



# Lecture 9: Compression

CREATING THE NEXT®

# Administrivia

---

- Please add team details in spreadsheet if not already done.
- Project proposals in class.

# Today's Agenda

---

Recap

Compression Background

Naïve Compression

Columnar Compression

Dictionary Compression

# Recap

# Thread Safety

---

- A piece of code is **thread-safe** if it functions correctly during simultaneous execution by multiple threads.
- In particular, it must satisfy the need for multiple threads to access the same shared data (**shared access**), and
- the need for a shared piece of data to be accessed by only one thread at any given time (**exclusive access**)

## 2Q Policy

---

Maintain two queues (FIFO and LRU)

- Some pages are accessed only once (*e.g.*, sequential scan)
  - Some pages are hot and accessed frequently
  - Maintain separate lists for those pages
  - **Scan resistant** policy
- 1 Maintain all pages in FIFO queue
  - 2 When a page that is currently in FIFO is referenced again, upgrade it to the LRU queue
  - 3 Prefer evicting pages from FIFO queue

Hot pages are in LRU, read-once pages in FIFO.

# Compression Background

## Observation

---

- I/O is the main bottleneck if the DBMS has to fetch data from disk
- Database compression will reduce the number of pages
  - ▶ So, fewer I/O operations (lower disk bandwidth consumption)
  - ▶ But, may need to decompress data (CPU overhead)



# Observation

---

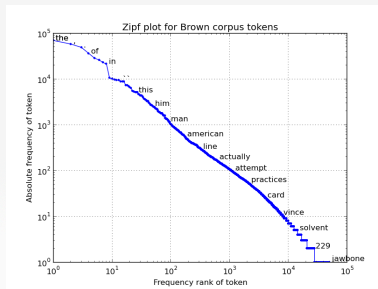
Key trade-off is decompression speed vs. compression ratio

- Disk-centric DBMS tend to optimize for compression ratio
- In-memory DBMSs tend to optimize for decompression speed. Why?
- Database compression reduces DRAM footprint and bandwidth consumption.

# Real-World Data Characteristics

---

- Data sets tend to have highly skewed distributions for attribute values.
  - ▶ Example: Zipfian distribution of the **Brown Corpus**



# Real-World Data Characteristics

---

- 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**, **Oracle**
- Approach 2: Zone Maps (Lossless)
  - ▶ 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.

```
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 compress data, the first question is what data do want to compress.
- This determines what compression schemes are available to us



# Compression Granularity

---

- Choice 1: Block-level
  - ▶ Compress a block of tuples of the same table.
- Choice 2: Tuple-level
  - ▶ Compress the contents of the entire tuple (**NSM-only**).
- Choice 3: Value-level
  - ▶ Compress a single attribute value within one tuple.
  - ▶ Can target multiple attribute values within the same tuple.
- Choice 4: Column-level
  - ▶ Compress multiple values for one or more attributes stored for multiple tuples (**DSM-only**).

# Naïve Compression

# Naïve Compression

---

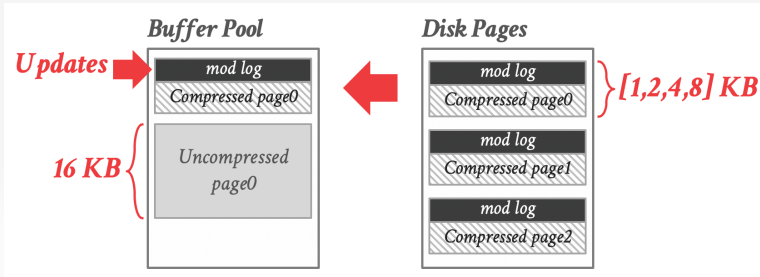
- Compress data using a general-purpose algorithm.
- Scope of compression is only based on the type of data provided as input.
- Encoding uses a dictionary of commonly used words
  - ▶ LZ4 (2011)
  - ▶ Brotli (2013)
  - ▶ Zstd (2015)
- Consideration
  - ▶ Compression vs. decompression speed.

# 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 the segment in the original data with a reference to the segment's position in the dictionary data structure.

## Case Study: MySQL InnoDB Compression



# Naïve Compression

---

- The DBMS must decompress data first before it can be read and (potentially) modified.
  - ▶ This limits the “complexity” 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 trickier...

```
SELECT *  
FROM Artists  
WHERE name = 'Mozart'
```

	<u>Artist</u>	<u>Year</u>
Original Table	Mozart	1756
	Beethoven	1770

```
SELECT *  
FROM Artists  
WHERE name = 1
```

	<u>Artist</u>	<u>Year</u>
Compressed Table	1	1756
	2	1770

# Columnar Compression



# Columnar Compression

---

- Null Suppression
- 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.
- Reference: [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 number of elements in the run.
- Requires the columns to be sorted intelligently to maximize compression opportunities.
- Reference: [Database Compression \(SIGMOD Record, 1993\)](#)

# 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	<i>RLE Triplet</i>
8	- Value
9	- Offset
	- Length

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

# 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	<i>RLE Triplet</i>
8	- Value
9	- Offset
	- Length

## Bitmap Encoding

---

- Store a separate bitmap for **each unique value** for an attribute where each bit in the bitmap corresponds to the value of the attribute in 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 **cardinality** of the attribute is small.
- Reference: **MODEL 204 architecture and performance (HPTS, 1987)**

# Bitmap Encoding

*Original Data*

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

$9 \times 8\text{-bits} = 72\text{ bits}$

*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

$2 \times 8\text{-bits} = 16\text{ bits}$

$9 \times 2\text{-bits} = 18\text{ bits}$

## Bitmap Encoding: Analysis

---

```
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.
  - ▶  $10000000 \times 32\text{-bits} = 40 \text{ MB}$
  - ▶  $10000000 \times 43000 = 53.75 \text{ GB}$
- Every time a txn inserts a new tuple, the DBMS must extend 43,000 different bitmaps.



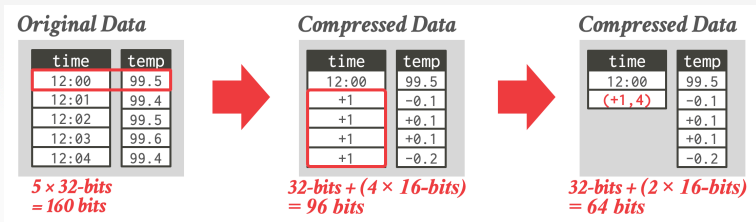
# Bitmap Encoding: Compression

---

- Approach 1: General Purpose Compression
  - ▶ Use standard compression algorithms (*e.g.*, LZ4, Snappy).
  - ▶ The DBMS must decompress before it can use the data to process a query.
  - ▶ Not useful for in-memory DBMSs.
- Approach 2: Byte-aligned Bitmap Codes
  - ▶ Structured run-length encoding compression.

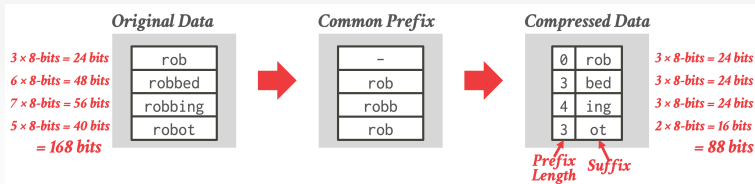
# Delta Encoding

- 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.



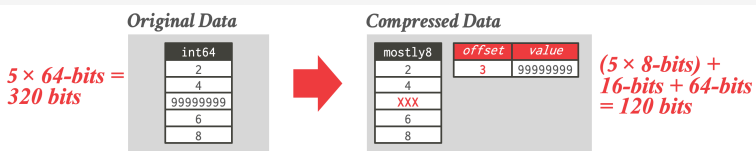
# Incremental Encoding

- Variant of delta encoding that avoids duplicating common prefixes/suffixes between consecutive tuples.
- This works best with sorted data.



# Mostly Encoding

- When values for an attribute are **mostly** less than the largest possible size for that attribute's data type, store them with a more compact data type.
  - ▶ The remaining values that cannot be compressed are stored in their raw form.
  - ▶ Reference: [Amazon Redshift Documentation](#)



# Dictionary Compression

# Dictionary Compression

---

- Probably the most useful compression scheme because it does not require pre-sorting.
- 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 Compression: Design Decisions

---

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



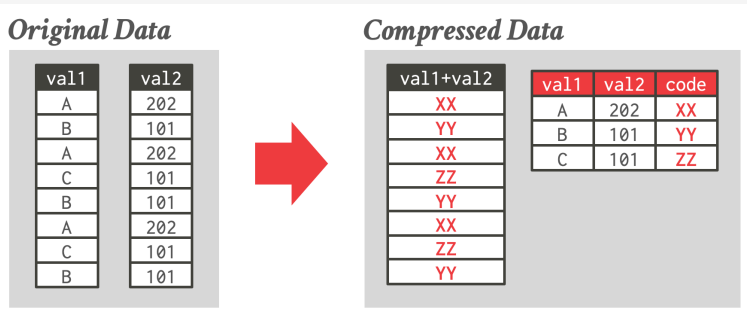
# Dictionary Scope

---

- 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. Why?
- 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 implements this.



# 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.

# Order-Preserving Encoding

---

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

```
SELECT *  
FROM Artists  
WHERE name LIKE 'M%'
```

	<u>Artist</u>	<u>Year</u>
Original Table	Mozart	1756
	Max Bruch	1838
	Beethoven	1770

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

	<u>Artist</u>	<u>Year</u>
Compressed Table	10	1756
	20	1838
	30	1770

# Order-Preserving Encoding

---

```
SELECT Artist
FROM Artists
WHERE name LIKE 'M%' -- Must still perform sequential scan

SELECT DISTINCT Artist
FROM Artists
WHERE name LIKE 'M%' -- ??
```

# 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.

## Conclusion

---

- 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.
- In the next lecture, we will learn about larger-than-memory databases (advanced lecture).