Better bitmap performance with Roaring bitmaps

S. Chambi[1], D. Lemire[2], O. Kaser[3], R. Godin[1]

[1]*Département d'informatique, UQAM, Montreal, Qc, Canada*
[2]*LICEF Research Center, TELUQ, Montreal, QC, Canada*
[3]*Computer Science and Applied Statistics, UNB Saint John, Saint John, NB, Canada*

SUMMARY

Bitmap indexes are commonly used in databases and search engines. By exploiting bit-level parallelism, they can significantly accelerate queries. However, they can use much memory, and thus we might prefer compressed bitmap indexes. Following Oracle's lead, bitmaps are often compressed using run-length encoding (RLE). Building on prior work, we introduce the *Roaring* compressed bitmap format: it uses packed arrays for compression instead of RLE. We compare it to two high-performance RLE-based bitmap encoding techniques: WAH (Word Aligned Hybrid compression scheme) and Concise (Compressed 'n' Composable Integer Set). On synthetic and real data, we find that Roaring bitmaps (1) often compress significantly better (e.g., 2×) and (2) are faster than the compressed alternatives (up to 900× faster for intersections). Our results challenge the view that RLE-based bitmap compression is best.

KEY WORDS: performance; measurement; index compression; bitmap index

1. INTRODUCTION

A bitmap (or bitset) is a binary array that we can view as an efficient and compact representation of an integer set, S. Given a bitmap of n bits, the ith bit is set to one if the ith integer in the range $[0, n-1]$ exists in the set. For example, the sets $\{3, 4, 7\}$ and $\{4, 5, 7\}$ might be stored in binary form as `10011000` and `10110000`. We can compute the union or the intersection between two such corresponding lists using bitwise operations (OR, AND) on the bitmaps (e.g., `10111000` and `10010000` in our case). Bitmaps are part of the Java platform (`java.util.BitSet`).

When the cardinality of S is relatively large compared to the universe size, n (e.g., $|S| > n/64$ on 64-bit processors), bitmaps are often superior to other comparable data structures such as arrays, hash sets or trees. However, on moderately low density bitmaps ($n/10000 < |S| < n/64$), compressed bitmaps such as Concise can be preferable [1].

Most of the recently proposed compressed bitmap formats are derived from Oracle's BBC [2] and use run-length encoding (RLE) for compression: WAH [3], Concise [1], EWAH [4], COMPAX [5], VLC [6], VAL-WAH [7], etc. Wu et al.'s WAH is probably the best known. WAH divides a bitmap of n bits into $\left\lceil \frac{n}{w-1} \right\rceil$ words of $w-1$ bits, where w is a convenient word length (e.g., $w = 32$). WAH distinguishes between two types of words: words made of just $w-1$ ones (`11···1`) or just $w-1$ zeros (`00···0`), are *fill words*, whereas words containing a mix of zeros and ones (e.g., `101110···1`) are *literal words*. Literal words are stored using w bits: the most significant bit is set to zero and the remaining bits store the heterogeneous $w-1$ bits. Sequences of homogeneous fill words (all ones or all zeros) are also stored using w bits: the most significant bit is set to 1, the

---

*Correspondence to: Daniel Lemire, LICEF Research Center, TELUQ, Université du Québec, 5800 Saint-Denis, Office 1105, Montreal (Quebec), H2S 3L5 Canada. Email: lemire@gmail.com

second most significant bit indicates the bit value of the homogeneous word sequence, while the remaining $w - 2$ bits store the run length of the homogeneous word sequence.

When compressing a sparse bitmap, e.g., corresponding to the set $\{0, 2(w-1), 4(w-1), \ldots\}$, WAH can use $2w$ bits per set bit. Concise reduces this memory usage by half [1]. It uses a similar format except for coded fill words. Instead of storing the run length r using $w - 2$ bits, Concise uses only $w - 2 - \lceil \log_2(w) \rceil$ bits, setting aside $\lceil \log_2(w) \rceil$ bits as *position* bits. These $\lceil \log_2(w) \rceil$ position bits encode a number $p \in [0, w)$. When $p = 0$, we decode $r + 1$ fill words. When it is non-zero, we decode r fill words preceded by a word that has its $(p - 1)^{\text{th}}$ bit flipped compared to the following fill words. Consider the case where $w = 32$. Concise can code the set $\{0, 62, 124, \ldots\}$ using only 32 bits/integer, in contrast to WAH which requires 64 bits/integer.

Though they reduce memory usage, these formats derived from BBC have slow random access compared to an uncompressed bitmap. That is, checking or changing the ith bit value is an $O(n)$-time operation. Thus, though they represent an integer set, we cannot quickly check whether an integer is in the set. This makes them unsuitable for some applications [8]. Moreover, RLE formats have a limited ability to quickly skip data. For example, suppose that we are computing the bitwise AND between two compressed bitmaps. If one bitmap has long runs of zeros, we might wish to skip over the corresponding words in the other bitmap. Without an auxiliary index, this might be impossible with formats like WAH and Concise.

Instead of using RLE and sacrificing random access, we propose to partition the space $[0, n)$ into *chunks* and to store dense and sparse chunks differently [9]. On this basis, we introduce a new bitmap compression scheme called *Roaring*. Roaring bitmaps store 32-bit integers in a compact and efficient two-level indexing data structure. Dense chunks are stored using bitmaps; sparse chunks use packed arrays of 16-bit integers. In our example ($\{0, 62, 124, \ldots\}$), it would use only ≈ 16 bits/integer, half of Concise's memory usage. Moreover, on the synthetic-data test proposed by Colantonio and Di Pietro [1], it is at least four times faster than WAH and Concise. In some instances, it can be hundreds of times faster.

Our approach is reminiscent of O'Neil and O'Neil's RIDBit external-memory system. RIDBit is a B-tree of bitmaps, where an RID list is used instead when a chunk's density is too small. However RIDBit fared poorly compared to FastBit—a WAH-based system [10]: FastBit was up to $10\times$ faster. In contrast to the negative results of O'Neil et al., we find that Roaring bitmaps can be several times faster than WAH bitmaps for in-memory processing. Thus one of our main contributions is to challenge the belief—expressed by authors such as by Colantonio and Di Pietro [1]—that WAH bitmap compression is the most efficient alternative.

A key ingredient in the performance of Roaring bitmaps are the new bit-count processor instructions (such as `popcnt`) that became available on desktop processors more recently (2008). Previously, table lookups were often used instead in systems like RIDBit [11], but they can be several times slower. These new instructions allow us to quickly compute the density of new chunks, and to efficiently extract the location of the set bits from a bitmap.

To surpass RLE-based formats such as WAH and Concise, we also rely on several algorithmic strategies (see § 4). For example, when intersecting two sparse chunks, we may use an approach based on binary search instead of a linear-time merge like RIDBit. Also, when merging two chunks, we predict whether the result is dense or sparse to minimize wasteful conversions. In contrast, O'Neil et al. report that RIDBit converts chunks after computing them [11].

2. ROARING BITMAP

We partition the range of 32-bit indexes ($[0, n)$) into chunks of 2^{16} integers sharing the same 16 most significant digits. We use specialized containers to store their 16 least significant bits.

When a chunk contains no more than 4096 integers, we use a sorted array of packed 16-bit integers. When there are more than 4096 integers, we use a 2^{16}-bit bitmap. Thus, we have two types of containers: an array container for *sparse* chunks and a bitmap container for *dense* chunks. The 4096 threshold insures that at the level of the containers, each integer uses no more than 16 bits: we

either use 2^{16} bits for more than 4096 integers, using less than 16 bits/integer, or else we use exactly 16 bits/integer.

The containers are stored in a dynamic array with the shared 16 most-significant bits: this serves as a first-level index. The array keeps the containers sorted by the 16 most-significant bits. We expect this first-level index to be typically small: when $n = 1\,000\,000$, it contains at most 16 entries. Thus it should often remain in the CPU cache. The containers themselves should never use much more than 8 kB.

Each container keeps track of its cardinality (number of integers) using a counter. Thus computing the cardinality of a Roaring bitmap can be done quickly: it suffices to sum at most $\lceil n/2^{16} \rceil$ counters. It also makes it possible to support rank and select queries faster than with a typical bitmap: rank queries count the number of set bits in a range $[0, i]$ whereas select queries seek the location of the ith set bit.

The overhead due to the containers and the dynamic array means that our memory usage can exceed 16 bits/integer. However, as long as the number of containers is small compared to the total number of integers, we should never use much more than 16 bits/integer. We assume that there are far fewer containers than integers. More precisely, we assume that the density typically exceeds $0.1\,\%$ or that $n/|S| > 0.001$. When applications encounter integer sets with lower density (less than $0.1\,\%$), a bitmap is unlikely to be the proper data structure.

3. ACCESS OPERATIONS

To check for the presence of a 32-bit integer x, we first seek the container corresponding to $x/2^{16}$, using binary search. If a bitmap container is found, we access the $(x \bmod 2^{16})^{\text{th}}$ bit. If an array container is found, we use a binary search again.

We insert and remove an integer x similarly. We first seek the corresponding container. When the container found is a bitmap, we set the value of the corresponding bit and update the cardinality accordingly. If we find an array container, we use a binary search followed by a linear-time insertion or deletion.

When removing an integer, a bitmap container might become an array container if its cardinality reaches 4096. When adding an integer, an array container might become a bitmap container when its cardinality exceeds 4096. When this happens, a new container is created with the updated data while the old container is discarded. Converting an array container to a bitmap container is done by creating a new bitmap container initialized with zeros, and setting the corresponding bit. To convert a bitmap container to an array container, we extract the location of the set bits using an optimized algorithm (see Algorithm 2).

4. LOGICAL OPERATIONS

We implemented various operations on Roaring bitmaps, including union (bitwise OR) and intersection (bitwise AND). A bitwise operation between two Roaring bitmaps consists of iterating and comparing the 16 high-bits integers (keys) on the first-level indexes. For better performance, we maintain sorted first-level arrays. Two keys are compared at each iteration. On equality, a second-level logical operation between the corresponding containers is performed. This always generates a new container. If the container is not empty, it is added to the result along with the common key. Then iterators positioned over the first-level arrays are incremented by one. When two keys are not equal, the array containing the smallest one is incremented by one position, and if a union is performed, the lowest key and a copy of the corresponding container are added to the answer. When computing unions, we repeat until the two first-level arrays are exhausted. And when computing intersections, we terminate as soon as one array is exhausted.

Sorted first-level arrays allows first-level comparisons in $O(n_1 + n_2)$ time, where n_1 and n_2 are the respective lengths of the two compared arrays. We also maintain the array containers sorted for the same advantages. As containers can be represented with two different data structures, bitmaps

and arrays, a logical union or intersection between two containers involves one of the three following scenarios:

Bitmap vs Bitmap: We iterate over 1024 64-bit words. For unions, we perform 1024 bitwise ORs and write the result to a new bitmap container. See Algorithm 1. The resulting cardinality is computed efficiently in Java using the `Long.bitCount` method.

Algorithm 1 Routine to compute the union of two bitmap containers.

1: **input**: two bitmaps A and B indexed as arrays of 1024 64-bit integers
2: **output**: a bitmap C representing the union of A and B, and its cardinality c
3: $c \leftarrow 0$
4: Let C be indexed as an array of 1024 64-bit integers
5: **for** $i \in \{1, 2, \ldots, 1024\}$ **do**
6:     $C_i \leftarrow A_i$ OR B_i
7:     $c \leftarrow c + \text{bitCount}(C_i)$
8: **return** C and c

It might seem like computing bitwise ORs and computing the cardinality of the result would be significantly slower than merely computing the bitwise ORs. However, four factors mitigate this potential problem.

1. Popular processors (Intel, AMD, ARM) have fast instructions to compute the number of ones in a word. Intel and AMD's `popcnt` instruction has a throughput as high as one operation per CPU cycle.

2. Recent Java implementations translate a call to `Long.bitCount` into such fast instructions.

3. Popular processors are superscalar: they can execute several operations at once. Thus, while we retrieve the next data elements, compute their bitwise OR and store it in memory, the processor can apply the `popcnt` instruction on the last result and increment the cardinality counter accordingly.

4. For inexpensive data processing operations, the processor may not run at full capacity due to cache misses.

On the Java platform we used for our experiments, we estimate that we can compute and write bitwise ORs at 700 million 64-bit words per second. If we further compute the cardinality of the result as we produce it, our estimated speed falls to about 500 million words per second. That is, we suffer a speed penalty of about 30 % because we maintain the cardinality. In contrast, competing methods like WAH and Concise must spend time to decode the word type before performing a single bitwise operation. These checks may cause expensive branch mispredictions or impair superscalar execution.

For computing intersections, we use a less direct route. First, we compute the cardinality of the result, using 1024 bitwise AND instructions. If the cardinality is larger than 4096, then we proceed as with the union, writing the result of bitwise ANDs to a new bitmap container. Otherwise, we create a new array container. We extract the set bits from the bitwise ANDs on the fly, using Algorithm 2. See Algorithm 3.

Bitmap vs Array: When one of the two containers is a bitmap and the other one is a sorted dynamic array, the intersection can be computed very quickly: we iterate over the sorted dynamic array, and verify the existence of each 16-bit integer in the bitmap container. The result is written out to an array container. Unions are also efficient: we create a copy of the bitmap and simply iterate over the array, setting the corresponding bits.

Array vs Array: For unions, if the sum of the cardinalities is no more than 4096, we use a merge algorithm between the two arrays. Otherwise, we set the bits corresponding to both arrays in a

---

Algorithm 2 Optimized algorithm to convert the set bits in a bitmap into a list of integers. We assume two-complement's arithmetic. The function bitCount returns the Hamming weight of the integer.

1: **input**: an integer w
2: **output**: an array S containing the indexes where a 1-bit can be found in w
3: Let S be an initially empty list
4: **while** $w \neq 0$ **do**
5:     $t \leftarrow w$ AND $-w$ (cf. [12, p. 12])
6:     append bitCount$(t-1)$ to S
7:     $w \leftarrow w$ AND $(w-1)$ (cf. [12, p. 11])
8: **return** S

Algorithm 3 Routine to compute the intersection of two bitmap containers. The function bitCount returns the Hamming weight of the integer.

1: **input**: two bitmaps A and B indexed as arrays of 1024 64-bit integers
2: **output**: a bitmap C representing the intersection of A and B, and its cardinality c if $c > 4096$ or an equivalent array of integers otherwise
3: $c \leftarrow 0$
4: **for** $i \in \{1, 2, \ldots, 1024\}$ **do**
5:     $c \leftarrow c + \text{bitCount}(A_i \text{ AND } B_i)$
6: **if** $c > 4096$ **then**
7:     Let C be indexed as an array of 1024 64-bit integers
8:     **for** $i \in \{1, 2, \ldots, 1024\}$ **do**
9:        $C_i \leftarrow A_i \text{ AND } B_i$
10:     **return** C and c
11: **else**
12:     Let D be an array of integers, initially empty
13:     **for** $i \in \{1, 2, \ldots, 1024\}$ **do**
14:        append the set bits in A_i AND B_i to D using Algorithm 2
15:     **return** D

---

bitmap container. We then compute the cardinality using fast instructions. If the cardinality is no more than 4096, we convert the bitmap container to an array container (see Algorithm 2).

For intersections, we use a simple merge (akin to what is done in merge sort) when the two arrays have cardinalities that differ by less than a factor of 64. Otherwise, we use galloping intersections [8]. The result is always written to a new array container. Galloping is superior to a simple merge when one array (r) is much smaller than other one (f) because it can skip many comparisons. Starting from the beginning of both arrays, we pick the next available integer r_i from the small array r and seek an integer at least as large f_j in the large array f, looking first at the next value, then looking at a value twice as far, and so on. Then, we use binary search to advance in the second list to the first value larger or equal to r_i.

We can also execute some of these operations *in place*:

- When computing the union between two bitmap containers, we can modify one of the two bitmap containers instead of generating a new bitmap container. Similarly, for the intersection between two bitmap containers, we can modify one of the two containers if the cardinality of the result exceeds 4096.

- When computing the union between an array and a bitmap container, we can write the result to the bitmap container, by iterating over the values of the array container and setting the corresponding bits in the bitmap container. We can update the cardinality each time by checking whether the word value has been modified.

In-place operations can be faster because they avoid allocating and initializing new memory areas.

When aggregating many bitmaps, we use other optimizations. For example, when computing the union of many bitmaps (e.g., hundreds), we first locate all containers having the same key (using a priority queue). If one such container is a bitmap container, then we can clone this bitmap container (if needed) and compute the union of this container with all corresponding containers in-place. In this instance, the computation of the cardinality can be done once at the end. See Algorithm 4.

Algorithm 4 Optimized algorithm to compute the union of many roaring bitmaps

---

 1: **input**: a set R of Roaring bitmaps as collections of containers; each container has a cardinality and a 16-bit key
 2: **output**: a new Roaring bitmap T representing the union
 3: Let T be an initially empty Roaring bitmap.
 4: Let P be the min-heap of containers in the bitmaps of R, configured to order the containers by their 16-bit keys.
 5: **while** P is not empty **do**
 6:     Let x be the root element of P. Remove from the min-heap P all elements having the same key as x, and call the result Q.
 7:     Sort Q by descending cardinality; Q_1 has maximal cardinality.
 8:     Clone Q_1 and call the result A. The container A might be an array or bitmap container.
 9:     **for** $i \in \{2, \ldots, |Q|\}$ **do**
10:         **if** A is a bitmap container **then**
11:             Compute the in-place union of A with Q_i: $A \leftarrow A$ OR Q_i. Do not re-compute the cardinality of A: just compute the bitwise-OR operations.
12:         **else**
13:             Compute the union of the array container A with the array container Q_i: $A \leftarrow A$ OR Q_i. If A exceeds a cardinality of 4096, then it becomes a bitmap container.
14:     If A is a bitmap container, update A by computing its actual cardinality.
15:     Add A to the output Roaring bitmap T.
16: **return** T

---

5. EXPERIMENTS

We performed a series of experiments to compare the time-space performance of Roaring bitmaps with the performance of other well-known bitmap indexing schemes: Java's `BitSet`, WAH and Concise. We use the CONCISE Java library for WAH and Concise (version 2.2) made available by Colantonio and Di Pietro [1].

Our Roaring-bitmap implementation code and data is freely available at `http://roaringbitmap.org/`. Our software has been thoroughly tested: our Java library has been adopted by major database systems like Apache Spark [13] and Druid [14].

Benchmarks were performed on an AMD FX$^{\text{TM}}$-8150 eight-core processor running at 3.60 GHz and having 16 GB of RAM. We used the Oracle Server JVM version 1.7 on Linux Ubuntu 12.04.1 LTS. All our experiments run entirely in memory.

To account for the just-in-time compiler in Java, we first run tests without recording the timings. Then we repeat the tests several times and report an average.

5.1. Synthetic experiments

We began by reproducing Colantonio and Di Pietro's synthetic experiments [1]. However, while they included alternative data structures such as Java's `HashSet`, we focus solely on bitmap formats for simplicity. Our results are generally consistent with Colantonio and Di Pietro's results, given the fact that we have a better processor.

(a) Compression: uniform distribution

(b) Compression: $\text{Beta}(0.5, 1)$ distribution

(c) Intersection: discretized $\text{Beta}(0.5, 1)$ distribution

(d) Union: discretized $\text{Beta}(0.5, 1)$ distribution

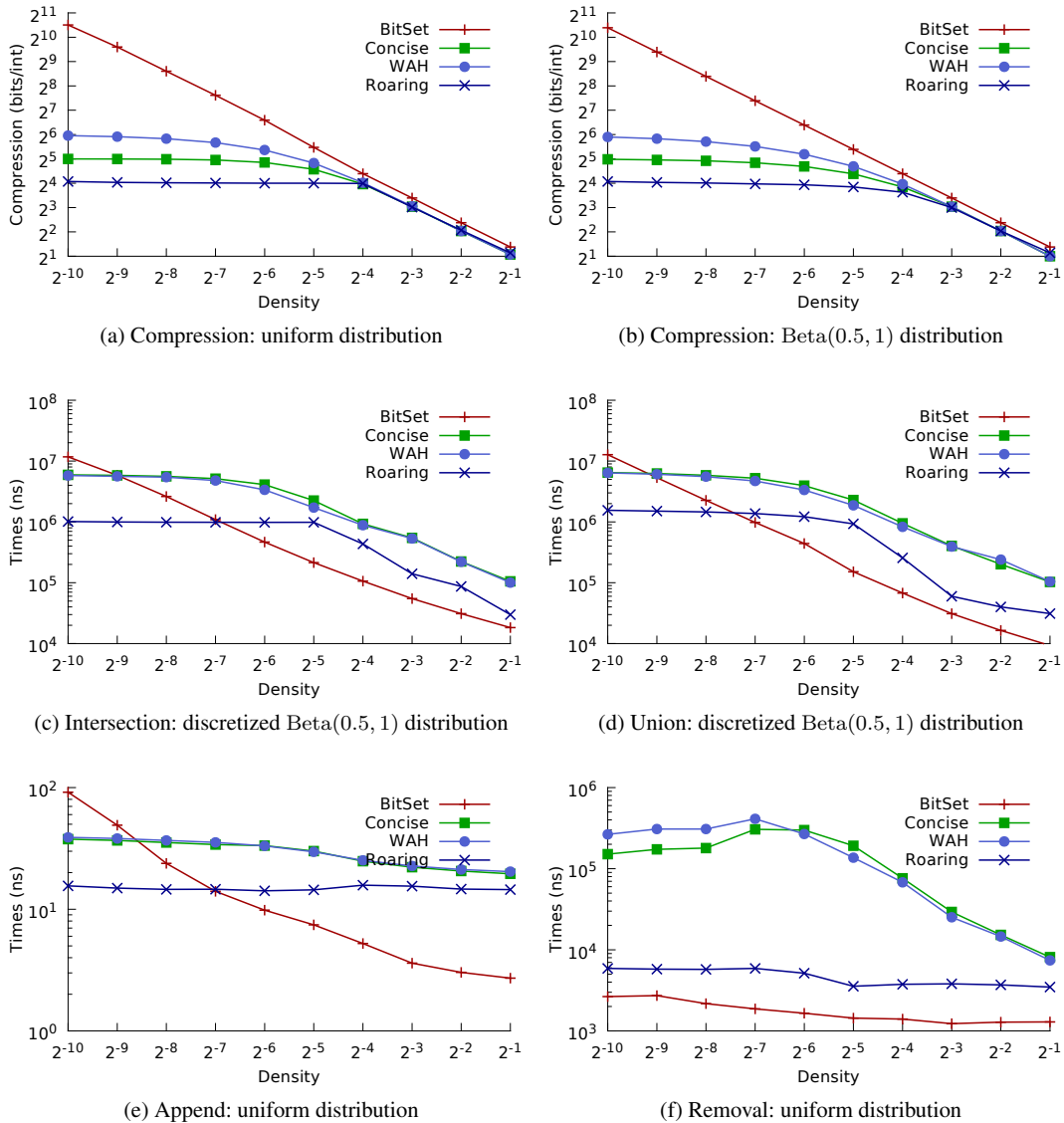(e) Append: uniform distribution

(f) Removal: uniform distribution

Figure 1. Times and compression measurements: average of 100 runs

Data sets of 10^5 integers were generated according to two synthetic data distributions: uniform and discretized $\text{Beta}(0.5, 1)$ distributions. (Colantonio and Di Pietro described the latter as a *Zipfian* distribution.) The four schemes were compared on several densities d varying from 2^{-10} to 0.5. To generate an integer, we first picked a floating-point number y pseudo-randomly in $[0, 1)$. When we desired a uniform distribution, we added $\lfloor y \times \max \rfloor$ to the set. In the β-distributed case, we added $\lfloor y^2 \times \max \rfloor$. The value max represents the ratio between the total number of integers to generate and the desired density (d) of the set, i.e.: $\max = 10^5/d$. Because results on uniform and $\text{Beta}(0.5, 1)$ distributions are often similar, we do not systematically present both.

We stress that our data distributions and benchmark closely follow Colantonio and Di Pietro's work [1]. Since they used this benchmark to show the superiority of Concise over WAH, we feel that it is fair to use their own benchmark to assess our own proposal against Concise.

Figs. 1a and 1b show the average number of bits used by Java's `BitSet` and the three bitmap compression techniques to store an integer in a set. In these tests, Roaring bitmaps require 50 % of the space consumed by Concise and 25 % of WAH space on sparse bitmaps.

Table I. Sampled bitmap characteristics and Roaring size.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Rows	4 277 807	199 523	1 178 559	1 015 367
Density	1.2×10^{-3}	1.7×10^{-1}	1.3×10^{-3}	6.4×10^{-2}
Bits/Item	18.7	2.92	22.3	5.83

The `BitSet` class uses slightly more memory even for dense bitmaps in our tests. This is due to its memory allocation strategy that doubles the size of the underlying array whenever more storage is required. We could recover this wasted memory by cloning the newly constructed bitmaps. Our roaring bitmap implementation has a `trim` method that can be used to get the same result. We did not call these methods in these tests.

We also report on intersection and union times. That is, we take two bitmaps and generate a new bitmap representing the intersection or union. For the `BitSet`, it means that we first need to create a copy (using the `clone` method), since bitwise operations are in-place. Figs. 1c and 1d present the average time in nanoseconds to perform intersections and unions between two sets of integers. Roaring bitmaps are $\times 4 - \times 5$ times faster than Concise and WAH for intersections on all tested densities. Results for unions are similar except that for moderate densities ($2^{-5} \le d \le 2^{-4}$), Roaring is only moderately (30 %) faster than Concise and WAH. `BitSet` outperforms the other schemes on dense data, but it is $> 10\times$ slower on sparse bitmaps.

We measured times required by each scheme to add a single element a to a sorted set S of integers, i.e.: $\forall i \in S : a > i$. Fig. 1e shows that Roaring requires less time than WAH and Concise. Moreover, WAH and Concise do not support the efficient insertion of values in random order, unlike Roaring bitmaps. Finally, we measured the time needed to remove one randomly selected element from an integers set (Fig. 1f). We observe that Roaring bitmaps have much better results than the two other compressed formats.

5.2. Real-data experiments

Tables I–II present results for the five real datasets used an earlier study of compressed bitmap indexes [15]. There are only two exceptions:

- We only use the September 1985 data for the WEATHER dataset (an approach others have used before [16]), which was otherwise too large for our test environment.

- We omitted the CENSUS2000 dataset because it contains only bitmaps having an average cardinality of 30 over a large universe ($n = 37 019 068$). It is an ill-suited scenario for bitmaps. Because of the structure overhead, Roaring bitmaps used $4\times$ as much memory as Concise bitmaps. Still, Roaring bitmaps were about $4\times$ faster when computing intersections.

The dataset were taken as-is: we did not sort them prior to indexing.

For each dataset, a bitmap index was built. Then we chose 200 bitmaps from the index, using an approach similar to stratified sampling to control for the large range of attribute cardinalities. We first sampled 200 attributes, with replacement. For each sampled attribute, we selected one of its bitmaps uniformly at random. The 200 bitmaps were used as 100 pairs of inputs for 100 pairwise ANDs and ORs; Tables IIb–IIc show the time factor increase if Roaring is replaced by `BitSet`, WAH or Concise. (Values below 1.0 indicate cases where Roaring was slower.) Table IIa shows the storage factor increase when Roaring is replaced by one of the other approaches.

Roaring bitmaps are always faster, on average, than WAH and Concise. On two datasets (CENSUS1881 and WIKILEAKS), Roaring bitmaps are faster than `BitSet` while using much less memory ($40\times$ less). On the two other datasets, `BitSet` is more than twice as fast as Roaring, but it also uses three times as much memory. When comparing the speed of `BitSet` and Roaring, consider that Roaring pre-computes the cardinality at a chunk level. Thus if we need the cardinality of the aggregated bitmaps, Roaring has the advantage. On the WIKILEAKS dataset, Concise and

Table II. Results on real data

(a) Size expansion if Roaring is replaced with other schemes.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	2.2	1.4	0.79	1.4
WAH	2.4	1.6	0.79	1.5
BitSet	42	2.9	55	3.5

(b) Time increase, for AND, if Roaring is replaced with other schemes.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	920	6.6	8.3	6.3
WAH	840	5.9	8.2	5.4
BitSet	730	0.42	28	0.64

(c) Time increases, for OR, if Roaring is replaced with other schemes.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	34	5.4	2.1	3.9
WAH	31	4.9	2.1	3.4
BitSet	29	0.43	6.7	0.48

WAH offer better compression than Roaring (by about 30%). This is due to the presence of long runs of ones ($11\cdots1$ fill words), which Roaring does not compress.

Results on the CENSUS1881 dataset are striking: Roaring is up to $900\times$ faster than the alternatives. This is due to the large differences in the cardinalities of the bitmaps. When intersecting a sparse bitmap with a dense one, Roaring is particularly efficient.

6. CONCLUSION

In this paper, we introduced a new bitmap compression scheme called Roaring. It stores bitmap set entries as 32-bit integers in a concise two-level index. In comparison with two competitive bitmap compression schemes, WAH and Concise, Roaring often uses less memory and is faster.

In future work, we will consider improving the performance further. We might add new types of containers. In particular, we might make use of fast packing techniques to optimize the storage use of the array containers [17]. We should also review other operations beside intersections and unions, such as threshold queries [18].

We plan to investigate further applications in information retrieval. There are reasons to be optimistic: Apache Lucene (as of version 5.0) has adopted a Roaring format [19] to compress document identifiers.

REFERENCES

1. Colantonio A, Di Pietro R. Concise: Compressed 'n' Composable Integer Set. *Information Processing Letters* 2010; **110**(16):644–650, doi:10.1016/j.ipl.2010.05.018.
2. Antoshenkov G. Byte-aligned bitmap compression. *DCC'95*, IEEE Computer Society: Washington, DC, USA, 1995; 476.
3. Wu K, Stockinger K, Shoshani A. Breaking the curse of cardinality on bitmap indexes. *SSDBM'08*, Springer, 2008; 348–365.
4. Lemire D, Kaser O, Aouiche K. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering* 2010; **69**(1):3–28, doi:10.1016/j.datak.2009.08.006.

5. Fusco F, Stoecklin MP, Vlachos M. NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic. *Proceedings of the VLDB Endowment* 2010; **3**:1382–1393, doi:10.14778/1920841.1921011.

6. Corrales F, Chiu D, Sawin J. Variable length compression for bitmap indices. *DEXA'11*, Springer-Verlag: Berlin, Heidelberg, 2011; 381–395.

7. Guzun G, Canahuate G, Chiu D, Sawin J. A tunable compression framework for bitmap indices. *ICDE'14*, IEEE, 2014.

8. Culpepper JS, Moffat A. Efficient set intersection for inverted indexing. *ACM T. Inform. Syst.* Dec 2010; **29**(1):1:1–1:25, doi:10.1145/1877766.1877767.

9. Kaser O, Lemire D. Attribute value reordering for efficient hybrid OLAP. *Inform. Sciences* 2006; **176**(16):2304–2336.

10. O'Neil E, O'Neil P, Wu K. Bitmap index design choices and their performance implications. *IDEAS'07*, 2007; 72–84.

11. Rinfret D, O'Neil P, O'Neil E. Bit-sliced index arithmetic. *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, ACM: New York, NY, USA, 2001; 47–57, doi:10.1145/375663.375669. URL http://doi.acm.org/10.1145/375663.375669.

12. Warren HS Jr. *Hacker's Delight*. 2nd ed. edn., Addison Wesley, 2013.

13. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, USENIX Association: Berkeley, CA, USA, 2010; 10–10. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

14. Yang F, Tschetter E, Léauté X, Ray N, Merlino G, Ganguli D. Druid: A real-time analytical data store. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, ACM: New York, NY, USA, 2014; 157–168, doi:10.1145/2588555.2595631. URL http://doi.acm.org/10.1145/2588555.2595631.

15. Lemire D, Kaser O, Gutarra E. Reordering rows for better compression: Beyond the lexicographical order. *ACM Transactions on Database Systems* 2012; **37**(3), doi:10.1145/2338626.2338633. Article 20.

16. Beyer K, Ramakrishnan R. Bottom-up computation of sparse and iceberg CUBEs. *SIGMOD Record* 1999; **28**(2):359–370, doi:10.1145/304181.304214.

17. Lemire D, Boytsov L. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 2013; doi:10.1002/spe.2203. In Press.

18. Kaser O, Lemire D. Compressed bitmap indexes: beyond unions and intersections. *Software: Practice and Experience* 2014; doi:10.1002/spe.2289. In Press.

19. Grand A. LUCENE-5983: RoaringDocIdSet. https://issues.apache.org/jira/browse/LUCENE-5983 [Last checked December 2014.] 2014.