03: Compression

Andrew Crotty // CS497 // Fall 2023

Real-World Data Characteristics

Datasets tend to have highly **skewed** distributions for attribute values.
→ Example: Zipfian distribution of the Brown Corpus

Data sets tend to have high **correlation** between attributes of the same tuple.
→ Example: Zip Code to City, Order Date to Ship Date

Observation

I/O is (traditionally) the main bottleneck during query execution. If the DBMS still needs to read data, we need to ensure that it maximizes the amount of useful work per I/O.

Key trade-off is **speed** vs. **compression ratio**

→ Compressing the data reduces DRAM requirements and processing.

→ But then it is slower to decompress/process.

Today's Agenda

Background
Naive Page Compression
Native Columnar Compression
Intermediate Data

Database Compression

Reduce the size of the physical representation of the DB to increase the # of values accessed and processed per unit of computation or I/O.

Goal #1: Ideally produce fixed-length values.

Goal #2: Should be a **lossless** scheme.

Goal #3: Ideally postpone decompression for as long as possible during query execution.

Lossless vs. Lossy Compression

When a DBMS uses compression, it should be **lossless** because people don't like losing data.

Any kind of **lossy** compression should be performed at the application level.

Reading less than the entire dataset during query execution is sort of like compression.
→ Approximate Query Processing

Database Compression

If we want to add compression to our DBMS, the first question we must ask ourselves is what we want to compress.

This determines what compression schemes are available to us…

Compression Granularity

Choice #1: Block-level

→ Compress a block (e.g., database page, RowGroup, single tuple) of tuples in a table.

Choice #2: Column-level

→ Compress multiple values for one or more attributes stored for multiple tuples (DSM-only).

Naive Compression

Compress data using a general-purpose algorithm. Scope of compression is only based on the data provided as input.
→ LZO (1996), LZ4 (2011), Snappy (2011), Brotli (2013), Oracle OZIP (2014), Zstd (2015)

Considerations
→ Computational overhead
→ Compress vs. decompress speed

Naive Compression

Compress data using a general-purpose algorithm. Scope of compression is only based on the data provided as input.
→ LZO (1996), LZ4 (2011), Snappy (2011), Brotli (2013), Oracle OZIP (2014), Zstd (2015)

Considerations
→ Computational overhead
→ Compress vs. decompress speed

Naive Compression

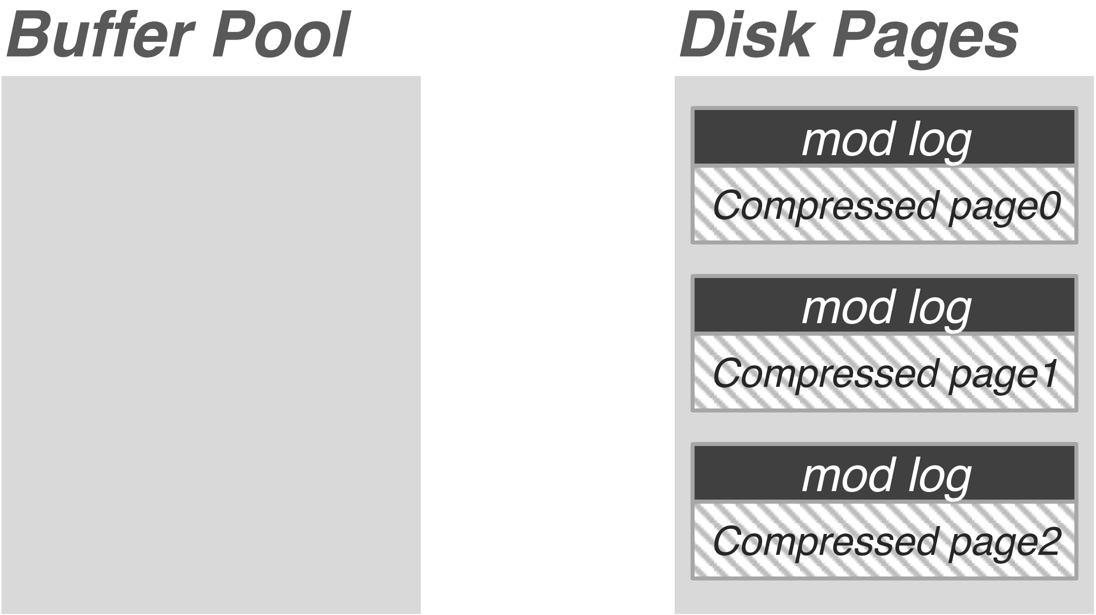
Choice #1: Entropy Encoding

→ More common sequences use less bits to encode, less common sequences use more bits to encode.

Choice #2: Dictionary Encoding

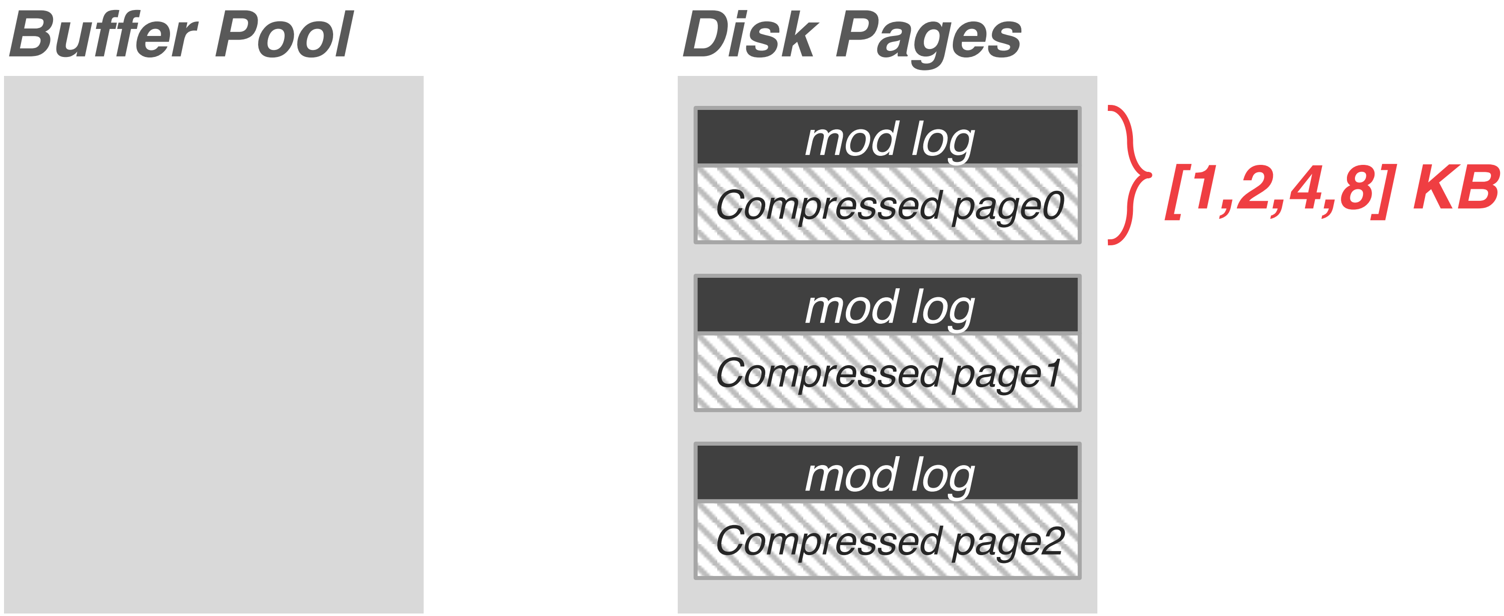
→ Build a data structure that maps data segments to an identifier. Replace those segments in the original data with a dictionary code.

MySQL InnoDB Compression



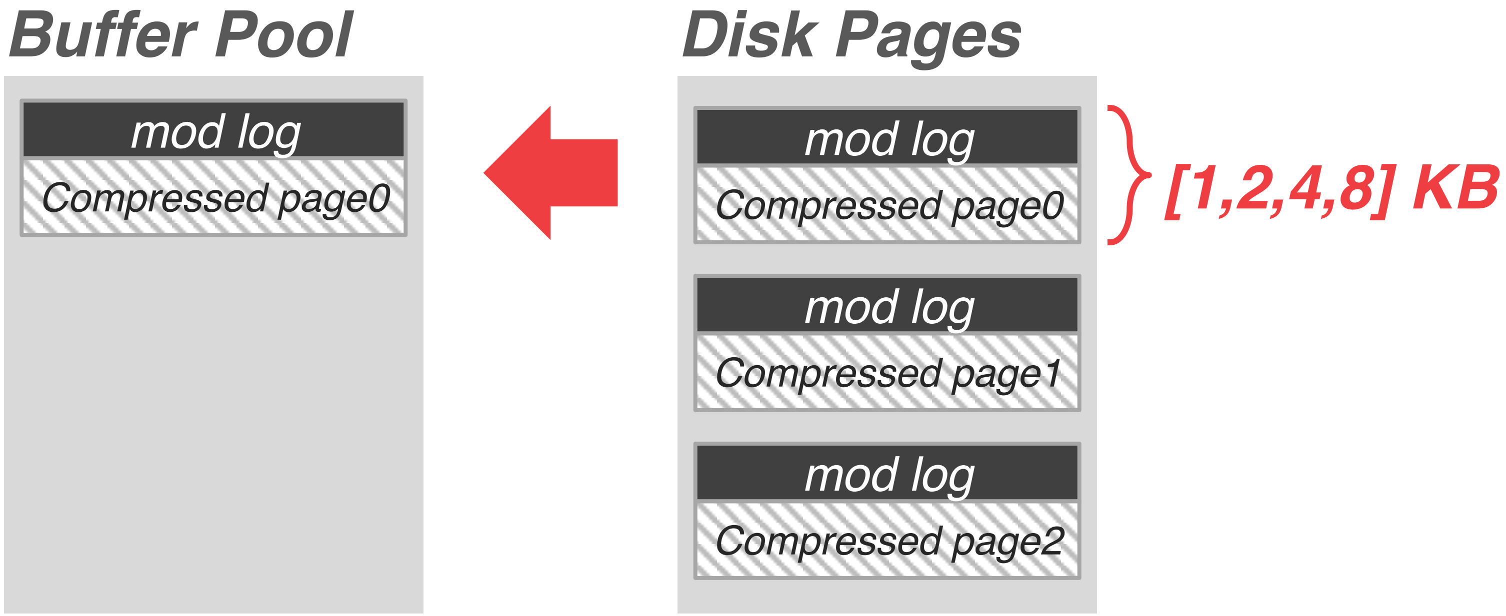
Source: [MySQL 5.7 Documentation]

MySQL InnoDB Compression



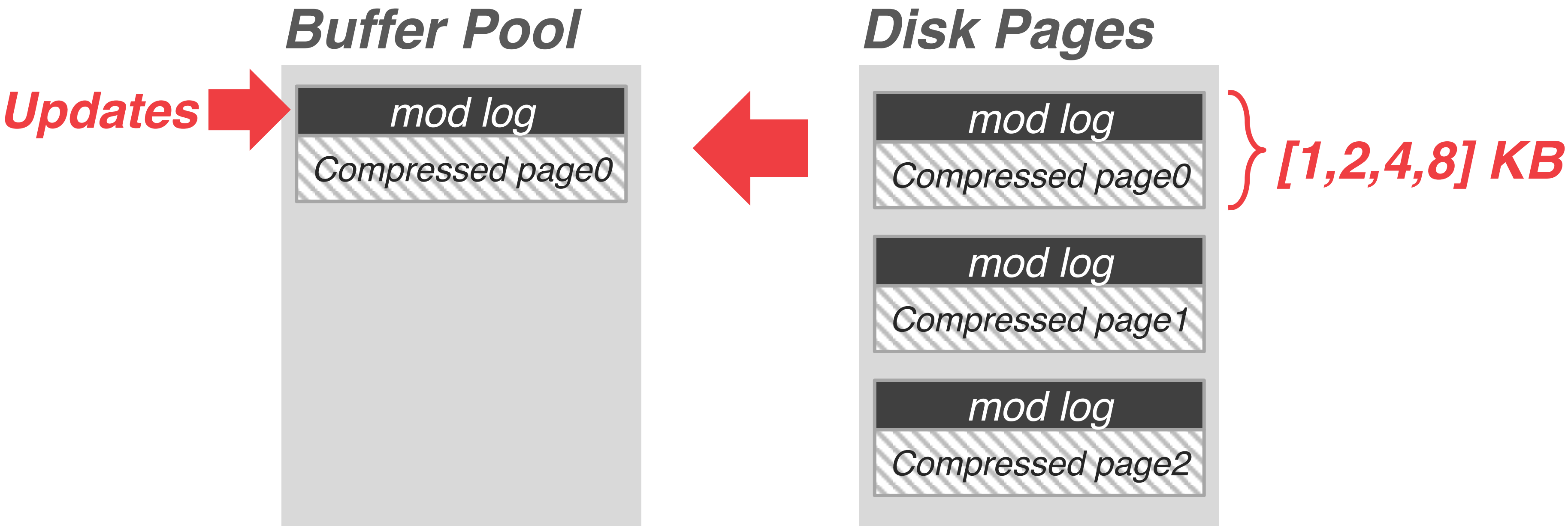
Source: [MySQL 5.7 Documentation]

MySQL InnoDB Compression



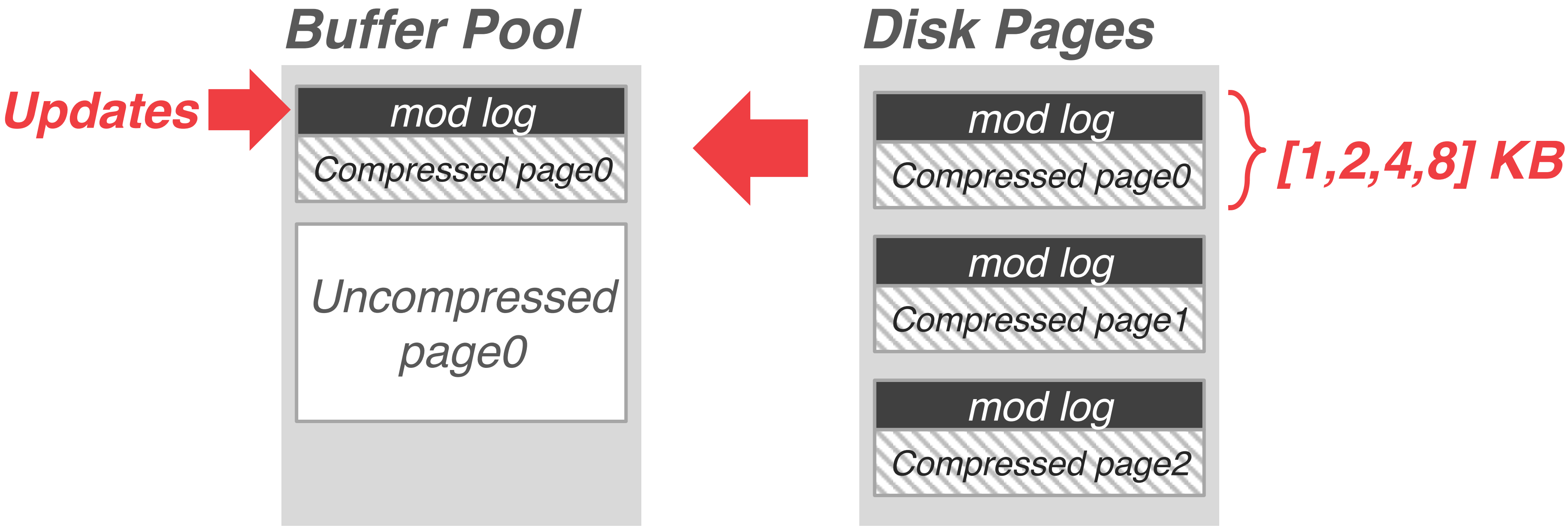
Source: [MySQL 5.7 Documentation]

MySQL InnoDB Compression



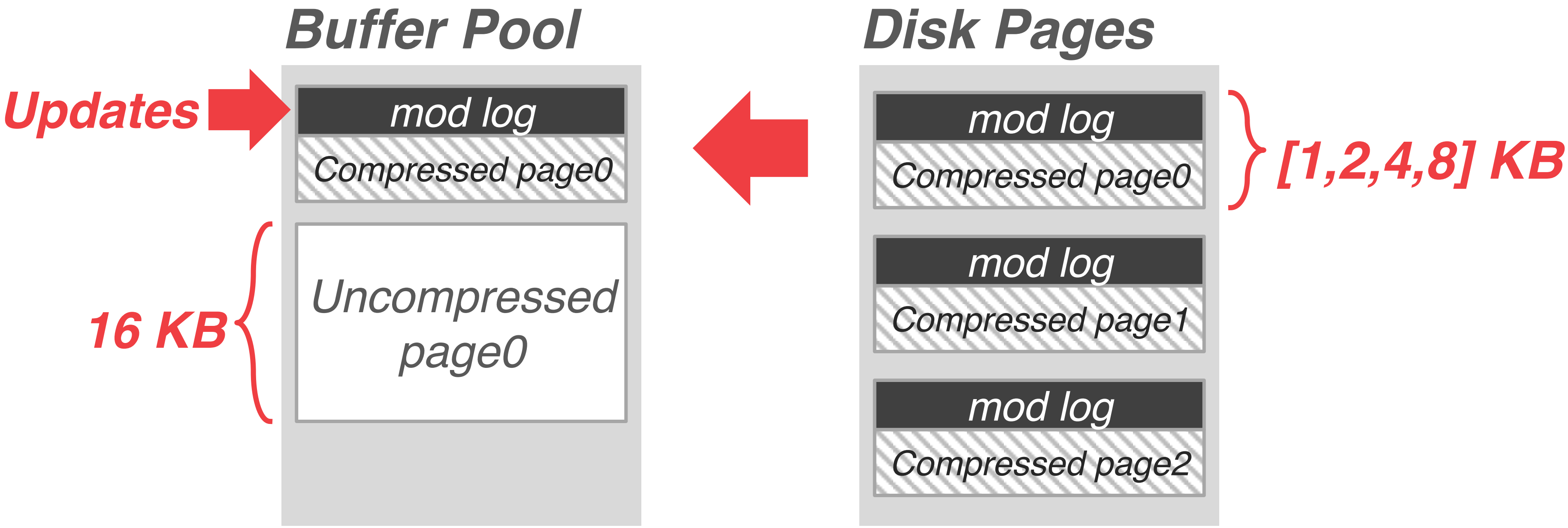
Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



Source: [MySQL 5.7 Documentation]

MySQL InnoDB Compression



Source: [MySQL 5.7 Documentation]()

Naive Compression

The DBMS must decompress data first before it can be read and (potentially) modified.
→ Even if the algo uses dictionary compression, the DBMS cannot access the dictionary's contents.
→ This limits the practical scope of the compression scheme.

These schemes also do not consider the high-level meaning or semantics of the data.

Observation

The DBMS can evaluate predicates without having to decompress tuples first if predicates and data are compressed the same way.

Observation

The DBMS can evaluate predicates without having to decompress tuples first if predicates and data are compressed the same way.

```
SELECT * FROM users
 WHERE name = 'Alice'
```

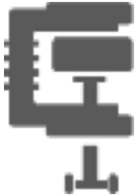
NAME	SALARY
Alice	99999
Bob	88888

Observation

The DBMS can evaluate predicates without having to decompress tuples first if predicates and data are compressed the same way.

```
SELECT * FROM users
 WHERE name = 'Alice'
```

NAME	SALARY
Alice	99999
Bob	88888



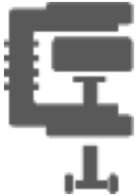
NAME	SALARY
XX	AA
YY	BB

Observation

The DBMS can evaluate predicates without having to decompress tuples first if predicates and data are compressed the same way.

SELECT * FROM users
WHERE name = 'Alice'

NAME	SALARY
Alice	99999
Bob	88888



SELECT * FROM users
WHERE name = XX

NAME	SALARY
XX	AA
YY	BB

Columnar Compression

Run-length Encoding

Dictionary Encoding

Bitmap Encoding

Delta Encoding

Bit Packing

Run-Length Encoding

Compress runs of the same value in a single column into triplets:
→ The value of the attribute.
→ The start position in the column segment.
→ The # of elements in the run.

Requires the columns to be sorted intelligently to maximize compression opportunities.

DATABASE COMPRESSION
SIGMOD RECORD 1993

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

id	lit
1	(Y,0,3)
2	(N,3,1)
3	(Y,4,1)
4	(N,5,1)
6	(Y,6,2)
7	
8	
9	

RLE Triplet
- ***Value***
- ***Offset***
- ***Length***

Run-Length Encoding

```
SELECT lit, COUNT(*)
  FROM users
 GROUP BY lit
```

Compressed Data

id
1
2
3
4
6
7
8
9

lit
(Y,0,3)
(N,3,1)
(Y,4,1)
(N,5,1)
(Y,6,2)

RLE Triplet
- *Value*
- *Offset*
- *Length*

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

id	lit
1	(Y,0,3)
2	(N,3,1)
3	(Y,4,1)
4	(N,5,1)
6	(Y,6,2)
7	
8	
9	

RLE Triplet
- *Value*
- *Offset*
- *Length*

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

id	lit
1	(Y,0,3)
2	(N,3,1)
3	(Y,4,1)
4	(N,5,1)
6	(Y,6,2)
7	
8	
9	

RLE Triplet
- Value
- Offset
- Length

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
6	Y
8	Y
9	Y
4	N
7	N

Compressed Data

Run-Length Encoding

Original Data

id	lit
1	Y
2	Y
3	Y
6	Y
8	Y
9	Y
4	N
7	N

Compressed Data

id
1
2
3
6
8
9
4
7

lit
(Y,0,6)
(N,7,2)

Dictionary Compression

Replace frequent values with smaller fixed-length codes and maintain a mapping (dictionary) from codes to the original values.

→ Typically, one code per attribute value.

→ Most widely used columnar compression scheme.

The ideal dictionary scheme supports fast encoding and decoding for both point and range queries.

INTEGRATING COMPRESSION AND EXECUTION IN COLUMN-ORIENTED DATABASE SYSTEMS
SIGMOD 2006

Dictionary Compression

When to construct the dictionary?

What is the scope of the dictionary?

What data structure do we use for the dictionary?

What encoding scheme to use for the dictionary?

Dictionary Construction

Choice #1: All-At-Once

→ Compute the dictionary for all the tuples at a given point of time.
→ New tuples must use a separate dictionary, or all tuples must be recomputed.
→ This is easy to do if the file is immutable.

Choice #2: Incremental

→ Merge new tuples in with an existing dictionary.
→ Likely requires re-encoding to existing tuples.

Dictionary Construction

Choice #1: All-At-Once

→ Compute the dictionary for all the tuples at a given point of time.
→ New tuples must use a separate dictionary, or all tuples must be recomputed.
→ This is easy to do if the file is immutable.

Choice #2: Incremental

→ Merge new tuples in with an existing dictionary.
→ Likely requires re-encoding to existing tuples.

Dictionary Scope

Choice #1: Block-level
→ Only include a subset of tuples within a single table.
→ DBMS must decompress data when combining tuples from different blocks (e.g., hash table for joins).

Choice #2: Table-level
→ Construct a dictionary for the entire table.
→ Better compression ratio, but expensive to update.

Choice #3: Multi-Table
→ Can be either subset or entire tables.
→ Sometimes helps with joins and set operations.

Dictionary Scope

Choice #1: Block-level

→ Only include a subset of tuples within a single table.
→ DBMS must decompress data when combining tuples from different blocks (e.g., hash table for joins).

Choice #2: Table-level

→ Construct a dictionary for the entire table.
→ Better compression ratio, but expensive to update.

Choice #3: Multi-Table

→ Can be either subset or entire tables.
→ Sometimes helps with joins and set operations.

Encoding / Decoding

Encode/Locate: For a given uncompressed value, convert it into its compressed form.

Decode/Extract: For a given compressed value, convert it back into its original form.

No magic hash function will do this for us.

Order-Preserving Encoding

The encoded values need to support sorting in the same order as original values.

Original Data

name
Andrea
Sarah
Andrew
Matt

Order-Preserving Encoding

The encoded values need to support sorting in the same order as original values.

Original Data

name
Andrea
Sarah
Andrew
Matt

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

The encoded values need to support sorting in the same order as original values.

```
SELECT * FROM users
 WHERE name LIKE 'And%'
```

Original Data

name
Andrea
Sarah
Andrew
Matt

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

The encoded values need to support sorting in the same order as original values.

```
SELECT * FROM users
 WHERE name LIKE 'And%'
```

```
SELECT * FROM users
 WHERE name BETWEEN 10 AND 20
```

Original Data

name
Andrea
Sarah
Andrew
Matt

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

Original Data

name
Andrea
Sarah
Andrew
Matt

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

```
SELECT name FROM users
 WHERE name LIKE 'And%'
```

???

Original Data

name
Andrea
Sarah
Andrew
Matt

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

```
SELECT name FROM users
WHERE name LIKE 'And%'
```

Still must perform seq scan

Original Data

name
Andrea
Sarah
Andrew
Matt

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

```
SELECT name FROM users
 WHERE name LIKE 'And%'
```

→ ***Still must perform seq scan***

```
SELECT DISTINCT name
  FROM users
 WHERE name LIKE 'And%'
```

???

Original Data

name
Andrea
Sarah
Andrew
Matt

→

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Order-Preserving Encoding

```
SELECT name FROM users
 WHERE name LIKE 'And%'
```

→ ***Still must perform seq scan***

```
SELECT DISTINCT name
  FROM users
 WHERE name LIKE 'And%'
```

→ ***Only need to access dictionary***

Original Data

name
Andrea
Sarah
Andrew
Matt

→

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andrew	20
Matt	30
Sarah	40

Dictionary Data Structures

Choice #1: Array

→ One array of variable length strings and another array with pointers that maps to string offsets.
→ Expensive to update so only usable in immutable files.

Choice #2: Hash Table

→ Fast and compact.
→ Unable to support range and prefix queries.

Choice #3: B+Tree

→ Slower than a hash table and takes more memory.
→ Can support range and prefix queries.

Dictionary: Array

First sort the values and then store them sequentially in an array of bytes.
→ Also need to store the size of the value if they are variable-length.

Replace the original data with dictionary codes that are the (byte) offset into this array.

Original Data

name
Andrea
Sarah
Andrew
Matt

INTEGRATING COMPRESSION AND EXECUTION IN COLUMN-ORIENTED DATABASE SYSTEMS
SIGMOD 2006

Dictionary: Array

First sort the values and then store them sequentially in an array of bytes.

→ Also need to store the size of the value if they are variable-length.

Replace the original data with dictionary codes that are the (byte) offset into this array.

Original Data

name
Andrea
Sarah
Andrew
Matt

len|val

6|Andrea

6|Andrew

4|Matt

5|Sarah

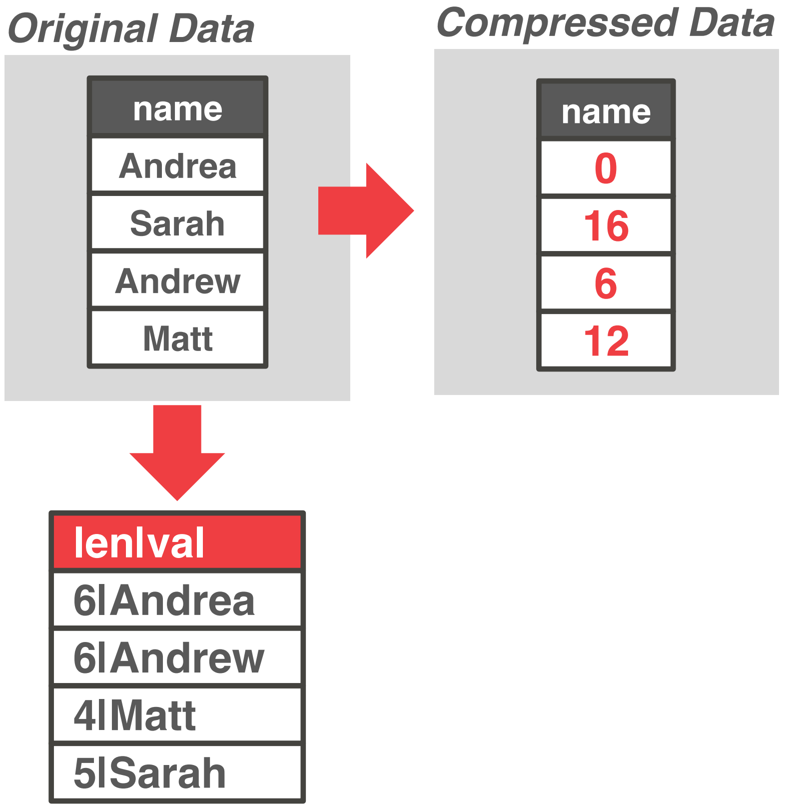
INTEGRATING COMPRESSION AND EXECUTION IN COLUMN-ORIENTED DATABASE SYSTEMS
SIGMOD 2006

Dictionary: Array

First sort the values and then store them sequentially in an array of bytes.

→ Also need to store the size of the value if they are variable-length.

Replace the original data with dictionary codes that are the (byte) offset into this array.



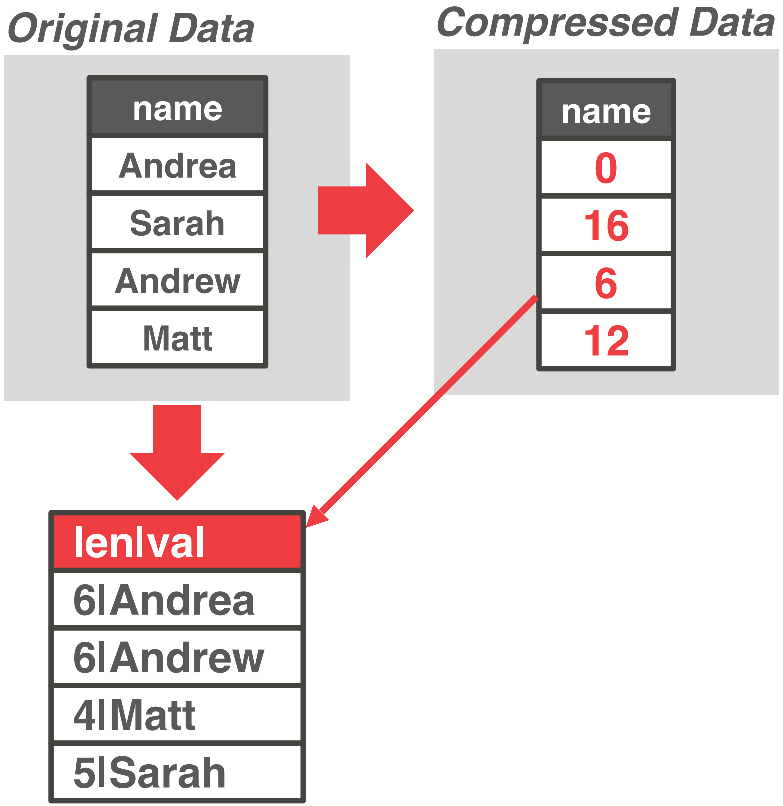
INTEGRATING COMPRESSION AND EXECUTION IN COLUMN-ORIENTED DATABASE SYSTEMS
SIGMOD 2006

Dictionary: Array

First sort the values and then store them sequentially in an array of bytes.

→ Also need to store the size of the value if they are variable-length.

Replace the original data with dictionary codes that are the (byte) offset into this array.



INTEGRATING COMPRESSION AND EXECUTION IN COLUMN-ORIENTED DATABASE SYSTEMS
SIGMOD 2006

Exposing Dictionaries

Parquet / ORC do not provide an API to directly access a file's compression dictionary.

This means the DBMS cannot perform predicate pushdown and operate directly on compressed data before decompressing it.

Google's Artus proprietary format for Procella supports this.

Columnar Compression

~~Run-length Encoding~~

~~Dictionary Encoding~~

Bitmap Encoding

Delta Encoding

Bit Packing

Bitmap Encoding

Bitmap encoding can reduce a column's storage size if its cardinality is low.

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Bitmap Encoding

Bitmap encoding can reduce a column's storage size if its cardinality is low.

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Bitmap Encoding

Bitmap encoding can reduce a column's storage size if its cardinality is low.

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

	lit	
id	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

Bitmap Encoding

Bitmap encoding can reduce a column's storage size if its cardinality is low.

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

	lit	
id	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

Bitmap Encoding

Bitmap encoding can reduce a column's storage size if its cardinality is low.

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

$$\left.\begin{matrix}\\\\\\\\\\\\\\\\\end{matrix}\right\}\ 9 \times 8\text{-bits} = 72 \text{ bits}$$

Compressed Data

	lit	
id	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

Bitmap Encoding

Bitmap encoding can reduce a column's storage size if its cardinality is low.

Original Data

id	lit
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

9 × 8-bits = 72 bits

Compressed Data

	lit	
id	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

2 × 8-bits = 16 bits

9 × 2-bits = 18 bits

Compressing Bitmaps

Approach #1: Block-based Compression
→ Use standard compression algorithms (e.g., Snappy, zstd).
→ Must decompress entire data chunk before DBMS can use it to process a query.

Approach #2: Byte-aligned Bitmap Codes
→ Structured run-length encoding compression.

Approach #3: Roaring Bitmaps
→ Hybrid of run-length encoding and value lists.

Oracle Byte-Aligned Bitmap Codes

Divide bitmap into chunks that contain different categories of bytes:
→ **Gap Byte**: All the bits are **0**s.
→ **Tail Byte:** Some bits are **1**s.

Encode each <u>**chunk**</u> that consists of some **Gap Bytes** followed by some **Tail Bytes**.
→ Gap Bytes are compressed with RLE.
→ Tail Bytes are stored uncompressed unless it consists of only 1-byte or has only one non-zero bit.

BYTE-ALIGNED BITMAP COMPRESSION
DATA COMPRESSION CONFERENCE 1995

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	01000000	00100010

Compressed Bitmap

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	01000000	00100010

Compressed Bitmap

Source: [Brian Babcock]

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000	#1
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	01000000	00100010	

Compressed Bitmap

Source: [Brian Babcock]

Oracle Byte-Aligned Bitmap Codes

Bitmap

	Gap Bytes	*Tail Bytes*	
00000000	00000000	00010000	#1
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	01000000	00100010	

Compressed Bitmap

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

	Gap Bytes	*Tail Bytes*	
00000000	00000000	00010000	#1
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	#2
00000000	00000000	00000000	
00000000	01000000	00100010	

Compressed Bitmap

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000	#1
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	01000000	00100010	

Compressed Bitmap

Chunk #1 (Bytes 1-3)

Header Byte:
- → Number of Gap Bytes (Bits 1-3)
- → Is the tail special? (Bit 4)
- → Number of verbatim bytes (if Bit 4=0)
- → Index of one-bit in tail byte (if Bit 4=1)

No gap length bytes since gap length < 7

No verbatim bytes since tail is special

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000	#1
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	00000000	00000000	
00000000	01000000	00100010	

Compressed Bitmap

#1 (010)(1)(0100)
1-3 4 5-7

Chunk #1 (Bytes 1-3)

Header Byte:

→ Number of Gap Bytes (Bits 1-3)
→ Is the tail special? (Bit 4)
→ Number of verbatim bytes (if Bit 4=0)
→ Index of one-bit in tail byte (if Bit 4=1)

No gap length bytes since gap length < 7

No verbatim bytes since tail is special

Source: [Brian Babcock]

Oracle Byte-Aligned Bitmap Codes

Bitmap

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

Compressed Bitmap

#1(010)(1)(0100)

Chunk #2 (Bytes 4-18)

Header Byte:
→ 13 gap bytes, two tail bytes
→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	01000000	00100010

Compressed Bitmap

#1 (010)(1)(0100)

#2 $\boxed{(111)}$(0)(0010)00001101
01000000 00100010

Chunk #2 (Bytes 4-18)

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

Compressed Bitmap

#1 (010)(1)(0100)

Gap Length

#2 (111)(0)(0010) $\boxed{00001101}$
01000000 00100010

Chunk #2 (Bytes 4-18)

Header Byte:
→ 13 gap bytes, two tail bytes
→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: [Brian Babcock]

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	01000000	00100010

Compressed Bitmap

#1 (010)(1)(0100)

#2 (111)(0)(0010)00001101
01000000 00100010

Chunk #2 (Bytes 4-18)

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

00000000	00000000	00010000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	01000000	00100010

Compressed Bitmap

#1 (010)(1)(0100)

#2 (111)(0)$\boxed{\text{(0010)}}$00001101
01000000 00100010

Chunk #2 (Bytes 4-18)

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

Oracle Byte-Aligned Bitmap Codes

Bitmap

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

Compressed Bitmap

#1 (010)(1)(0100)

#2 (111)(0)(0010)00001101
01000000 00100010

Verbatim Tail Bytes

Source: [Brian Babcock]

Chunk #2 (Bytes 4-18)

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Oracle Byte-Aligned Bitmap Codes

Bitmap

```
00000000  00000000  00010000
00000000  00000000  00000000
00000000  00000000  00000000
00000000  00000000  00000000
00000000  00000000  00000000
00000000  01000000  00100010
```

Compressed Bitmap

#1 (010)(1)(0100)

#2 (111)(0)(0010)00001101
01000000 00100010

Verbatim Tail Bytes

Source: [Brian Babcock]

Chunk #2 (Bytes 4-18)

Header Byte:
→ 13 gap bytes, two tail bytes
→ # of gaps is > 7, so must use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Original: 18 bytes

BBC Compressed: 5 bytes

Observation

Oracle's BBC is an obsolete format.
→ Although it provides good compression, it is slower than recent alternatives due to excessive branching.
→ Word-Aligned Hybrid (WAH) encoding is a patented variation on BBC that provides better performance.

None of these support random access.
→ If you want to check whether a given value is present, you must start from the beginning and decompress the whole thing.

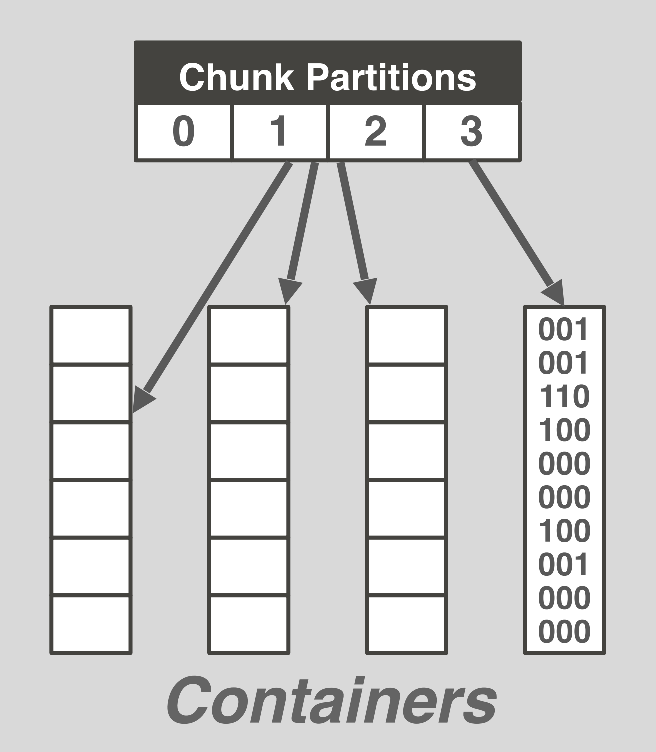
Roaring Bitmaps

Store 32-bit integers in a compact two-level indexing data structure.
→ Dense chunks are stored using bitmaps
→ Sparse chunks use packed arrays of 16-bit integers.

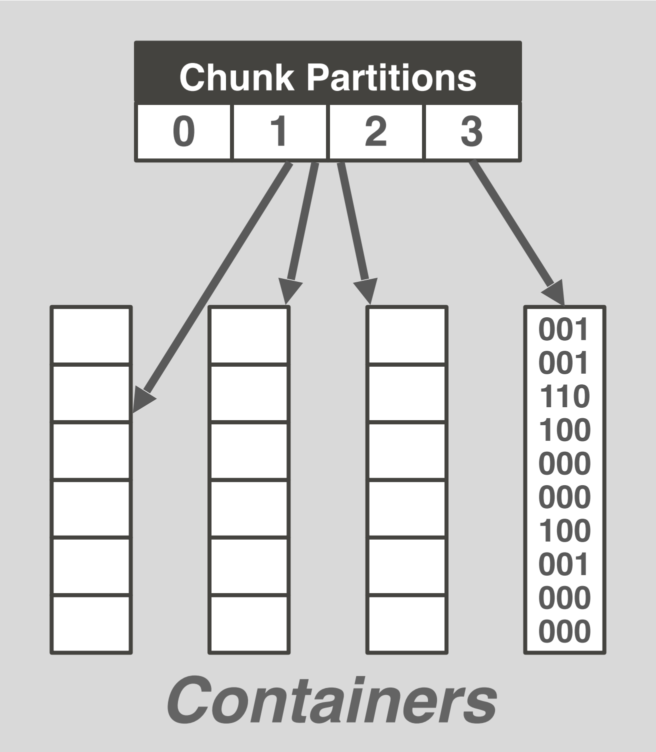
Used in Lucene, Hive, Spark, Pinot.

BETTER BITMAP PERFORMANCE WITH ROARING BITMAPS
SOFTWARE: PRACTICE AND EXPERIENCE 2015

Roaring Bitmaps

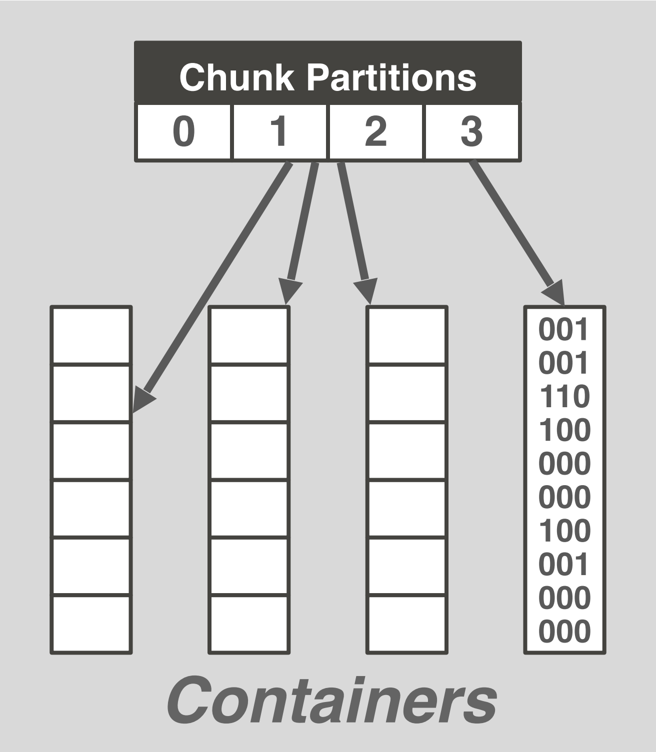


Roaring Bitmaps



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

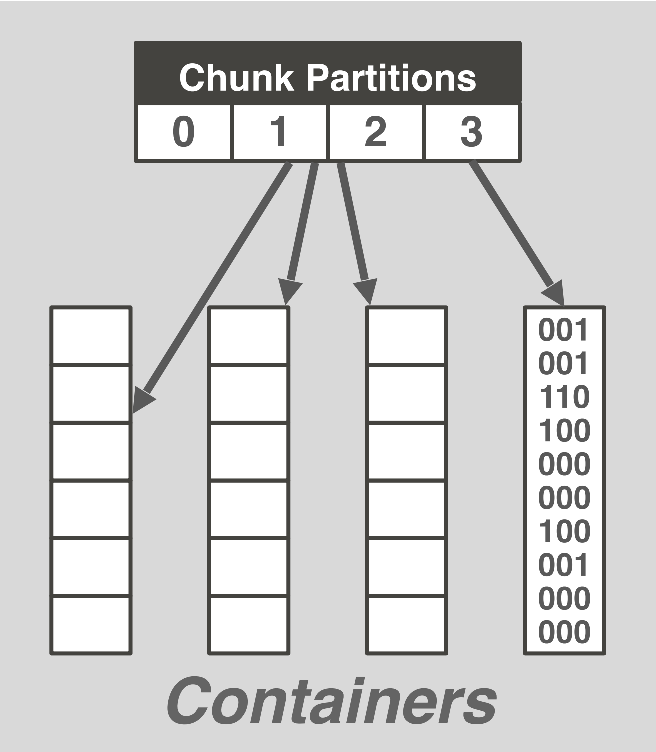
Roaring Bitmaps



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

Roaring Bitmaps

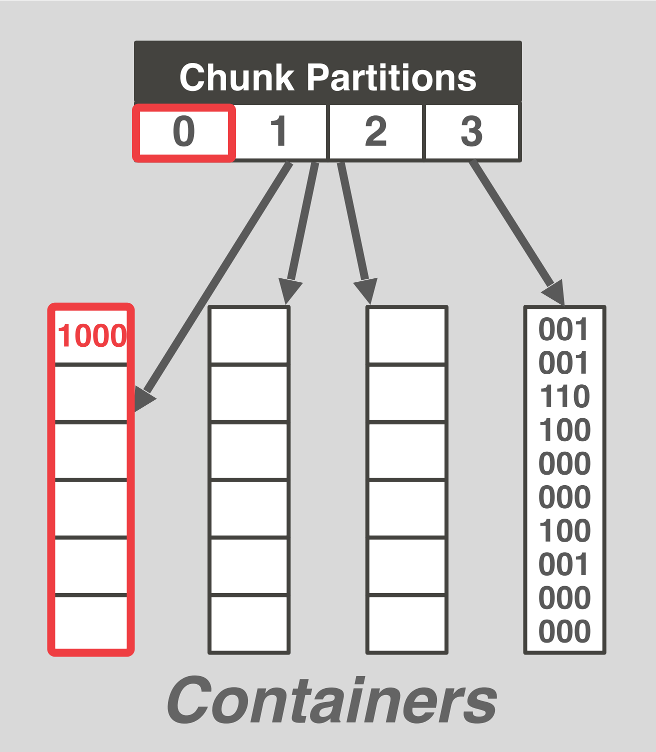


For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

Roaring Bitmaps



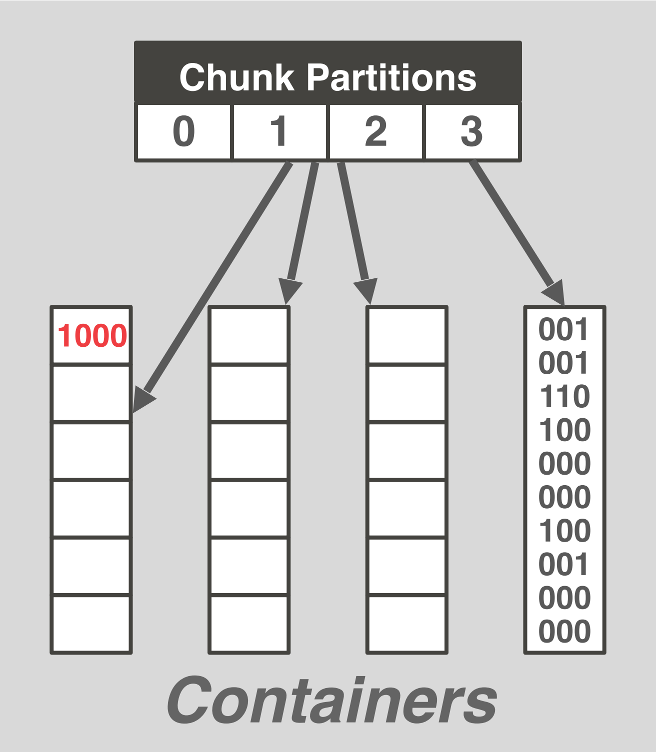
For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

$1000/2^{16}$=0

Roaring Bitmaps



For each value **k**, assign it to a chunk based on **k/2^16**.
→Store **k** in the chunk's container.

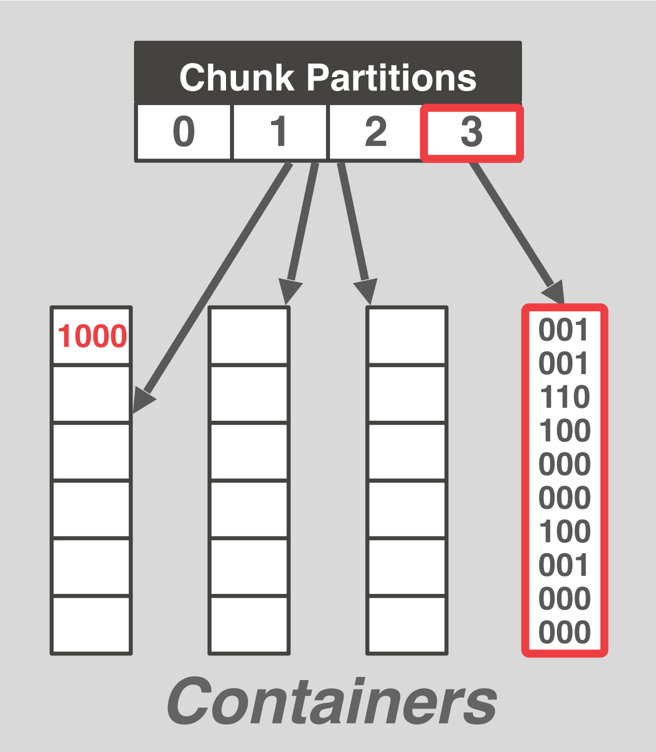
If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

$1000/2^{16}=0$

k=199658

Roaring Bitmaps



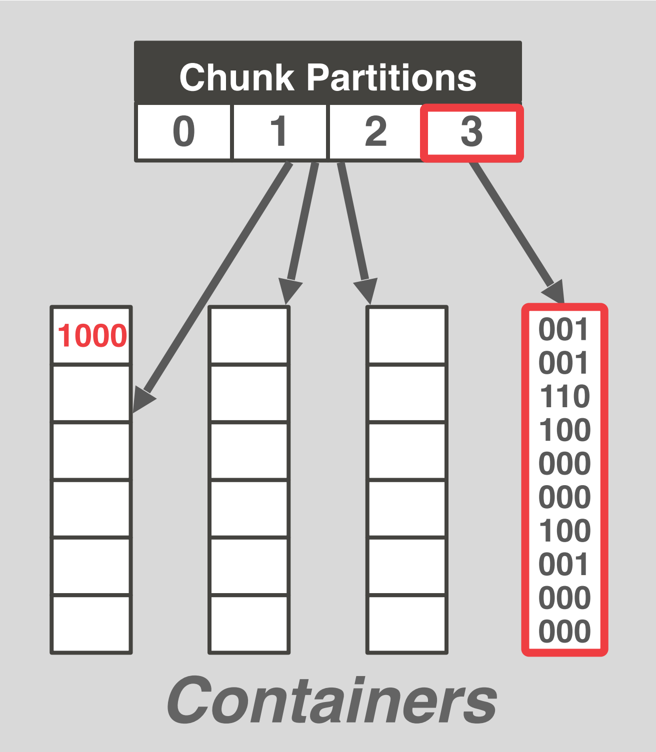
For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000
$1000/2^{16}=0$

k=199658
$199658/2^{16}=3$

Roaring Bitmaps



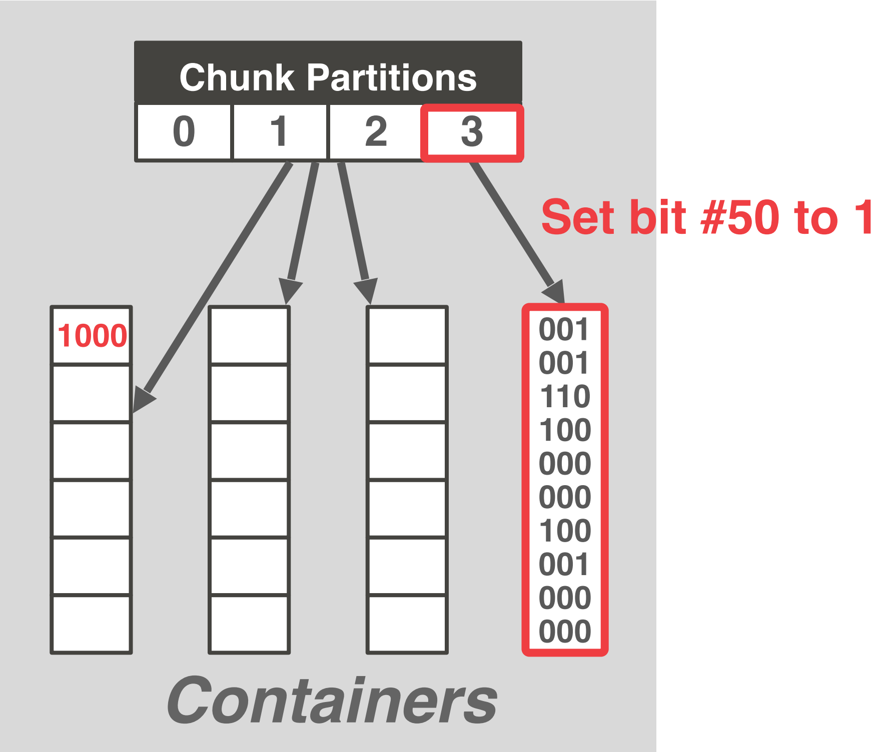
For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000
$1000/2^{16}=0$

k=199658
$199658/2^{16}=3$
$199658\%2^{16}=50$

Roaring Bitmaps



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.
→Store **k** in the chunk's container.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000
$1000/2^{16}=0$

k=199658
$199658/2^{16}=3$
$199658\%2^{16}=50$

Delta Encoding

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

Delta Encoding

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

Delta Encoding

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

Compressed Data

time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

Delta Encoding

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

Compressed Data

time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

Delta Encoding

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

Compressed Data

time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

Compressed Data

time	temp
12:00	99.5
(+1,4)	-0.1
	+0.1
	+0.1
	-0.2

Delta Encoding

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

5 × 32-bits = 160 bits

Compressed Data

time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

32-bits + (4 × 16-bits) = 96 bits

Compressed Data

time	temp
12:00	99.5
(+1,4)	-0.1
	+0.1
	+0.1
	-0.2

32-bits + (2 × 16-bits) = 64 bits

Bit Packing

If the values for an integer attribute is smaller than the range of its given data type size, then reduce the number of bits to represent each value.

Use bit-shifting tricks to operate on multiple values in a single word.

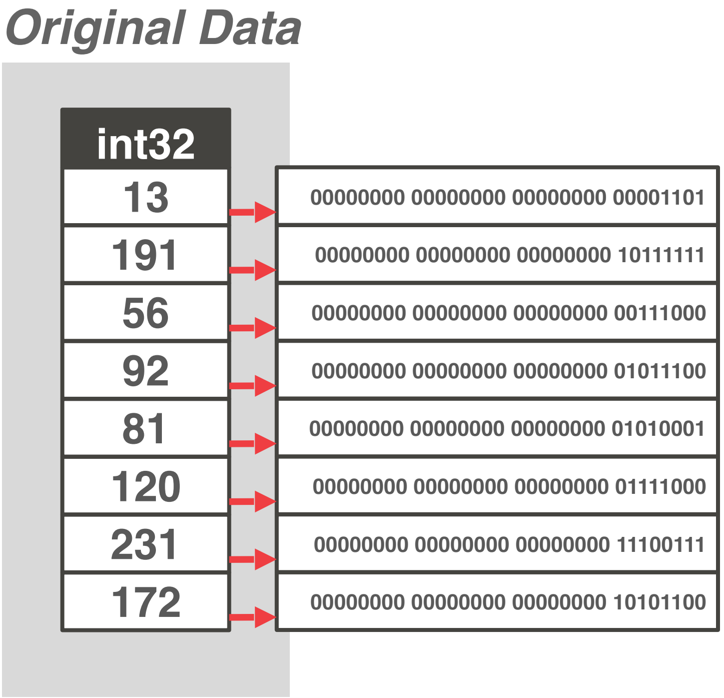
Original Data

int32
13
191
56
92
81
120
231
172

Bit Packing

If the values for an integer attribute is <u>smaller</u> than the range of its given data type size, then reduce the number of bits to represent each value.

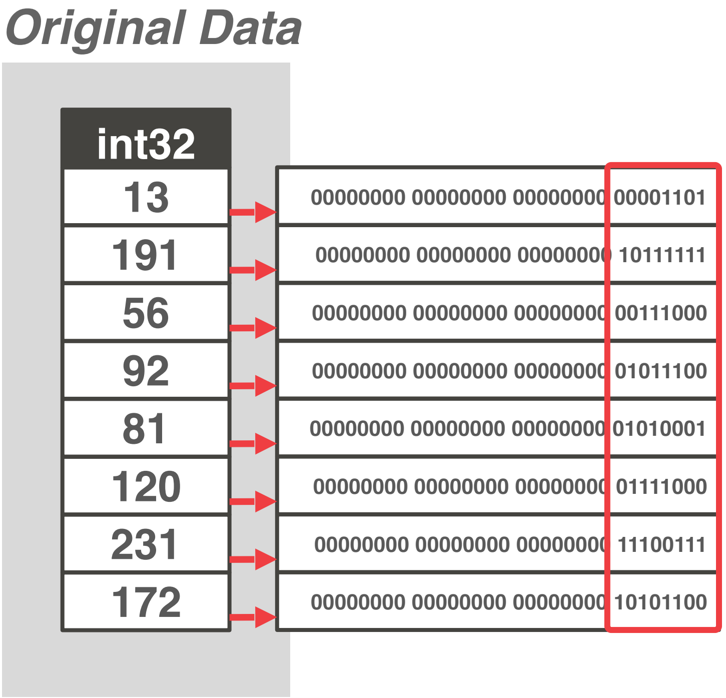
Use bit-shifting tricks to operate on multiple values in a single word.



Bit Packing

If the values for an integer attribute is smaller than the range of its given data type size, then reduce the number of bits to represent each value.

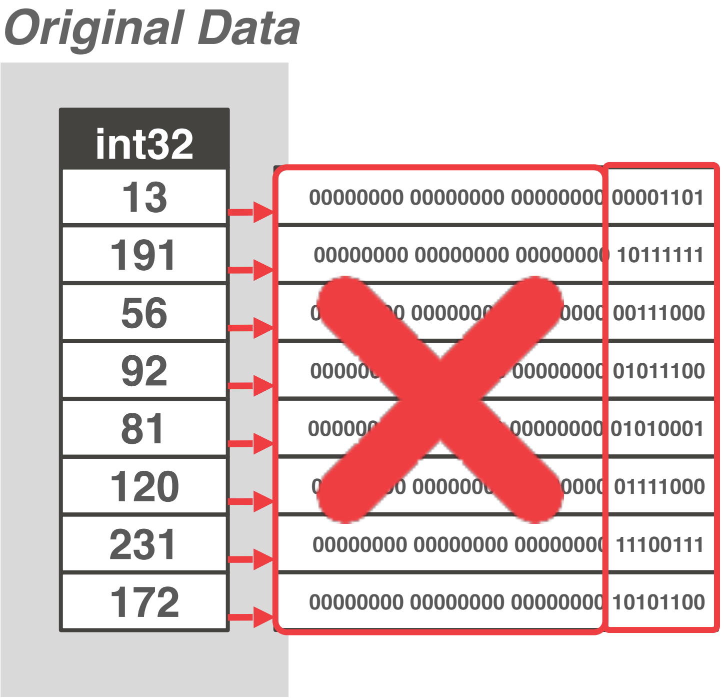
Use bit-shifting tricks to operate on multiple values in a single word.



Bit Packing

If the values for an integer attribute is <u>smaller</u> than the range of its given data type size, then reduce the number of bits to represent each value.

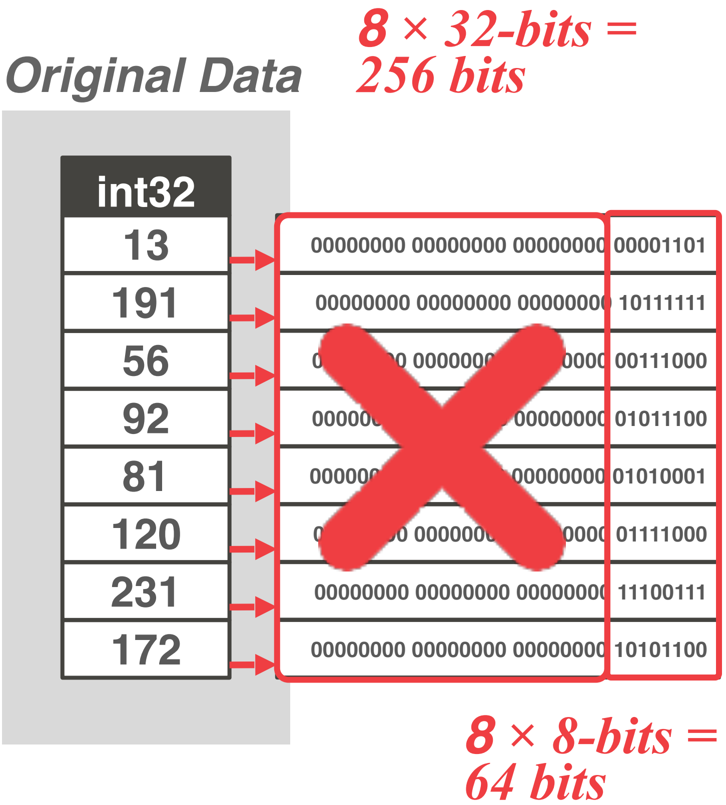
Use bit-shifting tricks to operate on multiple values in a single word.



Bit Packing

If the values for an integer attribute is <u>smaller</u> than the range of its given data type size, then reduce the number of bits to represent each value.

Use bit-shifting tricks to operate on multiple values in a single word.



Mostly Encoding

A variation of bit packing for when an attribute's values are "mostly" less than the largest size, store them with smaller data type.
→ The remaining values that cannot be compressed are stored in their raw form.

Original Data

int32
13
191
99999999
92
81
120
231
172

Source: [Redshift Documentation]

Mostly Encoding

A variation of bit packing for when an attribute's values are "mostly" less than the largest size, store them with smaller data type.

→ The remaining values that cannot be compressed are stored in their raw form.

Original Data

int32
13
191
99999999
92
81
120
231
172

Compressed Data

mostlv8
13
191
XXX
92
81
120
231
172

offset	value
3	99999999

Source: [Redshift Documentation]

Mostly Encoding

A variation of bit packing for when an attribute's values are "mostly" less than the largest size, store them with smaller data type.
→ The remaining values that cannot be compressed are stored in their raw form.

Original Data

int32
13
191
99999999
92
81
120
231
172

8 × 32-bits = 256 bits

Compressed Data

mostlv8
13
191
XXX
92
81
120
231
172

offset	value
3	99999999

(8 × 8-bits) + 16-bits + 32-bits = 112 bits

Source: [Redshift Documentation]

Intermediate Results

After evaluating a predicate on compressed data, the DBMS will decompress it as it moves from the scan operator to the next operator.

→ Example: Execute a hash join on two tables that use different compression schemes.

The DBMS (typically) does not recompress data during query execution. Otherwise, the system needs to embed decompression logic throughout the entire execution engine.

Parting Thoughts

Dictionary encoding is not always the most effective compression scheme, but it is the most used.

The DBMS can combine different approaches for even better compression.

Next Class

Data Skipping