Jérémy Barbay and Claire Kenyon. 2002. Adaptive intersection and t-threshold problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, David Eppstein (Ed.). ACM/SIAM, 390–399. http://dl.acm.org/citation.cfm?id=545381.545432

[9] Jérémy Barbay and Claire Kenyon. 2008. Alternation and redundancy analysis of the intersection problem. *ACM Transations on Algorithms* 4, 1 (2008), 4:1–4:18. https://doi.org/10.1145/1328911.1328915

[10] Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett.* 5, 3 (1976), 82–87. https://doi.org/10.1016/0020-0190(76)90071-5

[11] S. Büttcher, C. Clarke, and G. Cormack. 2010. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press.

[12] D. Clark. 1997. *Compact PAT trees.* Ph.D. Dissertation. University of Waterloo.

[13] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *Proc. 2nd ACM International Conference on Web Search and Data Mining (WSDM'09).* 1–1.

[14] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. 2000. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, David B. Shmoys (Ed.). ACM/SIAM, 743–752.

[15] Bolin Ding and Arnd Christian König. 2011. Fast Set Intersection in Memory. *Proc. VLDB Endow.* 4, 4 (2011), 255–266. https://doi.org/10.14778/1938545.1938550

[16] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260. https://doi.org/10.1145/321812.321820

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

[18] Ramez Elmasri and Shamkant B. Navathe. 2011. *Fundamentals of Database Systems, 6th Edition.* Pearson.

[19] R. M. Fano. 1971. On the Number of Bits Required to Implement an Associative Memory. Memorandum 61, Computation Structures Group Memo, MIT Project MAC Computer Structures Group.

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

[21] Aviezri S. Fraenkel and Shmuel T. Klein. 1996. Robust Universal Complete Codes for Transmission and Compression. *Discrete Applied Mathematics* 64, 1 (1996), 31–55. https://doi.org/10.1016/0166-218X(93)00116-H

[22] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (sep 1960), 490â€“499. https://doi.org/10.1145/367390.367400

[23] Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. 2012. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comput. Sci.* 426 (2012), 25–41.

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

[25] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.* ACM/SIAM, 841–850. http://dl.acm.org/citation.cfm?id=644108.644250

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

[27] Sunghwan Kim, Taesung Lee, Seung-won Hwang, and Sameh Elnikety. 2018. List Intersection for Web Search: Algorithms, Cost Models, and Optimizations. *Proc. VLDB Endow.* 12, 1 (2018), 1–13. https://doi.org/10.14778/3275536.3275537

[28] Shmuel T. Klein and Dana Shapira. 2002. Searching in Compressed Dictionaries. In *2002 Data Compression Conference (DCC 2002), 2-4 April, 2002, Snowbird, UT, USA.* IEEE Computer Society, 142. https://doi.org/10.1109/DCC.2002.999952

[29] Florian Kurpicz. 2022. Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors. In *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings (Lecture Notes in Computer Science)*, Diego Arroyuelo and Barbara Poblete (Eds.), Vol. 13617. Springer, 257–272. https://doi.org/10.1007/978-3-031-20643-6_19

[30] Ryan M. Layer and Aaron R. Quinlan. 2017. A Parallel Algorithm for N-Way Interval Set Intersection. *Proc. IEEE* 105, 3 (2017), 542–551. https://doi.org/10.1109/JPROC.2015.2461494

[31] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[32] 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. *Software: Practice & Experience* 48, 4 (2018), 867–895. https://doi.org/10.1002/spe.2560

[33] Jimmy Lin, Joel Mackenzie, Chris Kamphuis, Craig Macdonald, Antonio Mallia, MichaÅ, Siedlaczek, Andrew Trotman, and Arjen de Vries. 2020. Supporting Interoperability Between Open-Source Search Engines with the Common Index File Format. https://doi.org/10.48550/ARXIV.2003.08276

[34] J. Mackenzie, R. Benham, M. Petri, J. R. Trippas, J. S. Culpepper, and A. Moffat. 2020. CC-News-En: A Large English News Corpus. In *Proc. CIKM.* To Appear.

[35] V. Mäkinen and G. Navarro. 2007. Rank and select revisited and extended. *Theoretical Computer Science* 387, 3 (2007), 332–347.

[36] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, OSIRRC@SIGIR 2019, Paris, France, July 25, 2019.* 50–56. http://ceur-ws.org/Vol-2409/docker08.pdf

[37] Alistair Moffat and Lang Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval* 3, 1 (2000), 25–47.

[38] G. Navarro. 2016. *Compact Data Structures – A Practical Approach.* Cambridge University Press.

[39] D. Okanohara and K. Sadakane. 2007. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. of 9th Workshop on Algorithm Engineering and Experiments (ALENEX).* 60–70.

[40] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano indexes. In *Proc. of 37th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 273–282.

[41] Giulio Ermanno Pibiri. 2021. Fast and Compact Set Intersection through Recursive Universe Partitioning. In *31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23-26, 2021*, Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer (Eds.). IEEE, 293–302. https://doi.org/10.1109/DCC50243.2021.00037

[42] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Clustered Elias-Fano Indexes. *ACM Trans. Inf. Syst.* 36, 1 (2017), 2:1–2:33. https://doi.org/10.1145/3052773

[43] Giulio Ermanno Pibiri and Rossano Venturini. 2021. Techniques for Inverted Index Compression. *ACM Comput. Surv.* 53, 6 (2021), 125:1–125:36. https://doi.org/10.1145/3415148

[44] M. Pătraşcu. 2008. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE, 305–313.

[45] M. Pătraşcu and M. Thorup. 2006. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC).* ACM, 232–240.

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

[47] K. Sadakane and R. Grossi. 2006. Squeezing succinct data structures into entropy bounds. In *Proc. of 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 1230–1239.

[48] F. Silvestri. 2007. Sorting Out the Document Identifier Assignment Problem. In *Proc. of 29th European Conference on IR Research (ECIR) (LNCS 4425).* Springer, 101–112.

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

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

[51] Dimitris Tsirogiannis, Sudipto Guha, and Nick Koudas. 2009. Improving the Performance of List Intersection. *Proc. VLDB Endow.* 2, 1 (2009), 838–849. https://doi.org/10.14778/1687627.1687722

[52] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13

[53] Sebastiano Vigna. 2013. Quasi-succinct indices. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis (Eds.). ACM, 83–92.

[54] Hugh E. Williams and Justin Zobel. 1999. Compressing Integers for Fast File Access. *Comput. J.* 42, 3 (01 1999), 193–201. https://doi.org/10.1093/comjnl/42.3.193 arXiv:https://academic.oup.com/comjnl/article-pdf/42/3/193/962527/420193.pdf

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

[56] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web (WWW)*. 401–410.

[57] Jiangong Zhang, Xiaohui Long, , and Torsten Suel. 2008. Performance of compressed inverted list caching in search engines. In *Proc. 17th International Conference on World Wide Web (WWW)*. 387–396.

[58] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006), 6. https://doi.org/10.1145/1132956.1132959

[59] M. Zukowski, S. Héman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proc. of 22nd Int. Conference on Data Engineering (ICDE)*. IEEE Computer Society, 59.