Consistently faster and smaller compressed bitmaps with Roaring

D. Lemire[1], G. Ssi-Yan-Kai[2], O. Kaser[3]

[1] *LICEF Research Center, TELUQ, Montreal, QC, Canada*
[2] *42 Quai Georges Gorse, Boulogne Billancourt, France*
[3] *Computer Science and Applied Statistics, UNB Saint John, Saint John, NB, Canada*

SUMMARY

Compressed bitmaps indexes are used in databases and search engines. A wide range of bitmap compression techniques has been proposed, almost all relying primarily on run-length encoding (RLE), including BBC and WAH. However, on unsorted data, we can get superior performance with a hybrid compression technique that uses both uncompressed bitmaps and packed arrays inside a two-level tree. An instance of this technique, Roaring, has recently been proposed. Due to its good performance, it has been adopted by several production platforms (e.g., Apache Lucene, Apache Kylin and Druid).

Yet there are cases where run-length encoded bitmaps are smaller than the original Roaring bitmaps—typically when the data is sorted so that the bitmaps contain long compressible runs. To better cope with these cases, we build a new Roaring hybrid that combines uncompressed bitmaps, packed arrays and RLE compressed segments. The result is a new Roaring format that compresses better.

Overall, our new implementation of Roaring can be several times faster (up to two orders of magnitude) than the implementations of traditional RLE-based alternatives (WAH, Concise, EWAH) while compressing better. We review the design choices and optimizations that make these good results possible.

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

1. INTRODUCTION

Sets are a fundamental abstraction in software. They can be implemented in various ways, as hash sets, as trees, and so forth. In databases and search engines, sets are often an integral part of indexes. For example, we may need to maintain a set of all documents or rows (represented by numerical identifier) that satisfy some property. Besides adding or removing elements from the set, we need fast functions to compute the intersection, the union, the difference between sets, and so on.

To implement a set of integers, a particularly appealing strategy is the bitmap (also called bitset or bit vector). Using n bits, we can represent any set made of the integers from the range $[0, n)$: it suffices to set the i^{th} bit is set to one if integer i is present in the set. Commodity processors use words of $W = 32$ or $W = 64$ bits. By combining many such words, we can support large values of n. Intersections, unions and differences can then be implemented as bitwise AND, OR and AND NOT operations. More complicated set functions can also be implemented as bitwise operations [1]. When the bitset approach is applicable, it can be orders of magnitude faster than other possible implementation of a set (e.g., as a hash set) while using several times less memory [2].

Unfortunately, conventional bitmaps are only applicable when the cardinality of the set ($|S|$) is relatively large compared to the universe size (n), e.g., $|S| > n/64$. They are also suboptimal when the set is made of a few ranges of consecutive values (e.g., $S = \{1, 2, 3, 4, \ldots, 99, 100\}$).

[*] 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

Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada; contract/grant number: 261437

One popular approach has been to compress bitmaps with run-length encoding. Effectively, instead of using $\lceil n/W \rceil$ words of W-bits for all bitmaps, we look for runs of consecutive words containing only ones or only zeros, and we replace them with markers that indicate which value is being repeated, and how many repetitions there are. This approach was first put into practice by Oracle's BBC [3], using 8-bit words. Starting with WAH [4], there have been many variations on this idea, including Concise [2], EWAH [5], COMPAX [6], and so on. (See § 2.)

When processing such RLE-compressed formats, one may need to read every compressed word to determine whether a value is present in the set. Moreover, computing the intersection or union between two bitmaps B_1 and B_2 has complexity $O(|B_1| + |B_2|)$ where $|B_1|$ and $|B_2|$ are the compressed sizes of the bitmaps. This complexity is worse than that of a hash set, where we can compute an intersection with an expected-time complexity of $O(\min(|S_1|, |S_2|))$ where $|S_1|$ and $|S_2|$ are the cardinalities of the sets. Indeed, it suffices to iterate over the smallest sets, and for each value, check whether it is present in the larger set. Similarly, we can compute an in-place union, where the result is stored in the largest hash set, simply by inserting all of the values from the small set in the large set, in expected time $O(\min(|S_1|, |S_2|))$. It is comparatively more difficult to compute in-place unions with RLE-compressed bitmaps such as Concise or WAH. Moreover, checking whether a given value is present in an RLE-compressed bitmap like Concise or WAH may require a complete scan of the entire bitmap in $O(|B|)$ time. Such a scan can be hundreds of times slower than checking for the presence of a value in an uncompressed bitmap or hash map.

For better performance, Chambi et al. [7] proposed the Roaring bitmap format, and made it available as an open-source library.[†] Roaring partitions the space $[0, n)$ into *chunks* of 2^{16} integers ($[0, 2^{16}), [2^{16}, 2 \times 2^{16}), \ldots$). Each set value is stored in a container corresponding to its chunk. Roaring stores dense and sparse chunks differently. Dense chunks (containing more than 4096 integers) are stored using conventional bitmap containers (made of 2^{16} bits or $8\,\mathrm{kB}$) whereas sparse chunks use smaller containers made of packed sorted arrays of 16-bit integers. All integers in a chunk share the same 16 most-significant bits. The containers are stored in an array along with the most-significant bits. Though we refer to a Roaring bitmap as a bitmap, it is a hybrid data structure, combining uncompressed bitmaps with sorted arrays.

Roaring allows fast random access. To check for the presence of a 32-bit integer x, we seek a container corresponding to the 16 most significant bits of x, using binary search. If a bitmap container is found, we check the corresponding bit (at index $x \bmod 2^{16}$); if an array container is found, we use a binary search. Similarly, we can compute the intersection between two Roaring bitmaps without having to access all of the data. Indeed, suppose that we have a Roaring bitmap B_1 containing few values in the range $[0, 2^{16})$,which implies it uses an array container. Suppose we have another Roaring bitmap B_2 over the same range containing many values; it can only use a bitmap container. In that case, computing the intersection between B_1 and B_2 can be done in time $O(|B_1|)$, since it suffices to iterate over the set values of B_1 and check whether the corresponding bit is set in the bitmap container of B_2. Moreover, to compute the in-place union of B_1 and B_2, where the result is stored in the large bitmaps (B_2), it suffices to iterate through the values of B_1 and set the corresponding bits in B_2 to 1, in time $O(|B_1|)$. Checking whether a value is present in a Roaring bitmap is relatively fast. It suffices to locate the corresponding container (by a binary search). If the container is a bitmap, a simple bit-value check suffices, otherwise another binary search through a packed array provides the desired answer.

Fast intersection, union and difference operations are made possible through fast operations between containers, even when these containers have different types (i.e., bitmap-vs-bitmap, array-vs-array, array-vs-bitmap, bitmap-vs-array). Though conceptually simple, these operations between containers must produce new containers that are either arrays or bitmap containers. Because converting between container types is potentially expensive, we found it useful to predict the container type as part of the computation. For example, if we must compute the union between two array containers such that the sum of their cardinalities exceed 4096, we preemptively create a

[†] https://github.com/lemire/RoaringBitmap

bitmap container and store the result of the union. Only if the resulting cardinality falls below 4096 do we convert back the result to an array container. (See § 5 for more details.)

We found that Roaring could compress better than WAH and Concise while providing superior speed (up to two orders of magnitude). Consequently, many widely used systems have since adopted Roaring: Apache Lucene [8] and its extensions (Solr, Elastic), Apache Kylin, Druid [9], and so forth.

However, Roaring has a limitation in some scenarios because it does not compress long runs of values. Indeed, given a bitset made of a few long runs (e.g., all integers in $[10, 1000]$), Roaring—as presented so far—can only offer suboptimal compression. This was reported in Chambi et al. [7]: in one dataset, Concise and WAH offer better compression than Roaring (by about $30\,\%$). Practitioners working with Druid also reported to us that Roaring could use more space (e.g., $20\,\%$ or more) than Concise. As previously reported [7], even when Roaring offers suboptimal compression, it can still be expected to be faster for many important operations (see § 6.6). Nevertheless, there are cases where storing the information as runs will both reduce memory usage drastically and accelerate the computation. If we consider the case of a bitmap made of all integers in $[10, 1000]$, Roaring without support for runs would use $8\,\mathrm{kB}$, whereas a few bytes ought to suffice. Such unnecessarily large bitmaps can stress memory bandwidth. Moreover, computing the intersection of two bitmaps representing the ranges $[10, 1000]$ and $[500, 10000]$ can be done in a few cycles when using RLE-compressed bitmaps, but the original Roaring would require intersecting two bitmap containers and possibly thousands of cycles. Thus, there are cases where bitmaps without support for run compression are clearly at a disadvantage and they are sometimes seen in practice.

To solve this problem, we decided to add a third type of container to Roaring, one that is ideally suited to coding data made of runs of consecutive values. The new container is conceptually simple, given a run (e.g., $[10, 1000]$), we simply store the starting point (10) and its length minus one (1000). By packing the starting points and the lengths in pairs, using 16 bits each, we preserve the ability to support fast random access by a binary search algorithm through the coded runs.

Adding a third container type introduces several engineering problems, however. For one thing, instead of a handful of possible container type interactions (bitmap-bitmap, array-array and array-bitmap), we have about twice as many (bitmap-bitmap, array-array, array-bitmap, run-bitmap, run-array and run-run). Keeping in mind that we need to predict the container type as part of the computation to avoid expensive container conversions, new heuristics are needed. It is also not a priori clear whether introducing a new container type could comprehensively solve our compression issues. Moreover, assuming that a new Roaring format improves compression, is it at the expense of speed?

Thankfully, we are able to successfully implement a new Roaring model, made of three container types, that is superior in almost every way to WAH, Concise and EWAH in all our tests—including cases where the original Roaring performs more poorly. Compared to the original Roaring, the new Roaring can improve the compression ratios by up to an order of magnitude. In what might be our worst-case scenario (CENSUS1881), the new version has half the speed while saving only about 5% of storage: even in this case, the new version remains faster by one or two orders of magnitude when compared to an implementation of WAH and Concise. Thus we conclude that this new Roaring is consistently faster and smaller than popular RLE-based compression schemes (WAH, Concise and EWAH).

2. RELATED WORK

There are many RLE-based compression formats. For example, WAH organizes the data in literal and fill words. Literal words contain a mix of $W - 1$ zeros and ones (e.g., $01011 \cdots 01$) where W denotes the processor word size in bits: typically $W = 32$ or $W = 64$. Fill words are made of just $W - 1$ ones or just $W - 1$ zeros (i.e., $11 \cdots 11$ or $00 \cdots 00$). WAH compresses sequences of consecutive identical fill words. The most significant bit of each word distinguishes between fill and literal words. When it is set to one, the remaining $W - 1$ bits store, literally, the $W - 1$ bits of a literal word. When it is set to zero, the second most significant bit indicates the bit value whereas the remaining bits are used to store the number of consecutive identical fill words (the run length).

Concise is a variation that reduces the memory usage when the bitmap is sparse [2]. Instead of storing the run length using $W-2$ bits, Concise uses only $W-2-\lceil\log_2(W)\rceil$ bits to indicate a run length r, reserving $\lceil\log_2(W)\rceil$ bits to store a value p. When p is non-zero, we decode r fill words, plus a single $W-1$ bit word with its p^{th} bit flipped. Whereas WAH would use 64 bits per value to store the set $\{0, 62, 124, \ldots\}$, Concise would only use 32 bits per value.

EWAH is similar to WAH except that it uses a marker word that indicates the number of fill words to follow, their type, as well as the number of literal words to follow. Unlike WAH and Concise, which represent the bitmap as a series of $W-1$-bit words, EWAH uses W-bit words. The EWAH format [5] supports a limited form of skipping because it uses marker words to record the length of the sequences of fill and literal words. Thus, if there are long sequences of literal words, one does not need to access them all with EWAH when seeking data that is further along. Guzun et al. [10] found that EWAH offers better speed than WAH and Concise.

The general idea behind Roaring—using different container types depending on the data characteristics—is not novel. It is similar to O'Neil and O'Neil's RIDBit external-memory system: a B-tree of bitmaps, where a list is used instead when the density of a chunk is too small [11, 12]. Similarly, Culpepper and Moffat proposed a hybrid inverted index (HYB+M2) where some sets of document identifiers are stored as compressed arrays whereas others are stored as bitmaps [13]. To reduce cache misses, Lemire et al. partitioned the Culpepper-Moffat index into chunks that fit in L3 cache [14], effectively creating sets of bitmaps and arrays. Roaring is also reminiscent of C-store where different columns are stored in different formats depending on their data characteristics (number of runs, number of distinct values) [15]. It is likely that we could find many similar instances.

3. APPLICATION CONTEXT

Sets can be used for many purposes. We are interested by applications that use sets as part of an index. For example, one might index an attribute in a database or a word in a set of documents. Though we can expect updates, we assume that most of the processing is spent answering queries that do not require modifying the set. Thus, our application setting is one that might be described as analytical as opposed to transactional.

If we can assume that sets are immutable in the normal course of the application, this has the added benefit of simplifying parallelisation and concurrency. Moreover, the sets can be stored on disk and memory-mapped on demand.

We need to provide functions that enable the creation of the sets, as well as their serialization (for later reuse). Otherwise, we are most interested by the union and intersection between two or more sets, as a new set. It is not uncommon to need to process many more than two sets at once. We are also interested in random-value accesses, e.g., checking whether a value is contained in a set—primarily because RLE-based schemes make these queries particularly expensive.

4. ROARING BITMAP

For a detailed presentation of the original Roaring model, we refer the interested reader to Chambi et al. [7]. We summarize the main points and focus on the new algorithms and their implementations.

Roaring bitmaps are used to represent sets of 32-bit unsigned integers. At a high level, a Roaring bitmap implementation is a key-value data structure where each key-value pair represents the set S of all 32-bit integers that share the same highest 16 bits. The key is made of the shared 16 bits, whereas the value is a container storing a list of the remaining 16 least significant bits for each member of S. In our actual implementation, the key-value store is implemented as two arrays: an array of packed 16-bit values and an array of containers.

The arrays expand dynamically in a standard manner when there are insertions. Alternatively, we could use a tree structure for faster insertions, but we expect Roaring bitmaps to be immutable for most of the life of an application. An array minimizes storage. In a system such as Druid, the bitmaps are created, stored on disk and then memory-mapped as needed.

The structure of each container is straightforward (by design):

- A bitmap container is an object made of 1024 64-bit words (using $8\,\mathrm{kB}$) representing an uncompressed bitmap, able to store all sets of 16-bit integers. The container can be serialized as an array of 64-bit words. We also use a counter to record how many bits are set to 1, and this counter is kept up-to-date.

  Counting the number of 1-bits in a word can be relatively expensive if done naïvely, but modern processors have bit-count instructions—such as `popcnt` for x64 processors and `cnt` for the 64-bit ARM architecture—that can do this count, sometimes using as little as a single clock cycle. According to our tests, using dedicated processor instructions can be several times faster than using either tabulation or other conventional alternatives [16]. Henceforth, we refer to such a function as bitCount: it is provided in Java as the `Long.bitCount` intrinsic. We assume that the platform has a fast bitCount function.

- An array container is an object containing a counter keeping track of the number of integers followed by a packed array of sorted 16-bit unsigned integers. It can be serialized as a 16-bit counter followed by the corresponding number of 16-bit values.

  We implement array containers as dynamic arrays that grow their capacity using a standard approach. That is, we keep a count of the used entries in an underlying array that has typically some excess capacity. When the array needs to grow beyond its capacity, we allocate a larger array and copy the data to this new array. Our allocation heuristic is as follow: when the capacity is small (less than 64 entries), we double the capacity; when the capacity is moderate (between 64 and 1067 entries), we multiply the capacity by $3/2$; when the capacity is large (1067 entries and more), we multiply the capacity by $5/4$. Furthermore, we never allocate more than the maximum needed (4096) and if we are within one sixteenth of the maximum (> 3840), then we allocate the maximum right away (4096) to avoid any future reallocation. A simpler heuristic where we double the capacity whenever it is insufficient would be faster, but it might use more memory than needed. When the array container is no longer expected to grow, the programmer can use a `trim` function to copy the data to a new array with no excess capacity.

- Our new addition, the run container, is made of a packed array of pairs of 16-bit integers. The first value of each pair represents a starting value, whereas the second value is the length of a run. For example, we would store the values $11, 12, 13, 14, 15$ as the pair $11, 4$ where 4 means that beyond 11 itself, there are 4 contiguous values that follow. In addition to this packed array, we need to maintain the number of runs stored in the packed array. Like the array container, the run container is stored in a dynamic array. During serialization, we write out the number of runs, followed by the corresponding packed array.

  Unlike array or bitmap containers, a run container does not keep track of its cardinality; its cardinality can be computed on the fly by summing the lengths of the runs. In most applications, we expect the number of runs to be small: the computation of the cardinality, when needed, should be fast.

No container ever uses much more than $8\,\mathrm{kB}$ of memory. Several such small containers fit the L1 CPU cache of most processors: the last Intel desktop processor to have less than $64\,\mathrm{kB}$ of total (data and code) L1 cache was the P6 created in 1995, whereas most mobile processors have $32\,\mathrm{kB}$ (e.g., NVidia, Qualcomm) or $64\,\mathrm{kB}$ (e.g., Apple) of total L1 cache.

When starting from an empty Roaring bitmap, if a value is added, an array container is created. When inserting a new value in an array container, if the cardinality exceeds 4096, then the container is transformed into a bitmap container. In reverse, if a value is removed from a bitmap container so that its size falls under 4096 integers, then it is transformed into an array container. Whenever a container becomes empty, it is removed from the top-level key-value structure along with the corresponding key.

Thus, when first creating a Roaring bitmap, it is usually made of array and bitmap containers. Runs are not compressed. Upon request, the storage of the Roaring bitmap can be optimized. This triggers a scan through the array and bitmap containers that converts them, if helpful, to run containers. In a given application, this might be done prior to storing the bitmaps as immutable objects to be queried. Run containers may also arise from calling a function to add a range of values.

To decide the best container type, we are motivated to minimize storage. In serialized form, a run container uses $2 + 4r$ bytes given r runs, a bitmap container always uses 8192 bytes and an array container uses $2c + 2$ bytes, where c is the cardinality. Therefore, we apply the following rules:

- All array containers are such that they use no more space than they would as a bitmap container: they contain no more than 4096 values.
- Bitmap containers use less space than they would as array containers: they contain more than 4096 values.
- A run container is only allowed to exist if it is smaller than either the array container or the bitmap container that could equivalently store the same values. If the run container has cardinality greater than 4096 values, then it must contain no more than $\lceil (8192 - 2)/4 \rceil = 2047$ runs. If the run container has cardinality no more than 4096, then the number of runs must be less than half the cardinality.

Counting the number of runs A critical step in deciding whether an array or bitmap container should be converted to a run container is to count the number of runs of consecutive numbers it contains. For array containers, we count this number by iterating through the 16-bit integers and comparing them two by two in a straight-forward manner. Because array containers are limited to thousands of integers at the most, this computation is expected to be fast. For bitmap containers, Algorithm 1 shows how to compute the number of runs. We can illustrate the core operation of the algorithm using a single 32-bit word containing 6 runs of consecutive ones:

$$\begin{aligned} C_i &= \texttt{000111101111001011111011111000001}, \\ C_i \ll 1 &= \texttt{001111011110010111110111110000010}, \\ (C_i \ll 1) \text{ ANDNOT } C_i &= \texttt{001000010000010100000100000000010}. \end{aligned}$$

We can verify that $\mathrm{bitCount}((C_i \ll 1) \text{ ANDNOT } C_i) = 6$, that is, we have effectively computed the number of runs. In the case where a run continues up to the left-most bit, and does not continue in the next word, it does not get counted, but we add another term ($(C_i \gg 63) \text{ ANDNOT } C_{i+1}$ when using 64-bit words) to check for this case. We use only a few instructions for each word. Nevertheless, the computation is potentially expensive—exceeding the cost of computing the union or intersection between two bitmap containers. Thus, instead of always computing exactly the number of runs, we first determine the maximal number of runs needed for a conversion from bitmap container to run container and only the exact number of runs when it is under this threshold. The threshold is easily determined statically since a bitmap container uses up to 8 kB whereas a run container uses up to $2 + 4r$ bytes: any bitmap containing more than 2047 runs should not be converted to a run container. We found that a good heuristic is to compute the number of runs in blocks of 128 words using a function inspired by Algorithm 1. We proceed block by block. As soon as the number of runs exceeds the threshold, we conclude that converting to a run container is counterproductive and abort the computation of the number of runs. If the number of runs is small enough, it is finally computed exactly. We could also have applied to optimization to array containers as well, stopping the count of the number of runs at 2047, but this optimization is likely less useful because array containers have small cardinality compared to bitmap containers. A further possible optimization is to omit the last term from the sum in line 5 of Algorithm 1, thus underestimating the number of runs, typically by a few percent, but by up to 1023 in the worst case. If the underestimate exceeds 2047 runs, it is pointless to determine the exact run count. Computing this lower bound is

Algorithm 1 Routine to compute the number of runs in a bitmap. The left and right shift operators ($\ll$ and $\gg$) move all bits in a word by the specified number of bits, shifting in zeros. By convention $C \gg 63$ is the value (1 or 0) of the last bit of the word. We use the bitwise AND NOT operator.

```
input: bitmap $B$ as an array 1024 64-bit integers, $C_1$ to $C_{1024}$.
output: the number of runs $r$
$r \leftarrow 0$
for $i \in \{1, 2, \ldots, 1023\}$ do
    $r \leftarrow r + \text{bitCount}((C_i \ll 1) \text{ ANDNOT } C_i)$
            $+(C_i \gg 63) \text{ ANDNOT } C_{i+1}$
$r \leftarrow r + \text{bitCount}((C_{1024} \ll 1) \text{ ANDNOT } C_{1024})$
        $+C_{1024} \gg 63$
return $r$
```

nearly twice as fast as computing the exact count in our tests using a recent Intel processor (Haswell microarchitecture). For simplicity, we do not make use of this possible optimization.

Efficient conversions between containers are generally straight-forward, except for conversions from a bitmap container to another container type. Converting from a bitmap to an array container is reviewed in Chambi et al. [7]. As for the conversion to run containers, we use Algorithm 2 to extract runs from a bitmap. It is efficient as long as locating the least significant 1-bit in a word is fast. Thankfully, recent processors include fast instructions to find the index of the least significant 1-bit in a word or, equivalently, of the number of trailing zeros (`bsf` or `tzcnt` on x64 processors, `rbit` followed by `clz` on ARM processors). They are accessible in Java through the `Long.trailingZeroes` intrinsic. To locate the index of the least significant 0-bit, we simply negate the word, and then seek the least significant 1-bit. Otherwise the algorithm relies on inexpensive bit-manipulation techniques.

Algorithm 2 Algorithm to convert the set bits in a bitmap into a list of runs. We assume two-complement's 64-bit arithmetic. We use the bitwise AND and OR operations.

```
input: a bitmap $B$, as an array of 64-bit words $C_1$ to $C_{1024}$
output: an array $S$ containing runs of 1-bits found in the bitmap $B$
Let $S$ be an initially empty list
Let $i \leftarrow 1$
$T \leftarrow C_1$
while $i \leq 1024$ do
    if $T = 0$ then
        $i \leftarrow i + 1$
        $T \leftarrow C_i$
    else
        $j \leftarrow$ index of least significant 1-bit in $T$ ($j \in [1, 64]$)
        $x \leftarrow j + 64 \times (i - 1)$
        $T \leftarrow T \text{ OR } (T - 1)$                {all bits with indexes $< i$ are set to 1}
        while $i + 1 \leq 1024$ and $T = $ 0xFFFFFFFFFFFFFFFF do
            $i \leftarrow i + 1$
            $T \leftarrow C_i$
        if $T = $ 0xFFFFFFFFFFFFFFFF then
            $y \leftarrow 64 \times i$
        else
            $k \leftarrow$ index of least significant 0-bit in $T$ ($j \in [1, 64]$)
            $y \leftarrow k + 64 \times (i - 1)$
        append to $S$ a run that goes from $x$ to $y$ inclusively
        $T \leftarrow T \text{ AND } T + 1$                {all bits with indexes $< k$ are set to 0}
return $S$
```

5. LOGICAL OPERATIONS

5.1. Union and intersection

There are many necessary logical operations, but we present primarily the union and intersections. They are the most often used, and the most likely operations to cause performance bottlenecks.

An important algorithm for our purposes is galloping intersection (also called exponential intersection) to compute the intersection between two sorted arrays of sizes c_1, c_2. It has complexity $O(\min(c_1, c_2) \log \max(c_1, c_2))$ [17]. In this approach, we pick the next available integer from the smaller array and seek an integer at least as large in the large array, looking first at the next available value, then looking twice as far, and so on, until we find an integer that is not smaller than the integer from the small array. We then use a binary search in the large array to find the exact location of the first integer not smaller than the integer from the small array. We call this process a galloping search. We repeat it with each value from the small list.

A galloping search makes repeated random accesses in a container, and it could therefore cause expensive cache faults. However, in our case, the potential problem is mitigated by the fact that all our containers fit in CPU cache.

Intersections between two input Roaring bitmaps start by visiting the keys from both bitmaps, starting from the beginning. If a key is found in both input bitmaps, the corresponding containers are intersected and the result (if non-empty) is added to the output. Otherwise, we advance in the bitmap corresponding to the smallest key, up to the next key that is no smaller than the key of the other bitmap, using galloping search. When one bitmap runs out of keys, the intersection terminates.

Unions between Roaring data structures are handled in the conventional manner: we iterate through the keys in sorted order; if a key is present in both input Roaring bitmaps, we merge the two containers, add the result to the output and advance in the two bitmaps. Otherwise, we clone the container corresponding to the smaller key, add it to the output and advance in this bitmap. When one bitmap runs out of keys, we simply append all the remaining content of the other bitmap to the output.

Though we do not use this technique, instead of cloning the containers during unions, we could use a copy-on-write approach whereas a reference to container is stored and used, and a copy is only made if an attempt is made to modify the container further. This approach can be implemented by adding a bit vector containing one bit per container. Initially, this bit is set to 0, but when the container cannot safely be modified without making a copy, it is set to 1. Each time the container needs to be modified, this bit needs to be checked. Whether the copy-on-write approach is worth the added complexity is a subject for future study. However, container cloning was never found to a significant computational bottleneck in the course of our development. It is also not clear whether there are applications where it would lead to substantial reduction of the memory usage. In any case, merely copying a container in memory can be several times faster than computing the union between two containers so that copying containers is unlikely to be a major bottleneck.

We first briefly review the logical operations between bitmap and array containers, referring the reader to Chambi et al. [7] for algorithmic details.

Bitmap vs Bitmap: To compute the intersection between two bitmaps, we first compute the cardinality of the result using the bitCount function over the bitwise AND of the corresponding pairs of words. If the intersection exceeds 4096, we materialize a bitmap container by recomputing the bitwise AND between the words and storing them in a new bitmap container. Otherwise, we generate a new array container by, once again, recomputing the bitwise ANDs, and iterating over their 1-bits. We find it important to first determine the right container type as, otherwise, we would sometimes generate the wrong container and then have to convert it—an expensive process. The performance of the intersection operation between two bitmaps depends crucially on the performance of the bitCount function.

A union between two bitmap containers is straightforward: we execute the bitwise OR between all pairs of corresponding words. There are 1024 words in each container, so

1024 bitwise OR operations are needed. At the same time, we compute the cardinality of the result using the bitCount function on the generated words.

Bitmap vs Array: The intersection between an array and a bitmap container can be computed quickly: we iterate over the values in the array container, checking the presence of each 16-bit integer in the bitmap container and generating a new array container that has as much capacity as the input array container. The running time of this operation depends on the cardinality of the 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: The intersection between two array containers is always a new array container. We allocate a new array container that has its capacity set to the minimum of the cardinalities of the input arrays. When the two input array containers have similar cardinalities c_1 and c_2 ($c_1/64 < c_2 < 64c_1$), we use a straightforward merge algorithm with algorithmic complexity $O(c_1 + c_2)$, otherwise we use a galloping intersection with complexity $O(\min(c_1, c_2) \log \max(c_1, c_2))$ [17]. We arrived at this threshold ($c_1/64 < c_2 < 64c_1$) empirically as a reasonable choice, but it has not been finely tuned.

For unions, if the sum of the cardinalities of the array containers is 4096 or less, we merge the two sorted arrays. We allocate a new array container that has its capacity set to the sum of the cardinalities of the input arrays. Otherwise, we generate an initially empty bitmap container. Though we cannot know whether the result will be a bitmap container (i.e., whether the cardinality is larger than 4096), as a heuristic, we suppose that it will be so. Iterating through the values of both arrays, we set the corresponding bits in the bitmap to 1. Using the bitCount function, we compute cardinality, and then convert the bitmap into an array container if the cardinality is at most 4096.

Alternatively, we could be much more conservative and predict the cardinality of the union on the assumption that the two containers have independently distributed values over the whole chunk range (2^{16} values). Beside making the code slightly more complicated, it is not likely to change the performance characteristics as the naïve model is quite close to an independence-based model. Indeed, under such a model the expected cardinality of the intersection would be $\frac{c_1}{2^{16}} \times \frac{c_1}{2^{16}} \times 2^{16}$. The expected cardinality of the union would be $c_1 + c_2 - \frac{c_1 c_2}{2^{16}}$. The maximal threshold for an array container is 4096, so we can set $c_1 + c_2 - \frac{c_1 c_2}{2^{16}} = 4096$ and solve for c_1 as a function of c_2: $c_1 = \frac{2^{16}(4096 - c_2)}{2^{16} - c_2}$. In contrast, our simplistic model predicts a cardinality of $c_1 + c_2$ and thus a threshold at $c_1 = 4096 - c_2$. However, since $c_2 \leq 4096$, we have that $\frac{2^{16}}{2^{16} - c_2} = 1 + \frac{c_2}{2^{16} - c_2} \leq 1 + \frac{4096}{2^{16}} = 1.0625$. That is, for any fixed value of one container (c_2 here), the threshold on the cardinality of the other container beyond which we predict a bitmap container is at most $6.25\,\%$ larger under an independence-based estimate, compared with our naïve approach.

Given array and bitmap containers, we need to have them interact with run containers. For this purpose, we introduced several new algorithms and heuristics.

Run vs Run: When computing the intersection between two run containers, we first produce a new run container by a simple intersection algorithm. This new run container has its capacity set to the sum of the number of runs in both input containers. The algorithm starts by considering the first run, in each container. If they do not overlap, we advance in the container where the run occurs earlier until they do overlap, or we run out of runs in one of the containers. When we run out of runs in either container, the algorithm terminates. When two runs overlap, we always output their intersection. If the two runs end at the same value, then we advance in the two run containers. Otherwise, we advance only in the run container that ends first. Once we have computed the answer, after exhausting the runs in at least one container, we check whether the run container should be converted to either a bitmap (if it has too many runs) or to an array container (if its cardinality is too small compared to the number of runs).

The union algorithm is also conceptually simple. We create a new, initially empty, run container that has its capacity set to the sum of the number of runs in both input containers. We iterate over the runs, starting from the first run in each container. Each time, we pick a run that has a minimal starting point. We append it to the output either as a new run, or as an extension of the previous run. We then advance in the container where we picked the run. Once a container has no more runs, all runs remaining in the other container are appended to the answer. After we have computed the resulting run container, we convert the run container into a bitmap container if too many runs were created. Checking whether such a conversion is needed is fast, since it can be decided only by checking the number of runs.

Run vs Array: The intersection between a run container and an array container always outputs an array container. This choice is easily justified: the result of the intersection has cardinality no larger than the array container, and it cannot contain more runs than the array container. We can allocate a new array container that has its capacity set to the cardinality of the input array container. Our algorithm is straightforward. We iterate over the values of the array, simultaneously advancing in the run container. Initially, we point at the first value in the array container and the first run in the run container. While the run ends before the array value, we advance in the run container. If the run overlaps the array value, the array value is included in the intersection, otherwise it is omitted.

Determining the best container for storing the union between a run container and an array is less straightforward. We could process the run container as if it were an array container, iterating through its integers and re-use our heuristic for the union between two array containers. Unfortunately, this would always result in either an array or bitmap container. We found that it is often more beneficial to predict that the outcome of the union is a run container, and to convert the result to a bitmap container, if we must. Thus, we follow the heuristic for the union between two run containers, effectively treating the array container as a run container where all runs have length one. However, once we have computed the union, we must not only check whether to convert the result to a bitmap container, but also, possibly, to an array container. This check is slightly more expensive as we must compute the cardinality of the result.

Run vs Bitmap: The intersection between a run container and a bitmap container begins by checking the cardinality of the run container. If it is no larger than 4096, then we create an initially empty array container. We then iterate over all integers contained in the run container, and check, one by one, whether they are contained in the bitmap container: when an integer is found to be in the intersection, it is appended to the output in the array container. The running time of this operation is determined by the cardinality of the run container. Otherwise, if the input run container is larger than 4096, then we create a copy of the input bitmap container. Using fast bitwise operations, we set to zero all bits corresponding to the complement of the run container (see Algorithm 3). We then check the cardinality of the result, converting to an array container if needed.

The union between a run container and a bitmap container is computed by first cloning the bitmap container. We then set to one all bits corresponding to the integers in the run container, using fast bitwise OR operations (see again Algorithm 3).

In some instances, the result of an intersection or union between two containers could be most economically represented as a run container, even if we generate an array or bitmap container. It is the case when considering the intersection or union between a run container and a bitmap container. We could possibly save memory and accelerate later computations by checking whether the result should be converted to a run container. However, this would involve keeping track of the number of runs—a relatively expensive process.

Furthermore, we added the following optimization. Whenever we compute the union between a run container and any other container, we first check whether the run container contains a single run filling up the whole space of 16-bit values ($[0, 2^{16})$). In that case, we know that the union must be

Algorithm 3 Algorithm to set a range of bits to 0 or 1 in a bitmap.

```
input: a bitmap $B$, as an array of 64-bit words $C_1$ to $C_{1024}$
output: the same bitmap with all bits with indexes in $[i, j)$ set to
    • 1 if OP is the bitwise OR operation,
    • 0 if OP is the bitwise AND NOT operation.
$x \leftarrow \lfloor i/64 \rfloor$
$y \leftarrow \lfloor (j-1)/64 \rfloor$
$Z \leftarrow \mathrm{0xFFFFFFFFFFFFFFFF}$    $\{Z$ has all its bits set to $1\}$
$X \leftarrow Z \ll (i \bmod 64)$    $\{X = \overbrace{11\cdots1}^{64-(i \bmod 64)} \overbrace{00\cdots0}^{i \bmod 64}\}$
$Y \leftarrow Z \gg (64 - (j \bmod 64))$    $\{X = \overbrace{00\cdots0}^{64-(j \bmod 64)} \overbrace{11\cdots1}^{j \bmod 64}\}$
if $x = y$ then
    $C_x \leftarrow C_x$ OP $(X \text{ AND } Y)$
else
    $C_x \leftarrow C_x$ OP $X$
    for $i = x+1, x+2, \ldots, y-1$ do
        $C_x \leftarrow C_x$ OP $Z$
    $C_y \leftarrow C_y$ OP $Y$
return $B$
```

simply the other container and we can produce optimized code accordingly. The check itself can be computed in a few inexpensive operations.

The computation of the intersection is particularly efficient in Roaring when it involves a bitmap container and either an array container or a run container of small cardinality. It is also efficient when intersecting two array containers, with one having small cardinality compared to the other, as we use galloping intersections. Moreover, by design, Roaring can skip over entire chunks of integers, by skipping over keys that are present in only one of the two input bitmaps. Therefore, Roaring is well suited to the problem of intersecting a bitmap having a small cardinality with bitmaps having larger cardinalities. In contrast, RLE-based compression offers fewer opportunities to skip input data.

The computation of the union between two Roaring bitmaps is particularly efficient when it involves run containers or array containers being intersected with bitmap containers. Indeed, these computations involve almost no branching and minimal data dependency, and they are therefore likely to be executed efficiently on superscalar processors. RLE-based compression often causes a lot of data dependencies and branching.

Another feature of Roaring is that some of these logical operations can be executed *in place*. In-place computations avoid unnecessary memory allocations and improve data locality.

- The union of a bitmap container with any other container can be written out in the input bitmap container. The intersection between two bitmap containers, or between a bitmap container and some run containers, can also be written out to an input bitmap container.

- Though array containers do not support in-place operations, we find it efficient to support in-place unions in a run container with respect to either another run container or an array container. In these cases, it is common that the result of the union is either smaller or not much larger than combined sizes of the inputs. The runs in a run container are stored in a dynamic array that grows as needed, but it has typically some excess capacity. So we first check whether it would be possible to store both the input run container and the output (whose size is bounded by the sum of the inputs). Otherwise, we allocate the necessary capacity. We then copy the data corresponding to the input run container from the beginning of the array to the end. That is, if the input run container had r runs, they would be stored using the first

$4r$ bytes of an array, and we copy them to the last $4r$ bytes of the array—freeing the beginning of the array. We write the result of the union at the beginning of the array, as usual. Thus, given enough capacity, this approach enables repeated unions to the same run container without new memory allocation. We trade the new allocation for a copy within the same array. Since our containers can fit in CPU cache, such a copy can be expected to be fast. We could, similarly, enable in-place intersections within run or array containers.

Because we have optimized the previous We put in practice a heuristic for in-place unions between two run containers or between a run container and an array container. In these cases, we found it common that the result of the union is either smaller or not much larger than combined sizes of the inputs. Thus it would seem that we could often store the result of union in the input run container, thereby avoiding a new memory allocation. The runs in a run container are stored in a dynamic array that grows as needed, but it has typically some excess capacity. So we first check whether it would be possible to store both the input run container and the output (whose size is bounded by the sum of the inputs). SHIT

A common operation in applications is the aggregation of a long list of bitmaps. When the problem is to compute the intersection of many bitmaps, we can expect a naive algorithm to work well with Roaring: we can simply compute the intersection of the first two bitmaps, then intersect the result with the third bitmap, and so forth. With each new intersection, the result might become smaller, and Roaring can often compute efficiently the intersection between bitmap having small cardinality and bitmaps having larger cardinalities, as already stated. Computing the union of many bitmaps requires more care. As already remarked in Chambi et al. [7], it is wasteful to update the cardinality each and every time when computing the union between several bitmap containers. Though the bitCount function is fast, it can still use a significant fraction of the running time: Chambi et al. [7] report that it reduces the speed by about $30\,\%$. Instead, we proceed with what we call a "lazy union". We proceed as usually with the union, except that some unions between containers are handled differently:

- The union between a bitmap container and any other container type is done as usual, that is, we compute the resulting bitmap container, except that we do not attempt to compute the cardinality of the result. Internally, the cardinality is set to the flag value "-1", indicating that the cardinality is currently unknown.

- When computing the union of a run container with an array container, we always output a run container or, if the number of runs is too great, a bitmap container. In some instances, this could be slightly wasteful storage-wise, as an array container might be smaller.

After the final answer is generated, we "repair it" by computing the cardinality of any bitmap container, and by checking whether any run container should be converted to an array container. For even greater speed, we could even skip this repair phase, making it optional.

We consider two strategies to compute the union of many bitmaps. One approach is a naïve two-by-two union: we first compute the union of the first two bitmaps, then the union of the result and the third bitmap and so forth, doing the computation in-place if possible. The benefit of this approach is that we always keep just one intermediate result in memory. In some instances, however, we can get better results with other algorithms. For example, we can use a heap: put all original bitmaps in a minimum heap, and repeatedly poll the two smallest bitmaps, compute their union, and put them back in the heap, as long as the heap contains more than one bitmap. This approach may create many more intermediate bitmaps, but it can also be faster in some instances. To see why that must be the case, consider that the complexity of the union between two bitmaps of size B is $O(B)$, generating a result that might be of size $2B$. Thus, given N bitmaps of size B, the naïve approach has complexity $O(BN^2)$, whereas the heap-based approach has complexity $O(BN \log N)$. However, this computation model, favourable to the heap-based approach, does not always apply. Indeed, suppose that the bitmaps are uncompressed bitmaps over the same range of values—as is sometimes the case when Roaring bitmaps are made of bitmap containers. In that case, the union between two bitmaps of size B has complexity $O(B)$, and the output has size B. We then have that both algorithms, the naïve and heap-based ones, have the same optimal $O(BN)$ complexity. However, the naïve algorithm has storage requirements in $O(B)$ whereas the heap-based algorithm's storage requirements are in $O(BN)$, indicating that the latter has a worse performance. We expect that whether one algorithm or the other has better running time is data and format dependent, but in actual application, it might be advantageous to use the naïve algorithm if one wishes to have reduced memory usage.

5.2. *Other Operations*

A logically complete set of operations enables Roaring to be used, via the bit-slicing approach [18], to realize arbitrary Boolean operations. Thus, our Roaring software supports negation, although it uses the more general `flip` approach of Java's `BitSet` class, wherein negation occurs only within a range. (Besides adding additional flexibility, this approach means that there is no need to know the actual universe size, in the case when the bit set is intended to be over a smaller universe than 0 to $2^{32} - 1$.)

Observe that when a run container is smaller than a bitmap container, its negation would typically remain smaller than the original bitmap: a negation over the full range can increase the number of runs by at most one (in the case that the original bitmap begins and ends with zeros). When bits are flipped within a smaller range, the number of runs before and after the range are unchanged, whereas the number of runs inside the range can increase by at most one. A case analysis on the value of the bit immediately before the range and the new value of the first bit in the range can then establish the result. Indeed, inspecting four bits (those on either side of the range's boundaries) can determine whether an in-place negation is possible, without requiring space for an additional run. This situation differs significantly from other operations (AND, OR), where an input run container can give rise to a result with many more runs. As a result, we could leave the result of a run container's negation as a run container, although there are cases when the result could be half the size as an array. Our current implementation does check and convert to the smallest representation.

Although adding negation gives us logical completeness, efficiency is gained by supporting other Boolean operations directly. For instance, our Roaring software provides an XOR operation that is frequently useful for bit-sliced arithmetic[12] and that provides symmetric difference between sets. Roaring also provides an AND NOT operation that implements set difference, which is important in some applications. The implementation of the symmetric difference is similar to that of the union, with the added difficulty that the cardinality might be lower than that of any of the two inputs. The implementation of the difference is similar to that of the intersection.

Our software also supports fast rank and select functions: rank queries count the number of values present in a range whereas select queries seek the i^{th} value. These queries are accelerated because array and bitmap containers maintain their cardinality as a value that can be quickly queried. The software also supports the ability to add or remove all values in an interval, to quickly iterate over the values, and so forth.

6. EXPERIMENTS

To validate our results, we present a range of experiments on realistic datasets. We are mostly concerned with the computation of intersections and unions. We focus on bitmap formats: we refer the interested reader to other work [2] for comparisons between bitmaps and other implementations of sets (such as hash sets or trees).

6.1. *Hardware*

For benchmarking our algorithms, we used a Linux server with an Intel i7-4770 processor (3.4 GHz, 32 kB of L1 data cache, 256 kB of L2 cache per core and 8 MB of L3 cache). The server has 32 GB of RAM (DDR3-1600, double-channel). Because there is ample memory, we expect that all disk accesses are buffered. We disabled Turbo Boost and the processor is configured to always run at its highest clock speed. For the purpose of our experiments, all our algorithms and software are single-threaded, so a single core is used.

6.2. Software

We implemented our software in Java and published it as the version 0.5 of the Roaring bitmap open-source library. This library is used by major database systems like Apache Kylin and Druid [9]. We used Oracle's JDK 1.8u60 for Linux (x64) during benchmarking. Benchmarking Java code can be difficult since one needs to take into account just-in-time compilation, garbage collection and so on. To help cope with these challenges, we wrote all our benchmarks using Oracle's JMH benchmarking system.[‡] After an initial warm-up phase, all tests run five times and we record the average time (using wall-clock timings). In all our tests, the range of timings varies by no more than 2%, so that the average is representative. To ensure reproducibility, we make available all our testing software, including data and necessary scripts.[§] The interested reader should be able to reproduce all our results merely by launching a script after downloading our software package.

One of the benefits of using Java is that we have access to several high-quality implementations of competing bitmap formats. For WAH and Concise, we use Metamarkets' CONCISE library (version 1.3.4).[¶] This library has been used for many years by Metamarkets in their Druid database system [9]. It is derived from the original software produce by Colantonio and Di Pietro [2] in their work on Concise: we can expect it to perform adequately as an implementation of the Concise format. We also make use of the JavaEWAH library (version 1.0.6).[‖] JavaEWAH has been used in production for many years in systems like Apache Hive. The library has also been used by in several previous research work [1, 5, 19, 20]. The JavaEWAH library supports both a 32-bit and a 64-bit version of the EWAH format: we test both.

All these Java libraries enable memory-file mapping, by providing versions where the data storage mechanism is abstracted by a Java `ByteBuffer`. In Java, from a memory-mapped file, one can extract a `ByteBuffer` object, and this can be used to randomly access byte or integer values. It is also possible to wrap a regular Java Integer, effectively creating such a `ByteBuffer` object. Accessing data through a `ByteBuffer` can be slower, because the virtual machine is less able to optimize data access than when the data is in Java's native arrays. However, a `ByteBuffer` object representing a memory-mapped file uses little of Java's heap. Hence, this approach reduces the need for garbage collection and may make it less likely that the virtual machine will run out of memory. Moreover, compared to deserializing a bitmap to Java's memory from disk, memory-file mapping can be much faster. This approach works best when these bitmaps are immutable: we create them once, store them to disk and retrieve them as needed.

6.3. Data

We used four real datasets from earlier studies of compressed bitmap indexes [7, 21]. In one instance, the datasets were taken as-is: we did not sort them prior to indexing. In another instance, we sorted them lexicographically prior to indexing, with the smallest cardinality column being the primary sort key, the next-smallest cardinality column being the secondary sort key, and so forth [5]. The net result is that we have two sets of bitmaps from each data source: one from the data in its original order and one from sorted data. For each dataset, we built a bitmap index and chose 200 bitmaps, using stratified sampling to control for attribute cardinalities. We present the basic characteristics of these bitmaps in Table I. All of our datasets are publicly available.

[‡] http://openjdk.java.net/projects/code-tools/jmh/

[§] https://github.com/lemire/RoaringBitmap/tree/master/jmh

[¶] https://github.com/metamx/extendedset

[‖] https://github.com/lemire/javaewah

Table I. Characteristics of our four realistic datasets

(a) Description of the bitmaps

	# bitmaps	universe size	avg. count/bitmap
CensusInc	200	199 522	34 610.1
Weather	200	1 015 366	64 353.1
Census1881	200	4 277 805	5 019.3
Wikileaks	200	1 353 178	1 376.8

(b) Compressed sizes (bits/int), best results in bold

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run	WAH
CensusInc	2.9	3.9	3.3	2.7	**2.6**	2.9
CensusInc$^{\text{sort}}$	**0.55**	0.90	0.64	3.0	0.60	**0.55**
Census1881	25.6	43.8	33.8	16.0	**15.1**	43.8
Census1881$^{\text{sort}}$	2.5	4.6	2.9	6.1	**2.2**	2.5
Weather	5.9	7.9	6.7	**5.4**	**5.4**	5.9
Weather$^{\text{sort}}$	0.43	0.86	0.54	3.2	**0.34**	0.43
Wikileaks	10.2	19.5	10.9	16.5	**5.9**	10.2
Wikileaks$^{\text{sort}}$	2.2	4.7	2.7	10.7	**1.6**	2.2

6.4. Compressed Size

The compressed sizes are given in Table Ib in average bits per integer stored. That is, a bitmap using 100 bytes to store 100 values, would use 8 bits per integer.

Among the RLE-based formats (all of them but Roaring), the best compression is offered by Concise and WAH, with the worse compression by the 64-bit version of EWAH. Concise and WAH offer the same compression except for one dataset (Census1881) where Concise reduces the space usage by a factor of 1.7. The 64-bit version of EWAH can use twice the storage than the other schemes (all of which are 32-bit formats). However, if we omit the 64-bit version of EWAH, the space usage of all three formats (Concise, EWAH and WAH) is similar (within 30%).

The RLE-based formats compress sorted datasets much better than unsorted data, sometimes an order of magnitude better. For example, WAH and Concise use 5.9 bits per set value for Weather but only 0.43 bits per set value for Weather$^{\text{sort}}$. This result is consistent with earlier work [22] showing the importance of sorting the data prior to indexing it with RLE-compressed bitmap formats.

Roaring often offers better compression than the RLE-based schemes. For example, on Census1881, Roaring uses only 60% of the space used by Concise. However, on the sorted datasets, the results are much less positive for Roaring. In one case (Weather$^{\text{sort}}$), Concise and WAH use 7.4 times the space used by Roaring (0.43 bits vs. 3.2 bits).

However, once we enable run compression in Roaring, the results are again more positive for Roaring. In fact, there is just one case where Concise uses less storage (CensusInc$^{\text{sort}}$), and the difference is small (8%). In what was previously the worst case (Weather$^{\text{sort}}$), Roaring is down to 0.34 bits (from 3.2 bits) vs. 0.43 bits for Concise and WAH.

We might summarize the results as follows. When the data has been sorted prior to indexing, Roaring with run compression is as good, and even slightly better, than pure RLE-based schemes. When the data is not sorted prior to index, Roaring can be far superior as far as compression is concerned.

6.5. Serialization and Run Optimization

An expected typical use of run optimization in Roaring consists in calling the run optimization function prior to serializing the bitmaps. We want to assess the effect of run optimization on serialization. The results are presented in Table II. For this purpose, we serialize the bitmaps to a

Table II. Timings in milliseconds for Roaring serialization (200 bitmaps, best results in bold)

	serialization	runOptimize + serialization
CENSUSINC	**8.0**	8.2
CENSUSINC$^{\text{sort}}$	4.0	**1.7**
CENSUS1881	12	12
CENSUS1881$^{\text{sort}}$	2.6	**1.7**
WEATHER	**36**	38
WEATHER$^{\text{sort}}$	18	**8.2**
WIKILEAKS	2.1	**1.2**
WIKILEAKS$^{\text{sort}}$	1.7	**0.7**

byte array (in memory). The data indicates that though it can sometimes be slightly more expensive to run optimize and serialize (by 5%), it can sometimes be much faster (by over 2×) because we have to serialize less data. This suggests that the run optimization is efficient, at least compared to the cost of serializing the bitmaps. If we were to serialize the bitmaps to a slow disk or to a slow network, the benefit of producing less data would only be more significant.

6.6. *Performance of Queries in Java's Heap*

We begin by reviewing the performance of the various bitmap formats against a range of queries over in-memory bitmaps. That is, these bitmaps are stored in Java's heap, as most other Java objects are. The results are presented in Table III. In all tests, the source bitmaps are treated as being immutable: where applicable, new bitmaps containing the answer to the query are generated.

Because the absolute performance numbers are difficult to appreciate on their own, we present relative numbers, normalized against Roaring with run optimization (Roaring+Run receives value 1.0). Thus, a value of 2.0 would indicate that some operation takes twice as long as it would take with Roaring+Run.

In Table IIIa, we access the first, second and third quartile position in the universe, and check the presence of the value, for all 200 bitmaps in the set. We observe a slight decrease of the performance of Roaring when run optimization is applied (up to 25%) in some cases, with an improvement in others (up to 30%). These differences do not appear very significant compared to the difference between the Roaring formats and the RLE-based formats. Even if we compare against the fastest among them (64-bit EWAH), Roaring can be two orders of magnitude faster. Against Concise and WAH, Roaring is sometimes nearly three orders of magnitude faster. These drastic results are easily understood: RLE-compressed bitmaps do not sensibly support random access and require a full scan from the beginning.

We then consider the 199 intersections between successive bitmaps (Table IIIb), and the 199 unions between successive bitmaps (Table IIIc). In these tests, we compute the intersections or unions, and then check the cardinality of the result. Roaring with run optimization is never slower than Roaring (within 5%), but can be up to twice as fast on sorted datasets. Compared to Concise, Roaring+Run is consistently at least 3.5 times faster, and up to hundreds of times faster at computing intersections. The gap between Concise and Roaring+Run is less impressive for unions (between 1.7 times faster and 40 times faster), but still leaves Roaring with a significant advantage. In these tests, as in many others, the fastest RLE-based format is the 64-bit version of EWAH.

Another important test case is the union of all 200 bitmaps. The results are presented in Tables IIId and IIIe for the naïve and priority-queue approaches. In particular, one column in Table IIIe compares directly the naïve approach with the priority queue for Roaring+Run. We see that there is no clear winner between naïve approach and the priority queue: for WIKILEAKS, the priority queue is preferable, but for the non-sorted versions of CENSUSINC and CENSUS1881, the naïve approach is better for Roaring. In any case, no matter which approach is used, Roaring+Run is clearly preferable to the RLE-based formats like Concise, being anywhere from 1.8× to nearly 10× faster.

Though it is not our focus, we see that among the RLE-based schemes, the EWAH formats almost always offer better performance. This observation is consistent with earlier work [10] showing that EWAH offers better query performance. The benefits of EWAH are especially visible with the random access and successive intersections tests—reflecting the fact that EWAH supports a limited form of skipping. Generally the 64-bit version of EWAH is faster. However, for entire unions computed with the naïve algorithm, the 32-bit version is faster—reflecting the smaller size of the 32-bit EWAH bitmaps. Moreover, WAH is faster than Concise even though Concise rarely provides significant compression benefits. The slower speed of Concise compared to WAH is consistent with earlier work [2].

Even without run optimization, Roaring is fast. Consider WEATHER$^{\text{sort}}$ where WAH and Concise compress over 7 times better than the original Roaring (without run containers). In all cases, except for entire unions, the original Roaring is faster than Concise and WAH. And only for one case (CENSUSINC$^{\text{sort}}$ and successive unions) does the 64-bit EWAH matches the speed of the original Roaring. For entire unions using the naïve algorithm, there is only one case (CENSUSINC$^{\text{sort}}$) where WAH and Concise surpass the original Roaring. Further, the EWAH schemes surpass the original Roaring even in the unsorted version of this dataset (CENSUSINC). Only in the case where we execute entire unions using a priority queue on sorted datasets is the original Roaring outdone in some datasets (see Fig. IIIe). We observe that the priority queue algorithm is ineffective with the original Roaring, reflecting the fact that as we aggregate many bitmaps, it is unable to benefit from the long runs being created in the intermediate bitmaps. Except for one dataset (CENSUS1881$^{\text{sort}}$), the naïve union algorithm algorithm is more effective with the original Roaring since, as the result becomes denser, it creates bitmap containers that can compute following unions in-place. Comparing the original Roaring format (without runs) with the RLE-based formats on sorted datasets, we observe that it is generally at least slightly worse, being up to six times slower (WEATHER$^{\text{sort}}$) than Concise.

6.7. Memory-Mapped Performance

Table IV presents the timings of several tests on memory-mapped bitmaps, using different formats. We omit the WAH format because none of our libraries support it in memory-mapped mode: this is of little consequence given the similarity between the Concise and WAH formats and results. In the case of these tests, the bitmaps are first written to disk and then mapped in memory using Java's `ByteBuffer`. These bitmaps are immutable. In practice, because we have sufficient memory compared to the size of the datasets, we expect all queries to be buffered. When queries necessitate the generation of a new bitmap, it is created in Java's heap.

The results of the tests are similar to the case where we use regular bitmaps stored entirely in Java's heap, except that the relative differences between various formats tend to be less pronounced. That is to be expected since all queries suffer from a higher overhead due to the overhead of accessing the data through a mapped `ByteBuffer`.

For the random access test (Table IVa), we omit the results for Concise since the library we used does not have built-in support for random access. We could easily have implemented our own version using the intersection with a bitmap containing a single set bit, but such an approach might not offer the best efficiency. And, in any case, we cannot expect good results from Concise on this test from previous experiments.

For the results pertaining to the union of all 200 bitmaps (Tables IVd and IVe), we included a special approach marked by a star ($\star$) developed originally for the Druid engine. Unlike the naïve or priority-queue approaches, that always combine bitmaps two-by-two, this approach takes all bitmaps at once, and using a priority queue, merges the compressed words into a single output. Chambi et al. [7] described a similar approach for Roaring, where the priority queue worked over containers—this approach has been replaced by the simpler naïve algorithm as the default for the Roaring library. For sorted data sources, the *star* approach is always preferable in this instance. But even in these cases, Roaring+Run is always significantly faster. In two cases, Roaring+Run is over

Table III. Relative timings (Roaring+Run=1): in Java heap

(a) Random value access

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run	WAH
CensusInc	160	19	35	0.75	1.0	170
CensusInc$^{\text{sort}}$	33	8.0	12	0.84	1.0	35
Census1881	870	190	360	1.0	1.0	920
Census1881$^{\text{sort}}$	52	19	24	1.1	1.0	53
Weather	600	76	150	0.78	1.0	610
Weather$^{\text{sort}}$	90	35	42	0.91	1.0	98
Wikileaks	48	26	29	1.1	1.0	49
Wikileaks$^{\text{sort}}$	14	9.4	9.8	1.3	1.0	15

(b) successive intersections

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run	WAH
CensusInc	5.5	2.6	3.9	0.99	1.0	6.4
CensusInc$^{\text{sort}}$	3.5	1.4	1.7	1.1	1.0	3.0
Census1881	460	94	150	0.97	1.0	370
Census1881$^{\text{sort}}$	61	19	23	1.2	1.0	55
Weather	5.5	2.1	3.3	0.98	1.0	4.7
Weather$^{\text{sort}}$	5.7	2.4	2.6	1.4	1.0	5.1
Wikileaks	7.2	3.6	3.6	0.96	1.0	6.9
Wikileaks$^{\text{sort}}$	9.1	5.9	5.9	1.5	1.0	8.5

(c) successive unions

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run	WAH
CensusInc	4.6	2.4	4.0	1.0	1.0	4.2
CensusInc$^{\text{sort}}$	1.7	0.99	1.4	1.0	1.0	1.5
Census1881	43	22	43	0.92	1.0	38
Census1881$^{\text{sort}}$	8.8	6.8	8.4	1.2	1.0	7.8
Weather	4.4	2.0	3.8	0.93	1.0	3.9
Weather$^{\text{sort}}$	3.4	2.5	3.1	2.1	1.0	3.0
Wikileaks	3.8	4.2	4.4	1.3	1.0	3.6
Wikileaks$^{\text{sort}}$	2.3	2.6	2.9	1.6	1.0	2.3

(d) Entire unions (naïve)

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run	WAH
CensusInc	9.7	2.6	1.7	2.9	1.0	9.2
CensusInc$^{\text{sort}}$	8.1	5.5	4.5	25	1.0	6.9
Census1881	22	14	4.9	0.51	1.0	20
Census1881$^{\text{sort}}$	15	16	11	0.71	1.0	13
Weather	13	5.5	3.6	0.99	1.0	11
Weather$^{\text{sort}}$	3.4	3.1	2.4	2.0	1.0	3.0
Wikileaks	5.6	5.3	3.0	0.28	1.0	5.3
Wikileaks$^{\text{sort}}$	11	11	8.7	1.2	1.0	9.5

(e) Entire unions (priority queue)

	Concise	EWAH	EWAH	Roaring	Roaring	Roaring	Roaring	WAH
		64-bit		naive	pq	+Run nai.	+Run pq	
CensusInc	3.9	1.4	1.6	0.5	1.1	0.16	1.0	3.5
CensusInc$^{\text{sort}}$	7.3	4.3	5.0	24	51	1.0	1.0	6.7
Census1881	2.2	1.2	2.2	0.35	0.79	0.69	1.0	1.9
Census1881$^{\text{sort}}$	2.5	2.0	2.3	3.6	2.5	5.4	1.0	2.4
Weather	3.4	1.0	1.9	0.17	0.93	0.17	1.0	3.1
Weather$^{\text{sort}}$	1.8	4.2	4.5	12	45	5.8	1.0	1.8
Wikileaks	2.6	2.2	2.6	0.98	1.6	3.4	1.0	2.4
Wikileaks$^{\text{sort}}$	2.7	2.5	2.6	3.1	3.4	2.5	1.0	2.4

a hundred times faster that Concise. Even if we focus just on sorted data inputs, the priority-queue approach with Roaring+Run can be up to ten times faster than Concise.

We can once again compare the naïve and priority-queue approaches for Roaring+Run, this time using the fact that both tables (Tables IVd and IVe) present the same algorithm (Concise$\star$). Unsurprisingly, there is no clear winner. On sorted datasets, the priority queue is better, but, often, the opposite is true for the other datasets. We do expect, however, the naïve to often have lower memory usage, and since it is particularly simple, it is maybe a good default.

7. CONCLUSION

We have shown how a hybrid bitmap format, combining three container types (arrays, bitmaps and runs) in a two-level tree could generally surpass competitive implementations of other popular formats (Concise, WAH, EWAH), being up to hundreds of times faster. For analytical applications, where the bitmaps are not constantly updated, and where we can afford to sort the data prior to indexing, applying run compression to the Roaring format is particularly appealing. The new format has been adopted by existing systems such as Druid and Apache Kylin.

Among future work, we plan to make even better use of the existing and upcoming hardware. For example, operations over Roaring bitmaps could be parallelized at the level of the containers. We could explicitly seek to exploit single-instruction-multiple-data (SIMD) instructions. We could adapt Roaring bitmaps to other processor architectures such as GPUs and Intel's Xeon Phi.

REFERENCES

1. Kaser O, Lemire D. Compressed bitmap indexes: beyond unions and intersections. *Software: Practice and Experience* 2014; doi:10.1002/spe.2289. In Press.
2. 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.
3. Antoshenkov G. Byte-aligned bitmap compression. *DCC'95*, IEEE Computer Society: Washington, DC, USA, 1995; 476.
4. Wu K, Stockinger K, Shoshani A. Breaking the curse of cardinality on bitmap indexes. *SSDBM'08*, Springer: Berlin, Heidelberg, 2008; 348–365.
5. 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.
6. 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**(2):1382–1393, doi:10.14778/1920841.1921011.
7. Chambi S, Lemire D, Kaser O, Godin R. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* 2015; doi:10.1002/spe.2325.
8. Grand A. LUCENE-5983: RoaringDocIdSet. `https://issues.apache.org/jira/browse/LUCENE-5983` [last checked March 2015] 2014.
9. 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.
10. Guzun G, Canahuate G, Chiu D, Sawin J. A tunable compression framework for bitmap indices. *ICDE'14*, IEEE, 2014; 484–495.
11. O'Neil E, O'Neil P, Wu K. Bitmap index design choices and their performance implications. *IDEAS'07*, IEEE, 2007; 72–84.
12. 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.
13. 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.
14. Lemire D, Boytsov L, Kurz N. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience* 2015; doi:10.1002/spe.2326.
15. Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, *et al.*. C-Store: a column-oriented DBMS. *VLDB'05*, ACM: New York, NY, USA, 2005; 553–564.
16. Warren HS Jr. *Hacker's Delight*. 2nd ed. edn., Addison-Wesley: Boston, 2013.
17. Bentley JL, Yao ACC. An almost optimal algorithm for unbounded searching. *Information Processing Letters* 1976; **5**(3):82–87.
18. O'Neil P, Quass D. Improved query performance with variant indexes. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, 1997; 38–49.
19. Nagendra M, Candan KS. Efficient processing of skyline-join queries over multiple data sources. *ACM Trans. Database Syst.* Jun 2015; **40**(2):10:1–10:46, doi:10.1145/2699483.

Table IV. Memory-mapped relative timings (Roaring+Run=1)

(a) Random value access

	EWAH 64-bit	EWAH	Roaring	Roaring+Run
CensusInc	21	33	0.91	1.0
CensusInc$^{\text{sort}}$	9.2	11	0.92	1.0
Census1881	140	250	1.0	1.0
Census1881$^{\text{sort}}$	13	16	0.97	1.0
Weather	94	170	0.96	1.0
Weather$^{\text{sort}}$	27	31	0.88	1.0
Wikileaks	20	20	0.93	1.0
Wikileaks$^{\text{sort}}$	6.2	6.3	1.0	1.0

(b) successive intersections

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run
CensusInc	17	2.3	3.6	1.0	1.0
CensusInc$^{\text{sort}}$	6.2	1.3	1.6	1.3	1.0
Census1881	140	79	140	0.98	1.0
Census1881$^{\text{sort}}$	19	12	15	1.0	1.0
Weather	13	1.8	2.9	0.99	1.0
Weather$^{\text{sort}}$	6.4	2.2	2.4	1.6	1.0
Wikileaks	5.8	3.3	3.4	0.83	1.0
Wikileaks$^{\text{sort}}$	5.3	3.4	3.4	1.2	1.0

(c) successive unions

	Concise	EWAH 64-bit	EWAH	Roaring	Roaring+Run
CensusInc	28	2.5	4.1	1.0	1.0
CensusInc$^{\text{sort}}$	7.8	1.1	1.5	1.0	1.0
Census1881	210	21	38	0.9	1.0
Census1881$^{\text{sort}}$	22	3.7	4.3	1.0	1.0
Weather	27	2.2	4.2	0.96	1.0
Weather$^{\text{sort}}$	15	2.5	3.1	1.6	1.0
Wikileaks	15	3.7	3.9	1.5	1.0
Wikileaks$^{\text{sort}}$	8.5	1.8	2.1	1.3	1.0

(d) Entire unions (naïve)

	Concise	Concise$\star$	EWAH 64-bit	EWAH	Roaring	Roaring+Run
CensusInc	16	110	1.7	2.8	2.3	1.0
CensusInc$^{\text{sort}}$	2.7	2.1	0.88	1.1	3.3	1.0
Census1881	210	21	4.4	13	0.53	1.0
Census1881$^{\text{sort}}$	100	2.2	12	15	0.73	1.0
Weather	45	190	3.4	5.4	0.96	1.0
Weather$^{\text{sort}}$	14	1.4	2.6	3.3	1.7	1.0
Wikileaks	34	1.7	3.0	5.4	0.28	1.0
Wikileaks$^{\text{sort}}$	71	2.3	9.3	11	1.1	1.0

(e) Entire unions (priority queue)

	Concise	Concise	EWAH	EWAH	Roaring	Roaring	Roaring	Roaring
	PQ	$\star$	64-bit		naive	pq	+Run nai.	+Run pq
CensusInc	8.0	22	1.4	1.8	0.47	1.1	0.2	1.0
CensusInc$^{\text{sort}}$	8.2	8.0	3.1	4.0	12	26	3.5	1.0
Census1881	12	14	1.2	2.1	0.34	0.68	0.65	1.0
Census1881$^{\text{sort}}$	9.8	10	1.9	1.9	3.3	1.9	4.8	1.0
Weather	13	34	1.0	2.0	0.18	0.93	0.18	1.0
Weather$^{\text{sort}}$	4.1	4.0	2.9	3.3	5.2	12	3.0	1.0
Wikileaks	12	5.7	2.3	2.6	0.96	1.5	3.4	1.0
Wikileaks$^{\text{sort}}$	11	4.7	2.3	2.5	2.5	2.2	2.2	1.0

20. Nagarkar P, Candan KS, Bhat A. Compressed spatial hierarchical bitmap (cshb) indexes for efficiently processing spatial range query workloads. *Proc. VLDB Endow.* Aug 2015; **8**(12):1382–1393, doi:10.14778/2824032.2824038.
21. 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.
22. Lemire D, Kaser O, Aouiche K. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.* Jan 2010; **69**(1):3–28, doi:10.1016/j.datak.2009.08.006.