CONCISE: **Co**mpressed '**n**' **C**omposable **I**nteger **Se**t

Alessandro Colantonio[*,a], Roberto Di Pietro[a]

[a] *Università di Roma Tre, Dipartimento di Matematica, Roma, Italy*

Abstract

Bit arrays, or *bitmaps*, are used to significantly speed up set operations in several areas, such as data warehousing, information retrieval, and data mining, to cite a few. However, bitmaps usually use a large storage space, thus requiring compression. Consequently, there is a space-time tradeoff among compression schemes. The *Word Aligned Hybrid* (WAH) bitmap compression trades some space to allow for bitwise operations without first decompressing bitmaps. WAH has been recognized as the most efficient scheme in terms of computation time. In this paper we present CONCISE (**Co***mpressed* '**n**' **C***omposable* **I***nteger* **Se***t*), a new scheme that enjoys significantly better performances than WAH. In particular, when compared to WAH our algorithm is able to reduce the required memory up to 50%, while having comparable computation time. Further, we show that CONCISE can be efficiently used to represent sets of integral numbers in lieu of well-known data structures such as arrays, lists, hashtables, and self-balancing binary search trees. Extensive experiments over synthetic data show the effectiveness of our proposal.

Key words: bitmap compression, data structures, performance

1. Introduction

The term *bit array* or *bitmap* usually refers to an array data structure which stores individual bits. The main reason for adopting bitmaps is represented by their effectiveness at exploiting bit-level parallelism in hardware to speed-up operations. A typical bit array stores $k \times w$ bits where w is the *word* size, that is the number of bits that the given CPU is able to manipulate via bitwise instructions (typically 32 or 64 in modern architectures), and k is some non-negative integer. Bitmaps made up of n bits can be used to implement a simple data structure for the storage of any subset of the integers $\{1, 2, \ldots, n\}$. Specifically, if the i-th bit of a bitmap is "1" then the integer i is within the integer set, whereas a "0" bit indicates that the integer i is not in the set. This set data structure requires $\lceil n/w \rceil$ words of space—padding with zeros is a usual choice. The *least significant bit* (the "rightmost" in the typical bitmap representation) usually indicates the smallest-index number—this convention is adopted throughout this paper as well. Hence, if we want to compute the intersection/union over integer sets represented by bitmaps of length n, we require $\lceil n/w \rceil$ bitwise AND/OR operations each.

Because of their property of leveraging bit-level parallelism, bitmaps often outperform many other data structures (e.g., self-balancing binary search trees, hash tables, or simple arrays or linked lists of the entries) on practical data sets. Classical compression algorithms introduce a computation overhead that may limit the benefits of using bitmaps. For example, well-known algorithms such as DEFLATE [1] effectively reduce the memory footprint, but performing set operations requires that data be decompressed, which drastically increases the computation time [2]. That is why compression schemes that allow for bitwise operations without first decompressing bitmaps are usually preferred, at the cost of having a higher memory footprint than other schemes. In this scenario, the *Word Aligned Hybrid* (WAH) bitmap compression algorithm is currently recognized as the most efficient one, mainly from a computational perspective [2]. It was first proposed to compress bitmap indices of DBMS, but subsequent applications include visual analytics [3] and data mining [4], to cite a few. It uses a *run-length* encoding, where long sequences of 0's or 1's (*runs*) require a reduced number of bits by only storing the length of the sequences in place of the corresponding bit strings. WAH allows for set operations to be efficiently performed over compressed bitmaps. This property is guaranteed by the alignment with the machine word size. Figure 1 graphically explains what "alignment" means. Without loss of generality, suppose that words are made up of 32 bits. First, the bitmap to compress is logically partitioned into blocks of 31 bits, namely the word size minus one. In turn, sequences of consecutive 31-bit blocks containing all 0's or all 1's are identified. The compressed form is created as follows: if a 31-bit block contains both 0's and 1's, it is stored in a 32-bit word referred to as *literal* word, where the leftmost bit is set to 1. Otherwise, sequence of homogeneous 31-bit blocks of 0's or 1's are stored in a single 32-bit word referred to as *fill* word, where the first (leftmost) bit is 0, the second bit indicates the fill type (all 0's or all 1's) and the remaining 30 bits are used to store the number of 31-bit blocks.[1]

[*] Corresponding author

Email addresses: colanton@mat.uniroma3.it (Alessandro Colantonio), dipietro@mat.uniroma3.it (Roberto Di Pietro)

URL: http://ricerca.mat.uniroma3.it/users/colanton/ (Alessandro Colantonio), http://ricerca.mat.uniroma3.it/users/dipietro/ (Roberto Di Pietro)

[1] In the paper of Wu et al. [2], the most significant bit is complemented with

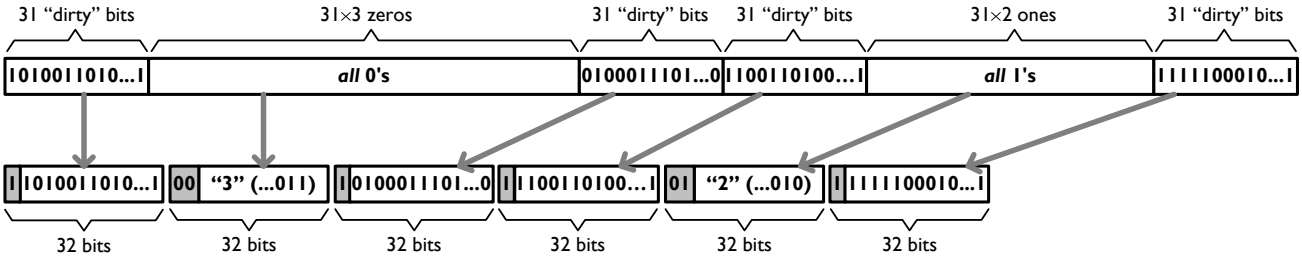


Figure 1: Word-aligned run-length encoding

Bitwise operations are fast because performing AND/OR over compressed bitmaps corresponds to performing AND/OR over 32-bit literals (with just one CPU instruction), while sequences can easily be managed due to the same block granularity of literals.

This paper proposes a new compression scheme, CONCISE (**Co***mpressed* '**n**' **C***omposable* **I***nteger* **Se***t*), that outperforms WAH by reducing the size of the compressed bitmaps up to 50%, without affecting the performance of bitwise operations. CONCISE is based on the observation that very sparse bitmaps (that is, when there are few set bits followed by long sequence of unset bits) can be further compressed compared to the WAH approach. Specifically, if n is the number of bits that equals 1 in a sparse uncompressed bitmap, a WAH-based compressed bitmap requires about $2n$ words [2, 5]. To achieve a better compression ratio on sparse bitmaps, CONCISE introduces the concept of *mixed* fill word. In particular, we allow to store the sequence length and the literal word made up of only one set bit within a single word of 32 bits. This reduces the worst case memory footprint to n words. Since n words is the minimum amount of memory required to represent n integral values with uncompressed data structures such as arrays, lists, hashtables, and self-balancing binary search trees, CONCISE is always more efficient than such data structures in terms of memory footprint. As for computation time, we show that it also outperforms efficient data structures on set operations.

The remainder of the paper is organized as follows. The following section offers a detailed description of the CONCISE algorithm. The benefits of the proposed algorithm are then experimentally demonstrated in Section 3. Finally, Section 4 provides concluding remarks.

2. CONCISE Algorithm

Figure 2 shows an example of CONCISE-compressed bitmap made up of 5 words. Words #0, #3, and #5 are *literal* words where, similarly to WAH, the leftmost bit indicates the block type ('1'), while the remaining bits are used to represent an uncompressed 31-bit block. Words #1, #2, and #4 are *fill* words: the first (leftmost) bit is the block type ('0'), the second bit is the fill type (a sequence of 0's or 1's), the following 5 bits are the *position* of a "flipped" bit within the first 31-bit block of the fill, and the remaining 25 bits count the number of 31-blocks that compose the fill minus one. When position bits equal 0 (binary '00000'), the word is a "pure" fill, similar to that of WAH. Otherwise, position bits indicate the bit to switch (from 0 to 1 in a sequence of 0's, or from 1 to 0 in a sequence of 1's) within the first 31-bit block of the sequence represented by the fill word. That is, 1 (binary '00001') indicates that we have to flip the rightmost bit, while 31 (binary '11111') indicates that we have to flip the leftmost one. If we consider bitmaps as a representation of integer sets, in Figure 2 Words #2 indicates that integers in the range 94–1022 are missing, but 93 is in the set since position bits say that the first number of the "missing numbers" sequence is an exception. Note that mixed fill words are different from the "headers" adopted by other byte/word-aligned compression schemes such as WBC [2] and BBC [6]. Indeed, such schemes are based on one or more kinds of control *atoms* (i.e., one word in WBC, or one byte in BBC) that specify how to decode subsequent words. These atoms require additional space to store them (especially for WBC), or additional time to analyze them (especially for BBC), as detailed in [2]. Unlike WBC or BBC, CONCISE does not use any header words. CONCISE has only two kinds of words, and mixed fill words have a fixed amount of bits (differently from BBC, which allows for a variable number of bytes) to identify the sequence length. As we will show in next section, the most important consequence is that decoding CONCISE-coded words requires almost the same computation time as decoding WAH-coded words. Concurrently with our research and independently, another improvement of WAH was presented in [7]. In particular, the authors introduced PLWAH, a word-aligned scheme that compress "nearly identical" words: a sequence of words with either only few or almost all set bits, are coded into a single fill word. According to [7], PLWAH reduces up to 50% the memory footprint if compared to WAH. However, PLWAH needs additional time to encode nearly identical words—the less identical they are, the more compression time is required. Instead, as we will show later on, CONCISE requires half the space of WAH while having almost the same computational complexity.

This approach greatly improves the compression ratio in the worst case. Indeed, instead of having $2n$ words as in the WAH-compressed form of n sparse integers, we only require n words in the CONCISE-compressed form for the same integer set. This way, CONCISE bitmaps always require less amount of memory

respect to the example of Figure 1, that is literals start with 0 and fills start with 1. Though this does not change the semantic of the approach, we use the configuration of Figure 1 since it reflects the proposed implementation of CONCISE.

Figure 2: Compressed representation of the set {3, 5, 31–93, 1024, 1028, 1 040 187 422}. The word #0 is used to represent integers in the range 0–30, word #1 for integers in 31–92, word #2 for integers 93–1022, word #3 for integers 1023–1053, word #4 for integers 1054–1 040 187 391, and word #5 for integers 1 040 187 392–1 040 187 422.

than WAH bitmaps. Since we have 25 bits for representing the length (minus one) of sequences of homogeneous bits, the maximum representable integer is $31\times2^{25}+30 = 1\,040\,187\,422$, that is almost half of the positive integers that can be represented in a two's complement representation over 32 bits.

Due to space limitation, Figure 3 illustrates with pseudocode the main parts of the proposed implementation of the CONCISE scheme only. The complete source code for CONCISE is available on *SourceForge* since January 2010.[2] In the figure, the following notation has been adopted:

- "|" indicates the bitwise OR operator, "&" the bitwise AND, "~" the bitwise NOT, "≪" the left-shift operator, and "≫" means the signed right-shift operator—namely, it replicates the leftmost bit.
- 80000000h is an instance of a 32-bit word expressed in hexadecimal notation. That is, 80000000h indicates a 32-bit word with the highest-order (leftmost) bit set to 1 and all other bits set to 0.

We also require additional operations that can be efficiently performed via the previous bit operations:

- BITCOUNT(n) counts the total number of set bits in the binary representation of the specified integral value n—also known as *population count* or *Hamming weight*. It is natively provided by some processors via a single instruction to calculate it (e.g., `POPCNT` of Intel SSE4 instruction set [8]). For processors lacking this feature, there are efficient branch-free algorithms which can compute the number of bits in a word using a series of simple bit operations [9].
- TRAILINGZEROS(n) counts the number of 0's following the lowest-order (rightmost) bit in the binary representation of the specified integral value.[3]
- CONTAINSONEBIT(n) checks whether the given number contains only one set bit, and it can be efficiently performed by verifying whether $n \,\&\, (n-1) = 0$.
- $n \times 31$ can be performed by doing $(n \ll 5) - n$. Several bit hacks exist to efficiently compute n mod 31.

3. Algorithm Analysis

We now report on the results of a comparative analysis among CONCISE, WAH, and some classical data structures to manage integer sets. The testbed was represented by a notebook equipped with an Intel Core™2 Duo CPU P8600 at 2.40 GHz and 3GB of RAM. To assure the portability among different platforms, all algorithms were coded in Java SE6. Since the original WAH algorithm was developed in a C++ environment, we also coded the WAH algorithm in Java according to the description provided in [2]. Regarding efficiency issues, note that the Java HotSpot™ Virtual Machine executes all frequently executed methods as machine-optimized code [10]. Optimization impairs performance only during the JVM warm-up phase. Hence, to trigger the JVM optimization mechanisms we repeated all the tests several times, excluding the warm-up phase in our analysis. Please also note that Java natively supports all the bitwise operations required for an efficient implementation of CONCISE and WAH, such as AND, OR, NOT, and shift. As for other data structures, we used the implementation provided by the Java package `java.util`. Pseudo-random numbers were generated through the algorithm described in [11] due to its good, provable uniform distribution, and very large period.

Table 1 reports some characteristic about the memory footprint of the data structure under analysis. For each data structure, we report the number of bytes required to store each integral number, whether the data structure allows for duplicate elements, and whether the items are sorted. CONCISE is the more efficient data structure in terms of memory occupation. In fact, classical data structures incur an additional space overhead for pointers, while WAH requires twice the memory of CONCISE when both algorithms are able to compress bitmaps—that is, in presence of sparse datasets.

Figure 4 reports on experimental time-space analysis results. In our experiments, we generated sets of 10^5 integers

[2] More details about the actual implementation of CONCISE, as well as the code used for the comparative analysis, can be found at `http://sourceforge.net/projects/concise`.

[3] For more details about bit hacks, see, for example, `http://graphics.stanford.edu/~seander/bithacks.html`.

```
1: procedure APPEND(words[·], top, max, i)
2:     if words[·] is empty then {first append}
3:         f ← ⌊i/31⌋
4:         if f = 0 then
5:             top ← 0
6:         else if f = 1 then
7:             top ← 1
8:             words[0] ← 80000000h {literal}
9:         else
10:            top ← 1
11:            words[0] ← f − 1 {fill}
12:        end if
13:        words[top] ← 80000000h | (1 ≪ (i mod 31))
14:    else
15:        b ← i − max + (max mod 31) {next bit to set}
16:        if b ≥ 31 then {zeros are required before}
17:            b ← b mod 31,   f ← ⌊b/31⌋
18:            if f > 0 then
19:                APPENDSEQUENCE(f, 00000000h)
20:            end if
21:            APPENDLITERAL(80000000h | (1 ≪ b))
22:        else
23:            words[top] ← words[top] | (1 ≪ b)
24:            if words[top] = FFFFFFFFh then
25:                top ← top − 1
26:                APPENDLITERAL(FFFFFFFFh)
27:            end if
28:        end if
29:    end if
30:    max ← i
31:    return words[·], top, max
32: end procedure
```

(a) Append of an integer i greater than the maximal appended integer max. It checks whether the bit to set is within the last literal, or if a sequence of 0's is needed before. top is the index of the last word in words.

```
1: procedure APPENDLITERAL(words[·], top, w)
2:     if words[·] is empty then {first append}
3:         top ← 0
4:         words[top] ← w
5:     else if w = 80000000h then {all 0's literal}
6:         if words[top] = 80000000h then
7:             words[top] ← 1 {sequence of 2 blocks of 0's}
8:         else if words[top] & C0000000h = 0 then
9:             words[top] ← words[top] + 1 {one more block}
10:        else if CONTAINSONEBIT(7FFFFFFFh & words[top]) then
11:            {convert the last one-set-bit literal in a mixed word of 2 blocks of 0's}
12:            words[top] ← 1 | ((1 + TRAILINGZEROS(words[top])) ≪ 25)
13:        else {nothing to merge}
14:            top ← top + 1,   words[top] ← w
15:        end if
16:    else if w = FFFFFFFFh then {all 1's literal}
17:        if words[top] = FFFFFFFFh then
18:            words[top] ← 40000001h {sequence of 2 blocks of 1's}
19:        else if words[top] & C0000000h = 40000000h then
20:            words[top] ← words[top] + 1 {one more block}
21:        else if CONTAINSONEBIT(~words[top]) then
22:            {convert the last one-unset-bit literal in a mixed word of 2 blocks of 1's}
23:            words[top] ← 40000001h | ((1 + TRAILINGZEROS(words[top])) ≪ 25)
24:        else {nothing to merge}
25:            top ← top + 1,   words[top] ← w
26:        end if
27:    else {nothing to merge}
28:        top ← top + 1,   words[top] ← w
29:    end if
30:    return words[·], top
31: end procedure
```

(b) Append of a literal word w. It checks whether the literal can be "merged" in a sequence with the last word of words. The main difference between this algorithm and its counterpart in [2] is represented by lines 10–12 and 21–23.

```
1: procedure APPENDSEQUENCE(words[·], top, ℓ, t)
2:     t ← t & 40000001h {retain only the fill type}
3:     if ℓ = 1 ∧ t = 0 then
4:         APPENDLITERAL(words[·], top, 80000000h)
5:     else if ℓ = 1 ∧ t = 40000000h then
6:         APPENDLITERAL(words[·], top, FFFFFFFFh)
7:     else if words[·] is empty then
8:         top ← 0,   words[top] ← t | (ℓ − 1)
9:     else if words[top] & 80000000h ≠ 0 then {the last word is a literal}
10:        if t = 0 ∧ words[top] = 80000000h then
11:            words[top] ← ℓ {make a sequence of ℓ + 1 blocks of 0's}
12:        else if t = 40000001h ∧ words[top] = FFFFFFFFh then
13:            words[top] ← 40000000h | ℓ {make a sequence of ℓ + 1 blocks of 1's}
14:        else if t = 0 ∧ CONTAINSONEBIT(7FFFFFFFh & words[top]) then
15:            words[top] ← ℓ | ((1 + TRAILINGZEROS(words[top])) ≪ 25)
16:        else if t = 40000001h ∧ CONTAINSONEBIT(~words[top]) then
17:            words[top] ← 40000000h | ℓ | ((1 + TRAILINGZEROS(words[top])) ≪ 25)
18:        else {nothing to merge}
19:            top ← top + 1,   words[top] ← t | (ℓ − 1)
20:        end if
21:    else {the last word is a sequence}
22:        if words[top] & C0000000h = t then
23:            words[top] ← words[top] + ℓ {increase the sequence}
24:        else {nothing to merge}
25:            top ← top + 1,   words[top] ← t | (ℓ − 1)
26:        end if
27:    end if
28:    return words[·], top
29: end procedure
```

(c) Append of a sequence word w. It checks whether the given sequence can be "merged" with the last word of words. The main difference between this algorithm and its counterpart in [2] is represented by lines 14–17.

```
1: procedure PERFORMOPERATION(S1, S2, ⋆)
2:     r1 ← new "run" for S1,   r2 ← new "run" for S2
3:     while r1.NOTEXHAUSTED() ∧ r2.NOTEXHAUSTED() do
4:         if r1.ISSEQUENCE() then
5:             if r2.ISSEQUENCE() then
6:                 c ← min{r1.count, r2.count}
7:                 R.APPENDSEQUENCE(c, r1.word ⋆ r2.word)
8:                 r1.NEXT(c),   r2.NEXT(c) {advance c blocks}
9:             else {convert the sequence r1.word into a literal}
10:                R.APPENDLITERAL(
11:                    (80000000h | (r1.word≪1)≫31) ⋆ r2.word)
12:                r1.NEXT(1),   r2.NEXT() {advance 1 block}
13:            end if
14:        else if r2.ISSEQUENCE() then {convert r2.word}
15:            R.APPENDLITERAL(
16:                r1.word ⋆ (80000000h | (r2.word≪1)≫31))
17:            r1.NEXT(),   r2.NEXT(1) {advance 1 block}
18:        else
19:            R.APPENDLITERAL(r1.word ⋆ r2.word)
20:            r1.NEXT(),   r2.NEXT() {advance 1 block}
21:        end if
22:    end while

23:    if ⋆ is bitwise OR or XOR then
24:        append to R all the remaining words of S1 and S2
25:    else if ⋆ is bitwise AND-NOT then
26:        append to R all the remaining words of S1
27:    end if
28:    return R
29: end procedure
```

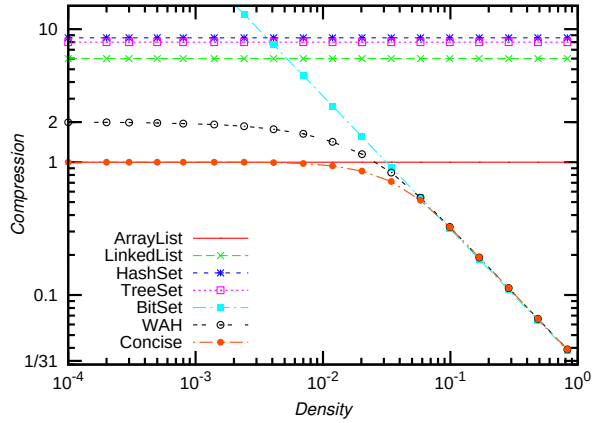
(d) Bitwise operations between two compressed bitmaps. This algorithm is perfectly equivalent to its counterpart in [2]. "⋆" indicates the desired binary operation, while the "runs" r_1 and r_2 are objects used to iterate over words—see [2] for more details.

Figure 3: The CONCISE algorithm. Figure (a) describes how to create new compressed bitmaps (indicated with the array words[·]) by "appending" integral numbers in ascending order. Figure (d) describes how to apply AND/OR/XOR/AND-NOT operations over compressed bitmaps. The other two algorithms are utility functions.

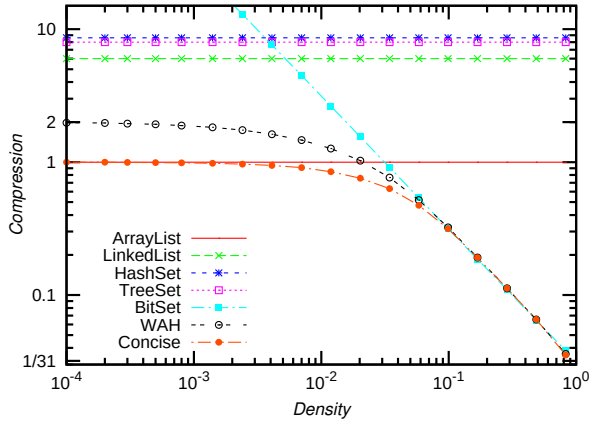
Table 1: Memory footprint analysis of standard Java structures from the package java.util, WAH, and CONCISE

Data Structure	*Distinct*	*Sorted*	*Bytes/Item*[a]	*Explanation*
ArrayList			4	Simple array. It is the simplest data structure. It is internally represented by an array of pointers (4 bytes each) to Integer instances.
LinkedList			24	Linked list. Each element of the list requires 4×3 bytes (4 bytes to point to the Integer instance, 4 bytes to point to the previous element, and 4 bytes to point to the next one), plus 8 bytes used by Java for each object instance, and 4 padding bytes.
HashSet	✓		$\geq$30	Hashtable. Each element requires 4×4 bytes (4 bytes to point to the key—the Integer instance—, 4 bytes to point to the value—not used in our tests—, 4 bytes to point to the next table entry in case of collision, and 4 bytes to store the hash value of the key), plus 8 bytes internally used by Java for the table entry. Moreover, we require an array of pointers (4 bytes for each element), considering that a hashtable must be greater than the maximum number of allowed elements in order to reduce the number of collisions.
TreeSet	✓	✓	32	Self-balancing, red-black binary search tree. Each node of the tree requires $4 \times 5 + 1$ bytes (4 bytes to point to the key—the Integer instance—, 4 bytes to point to the value—not used in our tests—, 4 bytes to point to the left node, 4 bytes to point to the right node, 4 bytes to point to the parent node, and 1 byte for the node color), plus 8 bytes internally used by Java for the node object, and 3 padding bytes.
BitSet	✓	✓	$1/8 \div 2^{28}$	Uncompressed bitmap. Each integral value is represented by a bit. In the worst case, we need a long sequence of zeros and then a word to store the integral. If we only consider positive integral numbers represented in two's complement over 32 bits, the greatest number is $2^{31} - 1$. In this case, we need a bitmap of 2^{28} bytes. In the best case, all integers represent a sequence of consecutive numbers, thus requiring only 1 bit on average.
WAH	✓	✓	$\sim 0 \div 8$	In the worst case, namely when numbers are very sparse, we need a literal word to store the integer (4 bytes) and a fill word to store a zero sequence (4 bytes). In the best case, all integers represent a sequence, thus requiring only 1 fill word (4 bytes) to represent all of them.
CONCISE	✓	✓	$\sim 0 \div 4$	In the worst case, namely when numbers are very sparse, we store each integer in each mixed fill word (4 bytes). In the best case, all integers represents a sequence, thus requiring only 1 fill word (4 bytes) to represent all of them.

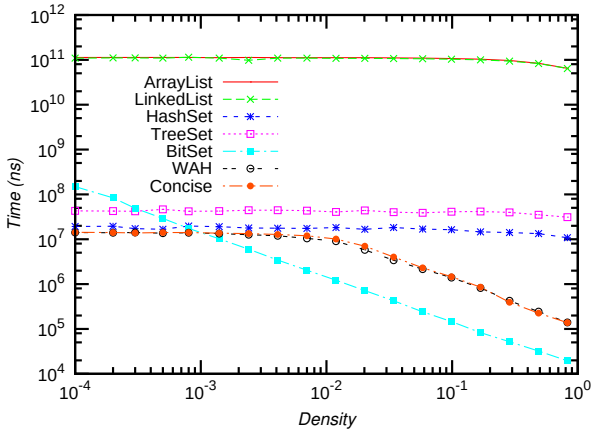
[a] Note that each Java object requires at least 8 + 8 bytes of memory: 8 bytes to represent pieces of information that are internally managed by the Java Virtual Machine (JVM), while user-defined object data should be aligned to a multiple of 8 bytes—in case, memory is padded with 0's. Moreover, in standard Java collections (namely any class implementing the Collection interface, such as ArrayList, LinkedList, HashSet, and TreeSet), integral numbers are stored within Integer instances. This means that each number to be stored requires 16 bytes (8 for internal JVM data, 4 for the integer, and 4 for padding) in addition to those reported in this table. Instead, BitSet, WAH, and CONCISE directly stores integers within an array of ints, hence avoiding the "waste" of this additional space.



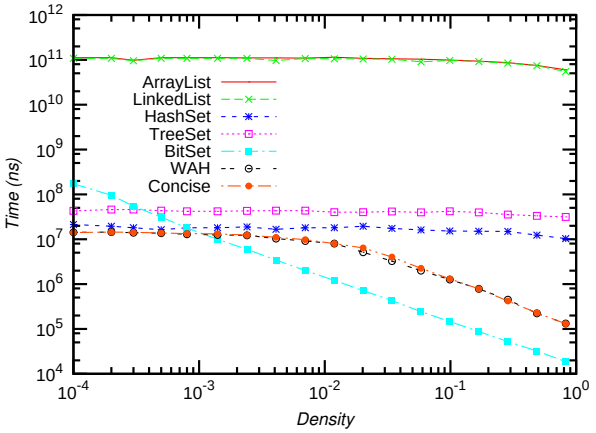
(a) Memory footprint—uniform distribution



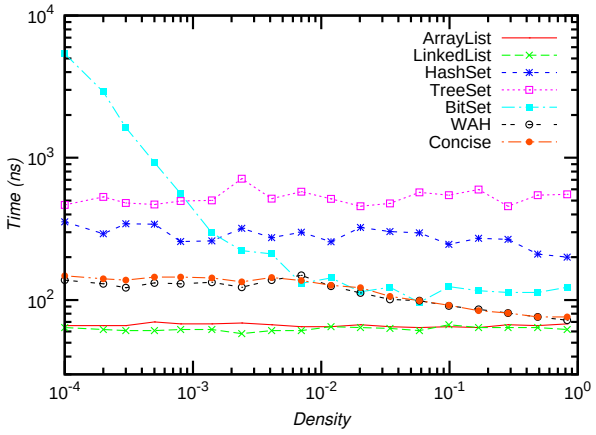
(b) Memory footprint—Zipfian distribution



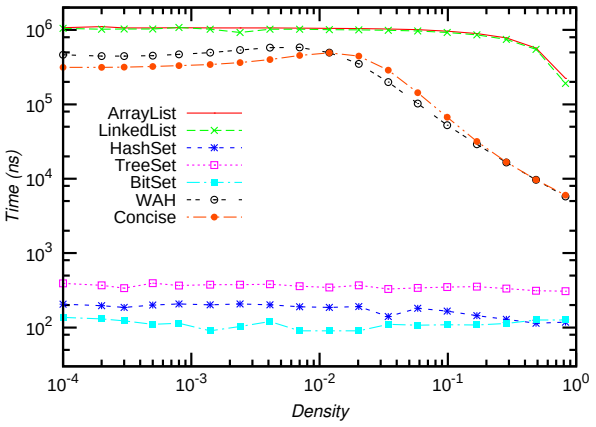
(c) Intersection—uniform distribution



(d) Intersection—Zipfian distribution



(e) Append—uniform distribution



(f) Removal—uniform distribution

Figure 4: Time and memory measurements. "Compression" means the ratio between the number of 32-bit words required to represent the compressed bitmap and the cardinality of the integer set. "Density" means the ratio between the cardinality of the set and the number range. Time measurements are expressed in nanoseconds. Since the experiments were performed in a multitasking environment, each experiment was performed 100 times and the average is reported.

at random, ranging from 0 to a maximum value max. The value of max depended on the desired *density* for each set, namely the ratio between the cardinality and the number range. In other words, given a density $d \in [0, 1]$, the maximum allowed integral value was $\text{max} = 10^5/d$. In all tests, we varied d from 0.0001 to 1. Integral numbers were generated according to two different probability distributions: *uniform* and *Zipfian*. As for uniform sets, for each generation of a pseudo-random number $a \in [0, 1)$ we added $\lfloor a \times \text{max} \rfloor$ to the set if not already existing. Regarding Zipfian sets, for each pseudo-random number $a \in [0, 1)$ we added $\lfloor a^2 \times \text{max} \rfloor$ to the set. This way, we generated skewed data such that integers were concentrated to lower values. The reason for using a Zipfian distribution is that, according to the Zipf's law, many types of data studied in the physical and social sciences can be approximated with a Zipfian distribution [12].

Figure 4a reports on the memory occupation of one randomly generated set with uniform distribution. It demonstrates that, according to [2], when density is below 0.05, WAH starts to compress the bitmap. Since CONCISE is able to compress the *same* bitmaps that WAH can compress, both algorithms start to compress after the same density threshold. However, CONCISE always shows a better compression ratio, which tends to be half of that of WAH when the density approaches zero. Figure 4b demonstrates that changing the distribution from uniform to Zipfian leads to very similar results. Both algorithms offer a slightly better compression ratio around a density of 10^{-2} in Zipfian sets because of the concentration of integers around lower values. In both cases, the compression ratio of WAH is twice of that of CONCISE as the data becomes more and more sparse. As expected, uncompressed bitmaps (e.g., BitSet) continue to increase as the maximum integer value grows, while other data structures are not affected by the data density.

Figure 4c reports on the intersection time of two sets, namely the time required for the identification of shared numbers between two sets. We do not show results for union and set difference because they have demonstrated a very similar computation time. For Java classes, intersecting corresponds to calling Collection.retainAll() and BitSet.and() methods. Notice that WAH and CONCISE are always faster than Java structures, apart from BitSet that is much faster when the set is not sparse. However, BitSet performance drastically decreases when data become very sparse. Again, other Java data structures are not affected by the density. In our experiments, we also noted (as expected) that the intersection time changes linearly compared to the cardinality of the set. Curves for Java data structures can be justified as follows: lists (ArrayList and LinkedList) require a full set scan to perform the intersection; binary tree (TreeSet) a logarithmic time search; and, hashtable (HashSet) an almost constant time search of shared elements.

In turn, we analyzed the time to add single numbers to a set. Figure 4e reports on the append time, namely on the addition of a new number strictly greater than the existing ones. Formally, $S \cup \{e\}$ when $\forall e' \in S : e' < e$. This corresponds to calling Java methods Collection.add()—BitSet.set() when numbers are first sorted. The append operation is the fastest way to add numbers to CONCISE and WAH bitmaps. Instead, sorting does not influence the performance of other data structures. The append time is constant for all data structures and, as we observed in our experiments, it does not change greatly as cardinality grows. However, for very sparse data, the BitSet class spends most of its time allocating memory, hence resulting in poor performances. Note that Figure 4e shows computation time for a random set with uniform distribution. We omitted to report a figure relative to the Zipfian distribution since it is very similar to the uniform case.

Finally, we analyzed the time needed to remove a single number from a set. In Figure 4f we report the corresponding execution time. Since both WAH and CONCISE do not explicitly support removal of single elements, we implemented it by performing the difference between the given set and a singleton. Note that the same thing can be done for the addition of new integers when the append operation is not possible, by just performing a union between the set and a singleton. In this case, the reduced size of the compressed bitmap makes CONCISE faster than WAH on sparse datasets. Similar to the previous case, we only report on the removal time for a random set with uniform distribution only, hence omitting the Zipfian distribution because very similar to the uniform case.

4. Conclusions

Because of their property of leveraging bit-level parallelism, computations over bitmaps often outperform computations over many other data structures such as self-balancing binary search trees, hash tables, or simple arrays. We confirmed, through experiment on synthetic datasets, that bitmaps can be very efficient when data are dense. However, when data become sparse, uncompressed bitmaps perform poorly due to the waste of memory. In this paper we introduced a new compression scheme for bitmaps, referred to as CONCISE, that is a good trade-off between the speed of uncompressed bitmaps and the required memory. Indeed, CONCISE outperformed all analyzed data structures in terms of memory occupation, as well as WAH, the best known compression algorithm that allows for set operations directly on the compressed form. Regarding computation time, CONCISE also outperformed classical data structures for set operations. Finally, accessing individual elements can be expensive for both CONCISE and WAH. If random access is more common than sequential access, and the integer set is relatively small, classical data structures may be preferable.

References

[1] P. Deutsch, Deflate compressed data format specification version 1.3, RFC 1951 (1996).

[2] K. Wu, E. J. Otoo, A. Shoshani, Optimizing bitmap indices with efficient compression, ACM Trans. Database Syst. 31 (1) (2006) 1–38.

[3] K. Wu *et al.*, FastBit: interactively searching massive data, Journal of Physics: Conference Series 180 (1) (2009) 012053.

[4] H. H. Malik, J. R. Kender, Optimizing frequency queries for data mining applications, in: Proc. ICDM, 2007, pp. 595–600.

[5] K. Wu, A. Shoshani, K. Stockinger, Analyses of multi-level and multi-component compressed bitmap indexes, ACM Trans. Database Syst. 35 (1) (2010) 1–52.

[6] G. Antoshenkov, Byte-aligned bitmap compression, Tech. rep., Oracle Corp., U.S. Patent number 5,363,098. (1994).

[7] F. Deliège, T. B. Pedersen, Position list word aligned hybrid: optimizing space and performance for compressed bitmaps, in: Proc. EDBT, 2010, pp. 228–239.
[8] Intel, Intel SSE4 programming reference (July 2007).
[9] P. Wegner, A technique for counting ones in a binary computer, Commun. ACM 3 (5) (1960) 322.
[10] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the Java HotSpot™ client compiler for Java 6, ACM Trans. Archit. Code Optim. 5 (1) (2008) 1–32.
[11] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30.
[12] G. Zipf, Human behavior and the principle of least effort, Addison-Wesley (1949).