Lecture #10

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Database Compression

@Andy_Pavlo // 15-721 // Spring 2019

UPCOMING DATABASE EVENTS

Splice Machine Tech Talk

→ Thursday Feb 21st @ 12:00pm
→ CIC 4th Floor
→ CEO/Co-Found Monte Zweben (CMU'85)

TODAY'S AGENDA

Compression Background
Naïve Compression
OLAP Columnar Compression
OLTP Index Compression

OBSERVATION

I/O is the main bottleneck if the DBMS has to fetch data from disk.

In-memory DBMSs are more complicated.

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

→ In-memory DBMSs (always?) choose speed.

→ Compressing the database reduces DRAM requirements and processing.

REAL-WORLD DATA CHARACTERISTICS

Data sets 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

DATABASE COMPRESSION

Goal #1: Must produce fixed-length values.
→ Only exception is var-length data stored in separate pool.

Goal #2: Postpone decompression for as long as possible during query execution.
→ Also known as **late materialization**.

Goal #3: Must be a **lossless** scheme.

LOSSLESS VS. LOSSY COMPRESSION

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

Any kind of **lossy** compression is has to be performed at the application level.

Reading less than the entire data set during query execution is sort of like of compression…

DATA SKIPPING

Approach #1: Approximate Queries (Lossy)

→ Execute queries on a sampled subset of the entire table to produce approximate results.
→ Examples: BlinkDB, SnappyData, XDB, Oracle (2017)

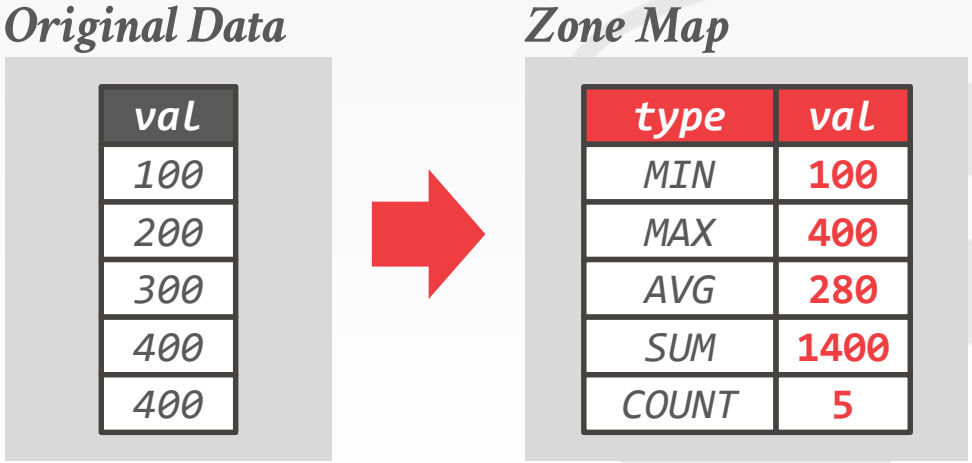
Approach #2: Zone Maps (Loseless)

→ Pre-compute columnar aggregations per block that allow the DBMS to check whether queries need to access it.
→ Examples: Oracle, Vertica, MemSQL, Netezza

ZONE MAPS

Pre-computed aggregates for blocks of data.

DBMS can check the zone map first to decide whether it wants to access the block.



ZONE MAPS

Pre-computed aggregates for blocks of data.

DBMS can check the zone map first to decide whether it wants to access the block.

```
SELECT * FROM table
 WHERE val > 600
```

Original Data

val
100
200
300
400
400

Zone Map

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

OBSERVATION

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

This determines what compression schemes are available to us…

COMPRESSION GRANULARITY

Choice #1: Block-level

→ Compress a block of tuples for the same table.

Choice #2: Tuple-level

→ Compress the contents of the entire tuple (NSM-only).

Choice #3: Attribute-level

→ Compress a single attribute value within one tuple.
→ Can target multiple attributes for the same tuple.

Choice #4: Column-level

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

NAÏVE COMPRESSION

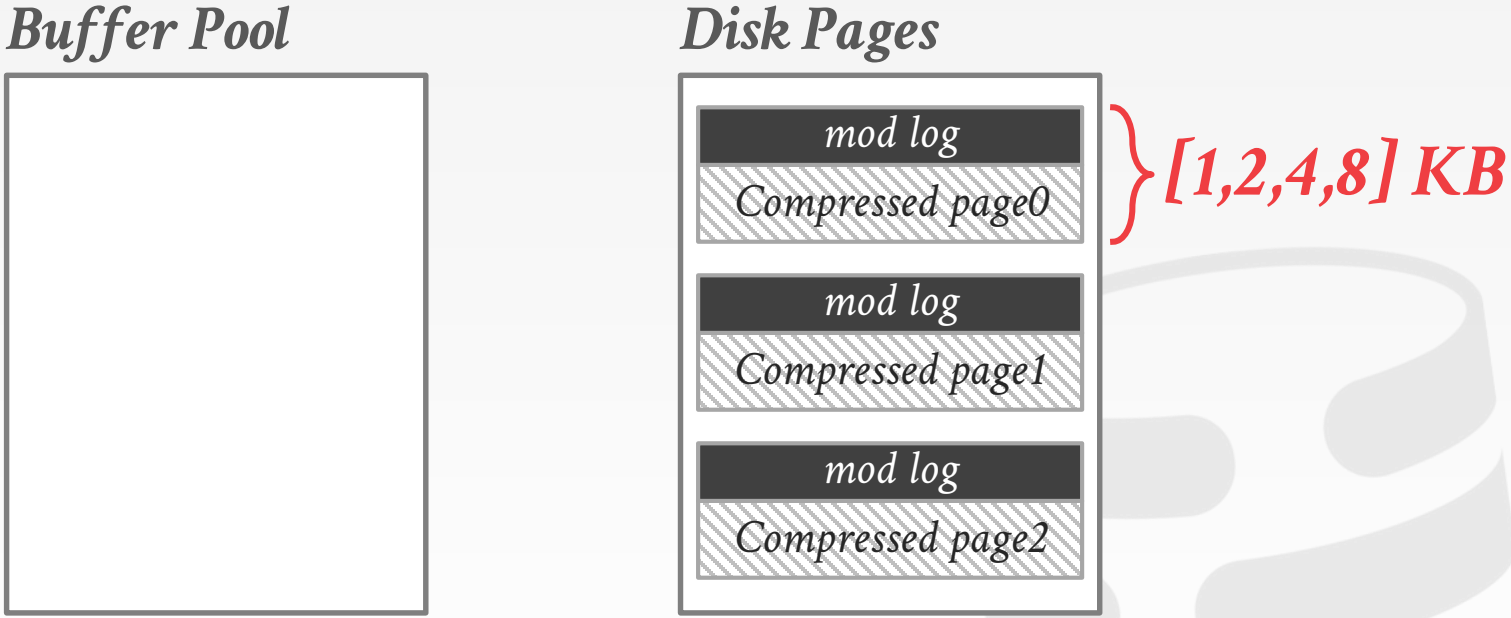
Compress data using a general purpose algorithm.
Scope of compression is only based on the data provided as input.

→ LZO (1996), LZ4 (2011), Snappy (2011), Brotli (2013), Oracle OZIP (2014), Zstd (2015)

Considerations

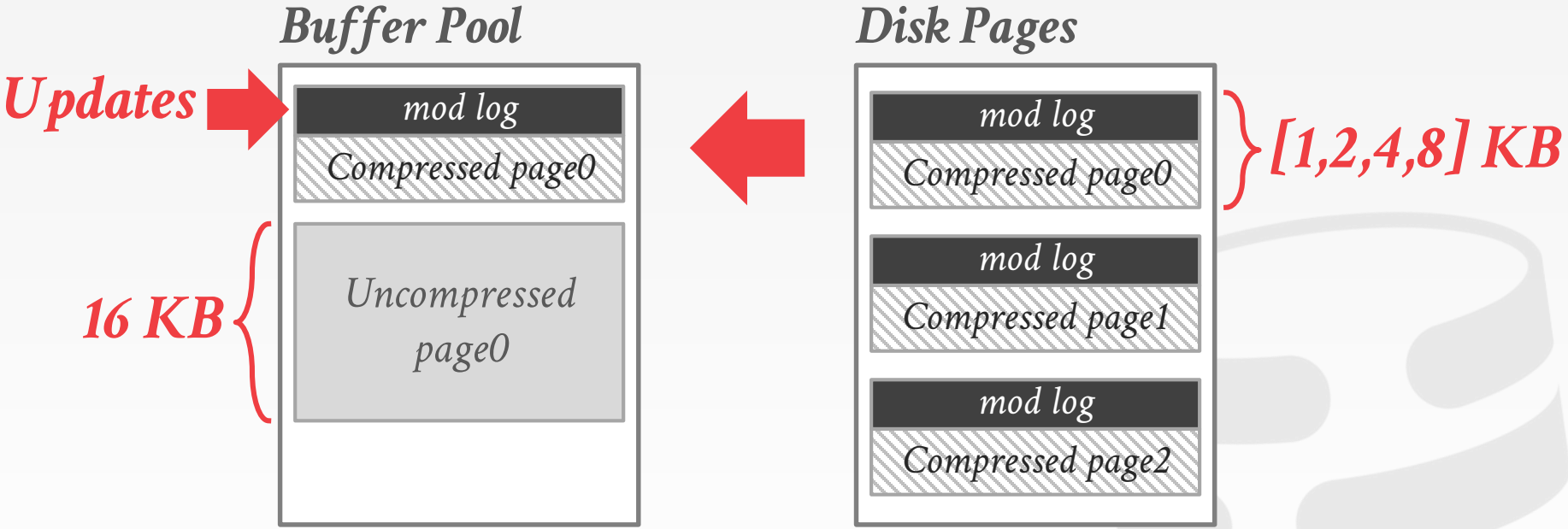
→ Computational overhead
→ Compress vs. decompress speed.

MYSQL INNODB COMPRESSION



Source: MySQL 5.7 Documentation

MYSQL INNODB COMPRESSION



Source: MySQL 5.7 Documentation

NAÏVE COMPRESSION

The data has to be decompressed first before it can be read and (potentially) modified.
→ This limits the "scope" of the compression scheme.

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

OBSERVATION

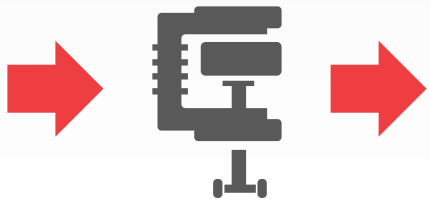
We can perform exact-match comparisons and natural joins on compressed data if predicates and data are compressed the same way.

→ Range predicates are more tricky…

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

```
SELECT * FROM users
 WHERE name = XX
```

NAME	SALARY
Andy	99999
Lin	88888



NAME	SALARY
XX	AA
YY	BB

COLUMNAR COMPRESSION

Null Supression
Run-length Encoding
Bitmap Encoding
Delta Encoding
Incremental Encoding
Mostly Encoding
Dictionary Encoding

NULL SUPPRESSION

Consecutive zeros or blanks in the data are replaced with a description of how many there were and where they existed.

→ Example: Oracle's Byte-Aligned Bitmap Codes (BBC)

Useful in wide tables with sparse data.

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.

RUN-LENGTH ENCODING

Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M

Compressed Data

id	sex
1	(M,0,3)
2	(F,3,1)
3	(M,4,1)
4	(F,5,1)
6	(M,6,2)
7	
8	
9	

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

RUN-LENGTH ENCODING

Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M

Compressed Data

id	sex
1	(M,0,3)
2	(F,3,1)
3	(M,4,1)
4	(F,5,1)
6	(M,6,2)
7	
8	
9	

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

RUN-LENGTH ENCODING

Sorted Data

id	sex
1	M
2	M
3	M
6	M
8	M
9	M
4	F
7	F

Compressed Data

id	sex
1	(M,0,6)
2	(F,7,2)
3	
6	
7	
9	
4	
7	

BITMAP ENCODING

Store a separate Bitmap for each unique value for a particular attribute where an offset in the vector corresponds to a tuple.

→ The i^{th} position in the Bitmap corresponds to the i^{th} tuple in the table.

→ Typically segmented into chunks to avoid allocating large blocks of contiguous memory.

Only practical if the value cardinality is low.

MODEL 204 ARCHITECTURE AND PERFORMANCE
HIGH PERFORMANCE TRANSACTION SYSTEMS 1987

BITMAP ENCODING

Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M

Compressed Data

id	sex	
	M	F
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

Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M

Compressed Data

id	sex	
	M	F
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

Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M

9 × 8-bits = 72 bits

Compressed Data

	sex	
id	M	F
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

BITMAP ENCODING: EXAMPLE

```
CREATE TABLE customer_dim (
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

Assume we have 10 million tuples.
43,000 zip codes in the US.
→ **10000000 × 32-bits = 40 MB**
→ **10000000 × 43000 = 53.75 GB**

Every time a txn inserts a new tuple, we have to extend 43,000 different bitmaps.

BITMAP ENCODING: COMPRESSION

Approach #1: General Purpose Compression

→ Use standard compression algorithms (e.g., LZ4, Snappy).

→ Have to decompress before you can use it to process a query. Not useful for in-memory DBMSs.

Approach #2: Byte-aligned Bitmap Codes

→ Structured run-length encoding compression.

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 **chunk** 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

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

```
#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 1 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   #2
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 have to 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 #2
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 have to 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   #2
00000000 00000000 00000000
00000000 01000000 00100010
```

Compressed Bitmap

```
#1 (010)(1)(0100)
                          Gap Length
#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 have to 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 #2
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 have to 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 #2
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 have to 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 #2
00000000 00000000 00000000
00000000 01000000 00100010
```

Compressed Bitmap

```
#1 (010)(1)(0100)

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

Verbatim Tail Bytes

Chunk #2 (Bytes 4-18)

Header Byte:

→ 13 gap bytes, two tail bytes

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

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Original: 18 bytes

BBC Compressed: 5 bytes.

Source: Brian Babcock

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 have to start from the beginning and decompress the whole thing.

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**.

→ Can be combined 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**.

→ Can be combined 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**.

→ Can be combined 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

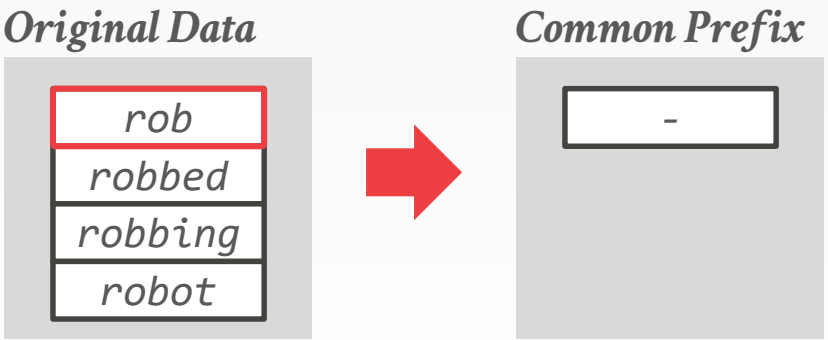
time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

Compressed Data

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

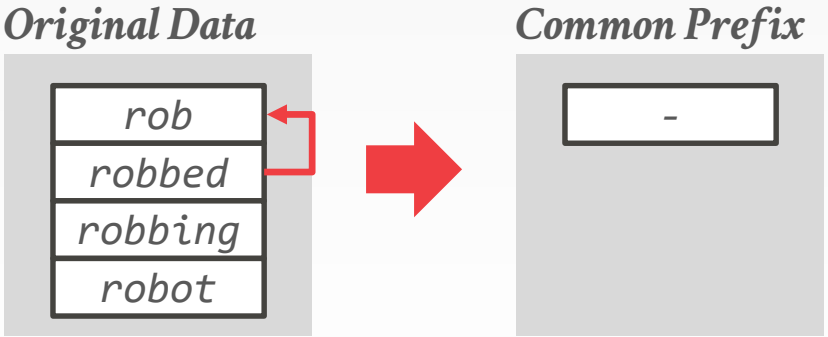
INCREMENTAL ENCODING

Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



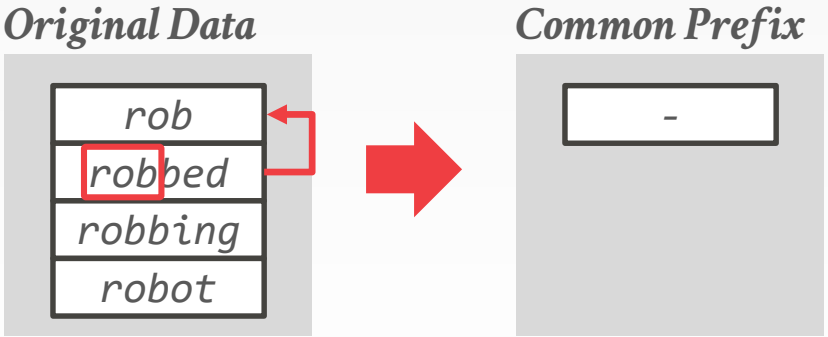
INCREMENTAL ENCODING

Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



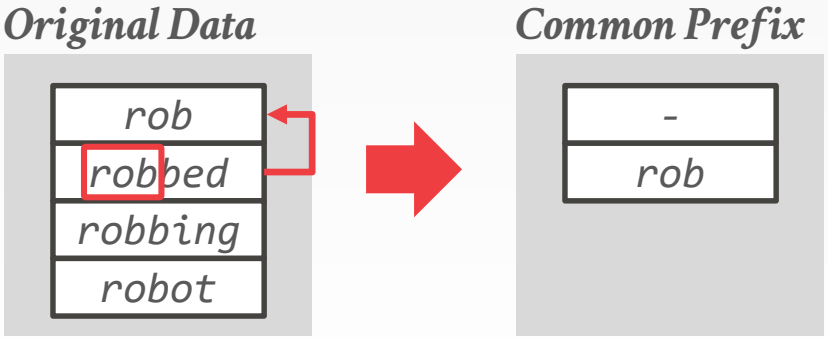
INCREMENTAL ENCODING

Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



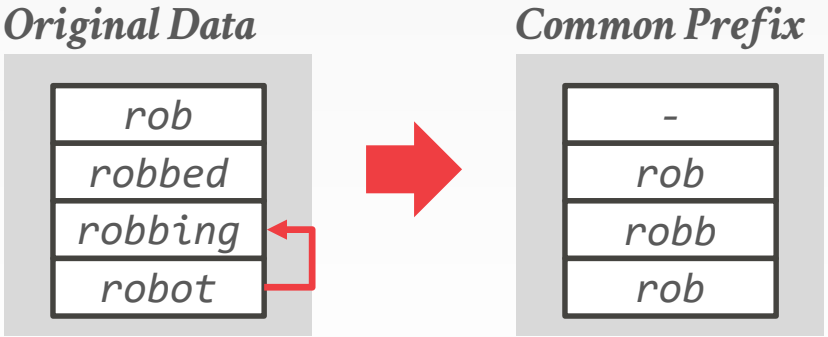
INCREMENTAL ENCODING

Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



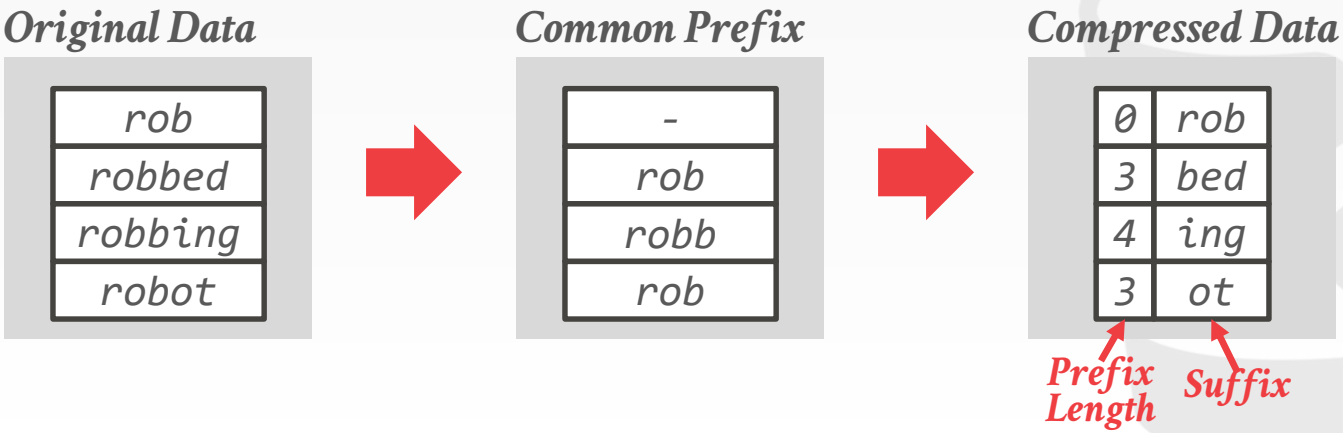
INCREMENTAL ENCODING

Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



INCREMENTAL ENCODING

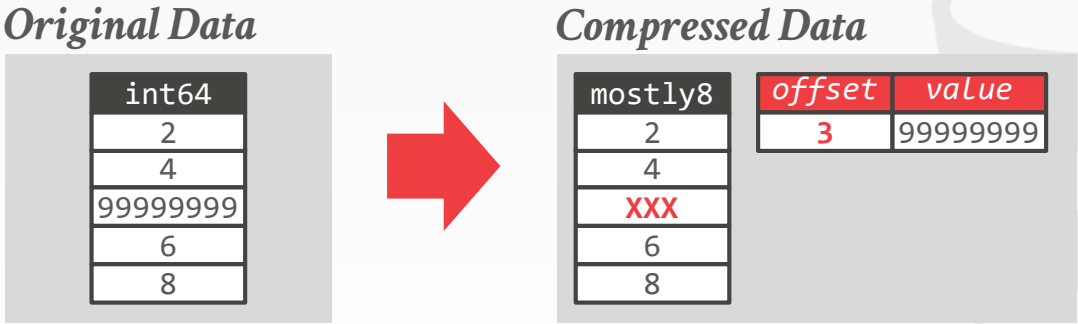
Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



MOSTLY ENCODING

When the values for an attribute are "mostly" less than the largest size, you can store them as a smaller data type.

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



Source: Redshift Documentation

DICTIONARY COMPRESSION

Replace frequent patterns with smaller codes.
Most pervasive compression scheme in DBMSs.

Need to support fast encoding and decoding.
Need to also support range queries.

ADAPTIVE STRING DICTIONARY COMPRESSION IN IN-MEMORY COLUMN-STORE DATABASE SYSTEMS
EDBT 2014

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 the all tuples must be recomputed.

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.
→ Potentially lower compression ratio, but can add new tuples more easily.

Choice #2: Table-level

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

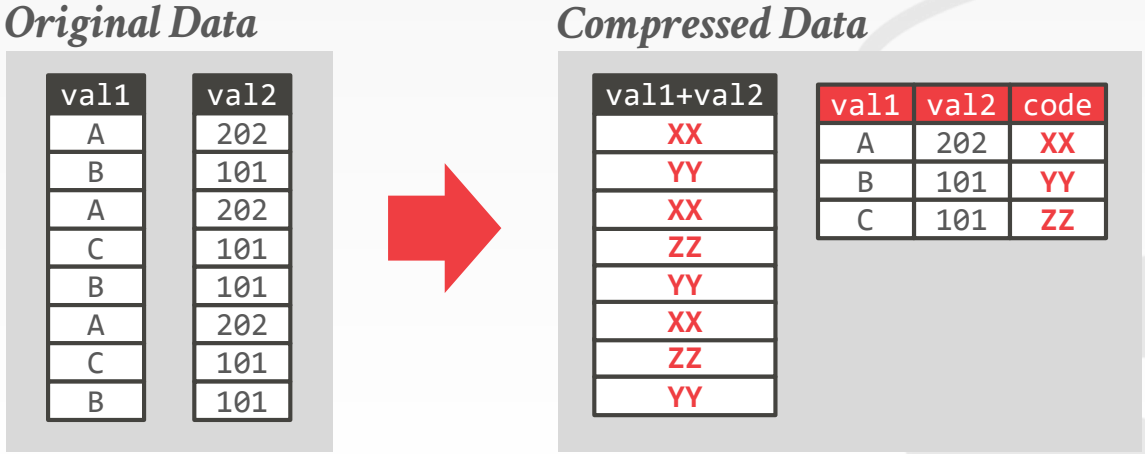
Choice #3: Multi-Table

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

MULTI-ATTRIBUTE ENCODING

Instead of storing a single value per dictionary entry, store entries that span attributes.

→ I'm not sure any DBMS actually implements this.



ENCODING / DECODING

A dictionary needs to support two operations:

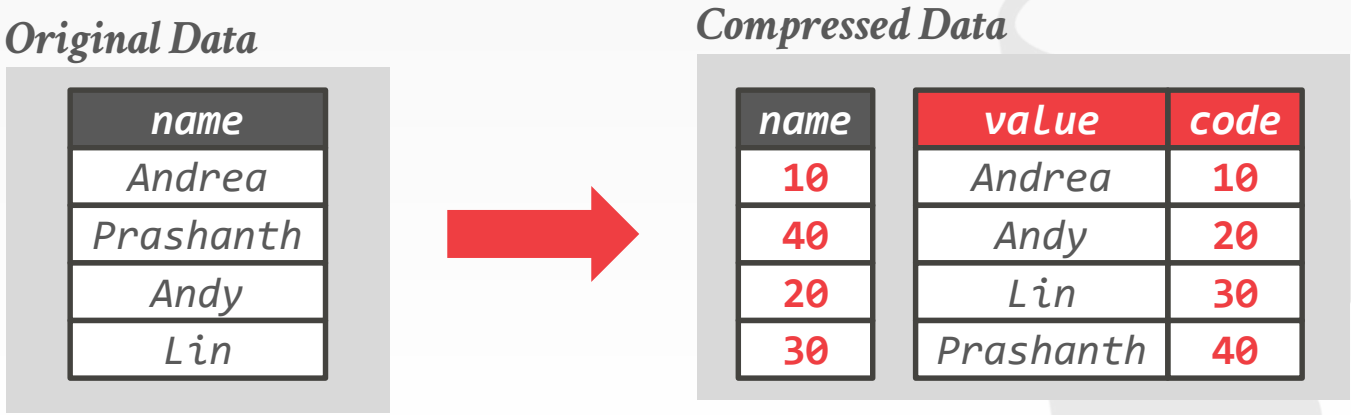
→ **Encode/Locate:** For a given uncompressed value, convert it into its compressed form.

→ **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.



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
Prashanth
Andy
Lin

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andy	20
Lin	30
Prashanth	40

ORDER-PRESERVING ENCODING

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

Still have to perform seq scan

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

Only need to access dictionary

Original Data

name
Andrea
Prashanth
Andy
Lin

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andy	20
Lin	30
Prashanth	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.

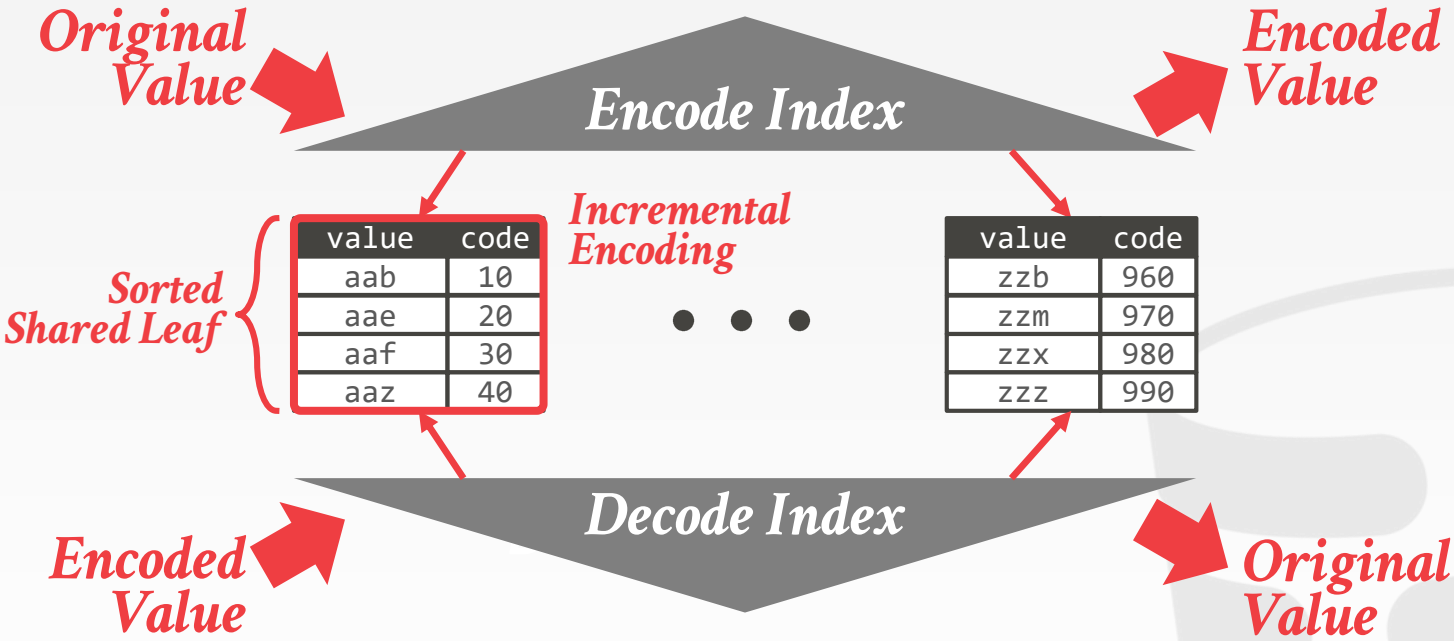
Choice #2: Hash Table

→ 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.

SHARED-LEAVES B+TREE



OBSERVATION

An OLTP DBMS cannot use the OLAP compression techniques because we need to support fast random tuple access.

→ Compressing & decompressing "hot" tuples on-the-fly would be too slow to do during a txn.

Indexes consume a large portion of the memory for an OLTP database…

OLTP INDEX OVERHEAD

	Tuples	*Primary Indexes*	*Secondary Indexes*
TPC-C	42.5%	33.5%	24.0%
Articles	64.8%	22.6%	12.6%
Voter	45.1%	54.9%	0%

OLTP INDEX OVERHEAD

	Tuples	*Primary Indexes*	*Secondary Indexes*	
TPC-C	42.5%	33.5%	24.0%	**57.5%**
Articles	64.8%	22.6%	12.6%	**35.2%**
Voter	45.1%	54.9%	0%	**54.9%**

HYBRID INDEXES

Split a single logical index into two physical indexes. Data is migrated from one stage to the next over time.

→ **Dynamic Stage:** New data, fast to update.
→ **Static Stage:** Old data, compressed + read-only.

All updates go to dynamic stage.
Reads may need to check both stages.

REDUCING THE STORAGE OVERHEAD OF MAIN-MEMORY OLTP DATABASES WITH HYBRID INDEXES
SIGMOD 2016

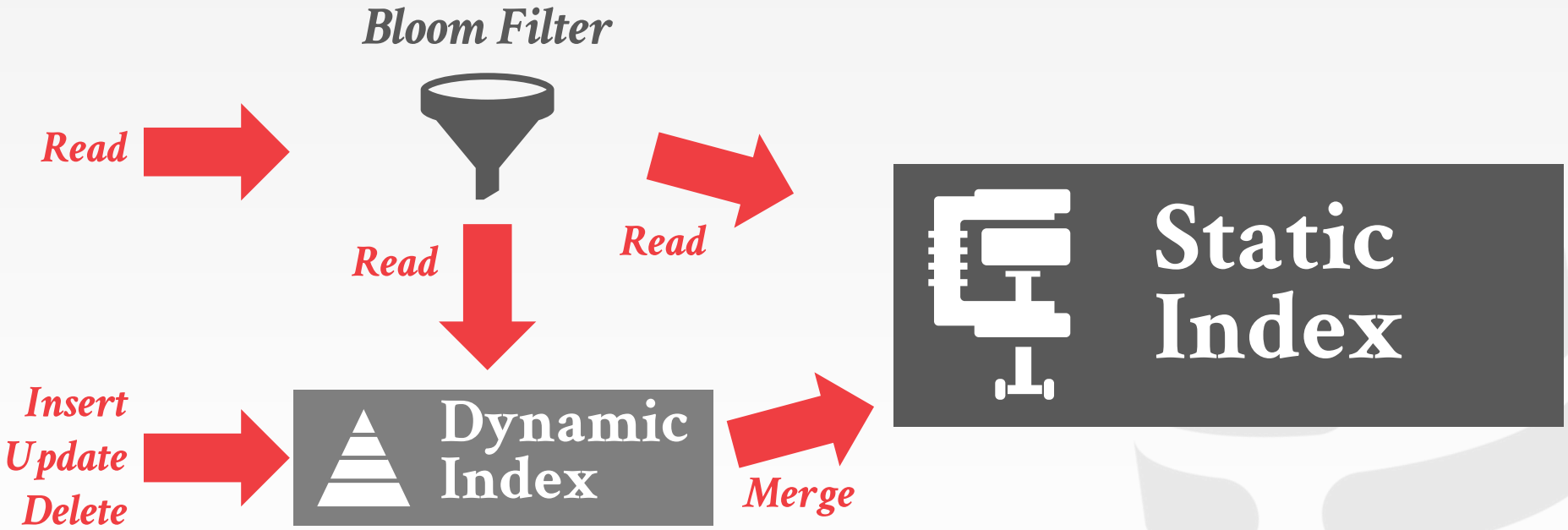
HYBRID INDEXES

Bloom Filter

Insert
Update
Delete

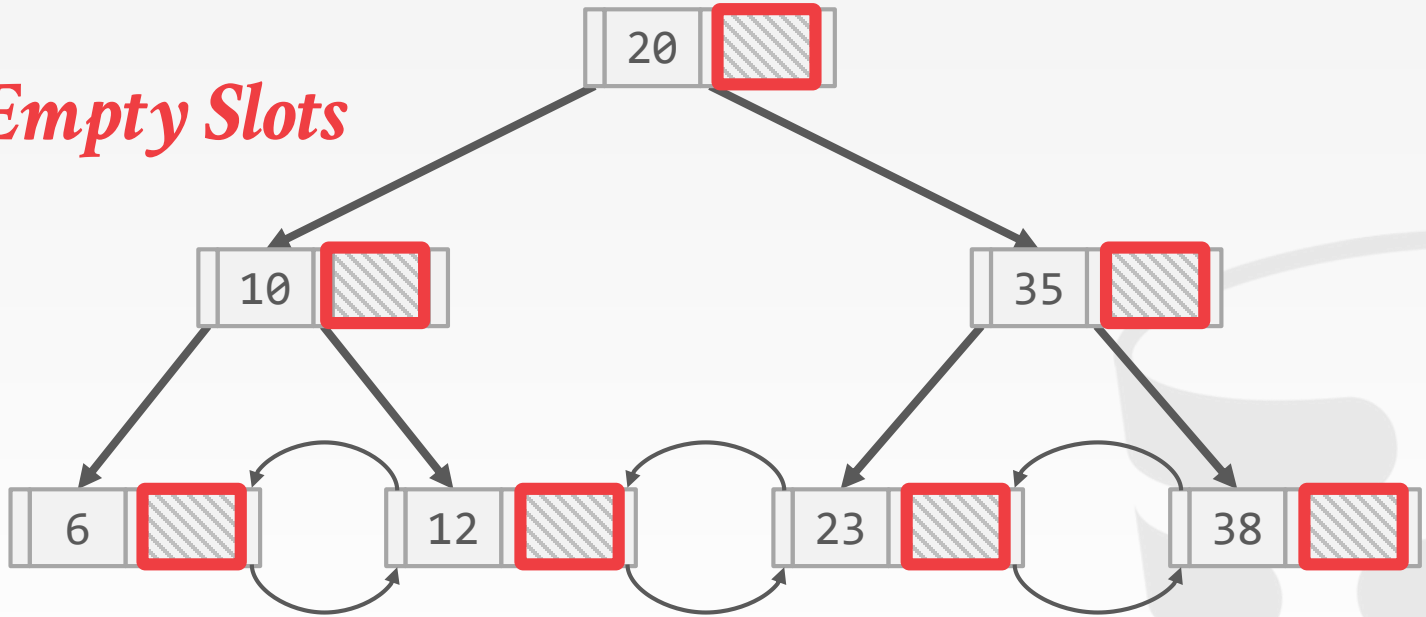
Merge

HYBRID INDEXES

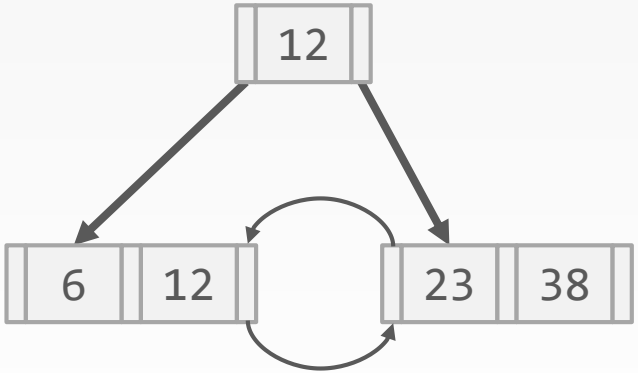


COMPACT B+TREE

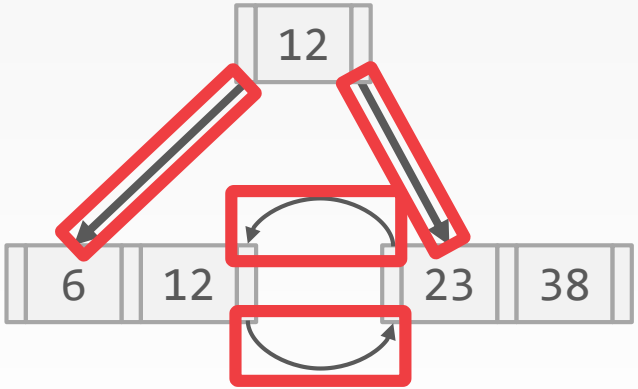
Empty Slots



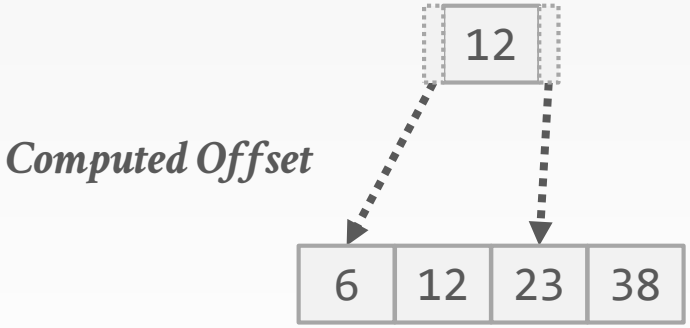
COMPACT B+TREE



COMPACT B+TREE

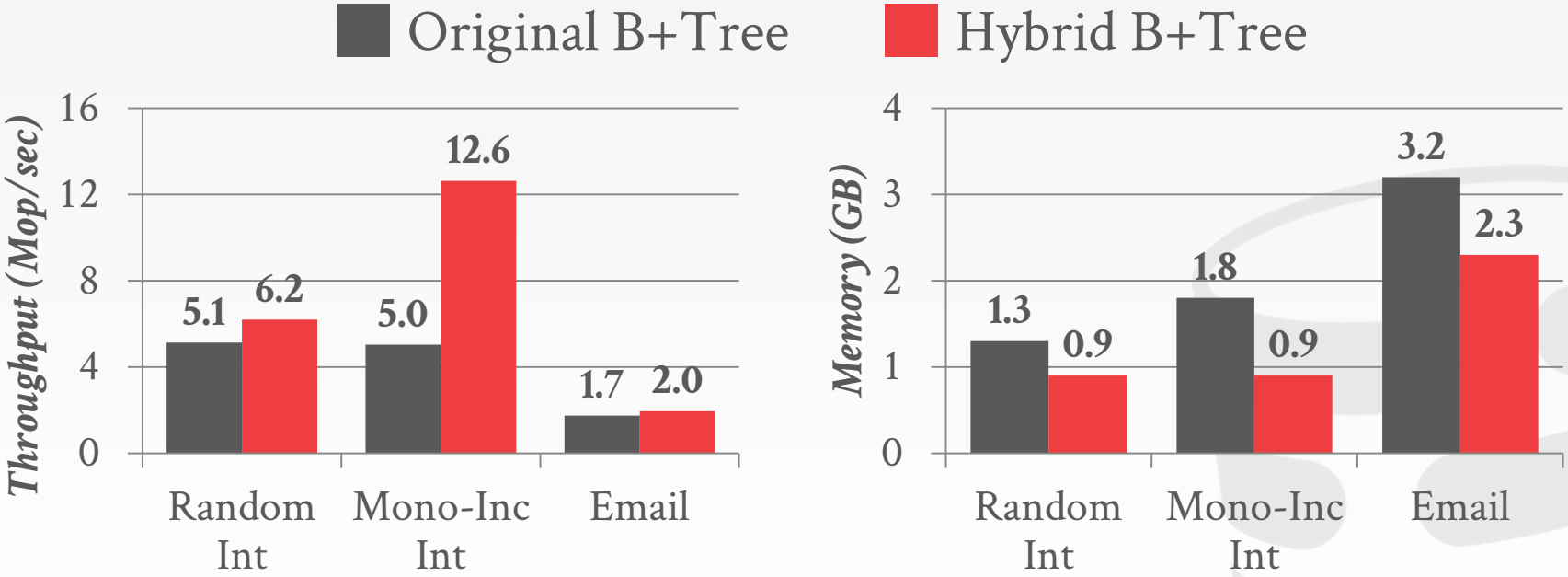


COMPACT B+TREE



HYBRID INDEXES

50% Reads / 50% Writes
50 million Entries



Source: Huanchen Zhang

PARTING THOUGHTS

Dictionary encoding is probably the most useful compression scheme because it does not require pre-sorting.

The DBMS can combine different approaches for even better compression.

It is important to wait as long as possible during query execution to decompress data.

NEXT CLASS

What happens if our database still does not fit in memory even though we compressed it?

Let's bring back disk storage and see what changes in our DBMS architecture…