
Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2015/16
Part XI

Online Analytical Processing (OLAP)
**Motivation**

**Scenario:** A bookstore chain collects sales data:

<table>
<thead>
<tr>
<th>Book</th>
<th>City</th>
<th>Month</th>
<th>Units Sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arlington Road Atlas</td>
<td>Arlington</td>
<td>January</td>
<td>134</td>
</tr>
<tr>
<td>Arlington Road Atlas</td>
<td>Arlington</td>
<td>February</td>
<td>327</td>
</tr>
<tr>
<td>Arlington Road Atlas</td>
<td>Springfield</td>
<td>December</td>
<td></td>
</tr>
<tr>
<td>Gone With the Wind</td>
<td>Arlington</td>
<td>January</td>
<td>9</td>
</tr>
<tr>
<td>Tropical Food</td>
<td>Springfield</td>
<td>December</td>
<td>374</td>
</tr>
</tbody>
</table>
Motivation

**Goal:** Spread sheet-style analyses (≈ “Pivot Table”)

<table>
<thead>
<tr>
<th></th>
<th>January</th>
<th>February</th>
<th>⋯</th>
<th>Grand Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arlington</td>
<td>198</td>
<td>449</td>
<td>⋯</td>
<td>1022</td>
</tr>
<tr>
<td>Boston</td>
<td>226</td>
<td>212</td>
<td>⋯</td>
<td>707</td>
</tr>
<tr>
<td>Miami</td>
<td>152</td>
<td>130</td>
<td>⋯</td>
<td>467</td>
</tr>
<tr>
<td>Springfield</td>
<td>304</td>
<td>498</td>
<td>⋯</td>
<td>1303</td>
</tr>
<tr>
<td>Grand Total</td>
<td>880</td>
<td>1289</td>
<td>⋯</td>
<td>3499</td>
</tr>
</tbody>
</table>

**Challenge:** Large data volumes

→ How do we **model** such data (e.g., in a relational system)?
→ How can we **implement** pivot tables efficiently?
→ What about **$k$-dimensional data**?
Idea: Model data as a multi-dimensional cube

Data cube:

- **Facts** are stored as **cells** of the cube.
- Facts have **measures** associated with them (here: sales counts).
- Cells may be empty.

Real-world:

- 4–12 dimensions
- **Project** to 2 or 3 for analysis/viewing
Star Schema:

- One **dimension table** per dimension
- **Fact table** entries reference dimension table entries.

<table>
<thead>
<tr>
<th>Cities</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CityID</td>
<td>City</td>
<td>State</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sales</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BookID</td>
<td>CityID</td>
<td>DayID</td>
<td>Sold</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Books</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BookID</td>
<td>Title</td>
<td>Genre</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DayID</td>
<td>Day</td>
<td>Month</td>
<td>Year</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Star Schema

Fact Table:
- One row per multidimensional fact.
- This table will hold the lion’s share of the entire database.

Dimension Tables:
- **Key**: Artificial key (usually an integer number)
- Typically: One column per level if dimension is **hierarchical**
  → **Redundancy**

OLAP is ran on data **extracted** from transactional system.
- Load data in **batches**; most of it goes into **fact table**.
- Fact table ends up approximately **ordered by date**.
Typical queries: **aggregate** over **sub-ranges** of the full cube.

```
SELECT SUM(Sold)
FROM Sales AS s, Books AS b
WHERE s.BookID = b.BookID
AND b.Title = "Gone..."
```
Roll-Up, Drill-Down, Pivot Tables

Analysts will want to look at aggregates from many different angles.

**Roll-Up / Drill-Down:**
→ For hierarchical dimensions, move up or down the hierarchy
→ See more or less details, “zoom” in or out

**Pivot Tables:**
→ Visualize roll-up/drill-down (≈ dedicated OLAP tools)

<table>
<thead>
<tr>
<th></th>
<th>January</th>
<th>February</th>
<th>…</th>
<th>Grand Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arlington</td>
<td>198</td>
<td>449</td>
<td>…</td>
<td>1022</td>
</tr>
<tr>
<td>Boston</td>
<td>226</td>
<td>212</td>
<td>…</td>
<td>707</td>
</tr>
<tr>
<td>Fiction</td>
<td>121</td>
<td>98</td>
<td>…</td>
<td>346</td>
</tr>
<tr>
<td>Cooking</td>
<td>105</td>
<td>114</td>
<td>…</td>
<td>361</td>
</tr>
<tr>
<td>Miami</td>
<td>152</td>
<td>130</td>
<td>…</td>
<td>467</td>
</tr>
<tr>
<td>Springfield</td>
<td>304</td>
<td>498</td>
<td>…</td>
<td>1303</td>
</tr>
<tr>
<td>Grand Total</td>
<td>880</td>
<td>1289</td>
<td>…</td>
<td>3499</td>
</tr>
</tbody>
</table>
A number of **SQL extensions** ease these tasks.

*E.g.*, multi-dimensional grouping (.pivot table):

```sql
SELECT c.City, t.Month, SUM(s.Sold)
FROM Sales AS s, Cities AS c, Time AS t
WHERE s.DayID = t.DayID AND s.CityID = c.CityID
GROUP BY CUBE (City, Month)
```

→ Likewise: GROUP BY ROLLUP (·)

*E.g.*, ranking, partitioning

```sql
SELECT c.City, t.Day,
    RANK () OVER (PARTITION BY City ORDER BY Sold)
FROM Sales AS s, Cities AS c, Time AS t, Books AS b
WHERE s.DayID = t.DayID AND s.CityID = c.CityID
    AND s.BookID = b.BookID AND b.Title = "Gone...
```
Star Join

The common query pattern is the **star join**.

How will a standard RDBMS execute such a query?
Indexes and Star Queries

**Strategy 1:** Index on value columns of dimension tables

1. For each **dimension table** $D_i$:
   a. Use index to find **matching dimension table rows** $d_{i,j}$.
   b. **Fetch** those $d_{i,j}$ to obtain **key columns** of $D_i$.
   c. Collect a list of **fact table rids** that reference those dimension keys.
   
   🆕 **How?**

2. **Intersect** lists of fact table rids.

3. **Fetch** remaining fact table rows, group, and aggregate.
Indexes and Star Queries

**Strategy 2:** Index on **primary key of dimension tables**

1. **Scan fact table**
2. For each fact table row \( f \):
   a. **Fetch** corresponding dimension table row \( d \).
   b. Check slice and dice conditions on \( d \); skip to next fact table row if predicate not met.
   c. Repeat 2.a for each dimension table.
3. Group and aggregate all remaining fact table rows.
Problems and advantages of Strategy 1?

- Fetch only **relevant** fact table rows (good for selective queries).
- Index → fetch → index → intersect → fetch is cumbersome. ⭐
- **List intersection** is expensive.
  1. Again, lists may be large, intersection small.
  2. Lists are generally **not sorted**.
Problem ★ can be reduced with a trick:

- Create an index that contains value and key column of the dimension table.
  → No fetch needed to obtain dimension key.
- Such indexes allow for index-only querying (↗ slide 174).
  → Acess only index, but not data pages of a table.

E.g.,

```
CREATE INDEX QuarterIndex
ON DateDimension ( Quarter, DateKey )
```

→ Will only use Quarter as a search criterion (but not DateKey).
Problems and advantages of Strategy 2?

+ For small dimension tables, all indexes might fit into memory.
  → On the other hand, indexes might not be worth it; can simply build a hash table on the fly.

− Fact table is **large** → **many** index accesses.

− **Individually**, each dimension predicate may have low selectivity.

  *E.g.*, four dimensions, each restricted with 10% selectivity:
  → Overall selectivity as low as 0.01%.
  → But as many as 10%/1%/... of fact table tuples pass individual dimension filters (and fact table is huge).

  **Together**, dimension predicates may still be highly selective.

• Cost is independent of predicate selectivities.
(Hopefully) dimension subsets are small enough → Hash table(s) fit into memory.

Here, hash joins effectively act like a filter.
Implementing Star Join Using Hash Joins

Problems:

- Which of the filter predicates is most restrictive? — Tough optimizer task!
- A lot of processing time is invested in tuples that are eventually discarded.
- This strategy will have real trouble as soon as not all hash tables fit into memory.
Hash-Based Filters

Use compact bit vector to **pre-filter** data.
Hash-Based Filters

- Size of bit vector is independent of dimension tuple size.
  → And bit vector is much smaller than dimension tuples.
- Filtering may lead to false positives, however.
  → Must still do hash join in the end.
- Key benefit: Discard tuples early.

Nice side effect:
- In practice, will do pre-filtering according to all dimensions involved.
  → Can re-arrange filters according to actual(!) selectivity.
Bloom filters can improve filter efficiency.

**Idea:**

- Create (empty) bit field $B$ with $m$ bits.
- Choose $k$ independent hash functions.
- For every dim. tuple, set $k$ bits in $B$, according to hashed key values.

To probe a fact tuple, check $k$ bit positions

→ Discard tuple if any of these bits is 0.
Bloom Filters

Parameters:

- Number of bits in $B$: $m$
  - Typically measured in “bits per stored entry”
- Number of hash functions: $k$
  - Optimal: about 0.7 times number of bits per entry.
  - Too many hash functions may lead to high CPU load!

Example:

- 10 bits per stored entry lead to a filter accuracy of about 1%.
Microsoft SQL Server uses hash-based pre-filtering since version 2008.
What do you think about this query plan?

Join dimension tables first, then fact table as last relation.
Joins between dimension tables are effectively **Cartesian products**.

→ Clearly won’t work if (filtered) dimension tables are large.
Hub Star Join

Idea:

- Cartesian product approximates the set of foreign key values relevant in the fact table.
- Join Cartesian product with fact table using **index nested loops join** (multi-column index on foreign keys).

Cartesian product approximates the set of foreign key values relevant in the fact table.

Join Cartesian product with fact table using **index nested loops join** (multi-column index on foreign keys).
Hub Star Join

Advantages:

+ Fetch only **relevant** fact table rows.
+ No **intersection** needed.
+ No **sorting** or **duplicate removal** needed.

Down Sides:

- Cartesian product **overestimates** foreign key combinations in the fact table.
  → Many key combinations won’t exist in the fact table.
  → Many unnecessary index probes.

Overall:

- Hub Join works well if **Cartesian product** is small.
### Zigzag Join

#### Cartesian Product of Dimension Keys

<table>
<thead>
<tr>
<th>prodkey</th>
<th>classkey</th>
<th>perkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

#### Multi-Column Index on aroma_sales

<table>
<thead>
<tr>
<th>prodkey</th>
<th>classkey</th>
<th>perkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Unproductive probes which are skipped**
- **Match**

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2015/16
Join Indices

To reduce join cost, we could **pre-compute** (partial) join results.

Database terminology: “materialize”

More generally: “materialized views”

Pre-computed join results are also called **join indices**.

**Example:**  $Cities \bowtie Sales$

- **Type 1:** join key $\rightarrow \langle \{ rid_{Cities}\}, \{ rid_{Sales}\} \rangle$
  
  (Record ids from $Cities$ and $Sales$ that contain given join key value.)

- **Type 2:** $rid_{Cities} \rightarrow \{ rid_{Sales}\}$
  
  (Record ids from $Sales$ that match given record in $Cities$.)

- **Type 3:** $dim \text{ value} \rightarrow \{ rid_{Sales}\}$
  
  (Record ids from $Sales$ that join with $Cities$ tuples that have given dimension value.)

(Conventional $B^+$-trees are often $value \rightarrow \{ rid\}$ mappings; cf. slide 79.)
**Example: Cities ✕ Sales Join Index**

<table>
<thead>
<tr>
<th>rid</th>
<th>CtyID</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁</td>
<td>6371</td>
<td>Arlington</td>
<td>VA</td>
</tr>
<tr>
<td>c₂</td>
<td>6590</td>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>c₃</td>
<td>7882</td>
<td>Miami</td>
<td>FL</td>
</tr>
<tr>
<td>c₄</td>
<td>7372</td>
<td>Springfield</td>
<td>MA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rid</th>
<th>BkID</th>
<th>CtyID</th>
<th>DayID</th>
<th>Sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>372</td>
<td>6371</td>
<td>95638</td>
<td>17</td>
</tr>
<tr>
<td>s₂</td>
<td>372</td>
<td>6590</td>
<td>95638</td>
<td>39</td>
</tr>
<tr>
<td>s₃</td>
<td>1930</td>
<td>6371</td>
<td>95638</td>
<td>21</td>
</tr>
<tr>
<td>s₄</td>
<td>2204</td>
<td>6371</td>
<td>95638</td>
<td>29</td>
</tr>
<tr>
<td>s₅</td>
<td>2204</td>
<td>6590</td>
<td>95638</td>
<td>13</td>
</tr>
<tr>
<td>s₆</td>
<td>1930</td>
<td>7372</td>
<td>95638</td>
<td>9</td>
</tr>
<tr>
<td>s₇</td>
<td>372</td>
<td>7882</td>
<td>65748</td>
<td>53</td>
</tr>
</tbody>
</table>
Star Join with Join Indices

1. For each of the dimensions, find matching Sales rids.
2. Intersect rid lists to determine relevant Sales.
Star Join with Join Indices

The strategy makes **rid list intersection** a critical operation.

→ Rid lists may be **sorted**.

→ Efficient implementation is (still) active research topic.

**Down side:**

- Rid list sorted only for (per-dimension) point lookups.

**Challenge:**

- Efficient **rid list implementation**.
**Idea:** Create **bit vector** for each possible column value.

**Example:** Relation that holds information about students:

<table>
<thead>
<tr>
<th>LegiNo</th>
<th>Name</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>Bachelor</td>
</tr>
<tr>
<td>2345</td>
<td>Marc Johnson</td>
<td>Master</td>
</tr>
<tr>
<td>3456</td>
<td>Rob Mercer</td>
<td>Bachelor</td>
</tr>
<tr>
<td>4567</td>
<td>Dave Miller</td>
<td>PhD</td>
</tr>
<tr>
<td>5678</td>
<td>Chuck Myers</td>
<td>Master</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSc</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

bit vector
Benefit of bitmap indexes:

- Boolean query operations (and, or, etc.) can be performed directly on bit vectors.

```
SELECT  
FROM    Cities
WHERE   State = 'MA'
    AND (City = 'Boston' OR City = 'Springfield')
```

- Bit operations are well-supported by modern computing hardware (SIMD).

\[
B_{MA} \land (B_{Boston} \lor B_{Springfield})
\]
Equality vs. Range Encoding

Alternative encoding for **ordered domains**:

<table>
<thead>
<tr>
<th>LegiNo</th>
<th>Name</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John Smith</td>
<td>3</td>
</tr>
<tr>
<td>2345</td>
<td>Marc Johnson</td>
<td>2</td>
</tr>
<tr>
<td>3456</td>
<td>Rob Mercer</td>
<td>4</td>
</tr>
<tr>
<td>4567</td>
<td>Dave Miller</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semester Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  2  3  4  5</td>
</tr>
<tr>
<td>1  1  1  0  0</td>
</tr>
<tr>
<td>1  1  0  0  0</td>
</tr>
<tr>
<td>1  1  1  1  0</td>
</tr>
<tr>
<td>1  0  0  0  0</td>
</tr>
</tbody>
</table>

(set $B_{c_i}[k] = 1$ for all $c_i$ smaller or equal than the attribute value $a[k]$).

**Why would this be useful?**

**Range predicates** can be evaluated more efficiently:

$$c_i > a[k] \geq c_j \Leftrightarrow (\neg B_{c_i}[k]) \land B_{c_j}[k].$$

(but equality predicates become more expensive).
## Data Warehousing Example

### Bitmap-Index

<table>
<thead>
<tr>
<th>RID</th>
<th>D4.id</th>
<th>D4.product</th>
<th>D4.brand</th>
<th>D4.group</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Latitude E6400</td>
<td>Dell</td>
<td>Computers</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Lenovo T61</td>
<td>Lenovo</td>
<td>Computers</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>SGH-i600</td>
<td>Samsung</td>
<td>Handheld</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>Axim X5</td>
<td>Dell</td>
<td>Handheld</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>i900 OMNIA</td>
<td>Samsung</td>
<td>Mobile</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>XPERIA X1</td>
<td>Sony</td>
<td>Mobile</td>
</tr>
</tbody>
</table>

- **Index:** D4.brand -> {RID}

- **Bitmap Index:** D4.brand
  - B_{Dell}:
    - 1
    - 0
    - 0
    - 0
    - 1
  - B_{Len}:
    - 0
    - 1
    - 0
    - 0
    - 0
  - B_{Sam}:
    - 0
    - 0
    - 1
    - 0
    - 0
  - B_{Sony}:
    - 0
    - 0
    - 0
    - 0
    - 1

- **Index:** D4.group -> {RID}

- **Bitmap Index:** D4.group
  - B_{Com}:
    - 1
    - 0
    - 0
    - 0
    - 0
  - B_{Hand}:
    - 0
    - 0
    - 1
    - 0
    - 0
  - B_{Mob}:
    - 0
    - 0
    - 0
    - 1
    - 1
Sales in group ‘Computers’ for brands ‘Dell’, ‘Lenovo’?

```sql
SELECT SUM(F.price)
FROM D4
WHERE group = 'Computer'
AND (brand = 'Dell'
OR brand = 'Lenovo')
```

→ Calculate bit-wise operation

\[ B_{Com} \land (B_{Dell} \lor B_{Len}) \]

to find matching records.
Bitmap Indices for Star Joins

Bitmap indices are useful to implement **join indices**.

**Here:** Type 2 index for *Cities ⊙ Sales*

<table>
<thead>
<tr>
<th>Cities</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>rid</em></td>
<td>CtyID</td>
<td>City</td>
<td>State</td>
</tr>
<tr>
<td>$c_1$</td>
<td>6371</td>
<td>Arlington</td>
<td>VA</td>
</tr>
<tr>
<td>$c_2$</td>
<td>6590</td>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>$c_3$</td>
<td>7882</td>
<td>Miami</td>
<td>FL</td>
</tr>
<tr>
<td>$c_4$</td>
<td>7372</td>
<td>Springfield</td>
<td>MA</td>
</tr>
</tbody>
</table>

| Sales | | | |
|---|---|---|---|---|
| *rid* | BkID | CtyID | DayID | Sold |
| $s_1$ | 372 | 6371 | 95638 | 17 |
| $s_2$ | 372 | 6590 | 95638 | 39 |
| $s_3$ | 1930 | 6371 | 95638 | 21 |
| $s_4$ | 2204 | 6371 | 95638 | 29 |
| $s_5$ | 2204 | 6590 | 95638 | 13 |
| $s_6$ | 1930 | 7372 | 95638 | 9 |
| $s_7$ | 372 | 7882 | 65748 | 53 |

<table>
<thead>
<tr>
<th>Idx</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$\cdots$</td>
</tr>
</tbody>
</table>

→ One bit vector per RID in *Cities*.

→ Length of bit vector $\equiv$ length of fact table (*Sales*).
Similarly: Type 3 index $State \rightarrow \{Sales.\text{rid}\}$

→ One bit vector per $State$ value in $Cities$.
→ Length of bit vector $\equiv$ length of fact table ($Sales$).
Space Consumption

For a column with \( n \) distinct values, \( n \) bit vectors are required to build a bitmap index.

For a table with \( N \) rows, this leads to a space consumption of

\[
N \cdot n \text{ bits}
\]

for the full bitmap index.

This suggests the use of bitmap indexes for \textbf{low-cardinality attributes}.

\( \rightarrow \) e.g., product categories, sales regions, etc.

For comparison: A 4-byte integer column needs \( N \cdot 32 \) bits.

\( \rightarrow \) For \( n \lesssim 32 \), a bitmap index is more compact.
Reducing Space Consumption

For larger $n$, space consumption can be reduced by

1. alternative bit vector representations or
2. compression.

Both may be a space/performance trade-off.

Decomposed Bitmap Indexes:

- Express all attribute values $v$ as a **linear combination**

$$v = v_0 + c_1 v_1 + c_1 c_2 v_2 + \cdots + c_1 \cdots c_k v_k \ (c_1, \ldots, c_k \text{ constants}).$$

- Create a **separate bitmap index** for each variable $v_i$. 
Example: Index column with domain [0, \ldots, 999].

- Regular bitmap index would require 1000 bit vectors.
- Decomposition (c_1 = c_2 = 10):
  \[ v = 1v_1 + 10v_2 + 100v_3 \].

- Need to create 3 bitmap indexes now, each for 10 different values → 30 bit vectors now instead of 1000.
- However, need to read 3 bit vectors now (and and them) to answer point query.
**Decomposed Bitmap Indexes**

- **Query:**
  \[a = 576 = 5 \times 100 + 7 \times 10 + 6 \times 1\]

- **RIDs:**
  \[B_{v3,5} \wedge B_{v2,7} \wedge B_{v1,6} = [0010...0]\]

\[\Rightarrow \text{RID 3, ...}\]
Space/Performance Trade-Offs

Setting $c_i$ parameters allows to trade space and performance:

\[
\text{Time (Expected Number of Bitmap Scans)} \quad \text{Space (Number of Bitmaps)}
\]

Orthogonal to bitmap decomposition: Use compression.

- *E.g.*, straightforward equality encoding for a column with cardinality $n$: $1/n$ of all entries will be 0.

Which compression algorithm would you choose?
Compression

**Problem:** Complexity of (de)compression $\leftrightarrow$ simplicity of bit operations.

- Extraction and manipulation of **individual bits** during (de)compression can be expensive.
- Likely, this would off-set any efficiency gained from logical operations on **large CPU words**.

**Thus:**

- Use (rather simple) **run-length encoding**, 
- but respect **system word size** in compression scheme.

Word-Aligned Hybrid (WAH) Compression

Compress into a sequence of 32-bit words:

Bit \(\square\) tells whether this is a **fill word** or a **literal word**.

- Fill word \((\square = 1)\):
  - Bit \(\square\) tells whether to fill with 1s or 0s.
  - Remaining 30 \(\square\) bits indicate the number of fill bits.
    - This is the number of **31-bit blocks** with only 1s or 0s.
    - *e.g.*, \(\square = 3\): represents 93 1s/0s.

- Literal word \((\square = 0)\):
  - Copy 31 \(\square\) bits directly into the result.
WAH: Effectiveness of Compression

WAH is good to counter the space explosion for high-cardinality attributes.

- At most 2 words per ‘1’ bit in the data set
- At most $\approx 2 \cdot N$ words for a table with $N$ rows, even for large $n$ (assuming a bitmap that uses equality encoding).

![Image of graph showing the expected size of bitmap indices on random data and Markov data with various clustering factors.]

**Proposition 4.**
Let $N$ be the number of rows in a table, and let $c$ be the cardinality of the attribute to be indexed. Then the total size $s$ of all compressed bitmaps in an index is such that

1. It never takes more than $4N$ words,
2. If $c < N/10$, the maximum size of the compressed bitmap index of the attribute is about $2N$ words.
WAH: Effectiveness of Compression

If (almost) all values are distinct, additional bookkeeping may need some more space (\(\sim 4 \cdot 10^8\) bits for cardinality \(10^8\)).
Bitmap Indexes in Oracle 8

Index Size

Cardinality

Size (Mbytes)

0
5
10
15
20
25

0
5
10
15
20
25

2
4
5
10
25
100
1000
10000
40000
100000
250000
500000
1000000

Bitmap

B-tree

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2015/16
The most space-efficient bitmap representation depends on the **number of distinct values** (i.e., the **sparseness** of the bitmap).

- **low attribute cardinality** (dense bitmap)
  - can use **un-compressed bitmap**
    WAH compression won’t help much (but also won’t hurt much)

- **medium attribute cardinality**
  - use (WAH-)**compressed bitmap**

- **high attribute cardinality** (many distinct values; sparse bitmap)
  - Encode “bitmap” as **list of bit positions**

In addition, compressed bitmaps may be a good choice for data with **clustered content** (this is true for many real-world data).
Bitvectors encode a list of integer positions. But we need RIDs. What gives?
Conversely, bitmaps may be a good way to encode lists of rows.

→ Represent RID lists in B-tree leaves as (compressed) bit vectors.

In practice:
- Divide table into segments (≈ 32,000 tuples/segment).
- Separate bitmap for each segment.
- Per segment can decide on WAH ↔ RID list.

→ E.g., Oracle’s bitmap indexes are essentially that (though exact encoding is proprietary).

Benefits:
- May be able to skip over entire segments.
- Keep update cost reasonable.