Intro to Database Systems (15-445/645)

05 Storage Models & Compression

Carnegie Mellon University

SPRING 2023

Charlie Garrod
Homework 1 due Friday (Feb 3rd).

Homework 2 available Monday, due February 17th.

Project 1 available, due February 19th.

Don’t forget to turn in the collaboration policy.
DATABASE WORKLOADS

On-Line Transaction Processing (OLTP)
→ Fast operations that only read/update a small amount of data each time.

On-Line Analytical Processing (OLAP)
→ Complex queries that read a lot of data to compute aggregates.

Hybrid Transaction + Analytical Processing
→ OLTP + OLAP together on the same database instance
DATABASE WORKLOADS

<table>
<thead>
<tr>
<th>Workload Focus</th>
<th>Operation Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTAP</td>
<td>Simple</td>
</tr>
<tr>
<td>OLTP</td>
<td>Simple</td>
</tr>
<tr>
<td>OLAP</td>
<td>Complex</td>
</tr>
</tbody>
</table>

Source: Mike Stonebraker
CREATE TABLE useracct (  
  userID INT PRIMARY KEY,  
  userName VARCHAR UNIQUE,  
  );

CREATE TABLE pages (  
  pageID INT PRIMARY KEY,  
  title VARCHAR UNIQUE,  
  latest INT  
  REFERENCES revisions (revID),  
  );

CREATE TABLE revisions (  
  revID INT PRIMARY KEY,  
  userID INT REFERENCES useracct (userID),  
  pageID INT REFERENCES pages (pageID),  
  content TEXT,  
  updated DATETIME  
  );
The relational model does not specify that the DBMS must store all a tuple's attributes together in a single page.

This may not actually be the best layout for some workloads…
OLTP

On-line Transaction Processing:
→ Simple queries that read/update a small amount of data that is related to a single entity in the database.

This is usually the kind of application that people build first.

**SELECT** P.*, R.*
**FROM** pages AS P
**INNER JOIN** revisions AS R
**ON** P.latest = R.revID
**WHERE** P.pageID = ?

**UPDATE** useracct
**SET** lastLogin = NOW(),
hostname = ?
**WHERE** userID = ?

**INSERT INTO** revisions **VALUES** (?, ?, ..., ?)
On-line Analytical Processing:
→ Complex queries that read large portions of the database spanning multiple entities.

You execute these workloads on the data you have collected from your OLTP application(s).

```sql
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
FROM useracct AS U
WHERE U.hostname LIKE '%.gov'
GROUP BY
       EXTRACT(month FROM U.lastLogin)
```
DATA STORAGE MODELS

The DBMS can store tuples in different ways that are better for either OLTP or OLAP workloads.

We have been assuming the *n-ary storage model* (aka "row storage") so far this semester.
The DBMS stores all attributes for a single tuple contiguously in a page.

Ideal for OLTP workloads where queries tend to operate only on an individual entity and insert-heavy workloads.
N-ARY STORAGE MODEL (NSM)

The DBMS stores all attributes for a single tuple contiguously in a page.
**N-ARY STORAGE MODEL (NSM)**

```
SELECT * FROM useracct
WHERE userName = ?
AND userPass = ?

INSERT INTO useracct
VALUES (?, ?, ..., ?)
```
N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
FROM useracct AS U
WHERE U.hostname LIKE '%.gov'
GROUP BY EXTRACT(month FROM U.lastLogin)
```
N-ARY STORAGE MODEL

Advantages
→ Fast inserts, updates, and deletes.
→ Good for queries that need the entire tuple.

Disadvantages
→ Not good for scanning large portions of the table and/or a subset of the attributes.
DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute for all tuples contiguously in a page. → Also known as a "column store"

Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table’s attributes.
DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute across multiple tuples contiguously in a page. → Also known as a "column store".
DECOMPOSITION STORAGE MODEL (DSM)

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
FROM useracct AS U
WHERE U.hostname LIKE '%.gov'
GROUP BY EXTRACT(month FROM U.lastLogin)
```
TUPLE IDENTIFICATION

Choice #1: Fixed-length Offsets
→ Each value is the same length for an attribute.

Choice #2: Embedded Tuple Ids
→ Each value is stored with its tuple id in a column.
DECOMPOSITION STORAGE MODEL (DSM)

**Advantages**
- Reduces the amount wasted I/O because the DBMS only reads the data that it needs.
- Better query processing and data compression (more on this later).

**Disadvantages**
- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.
DSM SYSTEM HISTORY

1970s: Cantor DBMS
1980s: DSM Proposal
1990s: SybaseIQ (in-memory only)
2000s: Vertica, VectorWise, MonetDB
2010s: Everyone
OBSERVATION

I/O is the main bottleneck if the DBMS fetches data from disk during query execution.

The DBMS can compress pages to increase the utility of the data moved per I/O operation.

Key trade-off is speed vs. compression ratio
→ Compressing the database reduces DRAM requirements.
→ It may decrease CPU costs during query execution.
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 must be performed at the application level.
**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 within one tuple (overflow).
→ 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.


Considerations
→ Computational overhead
→ Compress vs. decompress speed.
MYSQL INNODB COMPRESSION

Buffer Pool

<table>
<thead>
<tr>
<th>Uncompressed Page₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod log</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compressed Page₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod log</td>
</tr>
</tbody>
</table>

Disk Pages

<table>
<thead>
<tr>
<th>mod log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Page₀</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mod log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Page₁</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mod log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Page₂</td>
</tr>
</tbody>
</table>

Updates

16 KB

[1,2,4,8] KB

Source: MySQL 5.7 Documentation
NAÏVE COMPRESSION

The DBMS must decompress data 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

Ideally, we want the DBMS to operate on compressed data without decompressing it first.
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 within one tuple (overflow).
→ 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).
COLUMNAR COMPRESSION

Run-length Encoding
Bit-Packing Encoding
Bitmap Encoding
Delta Encoding
Incremental Encoding
Dictionary Encoding
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**

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

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

**Sorted Data**

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

**Compressed Data**

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(M,0,6)</td>
</tr>
<tr>
<td>2</td>
<td>(F,7,2)</td>
</tr>
<tr>
<td>3</td>
<td>(M,4,1)</td>
</tr>
<tr>
<td>6</td>
<td>(F,5,1)</td>
</tr>
<tr>
<td>8</td>
<td>(M,6,2)</td>
</tr>
</tbody>
</table>

*RLE Triplet*
- Value
- Offset
- Length
BIT-PACKING ENCODING

When values for an attribute are always less than the value's declared largest size, store them as smaller data type.

Original Data

\[
\begin{array}{cccc}
4 & 2 & 6 & 45 \\
18 &\end{array}
\]

\[
5 \times 64\text{-bits} = 320\text{ bits}
\]

Compressed Data

\[
\begin{array}{cccc}
00000010 & 00000011 & 0011101 & 010010 \\
00000010 & 00000011 & 0011101 & 00001110 & 0010010 \\
\end{array}
\]

\[
(5 \times 8\text{-bits}) = 40\text{ bits}
\]
MOSTLY ENCODING

Bit-packing variant that uses a special marker to indicate when a value exceeds largest size and then maintain a look-up table to store them.

Original Data

<table>
<thead>
<tr>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>99999999</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>18</td>
</tr>
</tbody>
</table>

Compressed Data

<table>
<thead>
<tr>
<th>mostly8</th>
<th>offset</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>99999999</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

$5 \times 64\text{-bits} = 320\text{ bits}$

$(5 \times 8\text{-bits}) + 16\text{-bits} + 64\text{-bits} = 120\text{ bits}$

Source: Redshift Documentation
Store a separate bitmap for each unique value for an 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.

Some DBMSs provide bitmap indexes.
**BITMAP ENCODING**

Original Data:

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>M</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>M</td>
</tr>
</tbody>
</table>

Compressed Data:

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

- **9 × 8-bits = 72 bits**
- **2 × 8-bits = 16 bits**
- **9 × 2-bits = 18 bits**
BITMAP ENCODING: EXAMPLE

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 the application inserts a new tuple, the DBMS must extend 43,000 different bitmaps.

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

<table>
<thead>
<tr>
<th>time</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:00</td>
<td>99.5</td>
</tr>
<tr>
<td>12:01</td>
<td>99.4</td>
</tr>
<tr>
<td>12:02</td>
<td>99.5</td>
</tr>
<tr>
<td>12:03</td>
<td>99.6</td>
</tr>
<tr>
<td>12:04</td>
<td>99.4</td>
</tr>
</tbody>
</table>

Compressed Data

<table>
<thead>
<tr>
<th>time</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:00</td>
<td>99.5</td>
</tr>
<tr>
<td>+1</td>
<td>-0.1</td>
</tr>
<tr>
<td>+1</td>
<td>+0.1</td>
</tr>
<tr>
<td>+1</td>
<td>+0.1</td>
</tr>
<tr>
<td>+1</td>
<td>-0.2</td>
</tr>
</tbody>
</table>

5 × 32-bits = 160 bits
32-bits + (4 × 16-bits) = 96 bits

Compressed Data

<table>
<thead>
<tr>
<th>time</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:00</td>
<td>99.5</td>
</tr>
<tr>
<td>(+1,4)</td>
<td>-0.1</td>
</tr>
<tr>
<td>+0.1</td>
<td></td>
</tr>
<tr>
<td>+0.1</td>
<td></td>
</tr>
<tr>
<td>-0.2</td>
<td></td>
</tr>
</tbody>
</table>

32-bits + (2 × 16-bits) = 64 bits
INCREMENTAL ENCODING

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

Original Data

Common Prefix

Compressed Data

- rob
  robbed
  robbing
  robot

- rob
  robb

0 rob
3 bed
4 ing
3 ot

$3 \times 8\text{-bits} = 24\text{ bits}$
$6 \times 8\text{-bits} = 48\text{ bits}$
$7 \times 8\text{-bits} = 56\text{ bits}$
$5 \times 8\text{-bits} = 40\text{ bits}$

$= 168\text{ bits}$

$3 \times 8\text{-bits} = 24\text{ bits}$
$3 \times 8\text{-bits} = 24\text{ bits}$
$3 \times 8\text{-bits} = 24\text{ bits}$
$2 \times 8\text{-bits} = 16\text{ bits}$

$= 88\text{ bits}$

$4 \times 8\text{-bits} = 32\text{ bits}$
DICTIONARY COMPRESSION

Build a data structure that maps variable-length values to a smaller integer identifier.
Replace those values with their corresponding identifier in the dictionary data structure.
→ Need to support fast encoding and decoding.
→ Need to also support range queries.

Most widely used compression scheme in DBMSs.
DICTIONARY COMPRESSION

SELECT * FROM users
WHERE name = 'Andy'

SELECT * FROM users
WHERE name = 30

Original Data

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
</tr>
<tr>
<td>Prashanth</td>
</tr>
<tr>
<td>Andy</td>
</tr>
<tr>
<td>Matt</td>
</tr>
<tr>
<td>Prashanth</td>
</tr>
</tbody>
</table>

Compressed Data

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Prashanth</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Andy</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Matt</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Prashanth</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
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 typical hash function will do this for us.
ORDER-PRESERVING ENCODING

The encoded values need to support the same collation as the original values.

Original Data

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
</tr>
<tr>
<td>Prashanth</td>
</tr>
<tr>
<td>Andy</td>
</tr>
<tr>
<td>Matt</td>
</tr>
<tr>
<td>Prashanth</td>
</tr>
</tbody>
</table>

Compressed Data

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Andrea</td>
<td>10</td>
</tr>
<tr>
<td>40</td>
<td>Andy</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>Matt</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>Prashanth</td>
<td>40</td>
</tr>
</tbody>
</table>
**ORDER-PRESERVING ENCODING**

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

Still must perform scan on column

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

Only need to access dictionary

Original Data

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
</tr>
<tr>
<td>Prashanth</td>
</tr>
<tr>
<td>Andy</td>
</tr>
<tr>
<td>Matt</td>
</tr>
<tr>
<td>Prashanth</td>
</tr>
</tbody>
</table>

Compressed Data

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Andy</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Matt</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Prashanth</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

Sorted Dictionary
CONCLUSION

It is important to choose the right storage model for the target workload:
→ OLTP = Row Store
→ OLAP = Column Store

DBMSs can combine different approaches for even better compression.
Dictionary encoding is probably the most useful scheme because it does not require pre-sorting.
Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.