An Experimental Study of Bitmap Compression vs. Inverted List Compression

Jianguo Wang Chunbin Lin Yannis Papakonstantinou Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego
{csjgwang, chunbinlin, yannis, swanson}@cs.ucsd.edu

ABSTRACT

Bitmap compression has been studied extensively in the database area and many efficient compression schemes were proposed, e.g., BBC, WAH, EWAH, and Roaring. Inverted list compression is also a well-studied topic in the information retrieval community and many inverted list compression algorithms were developed as well, e.g., VB, PforDelta, GroupVB, Simple8b, and SIMDPforDelta. We observe that they essentially solve the same problem, i.e., how to store a collection of sorted integers with as few as possible bits and support query processing as fast as possible. Due to historical reasons, bitmap compression and inverted list compression were developed as two separated lines of research in the database area and information retrieval area. Thus, a natural question is: *Which one is better between bitmap compression and inverted list compression?*

To answer the question, we present the first comprehensive experimental study to compare a series of **9** bitmap compression methods and **12** inverted list compression methods. We compare these **21** algorithms on synthetic datasets with different distributions (uniform, zipf, and markov) as well as **8** real-life datasets in terms of the space overhead, decompression time, intersection time, and union time. Based on the results, we provide many lessons and guidelines that can be used for practitioners to decide which technique to adopt in future systems and also for researchers to develop new algorithms.

1. INTRODUCTION

Bitmaps have been widely adopted in modern database systems including both row-stores and column-stores, e.g., PostgreSQL, Microsoft SQL Server, Oracle, MonetDB [32], C-store [1], Vertica [24], and Apache Hive [28]. A bitmap is allocated for a unique value in the indexed column. In particular, the i-th bit of the bitmap is 1 if and only if the i-th record contains that value. The number of bits in the bitmap is the number of records in the database (i.e., domain size). As an example of a smartphone sales database, assume "iPhone" appears at the 2nd, 5th, and 10th record in the `phone_name` column, then the bitmap of "iPhone" is 01001000010000000000 (assuming there are 20 records in total). Then many SQL queries can be answered efficiently with bitmaps. For example, finding the customers who bought "iPhone" from "California" can be framed as performing AND over the bitmaps of "iPhone" and "California". In practice, bitmaps are stored in a compact approach to save space [1, 18] because uncompressed bitmaps consume too much space especially for high-cardinality columns. As a result, many efficient bitmap compression methods were developed, e.g., BBC [22], WAH [22], EWAH [26], PLWAH [17], CONCISE [13], VALWAH [20], SBH [23], and Roaring [10].

Inverted lists are the standard data structure in modern information retrieval (IR) systems. All search engines, an important subset of IR systems, including Google [7, 16], Bing [21, 41], Yahoo! [6, 9], Apache Lucene, Apache Solr, Elasticsearch,[1] and Lucidworks[2] rely on inverted lists for finding relevant documents efficiently. An inverted list is a set of document IDs for those documents (e.g., web pages) that contain the term. To minimize space usage, inverted lists are typically stored in a compressed format [27], e.g., VB [15], PforDelta [43], NewPforDelta [40], Simple16 [42], GroupVB [16], Simple8b [3], PEF [30], SIMDPforDelta [25], and SIMDBP128 [25].

Motivation. Even though bitmap compression and inverted list compression are developed in two separated areas (database and information retrieval) individually, they essentially solve the same problem – *how to store a collection of sorted integers with as few as possible bits and support query processing (e.g., intersection) as fast as possible* – but from different angles. That is because a bitmap and an inverted list can be converted to each other equivalently since the positions of 1's in the bitmap are the actual values in the inverted list. For instance, the bitmap 01001000010000000000 can be converted to the inverted list $\{2, 5, 10\}$ and vice versa. Dating back to the 1970s, information retrieval researchers tried to consider bitmap compression to represent a collection of document IDs [34] but switched to inverted list compression since the 1990s [15]. But just in the 1990s [4], database researchers started to explore bitmap compression in SQL query processing and quickly established its important role in databases [11, 12, 38, 39]. Since then, bitmap compression and inverted list compression have become two

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064007

[1] https://www.elastic.co
[2] https://lucidworks.com

separated lines of research that usually do not refer to each other [2, 10, 16, 22, 23, 43].

We observe that a natural yet consistently overlooked issue in the literature is: *Which one is better between bitmap compression and inverted list compression*? By "better", we mean lower space and higher query performance. It is critical to answer the question by figuring out in what scenarios will bitmap compression or inverted list compression become better. Without a thorough understanding, as we show in experiments, an inappropriate method can incur up to 5000× more space overhead and 2000× performance degradation. Answering the question also provides insights and guidelines for researchers to develop new algorithms by considering the strengths and weaknesses of the two popular techniques.

Limitations of existing works. Previous studies mainly evaluated bitmap compression and inverted list compression individually. Usually, whenever a new bitmap compression method was proposed, it was solely compared against recent bitmap compression methods without comparing with inverted list compression methods [10, 13, 15, 17, 20, 23, 26]. Likewise, whenever a new inverted list compression method was invented, it was only compared with recent inverted list compression methods without comparing with bitmap compression methods [2–4, 16, 22, 25, 30, 40, 43]. As far as we know, there is only one existing work that compared bitmap compression with inverted list compression [8]. But there are some limitations observed in [8]:

- *Incomprehensive study.* It only compared one bitmap compression method WAH [22] with one inverted list compression method PforDelta [43]. However, neither is the best in their respective areas. For example, PLWAH [17], CONCISE [13], VALWAH [20], and Roaring [10] outperform WAH significantly. Also in the inverted list side, OptPforDelta [40], NewPforDelta [40], SIMDPforDelta [25], and SIMDBP128 [25] are better than PforDelta in many aspects. Therefore, it is inappropriate to draw conclusions by simply comparing WAH with PforDelta.
- *Uncontrolled memory and disk overhead.* In [8], the compressed data was stored on the disk but partially buffered in main memory through OS. However, this cannot guarantee that the OS buffer stores the same amount of data when running WAH and PforDelta. It is highly possible that the OS buffers more data for WAH than PforDelta (or vice versa) since it is completely up to the OS to control where the data is actually stored (disk or memory). As an example of the Figure 4 in [8], it took 28 seconds to execute Q2.1 for WAH on the SSB benchmark [29] with a scale factor of only 1. But it took almost zero time for PforDelta. Thus, it is likely that WAH incurred a higher penalty (than PforDelta) from accessing disk.
- *Inconsistent programming languages.* In [8], WAH was implemented in C++ but PforDelta was implemented in Java. Besides that, the compilation flags were not documented in that work. The O3 and O0 flags make a huge difference in execution time.
- *Insufficient datasets.* It used just one real dataset (SSB) [29]. However, there are many other real datasets used in prior works but were ignored by [8].

Contributions. The main contribution is that we present the first comprehensive experimental study to compare a series of **9** bitmap compression methods and **12** inverted list compression methods in main memory.[3] We evaluate these **21** compression algorithms on synthetic datasets with different distributions (uniform, zipf, and markov) as well as **8** real-life datasets regarding the space overhead, decompression time, intersection time, and union time.

Based on the results, we refresh the understanding of bitmap compression and inverted list compression. In particular, we (1) remedy many misunderstandings and inaccurate conclusions made in prior works; and (2) show the scenarios of when will bitmap compression outperform inverted list compression and vice versa.

Paper organization. The rest of this paper is structured as follows. Section 2 reviews existing bitmap compression techniques. Section 3 discusses inverted list compression methods. Section 4 describes the experimental setup. Section 5 presents experimental results on synthetic data. Section 6 shows experimental results on real data. Section 7 summarizes the paper and provides lessons.

2. BITMAP COMPRESSION

In this section, we review existing bitmap compression methods. Figure 1 shows a brief history.

Overview. All bitmap compression algorithms take as input an uncompressed bitmap and produce a compressed bitmap with as few as possible bits. They generally employ run-length encoding (RLE) to compress a sequence of identical bits with the bit value and count although there are exceptions such as Roaring. But different approaches differ in the way of handling the units of RLE (e.g., bytes or words), encoding the runs, and compressing the count.

2.1 WAH

WAH (Word-Aligned Hybrid) [22] is a classic bitmap compression algorithm. It partitions an uncompressed bitmap (input) into groups where each group contains 31 bits. It classifies those groups into two categories: *fill* groups and *literal* groups. A group is called a *fill group* if all the bits in the group are identical. For example, 00000000000000000000000000000000 (or 0^{31} in short) is a fill group. In particular, if all the bits are 1, it is called a 1-fill group; if all the bits are 0, it is called a 0-fill group. In contrast, a group is called a *literal group* if its bits contain both 0 and 1. WAH only compresses fill groups and does not compress literal groups. In particular, WAH compresses a sequence of consecutive fill groups together using just one word with the following information stored. The 1st bit is 1 indicating it stores the fill group. The 2nd bit indicates which fill group (0-fill or 1-fill). The rest 30 bits store the number of fill groups. WAH encodes a literal group using one word (32 bits), where the first bit of the word is 0 and the rest 31 bits are copied from the literal group. For example, assume the input uncompressed bitmap (160 bits) is $10^{20}1^30^{111}1^{25}$ (note that 0^{20} means twenty consecutive 0's). Then WAH partitions it into 6 groups: G_1 ($10^{20}1^30^7$), G_2 (0^{31}), G_3 (0^{31}), G_4 (0^{31}), G_5 ($0^{11}1^{20}$), G_6 ($0^{26}1^5$). Then WAH encodes G_1 using ($010^{20}1^30^7$); encodes G_2, G_3, and G_4 together using $100^{27}011$; encodes G_5 using $00^{11}1^{20}$; G_6 using $00^{26}1^5$.

[3] We leave the discussion of SSDs [36, 37] to future work.

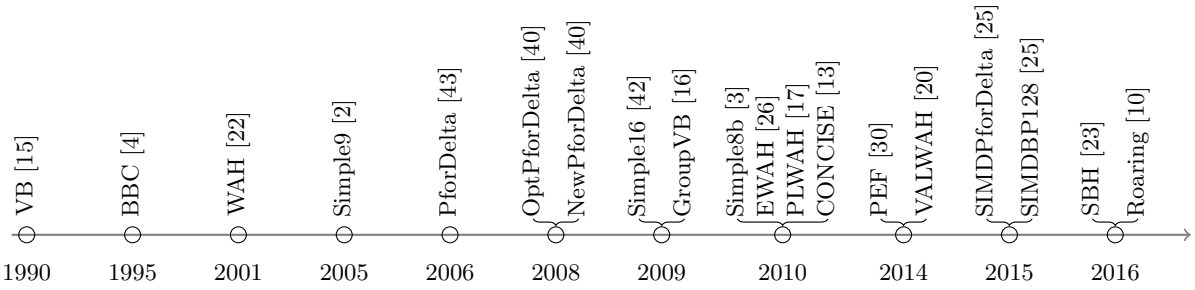


Figure 1: A brief history of representative bitmap compression and inverted list compression approaches

The advantage of WAH (as well as other bitmap compression methods) is that it supports intersection (AND) and union (OR) directly on compressed bitmaps without decompression. The intersection algorithm follows a merge-based style by allocating each compressed bitmap a pointer to denote the active word [22]. Every time, WAH performs intersection over two active words depending on different cases: literal word vs literal word, literal word vs fill word, fill word vs literal word, and fill word vs fill word. After that, it adjusts the pointers accordingly for further intersection.[4] The algorithm terminates once a bitmap does not have any active word. The union algorithm works in a similar way.

All the other bitmap compression methods adopt a similar intersection and union algorithm without decompression but differ in handling the intersection of two active bytes/words and interpreting the meaning of the active bytes/words.

2.2 EWAH

EWAH (Enhanced Word-Aligned Hybrid) [26] is a variant of WAH by observing that WAH takes too much space to store literal groups. EWAH divides an uncompressed bitmap (input) into 32-bit groups. It encodes a sequence of p ($p \leq 65535$) fill groups and q ($q \leq 32767$) literal groups into a marker word followed by q literal words (stored in the original form). The marker word stores the following information. The 1st bit indicates which fill group (0-fill or 1-fill), the next 16 bits store the number of fill groups p. The last 15 bits store the number of literal groups q. EWAH always starts with a marker word. For example, assume the input bitmap (160 bits) is $10^{20}1^30^{111}1^{25}$, then EWAH partitions it into 5 groups: G_1 ($10^{20}1^30^8$), G_2 (0^{32}), G_3 (0^{32}), G_4 (0^{32}), G_5 (0^71^{25}). Then EWAH encodes G_1 using $\underbrace{00^{16}0^{14}1}_{marker}10^{20}1^30^8$ ($p = 0$, $q = 1$); encodes G_2, G_3, G_4, G_5 using $\underbrace{00^{13}1000^{14}1}_{marker}0^71^{25}$ ($p = 4$, $q = 1$).

2.3 CONCISE

CONCISE (Compressed N Composable Integer Set) [13] is also proposed to improve WAH. It addresses a limitation of WAH that if a literal group has only one bit that is different from the next fill group (the literal group is called a *mixed fill group* and the bit is called an *odd bit*), then WAH still uses a full word to store the literal group. In contrast, CONCISE stores the mixed fill group with the next fill group together by storing the position of the odd bit. In particular, CONCISE partitions an uncompressed bitmap (input) into 31-bit groups. It uses one word to represent a literal group by setting the first bit to 1 and the rest 31 bits by copying the information from the 31-bit group. CONCISE also uses a word to represent a run of fill groups. It contains the following information. The 1st bit is set to 0. The 2nd bit indicates which fill group (0-fill or 1-fill). The following 5 bits store the position of the odd bit, but they are 00000 if there is no any odd bit. The remaining 25 bits store the number of fill groups minus one. For example, assume the input bitmap (160 bits) is $0^{23}10^{111}1^{25}$, then CONCISE partitions it into 6 groups: G_1 ($0^{23}10^7$), G_2 (0^{31}), G_3 (0^{31}), G_4 (0^{31}), G_5 ($0^{11}1^{20}$), G_6 ($0^{26}1^5$). Then WAH encodes G_1 to G_4 using $10001110^{22}011$; encodes G_5 using $00^{11}1^{20}$; G_6 using $00^{26}1^5$.

[4]WAH may also adjust the in-word pointer of a literal word to indicate which fill group is active.

2.4 PLWAH

PLWAH (Position List WAH) [17] is similar to CONCISE, but stores the mixed fill group in a different way. PLWAH compresses a bitmap by four steps. (1) It partitions an uncompressed bitmap (input) into 31-bit groups. (2) It identifies fill groups, i.e., groups whose bits are all 0 or 1. (3) It merges those consecutive fill groups together using a word. The 1st bit is 1 indicating it encodes the fill groups. The 2nd bit indicates which fill group (0-fill or 1-fill). The next 5 bits are 00000 indicating it contains pure fill groups. The rest 25 bits store the number of fill groups. (4) It identifies the literal group preceded by a fill group but the literal group has only one bit that is different from the fill group. It encodes all the fill group and the mixed fill group together using one word. Compared with the previous step, the only difference is that the unset 5 bits denote the odd bit position. For example, assume the input uncompressed bitmap (160 bits) is $10^{20}1^30^{111}1^{25}$. Then PLWAH partitions it into 6 groups: G_1 ($10^{20}1^30^7$), G_2 (0^{31}), G_3 (0^{31}), G_4 (0^{31}), G_5 ($0^{11}1^{20}$), G_6 ($0^{26}1^5$). Then PLWAH encodes G_1 using ($010^{20}1^30^7$); encodes G_2, G_3, and G_4 together using $10000000^{22}011$; encodes G_5 using $00^{11}1^{20}$; G_6 using $00^{26}1^5$.

2.5 VALWAH

VALWAH (Variable-Aligned Length WAH) [20] improves upon WAH by addressing the following limitation of WAH: WAH uses 30 bits to represent a run of up to $2^{30} - 1$ fill groups. However, in most cases, there are much fewer fill groups. As a result, VALWAH chooses a much smaller segment size s such that multiple segments can be encoded into a w-bit word (e.g., $w = 32$). Let b be the alignment factor, then a w-bit word can contain at most $\frac{w}{b}$ segments. However, it is also possible that fewer segments are stored.

In this case, the segment size is multiple of $(b-1)$ bits. Precisely, s can be determined by $s = 2^i \times (b-1)$, where $0 \leq i \leq (\log_2 w - \log_2 b)$. VALWAH also uses another parameter λ to make a tradeoff between space overhead and query performance. VALWAH encodes different bitmaps using different segment lengths to minimize the space overhead. However, this directly affects the query performance because of the segment alignment issue.

2.6 SBH

SBH (Super Byte-aligned Hybrid) [23] is a byte-aligned (instead of word-aligned) bitmap compression method in order to reduce the space overhead. SBH first divides an uncompressed bitmap (input) into a sequence of 7-bit groups. It encodes a literal group with one byte, where the 1st bit is set to 0 and the rest 7 bits are copied from the literal group. It encodes a sequence of consecutive k ($k \leq 4093$) fill groups as follows. (1) If there are k consecutive fill groups and $k \leq 63$, then SBH encodes those fill groups using one byte: the 1st bit is set to 0, the 2nd bit indicates which fill group (0-fill or 1-fill), and the rest 6 bits store the number k. (2) If $63 < k \leq 4093$, then SBH stores those fill groups using two bytes. For the first byte, the 1st bit is set to 0, the 2nd bit indicates which fill group, the rest 6 bits of the current byte stores the lower 6 bits of k. For the second byte, the first two bits are the same with the previous byte, and the rest 6 bits store the higher 6 bits of k. For example, assume the input bitmap (560 bits) is $10^{20}1^30^{511}1^{25}$, then SBH partitions it into 80 groups: G_1 (10^6), G_2 (0^7), G_3 (0^7), G_4 (1^70^4), G_5 to G_{76} (0^7), G_{77} (0^31^4), G_{78} (1^7), G_{79} (1^7), G_{80} (1^7). Thus, SBH encodes G_1 using 010^6; encodes G_2 and G_3 together using 10000010 ($k = 2$); encodes G_4 using 01^70^4; encodes G_5 to G_{76} using two bytes: 1000100010000001 ($k = 72$); encodes G_{77} using (00^31^4); encodes G_{78} to G_{80} using 11000011.

2.7 Roaring

Roaring [10] is a hybrid bitmap compression method and it is not based on run-length encoding. Roaring partitions the entire domain $[0, n)$ ($n < 2^{32}$) into different buckets of range 2^{16} where all the elements in the same chunk share the same 16 most significant bits. For example, the first three buckets manage the following ranges: $[0 \sim 65535]$, $[65536 \sim 65536 \times 2 - 1]$, and $[65536 \times 2 \sim 65536 \times 3 - 1]$. Roaring encodes a chunk depending on the number of actual elements k in that chunk. In particular, if $k > 4096$, Roaring uses a 65536-bit uncompressed bitmap to encode the elements; otherwise, it uses a sorted array of 16-bit short integers. Roaring chooses 4096 as the threshold because it guarantees that each integer uses no more than 16 bits to represent, because Roaring uses either 65536 bits to represent 4096 integers or at most 16 bits per integer for the array bucket. Note that, Roaring does not need to store the higher 16 bits for all the elements within the same bucket because they are the same. Thus, Roaring is a hybrid compression method that incorporates uncompressed (16-bit) integer list and uncompressed bitmap.

Roaring also supports intersection (AND) and union (OR) directly on compressed bitmaps. During each round, it intersects (or unions) two buckets from different bitmaps according to the following four combinations: bitmap vs bitmap, bitmap vs array, array vs bitmap, and array vs array.

2.8 BBC

Finally, we explain BBC (Byte-aligned Bitmap Code) [4, 22], which is one of the earliest bitmap compression algorithms. We put it at the end of this section because of its complexity. BBC partitions an uncompressed bitmap (input) into bytes where each byte contains 8 bits. It classifies those bytes into two categories: *fill* bytes and *literal* bytes. A byte is called a *fill byte* if all the bits in the byte are identical. In particular, if all the bits are 1, it is called a 1-fill byte; if all the bits are 0, it is called a 0-fill byte. In contrast, a byte is called a *literal byte* if its bits contain both 0 and 1. BBC compresses a collection of such bytes by identifying different patterns (or cases) and encodes each pattern individually to save space.

Pattern 1: a sequence of at most 3 fill bytes followed by at most 15 literal bytes. BBC stores this sequence of bytes by a header byte followed by a sequence of literal bytes. The header byte (8 bits) contains the following information. The 1st bit is 1 to indicate Pattern 1. The 2nd bit indicates which fill byte (0-fill byte or 1-fill byte). The next two bits (3rd and 4th bit) store the number of fill bytes. And the remaining four bits store the number of literal bytes. As an example of Figure 2a, it has two fill bytes and two literal bytes. BBC encodes the two fill bytes using a single byte 10100010 and the two literal bytes as they are.

Pattern 2: a sequence of at most 3 fill bytes followed by a single byte with only one bit that is different from the previous fill byte. BBC encodes this sequence of bytes in a single compact byte. The first two bits (1st and 2nd bit) are 01 to indicate Pattern 2. The 3rd bit indicates which fill byte (0-fill or 1-fill). The next two bits (4th and 5th bit) store the number of fill bytes. And the rest three bits store the position of the odd bit. As an example of Figure 2b, the input is 00000000 00000000 00000010, the output is 01010001.

Pattern 3: a sequence of at least 4 fill bytes followed by up to 15 literal bytes. BBC compresses this sequence of bytes by a header byte, followed by a multi-byte counter, and a sequence of literal bytes. The header byte stores the following information. The first three bits are 001 to indicate Pattern 3. The 4th bit indicates which fill byte (0-fill or 1-fill). The last four bits store the number of literal bytes. The multi-byte counter stores the number of fill bytes. The counter is compressed using VB compression (explained in Section 3.1). Figure 2c shows an example where the multi-byte counter (i.e., 4) is encoded as 00000100.

Pattern 4: a sequence of at least 4 fill bytes followed by a single byte with only one bit that is different from the previous fill byte. BBC encodes this sequence of bytes in a header byte and a multi-byte counter. The header byte stores the following information. The first four bits are 0001 to indicate Pattern 4. The 5th bit indicates which fill (0-fill or 1-fill). The last three bits store the position of the odd bit. The multi-byte counter stores the number of fill bytes encoded in VB. Figure 2d shows an example of this case.

3. INVERTED LIST COMPRESSION

In this section, we describe inverted list compression methods (see Figure 1 for a brief history).

Overview. Inverted list compression approaches usually follow a common wisdom of computing the deltas (a.k.a *d-gaps*) between two consecutive integers first and then compress the detals, although there are exceptions such as PEF and SIMDBP128*. For example, let $L = \{10, 16, 19, 28, 39, 48, 60\}$, then most solutions usually convert L to $L' = \{10, 6,$

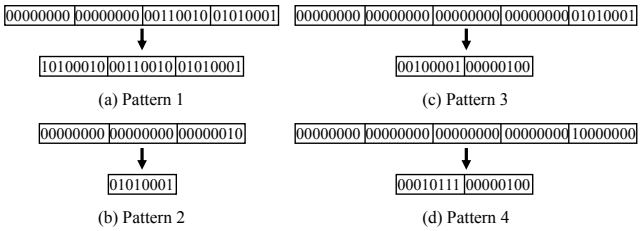


Figure 2: An example of BBC

3, 9, 11, 9, 12}, where $L'[0] = L[0]$ and $L'[i] = L[i] - L[i-1]$ ($i \geq 1$). To prevent from decompressing the entire list during query processing, it organizes those d-gaps into blocks (of say 128 elements per block[5]) and builds a skip pointer per block such that only a block of data needs to be decompressed [40, 42, 43].

Although the run-length encoding in bitmap compression algorithms bears some commonality with the d-gaps in inverted list compression algorithms, they are different. The run-length encoding is usually byte-level or word-level (instead of bit-level), e.g., BBC, WAH, EWAH, PLWAH, CONCISE, VALWAH, SBH, in order to leverage bit-wise operations (between two literal bytes or words). Thus, consecutive identical bits within the same byte or word are not compressed and they are kept as a literal byte or word. However, the d-gaps are essentially bit-level run-length encoding. As a result, inverted list compression algorithms cannot benefit from bit-wise operations but can benefit from efficient data skipping due to skip pointers.[6]

3.1 VB

VB (Variable Byte) [15] is a classic compression technique that is known under many names, e.g., VByte, Varint, VInt. VB encodes each integer (i.e., d-gap) in one or more bytes. It uses 7 bits of a byte to store the actual data while keeping the most significant bit (MSB) as a flag bit to indicate whether the next byte belongs to this integer. For example, consider the value 16385 whose binary expression is 100000000000001, then VB encodes it using 3 bytes: **1**0000001 **1**0000000 **0**0000001.

3.2 GroupVB

GroupVB (Group Varint Encoding) [16] is developed by Google by observing that VB incurs many branches during decompression. GroupVB compresses four values at the same time and it uses a 2-bit flag for each value. GroupVB factors out the four 2-bit flags into a single header byte followed by all the data bits. Such a layout makes it easier to decompress multiple integers simultaneously to reduce CPU branches.

3.3 PforDelta

PForDelta [43] is a mature compression algorithm. The basic idea is that it compresses a block of 128 d-gaps by choosing the smallest b in the block such that a majority of elements (say the threshold is 90%) can be encoded in b bits (called *regular values*). It then encodes the 128 values by allocating 128 b-bit slots, plus some extra space at the end to store the values that cannot be represented in b bits (called *exceptions*). Each exception takes 32 bits while each regular value takes b bits. In order to indicate which slots are exceptions, it uses the unused b-bit slots from the pre-allocated 128 b-bit slots to construct a linked list, such that the b-bit slot of one exception stores the offset to the next exception. In the case where two exceptions are more than 2^b slots apart, it adds additional forced exceptions between the two slots.

Besides the standard PforDelta, we also run PforDelta by setting the percentage of regular values in a block to be 100%. We refer to it as PforDelta*. PforDelta* chooses b such that *all* values are within 2^b instead of a *majority* of elements being within 2^b (as in PforDelta). Thus, PforDelta* does not have exceptions. The advantage of PforDelta* over PforDelta is that the decompression is ultra fast as it does not need to traverse the array of exceptions.

3.4 NewPforDelta

NewPforDelta [40] is a variant of PforDelta to reduce the space overhead of PforDelta. The main difference is that NewPforDelta stores the offset of exceptions in two arrays while PforDelta uses one array. Recall that, a limitation of PforDelta is that when two consecutive exceptions have a distance bigger than 2^b, PforDelta has to use more than one offset to represent the distance by adding forced additional exceptions between the two exceptions. NewPforDelta solves the problem by storing the offset values and parts of the exceptions in two additional arrays. More precisely, for an exception, NewPforDelta stores the lower b bits, instead of the offset to the next exception, in its corresponding b-bit slot. It then stores the higher overflow bits and the offset in two separated arrays. These two arrays can be further compressed.

3.5 OptPforDelta

OptPforDelta [40] is another variant of PforDelta. The main difference is that OptPforDelta uses the optimal b bits for each block while PforDelta chooses b such that most values are less than 2^b. Note that, setting a fixed threshold (say 90%) for the number of exceptions (as in PforDelta) does not give the best tradeoff between space overhead and query performance. Instead, OptPforDelta models the selection of b for each block as an optimization problem in order to get the optimal b.

3.6 Simple9

Simple9 [2] is a word-aligned compression method with the goal of packing as many small integers as possible to a 32-bit word. In Simple9, each word has 4 status bits and 28 data bits, where the data bits can represent 9 different combinations of values: 28 × 1-bit numbers, 14 × 2-bit numbers, 9 × 3-bit numbers (1 bit unused), 7 × 4-bit numbers, 5 × 5-numbers (3 bits unused), 4 × 7-bit numbers, 3 × 9-bit numbers (1 bit unused), 2 × 14-bit numbers, or 1 × 28-bit number. For example, if the next 14 values are all less than 4, then Simple9 stores them as 14 × 2-bit values. These 9 different combinations are indicated by 4 status bits.

3.7 Simple16

Simple16 [42] is similar to Simple9, but it has 16 cases. Recall that Simple9 uses 4 status bits to represent 9 cases

[5]The block size represents a tradeoff between space and time and several existing works suggest 128 as the block size [3, 42].

[6]Appendix B presents the details of using skip pointers for intersection.

and uses 28 bits to store data. Simple9 wastes bits in two ways. (1) Four status bits can express up to 16 cases while Simple9 only has 9 cases. (2) There are many unused bits. As an example of the 5 × 5-bit case in Simple9, there are 3 bits unused. Simple16 solves these two issues by introducing more new cases up to 16 cases. As an example the case of 5 × 5-bit numbers in Simple9, Simple16 replaces it using two cases: 3 × 6-bit numbers followed by 2 × 5-bit numbers, and 2 × 5-bit numbers followed by 3 × 6-bit numbers. Thus, Simple16 fully utilizes 4 status bits to represent 16 cases.

3.8 Simple8b

Simple8b [3] extends the codeword size to 64 bits instead of 32 bits. It retains the 4-bit selector such that each word has 4 status bits and 60 data bits. This saves space compared with Simple9 and Simple16 since fewer selectors are stored per encoded bit (4 bits per 60 bits rather than 4 bits per 28 bits). For example, Simple8b stores twelve 5-bit integers using one 64-bit codeword, but Simple9 needs three 32-bit codewords. In addition, Simple8b is more efficient than Simple9 and Simple16 because Simple8b leverages 64-bit instructions.

3.9 PEF

Different from other inverted list compression algorithms, PEF (Partitioned Elias Fano) [30] is not based on d-gaps. It is an improved version of EF (Elias Fano) encoding [35]. EF encodes a sequence of integers using a low-bit array and a high-bit array. The low-bit array stores the lower $b = \log \frac{U}{n}$ bits of each element contiguously where U is the maximum possible element and n is the number of elements in the list. The high-bit array then stores the remaining higher bits of each element as a sequence of unary-coded d-gaps. PEF improves EF by leveraging the clustering property of a list. It partitions a list and applies EF encoding within a partition [30].

3.10 SIMDPforDelta

SIMDPforDelta [25] is the SIMD version of PforDelta. It leverages modern CPU's SIMD instructions to accelerate the query performance and also decompression speed. A SIMD instruction operates on a s-bit register where s depends on different processors. Typically, s is 128, but more recent processors also support 256-bit or even 512-bit SIMD operation. In [25], it uses 128-bit SIMD instructions. The main idea of SIMDPforDelta is to reorganize data elements in a way such that a single SIMD operation processes multiple elements. SIMDPforDelta stores the input elements in an *interleaving* manner. In particular, let A be the input sorted array where each element takes b (say 10) bits, $\mathcal{S}$ be a 128-bit SIMD register, and $\mathcal{S}[i]$ be the i-th 32-bit bank. Then SIMDPforDelta stores $A[0] \sim A[3]$ at the lower b bits of $\mathcal{S}[0] \sim \mathcal{S}[3]$, respectively.

Similar to PforDelta* (Section 3.3), in the experiments, we also run the SIMD version of PforDelta*. We refer to it as SIMDPforDelta*.

3.11 SIMDBP128

SIMDBP128 [25] is one of the fastest compression methods for inverted lists. It partitions the input list L into 128-integer blocks and merges 16 blocks into a bucket of 2048 integers for SIMD acceleration. The metadata information of a bucket is a 16-byte array where each byte stores the number of bits used for encoding each block. Within each bucket, SIMDBP128 uses the same number of bits to encode each element.

In the experiments, we have also run SIMDBP128 at another version which we call it SIMDBP128*. It partitions a list into fixed-sized blocks where each block contains 128 elements. Then it maintains metadata for every 128-integer block and uses the same number of bits to represent the values within each block. Within a block, it organizes the elements in a SIMD-friendly manner.

4. EXPERIMENTAL SETUP

In this section, we present the experimental settings including experimental platform (Section 4.1), implementation details (Section 4.3), and evaluation metrics (Section 4.2).

4.1 Experimental platform

We conduct experiments on a commodity machine (Intel i7-4770 quad-core 3.40 GHz CPU, 64GB DRAM) with Ubuntu 14.04 installed. The CPU's L1, L2, and L3 cache sizes are 32KB, 256KB, and 8MB. The CPU is based on Haswell microarchitecture which supports AVX2 instruction set. We use mavx2 optimization flag for the SIMD acceleration. Besides that, all the algorithms are coded in C++ and compiled using GCC 4.4.7 with O3 enabled.

In all the experiments, we exclude the time of loading compressed data from disk to memory since we focus on evaluating the algorithmic performance in main memory. We leave in the future work to evaluate their performance on disks.

4.2 Evaluation metrics

We measure each compression algorithm mainly using the following four metrics:
(1) *Space overhead.* Any compression method aims for low space overhead to save memory footprint.
(2) *Decompression time.* Decompression overhead is critical to many other operations including intersection and union. For example, intersection needs to decompress part of the inverted lists even with skip pointers.
(3) *Intersection time.* Intersection is important in many applications including search engines and databases. For instance, intersection helps find the documents that contain all the query terms in search engines.
(4) *Union time.* Union is also important to both databases and search engines. For example, in databases, multi-criteria query and range query can be converted to the union of a collection of bitmaps.

4.3 Implementation

We implement all the bitmap compression methods (including Bitset, BBC [22], WAH [22, 39], EWAH [26], CONCISE [13], PLWAH [17], VALWAH [20], SBH [23], Roaring [10]) from scratch using C++. We try our best to implement each method as efficient as possible by using CPU's advanced instructions such as `popcnt` and `ctz` whenever possible. The intersection as well as union of two compressed bitmaps is a list of uncompressed integers.

In this work, we implement VB, PEF, PforDelta*, SIMDPforDelta*, and SIMDBP128* from scratch in C++. We implement the rest inverted list compression methods (including Simple9 [2], PForDelta [43], NewPforDelta [40], OptPforDelta [40], Simple16 [42], GroupVB [16], Simple8b [3], SIMDPforDelta [25], SIMDBP128 [25]) based on the existing

FastPFor[7] codebase in C++. For the intersection, we implement SvS [14] since it has been widely used in practice including Apache Lucene.[8] It works in the following way. Assume there are k lists $L_1, L_2, \cdots, L_k$ ($|L_1| \leq |L_2| \leq \cdots |L_k|$) that are compressed. SvS decompresses the shortest list L_1 first. Then for each element $e \in L_1$, SvS checks whether e appears in L_2. Note that SvS does not need to decompress the entire L_2 due to skip pointers and it only needs to decompress the block of data that potentially contains e. Then the results of L_1 and L_2 will be intersected with L_3 and the process continues until L_k. We implement the union by decompressing the lists first and merge them linearly. We provide more implementation details in Appendix B.

5. RESULTS ON SYNTHETIC DATASETS

In this section, we present the experimental results on evaluating bitmap compression and inverted list compression methods on synthetic datasets. We show the results of decompression (Section 5.1), intersection (Section 5.2), and union (Section 5.3).

Synthetic datasets. We generate synthetic datasets to understand when will bitmap compression methods outperform inverted list compression methods and vice versa. We generate data following the uniform distribution, zipf distribution, and markov distribution [39] respectively. Among all the distributions, the domain size is INTMAX, which is $2^{31}-1$. In particular, for the uniform distribution, each value is selected with the same probability. For the zipf distribution, each value is included with a different probability and the k-th value is included with a probability of $\frac{1/k^f}{\sum_{j=1}^{d}(1/j^f)}$ where f is the skewness factor and d is the domain size. For the markov distribution [39], the probability of transforming from 0 to 1 is $p = \frac{1}{f}$ and the probability of transforming from 1 to 0 is $q = \frac{\omega}{(1-\omega)f}$ where f is the clustering factor (which is 8 in our experiments following [39]) and ω is the density (i.e., the ratio of the list size and the domain size). Note that the probability of transforming from 0 to 0 is $(1-p)$ and the probability of transforming from 1 to 1 is $(1-q)$.

Note that, for all the legends in the figures, we use "Bitset" to represent uncompressed bitmap and "List" to represent uncompressed inverted list. Also, we measure the decompression overhead for an uncompressed list by allocating a new array and measuring the overhead of memory copy. Besides that, for all the inverted list compression methods (except uncompressed list), we partition a list into blocks of 128 elements and build skip pointers for the blocks following [27, 42]. Each skip pointer contains the offset (32 bits) and start value (32 bits). Although adding skip pointers increases the space overhead somehow, it will significantly improve query performance as we show in Appendix C.1. However, for bitmap compression methods, we do not build skip pointers following the convention [13,17,22,26], because otherwise, bitmaps cannot leverage efficient bit-wise operations for query processing.

5.1 Decompression

We start with evaluating the performance of decompression, because it is widely used in many operations including intersection and union. Figure 3 shows the decompression time and space overhead on uniform data and zipf data. We vary the list size from 1 million to 1 billion and set the domain size as INTMAX (2,147,483,647). The list size actually determines the density of the corresponding uncompressed bitmap: the longer the list is, the denser the bitmap becomes. Figure 3 shows that:

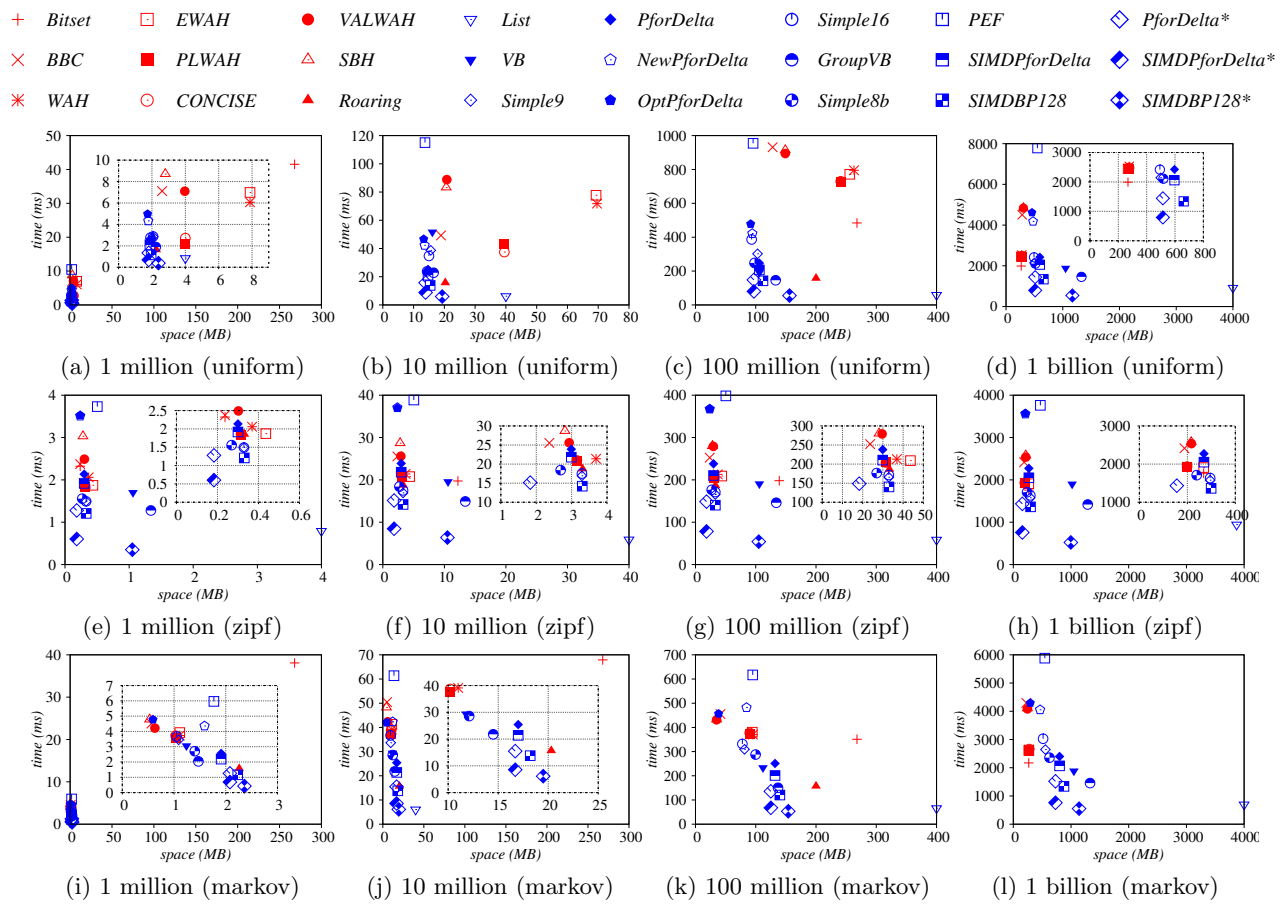
(1) In general, bitmap compression methods incur more space and higher decompression overhead than inverted list compression methods. An exception happens when the list size is 1 billion under the uniform distribution (Figure 3d) where bitmaps consume less space. Here are the reasons.

- Under the uniform distribution (Figure 3a – Figure 3d), when the list size is small, i.e., the uncompressed bitmap is sparse, there are many 0-fill words for bitmap compression methods (except Roaring). Recall that bitmap compression methods (except Roaring) require 32 bits to represent a 0-fill word. But inverted list compression methods do not need so many bits because any d-gap is smaller than 2^{32}. But when the list size increases to 1 billion, the uncompressed bitmap becomes very dense. As a result, bitmap compression methods (such as Bitset and WAH) only need around $\frac{d}{n}$ bits per integer where d is the domain size and n is the list size. However, inverted list compression methods tend to use more bits to represent an integer due to the outliers between two consecutive elements. In terms of decompression performance, bitmap compression methods tend to perform more bit operations because they need to access every bit for the literal words and extract one integer at a time for fill words. However, inverted list compression methods (e.g., PforDelta* and PforDelta) can extra many integers by looking at a word. Also, many inverted list compression methods can leverage SIMD capabilities, e.g., SIMDPforDelta* and SIMDPBP128.
- Under the zipf distribution, the case is largely similar to the uniform distribution. But when the list size is 1 billion, then all the integers become very concentrated at the beginning of the domain. In other words, the list becomes $\{1, 2, 3, 4, \cdots\}$. In this case, the number of outliers becomes very small such that inverted list compression methods can use the minimal number of bits to represent.
- Under the markov distribution, inverted list compression methods tend to have better decompression performance than bitmap compression methods because they can extract (or decompress) many integers by looking at a single word, especially with SIMD acceleration. However, bitmap compression methods need to access every bit for the literal words and extract one integer at a time for fill words. In terms of the space overhead, inverted list compression methods usually consume less space unless the list size is ultra large (e.g., 1 billion). That is because if the list size is 1 billion, then inverted list compression methods tend to have outliers that require more bits to represent when compared to bitmap compression methods.

(2) Among all the bitmap compression methods, Roaring is a winner in almost all cases in terms of both space overhead and decompression time. That is because Roaring is not based on run-length encoding and it incorporates uncompressed 16-bit integer list and uncompressed bitmap in a

[7]https://github.com/lemire/FastPFor

[8]Note that if two lists are of similar size, we switch to merge-based intersection for high performance.



(a) 1 million (uniform) (b) 10 million (uniform) (c) 100 million (uniform) (d) 1 billion (uniform)

(e) 1 million (zipf) (f) 10 million (zipf) (g) 100 million (zipf) (h) 1 billion (zipf)

(i) 1 million (markov) (j) 10 million (markov) (k) 100 million (markov) (l) 1 billion (markov)

Figure 3: Evaluating decompression with varying list sizes

hybrid way adaptively depending on the number of elements in a bucket.

(3) Among all the inverted list compression approaches, SIMDPforDelta* and SIMDBP128* are the top two most competitive techniques. Between the two, SIMDBP128* is faster but at the expense of consuming more space than SIMDPforDelta*. That is because SIMDPforDelta* stores the delta values to reduce space such that it needs extra time to compute prefix sums.

(4) Many bitmap compression methods (e.g., WAH and EWAH) can consume more space than the original list, i.e., uncompressed list, see Figure 3a, Figure 3b. However, inverted list compression methods never consume more space than the original list.

(5) For uncompressed bitmap (Bitset), it is dominated by Roaring in almost all cases. Figure 3 shows that Bitset only works well when it is very dense because its size as well as performance depends on the maximal element in the list (regardless of the list size). However, when the list size is long (e.g., 1 million), then Bitset is almost equivalent to Roaring because every bucket in Roaring turns to be represented by an uncompressed bitmap. However, when it is sparse, Roaring outperforms Bitset significantly.

(6) For BBC, its space overhead is almost the smallest when compared to other bitmap compression methods because of the four cases to minimize space overhead. However, its decompression speed is not excellent because it needs to handle many complicated cases.

(7) For SBH, it is worse than BBC in almost all cases. It is not surprising that SBH consumes more space because BBC has four cases in order to reduce space overhead. However, it is surprising that SBH is slower than BBC for decompression. That is because SBH needs to access the first two bits of the current and next byte during each iteration, which makes the algorithm slow. This contradicts with the original paper [23]. We speculate that the original paper may have implemented BBC poorly.

(8) For VB, it is not always worse than PforDelta, which contradicts with a prior paper [42]. Figure 3 shows that VB is faster than PforDelta under the zipf distribution. That is because VB operates on bytes instead of bits. However, when the list is very long, VB consumes much more space because it needs at least one byte to encode any integer. For example, Figure 3d shows that VB takes 1.76× more space than PforDelta.

(9) For PforDelta, it is a mature inverted list compression algorithm in the literature. There are many commonalities between PforDelta and WAH. First, both are very mature in each area; Second, PforDelta is based on d-gaps and WAH is based on run-lengths. Note that the d-gap is essentially the bit-level run-length. But it is interesting to see that PforDelta is better than WAH in terms of both decompression overhead and space overhead. The high performance is because WAH needs to perform many bit manipulations for literal words. The low space overhead is that PforDelta

	uniform				zipf		markov	
	1M	10M	100M	1B	100M	1B	100M	1B
Bitset	41.48	42.0	40.2	45.0	0.2	2.3	41.5	42.5
BBC	0.13	89.7	1004.1	269.1	0.3	3.4	260.8	1291.4
WAH	0.06	57.5	245.4	144.8	0.2	2.6	103.7	155.1
EWAH	0.10	91.2	135.4	47.0	0.2	2.3	106.6	56.6
PLWAH	0.15	82.7	329.3	156.9	0.2	2.5	134.9	176.4
CONCISE	0.09	76.2	577.6	301.6	0.2	2.4	137.8	263.8
VALWAH	0.06	87.5	805.6	980.4	0.3	3.2	207.7	1651.3
SBH	1.10	98.4	852.2	1128.0	0.3	3.9	228.6	1852.9
Roaring	0.03	1.7	14.0	10.9	0.2	1.9	5.3	5.7
List	0.01	2.3	23.7	241.9	5.9	64.7	9.3	103.6
VB	0.02	8.3	46.4	463.5	3.6	38.2	10.8	105.9
Simple9	0.02	6.9	60.2	475.8	3.3	35.5	12.3	119.2
PforDelta	0.03	5.6	55.9	526.5	3.8	39.5	9.6	100.2
NewPforDelta	0.02	7.3	74.9	728.9	4.2	48.7	10.5	110.9
OptPforDelta	0.02	7.9	82.3	804.9	4.2	49.2	10.8	117.4
Simple16	0.03	6.2	70.4	525.4	3.3	36.2	12.7	129.0
GroupVB	0.02	5.4	40.3	416.6	3.2	34.4	9.2	94.9
Simple8b	0.03	4.9	55.9	477.5	3.3	35.2	11.3	115.3
PEF	0.03	2.0	20.0	180.8	5.9	64.3	18.2	184.3
SIMDPforDelta	0.03	5.3	51.8	493.2	3.8	39.7	9.2	96.5
SIMDBP128	0.02	3.7	39.3	397.3	3.5	35.7	9.0	94.8
PforDelta*	0.03	4.1	41.8	401.6	3.2	34.5	9.1	93.4
SIMDPforDelta*	**0.01**	3.8	34.1	342.5	3.2	31.5	8.2	85.3
SIMDBP128*	**0.01**	3.0	31.6	315.2	2.7	30.3	7.7	81.3

Table 1: Evaluating intersection time (ms) with varying list sizes

encodes a majority of elements in a block using the same number of bits while dealing with exceptions separately.

(10) For the Simple family, Figure 3 shows that Simple8b consistently outperforms PforDelta under the zipf distribution in terms of both space overhead and execution time. This is a new result since previous works did not compare PforDelta and Simple8b [42]. But Simple9 and Simple16 are worse than PforDelta in most cases which confirms the results in [42].

(11) For GroupVB, its decompression performance is much better than PforDelta, sometimes, by a large margin. For example, Figure 3d shows that GroupVB is 1.6× faster in decompression than PforDelta. However, its space overhead is larger than PforDelta.

(12) For PEF, its decompression overhead is much worse than its competitors because it needs to access every bit of the high-bit array. This is a new result since the original paper did not evaluate its decompression overhead [30]. But an important property of PEF is that it does not need to decompress the entire block for some intersection as we explain in Section 5.2.

(13) For SIMDPforDelta, it takes the same amount of space with PforDelta but is around 1.2× faster in decompression performance. That is because a SIMD instruction can process multiple elements at the same time.

5.2 Intersection

We present the results of the intersection of two lists L_1 and L_2 ($|L_1| \leq |L_2|$) with different list sizes, while we evaluate the effect of skip pointers and list size ratios in Appendix C.1 and Appendix C.2.

Table 1 shows the effect of list size where both lists are generated following the uniform, zipf, and markov distribution. We set the $\frac{|L_2|}{|L_1|} = 1000$ and vary $|L_2|$ from 1 million to 1 billion. But on the zipf distribution and markov distribution, we omit the results of 1 million and 10 million due to space constraints. Table 1 shows that:

(1) In general, Roaring achieves the fastest intersection performance. The reason is that, Roaring stores uncompressed 16-bit integers and uncompressed bitmaps such that intersection can be performed efficiently on the compressed data. Moreover, Roaring can perform bucket-level intersection and skip many unnecessary buckets. As a result, it only intersects two promising buckets, i.e., two buckets sharing the same bucket number. Also, it can leverage in-bucket binary search for the array-array case to avoid unnecessary element-wise comparison. When the list size is 1 billion, the performance is extremely fast because there are many array-bitmap combinations that can be performed ultra efficiently since the array is essentially uncompressed (16-bit) and the bitmap is indeed uncompressed. This confirms the results made in [10]. The drawback of Roaring is that it consumes more space than inverted list compression methods such as SIMDPforDelta*.

(2) Among all the inverted list compression methods, PEF and SIMDBP128* are the most efficient algorithms. In particular, with the zipf distribution, SIMDBP128* is the fastest one while PEF is the best in other cases. That is because PEF does not need to decompress a whole block for intersection. However, under the zipf distribution, the elements tend to be concentrated at the beginning of the domain. As a result, all the elements in a block have to be decompressed. Recall in Figure 3 that PEF has the slowest decompression speed. Thus, PEF does not perform well in the cases where the entire lists need to be decompressed.

(3) VALWAH is much slower than WAH, sometimes, by a large margin. For example, under the uniform distribution, VALWAH is 1.3× to 6.7× slower than WAH, which is much more than the original paper claims (i.e., 3% slower) [20]. The main issue of VALWAH is that it has to deal with complicated segment alignment issue, which cannot process data as fast as it should be. But its space overhead is smaller than WAH because it uses various segment lengths for different lists.

(4) SBH is slower than BBC, which contradicts with the original paper [23]. The reason is that SBH needs to access more bits within a compressed word as explained in Section 5.1.

(5) VB is not always slower than PforDelta. This contradicts with the prior finding that PforDelta outperforms VB [42]. The advantage of VB comes from byte accesses instead of bit accesses.

(6) SIMDPforDelta is 5.1% to 7.2% faster than PforDelta but takes the same space overhead because a SIMD instruction processes more elements at the same time.

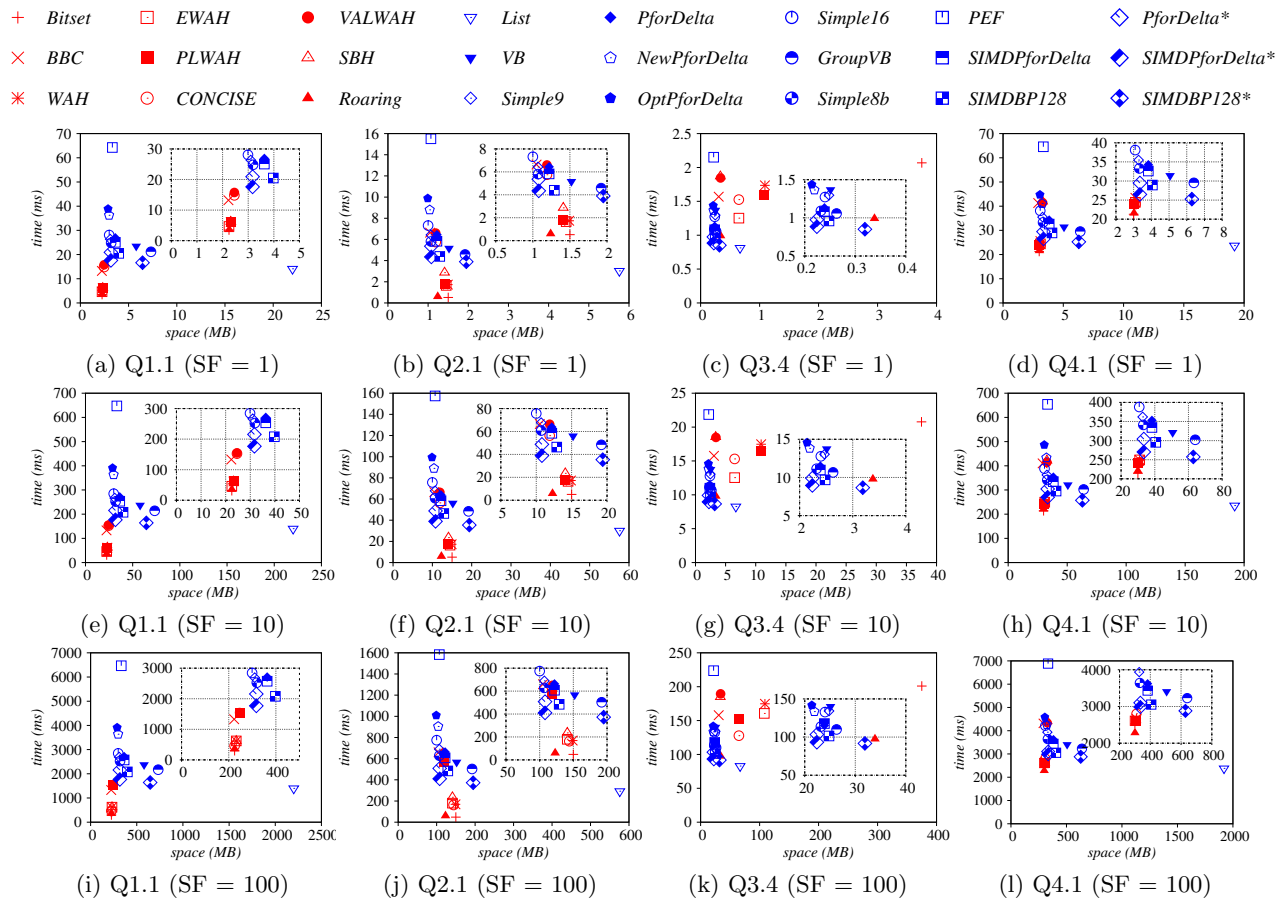
5.3 Union

Next, we show the experimental results on union. Union is an important operation because multi-criteria queries and range queries can be converted to the union of multiple bitmaps. We set $\frac{|L_2|}{|L_1|} = 1000$ and vary $|L_2|$. Table 2 shows the results.

(1) In general, inverted list compression methods are faster than bitmap compression methods (see Figure 3). That is because union tends to have high result size such that it requires many bit operations to extract all 1's to obtain a list of uncompressed integers for bitmap compression methods.

(2) Among all the bitmap compression methods, Roaring is the best in almost all cases in terms of space and time.

(3) Among all the inverted list compression approaches, SIMDBP128* and SIMDPforDelta* are very competitive in



(a) Q1.1 (SF = 1) (b) Q2.1 (SF = 1) (c) Q3.4 (SF = 1) (d) Q4.1 (SF = 1)

(e) Q1.1 (SF = 10) (f) Q2.1 (SF = 10) (g) Q3.4 (SF = 10) (h) Q4.1 (SF = 10)

(i) Q1.1 (SF = 100) (j) Q2.1 (SF = 100) (k) Q3.4 (SF = 100) (l) Q4.1 (SF = 100)

Figure 4: Results on SSB data

	uniform				zipf		markov	
	1M	10M	100M	1B	100M	1B	100M	1B
Bitset	59.0	129.0	510.9	2786.2	194.2	2908.3	359.4	3881.9
BBC	8.5	65.5	1144.1	7127.2	421.5	4232.9	632.2	13822.3
WAH	8.6	96.5	1007.9	4497.3	369.6	3676.3	552.4	10346.3
EWAH	3.7	59.4	906.4	4296.7	361.8	3671.4	537.1	10254.8
PLWAH	7.8	89.3	1157.6	5341.2	426.8	4313.0	548.7	11255.0
CONCISE	8.4	94.2	944.7	4334.9	365.5	3664.6	549.2	10982.3
VALWAH	10.5	101.5	1091.3	6797.4	438.1	4357.8	556.1	14138.8
SBH	4.3	53.4	887.5	4320.1	362.4	3704.5	614.7	12626.4
Roaring	3.2	31.4	314.8	3187.2	185.6	1906.5	174.5	3321.1
List	1.6	15.6	157.0	1579.6	156.8	1538.5	157.2	4406.8
VB	4.1	67.1	357.7	3581.6	354.0	3511.9	397.9	8037.7
Simple9	4.4	54.3	459.0	3778.9	323.5	3273.1	487.9	9532.7
PforDelta	4.0	42.1	417.9	4298.9	401.6	3956.7	422.3	9325.7
NewPforDelta	5.8	58.0	579.2	6284.7	550.6	5250.9	653.4	13229.7
OptPforDelta	6.6	61.9	630.6	6874.0	546.4	5241.6	624.0	14620.5
Simple16	4.4	50.5	544.1	4176.7	327.5	3297.1	511.8	10272.9
GroupVB	3.5	38.7	303.9	3087.0	303.5	3090.1	325.5	7217.1
Simple8b	4.3	39.8	406.6	3772.3	334.1	3373.5	428.8	8959.5
PEF	11.6	129.3	1109.0	9434.3	546.7	5432.1	790.2	15963.2
SIMDPforDelta	3.6	38.4	375.0	3902.9	379.0	3763.5	380.0	8575.6
SIMDBP128	2.7	29.6	291.7	2973.6	297.4	3022.52	293.5	6933.9
PforDelta*	2.9	31.6	306.1	3082.5	305.8	3132.3	311.9	7284.4
SIMDPforDelta*	2.3	24.8	237.8	2433.4	235.1	2455.0	240.3	5802.9
SIMDBP128*	2.0	21.8	212.8	2184.2	211.1	2210.9	214.4	5476.6

Table 2: Evaluating union time (ms) with varying list sizes

terms of both space and time. Note that Roaring is worse than SIMDBP128* and SIMDPforDelta* because Roaring cannot skip any element for the array-array case and also Roaring requires many bit operations to extract all 1's for the bitmap-bitmap case.

6. RESULTS ON REAL DATASETS

In this section, we present the results on real datasets. We have 8 real datasets in total. Due to space constraints, we present the results of SSB, TPCH, and Web data in Section 6.1, Section 6.2, and Section 6.3, respectively. We put the results on the other datasets in Appendix C.3 (Graph), Appendix C.4 (KDDCup), Appendix C.5 (Berkeleyearth), Appendix C.6 (Higgs), and Appendix C.7 (Kegg).

6.1 Results on SSB

The SSB (star schema benchmark)[9] is a typical database workload that includes one fact table (LINEORDER) and four dimension tables (CUSTOMER, SUPPLIER, PART, and DATE). We set the scale factor as 1, 10, and 100 so the number of rows in the fact table is around 6 million, 60 million, and 600 million. We use Q1.1, Q2.1, Q3.4, and Q4.1 for evaluation. Q1.1 involves an intersection of 3 lists with a selectivity of 1/7, 1/2, and 3/11 on the fact table. Q2.1 involves an intersection of 2 lists with a selectivity of 1/25 and 1/5 on the fact table. Q3.4 involves 5 lists (L_1, L_2, L_3, L_4, and L_5) with a selectivity of 1/250, 1/250, 1/250, 1/250, and 1/364. The query is $(L_1 \cup L_2) \cap (L_3 \cup L_4) \cap L_5$, which is a combination of intersection and union. Q4.1 involves 4

[9]http://www.cs.umb.edu/~poneil/StarSchemaB.PDF

lists (L_1, L_2, L_3, and L_4) with a selectivity of 1/5, 1/5, 1/5, and 1/5 on the fact table. The query is $L_1 \cap L_2 \cap (L_3 \cup L_4)$.

We present the results in Figure 4. (1) It shows that, for Q1.1, Q2.1, and Q4.1, Roaring and Bitset are the top two fastest approaches. That is because the lists involved in those queries are long (or dense). In this case, Roaring is similar to Bitset because most buckets in Roaring have more than 4096 elements. And they can leverage efficient bit-wise computations. For Q3.4, inverted list compression methods tend to be slightly better. In particular, SIMDPforDelta* and SIMDBP128* are the fastest. Roaring, on the other hand, is slightly slower than these two methods. That is because the lists in Q3.4 are short (or sparse) since the selectivities are 1/250 and 1/364. (2) In terms of space overhead, inverted list compression methods tend to consume less space. But when the lists are dense, e.g., Q4.1, bitmap compression methods consume similar space to inverted list compression methods. In particular, SIMDPforDelta* is the most space-efficient compression algorithm.

6.2 Results on TPCH

Then we show the results on TPCH, which is a popular decision support benchmark.[10] We set the scale factor as 1, 10, and 100. We choose a variant of Q6 and Q12 following [5]. In which, Q6 contains 3 lists (L_1, L_2, and L_3) with a selectivity of 1/7, 3/11, and 1/50. The query is $L_1 \cap L_2 \cap L_3$. Q12 also involves 3 lists (L_1, L_2, and L_3) but with a selectivity of 1/10, 1/10, and 1/364. The query is $(L_1 \cup L_2) \cap L_3$.

Figure 5 shows that for Q6, Roaring is the best compression method among all bitmap compression methods as well as inverted list compression methods. It is even faster than uncompressed list. That is because the lists are very dense and Roaring can leverage efficient bit-wise computations. However, for Q12, Roaring consumes more space than inverted list compression methods although it is still faster. In particular, SIMDPforDelta* has the least space overhead for Q12.

6.3 Results on Web data

Then we present the results on a typical information retrieval workload – Web data. It is a collection of 41 million Web documents (around 300GB) crawled in 2012.[11] It is a standard benchmark in the information retrieval community. We parse the documents and build inverted lists for each term. The query log contains 1000 real queries randomly selected from the TREC[12] 2005 and 2006.

We run those queries on the web data, report the average intersection and union time, and show the results in Figure 6. (1) In terms of intersection performance, Figure 6 clearly shows that Roaring outperforms the other compression methods. It is even faster than uncompressed list. That is because Roaring can leverage the merits of both uncompressed integer lists (16 bits) and uncompressed bitmaps. This means that, although information retrieval systems always use inverted lists for query processing, it does not mean inverted lists always achieve the best performance. Besides that, among all the inverted list compression methods (except uncompressed list), SIMDBP128* and SIMDPforDelta* are the top two fastest algorithms. (2) In terms of

[10]http://www.tpc.org/tpch/
[11]http://www.lemurproject.org/clueweb12.php
[12]http://trec.nist.gov/

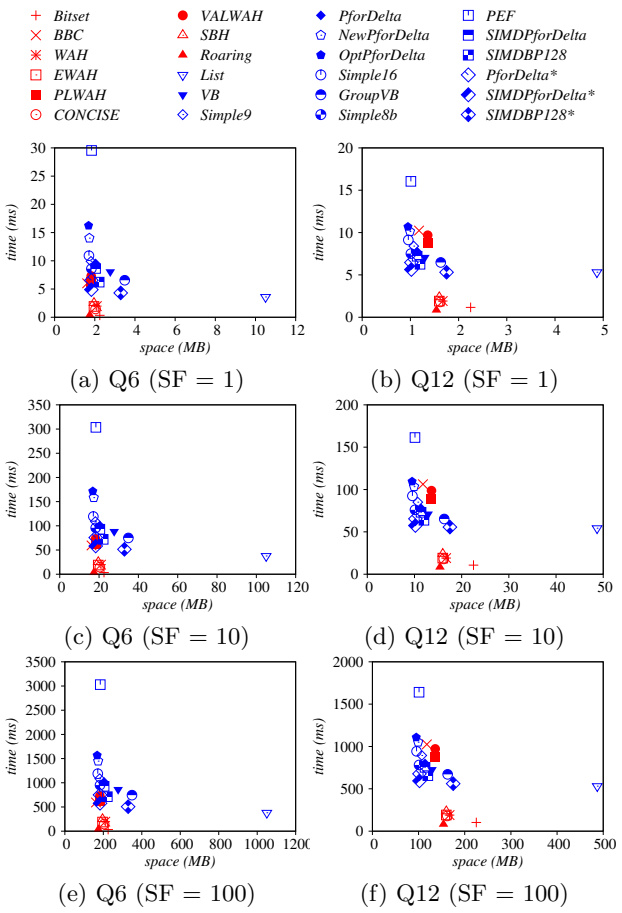


Figure 5: Results on TPCH data

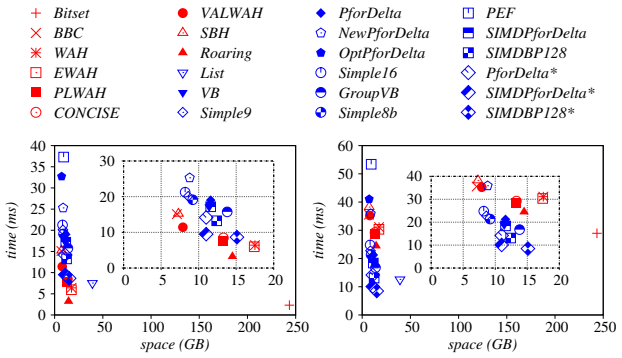


Figure 6: Results on Web data

union performance, Figure 6 clearly shows that inverted list compression methods outperform bitmap compression methods (including Roaring). In particular, SIMDPforDelta* and SIMDBP128* are the fastest. (3) In terms of space overhead, Figure 6 shows that inverted list compression methods tend to consume less space.

7. SUMMARY AND FURTHER DIRECTIONS

In this section, we summarize the main findings of the work and discuss future directions based on the results.

7.1 Bitmap compression or inverted list compression?

Let's come back to the question: *Which one is better between bitmap compression and inverted list compression*? Clearly it is not a black-white question. Although it is safe to say that the answer depends on different situations – indeed it is – that is not useful. Here we try to give a simple and decisive answer that can be used for future systems.

1. *Space overhead*. Inverted list compression methods generally take less space than bitmap compression methods unless the list L is ultra dense. In particular, (1) if L follows the uniform distribution or markov distribution, then if L is short or moderate long, e.g., $\frac{|L|}{d} < \frac{1}{5}$ (see Figure 4) where d is the domain size, then inverted list compression methods consume less space. In particular, SIMDPforDelta* takes the least space. If L follows the uniform distribution or markov distribution, then if L is extremely long, e.g., $\frac{|L|}{d} \geq \frac{1}{5}$, then bitmap compression methods consume less space. In particular, Roaring and Bitset consume the least space. (2) If L follows the zipf distribution, then no matter whether L is short or long, SIMDPforDelta* takes the least space.

2. *Decompression time*. Inverted list compression methods are generally faster than bitmap compression methods in nearly all cases. In particular, SIMDBP128* achieves the best decompression performance.

3. *Intersection time*. In general, Roaring bitmap achieves the fastest intersection performance among all the compression methods.

4. *Union time*. Inverted list compression methods generally have better union performance than bitmap compression methods. In particular, SIMDBP128* is the fastest one in nearly all cases.

Next, we discuss which technique should be used from a viewpoint of query-level (instead of operation-level).

1. Top-k query in information retrieval (Section A.1). We recommend Roaring because Roaring has the fastest intersection performance in general and intersection is the most time-consuming part in top-k query processing as explained in Section A.1.

2. Conjunctive query in databases (Section A.2). We recommend Roaring because it has the fastest intersection performance in general among all the compression methods.

3. Disjunctive query in databases (Section A.2). We recommend SIMDBP128* as it has the fastest union performance in nearly all cases.

4. Star join in databases (Section A.2). We recommend Roaring because a star join can be framed as the intersection as described in Section A.2 and Roaring has the highest intersection performance in general.

5. Range query in databases (Section A.2). We recommend SIMD-BP128* because a range query can be converted to the union as described in Section A.2 and SIMDBP128* generally has the highest union performance.

Overall recommendations. In summary, we recommend SIMDBP128*, SIMDPforDelta*, and Roaring as three competitive compression methods. We do not recommend Bitset since it consumes too much space in many cases. Our results are new and different from the prior conclusion that inverted list compression methods always outperform bitmap compression methods [8].

7.2 Lessons

We provide the lessons that people can learn from this work.

1. Although database systems preferred bitmap compression for around 20 years and information retrieval systems preferred inverted list compression for decades, it does not mean that bitmap compression is always better than inverted list compression or vice versa. Both techniques can learn from each other to develop a better unified compression method.

2. Although prior works claimed that bitmaps are suitable for high-cardinality columns via compression (e.g., BBC and WAH) [38, 39], this work shows that the space overhead is much higher than inverted list compression in that case. Thus, we amend the conclusion as: Bitmaps are still suitable for low-cardinality columns via compression and the compression method should be Roaring (instead of BBC and WAH).

3. Use Roaring for bitmap compression whenever possible. Do not use other bitmap compression methods such as BBC [22], WAH [22], EWAH [26], PLWAH [17], CONCISE [13], VALWAH [20], and SBH [23].

4. Be sure to keep it simple when you invent a new bitmap compression. A complicated bitmap compression algorithm (e.g., BBC and VALWAH) tends to incur high performance overhead.

5. Use SIMDBP128* and SIMDPforDelta* for inverted list compression for high query performance (e.g., intersection and union) and low space overhead.

6. Use VB [15] if you concern more on the implementation overhead. Based on our experience, VB is perhaps the simplest one to implement that requires less than 20 lines of C++ code (while BBC requires more than 3000 lines of code).

7. A compression method that is good for decompression may not be good for intersection, and a compression method that is good for intersection may also not good for union. This is an overlooked issue in many prior papers [2, 3, 25, 40, 43].

8. Be sure to build skip pointers on compressed inverted lists. It will not increase the space overhead that much (3% to 5%), but can improve the intersection performance dramatically.

9. Be sure to design a SIMD-aware compression method. By reorganizing the data layout will not increase the space overhead, but it will improve query performance.

8. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *IR*, 8(1):151–166, 2005.

[3] V. N. Anh and A. Moffat. Index compression using 64-bit words. *SPE*, 40(2):131–147, 2010.

[4] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC*, page 476, 1995.

[5] M. Athanassoulis, Z. Yan, and S. Idreos. Upbit: Scalable in-memory updatable bitmap indexing. In *SIGMOD*, pages 1319–1332, 2016.

[6] R. A. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, 2007.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] T. A. Bjørklund, N. Grimsmo, J. Gehrke, and O. Torbjørnsen. Inverted indexes vs. bitmap indexes in decision support systems. In *CIKM*, pages 1509–1512, 2009.

[9] B. B. Cambazoglu and R. A. Baeza-Yates. Scalability and efficiency challenges in large-scale web search engines. In *SIGIR*, pages 1223–1226, 2016.

[10] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *SPE*, 46(5):709–719, 2016.

[11] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, pages 355–366, 1998.

[12] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, pages 215–226, 1999.

[13] A. Colantonio and R. D. Pietro. Concise: Compressed 'n' composable integer set. *IPL*, 110(16):644–650, 2010.

[14] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[15] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *SIGIR*, pages 405–411, 1990.

[16] J. Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *WSDM*, page 1, 2009.

[17] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps. In *EDBT*, pages 228–239, 2010.

[18] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2008.

[19] G. Guzun and G. Canahuate. Hybrid query optimization for hard-to-compress bit-vectors. *VLDBJ*, 25(3):339–354, 2016.

[20] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *ICDE*, pages 484–495, 2014.

[21] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, pages 161–175, 2015.

[22] A. S. Kesheng Wu, Ekow J. Otoo and H. Nordberg. Notes on design and implementation of compressed bit vectors, 2001.

[23] S. Kim, J. Lee, S. R. Satti, and B. Moon. Sbh: Super byte-aligned hybrid bitmap compression. *IS*, 62:155–168, 2016.

[24] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[25] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *SPE*, 45(1):1–29, 2015.

[26] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *DKE*, 69(1):3–28, 2010.

[27] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[28] S. Mehrotra, S. Chauhan, and H. Bansal. *Apache Hive Cookbook*. Packt Publishing, 2016.

[29] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. 2009.

[30] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *SIGIR*, pages 273–282, 2014.

[31] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD*, pages 773–784, 2007.

[32] K. Stockinger, J. Cieslewicz, K. Wu, D. Rotem, and A. Shoshani. Using bitmap index for joint queries on structured and text data. In *New Trends in Data Warehousing and Data Analysis*, pages 1–23. 2009.

[33] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, pages 963–972, 2011.

[34] L. Thiel and H. Heaps. Program design for retrospective searches on large data bases. *IPM*, 8(1):1 – 20, 1972.

[35] S. Vigna. Quasi-succinct indices. In *WSDM*, pages 83–92, 2013.

[36] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. The impact of solid state drive on search engine cache management. In *SIGIR*, pages 693–702, 2013.

[37] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. Cache design of ssd-based search engine architectures: An experimental study. *TOIS*, 32(4):1–26, 2014.

[38] K. Wu, E. J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB*, pages 24–35, 2004.

[39] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *TODS*, 31(1):1–38, 2006.

[40] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[41] J. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *SIGIR*, pages 63–72, 2015.

[42] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[43] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.

APPENDIX

In the appendix, we provide applications (Section A), implementation details (Section B), and more experiments (Section C).

A. APPLICATIONS

In this section, we provide more details on the applications that require inverted list compression or bitmap compression.

A.1 Information retrieval

Information retrieval (IR) is a killer application where inverted lists are used to answer user queries with multiple terms [27]. IR systems store an inverted list for each term with all the document ids that contain the term. To save space overhead, IR systems usually compress inverted lists using VB [15], PforDelta [43], NewPforDelta [40], Simple16 [42], GroupVB [16], Simple8b [3], PEF [30], SIMDPforDelta [25], or SIMDBP128 [25].

In IR systems, typical query patterns include intersection,

union, list traversal, and top-k query processing [27]. In which, intersection or union of the lists for a set of query terms identifies those documents that contain all or at least one of the terms. List traversal is required for processing single-term queries and it is also important for supporting intersection and union. Also, top-k query processing helps find the most relevant documents to user queries.

Among those typical query patterns, there are three fundamental operations over compressed inverted lists: intersection, union, and decompression. (1) Intersection is used to answer intersection queries and it is also the most time-consuming part in top-k query processing [33]. Usually in IR systems, there are two steps to find the top k most relevant documents [33]: (i) find a list of candidate documents that contain all the query terms (via intersection) because those documents tend to be more relevant; (ii) compute the similarity between the query and each candidate document based on the payload information (e.g., document frequency) and return the top k most relevant documents. Among the two steps, the first step – intersection – is the most time-consuming part [33]. (2) Union is used to return documents that contain at least one of the query terms, which is extremely helpful when the intersection returns very few (or even empty) results. (3) Decompression is required for list traversal, intersection, and union.

A.2 Database

Database systems heavily use bitmaps for supporting efficient query processing. A bitmap contains a collection of 0's and 1's to indicate whether the corresponding rows contain that particular value. To save space, bitmaps are usually stored in a compressed manner, e.g., BBC [22], WAH [22], EWAH [26], PLWAH [17], CONCISE [13], VALWAH [20], SBH [23], and Roaring [10].

Bitmaps can be used to support many types of queries, e.g., conjunctive queries, disjunctive queries, table scans, range queries, and also star joins. Among them, there are three fundamental query operations over compressed bitmaps: intersection, union, and decompression. (1) Intersection is used to answer conjunctive queries and star joins. For instance, consider the star join example in [31] (Example 1) that analyzes the coffee sales. The star join can be framed as an intersection query between different bitmaps (or inverted lists) where each bitmap (or inverted list) is precomputed for a predicate as illustrated in [31]. (2) Union is used to support disjunctive queries and range queries. Note that a range query can be framed as a union between different bitmaps (or inverted lists) [38]. For example, finding the records whose ages are from 25 to 26 can be converted to the union of two bitmaps where the first bitmap is dedicated for age 25 and the second bitmap is dedicated for age 26. (3) Decompression is used to support table scan as well as intersection and union.

B. IMPLEMENTATION DETAILS

In this section, we provide more implementation details.

B.1 Bitmap compression implementation

We implement all the bitmap compression methods (including Bitset, BBC [22], WAH [22, 39], EWAH [26], CONCISE [13], PLWAH [17], VALWAH [20], SBH [23], Roaring [10]) from scratch using C++. Although there are open-source implementations for a number of bitmap compression methods (such as WAH, EWAH, and CONCISE), we do not adopt them due to the following limitations. (1) The implementations are in different languages: some are implemented in java and some are implemented in C++. (2) The query results are not returned in the same format: some implementations return the compressed results for query processing but others do not. For example, the existing implementation of EWAH returns a compressed bitmap for AND/OR but WAH returns a non-compressed bitmap. (3) Existing implementations have many bugs in our datasets, e.g., CONCISE throws an exception whenever the number of elements is greater than 1 billion and WAH returns incorrect results for many intersections and unions. (4) Some implementations do not support the query processing over more than two bitmaps, and it is non-trivial to extend them to multiple bitmaps if the returned results are non-compressed.

Due to the above limitations, we determine to implement all the bitmap compression methods by ourselves. We try our best to implement each algorithm as efficient as possible. In particular, we use an array of 32-bit integers as the backend storage and perform bit manipulations directly over the integer array. Whenever possible, we apply the CPU's instructions to perform fast bit operations. For example, we use the CPU's `popcnt` instruction to obtain the number of 1's in a word. We also use the `ctz` instruction to count the trailing zeros in order to find the positions of 1's in a word or find the odd bit position in a byte (for BBC) or a word (for EWAH). For intersection over two bitmaps, we have two versions with different inputs: (1) bitmap vs bitmap; (2) bitmap vs list. The intersection results are uncompressed such that they can be either directly returned to end users or as the input of other operations. The intersection between more than two bitmaps is implemented by interesting the first two bitmaps. Then the results (uncompressed integer list) will be intersected with the next compressed bitmap.

B.2 Inverted list compression implementation

For inverted list compression methods, we implement VB, PEF, PforDelta*, SIMDPforDelta*, and SIMDBP128* from scratch in C++. We implement the rest compression methods (including Simple9 [2], PForDelta [43], NewPforDelta [40], OptPforDelta [40], Simple16 [40], GroupVB [16], Simple8b [3], SIMDPforDelta [25], SIMDBP128 [25]) based on the existing FastPFor codebase[13] that has been used in [25]. The existing codebase only provides functions for compression and decompression. For each compression methods, we add the intersection and union operation.

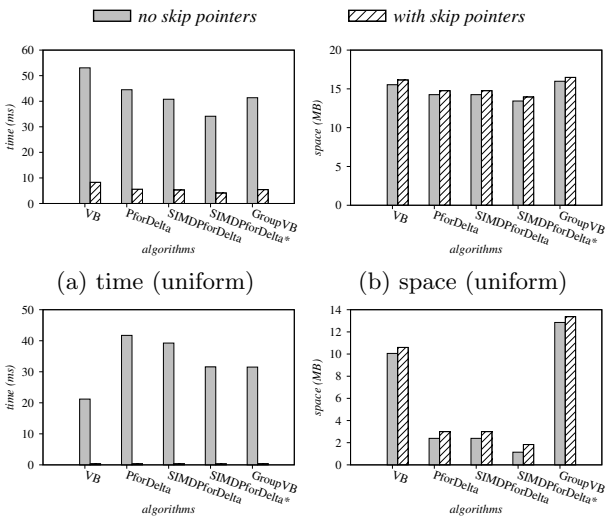
C. MORE EXPERIMENTS

In this section, we provide more experiments. Section C.1 evaluates the impact of building skip pointers for inverted list compression. Section C.2 evaluates the effect of list size ratios to intersection. Section C.3 presents the results on Graph data. Section C.4 presents the results on KDDCup data. Section C.5 presents the results on Berkeleyearth data. Section C.6 presents the results on Higgs data. Section C.7 presents the results on Kegg data.

C.1 Effect of the skip pointers

In this set of experiments, we evaluate the impact of skip pointers for inverted list compression. We pick up five in-

[13]https://github.com/lemire/FastPFor



(a) time (uniform) (b) space (uniform)

(c) time (zipf) (d) space (zipf)

Figure 7: The impact of skip pointers

	uniform		zipf		markov	
	$\theta = 1$	$\theta = 10$	$\theta = 1$	$\theta = 10$	$\theta = 1$	$\theta = 10$
Bitset	92.9	44.6	193.5	19.8	67.7	43.2
BBC	2655.6	1178.7	251.4	33.5	679.6	315.5
WAH	556.4	349.6	161.2	24.0	298.6	125.1
EWAH	683.9	422.7	171.5	18.8	351.0	136.7
PLWAH	632.8	756.8	255.7	23.7	367.0	161.8
CONCISE	978.6	355.2	148.8	44.4	370.8	162.9
VALWAH	1711.6	966.4	263.2	58.4	445.2	231.4
SBH	1994.8	669.8	156.7	25.9	526.1	265.6
Roaring	804.2	207.7	139.7	18.7	279.0	150.0
List	856.8	165.4	188.0	74.8	296.8	113.7
VB	1171.8	415.6	633.7	287.3	739.2	380.9
Simple9	1396.1	517.0	561.0	350.3	903.6	467.1
PforDelta	1384.4	452.3	725.1	512.9	773.7	395.2
NewPforDelta	1559.7	662.0	1028.9	510.0	1260.4	732.8
OptPforDelta	1684.9	724.5	1029.2	324.3	1285.4	763.3
Simple16	1573.0	601.9	567.1	233.2	950.5	492.0
GroupVB	1081.3	341.6	564.6	245.5	584.2	294.3
Simple8b	1287.2	444.7	577.2	164.6	803.0	403.4
PEF	2704.5	1240.9	1020.7	137.8	1536.2	802.3
SIMDPforDelta	1217.0	412.0	687.5	272.0	715.8	359.2
SIMDBP128	1054.8	320.6	505.2	261.2	536.1	260.8
PforDelta*	1087.2	336.9	517.1	265.4	574.1	276.1
SIMDPforDelta*	951.5	261.9	419.9	239.7	430.6	197.9
SIMDBP128*	901.0	233.9	369.7	507.6	378.3	168.7

Table 3: Evaluating intersection time (ms) with varying list size ratios

verted list compression algorithms (VB, PforDelta, SIMDPforDelta, SIMDPforDelta*, and GroupVB) to demonstrate the effect for intersection. Note that bitmap compression methods usually do not build skip pointers in order to perform bit-wise operations. Building skip pointers prevents the alignment of bitmaps. Figure 7 shows the results by setting $|L_2|$ as 10 million and $|L_2|/|L_1|$ as 1000. It shows that adding skip pointers increases the space overhead within 5% in most cases. However, it can dramatically improve the query performance, e.g., the improvement can be 8.3× on the uniform distribution and 124× on the zipf distribution.

C.2 Effect of the list size ratio

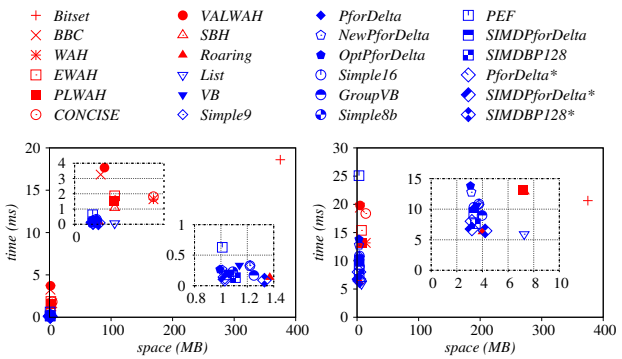
Next we investigate the effect of list size ratio. It is an important parameter that determines to what extent the skipping happens. We define the list size ratio $\theta = \frac{|L_2|}{|L_1|}$, set $|L_2|$ as 100 million, and vary θ from 1 to 10. Table 3 shows the results.

(1) In general, bitmap compression methods are faster than inverted list compression methods. That is because inverted list compression methods have to follow merge-based intersection when the list size ratio θ is 1 or 10. In this case, bitmap compression methods can leverage efficient bit-wise computations. This contradicts with the prior conclusions made in [8]. But bitmap compression methods tend to consume more space than inverted list compression methods due to the fill words in general.

(2) Among all the bitmap compression methods, Roaring is the fastest in general. An exception happens in the uniform distribution where the list size is 1 million, then Bitset is much faster than Roaring. That is because Bitset can directly perform efficient bit-wise operations but Roaring needs to handle the extra bucket alignment issue – finding out the promising buckets that intersect – before performing bit-wise operations.

(3) Among all the inverted list compression approaches, SIMDBP128* and SIMDPforDelta* are very competitive.

(4) It is worth noting that PEF turns out to be the slowest compression method because it has to decompress the entire list when θ is small.



(a) Q1 (b) Q2

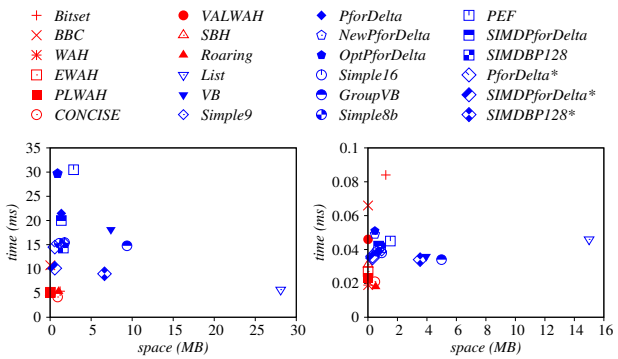
Figure 8: Results on Graph data

C.3 Results on Graph data

We next show the results on graph data, which is a subset of twitter dataset consisting of 52,579,682 vertices and 1,963,263,821 edges. Each list is an adjacency list of a vertex. We generate two intersection queries where each query involves 3 lists (L_1, L_2, and L_3). In Q1, $|L_1| = 960$, $|L_2| = 50913$, and $|L_3| = 507777$. In Q2, $|L_1| = 507777$, $|L_2| = 526292$, and $|L_3| = 779957$. Note that, the longest list is 779957, i.e., the one with the maximum degree. Figure 8 shows the results. Overall, inverted list compression methods outperform bitmap compression methods for Q1 and Q2. In particular, SIMDBP128* and SIMDPforDelta* are very competitive among all the compression methods.

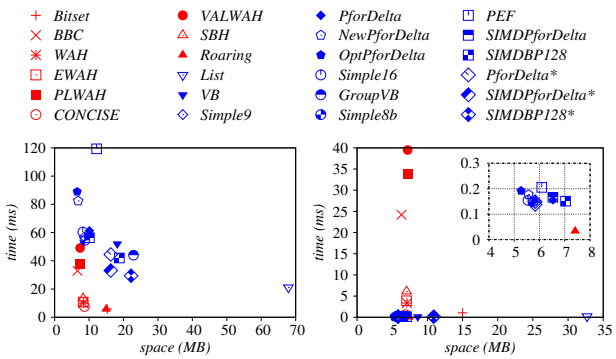
C.4 Results on KDDCup data

KDDCup is a dataset used for distinguishing the abnor-



(a) Q1 (b) Q2

Figure 9: Results on KDDCup data



(a) Q1 (b) Q2

Figure 10: Results on Berkeleyearth data

mal network connections.[14] The dataset is used in [20]. It contains 4,898,431 rows and 42 attributes. We choose two queries Q1 and Q2 with different sizes (i.e., selectivities). In which, Q1 and Q2 are two intersection queries that include two lists L_1 and L_2 but with different sizes. For Q1, $|L_1| = 2833545$ and $|L_2| = 4195364$. Since domain size is 4898431, thus, the selectivities are 0.58 and 0.86. For Q2, $|L_1| = 1051$ and $|L_2| = 3744328$, thus, the selectivities are 0.0002 and 0.76.

Figure 9 shows that, bitmap compression methods outperform inverted list compression methods on both Q1 and Q2. In particular, Roaring is the best among all the competitors. That is because for Q1, the lists are very dense such that bitmap compression methods can leverage efficient bit-wise computations. But for Q2, L_1 is extremely short compared with L_2 such that the space overhead is dominated by L_2, which is very long.

C.5 Results on Berkeleyearth data

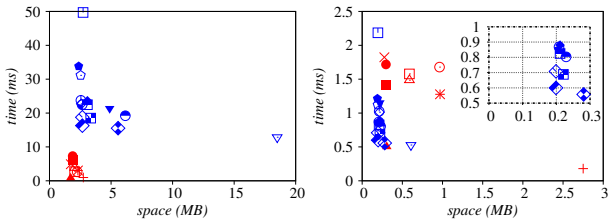
The Berkeleyearth data contains measurements from 1.6 billion temperature reports (used in [5]). We use a subset that contains 61,174,591 rows. We choose two queries where each query involves two lists (L_1 and L_2) but with different sizes. In Q1, $|L_1| = 7730307$ and $|L_2| = 9254744$. In Q2, $|L_1| = 5395$ and $|L_2| = 8174163$.

Figure 10 shows that for Q1, bitmap compression methods outperform inverted list compression methods since the two lists are dense. However for Q2, inverted list compression methods generally outperform bitmap compression methods (except for Roaring) since the lists are sparse. But Roaring is the fastest one in execution time.

C.6 Results on Higgs data

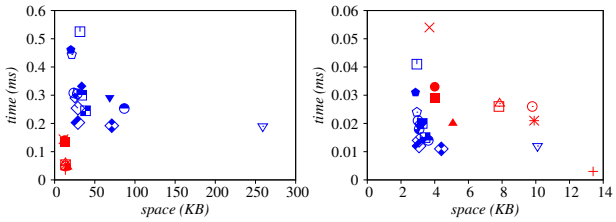
The Higgs dataset is a signal database that is used to determine whether a signal process produces Higgs bosons or not.[15] The dataset is used in [19]. It contains 11,000,000 rows. We generate two intersection queries where each query contains two lists (L_1 and L_2) but with different sizes. In Q1, $|L_1| = 172380$ and $|L_2| = 4446476$. In Q2, $|L_1| = 49170$ and $|L_2| = 102607$.

Figure 11 shows the results. For Q1, it clearly shows that Roaring is the best in terms of space overhead and intersection time. That is because the two lists are very dense. However, for Q2, SIMDBP128* and SIMDPforDelta* are the most competitive compression methods.



(a) Q1 (b) Q2

Figure 11: Results on Higgs data (legends can be found at Figure 10)



(a) Q1 (b) Q2

Figure 12: Results on Kegg data (legends can be found at Figure 10)

C.7 Results on Kegg data

The Kegg dataset is a database of KEGG Metabolic pathways that consists of 53414 records.[16] The dataset is used in [19]. We generate two intersection queries (Q1 and Q2) where each query involves two lists (L_1 and L_2). In Q1, $|L_1| = 16965$ and $|L_2| = 47783$. In Q2, $|L_1| = 1082$ and $|L_2| = 1438$.

Figure 12 shows the results. It shows that for Q1, Roaring and Bitset are the top two best compression methods in terms of space overhead and intersection time, since the lists are very dense. But for Q2, SIMDBP128* and SIMDPforDelta* are the top two best compression methods because the lists are sparse.

[14] https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[15] https://archive.ics.uci.edu/ml/datasets/HIGGS

[16] https://archive.ics.uci.edu/ml/datasets/KEGG+Metabolic+Relation+Network+(Directed)