OLAP Indexes
LAST CLASS

We discussed the advantages of the columnar storage model via PAX for OLAP workloads.

All attributes in a columnar database must be fixed-length to enable offset addressing.

If the DBMS assumes that its data files are immutable, it enables several optimization opportunities.
**OBSERVATION**

OLTP DBMSs use indexes to find individual tuples without performing sequential scans.

→ Tree-based indexes (B+Trees) are meant for queries with low selectivity predicates.
→ Also need to accommodate incremental updates.

But OLAP queries don't necessarily need to find individual tuples and data files are read-only.

How can we speed up sequential scans?
SEQUENTIAL SCAN OPTIMIZATIONS

Data Prefetching
Task Parallelization / Multi-threading
Clustering / Sorting
Late Materialization
Materialized Views / Result Caching
Data Skipping
Data Parallelization / Vectorization
Code Specialization / Compilation
Approach #1: Approximate Queries (Lossy)
→ Execute queries on a sampled subset of the entire table to produce approximate results.
→ Examples: BlinkDB, Redshift, ComputeDB, XDB, Oracle, Snowflake, Google BigQuery, DataBricks

Approach #2: Data Pruning (Loseless)
→ Use auxiliary data structures for evaluating predicates to quickly identify portions of a table that the DBMS can skip instead of examining tuples individually.
→ DBMS must consider trade-offs between scope vs. filter efficacy, manual vs. automatic.
DATA CONSIDERATIONS

Predicate Selectivity
→ How many tuples will satisfy a query's predicates.

Skewness
→ Whether an attribute has all unique values or contain many repeated values.

Clustering / Sorting
→ Whether the table is pre-sorted on the attributes accessed in a query's predicates.
TODAY’S AGENDA

Zone Maps
Bitmap Indexes
Bit-Slicing
Bit-Weaving
Column Imprints
Column Sketches
ZONE MAPS

Pre-computed aggregates for the attribute values in a block of tuples. DBMS checks the zone map first to decide whether it wants to access the block.

→ Originally called **Small Materialized Aggregates** (SMA)
→ DBMS automatically creates/maintains this meta-data.

**Original Data**

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>400</td>
</tr>
</tbody>
</table>

**Zone Map**

<table>
<thead>
<tr>
<th>type</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>100</td>
</tr>
<tr>
<td>MAX</td>
<td>400</td>
</tr>
<tr>
<td>AVG</td>
<td>280</td>
</tr>
<tr>
<td>SUM</td>
<td>1400</td>
</tr>
<tr>
<td>COUNT</td>
<td>5</td>
</tr>
</tbody>
</table>

**QUERY**

```
SELECT * FROM table
WHERE val > 600
```
Parquet: data organization

- Data organization
  - Row-groups (default 128MB)
  - Column chunks
    - Pages (default 1MB)
      - Metadata
      - Min
      - Max
      - Count
      - Rep/def levels
      - Encoded values

SELECT * FROM table
WHERE val > 600
OBSERVATION

Trade-off between scope vs. filter efficacy.
→ If the scope is too large, then the zone maps will be useless.
→ If the scope is too small, then the DBMS will spend too much checking zone maps.

Zone Maps are only useful when the target attribute's position and values are correlated.
BITMAP INDEXES

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.

→ Example: One per row group in PAX.
### BITMAP INDEXES

#### Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>lit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
</tr>
</tbody>
</table>

#### Compressed Data

<table>
<thead>
<tr>
<th>id</th>
<th>lit?</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>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>
BITMAP INDEXES

Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>lit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
</tr>
</tbody>
</table>

Compressed Data

<table>
<thead>
<tr>
<th>id</th>
<th>lit?</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>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>
BITMAP INDEXES: EXAMPLE

Take the intersection of three bitmaps to find matching tuples.

Assume we have 10 million tuples. 43,000 zip codes in the US.
\[10000000 \times 43000 = 53.75\text{GB}\]

This is **wasteful** because most entries in the bitmaps will be zeros.
\[\text{Original: } 10000000 \times 32\text{-bit} = 40\text{MB}\]
BITMAP INDEX: DESIGN CHOICES

Encoding Scheme
→ How to represent and organize data in a Bitmap.

Compression
→ How to reduce the size of sparse Bitmaps.
Approach #1: Equality Encoding
→ Basic scheme with one Bitmap per unique value.

Approach #2: Range Encoding
→ Use one Bitmap per interval instead of one per value.
→ Example: PostgreSQL BRIN

Approach #3: Hierarchical Encoding
→ Use a tree to identify empty key ranges.

Approach #4: Bit-sliced Encoding
→ Use a Bitmap per bit location across all values.
HIERARCHICAL ENCODING

Offsets: 1, 3, 9, 12, 13, 14, 38, 40

HIERARCHICAL BITMAP INDEX: AN EFFICIENT AND SCALABLE INDEXING TECHNIQUE FOR SET-VALUED ATTRIBUTES
ADVANCES IN DATABASES AND INFORMATION SYSTEMS 2003
HIERARCHICAL ENCODING

Offsets: 1, 3, 9, 12, 13, 14, 38, 40

Original: 8 bytes
Encoded: 4 bytes
**BIT-SLICED ENCODING**

**Original Data**

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

**Bit-Slices**

```
null 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 0 0 1 1 0 1 0 0 1 1 0 0 0
```

\[
\text{bin}(21042) \rightarrow 00101001000110010
\]

Source: Jignesh Patel
SELECT * FROM customer_dim WHERE zipcode < 15217

Walk each slice and construct a result bitmap.

Source: Jignesh Patel

16-721 (Spring 2023)
**BIT-SLICED ENCODING**

### Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

### Bit-Slices

|   | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| null | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  |
|      | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  |
|      | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  |
|      | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  |
|      | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  |
|      | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  |

**Walk each slice and construct a result bitmap.**
**Skip entries that have 1 in first 3 slices (16, 15, 14).**

---

Source: [Jignesh Patel](https://www.cs.cmu.edu/~jignesh/)

15-721 (Spring 2023)
Bit-slices can also be used for efficient aggregate computations.

Example: $\text{SUM}(\text{attr})$ using Hamming Weight

$\rightarrow$ First, count the number of 1s in $\text{slice}_{17}$ and multiply the count by $2^{17}$

$\rightarrow$ Then, count the number of 1s in $\text{slice}_{16}$ and multiply the count by $2^{16}$

$\rightarrow$ Repeat for the rest of slices...

Intel added POPCNT SIMD instruction in 2008.
Alternative storage layout for columnar databases that is designed for efficient predicate evaluation on compressed data using SIMD.

- Order-preserving dictionary encoding.
- Bit-level parallelization.
- Only require common instructions (no scatter/gather)

Implemented in Wisconsin’s QuickStep engine.
- Became an Apache Incubator project in 2016 but then died in 2018.
BITWEAVING – STORAGE LAYOUTS

Approach #1: Horizontal
→ Row-oriented storage at the bit-level

Approach #2: Vertical
→ Column-oriented storage at the bit-level
## Horizontal Storage

### Segment #1

| $t_0$ | 0 | 0 | 1 | $=$1 |
| $t_1$ | 1 | 0 | 1 | $=$5 |
| $t_2$ | 1 | 1 | 0 | $=$6 |
| $t_3$ | 0 | 0 | 1 | $=$1 |
| $t_4$ | 1 | 1 | 0 | $=$6 |
| $t_5$ | 1 | 0 | 0 | $=$4 |
| $t_6$ | 0 | 0 | 0 | $=$0 |
| $t_7$ | 1 | 1 | 1 | $=$7 |

### Segment #2

| $t_8$ | 1 | 0 | 0 | $=$4 |
| $t_9$ | 0 | 1 | 1 | $=$3 |
HORIZONTAL STORAGE

Segment #1

Segment #2

Delimiter

Processor Word

Processor Word
**BITWEAVING/H – EXAMPLE**

SELECT * FROM table
WHERE val < 5

- **X** = 0 0 0 1 0 1 1 0
- **Y** = 0 1 0 1 0 1 0 1
- **mask** = 0 1 1 1 0 1 1 1

\[(Y + (X \oplus \text{mask})) \land \neg \text{mask}\] 1 0 0 0 0 0 0 0

- Only requires three instructions to evaluate a single word.
- Works on any word size and encoding length.
- Paper contains algorithms for other operators.

Source: Jignesh Patel
SELECT * FROM table
WHERE val < 5

Source: Jignesh Patel
SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate. → DBMS must convert it into column offsets.

**Approach #1: Iteration**

```java
for (i=0; i<n; i++) {
    if sv[i] == 1
        tuples.add(i);
}
```

**Approach #2: Pre-computed Positions Table**
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate.
→ DBMS must convert it into column offsets.

Approach #1: Iteration
Approach #2: Pre-computed Positions Table
VERTICAL STORAGE

Segment #1

Segment #2

$\begin{array}{cccccccc}
    & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \\
\hline
v_0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
v_1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
v_2 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}$

$\begin{array}{cccccccc}
    & t_8 & t_9 & - & - & - & - & - \\
\hline
v_3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$
VERTICAL STORAGE

Segment #1

Segment #2

Processor Word

Processor Word
### Segment #1

**SQL Query**

```sql
SELECT * FROM table
WHERE val = 2
```

**Data Vectors**

- \(v_0\): 0 1 1 0 1 1 0 1
- \(v_1\): 0 0 1 0 1 0 0 1
- \(v_2\): 1 1 0 1 0 0 0 1

**SIMD Compare Result**

1 0 0 1 0 0 1 0
Can perform early pruning just like in BitMap indexes.

The last vector is skipped because all bits in previous comparison are zero.
TODAY’S AGENDA

Zone Maps
Bitmap Indexes
Bit-Slicing
Bit-Weaving
Column Imprints
Column Sketches
OBSERVATION

All the previous Bitmap schemes were about storing exact/loseless representations of columnar data.

The DBMS could give up some accuracy in exchange for faster evaluation in the common case. → It still must always check the original data to avoid false positives.
COLUMN IMPRINTS

Store a bitmap that indicates whether there is a bit set at a bit-slice of cache-line values.
COLUMN SKETCHES

A variation of range-encoded Bitmaps that uses a smaller sketch codes to indicate that a tuple's value exists in a range.

DBMS must automatically figure out the best mapping of codes.

→ Trade-off between distribution of values and compactness.
→ Assign unique codes to frequent values to avoid false positives.
COLUMN SKETCHES

Original Data

<table>
<thead>
<tr>
<th>val</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>191</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
</tr>
<tr>
<td>231</td>
<td></td>
</tr>
<tr>
<td>172</td>
<td></td>
</tr>
</tbody>
</table>

8-bits

Histogram

<table>
<thead>
<tr>
<th>Frequency</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Domain</td>
<td>0</td>
<td>60</td>
<td>132</td>
<td>170</td>
</tr>
</tbody>
</table>

Compression Map

<table>
<thead>
<tr>
<th>60</th>
<th>132</th>
<th>170</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Sketched Column

<table>
<thead>
<tr>
<th>code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

2-bits

Source: Brian Hentschel
**COLUMN SKETCHES**

**Histogram**

Value Domain:

<table>
<thead>
<tr>
<th>Value</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>60</td>
<td>01</td>
</tr>
<tr>
<td>132</td>
<td>10</td>
</tr>
<tr>
<td>170</td>
<td>11</td>
</tr>
<tr>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

**Compression Map**

SELECT * FROM table
WHERE val < 90

map(90) = 01

**Sketched Column**

```
code  | val
------|-----
00    | 13  |
10    | 191 |
11    | 56  |
00    | 92  |
01    | 81  |
01    | 140 |
10    | 231 |
11    | 172 |
```

Source: Brian Hentschel
PARTING THOUGHTS

Zone Maps are the most widely used method to accelerate sequential scans.

Bitmap indexes are more common in NSM DBMSs than columnar OLAP systems.

We’re ignoring multi-dimensional and inverted indexes...
Data Compression (Tuples, Indexes)
PROJECT #1 – FOREIGN DATA WRAPPER

You will create a foreign data wrapper for PostgreSQL for performing simple scans with predicate pushdown on a custom columnar data format.

The goal is to get you familiar with extending Postgres and writing a simple scan operator.

Due Date: Sunday Feb 26th

https://15721.courses.cs.cmu.edu/spring2023/project1.html
PROJECT #1 – TASKS

We provide a simple columnar data file format. Write a parser for this file that can scan individual columns and stitching tuples back together. Integrate it with Postgres as a foreign table. Add support for predicate evaluation.
DEVELOPMENT HINTS

Write the basic parser code separately first, then connect it to Postgres.

We recommend completing this project in C++ using PGXS.
→ You can try asking Chi about doing it in Rust but he has better things to do in his life...

Gradescope is for meant for grading, not debugging. Write your own local tests.
THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you fork our version of Postgres.

Post your questions on Piazza or come to TA office hours.
Your project implementation must be your own work.
→ You may not copy source code from other groups or the web.

Plagiarism will not be tolerated. See CMU's Policy on Academic Integrity for additional information.
Two-Phase Locking
Isolation Levels