Fast and Compact Set Intersection through Recursive Universe Partitioning

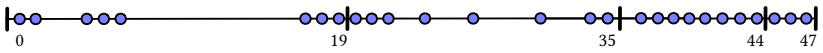
Giulio Ermanno Pibiri
ISTI-CNR, Pisa, Italy
giulio.ermanno.pibiri@isti.cnr.it

Abstract

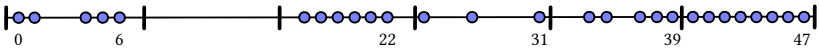
We present a data structure that encodes a sorted integer sequence in small space allowing, at the same time, fast intersection operations. The data layout is carefully designed to exploit word-level parallelism and SIMD instructions, hence providing good practical performance. The core algorithmic idea is that of *recursive partitioning the universe* of representation: a markedly different paradigm than the widespread strategy of partitioning the sequence based on its length. Extensive experimentation and comparison against several competitive techniques shows that the proposed solution embodies an improved space/time trade-off for the set intersection problem.

1 Introduction

The problem we take into account is that of introducing a *compressed* representation for a sorted integer sequence $\mathcal{S}[0..n)$ whose values are drawn from a universe $u \geq \mathcal{S}[n-1]$, so that intersecting two such sequences is done efficiently. We assume $\mathcal{S}$ to be strictly increasing, i.e., $\mathcal{S}[i-1] < \mathcal{S}[i]$ for $0 < i < n$. In this work we focus on the static version of the problem, thus we are not allowed to insert or delete integers from $\mathcal{S}$. The most notable application of this problem is the efficient storage of *inverted lists* [14, 21]. Many different techniques were developed to represent integer sequences, each of them exposing a different space/time trade-off. We point the reader to the survey by Zobel and Moffat [21] for general background on inverted indexes, and to that by Pibiri and Venturini [14] for a description and discussion of compression techniques. The key point is that, for all such different representations, the efficient execution of intersection relies on *partitioning* the sequence because of the following simple observation: when we ask whether the integer x is present or not in the sequence $\mathcal{S}$, we can safely *skip* all partitions of $\mathcal{S}$ whose maximum integer is smaller than x. In fact, since $\mathcal{S}$ is sorted, none of the integers smaller than x should be considered. Based on this observation, integer sequences have been traditionally partitioned *by cardinality* (CP), i.e., consecutive elements are grouped together into fixed-size or variable-size partitions as exemplified in Figure 1a. In this setting, prior work [2, 8, 11] has extensively demonstrated that fast intersection is achieved by resorting to the $\mathsf{nextGEQ}(x)$ primitive that returns the smallest integer $z \in \mathcal{S}$ that is greater-than or equal-to x. The algorithm is as follows. Suppose that $\mathcal{S}_1$ is shorter than $\mathcal{S}_2$. We search $\mathcal{S}_2$ for the first value x of $\mathcal{S}_1$ with $\mathcal{S}_2.\mathsf{nextGEQ}(x)$: if the value z returned by the operation is equal to x then it is a member of the intersection and we can just repeat this search step for the *next* value of $\mathcal{S}_1$; otherwise z gives us a *candidate* value to be searched next, indeed allowing to skip the searches for all values



(a) cardinality partitioning – CP



(b) universe partitioning – UP

Figure 1: Example of a sequence of 27 values represented as dots drawn from a universe of size 47 as partitioned by cardinality (CP) and by universe (UP), using partitions of 8 integers. We also indicate the maximum integer in each partition.

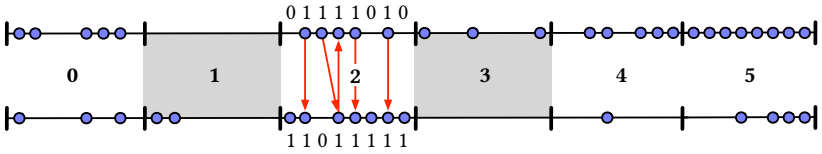


Figure 2: A graphical visualization of the algorithm intersecting two sequences partitioned by universe: partitions **1** and **3** can be safely skipped because they are empty in (at least) one of the two sequences. The arrows represent the searches issued by the intersection algorithm we described for CP using nextGEQ as if it were used to intersect partitions of identifier **2**.

between x and z. In fact, since we have that $z \geq x$, $\mathcal{S}_2.\mathsf{nextGEQ}(y)$ will be equal to z also for *all* values y such that $x < y < z$, hence none of such integers can be a member of the intersection.

Our Contributions. In this work, instead, we explore the applicability of another partitioning paradigm for efficient set intersection in compressed space: *partitioning by universe* (UP). In a simple implementation of the paradigm, a universe *span* s is chosen and all integers x of $\mathcal{S}$ such that $sk \leq x < s(k+1)$, are compressed into the same k-th partition. For example, Figure 1b shows the same sequence as in Figure 1a now partitioned by universe with a span of size 8. Also this strategy permits to skip over the sequence's values because only partitions relative to the *same* universe have to be intersected. In this case, intersection proceeds by identifying all common partitions and, for each of them, by resolving a smaller intersection of at most s integers. Depending on the cardinality of a partition, different compression strategies and intersection algorithms can be adopted, thus *not necessarily* relying on the nextGEQ primitive.

To better understand what we mean, let us discuss the graphical visualization in Figure 2, showing two sequences being intersected. Only partitions 0, 2, 4, and 5 are identified by the algorithm because these are in common between the two sequences; partitions 1 and 3 are discarded. More importantly, if partition 2, is represented in both cases as a binary vector of 8 bits, i.e., the set $\{17, 18, 19, 20, 22\}$ as `01111010` and the set $\{16, 17, 19, 20, 21, 22, 23\}$ as `11011111`, then the intersection can be preformed by just computing the bitwise AND between the two bytes: AND(`01111010`, `11011111`) = `01011010`. This is considerably faster than issuing the sequences of 5

nextGEQ searches (represented with the arrows in the picture) that the intersection algorithm for CP would perform to compute the same result.

After discussing related results in Section 2, we then describe a tree-shaped data structure in Section 3 that recursively partitions the universe to better adapt to the distribution of the integers being encoded, thus attaining good space effectiveness. Furthermore, this approach guarantees that – at any level of the tree – the k-th slices of two *different* sequences, say $\mathcal{S}_1$ and $\mathcal{S}_2$, are aligned to *the same universe boundary*, which makes the use of binary vectors, word-level parallelism, and SIMD instructions natural key ingredients for fast intersection.

We experimentally compare our proposal against a rich set of competitive approaches in Section 4 using large real-world datasets, showing that its enhanced intersection efficiency paired with good space effectiveness provides a new space/time trade-off for the problem.

Although in this work we focus on the intersection operation for space constraints, we remark that the data structure also supports other operations very efficiently, such as: sequential decoding; union queries, i.e., $\mathcal{S}_1 \cup \mathcal{S}_2$; nextGEQ and random access (given an index $0 \leq i < n$, returns $\mathcal{S}[i]$). We point the interested reader to our related and extended version of the paper [12] and to Section 6 of [14] for further experiments. Lastly, the whole C++ implementation is freely available on GitHub at `https://github.com/jermp/s_indexes`.

2 Previous Work

Apart from the encoding of sparse bitvectors [6, 17], the universe partitioning paradigm has been used sporadically in the coding and data structure design areas. We mention an old, but yet very meaningful, example: the Elias-Fano encoding algorithm [4, 5]. Elias-Fano represents $\mathcal{S}$ in at most $n\phi + n + \lceil u/2^{\phi} \rceil$ bits, and it can be shown that choosing $\phi = \lfloor \log_2(u/n) \rfloor$ minimizes the number of bits of the encoding [4]. In other words, the values of $\mathcal{S}$ are partitioned by universe into chunks containing at most 2^{ϕ} integers each. We refer to this split as *parametric*, because it is dependent on the value of u and n.

But a non-parametric split is possible as well. This other approach is used by *Roaring* [9], a practical data structure that has been shown to outperform all previously proposed bitmap indexes and now widely used in commercial applications. Specifically, Roaring partitions u into chunks of 2^{16} integers and represents all the integers falling into a chunk in two different ways according to the cardinality of the chunk: if a chunk contains less than 4096 elements, then it is represented as a sorted array of 16-bit integers; otherwise it is represented as a bitmap of 2^{16} bits. Finally, extremely dense chunks can also be represented with *runs*. For example, the two runs $(13, 42)(60, 115)$ mean that all the integers $13 \leq x \leq 13 + 42$ and $60 \leq x \leq 60 + 115$ belong to the chunk.

While set intersection over *uncompressed* sequences has received much attention (for example, see [1, 3] and references therein), to the best of our knowledge, only a few previous works have studied the problem over *compressed* representations. Given a set of many sequences (i.e., an inverted index), Culpepper and Moffat [2] proposed

a framework called HYB+M2 that, given a parameter B, represents all sequences having an average gap smaller than or equal to B with binary vectors and all other sequences using compressed byte-codes (e.g., Variable-Byte). Intersections between a compressed sequence and a bitvector are resolved by checking whether each integer in the sequence belongs to the bitvector using a series of constant-time operations. Lemire et al. [8] optimized this approach by relying on the parallelism of SIMD instructions to speed up sequential decompression and intersections, achieving the fastest query timings for the problem reported in the literature.

We compare against all such results in our experimental analysis in Section 4, using many billions of compressed integers.

Also Kane and Tompa [7] resort to bitvectors for low-gap regions of the sequences. In particular, they exhibit good results using a custom reordering of document identifiers that produces highly skewed distributions near the front of the sequences. We make use, instead, of the traditional and mostly-used URL assignment in our experiments. Unfortunately they did not make the code available to replicate their results. However, Lemire et al. [8] evaluated their approach and found it to be slower than HYB+M2 optimized with SIMD, against which we compare our approach in Section 4 (and improve upon).

3 Recursive Universe Partitioning

Data Structure. From a high-level point of view, we logically represent $\mathcal{S}$ using a tree of height at most 3, where the root has fanout s_1 and its children have fanout s_2. The root of the tree logically corresponds to the interval $[0, u)$ that is partitioned into slices spanning s_1 integers each (except, possibly, the last slice which may contain less integers). In what follows, we refer to such s_1-long slices as *chunks*. A header array H_1 is used to classify chunks into 4 different types according to their cardinality: *full*, *dense*, *sparse*, and *empty*. Full chunks, i.e., containing exactly s_1 integers, and empty chunks (containing no integers at all) are represented implicitly by their types. A dense chunk spanning the k-th universe slice $[s_1 k, s_1(k+1))$ is represented with a bitmap of s_1 bits by setting the i-th bit if the integer $i - s_1 k$ belongs to the slice. In particular, we regard a chunk to be dense if its cardinality is at least $s_1/2$. Doing so guarantees that the average number of bits spent for each integer belonging to a dense chunk is at most 2. Therefore, full, empty and dense chunks have no children. Instead, sparse chunks are encoded by re-applying the same strategy: since every sparse chunk now logically corresponds to a (smaller) universe slice of size s_1, the interval $[0, s_1)$ is partitioned into slices spanning s_2 integers each (again, except possibly the last one). We refer to such s_2-long slices as *blocks*. As before, a header array H_2 is used to distinguish between different block types. However, given the smaller universe slice, we only distinguish between two block types, namely dense and sparse, in order to better amortize the cost of H_2. In particular, a dense block is encoded with a bitmap of s_2 bits; a sparse block is represented with a sorted array of $\lceil \log_2 s_2 \rceil$-bit integers: if c is its cardinality, we require that $c\lceil \log_2 s_2 \rceil < s_2$ bits, otherwise we go back to the previous (dense) case.

In what follows, we fix $s_1 = 2^{16}$ and $s_2 = 2^8$. Since we consider the case where $u \leq 2^{32}$ and for our choice of $s_1 = 2^{16}$, the number of chunks is always at most 2^{16}, thus we encode this quantity into 16 bits. Similarly, it follows that each sparse chunk is sliced into at most 2^8 blocks. With this choice of s_1 and s_2, a dense chunk is a bitmap of 1024 bytes; a dense block is a bitmap of 32 bytes; a sparse block whose cardinality is c is a sorted array of 8-bit integers, hence taking c bytes overall.

For each *non-empty* block $[k2^8, (k+1)2^8)$, the array H_2 stores its identifier k and its cardinality. Both quantities are always at most 2^8, thus they take one byte each. Knowing the cardinality $0 < c \leq 2^8$ of a block we derive the number of bytes needed by its representation. If $c < 2^8/8 - 1 = 31$, the block is considered to be sparse, thus it takes c bytes; otherwise it is dense and takes 32 bytes. Therefore, a sparse block consumes at most $(2^8/8-2)\times 8+8 = 248$ bits. Note that we do not set the sparseness threshold to $2^8/8 = 32$ because otherwise a sparse block would consume at most 256 bits that is equal to the cost of a dense block and a bitmap would suffice.

For each *non-empty* chunk $[k2^{16}, (k+1)2^{16})$, the array H_1 stores, instead, the following quantities: its identifier k, its cardinality, and the number of bytes needed by its encoding. Similarly to the case of a sparse block, we require the encoding of a sparse chunk to take *less than* 2^{16} bits, otherwise a bitmap of 2^{16} bits would suffice. Therefore, each of these 3 quantities easily fits into a 16-bit integer. Although we could derive the type of a chunk from its cardinality as done for a block, we also store the type explicitly using 16 bits. When a chunk is sparse, we need to know the number of its blocks, i.e., the number of non-empty 2^8-size slices. This number is the size of the corresponding header H_2. Therefore, we write this quantity in 8 bits and interleaved with the 16 bits dedicated to the type information. In conclusion, we spend a 64-bit overhead per chunk.

The description above also opens the possibility for better compression. For example, we could use a different representation for sparse blocks, e.g., bit-aligned universal codes such as Elias' δ. Whatever representation we use, that will give birth to interesting time/space trade-offs that we left for future investigation. The choice adopted here of $s_1 = 2^{16}$, $s_2 = 2^8$, and the use of 8-bit integer arrays clearly favours time efficiency given that both bitmaps and packed arrays are aligned to byte boundaries. In fact, as we are going to show next, the use of bitmaps joint with universe-aligned partitions is particularly effective for fast query execution because operations can be implemented via inexpensive bitwise instructions, hence exploiting word-level parallelism, and are suitable for even more advanced instructions, such as SIMD.

Sequential Decompression. This operation is implemented with the function $\mathcal{S}.\mathsf{decode}(\mathit{output})$ that uncompress $\mathcal{S}$ sequentially by writing each decoded value into an *output* buffer of 32-bit integers. We loop through each chunk and, depending on its type, we decode it accordingly appending the result into the *output* buffer. This permits to code different specialized functions to handle a universe partition differently based on its type. By design, we only need to optimize two routines: decoding of bitmaps and decoding of arrays of 8-bit integers.

Bitmaps can be efficiently decoded using the built-in function `ctzll` which counts the number of trailing zero in a word of 64 bits [9] (we also tested a SIMD algorithm

to decode larger bitmaps but got almost no speed improvement).

Our sorted arrays contain at most $2^8/8-2$ integers, each value taking 8 bits. A key ingredient to decode such arrays is the instruction `_mm256_cvtepu8_epi32`, that zero extend packed (unsigned) 8-bit integers to 32-bit integers. Doing so, we can efficiently decode 8 values at a time, requiring only the above `cvtepu8` instruction and a `store`. We also experimentally observed that this approach is even more efficient when paired with loop unrolling. Therefore, if the cardinality c is less than or equal to 8, we only execute one `cvtepu8` instruction; otherwise we always execute four instructions. We determined that the distribution of the cardinality of sparse blocks goes as a power law (i.e., very skewed), thus we expect the case $c \leq 8$ to happen most of the time.

Intersection and Union. To support efficient intersection we loop through the header arrays of the sequences, intersecting only chunks/blocks that are shared by the two, i.e., they have the same partition identifier. Therefore, we reduce the original problem to a smaller instance of the same problem performing intersections between (1) two bitmaps, or (2) two arrays, or a (3) bitmap and an array.

Case (1) – the intersection between two bitmaps – translates into a sequence of inexpensive bitwise AND instructions between 64-bit words with (usually) automatic compiler vectorization. Case (2) has to intersect tiny 8-bit sorted arrays. While a scalar textbook intersection algorithm between uncompressed arrays would suffice, we can accelerate the process using a variation of the vectorized approach by Schlegel et al. [18]. The core idea of the algorithm is to use the SIMD instruction `cmpestrm` to compare strings of bytes. In our case we can execute an *all-versus-all* comparison in parallel between sets of 16×8-bit integers (there is currently no instruction working with 256-bit registers). Matching integers, i.e., integers in common between the two sets, are marked with a 32-bit bitmap returned as the result of the comparison. We can use this 32-bit value as an index in a pre-computed universal table $T[2^{16}][16]$ of 8-bit integers to obtain a permutation of bytes indexes, indicating how the matching integers should be permuted to pack them at the beginning of a 128-bit register. Case (3) – the intersection between a bitmap and an array – is implemented by checking if the values of the array correspond to bits set in the bitmap.

For the union operation we basically follow the same skeleton described for the intersection, albeit we do not rely on specific SIMD optimizations: bitmaps are merged using bitwise OR within 64-bit words; sorted arrays using scalar code; the case with a bitmap and an array is handled by first converting the sorted array into a bitmap, then using the parallelism of bitwise OR.

4 Experiments

Competitors. We compare our proposal – indicated as RUP in the following – against a rich set of competitive approaches. Among inverted list representations, we test: Elias' δ; Variable-Byte (VByte) paired with the SIMD decoding algorithm devised by Plaisance et al. [16]; Simple16 [20]; optimized PForDelta (Opt-PFor) [19]; the ϵ-optimal partitioned Elias-Fano (PEF) [11]; Interpolative [10]; the partitioned version of Variable-Byte (Opt-VByte) [13]; and the dictionary-based compressor DINT [15]

Table 1: Basic statistics for the test collections.

Quantity	Gov2	ClueWeb09	CCNews
Lists	39,177	96,722	76,474
Universe	24,622,347	50,131,015	43,530,315
Integers	5,322,883,266	14,858,833,259	19,691,599,096

(the version tested here uses a single packed dictionary with optimal parsing; refer for all details to the original paper). As the most efficient compressed bitmap index, we test Roaring [9]. Lastly, we evaluate the efficiency of the HYB+M2 intersection framework proposed by Culpepper and Moffat [2] and optimized by Lemire et al. [8] with the fast SIMD-ized format called SIMD-BP128. As we reviewed in Section 2, the framework depends on a parameter B that we vary in $\{8, 16, 32\}$ as done in previous work [8], to trade-off between space and time. Whenever a block size is required for partitioning the sequences by cardinality, we use the value of 128 as done in previous works [8, 11, 13].

To ensure a fair comparison, for all such competitors we used the C++ code made available by the respective authors. The interested reader can find the link to the corresponding source code in the References.

Datasets. We perform the experiments on the following standard, real-world, test collections. Gov2 is the TREC 2004 Terabyte Track test collection, consisting in roughly 25 million .gov sites crawled in early 2004. ClueWeb09 is the ClueWeb 2009 TREC Category B test collection, consisting in roughly 50 million English web pages crawled between January and February 2009. CCNews is a dataset of news freely available from CommonCrawl; precisely, it consists of the news appeared from 09/01/16 to 30/03/18. Identifiers were assigned to documents according to the lexicographic order of their URLs. From the original collections we retain all lists whose size is larger than 4096. Table 1 reports the basic statistics for the collections.

Hardware and Software. All the experiments are performed on a server machine with 4 Intel i7-7700 cores (@3.6 GHz), 64 GB of RAM DDR3 (@2.133 GHz) and running Linux 4.4.0 (64 bits). Caches have the following sizes: 32 K for both instruction and data L1 cache; 256 K for L2 cache; 8192 K for L3 cache. We compiled the code with `gcc` 7.3.0 using the highest optimization setting, i.e., with compilation flags `-O3` and `-march=native`. Our C++ implementation is available at `https://github.com/jermp/s_indexes`.

Space Effectiveness. Consider Table 2. As expected, all methods that are partitioned by cardinality (first group from the top of the table) achieve the best space effectiveness, with Interpolative and PEF being leaders in this regard, and with VByte being an exception for its considerably larger space. Roaring, VByte, and HYB+M2 with $B = 32$ are consistently the largest, roughly requiring $3 - 4$ bits/int more than the most effective methods in the first group. The RUP solution stands in a middle position between these two edge points, taking $2 - 2.7$ bits/int less than Roaring but also $1.3 - 2$ bits/int more than the most space efficient methods. Also note that the

Table 2: Space in bits per integer and average milliseconds per AND query.

Method		Gov2			ClueWeb09			CCNews		
		bits/int	ms/AND		bits/int	ms/AND		bits/int	ms/AND	
VByte		8.81	2.08		9.20	10.50		9.29	15.91	
Opt-VByte		3.89	2.48		5.72	12.22		6.42	17.83	
Interpolative		2.94	7.90		4.43	40.14		5.24	60.57	
δ		3.74	5.04		5.17	25.62		6.36	36.51	
PEF		3.12	2.48		4.99	12.54		5.45	19.72	
DINT		3.53	2.51		5.35	12.17		6.44	18.89	
Opt-PFor		3.63	2.81		5.46	14.78		6.07	20.37	
Simple16		4.19	2.86		5.85	14.63		6.41	21.56	
Roaring		6.63	0.32		9.78	1.64		9.49	1.29	
RUP		4.31	0.32		7.06	1.67		7.78	1.78	
	B		scalar	SIMD		scalar	SIMD		scalar	SIMD
HYB+M2	8	5.63	1.05	0.79	7.34	3.90	2.90	6.60	4.35	3.02
	16	6.93	0.79	0.64	8.05	2.69	2.19	7.21	3.12	2.37
	32	9.67	0.60	0.53	9.90	2.08	1.87	8.80	2.38	2.07

RUP approach achieves better compression than HYB+M2 with $B = 8, 16$ on both Gov2 and ClueWeb09, but it is larger on CCNews.

Intersection. To perform intersection queries, the indexes are first memory mapped and then a warming-up run is executed to fetch the necessary pages from disk. From the TREC 2005 and TREC 2006 Efficiency Track topics, we selected all queries whose terms are in the lexicons of the tested collection. We used a random sampling of 1000 such queries to test the speed of the indexes. Each run of queries is repeated 10 times to smooth fluctuations during measurements. The time reported is the average among these runs.

Table 2 shows the net result of the experiment: methods partitioned by universe, i.e., Roaring and RUP outperform all other methods, both the ones using cardinality partitioning and the HYB+M2 framework. Excluding Interpolative and δ codes that are always the slowest, the first group of techniques is strongly clustered around 2.5, 12.8, and 19 ms on average for Gov2, ClueWeb09 and CCNews respectively. In contrast, RUP completes in 0.32, 1.67, and 1.78 ms, thus being 7.8×, 7.6× and 10.6× faster on average *and* being competitive in space effectiveness. Observe that RUP is slightly slower than Roaring because the recursive partitioning induces more branches, hence partially eroding the instruction throughput.

Lastly, we report the results for the HYB+M2 framework evaluated using both scalar and SIMD-ized intersection routines. As expected, the joint use of SIMD instruction and bitvectors consistently improves efficiency, with the parameter B trading-off between space and time. The RUP approach is, however, always faster and, when similar in speed to the fastest framework configuration HYB+M2 with $B = 32$, it requires $1 - 5.3$ bits/int less.

Table 3: Average ns per decoded int and average milliseconds per OR query.

Method	Gov2		ClueWeb09		CCNews	
	ns/int	ms/OR	ns/int	ms/OR	ns/int	ms/OR
VByte	0.51	5.48	0.67	17.68	0.62	22.26
Opt-VByte	0.76	6.82	0.91	22.77	0.77	27.16
Interpolative	6.37	16.10	7.96	56.12	8.92	74.83
δ	4.01	12.84	4.15	38.78	4.26	47.36
PEF	0.91	6.36	1.32	23.20	1.57	29.65
DINT	0.82	6.18	1.24	21.48	1.43	27.25
Opt-PFor	1.06	6.54	1.53	22.95	1.19	27.50
Simple16	1.15	6.70	1.54	22.75	1.55	28.74
Roaring	0.57	1.35	0.82	5.40	0.71	4.60
RUP	0.69	1.61	0.86	6.28	0.83	7.25

Sequential Decompression and Union. Table 3 reports the average nanoseconds spent per integer, measured by decoding every list in the index. (The HYB+M2 framework is excluded from the following experiments because it only computes intersections.) The methods VByte, Roaring, and RUP are the fastest. However, for the methods encoding a stream of *gaps*, i.e., differences between consecutive integers, we skipped the final prefix-summing scan in this experiment since this represents a fixed overhead for all such methods. These are the methods in the first group, except Interpolative and PEF. The Interpolative mechanism does not feature specific optimizations, except when decoding runs of consecutive integers and is, on average, one order of magnitude slower than the fastest methods.

In Table 3 we also report the result for unions, evaluated using the same set of TREC queries as for intersections. Again, we can see that the solutions partitioned by universe are superior. However, due to the scan-based nature of unions, the performance gap with respect to the solutions partitioned by cardinality is not as high as that for intersections. Excluding Interpolative and δ from the comparison since they are much slower than the others, the mentioned gap is anyway consistent and equal to $3.4 - 4.2\times$, $2.8 - 3.7\times$, and $3 - 4\times$ for Gov2, ClueWeb09, and CCNews respectively. Finally, notice that, for reasons already discussed and its (intentionally) simpler design, Roaring is more efficient than RUP.

References

[1] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *Journal of Experimental Algorithmics (JEA)*, 14:7, 2009.

[2] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1):1, 2010.

[3] B. Ding and A. C. König. Fast set intersection in memory. *Proceedings of the VLDB Endowment*, 4(4):255–266, 2011.

[4] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[5] R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

[6] A. Golynski, A. Orlandi, R. Raman, and S. S. Rao. Optimal indexes for sparse bit vectors. *Algorithmica*, 69(4):906–924, 2014.

[7] A. Kane and F. W. Tompa. Skewed partial bitvectors for list intersection. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 263–272. ACM, 2014.

[8] D. Lemire, L. Boytsov, and N. Kurz. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016. URL `https://github.com/lemire/SIMDCompressionAndIntersection`.

[9] 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 and Experience*, 48(4):867–895, 2018. URL `https://github.com/RoaringBitmap/CRoaring`.

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

[11] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proceedings of the 37th International Conference on Research and Development in Information Retrieval*, pages 273–282, 2014. URL `https://github.com/ot/ds2i`.

[12] G. E. Pibiri. On slicing sorted integer sequences. *CoRR*, abs/1907.01032, 2019. URL `http://arxiv.org/abs/1907.01032`.

[13] G. E. Pibiri and R. Venturini. On optimally partitioning variable-byte codes. *IEEE Transactions on Knowledge and Data Engineering*, 32(9):1812–1823, 2020. URL `https://github.com/jermp/opt_vbyte`.

[14] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):1–36, 2020.

[15] G. E. Pibiri, M. Petri, and A. Moffat. Fast Dictionary-Based Compression for Inverted Indexes. In *International ACM Conference on Web Search and Data Mining*, page 9, 2019. URL `https://github.com/jermp/dint`.

[16] J. Plaisance, N. Kurz, and D. Lemire. Vectorized VByte decoding. In *International Symposium on Web Algorithms*, 2015. URL `https://github.com/lemire/MaskedVByte`.

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

[18] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS@ VLDB*, pages 1–8, 2011.

[19] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, pages 401–410, 2009.

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

[21] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56, 2006.