An Experimental Study of Bitmap Compression vs. Inverted List Compression

Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, Steven Swanson
Presented By:
Abdullah Alomar

Outline

- Introduction & Motivation
- Bitmap Compression
- Inverted List Compression
- Empirical Evaluation
- Lessons Learned

Part 1:

Introduction & Motivation

Introduction.

- **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.
- The authors observe that they essentially solve the **same problem**,

  how to store a collection of sorted integers with as few as possible bits and support query processing as fast as possible.
- This work is a **comprehensive experimental study** to compare a series of **9** bitmap compression methods and **12** inverted list compression methods.

Bitmaps.

- Bitmaps have been widely adopted in modern **database systems** including both row-stores and column-stores, e.g., PostgreSQL, Microsoft SQL Server, Oracle, MonetDB..
- 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).
- **Example:**
- **Smartphones sales, where the "iPhone" appears at the 2nd, 5th, and 10th record in the phone_name column, then the bitmap of "iPhone" is** 0100100001000000..
- 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".

Inverted List.

- Inverted lists are the standard data structure in modern information retrieval (IR) systems.
- An inverted list is a set of document IDs for documents (e.g., web pages) that contain a certain term.
- **Example:**
- If the aforementioned database records are thought of as documents, then the inverted list is: {2, 5, 10}

- Note that that bitmaps can be converted to an inverted list: e.g. the bitmap 11001000010 can be converted to the inverted list {1,2, 5, 10}

The need for this empirical study

- As both methods do the same task; this begs the question of Which one is better?
- By “better”, we mean lower space and higher query performance.
- Also, In which scenario each method is more appropriate.
- This study aims to shed light into these questions.

Part 2:

Bitmap Compression

Bitmap compression

- In this section, we will review several bitmap compression methods**.**
- bitmap compression algorithms take as **input** an **uncompressed bitmap** and produce a **compressed bitmap** with as few as possible bits.
- Generally, these methods compress a sequence of identical bits with the bit value and its count (run-length encoding, RLE).
- But they differ in the way of handling the units of RLE (e.g., bytes or words), encoding the runs, and compressing the count.

WAH

- **WAH** (Word-Aligned Hybrid) partitions an uncompressed bitmap (input) into groups where each group contains 31 bits.
- Each group is either a:
  - **fill-group:** If all 31 bits are identical
  - **literal group.**
- WAH **Only** compress fill group.
- The compressed sequences consist of words, where the first bit in each word determines whether a group is a fill-group or not. The compression depends on whether they are fill groups or literal groups.

WAH

- **Example:**
  - **Input:** $10^{20}1^3\ 0^{111}1^{25}$: 164 bits (Where 0^{20} means twenty consecutive 0's).

1. **WAH partition inputs into Groups of 31 bits:** $G_1(1\ 0^{20}\ 1^3\ 0^7)$, $G_2(\ 0^{31})$, $G_3(\ 0^{31})$, $G_4(\ 0^{31})$, $G_5(\ 0^{11}\ 1^{20}\ )$, $G_6(\ 1^5\ )$
2. **For consecutive fill group** (G2, G3, G4), they are compressed together into one word, The **first bit (=1)** determines whether it is a fill group or not, and the **second bit** in a fill group determines their bit (0 in this case). The rest of the bits determine how many groups we have (011 in this case).
3. **For literal groups, they are** are written as is, except for an additional **first bit (=0)** which determines whether it is a fill group or not,.
4. The output sequence is as follow:

   G1 --> $(01\ 0^{20}\ 1^3\ 0^7)$; G2, G3, G4 --> $(100^{27}011)$; G5 --> $(0\ 0^{11}\ 1^{20})$; G6 --> $(0\ 0^{26}\ 1^5)$.

   $01\ 0^{20}\ 1^3\ 0^7\ 100^{27}0110\ 0^{11}\ 1^{20}\ 0\ 0^{26}\ 1^5$ (128 bits)

EWAH

- Enhanced WAH:
- EWAH divides an uncompressed bitmap (input) into **32-bit groups**.
- It encodes a sequence of **p (p ≤ 65535)** fill groups and **q (q ≤ 32767)** literal groups into a marker word followed by **q** literal words **(i.e. q+1 words)**. The **first bit** in the marker work determines which fill group type (1 or 0), the next 16 bits determine **the number of fill groups**, and the rest the **number of literal groups**.
- Using the same **example** before:
- **Input:** $10^{20}1^3\,0\,^{111}1^{25}$: 164 bits → $G_1(1\,0^{20}\,1^3\,0^8)$, $G_2(\,0^{32})$, $G_3(\,0^{32})$, $G_4(\,0^{32})$, $G_5(\,0^7\,1^{25}\,)$
- Output:
- G1 --> $0\,0^{16}\,0^{14}1$ $10^{20}\,1^3\,0^8$;
- G2, G3, G4, G5 --> $0\,0^{13}\,011\,0^{14}1$ $0^7 1^{25}$

  $0\,0^{16}\,0^{14}1$ $10^{20}\,1^3\,0^8$ $0\,0^{13}\,011\,0^{14}1$ $0^7 1^{25}$ (128 bits)

Concise

- CONCISE (Compressed N Composable Integer Set)
- Improve over WAH, where a literal group that has only one different bit is still compressed as fill group.
- Fill groups are encoded as follow: **The 1st bit** is set to 1. **The 2nd bit** indicates which fill group (0 or 1). The **following 5 bits** store the position of the odd bit, if any**. The remaining 25 bits store the number of fill groups minus one.**
- Using the same **example** before:
- **Input:** $0^{23}10\ ^{111}1^{25}$: 164 bits → $G_1(0^{23}\,1\,0^{7})$, $G_2(\,0^{31})$, $G_3(\,0^{31})$, $G_4(\,0^{31})$, $G_5(\,0^{11}\,1^{20}\,)$, $G_6(\,1^{5}\,)$

  Output:
- G1 to G4 --> **1000111$0^{22}$011**
- G5 --> 0 $0^{11}\,1^{20}$ G6 --> 0 $0^{26}\,1^{5}$.
- **1000111$0^{22}$011** 0 $0^{11}\,1^{20}$ 0 $0^{26}\,1^{5}$ (96 bits)

Other variants of WAH

- **PLWAH:** Which is very similar to CONCISE, except that the process of storing the mixed fill group is slightly different.
- **VALWAH (Variable-Aligned Length WAH) :** Which encodes different bitmaps using different segment lengths to minimize the space overhead.
- **SBH (**Super Byte-aligned Hybrid): Which is very similar to WAH, except that it is a byte-aligned method. It can encode a sequence of consecutive k ($k \leq 4093$) fill groups (Bytes) into one or two bytes.

Roaring

- This method is not based on run-length encoding.
- Roaring partitions the entire domain [0, n) into different buckets of range 2^{16} . (i.e., all elements in the same chunk share the same 16 most significant bits)
- If a chunk has > 4096, Roaring uses a 65536-bit uncompressed bitmap to encode the elements;
- otherwise, it uses a sorted array of 16-bit short integers.

BBC

- BBC (Byte-aligned Bitmap Code) is one of the earliest bitmap compression algorithms.
- BBC partitions the input into bytes, as shown before, each byte is either a. fill byte or a literal byte.
- Unlike WAH and SBH, BBC compresses a collection of such bytes by identifying different patterns (or cases) and encodes each pattern individually to save space.

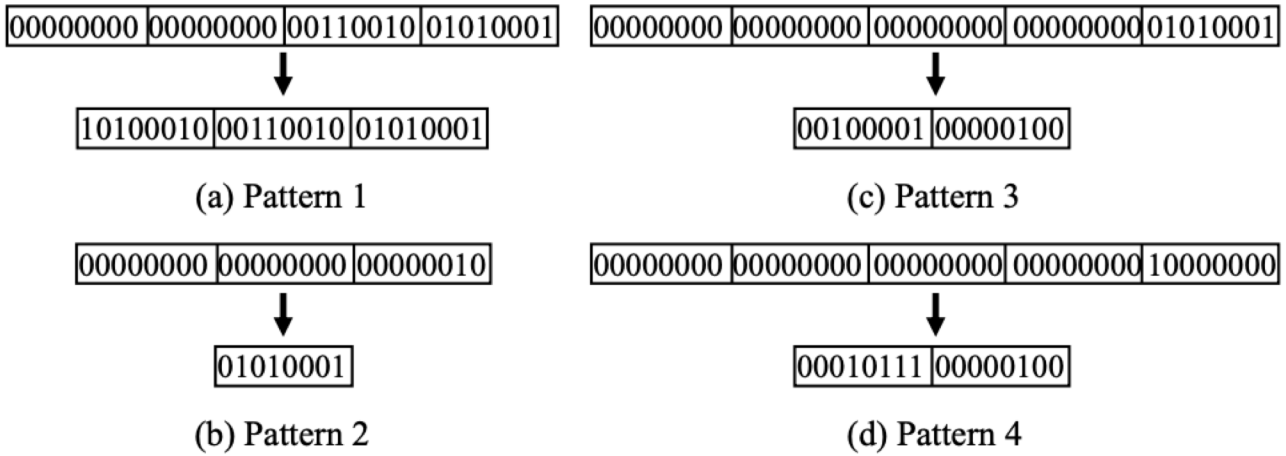

Figure 2: An example of BBC

Part 3:

Inverted List Compression

Inverted List Compression

In this section, we will review several **Inverted List** compression methods**.**

- Inverted list compression approaches usually follow a common approach of computing the deltas (a.k.a d-gaps) between two consecutive integers first and then compress the result.
- To speed up decompression time, the d-gaps are organized into blocks (e.g.128 elements per block) and builds a skip pointer per block such that only a block of data needs to be decompressed.
- The d-gaps are essentially bit-level run-length encoding. Hence, it cannot benefit from bit-wise operations but can benefit from efficient data skipping due to skip pointers.

Inverted List Compression

VB

- Variable size (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 as a flag bit to indicate whether the next byte belongs to this integer.
- For example: 100000000000001 → **1**0000001 **1**0000000 **0**0000001.

GroupVB

- Group Varint Encoding (**GroupVB**) compresses four values at the same time and it uses a 2-bit flag for each value.
- 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.

Inverted List Compression

PforDelta

- PForDelta compresses a block of 128 d-gaps by choosing the smallest number of bits **b** that most elements can be encoded into.
- It 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).

NewPforDelta

- NewPforDelta is a variant of PforDelta that reduce the space overhead of PforDelta. The main difference is in how NewPforDelta handles the storage of the exceptions

OPTforDelta

- OPTforDelta is another variant of PforDelta. The main difference is that OptPforDelta uses the optimal number of bits **b**.

Inverted List Compression

Simple9

- Simple9 is a word-aligned method that packs 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 ,…, or 1 × 28-bit number.
- **Example:** if the next 14 values are all less than 4, then Simple9 stores them as 14 × 2-bit values.

Simple16

- The same as Simple9 with 16 combinations.

Simple8b

- The same as Simple9 but it extends the codeword into 64 bits.

Inverted List Compression

PEF

- PEF (Partitioned Elias Fano) is different from other inverted list compression algorithms as it is not based on d-gaps.
- It is an improved version of Elias Fano Encoding.

SIMDPforDelta

- SIMDPforDelta is the SIMD version of PforDelta. It leverages modern CPU's SIMD instructions to accelerate the query performance and also decompression speed.
- The main idea of SIMDPforDelta is to reorganize data elements in a way such that a single SIMD operation processes multiple elements

Inverted List Compression

SIMDBP128

- SIMDBP128 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.

Part 4:

Empirical Evaluation

EXPERIMENTAL SETUP

Evaluation metrics

- Each compression algorithm is measured via 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. 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.

Results on Synthetic Datasets

Synthetic datasets.

- Three synthetic datasets were generated following:

1. A uniform distribution, where each integer have the same probability.
2. zipf distribution, where each integer k included with a probability $\propto \frac{1}{k^f}$
3. Markov chain, with a certain transition probability between 0-1 and 1-0

Results on Synthetic Datasets

Decompression Results

- Figure below show time and space overhead on all synth. data.
- The list size is varied from 1 million to 1 billion.

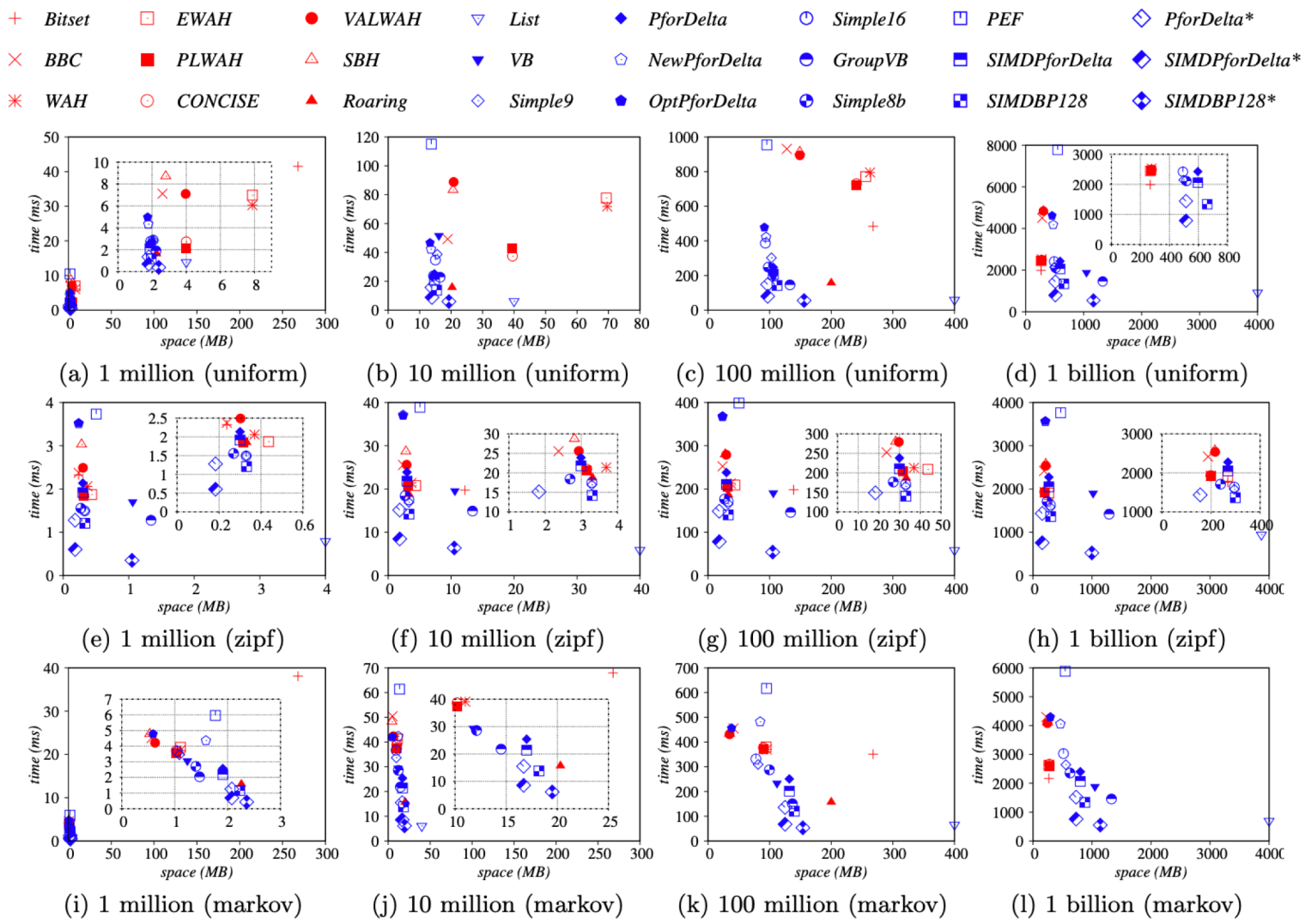

Figure 3: Evaluating decompression with varying list sizes

Results on Synthetic Datasets

Decompression Results

1. Bitmap compression methods incur more space and higher decompression overhead than inverted list compression methods. For uniform and Markov, when the list size is 1 billion bitmaps consume less space, otherwise inverted lists fare better.

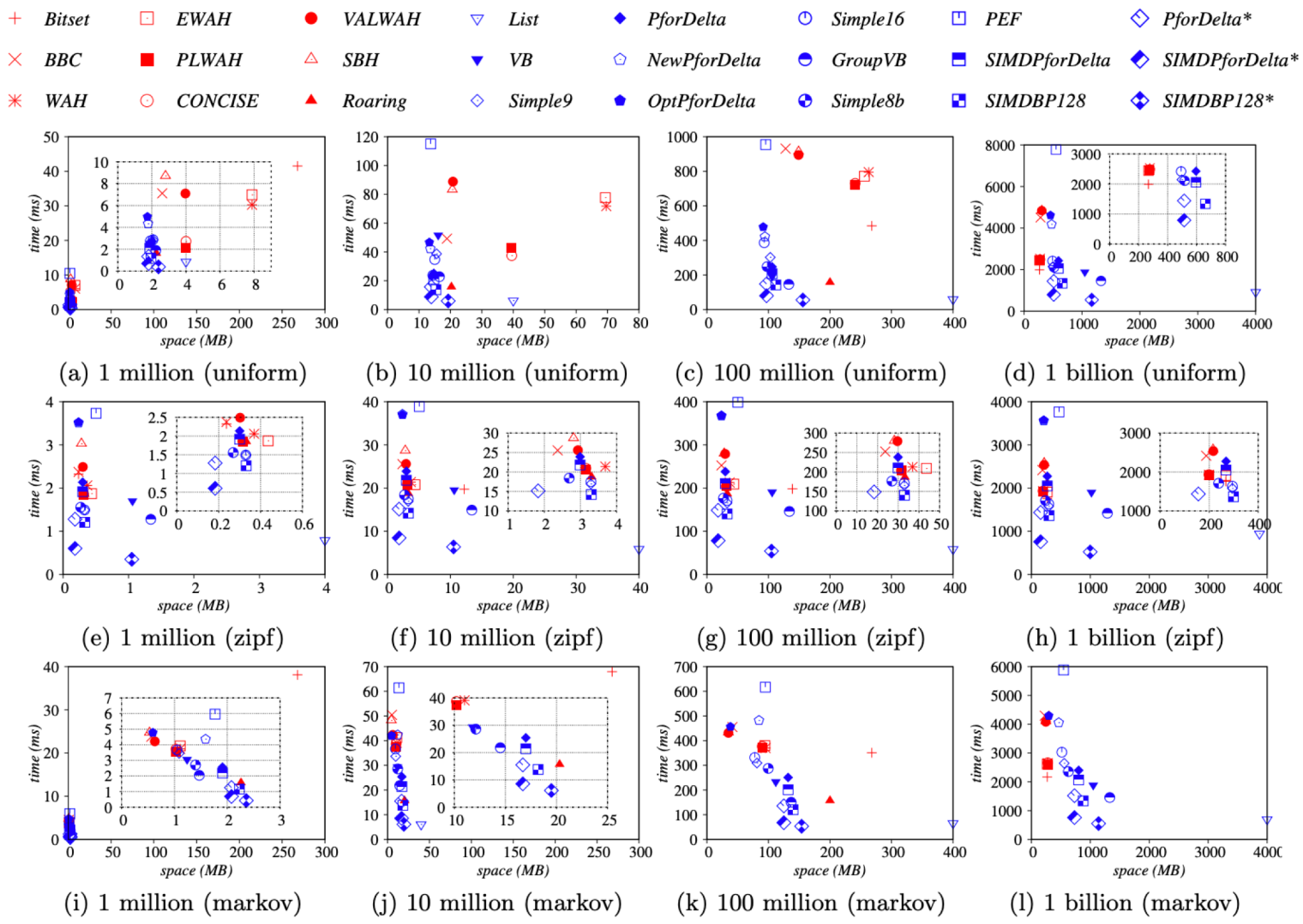

Figure 3: Evaluating decompression with varying list sizes

Results on Synthetic Datasets

Decompression Results

2. Bitmap 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.

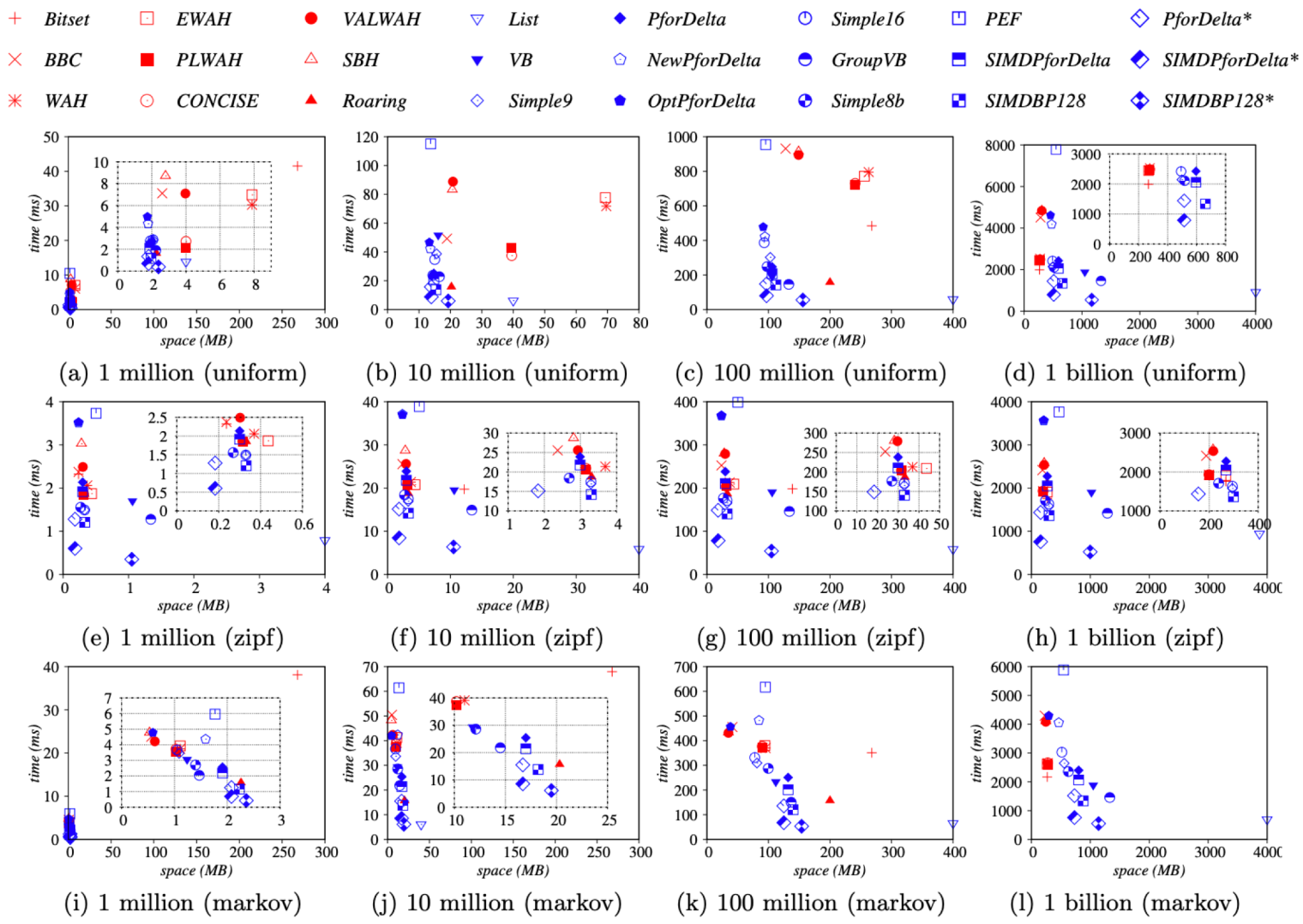

Figure 3: Evaluating decompression with varying list sizes

Results on Synthetic Datasets

Decompression Results

3. Among 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. 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. Bitset only works well when it is very dense because its size as well as performance depends on the maximal element in the list.
6. For BBC, its space overhead is almost the smallest, but its decompression speed is not excellent because it needs to handle many complicated cases.

Results on Synthetic Datasets

Intersection Results

1. In general, Roaring achieves the fastest intersection, as it only intersects two promising buckets, i.e., two buckets sharing the same bucket number.
2. Among all the inverted list compression methods, PEF and SIMDBP128* are the most efficient algorithms.

	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

Results on Synthetic Datasets

Union Results

1. In general, inverted list compression methods are faster than bitmap compression methods
2. Among all the bitmap compression methods, Roaring is the best in almost all cases in terms of union time.

	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

Results on Real world Datasets

- The authors present the results on 8 real datasets.
- The results have similar conclusion to the synthetic datasets.
- I omit them here for time constraints.

Part 5:

Lessons Learned

Summary

- **Space overhead:** Inverted list compression methods generally take less space than bitmap compression methods unless the list L is ultra dense.
- **Decompression time**. Inverted list compression methods are generally faster than bitmap compression methods in nearly all cases.
- **Intersection time**. In general, Roaring bitmap achieves the fastest intersection performance among all the compression methods.
- **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.

Lessons Learned

1. Although database systems preferred bitmap compression and information retrieval systems preferred inverted list compression, it does not mean that bitmap compression is always better than inverted list compression or vice versa.
2. This work shows that Bitmaps are suitable for low-cardinality columns via compression and the compression method should be Roaring.
3. Use Roaring for bitmap compression whenever possible. Do not use other bitmap compression methods.
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 if you're concerned about implementation overhead. As VB is perhaps the simplest one to implement.
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.

Thank you!