

Roaring Bitmaps: Implementation of an Optimized Software Library

Daniel Lemire¹, Owen Kaser², Nathan Kurz³, Luca Deri⁴, Chris O’Hara⁵, François Saint-Jacques⁶, Gregory Ssi-Yan-Kai⁷

¹*Université du Québec (TELUQ), Montreal, QC, Canada*

²*Computer Science, UNB Saint John, Saint John, NB, Canada*

³*Orinda, California USA*

⁴*IIT/CNR, ntop, Pisa, Italy*

⁵*Kissmetrics, Bay Area, California, USA*

⁶*AdGear Technologies, Inc., Montreal, QC, Canada*

⁷*42 Quai Georges Gorse, Boulogne-Billancourt, France*

SUMMARY

Compressed bitmap indexes are used in systems such as Git or Oracle to accelerate queries. They represent sets and often support operations such as unions, intersections, differences, and symmetric differences. Several important systems such as Elasticsearch, Apache Spark, Netflix’s Atlas, LinkedIn’s Pivot, Metamarkets’ Druid, Pilosa, Apache Hive, Apache Tez, Microsoft Visual Studio Team Services and Apache Kylin rely on a specific type of compressed bitmap index called Roaring. We present an optimized software library written in C implementing Roaring bitmaps: CRoaring. It benefits from several algorithms designed for the single-instruction-multiple-data (SIMD) instructions available on commodity processors. In particular, we present vectorized algorithms to compute the intersection, union, difference and symmetric difference between arrays. We benchmark the library against a wide range of competitive alternatives, identifying weaknesses and strengths in our software. Our work is available under a liberal open-source license.

KEY WORDS: bitmap indexes; vectorization; SIMD instructions; database indexes; Jaccard index

1. INTRODUCTION

Contemporary computing hardware offers performance opportunities through improved parallelism, by having more cores and better single-instruction-multiple-data (SIMD) instructions. Meanwhile, software indexes often determine the performance of big-data applications. Efficient indexes not only improve latency and throughput, but they also reduce energy usage [1].

Indexes are often made of sets of numerical identifiers (stored as integers). For instance, inverted indexes map query terms to document identifiers in search engines, and conventional database indexes map column values to record identifiers. We often need efficient computation of the intersection ($A \cap B$), the union ($A \cup B$), the difference ($A \setminus B$), or the symmetric difference ($((A \setminus B) \cup (B \setminus A))$) of these sets.

The bitmap (or bitset) is a time-honored strategy to represent sets of integers concisely. Given a universe of n possible integers, we use a vector of n bits to represent any one set. On a 64-bit processor, $\lceil n/64 \rceil$ inexpensive bitwise operations suffice to compute set operations between two bitmaps:

*Correspondence to: Daniel Lemire, 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

- the intersection corresponds to the bitwise AND;
- the union corresponds to the bitwise OR;
- the difference corresponds to the bitwise ANDNOT;
- the symmetric difference corresponds to the bitwise XOR.

Unfortunately, when the range of possible values (n) is too wide, bitmaps can be too large to be practical. For example, it might be impractical to represent the set $\{1, 2^{31}\}$ using a bitset. For this reason, we often use compressed bitmaps.

Though there are many ways to compress bitmaps (see § 2.1), several systems rely on an approach called Roaring including Elasticsearch [2], Metamarkets' Druid [3], eBay's Apache Kylin [4], Netflix's Atlas [5], LinkedIn's Pinot [6], Pilosa [7], Apache Hive, Apache Tez, Microsoft Visual Studio Team Services [8], and Apache Spark [9, 10]. In turn, these systems are in widespread use: e.g., Elasticsearch provides the search capabilities of Wikipedia [11]. Additionally, Roaring is used in machine learning [12], in data visualization [13], in natural language processing [14], in RDF databases [15], and in geographical information systems [16]. Wang et al. recommend Roaring bitmaps as a superior alternative [17] to other compressed bitmaps.

Roaring partitions the space $[0, 2^{32})$ into *chunks* consisting of ranges of 2^{16} integers $([0, 2^{16}), [2^{16}, 2^{17}), \dots)$ [18, 19].[†] For a value in the set, its least significant sixteen bits are stored in a container corresponding to its chunk (as determined by its most significant sixteen bits), using one of three possible container types:

- bitset containers made of 2^{16} bits or 8 kB;
- array containers made of up to 4096 sorted 16-bit integers;
- run containers made of a series of sorted $\langle s, l \rangle$ pairs indicating that all integers in the range $[s, s + l]$ are present.[‡]

At a high level, we can view a Roaring bitmap as a list of 16-bit numbers (corresponding to the most-significant 2 B of the values present in the set), each of which is coupled with a reference to a container holding another set of 16-bit numbers corresponding to the least-significant 2 B of the elements sharing the same prefix. See Fig. 1.

We dynamically pick the container type to minimize memory usage. For example, when intersecting two bitset containers, we determine whether the result is an array or a bitset container on-the-fly. As we add or remove values, a container's type might change. No bitset container may store fewer than 4097 distinct values; no array container may store more than 4096 distinct values. If a run container has more than 4096 distinct values, then it must have no more than 2047 runs, otherwise the number of runs must be less than half the number of distinct values.

Roaring offers logarithmic-time random access: to check for the presence of a 32-bit integer, we seek the container corresponding to the sixteen most-significant bits using a binary search. If this prefix is not in the list, we know that the integer is not present. If a bitmap container is found, we check the corresponding bit; if an array or run container is found, we use a binary search.

The early implementations of Roaring are in Java [18, 21]. Though Java offers high performance, it also abstracts away the hardware, making it more difficult for the programmer to tune the software to the microarchitecture. As an experiment, we built an optimized Roaring bitmap library in C (CRoaring). Based on this work, we make two main contributions:

- We present several non-trivial algorithmic optimizations. See Table 1. In particular, we show that a collection of algorithms exploiting single-instruction-multiple-data (SIMD) instructions can enhance the performance of a data structure like Roaring in some cases, above and beyond what state-of-the-art optimizing compilers can achieve. To our knowledge, it is the first work to report on the benefits of advanced SIMD-based algorithms for compressed bitmaps.

Though the approach we use to compute array intersections using SIMD instructions in § 4.2 is not new [22, 23], our work on the computation of the union (§ 4.3), difference (§ 4.4) and

[†]There are 64-bit extensions to Roaring [20].

[‡]The Roaring implementation [2] found in Elasticsearch lacks run containers but has “negated” array containers, where all but the missing values are stored.

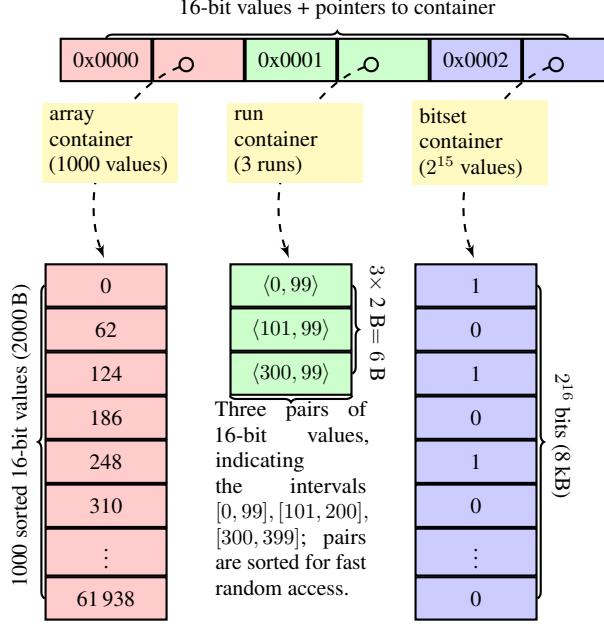


Figure 1. Roaring bitmap containing the first 1000 multiples of 62, all integers in the intervals $[2^{16}, 2^{16} + 100]$, $[2^{16} + 101, 2^{16} + 201]$, $[2^{16} + 300, 2^{16} + 400]$ and all even integers in $[2 \times 2^{16}, 3 \times 2^{16}]$.

symmetric difference (§ 4.4) of arrays using SIMD instructions might be novel and of general interest.

- We benchmark our C library against a wide range of alternatives in C and C++. Our results provide guidance as to the strengths and weaknesses of our implementation.

We focus primarily on our novel implementation and the lessons we learned: we refer to earlier work for details regarding the high-level algorithmic design of Roaring bitmaps [18, 19]. Because our library is freely available under a liberal open-source license, we hope that our work will be used to accelerate information systems.

2. INTEGER-SET DATA STRUCTURES

The simplest way to represent a set of integers is as a sorted array, leading to an easy implementation. Querying for the presence of a given value can be done in logarithmic time using a binary search. Efficient set operations are already supported in standard libraries (e.g., in C++ through the Standard Template Library or STL). We can compute the intersection, union, difference, and symmetric difference between two sorted arrays in linear time: $O(n_1 + n_2)$ where n_1 and n_2 are the cardinalities of the two arrays. The intersection and difference can also be computed in time $O(n_1 \log n_2)$, which is advantageous when one array is small.

Another conventional approach is to implement a set of integers using a hash set—effectively a hash table without any values attached to the keys. Hash sets are supported in the standard libraries of several popular languages (e.g., in C++ through `unordered_set`). Checking for the presence of a value in a hash set can be done in expected constant time, giving other hash-set operations favorable computational complexities. For example, adding a set of size n_2 to an existing set of size n_1 can be done in expected linear time $O(n_2)$ —an optimal complexity. The intersection between two sets can be computed in expected $O(\min(n_1, n_2))$ time. However, compared to a sorted array, a hash set is likely to use more memory. Moreover, and maybe more critically, accessing data in the hash set involves repeated random accesses to memory, instead of the more efficient sequential access made possible by a sorted array.

Table 1. Our Roaring Optimizations and how they relate to prior work.

containers	optimization	section	prior work
bitset → array	converting bitsets to arrays	§ 3.1	
bitset+array	setting, flipping or resetting the bits of a bitset at indexes specified by an array, with and without cardinality tracking	§ 3.2	
bitset	computing the cardinality using a vectorized Harley-Seal algorithm	§ 4.1.1	[24]
bitset+bitset	computing AND/OR/XOR/ANDNOT between two bitsets with cardinality using a vectorized Harley-Seal algorithm	§ 4.1.2	[24]
array+array	computing the intersection between two arrays using a vectorized algorithm	§ 4.2	[22, 23]
array+array	computing the union between two arrays using a vectorized algorithm	§ 4.3	
array+array	computing the difference between two arrays using a vectorized algorithm	§ 4.4	
array+array	computing the symmetric difference between two arrays using a vectorized algorithm	§ 4.5	

2.1. Compressed Bitsets

A bitset (or bitmap) has both the benefits of the hash set (constant-time random access) and of a sorted array (good locality), but suffers from impractical memory usage when the universe size is too large compared to the cardinality of the sets. So we use compressed bitmaps. Though there are alternatives [25], the most popular bitmap compression techniques are based on the word-aligned RLE compression model inherited from Oracle (BBC [26]): WAH [27], Concise [28], EWAH [29] (used by the Git [30] version control system), COMPAX [31], VLC [32], VAL-WAH [33], among others [34, 35, 36]. On a W -bit system, the r bits of the bitset are partitioned into sequences of W' consecutive bits, where $W' \approx W$ depends on the technique used; for EWAH, $W' = W$; for WAH and many others, $W' = W - 1$. When such a sequence of W' consecutive bits contains only 1s or only 0s, we call it a *fill* word, otherwise we call it a *dirty* word. For example, using $W' = 8$, the uncompressed bitmap 0000000001010000 contains two words, a fill word (00000000) and a dirty word (01010000). Techniques such as BBC, WAH or EWAH use special marker words to compress long sequences of identical fill words. When accessing these formats, it may be necessary to read every compressed word to determine whether it indicates a sequence of fill words, or a dirty word. The EWAH format supports a limited form of skipping because its marker words not only to give the length of the sequences of fill words, but also the number of consecutive dirty words. EWAH was found to have superior performance to WAH and Concise [33] in an extensive comparison. A major limitation of formats like BBC, WAH, Concise or EWAH is that random access is slow. That is, to check whether a value is present in a set can take linear time $O(n)$, where n is the compressed size of the bitmap. When intersecting a small set with a more voluminous one, these formats may have suboptimal performance.

Guzun et al. [37] compared Concise, WAH, EWAH, Roaring (without run containers) with a hybrid system combining EWAH and uncompressed bitsets. In many of their tests (but not all), Roaring was superior to Concise, WAH, and EWAH. Similarly, Wang et al. [17] compared a wide range of bitmap compression techniques. They conclude their comparison by a strong recommendation in favor of Roaring: “Use Roaring for bitmap compression whenever possible. Do not use other bitmap compression methods such as BBC, WAH, EWAH, PLWAH, CONCISE, VALWAH, and SBH.”

2.2. BitMagic

Mixing container types in one data structure is not unique to Roaring. O’Neil and O’Neil’s RIDBit is an external-memory B-tree of bitsets and lists [38, 39]. Culpepper and Moffat’s HYB+M2 mixes compressed arrays with bitmaps [40]—Lemire et al. [22] decomposed HYB+M2 into chunks of bitsets and arrays that fit in CPU cache.

The BitMagic library is probably the most closely related data structure [41] with a publicly available implementation. Like Roaring, it is a two-level data structure similar to RIDBit [38, 39]. There are, however, a few differences between Roaring and BitMagic:

- In BitMagic, special pointer values are used to indicate a full (containing 2^{16} values) or an empty container; Roaring does not need to mark empty containers (they are omitted) and it can use a run container to represent a full container efficiently.
- Roaring relies on effective heuristics to generate a memory-efficient container. For example, when computing the union between two array containers, we guess whether the output is likely to be more efficiently represented as a bitset container, as opposed to an array. See Lemire et al. [19] for a detailed discussion. BitMagic does not attempt to optimize the resulting containers: e.g., the intersection of two bitset containers is a bitset container, even when another container type could be more economical.

Roaring keeps a cardinality counter updated for all its bitset containers. BitMagic keeps track of the set cardinality, but not at the level of the bitset containers. Hence, whereas deleting a value in a bitset container in Roaring might force a conversion to an array container, BitMagic cannot know that a conversion is necessary without help: the user must manually call the `optimize` method.

When aggregating bitmaps, it is not always useful to maintain the cardinality of the intermediate result: for this purpose, Roaring allows specific *lazy* operations. The BitMagic library, instead, uses a flag to indicate whether the precomputed cardinality is valid, and systematically invalidates it following any aggregation operation. Thus, by default, BitMagic operations can be faster because the cardinality of the bitset containers is not tracked, but the resulting bitmaps may not be compressed as well as they could be: bitset containers might be used to store few values.

Intuitively, one might think that tracking the cardinality of a bitset as we operate on it—as we do in Roaring by default—could be expensive. However, as we show in § 3 and § 4, careful optimization can make the overhead of tracking the cardinality small on current processors.

- Roaring uses a key-container array, with one entry per non-empty container; BitMagic’s top-level array has $\lceil n/2^{16} \rceil$ entries to represent a set of values in $[0, n)$. Each entry is made of a pointer to a container. Such a flat layout might give BitMagic a speed advantage over Roaring in some cases. For example, when doing random-access queries, Roaring relies on a moderately expensive binary search at the top level whereas BitMagic has direct access. Nevertheless, BitMagic’s simpler layout has a higher memory usage when many of its containers are empty.
- In addition to full and empty containers, BitMagic supports two kinds of containers: bitsets and “gap” containers. They are equivalent to Roaring’s bitset and run containers, but BitMagic has no equivalent to Roaring’s array containers. Moreover, while both CRoaring and BitMagic use dynamic arrays as containers, CRoaring attempts to use as little memory as possible, while BitMagic always uses one of a handful of possible run-container sizes (i.e., in 2-byte words: 128, 256, 512 or 1280). This difference suggests that Roaring may offer better compression ratios for some datasets. However, it may be easier to provide a custom memory allocator for BitMagic.
- When appropriate, Roaring uses an intersection approach based on binary search between array containers instead of systematically relying on a linear-time merge like RIDBit [38, 39]. That is, we use galloping intersection (also known as exponential intersection) to aggregate two sorted arrays of sizes n_1, n_2 in linearithmic time ($O(\min(n_1, n_2) \log \max(n_1, n_2))$) [42]. BitMagic does not use array containers.

- Unlike the current Roaring implementations, the BitMagic library uses *tagged pointers* to distinguish the container types: it uses the least significant bit of the address value as a container-type marker bit. In our C implementation, Roaring uses an array of bytes, using one byte per container, to indicate the container type.
- BitMagic includes algorithms for fast random sampling unlike Roaring.

Overall, BitMagic is simpler than Roaring, but we expect that it can sometimes use more memory.

BitMagic includes the following SIMD-based optimizations on x64 processors with support for the SSE2 and SSE4 instruction sets:

- It uses manually optimized SIMD instructions to compute the AND, OR, XOR and ANDNOT operations between two bitset containers.
- It uses a mix of SIMD instructions and the dedicated population-count instruction (`popcnt`) for its optimized functions that compute only the cardinality of the result from AND, OR, XOR and ANDNOT operations between two bitset containers.
- It uses SIMD instructions to negate a bitset quickly.

Naturally, many of our Roaring optimizations (see § 4) would be applicable to BitMagic and similar formats.

3. FASTER ARRAY-BITSET OPERATIONS WITH BIT-MANIPULATION INSTRUCTIONS

Like most commodity processors, Intel and AMD processors benefit from *bit-manipulation instructions* [43]. Optimizing compilers often use them, but not always in an optimal manner.

3.1. Converting Bitsets To Arrays

Two useful bit-manipulation instructions are `blsi`, which sets all but the least significant 1-bit to zero (i.e., $x \& -x$ in C), and `tzcnt`, which counts the number of trailing zeroes (largest i such as that $x/2^i$ is an integer). Using the corresponding Intel intrinsics (`_blsi_u64` and `_mm_tzcnti_64`), we can extract the locations of all 1-bits in a 64-bit word (`w`) to an array (`out`) efficiently.

```
pos = 0
while (w != 0) {
    uint64_t temp = _blsi_u64(w);
    out[pos++] = _mm_tzcnti_64(w);
    w ^= temp;
}
```

Such code is useful when we need to convert a bitset container to an array container. We can ensure that only a handful of instructions are needed per bit set in the bitset container.

3.2. Array-Bitset Aggregates

Several other such instructions are useful when implementing Roaring. For example, the `bt` (bittest) instruction returns the value of a bit whereas other bit-manipulation instructions can set (`bts`), clear (`btr`) and flip bits (`btc`) as they query their value. On a Pentium 4, these instructions had a throughput of one instruction every two cycles and a latency of six cycles. However, starting with the Intel Sandy Bridge microarchitecture (2011), they became much faster with a throughput of two instructions every cycle and a latency of one cycle [44] for data in registers.

Roaring has bitset containers, which are implemented as arrays of 64-bit words. Some of the most common operations on bitsets involve getting, setting, flipping or clearing the value of a single bit in one of the words. Sometimes it is also necessary to determine whether the value of the bit was changed.

We are interested in the following scenario: given a bitset and an array of 16-bit values, we wish to set (or clear or flip) all bits at the indexes corresponding to the 16-bit values. For example, if the array is made of the values $\{1, 3, 7, 96, 130\}$, then we might want to set the bits at indexes 1, 3, 7, 96, 130 to 1. This operation is equivalent to computing the union between a bitset and an array container.

- If we do not care about the cardinality, and merely want to set the bit, we can use the simple C expression `(w[pos >> 6] |= UINT64_C(1) << (pos & 63))`. Compilers can translate this expression into two shifts (one to identify the word index and one to shift the single set bit) and a bitwise OR.

On a recent x64 processor (see § 5.2) and using a recent popular C compiler (see § 5.1), we estimate that we can set one bit every ≈ 3.6 cycles, taking into account the time required to read the positions of the bits to be set from an array.

We can do better by using a single shift (to identify the word index) followed by the instruction `bts` (“bit test and set”). Indeed, if the position of the bit in the bitset is given by the `pos` variable, then we shift `pos` right by 6 bits and store the result in `offset`. With the help of `offset`, we fetch the affected word from memory into a variable `load`. We call `bts` with `pos` and `load`. This sets the corresponding bit in `load`, so we can store it back in memory at index `offset`. Thus we generate a total of four instructions (one load, one shift, one `bts` and one store)—assuming that the position of the bit to set is already in a register. We find that we are able to set bits about twice as fast (one every ≈ 1.7 cycles). We benefit from the fact that processors are superscalar, so that they can execute more than one instruction per cycle.

- We could set bits and track the cardinality but we expect branch predictions to be sometimes difficult, and branch mispredictions can be expensive (e.g., over 10 cycles of penalty per misprediction). Using the following C code, we set bits and track the cardinality while avoiding branches.

```
uint64_t old_w = words[pos >> 6];
uint64_t new_w = old_w | (UINT64_C(1) << (pos & 63));
cardinality += (old_w ^ new_w) >> (pos & 63);
words[pos >> 6] = new_w;
```

This code first retrieves the word `old_w` at index `pos/64` (or equivalently `pos >> 6`) in an array of 64-bit words `words`. We then set the bit at index `pos % 64` (or equivalently `pos & 63`) to 1, and call the result `new_w`. We can write back `new_w` to the array `words`, and use the bitwise XOR between `old_w` and (`new_w`) to determine whether the bit value was changed. We get about ≈ 3.6 cycles per bit set, or about the same speed we get when we use similar C code without tracking the cardinality.

We can also achieve the same result, set the bit, and adjust the cardinality counter as follows: Use a shift (to identify the word index), the instruction `bts`, then an instruction like `sbb` (“integer subtraction with borrow”) to modify the cardinality counter according to the carry flag eventually set by `bts`. Indeed, if the position of the bit in the bitset is given by the `pos` variable, then we can proceed as follows:

- We shift `pos` right by 6 bits (e.g., `shrx`) and store the result in `offset`.
- We load the 64-bit word at index `offset` and store it in `load`.
- We call `bts` with `pos` and `load`. This sets the corresponding bit in `load`, so we can store it back in memory at index `offset`.
- The call to `bts` stored the value of the bit in the carry (CF) flag. If the bit was set, then the carry flag has value 1, otherwise it has value 0. We want to increment our counter when the carry flag has value 1, and leave it unchanged when it has value 0. Thus we can call the subtract with borrow (`sbb`) instruction, passing our cardinality counter as well as the parameter -1. It will add -1 to the carry flag (getting 0 and -1), and then subtract this value from the counter, effectively adding 0 or 1 to it.

In total, we need five instructions (one load, one shift, one `bts`, one store and one `sbb`), which is one more instruction than if we did not have to track the cardinality.

With this approach, we find that we can set a bit while maintaining a cardinality counter every ≈ 1.8 cycles on our processor. It is only 0.1 cycles per bit set slower than if we do not track the cardinality.

According to our experiments, you can set a bit, or set a bit while maintaining the cardinality, with almost the same throughput—even when all the data is in L1 processor cache. A similar conclusion

holds for bit flipping or bit clearing. Thus it is inexpensive to track the cardinality of bitset containers as we modify individual bits on recent x64 processors.

We also find that if we use bit-manipulation instructions and hand-tuned assembly, we can roughly double the speed at which we can change bits in a bitset, compared to compiler-generated machine code with all optimization flags activated. It is true not only when setting bits, but also when clearing or flipping them—in which cases we need to use the btr and btc instructions instead. Informally, we tested various C compilers and could not find any that could perform nearly as well as hand-tuned assembly code. This may, of course, change in the future.

4. VECTORIZED PROCESSING

Modern commodity processors use parallelism to accelerate processing. SIMD instructions offer a particular form of processor parallelism [45] that proves advantageous for processing large volumes of data [46]. Whereas regular instructions operate on a single machine word (e.g., 64 bits), SIMD instructions operate on large registers (e.g., 256 bits) that can be used to represent “vectors” containing several distinct values. For example, a single SIMD instruction can add sixteen 16-bit integers in one 32 B vector register to the corresponding 16-bit integers in another vector register using a single cycle. Moreover, modern processors are capable of superscalar execution of certain vector instructions, allowing the execution of several SIMD instructions per cycle. For example, an Intel Skylake processor is capable of executing two vector additions every cycle [44].

SIMD instructions are ideally suited for operations between bitset containers. When computing the intersection, union, difference or symmetric difference between two (uncompressed) bitsets, we only have to load the data in registers, apply a logical operation between two words (AND, OR, AND NOT, or XOR) and, optionally, save the result to memory. All of these operations have corresponding SIMD instructions. So, instead of working over 64-bit words, we work over larger words (e.g., 256 bits), dividing the number of instruction (e.g., $\div 4$) and giving significantly faster processing.

Historically, SIMD instructions have gotten wider and increasingly powerful. The Pentium 4 was limited to 128-bit instructions. Later x64 processors introduced increasingly powerful 128-bit instruction sets: SSSE3, SSE4, SSE4.1, SSE4.2. Today, we find rich support for 256-bit vector registers in the AVX2 instruction set available in recent x64 processors from Intel (starting with the Haswell microarchitecture, 2013) and AMD (starting with the Excavator microarchitecture, 2015). Upcoming Intel commodity processors will support 512-bit SIMD instructions (AVX-512). In many cases, the benefits of SIMD instructions increase as they get wider, which means that SIMD instructions tend to become more compelling with time.

Compilers can automatically translate C or C++ code into the appropriate SIMD instructions, even when the code does not take into account vectorization (scalar code). However, for greater gains, it is also possible to design algorithms and code that take into account the vector instructions. In these instances, we can use the SIMD intrinsics available in C and C++ to call SIMD instructions without having to use assembly code. See Table 2. These intrinsics are supported across several compilers (LLVM’s Clang, GNU GCC, Intel’s compiler, Visual Studio).

4.1. Vectorized Population Count Over Bitsets

We can trivially vectorize operations between bitsets. Indeed, it suffices to compute bitwise operations over vectors instead of machine words. By aggressively unrolling the resulting loop, we can produce highly efficient code. Optimizing compilers can often automatically vectorize such code. It is more difficult, however, to also compute the cardinality of the result efficiently. Ideally, we would like to vectorize simultaneously the operations between bitsets and the computation of the cardinality of the result.

4.1.1. Vectorized Harley-Seal Population Count Commodity processors have dedicated instructions to count the 1-bits in a word (the “population count”): popcnt for x64 processors and cnt for the

Table 2. Some relevant AVX and AVX2 instructions with reciprocal throughput^a in CPU cycles on recent (Skylake) Intel processors.

instruction	C intrinsic	description	rec. throughput [44]
vpand	_mm256_and_si256	256-bit AND	0.33
vpor	_mm256_or_si256	256-bit OR	0.33
vpxor	_mm256_xor_si256	256-bit XOR	0.33
vpsadbw	_mm256_sad_epu8	sum of the absolute differences of the byte values to the low 16 bits of each 64-bit word	1
vpaddq	_mm256_add_epi64	add 64-bit integers	0.5
vpshufb	_mm256_shuffle_epi8	shuffle bytes within 128-bit lanes	1

^a The reciprocal throughput is the number of processor clocks it takes for an instruction to execute.
A reciprocal throughput of 0.5 means that we execute two such instructions per clock.

64-bit ARM architecture. On recent Intel processors, `popcnt` has a throughput of one instruction per cycle for both 32-bit and 64-bit registers [44].

Maybe surprisingly, Muła et al. found that it is possible to do better [24]. We can use a vectorized approach based on the circuit-based Harley-Seal algorithm [47]. It is inspired by a carry-save adder (CSA) circuit: given 3 bit values ($a, b, c \in \{0, 1\}$), we can sum them up into the two-bit value ($a \text{ XOR } b$) XOR c (least significant) and ($a \text{ AND } b$) OR (($a \text{ XOR } b$) AND c) (most significant). We can sum 3 individual bit values to a 2-bit counter using 5 logical operations, and we can generalize this approach to 256-bit vectors. Starting with 3 input vectors, we can generate two new output vectors, one containing the least significant bits and one containing the most significant (or *carry*) bits with 5 bitwise logical operations (two XORs, two ANDs and one OR):

```
void CSA(_m256i *h, _m256i *l, _m256i a,
         _m256i b, _m256i c) {
    // starts with a,b,c, sets H,L, u is temp.
    _m256i u = _mm256_xor_si256(a, b);
    *h = _mm256_or_si256(_mm256_and_si256(a, b),
                          _mm256_and_si256(u, c));
    *l = _mm256_xor_si256(u, c);
}
```

From such an adder function, we can derive an efficient population-count function, effectively composing 3-input adders into a circuit with 16 inputs, which encodes the population count of its 16 inputs as a 5-bit binary number. For instance, Knuth presents a construction [48] requiring only 63 logical operations in the case of a 16-input circuit. Although using slightly more logical operations (75), the iterated Harley-Seal approach goes further: it increments the existing contents of a 5-bit binary number by the population count.

In a SIMD setting, our Harley-Seal circuit takes sixteen 256-bit vectors and updates five 256-bit vectors serving as a bit-sliced accumulator [49]: in other words, if one counts the number of 1s amongst the least-significant bits of the 16 inputs, the population count (0 to 16) affects the least-significant bits of the 5 accumulator vectors. Similarly, the population count of the second-least-significant bits of the 16 inputs affects the second-least-significant bits of the accumulator vectors. Equivalently, the 5 accumulator vectors can be viewed as providing 256 different 5-bit accumulators. The accumulator vectors are named ones, twos, fours, eights, and sixteens, reflecting their use in the accumulator: if the least significant bits of the accumulator vectors are all initially zero and the population count of the least-significant bits of the 16 input vectors is 9, then our Harley-Seal approach will result in the least-significant bits of eights and ones being set, whereas the least-significant bits of sixteens, fours and twos would be clear. (The accumulator went from storing 0 to storing 9.) If the accumulator updated a second time with the same inputs, then the least-significant bits of sixteens and twos (only) would be set.

Fig. 4 (from [24]) illustrates a simplified case with only ones, twos and fours. Fig. 5 shows the high-level process: We load 16 vectors from the source bitset, we compute the bitwise operations,

```

__m256i popcount256(__m256i v) {
    __m256i lookuppos = _mm256_setr_epi8(
        4 + 0, 4 + 1, 4 + 1, 4 + 2,
        4 + 1, 4 + 2, 4 + 2, 4 + 3,
        4 + 1, 4 + 2, 4 + 2, 4 + 3,
        4 + 2, 4 + 3, 4 + 3, 4 + 4,
        4 + 0, 4 + 1, 4 + 1, 4 + 2,
        4 + 1, 4 + 2, 4 + 2, 4 + 3,
        4 + 1, 4 + 2, 4 + 2, 4 + 3,
        4 + 2, 4 + 3, 4 + 3, 4 + 4);
    __m256i lookupneg = _mm256_setr_epi8(
        4 - 0, 4 - 1, 4 - 1, 4 - 2,
        4 - 1, 4 - 2, 4 - 2, 4 - 3,
        4 - 1, 4 - 2, 4 - 2, 4 - 3,
        4 - 2, 4 - 3, 4 - 3, 4 - 4,
        4 - 0, 4 - 1, 4 - 1, 4 - 2,
        4 - 1, 4 - 2, 4 - 2, 4 - 3,
        4 - 1, 4 - 2, 4 - 2, 4 - 3,
        4 - 2, 4 - 3, 4 - 3, 4 - 4);
    __m256i low_mask = _mm256_set1_epi8(0x0f);
    __m256i lo = _mm256_and_si256(v, low_mask);
    __m256i hi = _mm256_and_si256(_mm256_srli_epi16(v, 4),
        low_mask);
    __m256i popcnt1 = _mm256_shuffle_epi8(lookuppos, lo);
    __m256i popcnt2 = _mm256_shuffle_epi8(lookupneg, hi);
    return _mm256_sad_epu8(popcnt1, popcnt2);
}

```

Figure 2. A C function using AVX2 intrinsics to compute the four population counts of the four 64-bit words in a 256-bit vector.

and obtain the result in ones, twos, fours, eights and sixteens. This processes a 4 kb block of an input bitset using approximately 75 AVX operations, not considering any data-movement instructions (see Fig. 3). We can compute the total value of all 256 accumulators as $16 \times \text{count(sixteens)} + 8 \times \text{count(eights)} + 4 \times \text{count(fours)} + 2 \times \text{count(twos)} + \text{count(ones)}$ (13 more operations), where $\text{count}(b)$ denotes the population count of b . Iterated 16 times, we can process the entire input bitset. Moreover, it is only necessary to read the total value of all 256 accumulators after the last iteration. However, we need to ensure that, at the beginning of each iteration, none of our 5-bit accumulators ever begins with a count above 15 — otherwise, an accumulator might need to store a value exceeding 31. Therefore, at the end of each block we count the 1-bits in each of the four 64-bit words of sixteens and add the result to a vector counter c before zeroing sixteens. To count the 1-bits in sixteens quickly, we use vector registers as lookup tables mapping 4-bit values (integers in $[0, 16]$) to their corresponding population counts, and the vpshufb can effectively look up 32 byte values at once. Thus we can divide each byte of sixteens into its low nibble and high nibble and look up the population count of each. To get a population count for the 64-bit words, we use the instruction vpsadbw (with the intrinsic `_mm256_sad_epu8`): this instruction adds the absolute values of the differences between byte values within 64-bit subwords. We illustrate the code in Fig. 2: the vectors `lookuppos` and `lookupneg` store the positives and negatives of the population counts with a canceling offset of 4 so that their subtraction is the sum of the population counts. After the 16th iteration, the population count of the entire input bitset is computed as $16 \times c + 8 \times \text{count(eights)} + 4 \times \text{count(fours)} + 2 \times \text{count(twos)} + \text{count(ones)}$.

By contrast, in the approach using the dedicated population-count instruction, the compiler generates one load, one popcnt and one add per input 64-bit word. Current x64 processors decompose complex machine instructions into low-level instructions called μ ops. The popcnt approach generates three μ ops per word. For the SIMD approach, we process sixteen 256-bit vectors using 98 μ ops including 16 loads, 32 bitwise ANDs (vpand), 15 bitwise ORs (vpor) and 30 bitwise XORs (vxpor)—or about 1.5 μ ops to process each 64-bit word. That is, the vectorized approach generates half the

```

CSA(&twosA, &ones, ones, A[i], A[i + 1]);
CSA(&twosB, &ones, ones, A[i + 2], A[i + 3]);
CSA(&foursA, &twos, twos, twosA, twosB);
CSA(&twosA, &ones, ones, A[i + 4], A[i + 5]);
CSA(&twosB, &ones, ones, A[i + 6], A[i + 7]);
CSA(&foursB, &twos, twos, twosA, twosB);
CSA(&eightsA, &fours, fours, foursA, foursB);
CSA(&twosA, &ones, ones, A[i + 8], A[i + 9]);
CSA(&twosB, &ones, ones, A[i + 10], A[i + 11]);
CSA(&foursA, &twos, twos, twosA, twosB);
CSA(&twosA, &ones, ones, A[i + 12], A[i + 13]);
CSA(&twosB, &ones, ones, A[i + 14], A[i + 15]);
CSA(&foursB, &twos, twos, twosA, twosB);
CSA(&eightsB, &fours, fours, foursA, foursB);
CSA(&sixteens, &eights, eights, eightsA, eightsB);

```

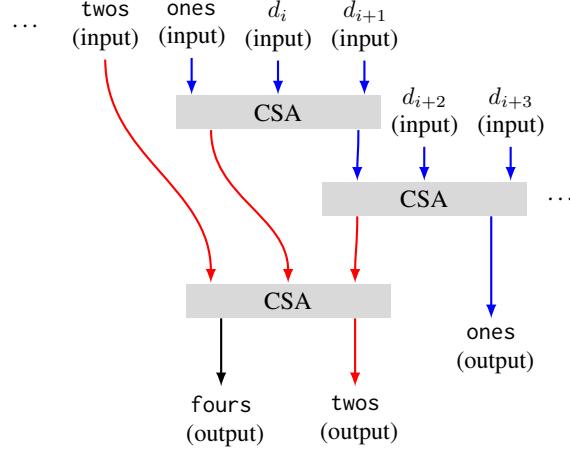
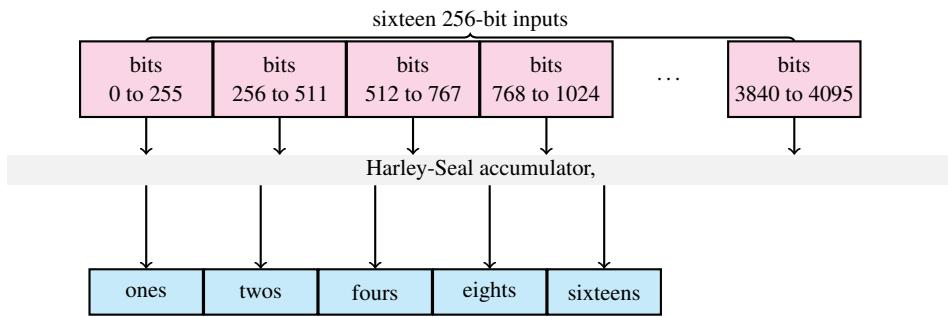
Figure 3. Accumulator circuit over sixteen inputs ($A[i+0], \dots, A[i+15]$)Figure 4. Illustration of the Harley-Seal algorithm aggregating four new inputs ($d_i, d_{i+1}, d_{i+2}, d_{i+3}$) to inputs ones and twos, producing ones, twos and fours [24].

Figure 5. Vectorized population count for a 4 kb bitset, folding sixteen 256-bit inputs into five 256-bit outputs.

number of μ ops. We analyzed our implementations with the IACA code analyser [50] for our target x64 microarchitecture (§ 5.2). Unsurprisingly, IACA predicts a throughput of one word per cycle with the dedicated population-count function, and a throughput of slightly more than two words per cycle with the vectorized approach. Microbenchmarking agrees with IACA and shows that our approach is roughly twice as fast as relying exclusively on dedicated population-count instructions.

4.1.2. Vectorized Operations Over Bitsets With Population Counts To aggregate two bitsets (e.g., with the intersection/AND, union/OR, etc.) while maintaining the cardinality, we found that an efficient approach is to load two machine words from each bitset, compute the bitwise operation, apply the population-count instruction on the two resulting words, and write them out before processing the next words. To benefit from vectorization, we could compute the bitwise logical operations using SIMD instructions, and then reload the result from memory and apply the population-count instructions. The interplay between SIMD instructions and a population-count function operating over 64-bit words would not be ideal.

Though an obvious application of the vectorized population count is the efficient computation of the cardinality of a bitset container, we put also it to good use whenever we need to compute the intersection, union, difference or symmetric difference between two bitset containers, while, at the same time, gathering the cardinality of the result. We can also use it to compute the cardinality of the intersection, union, difference or symmetric difference, without materializing them. As reported by Muła et al. [24], in these cases, the benefits of the vectorized approach can exceed a factor of two. Indeed, if we use a vectorized approach to the population count, we can count the bits directly on the vector registers, without additional data transfer.

4.2. Vectorized Intersections Between Arrays

Because array containers represent integers values as sorted arrays of 16-bit integers, we can put to good use an algorithm based on a vectorized string comparison function present in recent x64 processors [22, 23] (SSE 4.1’s `pcmpistrm`). While intended for string comparisons, we can repurpose the instruction to compute intersections. Each input array is divided into blocks of eight 16-bit integers. There are two block pointers (one for each array) initialized to the first blocks. We use the `pcmpistrm` instruction to conduct an all-against-all comparison of 16-bit integers between these two blocks and we extract matching integers from a mask produced by the instruction. The number of 1s in the resulting mask indicates the number of matching integers. To avoid the cost of comparing pairs of blocks that cannot match, a merge-like process considers the maximum element within each block to find those block-pairs to which `pcmpistrm` should be applied. That is, whenever the current maximum value in the block of one array is smaller or equal to the maximum of the other block, we load the next block. In particular, when the maxima of the two blocks are equal, we load a new block in each array. See Algorithm 1.

Algorithm 1 Generic block-based intersection algorithm.

Require: two non-empty arrays made of a multiple of K distinct values in sorted order

- 1: load a block B_1 of K elements from the first array
 - 2: load a block B_2 of K elements from the second array
 - 3: **while** true **do**
 - 4: write the intersection between the two blocks $B_1 \cap B_2$ to the output
 - 5: **if** $\max B_1 = \max B_2$ **then**
 - 6: load a new block B_1 of K elements from the first array, or terminate if exhausted
 - 7: load a new block B_2 of K elements from the second array, or terminate if exhausted
 - 8: **else if** $\max B_1 < \max B_2$ **then**
 - 9: load a new block B_1 of K elements from the first array, or terminate if exhausted
 - 10: **else if** $\max B_2 < \max B_1$ **then**
 - 11: load a new block B_2 of K elements from the second array, or terminate if exhausted
-

We illustrate a simplified function in Fig. 6. The `_mm_cmpistrm` intrinsic compares all 16-bit integers from one 128-bit vector with all 16-bit integers from another vector, returning a mask. From this mask, we compute the number of matching values (using the `_mm_popcnt_u32` intrinsic to call the `popcnt` instruction). Moreover, we can shuffle the values in one of the vectors (vA) using a shuffling intrinsic (`_mm_shuffle_epi8`) and a table of precomputed shuffling masks (`mask16`). We could design an equivalent AVX2 function [22] using 256-bit vectors, but AVX2 lacks the equivalent of the `_mm_cmpistrm` intrinsic.

```

int32_t intersect(uint16_t *A, size_t lengthA,
                  uint16_t *B, size_t lengthB,
                  uint16_t *out) {
    size_t count = 0; // size of intersection
    size_t i = 0, j = 0;
    int vectorlength = sizeof(__m128i) / sizeof(uint16_t);
    __m128i vA, vB, rv, sm16;
    while (i < lengthA) && (j < lengthB) {
        vA = _mm_loadu_si128((__m128i *)&A[i]);
        vB = _mm_loadu_si128((__m128i *)&B[j]);
        rv = _mm_cmplistrm(vB, vA,
                            _SIDD_UWORD_OPS | _SIDD_CMP_EQUAL_ANY | _SIDD_BIT_MASK);
        int r = _mm_extract_epi32(rv, 0);
        sm16 = _mm_load_si128(mask16 + r);
        __m128i p = _mm_shuffle_epi8(vA, sm16);
        _mm_storeu_si128((__m128i *)out + count), p);
        count += _mm_popcnt_u32(r);
        uint16_t a_max = A[i + vectorlength - 1];
        uint16_t b_max = B[j + vectorlength - 1];
        if (a_max <= b_max) i += vectorlength;
        if (b_max <= a_max) j += vectorlength;
    }
    return count;
}

```

Figure 6. Optimized intersection function between two sorted 16-bit arrays using SSE4.1 intrinsics.

Special-case processing is required when our arrays do not have lengths divisible by eight: we need to finish the algorithm with a small scalar intersection algorithm. Special processing is also required if an array starts with zero, because the `pcmpistrm` instruction is designed for string values and it processes null values as special (string ending) values.

Evidently, we can easily modify this function if we only wish to get the cardinality. For the most part, we merely skip the memory store instructions.

When the `pcmpistrm` instruction is available, as it is on all recent x64 processors, it is advantageous to replace a conventional list intersection function by such a vectorized approach. However, we still rely on a galloping intersection when one of the two input arrays is much smaller. For example, the vectorized approach is obviously not helpful to intersect an array made of 1000 values with one made of two values. There are vectorized versions of the galloping intersection [22] which could further improve our performance in these cases, but we leave such an optimization for future work.

4.3. Vectorized Unions Between Arrays

To compute the union between two array containers, we adapt an approach originally developed for merge sort using SIMD instructions [51]. Given two sorted vectors, we can generate two sorted output vectors that are the result of a “merger” between the two input vectors, by using a sorting network [52, 53] whose branch-free implementation uses SIMD minimum and maximum instructions. We can compute such a merger with eight minimum instructions and eight maximum instructions, given two vectors made of eight 16-bit integers (see Fig. 7).

We can put this fast merge function to good use by initially loading one vector from each sorted array and merging them, so that the small values go into one vector (B_1) and the large values go into the other (B_2). Vector B_1 can be queued for output. We then advance a pointer in one of the two arrays, choosing the array with a smallest new value, and then load a new vector. We merge this new vector with B_2 . We repeat this process; see Algorithm 2. We terminate with a scalar merge because the arrays’ lengths might not be a multiple of the vector size.

The queued vector might include duplicate values. To “deduplicate” them we use a single vectorized comparison between each new vector with a version of itself offset back by one element (taking care to handle the final previously stored value). For the mask resulting from this comparison, we

Algorithm 2 Generic block-based union algorithm.

Require: two non-empty arrays made of a multiple of K distinct values in sorted order

- 1: load a block B_1 of K elements from the first array
- 2: load a block B_2 of K elements from the second array
- 3: Merge blocks B_1 and B_2 so that the smallest elements are in block B_1 and the largest are in block B_2 . Output the elements from B_1 , after removing the duplicates
- 4: **while** there is more data in both lists **do**
- 5: **if** the next value loaded from the first array would be smaller than the next value in the second array **then**
- 6: load a new block B_1 of K elements from the first array
- 7: **else**
- 8: load a new block B_1 of K elements from the second array
- 9: Merge blocks B_1 and B_2 so that the smallest elements are in block B_1 and the largest are in block B_2
- 10: output the elements from B_1 , after removing the duplicates while taking into account the last value written to output
- 11: **while** there is more data in one list **do**
- 12: load a new block B_1 of K elements from the remaining array
- 13: Merge blocks B_1 and B_2 so that the smallest elements are in block B_1 and the largest are in block B_2 ,
- 14: output the elements from B_1 , after removing the duplicates while taking into account the last value written to output
- 15: Output the elements from B_2 , after removing the duplicates

can quickly extract the integers that need to be written, using a look-up table and an appropriate permutation. See Fig. 8.

Our vectorized approach is implemented to minimize mispredicted branches. We use 128-bit SIMD vectors instead of 256-bit AVX vectors. Larger vectors would further reduce the number of branches, but informal experiments indicate that the gains are modest. A downside of using larger vectors is that more of the processing occurs using scalar code, when the array lengths are not divisible by the vector lengths.

4.4. Vectorized Differences Between Arrays

Unlike union and intersection, the difference operator is an asymmetric operation: $A \setminus B$ differs from $B \setminus A$. We remove the elements of the second array from the first array.

If we can compute the intersection quickly between blocks of data (e.g., with the `_mm_cmplstrm` intrinsic), we can also quickly obtain the difference. In our vectorized intersection algorithm (§ 4.2), we identify the elements that are part of the intersection using a bitset. In a difference algorithm, picking blocks of values from the first array, we can successively remove the elements that are present in the second array, by iterating through its data in blocks. The intersection, expressed as a bitset, marks the elements for deletion, and we can conveniently accumulate these indications by taking the bitwise OR of the bitsets. Indeed, suppose that we want to compute $A \setminus B$ where $A = \{1, 3, 5, 7\}$ and $B = \{1, 2, 3, 4, 6, 7, 8, 9\}$ using blocks of 4 integers:

- We compute the intersection between the first (and only) block from A given by $\{1, 3, 5, 7\}$ and the first block from B given by $\{1, 2, 3, 4\}$. We find that 1, 3 are part of the intersection, but not 5, 7. We can represent this result with the bitset `0b0011` (or `0x3`).
- We advance to the next block in the second array, loading $\{6, 7, 8, 9\}$. The intersection between the two blocks is $\{1, 3, 5, 7\} \cap \{6, 7, 8, 9\} = \{7\}$ which we can represent with the bitset `0b1000`.

```

void sse_merge(const __m128i *vInput1,
               const __m128i *vInput2, // input 1 & 2
               __m128i *vecMin, __m128i *vecMax) {
    vecTmp = _mm_min_epu16(*vInput1, *vInput2);
    *vecMax = _mm_max_epu16(*vInput1, *vInput2);
    vecTmp = _mm_alignr_epi8(vecTmp, vecTmp, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    vecTmp = _mm_alignr_epi8(*vecMin, *vecMin, 2);
    *vecMin = _mm_min_epu16(vecTmp, *vecMax);
    *vecMax = _mm_max_epu16(vecTmp, *vecMax);
    *vecMin = _mm_alignr_epi8(*vecMin, *vecMin, 2);
}

```

Figure 7. Fast merge function using Intel SIMD intrinsics, takes two sorted arrays of eight 16-bit integers and produces two vectors (vecMin, vecMax) containing the sixteen integer inputs in a sorted sequence, with the eight smallest integers in vecMin, and eight largest in vecMax.

- We can compute the bitwise OR between the two bitsets `0b0011` and `0b1000` to get `0b1011`. It indicates that all but the third value in the block $\{1, 3, 5, 7\}$ must be removed. So we output $\{3\}$.

Algorithm 3 describes the difference algorithm at a high level.

As with our other algorithms, we need to finish the computation using scalar code because the inputs may not be divisible by the vector size. Moreover, if we only need to compute the size of the difference, we can use the same algorithm, but we simply do not store the result, only tracking the cardinality.

4.5. Vectorized Symmetric Differences Between Arrays

The symmetric difference can be computed using a block-based algorithm similar to the union algorithm (Algorithm 2). When computing the union, we remove duplicates, keeping just one instance of a value repeated twice. E.g., given $\{1, 1, 2, 2, 3\}$, we want to output $\{1, 2, 3\}$. When computing the symmetric difference, we need to entirely remove a value if it is repeated. Thus, given $\{1, 1, 2, 2, 3\}$, we want to get the set $\{3\}$. We can use a fast vectorized function for this purpose (see Fig. 9), and it is similar to the corresponding deduplication function used for the vectorized union (Fig. 8).

In the union, we can immediately write all values after removing duplicates. With the symmetric difference, the largest value in the current block should not be written immediately. Indeed, we need to account for the fact that the next block might start with this largest value, and that it may become necessary to omit it. Thus, we never write the current largest value of the current block when computing the symmetric difference. Instead, it is (potentially) added when the next block is processed — note that `val` in Fig. 9 is computed from `v2` (which has this value), rather than from `n`.

```

// returns how many unique values are found in n
// the last integer of "o" is last written value
int store_unique(_m128i o, _m128i n, uint16_t *output) {
    // new vector v starting with last value of "o"
    // and containing the 7 first values of "n"
    _m128i v = _mm_alignr_epi8(n, o, 16 - 2);
    // compare v and n and create a mask
    int M = _mm_movemask_epi8(
        _mm_packs_epi16(_mm_cmpeq_epi16(v, n), _mm_setzero_si128()));
    // "number" represents the number of unique values
    int number = 8 - _mm_popcnt_u32(M);
    // load a "shuffling" mask
    _m128i key = _mm_loadu_si128(uniqshuf + M);
    // the first "number" values of "val" are the unique values
    _m128i val = _mm_shuffle_epi8(n, key);
    // we store the result to "output"
    _mm_storeu_si128((_m128i *)output, val);
    return number;
}

```

Figure 8. Fast function that takes an vector containing sorted values with duplicates, and stores the values with the duplicates removed, returning how many values were written. We provide a second vector containing the last value written as its last component.

Algorithm 3 Generic block-based difference algorithm.

Require: two non-empty arrays made of a multiple of K distinct values in sorted order

- 1: load a block B_1 of K elements from the first array
 - 2: load a block B_2 of K elements from the second array
 - 3: **while** true **do**
 - 4: mark for deletion the elements in $B_1 \cap B_2$
 - 5: **if** $\max B_1 = \max B_2$ **then**
 - 6: write B_1 to the output after removing the elements marked for deletion
 - 7: load a new block B_1 of K elements from the first array, or terminate if exhausted
 - 8: load a new block B_2 of K elements from the second array, or break the loop if exhausted
 - 9: **else if** $\max B_1 < \max B_2$ **then**
 - 10: write B_1 to the output after removing the elements marked for deletion
 - 11: load a new block B_1 of K elements from the first array, or terminate if exhausted
 - 12: **else if** $\max B_2 < \max B_1$ **then**
 - 13: load a new block B_2 of K elements from the second array, or break the loop if exhausted
 - 14: write B_1 after removing the elements marked for deletion as well as all unprocessed data from the first array to the output
-

Of course, when the algorithm terminates, we must write the largest value encountered if it were not repeated. In any case, we need to terminate the algorithm with scalar code given that the size of our input lists may not be divisible by the vector size.

5. EXPERIMENTS

We present experiments on realistic datasets. Our benchmarks directly carry out operations on collections of in-memory sets. These original sets are immutable: they are not modified during the benchmarks. For a validation of Roaring in a database system with external memory processing, see for example Chambi et al. [3] where they showed that switching to Roaring from Concise improved system-level benchmarks.

```

int store_xor(_m128i o, _m128i n, uint16_t *output) {
    _m128i v1 = _mm_alignr_epi8(n, o, 16 - 4);
    _m128i v2 = _mm_alignr_epi8(n, o, 16 - 2);
    _m128i el = _mm_cmpeq_epi16(v1, v2);
    _m128i er = _mm_cmpeq_epi16(v2, n);
    _m128i erl = _mm_or_si128(el, er);
    int M = _mm_movemask_epi8(
        _mm_packs_epi16(erl, _mm_setzero_si128()));
    int nv = 8 - _mm_popcnt_u32(M);
    _m128i key = _mm_lddq_si128(uniqshuf + M);
    _m128i val = _mm_shuffle_epi8(v2, key);
    _mm_storeu_si128((_m128i *)output, val);
    return nv;
}

```

Figure 9. Fast function taking a vector containing sorted values with duplicates and entirely removing the duplicated values, returning how many values were written. We provide a second vector containing the last value written as its last component.

5.1. Software

We use the GNU GCC compiler (version 5.4) on a Linux server. All code was compiled with full optimization (-O3 -march=native) as it provided the best performance.

We published our C code as the CRoaring library; it is available on GitHub under an Apache license[§]. The CRoaring library is amalgamated into a single source file (`roaring.c`) and a single header file (`roaring.h`) [54]. The code is portable, having been tested on several operating systems (Linux, Windows, macOS), on several processors (ARM, x64) and on several compilers (GNU GCC, LLVM’s Clang, Intel, Microsoft Visual Studio). We offer interfaces to the library in several other languages (e.g., C#, C++, Rust, Go, Python); it is also available as a module of the popular open-source in-memory database Redis [55]. When instantiating the original Roaring bitmaps, we call the functions `roaring_bitmap_run_optimize` and `roaring_bitmap_shrink_to_fit` to minimize memory usage. We use version 0.2.33 of the CRoaring library in our experiments.

To assess the library and compare it with alternatives, we prepared benchmarking software in C and C++, also freely available on GitHub.[¶] We consider the following alternatives:

- We use the C++ BitMagic library^{||} (version 3.7.1, published in July 2016). During compilation,

we enable the SIMD optimizations offered by the library by defining the appropriate flags (BMSSE2OPT and BMSSE4OPT). We instantiate bitmaps using the default C++ template (`bm::bvector<>`), since there is no other documented alternative. We follow the documented approach to optimizing memory usage, calling `set_new_blocks_strat(bm::BM_GAP)` followed by `optimize()` after creating the original bitmaps. We do not call `optimize()` during the timed benchmarks as it may affect negatively the timings.

During the optimization procedure, BitMagic allocates a temporary 8 kB memory block that fails to get deallocated. This adds an unnecessary 8 kB to the memory usage of each bitmap, an undesirable effect. Because this memory allocation is part of the private code of the library, we cannot manually deallocate it without modifying the library. However, as a workaround, we copy each of the original bitmaps after calling `optimize()` using the copy constructor: this new copy does not have such a useless 8 kB memory block.

This extra copy to improve BitMagic’s memory usage makes the creation of the bitmaps slightly slower, but it does not harm BitMagic’s timings in our benchmarks.

[§]<https://github.com/RoaringBitmap/CRoaring>

[¶]<https://github.com/RoaringBitmap/CBitmapCompetition>

^{||}<https://sourceforge.net/projects/bmagic/>

- We use the C++ EWAHBoolArray library for comparison with the EWAH format (version 0.5.15).^{**} Code from this library is part of the Git version-control tool [30]. After creating the original bitmaps, we call their `trim` method to minimize memory usage. The `trim` method is not called during the timed benchmarks.
- For comparison against uncompressed bitsets, we developed a portable C library, `cbitset`^{††} (version 0.1.6). The library is written using portable C, without processor-specific optimizations. It uses 64-bit words. To represent a set of integers in $[0, n)$, it attempts to use as little as $\lceil n/64 \rceil \times 8$ bytes, plus some overhead.
- For comparison against the WAH and Concise formats [28], we use the Concise C++ library^{‡‡} (version 0.1.8). Since Concise and WAH are similar formats, we can support both of them using the same C++ code without sacrificing performance with a C++ template. After creating the original bitmaps, we call their `compact` method to minimize memory usage, but never during the timed benchmarks.

The EWAH, WAH, and Concise formats can work with words of various sizes (e.g., 32 bits, 64 bits). When using 64 bits, we sometimes gain some speed at the expense of a significant increase in memory usage [19]. For simplicity, we only consider 32-bit words, thus maximizing compression.

We also include the standard C++ library bundled with our compiler which provides `std::vector` and `std::unordered_set`.

- We use `std::unordered_set` as the basis for testing hash-set performance. The `std::unordered_set` template is hardly the sole way to implement a hash set in C++, but since it is part of the standard library, we consider it a good default implementation. To reduce memory usage, we call the `rehash` method on our `std::unordered_set` instances after constructing them, passing the number of items as an argument. Implementing set operations for `std::unordered_set` is straightforward. For example, to compute an intersection between two `std::unordered_set`, we pick the smallest, iterate over its content and check each value in the largest one. Our implementations have good computational complexity. Thus computing the intersection between two hash sets of size n_1 and n_2 has complexity $O(\min(n_1, n_2))$, the union and symmetric differences have complexity $O(n_1 + n_2)$, and the difference has complexity $O(n_1)$.
- To minimize the memory usage of our original `std::vector` instances, we call their `shrink_to_fit` method after constructing them, but never during the timed benchmarks. To implement set operations over the `std::vector` data structure, we keep the values sorted and make use of the functions provided by STL (`std::set_intersection`, `std::set_union`, `std::set_difference`, `std::set_symmetric_difference`, ...). These functions take iterators pointing at the inputs and output, so we can use them to, for example, measure the size of the output without necessarily materializing it. To locate a value within an `std::vector`, we use `std::binary_search`, a logarithmic-time function.

Of course, when comparing different software libraries, we expect performance differences due to specific implementation issues. Though this is unavoidable, we alleviate the problem by ensuring that all of our code, and all of the libraries' code is freely available, thus easing reproducibility and further work.

5.2. Hardware

We use a Linux server with an Intel i7-6700 processor (Skylake microarchitecture, 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 (DDR4 2133, double-channel). Our benchmarks do not access disk and are single-threaded. The processor always runs at its highest clock speed and Turbo Boost is disabled. Times are collected

^{**}<https://github.com/lemire/EWAHBoolArray>

^{††}<https://github.com/lemire/cbitset>

^{‡‡}<https://github.com/lemire/Concise>

using the `rdtsc` instruction, as described in Intel’s documentation [56]. Prior to benchmarking, we checked that all results could be reproduced reliably within a margin of error of 5 % or better.

5.3. Data

We focus on the cases where the integer values span a wide range of values and where sets contain many integers. Thus, we make use of eight datasets from earlier work [19, 57]. Our datasets are derived from bitmap indexes built on real-world tables. Some are taken as-is while others are generated from tables that have been lexicographically sorted prior to indexing [29]. We have two sets of bitmaps from each data source (e.g., `CENSUS1881` and `CENSUS1881sort`). For each dataset, we randomly chose 200 sets, each of which contains the record ids of rows matching predicate $A = v$, for some column A and column-value v . We present the basic characteristics of these sets in Table 3. All of the datasets are publicly available as part of our CRoaring software library. The universe size of a dataset is the smallest value n such that all sets are contained in $[0, n)$. The density is the cardinality divided by the universe size.

Table 3. Characteristics of our realistic datasets

	universe size	average cardinality per set	average density
<code>CENSUSINC</code>	199 523	34 610.1	0.17
<code>CENSUSINC^{sort}</code>	199 523	30 464.3	0.15
<code>CENSUS1881</code>	4 277 806	5019.3	0.001
<code>CENSUS1881^{sort}</code>	4 277 735	3404.0	0.0008
<code>WEATHER</code>	1 015 367	64 353.1	0.063
<code>WEATHER^{sort}</code>	1 015 367	80 540.5	0.079
<code>WIKILEAKS</code>	1 353 179	1376.8	0.001
<code>WIKILEAKS^{sort}</code>	1 353 133	1440.1	0.001

We take our datasets as reasonable examples that can serve to illustrate the benefits (as well as the lack of benefit) of our Roaring optimizations. We stress that no single data structure (bitset, array, hash set, compressed bitset) can be best for all datasets and all applications. The choice depends on the data characteristics and is non-trivial [58].

We want to benchmark computational efficiency to evaluate our optimization. Thus we want to avoid disk access entirely. Hence our datasets easily fit in memory by design. We further validate our results with a large in-memory dataset in Appendix B.

In our benchmarks, the original collection of sets is immutable. For example, if we compute the intersection between two sets, we generate a new data structure with the result. This reflects a common usage scenario: e.g., Druid [3] relies on collections of immutable and persistent compressed bitmaps to accelerate queries.

5.4. Memory Usage

Data structures that fail to fit in memory must be retrieved from slower storage media. Thus, everything else being equal, we want our data structures to use as little memory as possible.

Exactly computing the memory usage of complex data structures in C and C++ can be difficult; the same C code running with different libraries or a different operating system might use memory differently. Thus we need to proceed with care.

For the `std::vector` implementation, we know from first principles that the memory usage (when ignoring a small amount of constant overhead) is 32 bits per entry. For the hash sets (`std::unordered_set`), we use a custom memory allocator that keeps track of allocation requests and record the provided byte counts, assuming for simplicity that memory allocations carry no overhead. For continuous storage, this estimate should be accurate, but for a hash set (where there

Table 4. Memory usage in bits per value. For each dataset, the best result is in bold. The hash set is implemented using the STL’s (unordered_set).

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	5.66	32.0	195	4.46	2.60	3.29	3.37	2.94
CENSUSINC ^{sort}	6.01	32.0	195	1.89	0.600	0.640	0.650	0.550
CENSUS1881	524	32.0	195	46.3	15.1	33.8	34.3	25.6
CENSUS1881 ^{sort}	888	32.0	195	16.0	2.16	2.91	2.95	2.48
WEATHER	15.3	32.0	195	8.79	5.38	6.67	6.82	5.88
WEATHER ^{sort}	11.4	32.0	195	0.960	0.340	0.540	0.540	0.430
WIKILEAKS	796	32.0	195	29.8	5.89	10.8	10.9	10.2
WIKILEAKS ^{sort}	648	32.0	195	24.0	1.63	2.63	2.67	2.23

are numerous allocations), we underestimate the true memory usage. Because our custom memory allocation might add an unnecessary computational overhead, we run the timing tests without it.

We proceed similarly with Roaring. Because we implemented Roaring using the C language, we instrument the malloc/free calls so that they register the memory allocated and freed. We also run our experiments with and without this instrumentation, to make sure that we can record the speed without undue overhead. The CRoaring library supports a compact and portable serialization format. For most of our datasets, the in-memory size and the compact serialized size are nearly identical. There are only three datasets where there were noteworthy differences between in-memory and serialized sizes: CENSUS1881^{sort} (2.77 bits per value vs. 2.16 bits per value), WIKILEAKS (2.58 bits per value vs. 1.63 bits per value), and WIKILEAKS^{sort} (7.04 bits per value vs. 5.89 bits per value). Not coincidentally, these are our datasets with the fewest values per set. The difference can be explained by the overhead introduced by each new container. It suggests that the memory layout in the CRoaring library could be improved in some cases, by reducing the memory overhead of the containers.

For uncompressed bitsets, EWAH, WAH, and Concise, we rely on the memory usage in bytes reported by the corresponding software libraries. Because they store all their data in a single array, we can be confident that this approach provides a reliable count.

For the BitMagic library, we rely on the numbers provided by the calc_stat method, retrieving the memory_used attribute. An examination of the source code reveals that it should provide a fair estimate of the memory usage.

We present the estimated memory usage in Table 4:

- The memory usage of the uncompressed bitset approach can be excessive (up to 888 bits per 32-bit value). However, it is less than 32 bits per value in four datasets: CENSUSINC, CENSUSINC^{sort}, WEATHER and WEATHER^{sort}.
- The chosen hash-set implementation is clearly not economical memory-wise, using 195 bits per 32-bit integer. The underlying data structure is a hash table composed of an array of buckets. Because the maximum load factor is 1.0 by default, we have at least as many buckets as we have values. Each bucket is a singly linked list starting with a 64-bit pointer to the first value node. Each value node uses 128 bits: 64 bits for a pointer to the next value node, and 64 bits to store the actual integer value. Though we use 32-bit integers as values, the implementation pads the node size to 128 bits for memory alignment. Thus at least 192 bits (64 + 128) are necessary to store each value, which explains the 195 bits encountered in practice in our datasets.
- The memory usage of EWAH, WAH, and Concise are similar to one another (within 20%). Concise is slightly more economical in most cases. Roaring uses less memory than Concise in all datasets except CENSUSINC^{sort}, where Concise uses 10% less memory. In two cases (CENSUS1881 and WIKILEAKS), Roaring uses much less memory than Concise (30–40%).
- In some cases, BitMagic uses a surprising amount of memory (up to 46.3 bits per 32-bit integer). Something similar was observed by Pieterse et al. [59] who wrote that “although it

Table 5. Time needed to iterate through all values, checking the cardinality in the process. We report the number of CPU cycles used per value.

	bitset	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	7.53	28.5	20.4	5.87	13.1	9.23	9.34
CENSUSINC ^{sort}	6.79	12.6	6.86	5.33	6.03	7.63	7.85
CENSUS1881	42.7	67.5	70.9	5.16	29.4	29.4	31.7
CENSUS1881 ^{sort}	41.4	20.4	11.0	6.32	9.10	10.0	10.1
WEATHER	10.0	40.7	31.7	6.43	15.9	12.4	12.8
WEATHER ^{sort}	6.73	12.9	6.79	5.25	5.87	7.46	7.68
WIKILEAKS	46.2	24.6	12.2	9.61	18.4	16.1	16.6
WIKILEAKS ^{sort}	33.2	13.1	7.96	5.86	7.95	8.95	9.19

[BitMagic] maintained acceptable performance times, its memory usage is exorbitant.” The BitMagic library does not appear to be optimized for low memory usage.

5.5. Sequential Access

It is useful to be able to iterate through all values of a set. All of the data structures under consideration allow for efficient in-order access, except for the hash set. The fastest data structure for this problem is the sorted array. Setting it aside, we compare the other software implementations in Table 5. In this benchmark, we simply go through all values, as a means to compute the total cardinality. WAH and Concise have similar performance, with WAH being marginally faster. EWAH has also similar performance, being slightly faster for some datasets, and slightly slower for others. Depending on the dataset, the worst performance is either with the bitset or the hash set. Indeed, iterating through a bitset can be slow when it is memory inefficient.

The hash set, BitMagic, EWAH, WAH and Concise implementation rely on an iterator approach, as is typical in C++. Our benchmark code takes the form of a simple for loop.

```
for(auto j = b.begin(); j != b.end(); ++j)
    count++;
```

The bitset and CRoaring implementations favor an approach that is reminiscent of the functional for-each construct, as implemented by STL’s `std::for_each` and Java’s `forEach`. They take a function as a parameter, calling the function for each element of the set, continuing as long as the function returns true. The expected function signature has an optional void-pointer parameter.

```
typedef bool (*roaring_iterator)(uint32_t value, void *param);
```

In our benchmark, the void pointer is used to pass a variable that gets incremented with each value encountered.

5.6. Membership Queries

All of the software libraries under consideration support membership queries: given an integer x , we can ask whether it belongs to the set. Recall that for the `std::vector` tests, we use the `std::binary_search` function provided by the standard library.

To test the performance of random access, for each data set, we compute the size of the universe $([0, n])$ and check for the presence of the integers $[n/4]$, $[n/2]$ and $[3n/4]$ in each set of the dataset. We present the results in Table 6.

The bitset consistently has the best performance, using less than a handful of CPU cycles per query. The hash set and BitMagic have the next best random-access performance, followed by Roaring, followed by the arrays (vector). The arrays have much worse performance than Roaring, sometimes by an order of magnitude.

Roaring is inferior to BitMagic in these tests even though they implement a similar two-level data structure. Two design elements can explain this inferior performance:

- Roaring uses a binary search is used to first locate (if it exists) a corresponding container, before searching the container. BitMagic does not require such a binary search because, given values in the range $[0, n]$, it materializes $[n/2^{16}]$ containers in a flat array, with containers marked as empty or full as needed. Thus BitMagic can locate the corresponding array directly, without any branching.
- Roaring can store up to 4096 values in array containers whereas BitMagic never stores more than 1280 values in its array containers. So where Roaring uses large array containers, BitMagic use bitset containers. It is faster to look up the value of a bit within a bitset container than to do a binary search within an array container having more than 1280 values.

Both of these elements also contribute to BitMagic’s higher memory usage.

Both the bitset and the hash set have expected constant-time random-access complexity. However, the bitset is an order of magnitude faster in this instance.

EWAH, WAH, and Concise have terrible random-access performance as expected, with EWAH faring slightly better. Indeed, unlike our other data structures, these suffer from a linear-time random-access complexity, meaning that the time required to check the value of a bit is proportional to the compressed size of the bitmap. Though Concise usually compresses better than either EWAH or WAH, we see that it can also have a slightly worse random access performance.

Table 6. Performance comparisons for random-access checks (membership queries). We report the number of CPU cycles used on average per queries.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	3.74	415	49.2	33.6	63.6	3260	19300	19100
CENSUSINC ^{sort}	3.70	547	46.7	49.7	67.4	947	3690	3800
CENSUS1881	2.36	174	38.2	11.6	17.4	8820	26700	30000
CENSUS1881 ^{sort}	2.51	159	36.6	11.0	18.3	561	1810	1880
WEATHER	3.54	564	53.0	27.4	76.8	13700	65200	69000
WEATHER ^{sort}	3.47	661	48.8	42.0	53.7	2270	7490	7330
WIKILEAKS	3.68	172	45.4	20.9	27.7	1030	2410	2350
WIKILEAKS ^{sort}	2.79	152	43.2	14.8	20.1	248	531	493

5.7. Intersections, Unions, Differences, and Symmetric Differences

Recall that our datasets are made of 200 sets each. We consider the 199 intersections, unions, differences, and symmetric differences between successive sets. Our functions leave the inputs unchanged, but they materialize the result as a new set and so the timings include the construction of a new data structure. After each computation, we check the cardinality of the result against a precomputed result—thus helping to prevent the compiler from optimizing the computation. See Table 7.

The relative rankings of the different implementations is similar, whether we consider intersections, unions, differences, or symmetric differences. Intersections run faster, whereas unions and symmetric differences take longer.

What is immediately apparent is that for the CENSUSINC, CENSUSINC^{sort}, and WEATHER datasets, an uncompressed bitset is faster. However, Roaring comes in second, having sometimes half the speed. However, Roaring uses $10\times$ less memory than an uncompressed bitset on CENSUSINC^{sort}. The performance of an uncompressed bitset is much worse on several other datasets such as CENSUS1881, CENSUS1881^{sort}, WIKILEAKS^{sort} and WIKILEAKS.

Excluding the uncompressed bitset, the fastest data structure is generally Roaring. On the CENSUS1881 dataset, Roaring is an order of magnitude faster than any other approach at computing intersections, and generally much faster.

In most instances, the hash set offers the worst performance. This may seem surprising as hash sets have good computational complexity. However, hash sets have also poor locality: i.e., nearby values are not necessarily located in nearby cache lines.

Table 7. Performance comparisons for intersections, unions, differences and symmetric differences of integer sets. We report the number of CPU cycles used, divided by the number of input values. For each dataset, the best result is in bold.

(a) Two-by-two intersections: given 200 sets, we compute 199 intersections and check the cardinality of each result.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.130	7.84	55.3	0.380	0.220	1.41	1.80	2.15
CENSUSINC ^{sort}	0.150	4.52	28.5	0.250	0.160	0.280	0.450	0.560
CENSUS1881	19.5	3.28	0.480	1.07	0.090	3.74	14.5	20.9
CENSUS1881 ^{sort}	33.0	3.96	1.41	0.830	0.140	0.500	1.62	1.94
WEATHER	0.420	7.36	35.5	0.590	0.380	2.64	3.58	4.53
WEATHER ^{sort}	0.330	4.36	15.1	0.160	0.080	0.190	0.340	0.410
WIKILEAKS	26.4	4.15	14.9	3.06	1.43	2.86	6.10	6.38
WIKILEAKS ^{sort}	19.6	4.15	26.5	1.92	0.580	0.730	1.28	1.43

(b) Two-by-two unions: given 200 sets, we compute 199 unions and check the cardinality of each result.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.130	7.69	210	0.350	0.270	2.02	2.11	2.44
CENSUSINC ^{sort}	0.140	5.45	173	0.270	0.300	0.560	0.650	0.760
CENSUS1881	13.6	6.10	265	3.68	1.13	28.2	29.3	33.1
CENSUS1881 ^{sort}	23.7	6.18	201	2.47	0.950	2.50	3.03	3.38
WEATHER	0.400	7.70	250	0.600	0.570	4.46	4.59	5.41
WEATHER ^{sort}	0.310	6.13	181	0.200	0.170	0.480	0.560	0.620
WIKILEAKS	20.6	6.95	229	5.74	3.87	9.23	10.9	11.3
WIKILEAKS ^{sort}	18.4	7.15	182	4.20	2.16	2.37	2.57	2.63

(c) Two-by-two differences: given 200 sets, we compute 199 differences and check the cardinality of each result.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.120	8.38	177	0.340	0.280	1.72	1.98	2.34
CENSUSINC ^{sort}	0.120	5.54	134	0.260	0.240	0.420	0.540	0.650
CENSUS1881	11.1	5.03	236	2.56	0.610	15.8	21.9	26.7
CENSUS1881 ^{sort}	19.4	5.76	171	2.94	0.570	1.46	2.31	2.59
WEATHER	0.390	8.32	248	0.550	0.440	3.60	4.28	5.10
WEATHER ^{sort}	0.290	5.78	149	0.210	0.150	0.340	0.450	0.510
WIKILEAKS	18.4	6.05	215	6.00	2.65	5.98	8.25	8.44
WIKILEAKS ^{sort}	15.2	5.74	161	4.43	1.27	1.59	1.93	1.99

(d) Two-by-two symmetric differences: given 200 sets, we compute 199 symmetric differences and check the cardinality.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.120	8.79	266	0.350	0.370	1.96	2.04	2.38
CENSUSINC ^{sort}	0.130	5.99	215	0.280	0.350	0.620	0.660	0.790
CENSUS1881	13.2	6.38	369	3.61	1.13	27.5	28.9	33.3
CENSUS1881 ^{sort}	22.5	6.41	243	2.56	0.950	2.47	2.97	3.38
WEATHER	0.400	8.72	359	0.590	0.760	4.37	4.50	5.41
WEATHER ^{sort}	0.310	6.23	237	0.220	0.220	0.510	0.560	0.630
WIKILEAKS	20.0	7.41	301	5.60	3.92	8.96	10.7	11.4
WIKILEAKS ^{sort}	17.7	6.67	228	4.24	1.95	2.33	2.51	2.66

5.8. Wide Unions

In applications, it is not uncommon that we need to compute the union of many sets at once. Thus we benchmark the computation of the union of all 200 sets within a dataset. Though various optimizations of this problem are possible, we proceed sequentially, taking the first set, computing the union with

Table 8. Given 200 sets, we compute a single union, we report the number of CPU cycles used, divided by the number of input values.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.090	43.4	98.4	0.160	0.050	0.500	1.83	2.25
CENSUSINC ^{sort}	0.100	37.2	60.2	0.190	0.110	0.160	0.470	0.590
CENSUS1881	9.85	542	1010	2.18	2.56	276	388	402
CENSUS1881 ^{sort}	16.2	619	353	4.58	3.26	172	198	225
WEATHER	0.350	94.1	237	0.280	0.160	2.35	4.92	6.02
WEATHER ^{sort}	0.250	69.1	82.3	0.210	0.040	0.320	0.580	0.690
WIKILEAKS	15.4	515	413	11.9	3.01	328	415	450
WIKILEAKS ^{sort}	14.5	496	274	8.92	1.94	116	133	149

the second set and so forth. This simple approach uses little memory and provides good performance. In earlier work, this sequential approach was compared to other strategies in some detail [19].

Though we leave the inputs unchanged, the bitset, BitMagic, and Roaring implementations can do some of the operations in-place. For example, we can copy the first bitset, and then compute the bitwise OR operation in-place with the following bitsets. Roaring has a convenience function to compute the union of several bitmaps at once (`roaring_bitmap_or_many`).

We present our result in Table 8. Roaring is generally faster, except for two cases:

- On CENSUSINC^{sort}, the uncompressed bitset is slightly faster (by about 20%).
- On CENSUS1881, BitMagic is slightly faster than Roaring (by about 10%).

In several instances, Roaring is several times faster than any other alternative.

5.9. Fast Counts

Sometimes, we do not wish the materialize the result of an intersection or of a union, we merely want to compute the cardinality of the result as fast as possible. For example, one might want to compute the Jaccard index (Tanimoto similarity) between two sets, $(|A \cap B| / |A \cup B|)$ or the cosine similarity $(|A \cap B| / \sqrt{|A||B|})$.

Given a Roaring data structure, we can compute the total cardinality efficiently. For all bitset and array containers, we can simply look up the cardinality, whereas a simple sum over the length of all runs can determine the cardinality of a run container. The same is true with other data structures such as BitMagic, the vector, and so forth.

With this in mind, we can reduce the problem of computing the resulting cardinalities of various operations to the problem of computing the size of the intersection:

- The size of the union is $|A| + |B| - |A \cap B|$.
- The size of the difference is $|A| - |A \cap B|$.
- The size of the symmetric difference is $|A| + |B| - 2|A \cap B|$.

We present the time required to compute the size of the intersection in Table 9a. Compared to Table 7a where we present the time necessary to compute the full intersection and measure its size, these new timings can be twice as small or better. Unsurprisingly, it is efficient to avoid materializing the intersection.

The results are otherwise somewhat similar: except for a few datasets, Roaring is best. However, we observe that BitMagic is much closer in performance to Roaring, even surpassing it slightly in one dataset, than in Table 7a. As reported in § 2.2, the BitMagic library includes optimizations specific to the computation of the cardinality.

In Table 9, we also present results for the computation of the sizes of the unions, differences and symmetric differences of integer sets. We can compare these timings with Table 7 where we compute the complete results.

5.10. Effect of our Optimizations

When compiling scalar C code, the compiler and its standard library use advanced (e.g., SIMD) instructions. However, we are interested in the effect of our specific SIMD optimizations (see § 3 and § 4). Within the CRoaring library, we can deactivate these optimizations at compile time, falling back on portable C code. The compiler still uses advanced instructions, but without our direct assistance. In Table 10, we present a comparison, using two datasets where the optimizations were especially helpful. In some cases, we double the speed, or better. In these datasets, the functions to compute the size (or count) of the intersection, union, difference and symmetric differences were all similarly helped. In other datasets, our optimizations had smaller benefits or no benefit at all (though they did no harm).

We find that these results stress the value of our optimizations for several reasons:

- As already stated, even when our optimizations are disabled, the compiler still relies on advanced (SIMD) instructions.
- We are attempting to optimize already efficient code: without our optimizations, CRoaring is already faster than most alternatives in our benchmarks.
- Our optimizations focus on only some specific cases (see § 4). The bulk of the code base remains the same when we deactivate our optimizations. For this reason, we should not expect them to help in all or even most benchmarks. For example, none of the operations involving run containers have received specific optimizations.

6. CONCLUSION

No single approach can be best in all cases, but Roaring offers good all-around performance in our tests as was already reported [17, 19]. Our advanced SIMD-based optimizations further improved already good speeds in several cases. Other similar data structures (like BitMagic) could make use of our optimizations.

We have identified two relative weaknesses of our implementation that should be the subject of future work:

- A limitation of our implementation is the lack of advanced optimizations for all operations between a run container and another container type, including another run container. We have determined, by profiling, that these operations are frequently a significant cost (see Appendix A), so they may offer an avenue for further performance gains. In particular, it may prove interesting to attempt to vectorize operations over run containers.
- When comparing BitMagic and Roaring, two relatively similar designs, we found that BitMagic often has better random access performance (membership queries), e.g., by up to about a factor of three. This can be explained by the more compact layout used by Roaring. Careful optimizations might be able to improve Roaring’s performance on these queries [60].

Our investigations have focused on recent x64 microarchitectures. Our CRoaring library can run on a wide range of processors. In the future, we might investigate performance issues on other important architectures such as ARM.

Several features of CRoaring will be reviewed in the future. For example, CRoaring supports copy-on-write at the container level, meaning that we can copy Roaring bitmaps, in part or in full, quickly. Given that we have determined that memory allocations and data copies can sometimes be a significant cost (see Appendix A), it seems that this might be helpful. However, the copy-on-write capability adds implementation complexity. Moreover, CRoaring is designed to allow memory-file mapping, thus potentially saving extraneous memory allocations when bitmaps are retrieved from external memory. We can potentially combine copy-on-write containers with memory-file mapping.

Bitmaps can do more than just intersections, differences, symmetric differences and unions, they can support advanced queries such as top- k [61]. Further work should expand on our experimental validation.

The BitMagic library is similar to CRoaring. However, one distinguishing feature of BitMagic is the ability to produce highly compressed serialized outputs for storage on disk. Future work could consider adding this aspect to CRoaring. Moreover, CRoaring could—like BitMagic—use tagged pointers and a customized memory allocator for reduced memory usage and better speed.

One possible application for CRoaring would be as a drop-in replacement for an existing Java library through the Java Native Interface (JNI). Future work could assess the potential gains and downsides from such a substitution (see Appendix C).

7. ACKNOWLEDGEMENTS

We thank the author of the BitMagic library (A. Kuznetsov) for reviewing our manuscript and helping us tune the BitMagic benchmark.

References

1. Graefe G. Database servers tailored to improve energy efficiency. *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, ACM, 2008; 24–28.
2. Grand A. Frame of Reference and Roaring Bitmaps. <https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps> [last checked April 2017] 2015.
3. Chambi S, Lemire D, Godin R, Boukhalfa K, Allen CR, Yang F. Optimizing Druid with Roaring bitmaps. *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS ’16*, ACM: New York, NY, USA, 2016; 77–86, doi:10.1145/2938503.2938515.
4. Apache. Kylin: Extreme OLAP engine for Big Data. <http://kylin.apache.org/> [last checked April 2017] 2017.
5. Harrington B, Rapoport R. Introducing Atlas: Netflix’s primary telemetry platform. <https://medium.com/netflix-techblog/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a> [last checked April 2017] 2014.
6. Naga PN. Real-time analytics at massive scale with Pinot. <https://engineering.linkedin.com/analytics/real-time-analytics-massive-scale-pinot> [last checked April 2017] 2017.
7. Germond C. Pilosa launches breakthrough open source software to dramatically accelerate data queries. <https://www.pilosa.com/press/pilosa-launches-breakthrough-open-source-software/> [last checked April 2017] 2017.
8. Wu C. VSTS now uses roaring bitmaps 2017. Private communication.
9. 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.
10. Interlandi M, Shah K, Tetali SD, Gulzar MA, Yoo S, Kim M, Millstein T, Condie T. Titian: Data provenance support in Spark. *Proc. VLDB Endow.* Nov 2015; **9**(3):216–227, doi:10.14778/2850583.2850595.
11. Wikimedia. Cirrussearch. GitHub Engineering Blog, <https://www.mediawiki.org/wiki/Help:CirrusSearch> [last checked April 2017] 2017.
12. Abuzaid F, Bradley JK, Liang FT, Feng A, Yang L, Zaharia M, Talwalkar AS. Yggdrasil: An optimized system for training deep decision trees at scale. *Advances in Neural Information Processing Systems*, 2016; 3810–3818.
13. Siddiqui T, Kim A, Lee J, Karahalios K, Parameswaran A. Effortless data exploration with Zenvisage: An expressive and interactive visual analytics system. *Proc. VLDB Endow.* Nov 2016; **10**(4):457–468, doi:10.14778/3025111.3025126.
14. van Cranenburgh A, Scha R, Bod R. Data-oriented parsing with discontinuous constituents and function tags. *Journal of Language Modelling* 2016; **4**(1):57–11, doi:10.15398/jlm.v4i1.100.
15. Fokou G, Jean S, Hadjali A, Baron M. Cooperative techniques for SPARQL query relaxation in RDF databases. *Proceedings of the 12th European Semantic Web Conference on The Semantic Web. Latest Advances and New Domains - Volume 9088*, Springer-Verlag New York, Inc.: New York, NY, USA, 2015; 237–252, doi:10.1007/978-3-319-18818-8_15.
16. Krogh B, Jensen CS, Torp K. Efficient in-memory indexing of network-constrained trajectories. *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS ’16*, ACM: New York, NY, USA, 2016; 17:1–17:10, doi:10.1145/2996913.2996972.
17. Jianguo Wang YP Chunbin Lin, Swanson S. An experimental study of bitmap compression vs. inverted list compression. *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, SIGMOD ’17*, ACM: New York, NY, USA, 2017.
18. Chambi S, Lemire D, Kaser O, Godin R. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* 2016; **46**(5):709–719, doi:10.1002/spe.2325.
19. Lemire D, Ssi-Yan-Kai G, Kaser O. Consistently faster and smaller compressed bitmaps with Roaring. *Software: Practice and Experience* 2016; **46**(11):1547–1569.
20. Chambi S, Lemire D, Godin R. Nouveaux modèles d’index bitmap compressés à 64 bits. *Revue des Nouvelles Technologies de l’Information* 2016; **RNTI-B-12**:1–16.
21. Grand A. LUCENE-5983: RoaringDocIdSet. <https://issues.apache.org/jira/browse/LUCENE-5983> [last checked April 2017] 2014.

22. Lemire D, Boytsov L, Kurz N. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience* 2016; **46**(6):723–749, doi:10.1002/spe.2326.
23. Schlegel B, Willhalm T, Lehner W. Fast sorted-set intersection using SIMD instructions. *Proceedings of the 2nd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, ADMS '11, 2011; 1–8.
24. Mula W, Kurz N, Lemire D. Faster population counts using AVX2 instructions. <https://arxiv.org/abs/1611.07612> [last checked April 2017].
25. Navarro G, Providel E. Fast, small, simple rank/select on bitmaps. *Experimental Algorithms, Lecture Notes in Computer Science*, vol. 7276, Klasing R (ed.). Springer Berlin Heidelberg, 2012; 295–306, doi:10.1007/978-3-642-30850-5_26.
26. Antoshenkov G. Byte-aligned bitmap compression. *DCC'95*, IEEE Computer Society: Washington, DC, USA, 1995; 476.
27. Wu K, Stockinger K, Shoshani A. Breaking the curse of cardinality on bitmap indexes. *SSDBM'08*, Springer: Berlin, Heidelberg, 2008; 348–365.
28. 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.
29. Lemire D, Kaser O, Aouiche K. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering* 2010; **69**(1):3–28, doi:10.1016/j.datap.2009.08.006.
30. Martí V. Counting objects. GitHub Engineering Blog, <http://githubengineering.com/counting-objects/> [last checked April 2017] 2015.
31. 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.
32. Corrales F, Chiu D, Sawin J. Variable length compression for bitmap indices. *DEXA'11*, Springer-Verlag: Berlin, Heidelberg, 2011; 381–395.
33. Guzun G, Canahuate G, Chiu D, Sawin J. A tunable compression framework for bitmap indices. *ICDE'14*, IEEE, 2014; 484–495.
34. Chang J, Chen Z, Zheng W, Cao J, Wen Y, Peng G, Huang WL. SPLWAH: A bitmap index compression scheme for searching in archival internet traffic. *2015 IEEE International Conference on Communications (ICC)*, 2015; 7089–7094, doi:10.1109/ICC.2015.7249457.
35. Wu Y, Chen Z, Cao J, Li H, Li C, Wang Y, Zheng W, Chang J, Zhou J, Hu Z, et al.. CAMP: A new bitmap index for data retrieval in traffic archival. *IEEE Communications Letters* June 2016; **20**(6):1128–1131, doi:10.1109/LCOMM.2016.2551719.
36. Wu Y, Chen Z, Wen Y, Zheng W, Cao J. COMBAT: a new bitmap index coding algorithm for big data. *Tsinghua Science and Technology* April 2016; **21**(2):136–145, doi:10.1109/TST.2016.7442497.
37. Guzun G, Canahuate G. Hybrid query optimization for hard-to-compress bit-vectors. *The VLDB Journal* 2016; **25**(3):339–354, doi:10.1007/s00778-015-0419-9. URL <http://dx.doi.org/10.1007/s00778-015-0419-9>.
38. O’Neil E, O’Neil P, Wu K. Bitmap index design choices and their performance implications. *Proceedings of the 11th International Database Engineering and Applications Symposium*, IDEAS ’07, IEEE Computer Society: Washington, DC, USA, 2007; 72–84, doi:10.1109/IDEAS.2007.19.
39. 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.
40. Culpepper JS, Moffat A. Efficient set intersection for inverted indexing. *ACM Transactions Information Systems* Dec 2010; **29**(1):1:1–1:25, doi:10.1145/1877766.1877767.
41. Kuznetsov A. The BitMagic C++ library. <https://github.com/tlk00/BitMagic> [last checked August 2017] 2016.
42. Bentley JL, Yao ACC. An almost optimal algorithm for unbounded searching. *Information Processing Letters* 1976; **5**(3):82–87.
43. Wikipedia. Bit Manipulation Instruction Sets. https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets [last checked April 2017] 2017.
44. Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Technical Report*, Copenhagen University College of Engineering 2016. <http://www.agner.org/optimize/instruction.tables.pdf> [last checked April 2017].
45. Flynn MJ. Some computer organizations and their effectiveness. *IEEE Transactions on Computers* Sept 1972; **C-21**(9):948–960, doi:10.1109/TC.1972.5009071.
46. Koblenz G, Ishizaki K, Inoue H. Bringing Apache Spark Closer to SIMD and GPU. <http://www.spark.tc/simd-and-gpu/> [last checked August 2017] 2017.
47. Warren HS Jr. The quest for an accelerated population count. *Beautiful Code: Leading Programmers Explain How They Think*, Wilson G, Oram A (eds.). chap. 10, O'Reilly Media: Sebastopol, California, 2007; 147–160.
48. Knuth DE. *Combinatorial Algorithms, Part 1, The Art of Computer Programming*, vol. 4A. Addison-Wesley: Boston, Massachusetts, 2011.
49. 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.
50. Intel Corporation. Intel IACA tool: A Static Code Analyser. <https://software.intel.com/en-us/articles/intel-architecture-code-analyzer> [last checked April 2017] 2012.
51. Inoue H, Taura K. SIMD- and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment* Jul 2015; **8**(11):1274–1285, doi:10.14778/2809974.2809988.
52. Batcher KE. Sorting networks and their applications. *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, ACM: New York, NY, USA, 1968; 307–314, doi:10.1145/1468075.1468121.
53. Knuth DE. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc.: Redwood City, CA, USA, 1998.

54. Hipp R. The SQLite amalgamation. <https://www.sqlite.org/amalgamation.html> [last checked April 2017] 2015.
55. Sanfilippo S. Redis Modules. <https://redis.io/modules> [last checked August 2017] 2017.
56. Paoloni G. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, Santa Clara, CA 2010.
57. Lemire D, Kaser O, Gutierrez 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.
58. Damme P, Habich D, Hildebrandt J, Lehner W. Insights into the comparative evaluation of lightweight data compression algorithms. *Proceedings of the 20th International Conference on Extending Database Technology*, EDBT '17, OpenProceedings: Konstanz, Germany, 2017.
59. Pieterse V, Kourie DG, Cleophas L, Watson BW. Performance of C++ bit-vector implementations. *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, ACM: New York, NY, USA, 2010; 242–250, doi:10.1145/1899503.1899530.
60. Khuong PV, Morin P. Array layouts for comparison-based searching. <https://arxiv.org/abs/1509.05053> [last checked April 2017].
61. Kaser O, Lemire D. Compressed bitmap indexes: beyond unions and intersections. *Software: Practice and Experience* 2016; **46**(2):167–198, doi:10.1002/spe.2289.
62. Anh VN, Moffat A. Index compression using 64-bit words. *Software: Practice and Experience* 2010; **40**(2):131–147, doi:10.1002/spe.v40:2.

Table 9. Performance comparisons for computing the sizes of the intersections, unions, differences and symmetric differences of integer sets. We report the number of CPU cycles used, divided by the number of input values. For each dataset, the best result is in bold.

(a) Two-by-two intersections: given 200 sets, we compute the sizes of 199 intersections, without materializing the intersections.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.090	4.85	13.3	0.140	0.130	0.800	0.980	1.25
CENSUSINC ^{sort}	0.090	2.08	6.76	0.120	0.110	0.190	0.260	0.340
CENSUS1881	6.31	1.45	0.460	0.090	0.080	3.71	9.04	12.1
CENSUS1881 ^{sort}	11.4	1.70	1.05	0.120	0.060	0.440	1.02	1.27
WEATHER	0.270	4.54	17.5	0.210	0.260	1.64	2.09	2.75
WEATHER ^{sort}	0.190	1.97	7.19	0.090	0.060	0.150	0.220	0.260
WIKILEAKS	11.7	2.12	12.8	1.57	0.870	2.46	3.85	4.11
WIKILEAKS ^{sort}	8.71	1.69	10.4	0.510	0.260	0.540	0.760	0.830

(b) Two-by-two unions: given 200 sets, we compute the sizes of 199 unions, without materializing the unions.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.070	4.62	13.4	0.370	0.130	1.11	0.990	1.32
CENSUSINC ^{sort}	0.070	1.87	6.76	0.200	0.120	0.290	0.280	0.380
CENSUS1881	7.62	1.27	0.220	1.18	0.100	12.1	11.6	16.8
CENSUS1881 ^{sort}	12.6	1.57	0.840	0.520	0.140	1.06	1.28	1.65
WEATHER	0.230	4.36	17.6	0.570	0.270	2.50	2.10	2.92
WEATHER ^{sort}	0.180	1.75	7.20	0.140	0.060	0.230	0.240	0.300
WIKILEAKS	11.8	2.37	10.6	2.22	1.33	4.08	4.46	4.89
WIKILEAKS ^{sort}	10.5	1.57	9.40	1.16	0.490	1.00	1.02	1.16

(c) Two-by-two differences: given 200 sets, we compute the sizes of 199 differences, without materializing the differences.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.090	5.21	47.4	0.280	0.130	0.950	1.00	1.26
CENSUSINC ^{sort}	0.100	2.28	29.5	0.190	0.120	0.230	0.280	0.350
CENSUS1881	8.26	1.63	44.6	1.60	0.080	8.14	10.8	14.5
CENSUS1881 ^{sort}	14.2	1.99	21.5	3.19	0.090	0.720	1.04	1.28
WEATHER	0.280	5.11	61.4	0.450	0.270	2.05	2.14	2.77
WEATHER ^{sort}	0.210	2.19	32.3	0.170	0.060	0.190	0.230	0.270
WIKILEAKS	13.5	2.53	40.3	5.27	1.09	3.28	4.15	4.29
WIKILEAKS ^{sort}	11.0	1.88	25.8	3.64	0.370	0.790	0.950	1.03

(d) Two-by-two symmetric differences: given 200 sets, we compute the sizes of 199 symmetric differences, without materializing the symmetric differences.

	bitset	vector	hash set	BitMagic	Roaring	EWAH	WAH	Concise
CENSUSINC	0.090	4.80	13.3	0.390	0.130	1.13	0.980	1.30
CENSUSINC ^{sort}	0.100	2.10	6.77	0.220	0.120	0.310	0.280	0.380
CENSUS1881	9.79	1.46	0.470	1.11	0.080	12.0	11.6	16.9
CENSUS1881 ^{sort}	16.3	1.78	0.800	0.520	0.130	1.07	1.27	1.66
WEATHER	0.290	4.51	17.6	0.600	0.270	2.52	2.09	2.90
WEATHER ^{sort}	0.220	1.97	7.19	0.140	0.060	0.240	0.240	0.300
WIKILEAKS	14.8	2.23	11.3	2.21	1.29	4.09	4.47	4.93
WIKILEAKS ^{sort}	12.8	1.70	8.77	1.14	0.470	0.990	1.02	1.16

Table 10. CPU cycles used per input element before and after our SIMD-based optimizations (§ 4)

		CENSUSINC	WEATHER
2-by-2 intersections	scalar code	0.450	0.990
	SIMD code	0.220	0.380
	scalar/SIMD	2.0	2.6
2-by-2 unions	scalar code	0.410	0.880
	SIMD code	0.270	0.570
	scalar/SIMD	1.5	1.5
2-by-2 difference	scalar code	0.520	1.10
	SIMD code	0.280	0.440
	scalar/SIMD	1.9	2.5
2-by-2 sym. diff.	scalar code	0.540	1.06
	SIMD code	0.370	0.760
	scalar/SIMD	1.5	1.4
2-by-2 intersection counts	scalar code	0.360	0.890
	SIMD code	0.130	0.260
	scalar/SIMD	2.8	3.4
wide union	scalar code	0.060	0.250
	SIMD code	0.050	0.160
	scalar/SIMD	1.2	1.6

A. PROFILING OF ROARING_BITMAP COMPUTATIONS

As is typical, given a dataset and an operation, the software spends most of its time in a few functions spanning few lines of code. Using Linux’s `perf` tool, we can identify these hot functions using our test platform and datasets (see § 5). As part of our publicly available benchmark package, we include a script to automatically run a profiling benchmark. A detailed analysis would be difficult, as there are too many details to take into account. However, we can at least identify the most expensive operation. We also report the percentage of the total time spent in the corresponding operation. This percentage should be viewed as a rough approximation. When the percentage is relatively low (e.g., 20%), then it is an indication that there is no clear bottleneck and that many functions contribute to the total running time. Though we collected detailed data on all operations and all datasets, we only present the results for some key operations, for simplicity (see Table 11). The various benchmarks and datasets are presented in our experimental section (§ 5).

In several instances, the same operation is identified as most significant both with regular code and with optimizations (as per § 3 and § 4), but the percentage is lower with optimizations. This reflects the fact that the optimizations were successful at reducing the running time. In cases where the function does not benefit from optimizations, the percentage is mostly unchanged, as expected. It is the case when run containers are involved, as operations over these containers have not benefited from particular optimizations.

In some instances, e.g., for the union between bitmaps in the `CENSUS1881` data set, the `memcpy` function as well as accompanying memory allocations become a significant cost because we clone many containers. In others cases (e.g., `CENSUS1881sort`, `WIKILEAKSsort`), memory allocation is significant.

When intersecting bitmaps in the `CENSUS1881sort` dataset, much of the time is spent accessing and comparing the high-level 16-bit keys, without access to the containers.

B. LARGE IN-MEMORY DATASET

To validate our results on a large dataset, we generated 100 arrays of 10 million distinct integers in the range $[0, 10^9]$ using the `ClusterData` distribution from Anh and Moffat [62]. This distribution leaves relatively small gaps between successive integers, with occasional large gaps. As with our main experiments, all our source code and scripts are available online. We ran these experiments using the same hardware and software as in § 5—using a separate script that is included with our benchmarking software.

In total, we generated one billion integers; thus our benchmarks could use gigabytes of memory (RAM). We excluded the hash-set implementation (`std::unordered_set`) from these tests because it failed to complete all of them without exceeding the memory available on our benchmarking machine.

We present our comparison in Table 12. Compared with our main experiments (on smaller datasets), the most obvious difference is that the membership queries are significantly worse for the run-length encoded schemes (EWAH, WAH and Concise), which is not surprising: that membership queries are resolved by scanning the data from the beginning of the set each time in these cases. To a lesser extent, Roaring also suffers from a worse performance for membership queries when compared to the bitset or to BitMagic.

We compare Roaring without our manual SIMD-based optimizations with the scalar code in Table 13. Our SIMD-based optimizations are worthwhile, sometimes multiplying the performance by a factor of five.

C. JAVA VERSUS C

In earlier work, we used Java to benchmark Roaring against several other set data structures, including 32-bit EWAH [19]. There are many differences between Java and C that make comparisons difficult

Table 11. Most expensive operations for each benchmark and dataset as determined by software profiling

2-by-2 intersection	regular code	with optimizations
CENSUSINC	array-array (56%)	array-array (19%)
CENSUSINC ^{sort}	array-run (42%)	array-run (45%)
CENSUS1881	galloping array-array (38%)	galloping array-array (39%)
CENSUS1881 ^{sort}	high-level (26%)	high-level (26%)
WEATHER	array-array (72%)	array-array (35%)
WEATHER ^{sort}	array-run (30%)	array-run (31%)
WIKILEAKS	run-run (38%)	run-run (38%)
WIKILEAKS ^{sort}	run-run (19%)	run-run (20%)
2-by-2 union	regular code	with optimizations
CENSUSINC	array-array (47%)	array-array (20%)
CENSUSINC ^{sort}	array-run (44%)	array-run (46%)
CENSUS1881	memcpy (20%)	memcpy (19%)
CENSUS1881 ^{sort}	memory allocation (20%)	memory allocation (20%)
WEATHER	array-array (49%)	bitset→array conversion (23%)
WEATHER ^{sort}	array-run (26%)	array-run (25%)
WIKILEAKS	run-run (20%)	run-run (20%)
WIKILEAKS ^{sort}	memory allocation (15%)	memory allocation (13%)
2-by-2 diff.	regular code	with optimizations
CENSUSINC	array-array (52%)	bitset→array conversion (18%)
CENSUSINC ^{sort}	array-run (25%)	array-run (27%)
CENSUS1881	array-array (19%)	memcpy (18%)
CENSUS1881 ^{sort}	memory allocation (20%)	memory allocation (20%)
WEATHER	array-array (26%)	array-array (25%)
WEATHER ^{sort}	run-run (22%)	run-run (22%)
WIKILEAKS	run-run (24%)	run-run (22%)
WIKILEAKS ^{sort}	memory allocation (14%)	memory allocation (16%)
2-by-2 sym. diff.	regular code	with optimizations
CENSUSINC	array-array (52%)	bitset-array (24%)
CENSUSINC ^{sort}	array-array (21%)	run-run (18%)
CENSUS1881	memcpy (21%)	memcpy (21%)
CENSUS1881 ^{sort}	memory allocation (20%)	memory allocation (20%)
WEATHER	array-array (72%)	array-array (40%)
WEATHER ^{sort}	run-run (22%)	run-run (21%)
WIKILEAKS	run-run (14%)	run-run (14%)
WIKILEAKS ^{sort}	memory allocation (15%)	memory allocation (15%)
single-union	regular code	with optimizations
CENSUSINC	bitset-array (56%)	bitset-array (52%)
CENSUSINC ^{sort}	run-bitset (62%)	run-bitset (70%)
CENSUS1881	bitset-array (81%)	bitset-array (36%)
CENSUS1881 ^{sort}	bitset→array conversion (61%)	bitset→array conversion (63%)
WEATHER	bitset-array (80%)	bitset-array (78%)
WEATHER ^{sort}	run-bitset (66%)	run-bitset (68%)
WIKILEAKS	run-bitset (64%)	run-bitset (66%)
WIKILEAKS ^{sort}	run-bitset (25%), bitset→array (25%)	run-bitset (26%), bitset→array (26%)
intersection-count	regular code	with optimizations
CENSUSINC	array-array (71%)	array-array (29%)
CENSUSINC ^{sort}	array-run (55%)	array-run (62%)
CENSUS1881	galloping array-array (54%)	galloping array-array (62%)
CENSUS1881 ^{sort}	high-level (46%)	high-level (47%)
WEATHER	array-array (81%)	array-array (45%)
WEATHER ^{sort}	run-run (42%)	run-run (46%)
WIKILEAKS	run-run (66%)	run-run (66%)
WIKILEAKS ^{sort}	run-run (47%)	run-run (52%)

Table 12. Memory usage in bits per value as well as timings in cycles per value for various operations for the large ClusterData in-memory dataset. For each metric, the best reported result is in bold. For the vector, sequential access is practically free.

	bitset	vector	BitMagic	Roaring	EWAH	WAH	Concise
bits/value	100	32.0	38.5	13.5	33.4	33.7	22.6
seq. access	28.9	—	46.3	6.87	25.5	27.3	27.6
membership	4.92	1910	82.0	381	19 900 000	58 600 000	48 000 000
intersections	25.4	11.2	9.11	2.58	18.0	24.5	27.8
unions	25.4	14.5	9.77	4.97	33.5	34.9	37.5
differences	25.3	14.4	8.05	2.48	26.2	31.4	33.0
sym. differences	25.3	16.6	9.91	4.09	32.6	34.8	37.8
wide unions	5.01	457	3.92	3.87	53.0	68.7	69.5
intersection counts	2.27	8.23	5.16	1.37	14.0	16.2	18.9
union counts	2.14	8.92	8.89	1.35	17.7	16.5	19.3
diff. counts	2.28	9.27	7.24	1.37	16.5	16.8	18.9
sym. diff. counts	2.27	8.29	9.17	1.35	17.8	16.6	19.2

Table 13. CPU cycles used per input element before and after our SIMD-based optimizations (§ 4)

ClusterData		
2-by-2 intersections	scalar code	8.11
	SIMD code	2.58
	scalar/SIMD	3.1
2-by-2 unions	scalar code	9.34
	SIMD code	4.97
	scalar/SIMD	1.9
2-by-2 difference	scalar code	8.33
	SIMD code	2.48
	scalar/SIMD	3.4
2-by-2 sym. diff.	scalar code	9.19
	SIMD code	4.09
	scalar/SIMD	2.2
2-by-2 intersection counts	scalar code	6.82
	SIMD code	1.37
	scalar/SIMD	5.0
wide union	scalar code	4.79
	SIMD code	3.87
	scalar/SIMD	1.2

with our current results in C. For example, Java uses a garbage collector whereas C does not. Nevertheless, we can run the SIMD-optimized C and Java versions side-by-side on the same machine and report the results. We make our dual-language benchmark freely available to ensure reproducibility (https://github.com/lemire/java_vs_c_bitmap_benchmark). For simplicity we focus on a single dataset (CENSUS1881). On our testing hardware (§ 5.2), we use Oracle’s JDK 1.8.0_101 for Linux (x64), the Java EWAH library (version 1.0.6) and the RoaringBitmap library (version 0.6.51). We use Oracle’s JMH benchmarking system to record the average wall-clock time in Java after many warm-up trials. Our configuration differs in many ways from our earlier work, so results are not directly comparable [19].

Because the Java and C benchmarks are written using entirely different software, and because the Java timings rely on an extensive framework (JMH), we cannot be certain that the Java-to-C comparison is entirely fair, especially for small timings. For example, the Java benchmark may not account for the memory allocation in the same manner as the C timings. To help improve fairness, we added a few “warm-up” passes in C/C++, just like the warm-up trials generated in Java by JMH:

Table 14. Performance comparisons for intersections between Java and C/C++ implementations. We report timings in microseconds for executing the whole benchmark (e.g., compute all pairwise intersections). A ratio greater than 1 indicates that Roaring is faster.

	C/C++			Java			Java/C ratio	
	EWAH	Roaring	ratio	EWAH	Roaring	ratio	EWAH	Roaring
2-by-2 intersections	1900	42	45	2480	43	58	1.3	1.0
2-by-2 unions	12 800	605	21	28 500	750	38	2.2	1.2
wide union	77 600	677	114	107 000	1240	86	1.4	1.8

it helped performance slightly. The small beneficial effect is probably related to caching and branch prediction.

We report timings (in microseconds) and as well give the speed ratio between 32-bit EWAH and Roaring in Table 14. The relative benefits of Roaring over EWAH in these tests are sometimes more important in C/C++ and sometimes more important in Java. The Java code is slower by up to a factor of two, but the difference between Java and C can also be negligible. The performance differences between Roaring and EWAH in these tests dominate the differences that are due to the programming languages and implementations. That is, the Java version of Roaring is several times faster than the C++ version of EWAH.