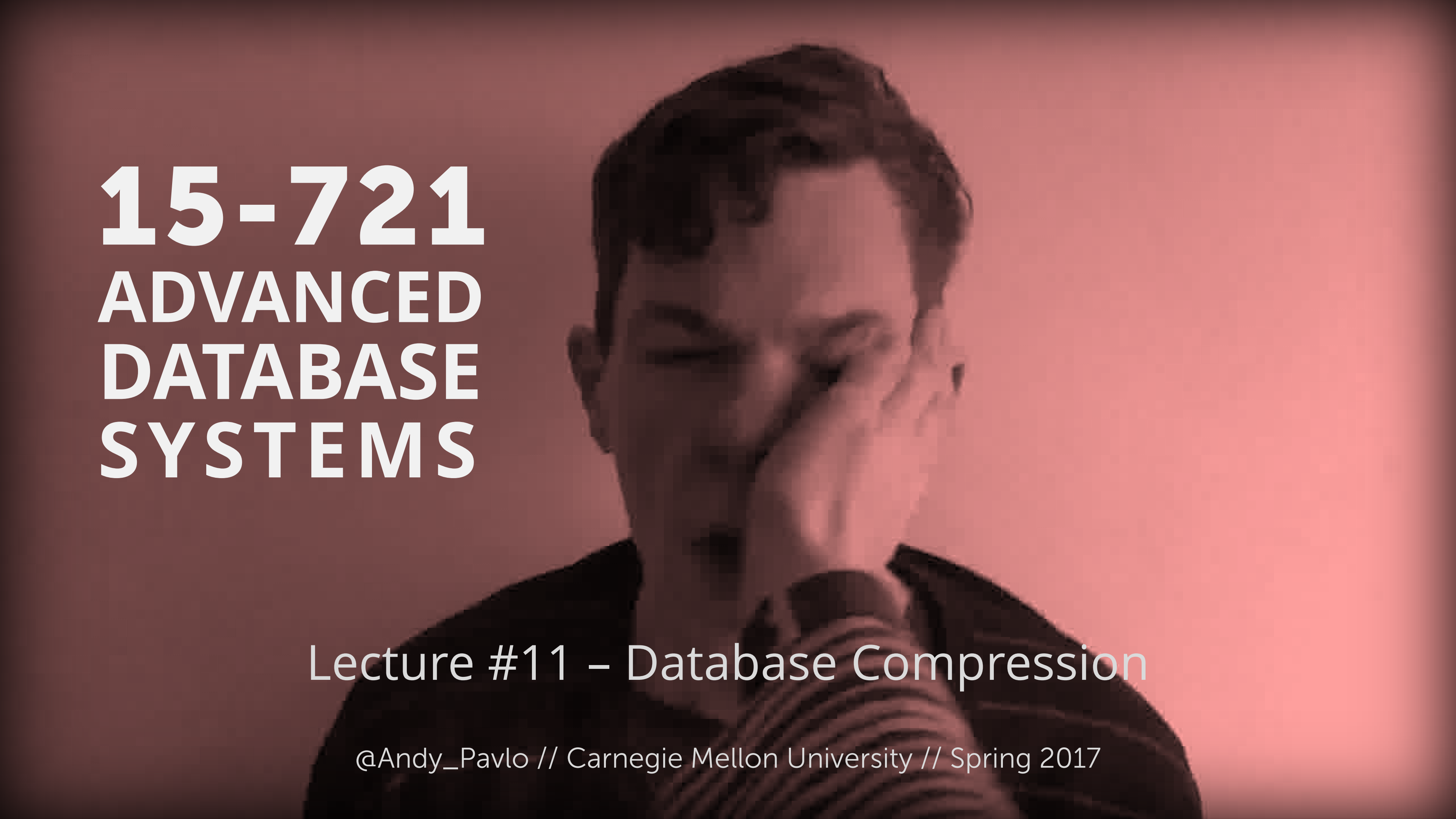


15-721 ADVANCED DATABASE SYSTEMS

Lecture #11 – Database Compression

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

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
→ Compressing the database reduces DRAM requirements and processing.

Key trade-off is **speed** vs. **compression ratio**
→ In-memory DBMSs (always?) choose speed.

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.

Goal #2: Allow the DBMS to postpone decompression as long as possible during query execution.

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.

Some new DBMSs support approximate queries
→ Example: BlinkDB, SnappyData

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).

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

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), Zstd (2015)

Considerations
→ Computational overhead
→ Compress vs. decompress speed.

NAÏVE COMPRESSION

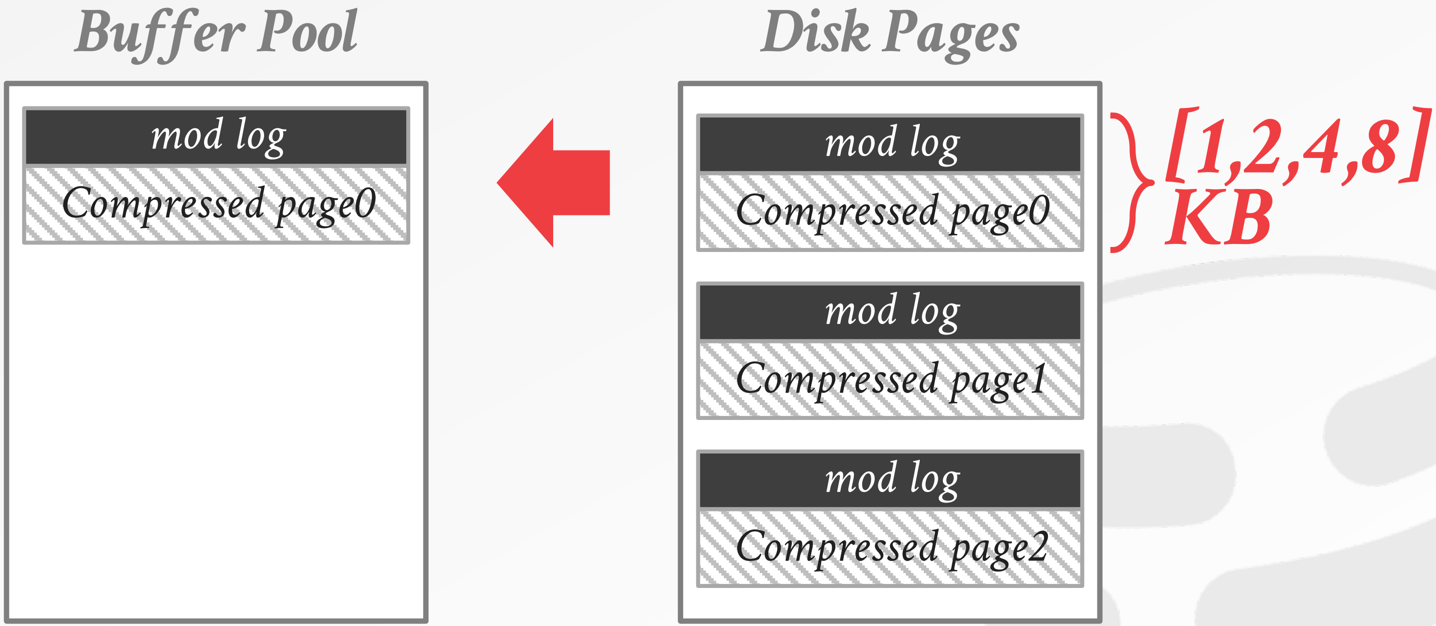
Choice #1: Entropy Encoding

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

Choice #2: Dictionary Encoding

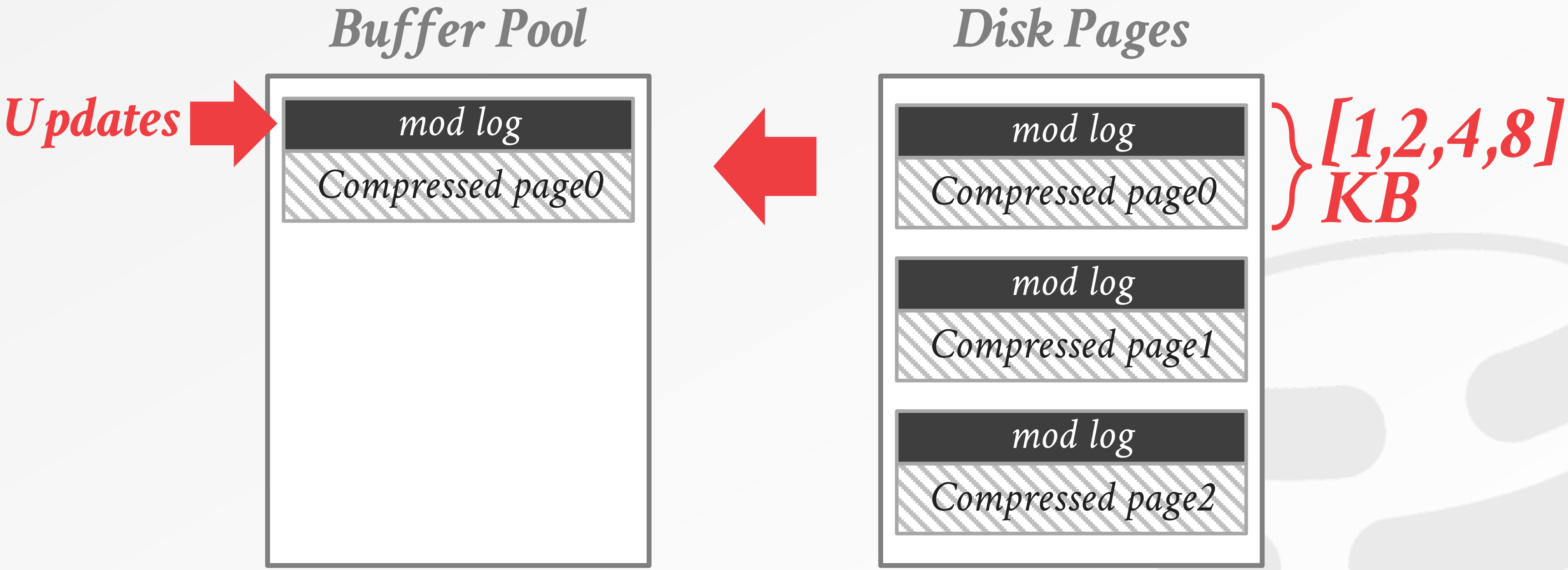
→ Build a data structure that maps data segments to an identifier. Replace those segments in the original data with a reference to the segments position in the dictionary data structure.

MYSQL INNODB COMPRESSION



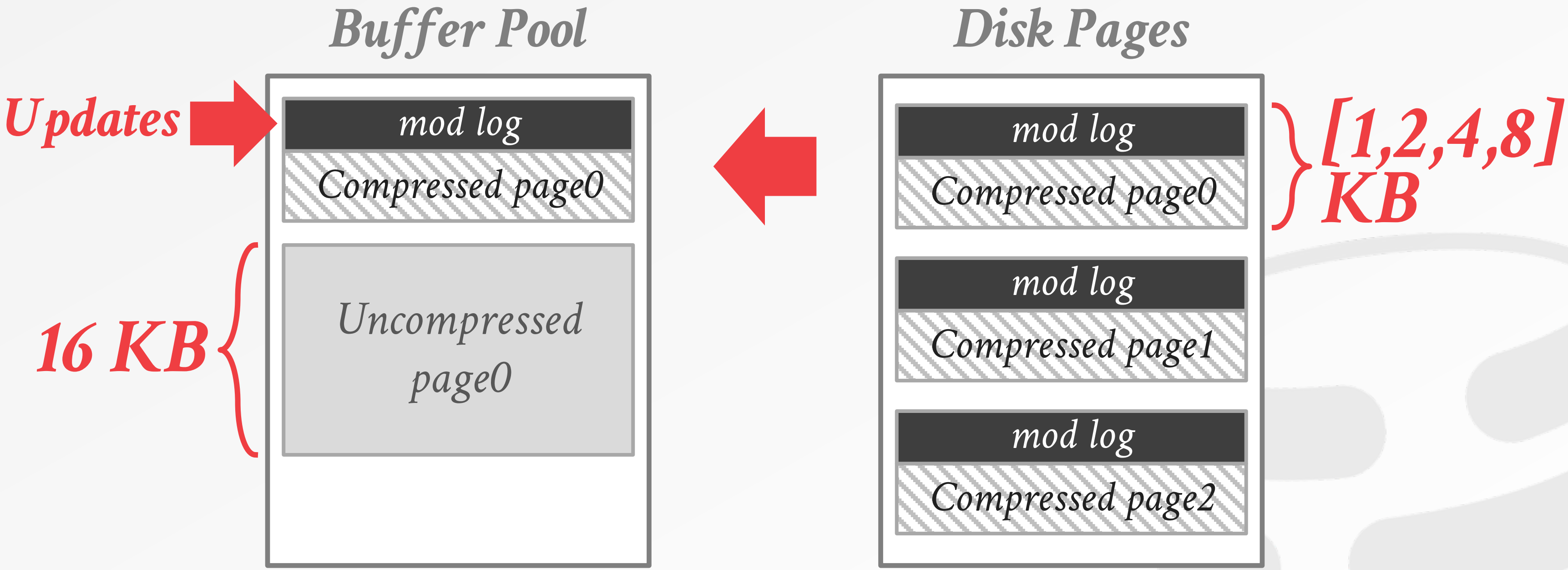
Source: MySQL 5.7 Documentation

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.

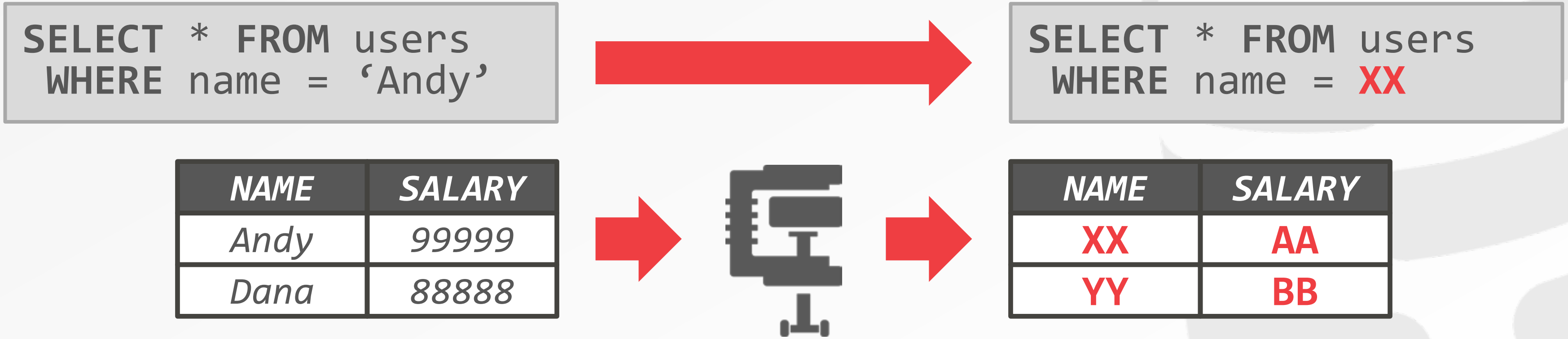
→ This limits the "scope" of the compression scheme.

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’
```

NAME	SALARY
Andy	99999
Dana	88888

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

NAME	SALARY
XX	AA
YY	BB

COLUMNAR COMPRESSION

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

COMPRESSION VS. MSSQL INDEXES

The MSSQL columnar indexes were a second copy of the data (aka fractured mirrors).
→ The original data was still stored as in NSM format.

We are now talking about compressing the primary copy of the data.
Many of the same techniques are applicable.

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.

DATABASE COMPRESSION
SIGMOD RECORD 1993

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.

BITMAP ENCODING

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

- → Can use the same compression schemes that we talked about for Bitmap indexes.

Only practical if the value cardinality is low.

MODEL 204 ARCHITECTURE AND PERFORMANCE
High Performance Transaction Systems 1987

DELTA ENCODING

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

→ The base value can be stored 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.

- → The base value can be stored 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

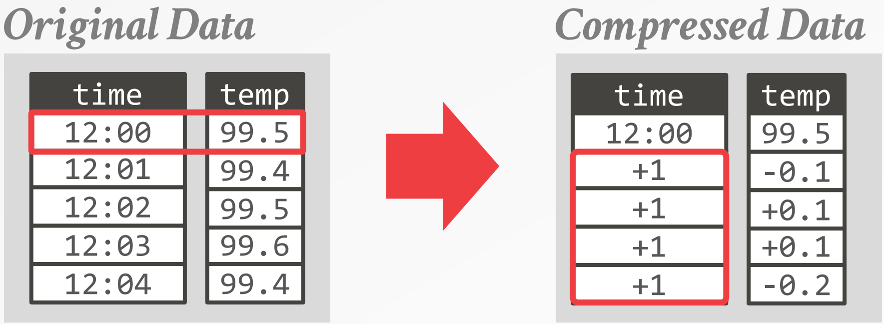
time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

DELTA ENCODING

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

→ The base value can be stored 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

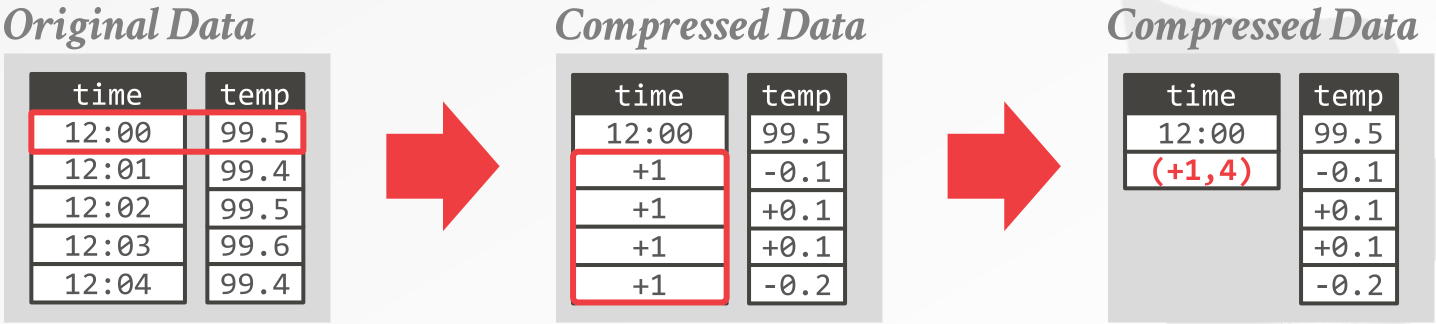
Compressed Data

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.

- → The base value can be stored 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

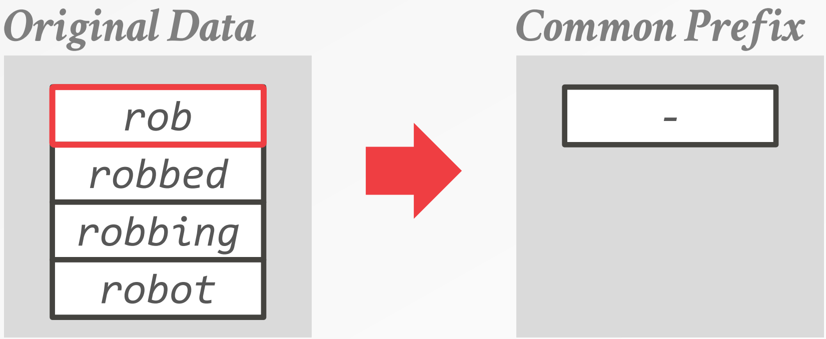
time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

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.



Original Data

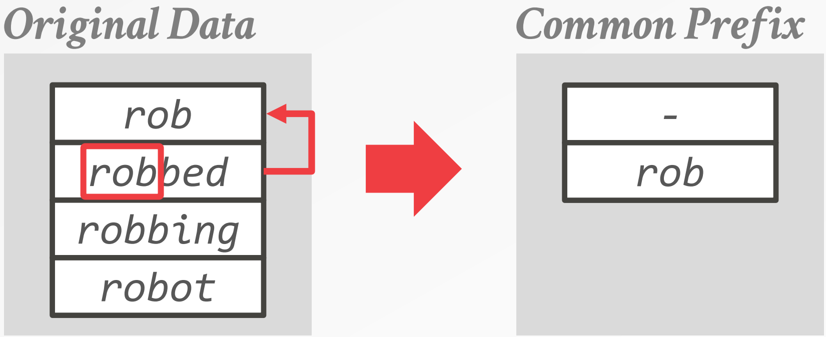
rob
robbed
robbing
robot

Common Prefix

-

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.



Original Data

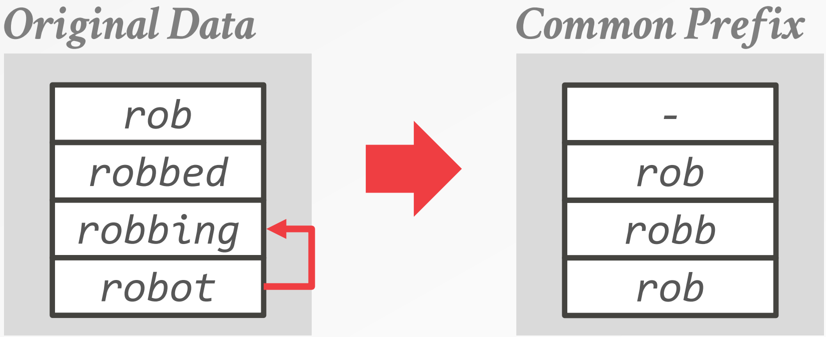
rob
robbed
robbing
robot

Common Prefix

-
rob

INCREMENTAL ENCODING

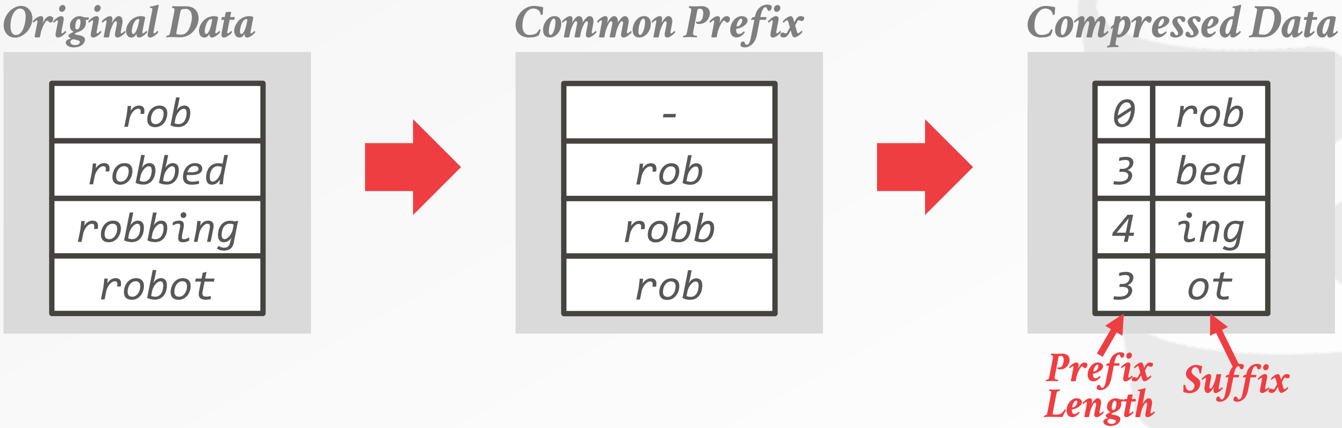
Type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated.
This works best with sorted data.



Original Data	*Common Prefix*
rob	-
robbed	rob
robbing	robb
robot	rob

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.



Original Data

rob
robbed
robbing
robot

Common Prefix

-
rob
robb
rob

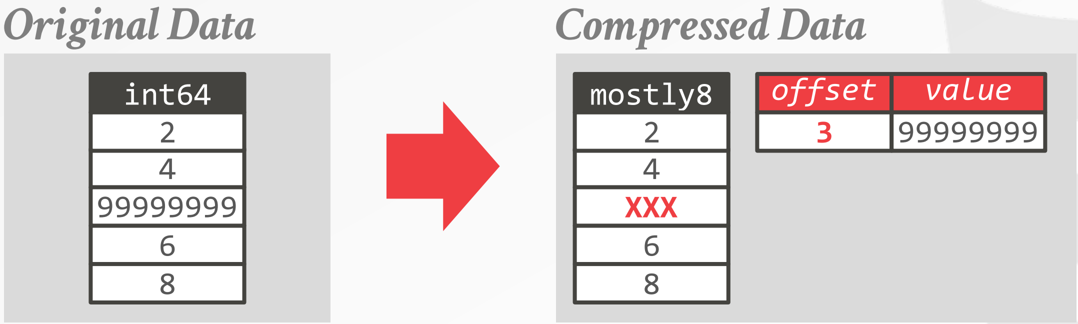
Compressed Data

0	rob
3	bed
4	ing
3	ot

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.

DICTIONARY-BASED ORDER-PRESERVING STRING COMPRESSION FOR MAIN MEMORY COLUMN STORES
SIGMOD 2009

DICTIONARY COMPRESSION

When to construct the dictionary?

What should the scope be of the dictionary?

How do we allow for range queries?

How do we enable fast encoding/decoding?

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.

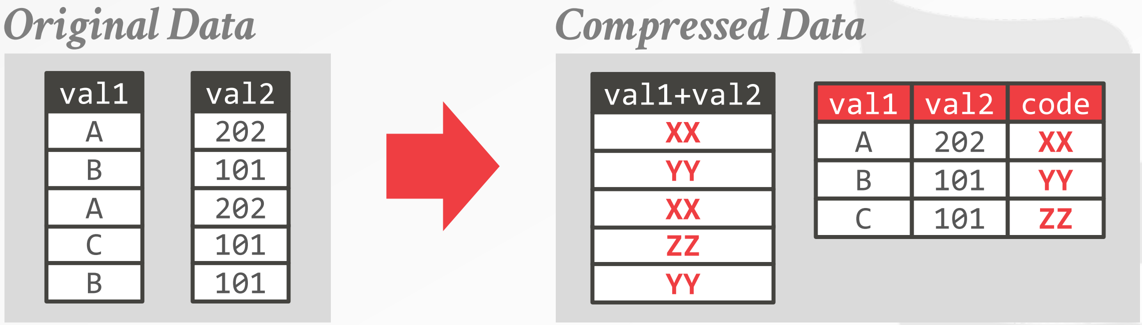
Choice #3: Multi-Table

→ 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.



Original Data

val1
A
B
A
C
B

val2
202
101
202
101
101

Compressed Data

val1+val2
XX
YY
XX
ZZ
YY

val1	val2	code
A	202	XX
B	101	YY
C	101	ZZ

ENCODING / DECODING

A dictionary needs to support two operations:

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

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

No magic hash function will do this for us.

We need two data structures to support operations in both directions.

ORDER-PRESERVING COMPRESSION

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

Original Data

name
Andrea
Joy
Andy
Dana

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andy	20
Dana	30
Joy	40

ORDER-PRESERVING COMPRESSION

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
Joy
Andy
Dana

Compressed Data

name
10
40
20
30

value	*code*
Andrea	10
Andy	20
Dana	30
Joy	40

ORDER-PRESERVING COMPRESSION

```
SELECT name FROM users
 WHERE name LIKE ‘And%’
```

Still have to perform seq scan

Original Data

name
Andrea
Joy
Andy
Dana

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andy	20
Dana	30
Joy	40

ORDER-PRESERVING COMPRESSION

```
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
Joy
Andy
Dana

Compressed Data

name
10
40
20
30

value	code
Andrea	10
Andy	20
Dana	30
Joy	40

DICTIONARY IMPLEMENTATIONS

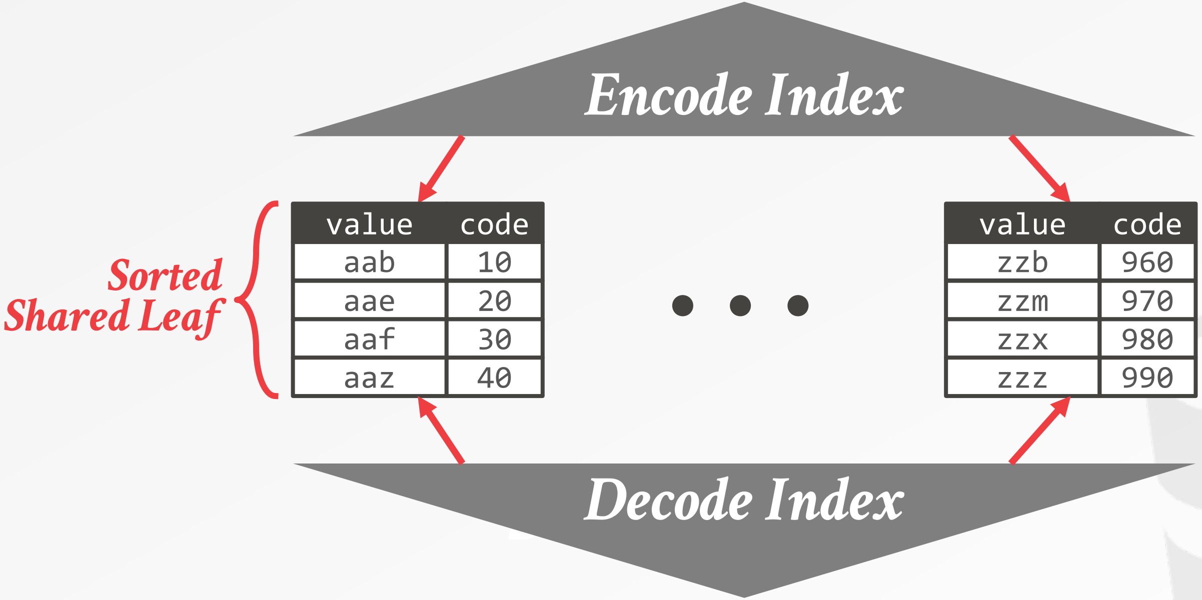
Hash Table:

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

B+Tree:

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

SHARED-LEAVES TREES



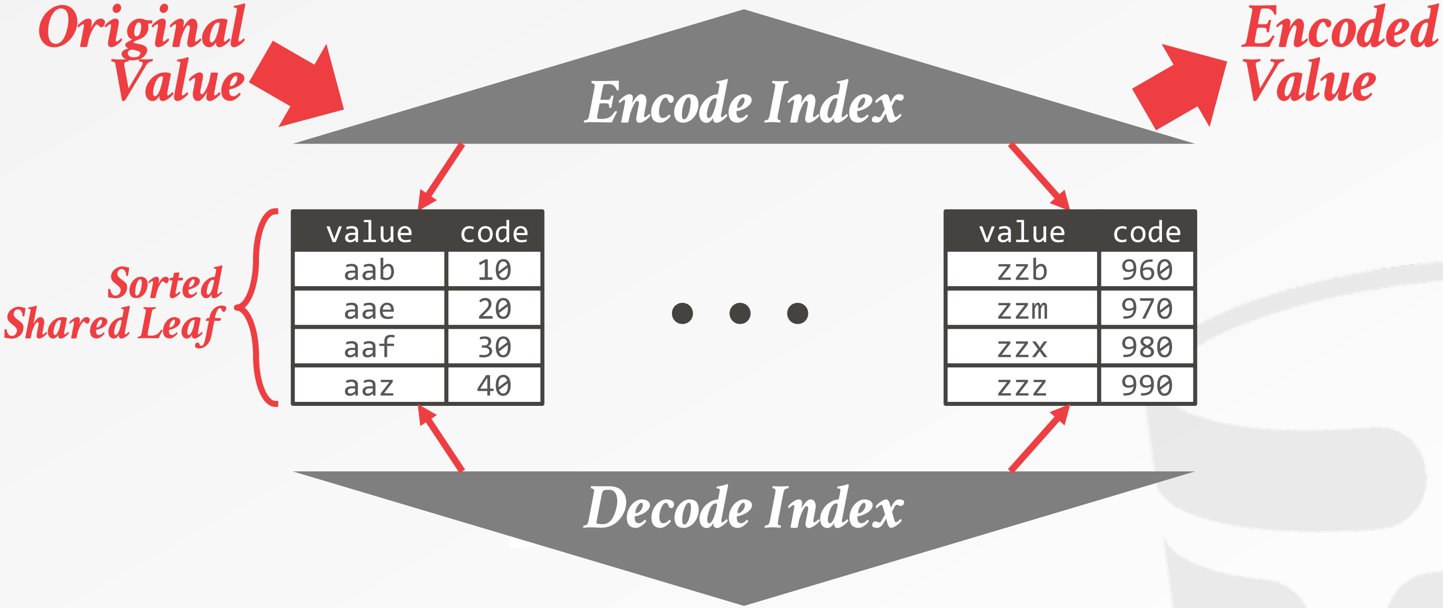
value	code
aab	10
aae	20
aaf	30
aaz	40

value	code
zzb	960
zzm	970
zzx	980
zzz	990

DICTIONARY-BASED ORDER-PRESERVING STRING COMPRESSION FOR MAIN MEMORY COLUMN STORES
SIGMOD 2009

CARNEGIE MELLON
DATABASE GROUP

SHARED-LEAVES TREES

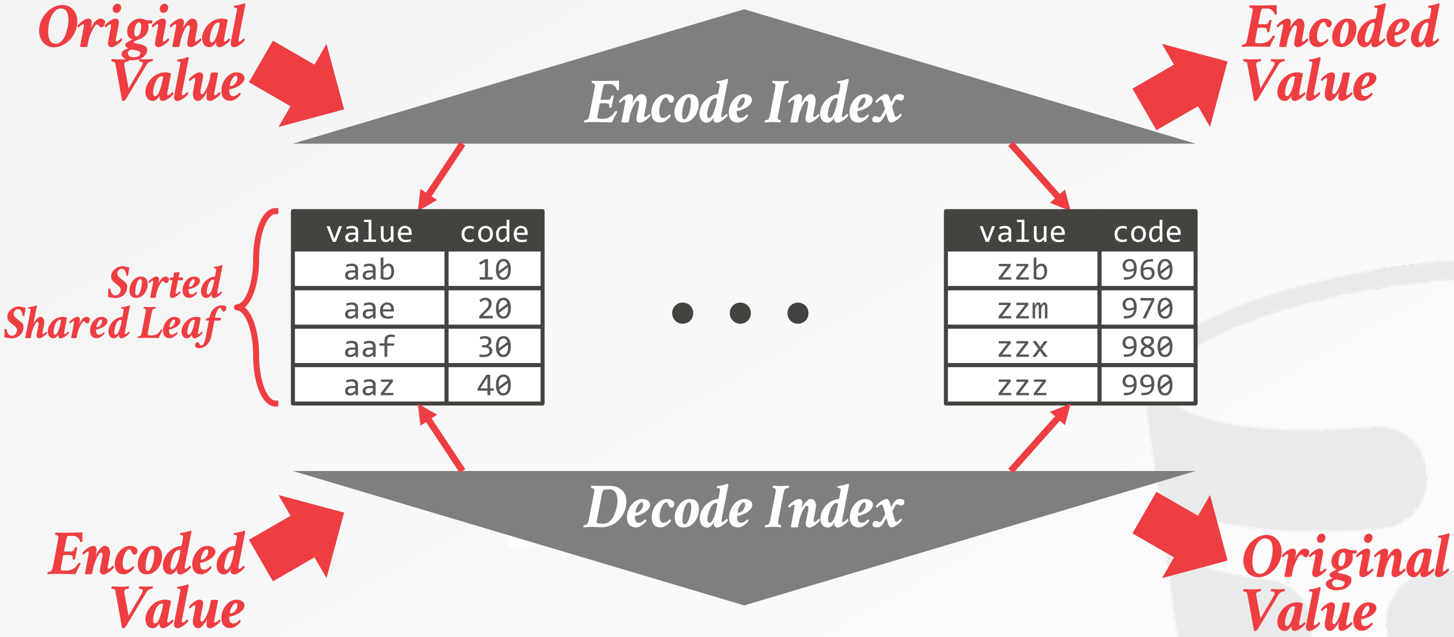


value	code
aab	10
aae	20
aaf	30
aaz	40

value	code
zzb	960
zzm	970
zzx	980
zzz	990

DICTIONARY-BASED ORDER-PRESERVING STRING COMPRESSION FOR MAIN MEMORY COLUMN STORES
SIGMOD 2009

SHARED-LEAVES TREES



Original Value

Encode Index

Encoded Value

Sorted Shared Leaf

value	code
aab	10
aae	20
aaf	30
aaz	40

• • •

value	code
zzb	960
zzm	970
zzx	980
zzz	990

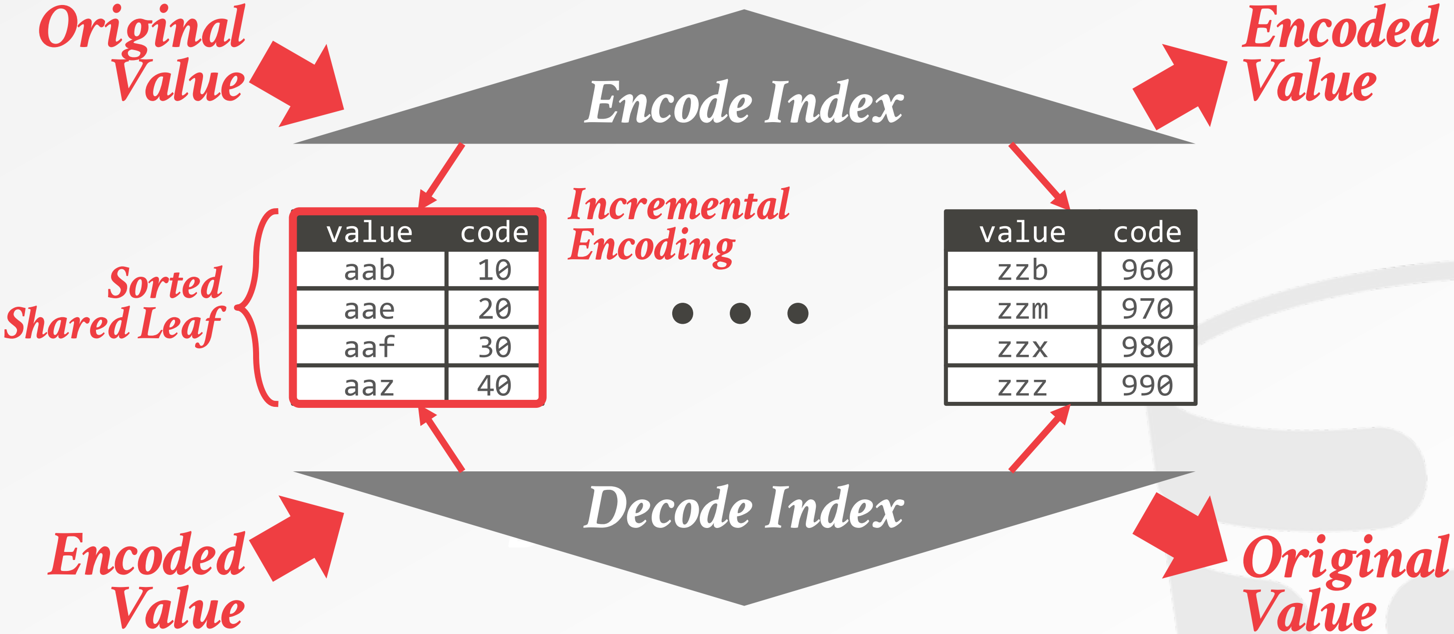
Encoded Value

Decode Index

Original Value

DICTIONARY-BASED ORDER-PRESERVING STRING COMPRESSION FOR MAIN MEMORY COLUMN STORES
SIGMOD 2009

SHARED-LEAVES TREES



value	code
aab	10
aae	20
aaf	30
aaz	40

value	code
zzb	960
zzm	970
zzx	980
zzz	990

DICTIONARY-BASED ORDER-PRESERVING STRING COMPRESSION FOR MAIN MEMORY COLUMN STORES
SIGMOD 2009

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%	**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.

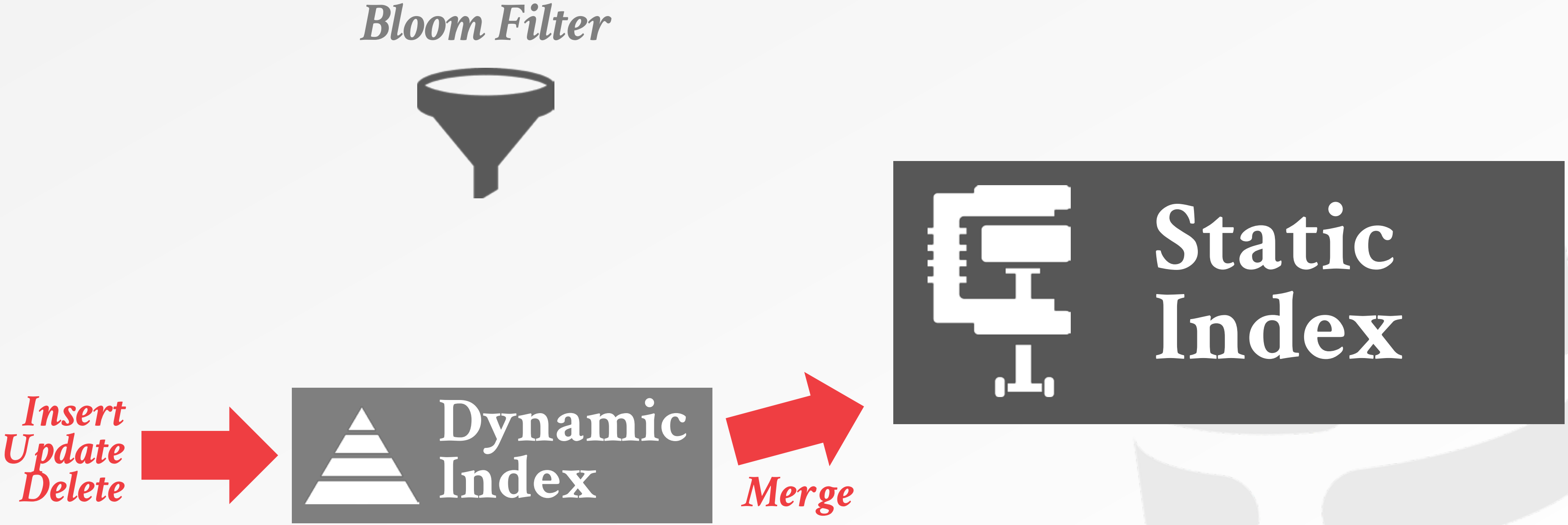
→ **Static Stage:** Old data, compressed + read-only.

All updates go to dynamic stage.

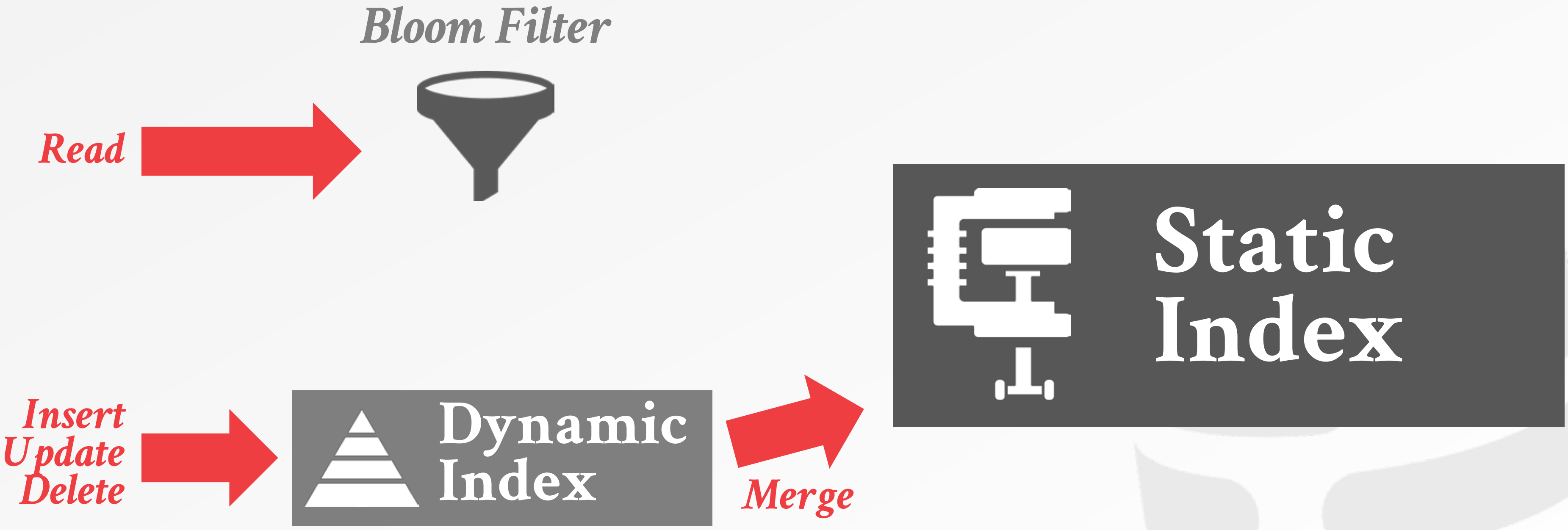
Reads may need to check both stages.

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

HYBRID INDEXES



HYBRID INDEXES


Bloom Filter

Read

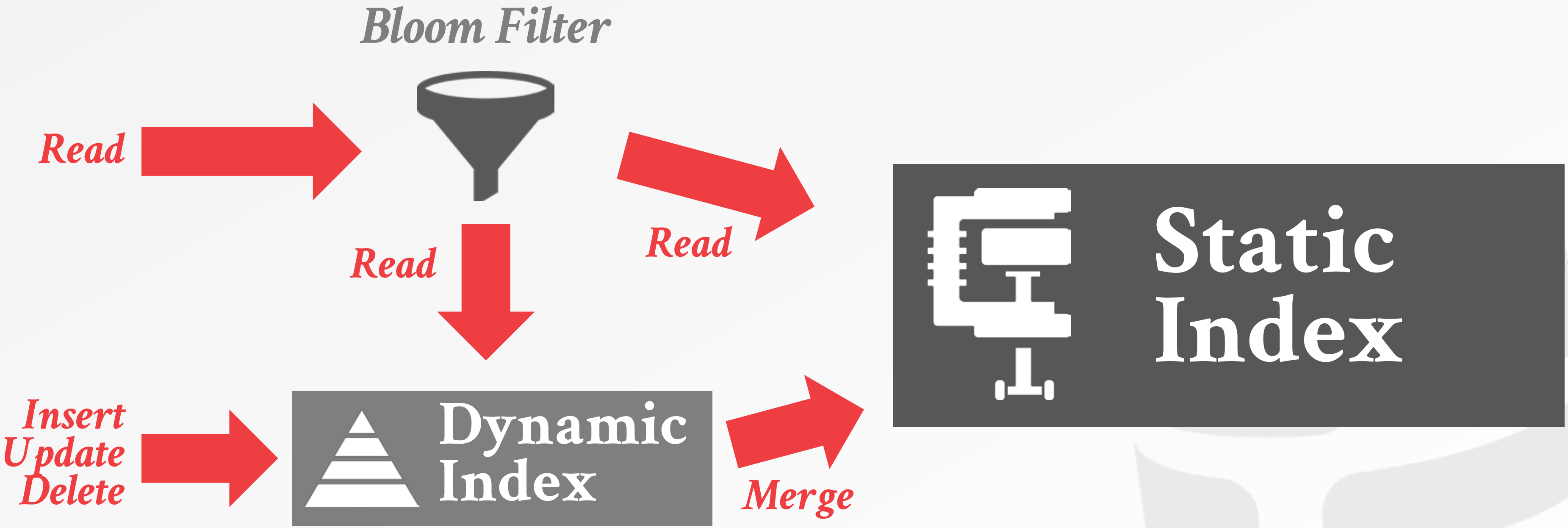
Insert
Update
Delete

Dynamic Index

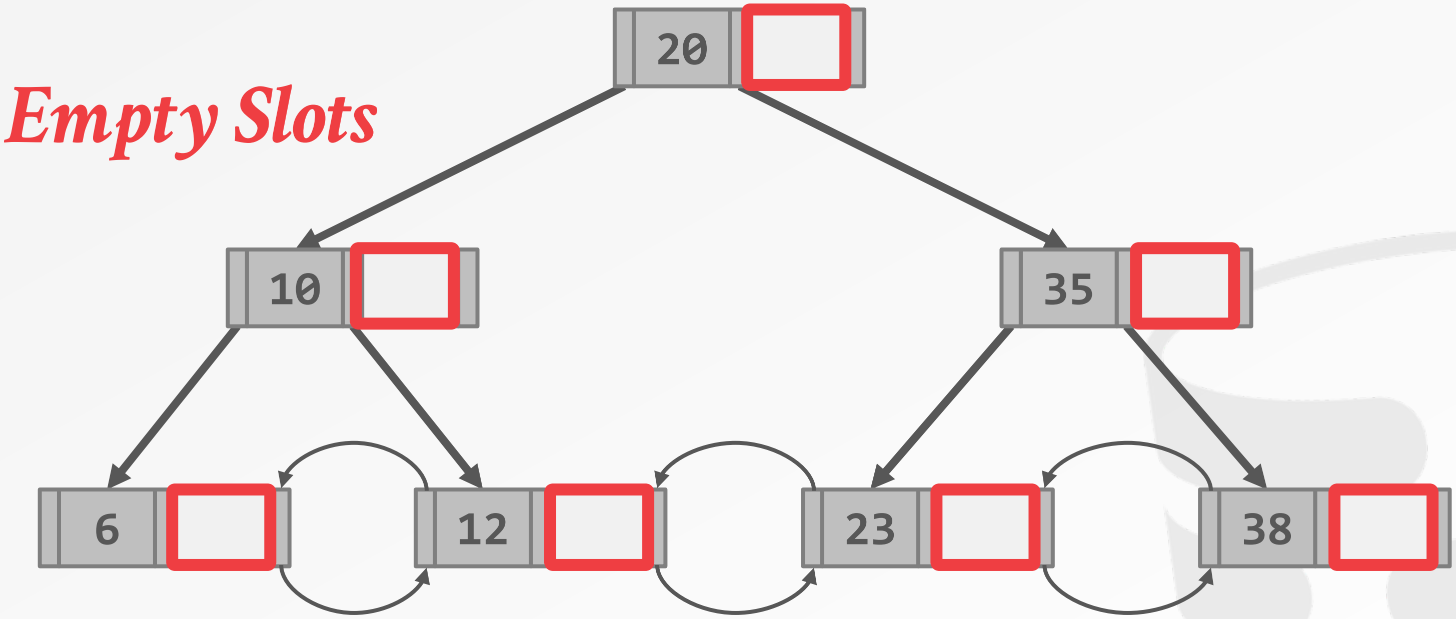
Merge

Static Index

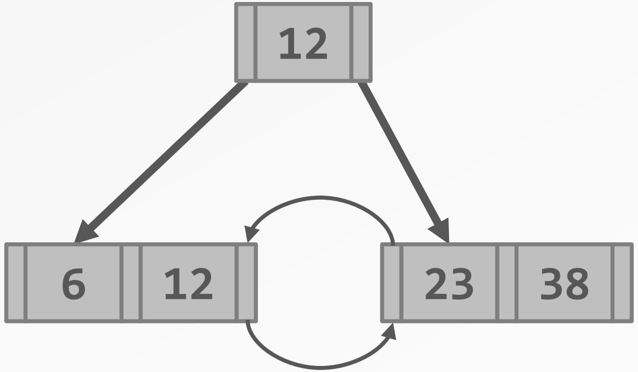
HYBRID INDEXES



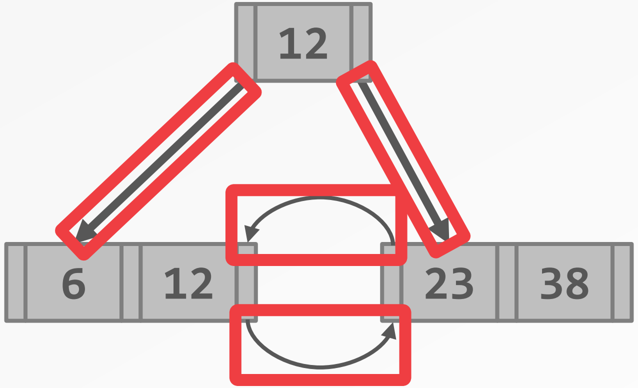
COMPACT B+TREE



COMPACT B+TREE

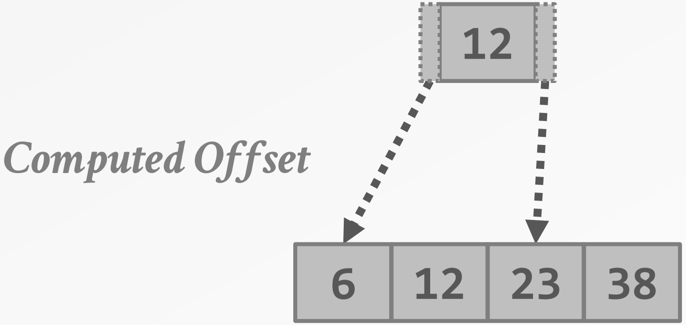


COMPACT B+TREE



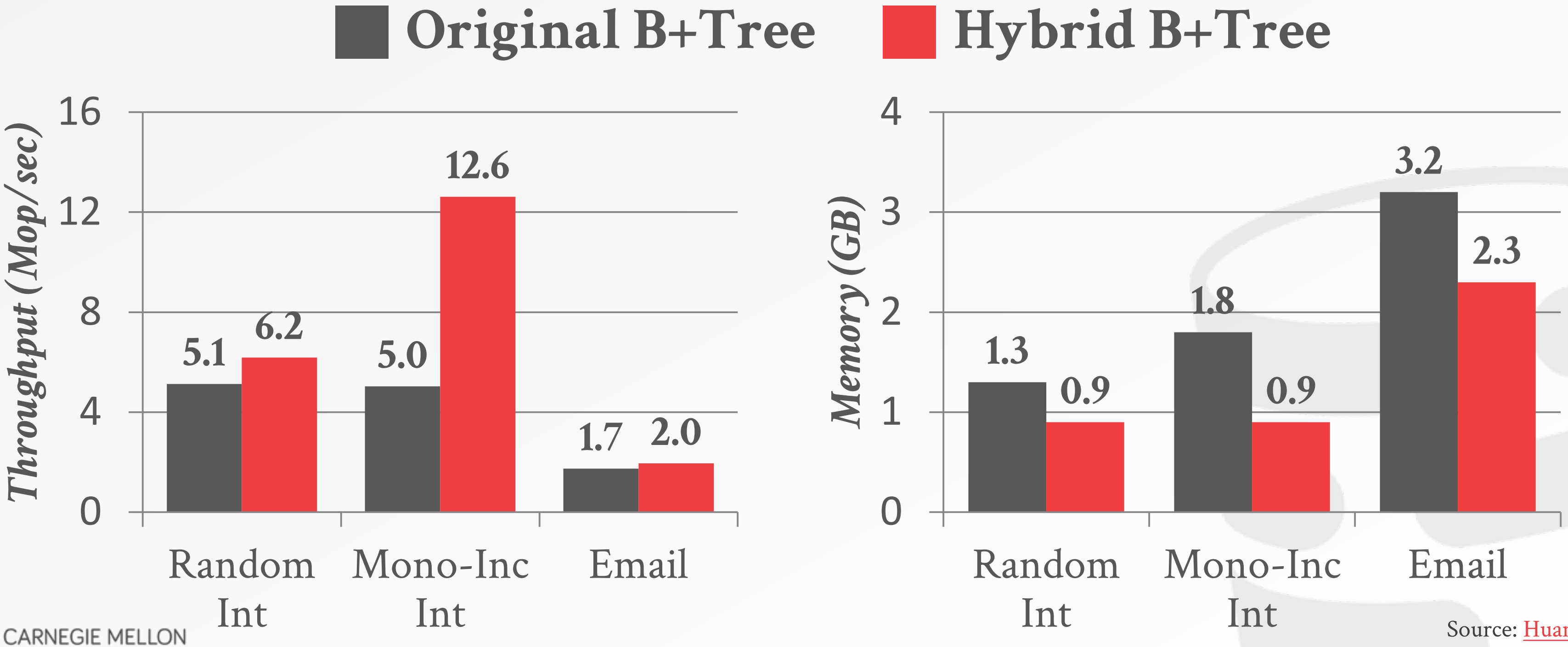
Pointers

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

Physical vs. Logical Logging