Data Warehouse Physical Design: Part I

Robert Wrembel
Poznan University of Technology
Institute of Computing Science
Robert.Wrembel@cs.put.poznan.pl
www.cs.put.poznan.pl/rwrembel

Lecture outline

Basic index structures
- B-tree
- bitmap
- join
- bitmap join (Oracle)
- clustered (DB2)
- multidimensional cluster (DB2)
- indexing dimensions
### B-tree

- Foundation for other indexes (join, bitmap, bitmap join, clustering, MDC)

### Star schema and queries

```sql
select sum(SalesPrice), ProdName, Country, Year
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
and p.Category in ('electronics')
and t.Year in (2009, 2010)
group by ProdName, Country, Year;
```
Join index

Materialized join of 2 tables (typically fact and dimension(s))

<table>
<thead>
<tr>
<th>Products</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWID</td>
<td>price</td>
</tr>
<tr>
<td>BFF1</td>
<td>100</td>
</tr>
<tr>
<td>BFF2</td>
<td>230</td>
</tr>
<tr>
<td>BFF3</td>
<td>300</td>
</tr>
<tr>
<td>ROWID</td>
<td>salesPrice</td>
</tr>
<tr>
<td>0AA0</td>
<td>...</td>
</tr>
<tr>
<td>0AA1</td>
<td>...</td>
</tr>
<tr>
<td>0AA2</td>
<td>...</td>
</tr>
<tr>
<td>0AA3</td>
<td>...</td>
</tr>
<tr>
<td>0AA4</td>
<td>...</td>
</tr>
<tr>
<td>0AA5</td>
<td>...</td>
</tr>
</tbody>
</table>

Join index

In order to make searching the join index faster, the join index is physically ordered (clustered) by one of the attributes.

Alternatively, the access to the join index can be organized by means of a B-tree or a hash index.
DW queries

- DW queries typically are not selective
  - select dozens % of rows in a fact table
- B-tree indexes are efficient up to 10% selectivity of a query
- Other types of indexes are needed

Bitmap index - definitions

- Attribute cardinality \( \Rightarrow \) domain size
  - \( \text{card(make)} = 4, \text{card(color)} = 4 \)
- Bitmap \( \Rightarrow \) vector of bits
  - a bit corresponds to a row in a table

<table>
<thead>
<tr>
<th>Car/Sales</th>
<th>make</th>
<th>...</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subaru</td>
<td>red</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mercedes</td>
<td>green</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mercedes</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subaru</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BMW</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BMW</td>
<td>green</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BMW</td>
<td>red</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Audi</td>
<td>red</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Audi</td>
<td>black</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BMW</td>
<td>black</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subaru</td>
<td>red</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mercedes</td>
<td>green</td>
<td></td>
</tr>
</tbody>
</table>

B: red

1
0
0
0
0
1
1
0
0
1
0
**Bitmap index - definitions**

- **Bitmap index**
  - collection of bitmaps
  - one bitmap for one value from attribute domain

- **Organizing bitmaps**
  - 2-dimensional table
  - B-tree index
  - ...

<table>
<thead>
<tr>
<th>make</th>
<th>color</th>
<th>B: red</th>
<th>B: green</th>
<th>B: blue</th>
<th>B: black</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subaru</td>
<td>red</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mercedes</td>
<td>green</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mercedes</td>
<td>blue</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Subaru</td>
<td>blue</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BMW</td>
<td>blue</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BMW</td>
<td>green</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BMW</td>
<td>red</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Audi</td>
<td>red</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Audi</td>
<td>black</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BMW</td>
<td>black</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Subaru</td>
<td>red</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mercedes</td>
<td>green</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bitmap index in queries**

```sql
select count(*) from CarSales
where make in ('Audi', 'Mercedes', 'BMW')
and type = 'sport'
and color = 'red';
```

```
<table>
<thead>
<tr>
<th>B:Audi</th>
<th>OR</th>
<th>B:Mercedes</th>
<th>OR</th>
<th>B:BMW</th>
<th>AND</th>
<th>B:sport</th>
<th>AND</th>
<th>B:red</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Mapping bits to ROWIDs

**Easy solution:** fixed number of rows per DB block - rpb

- Block 1:
  - 1
  - 2
  - 3
  - 4
  - 5

- Block 2:
  - 6
  - 7
  - 8
  - 9
  - 10

- Block 3:
  - 11
  - 12
  - 13
  - 14
  - 15

\[ pgNo = \left\lceil \frac{bitNo}{rpb} \right\rceil \]

\[ slotNo = bitNo \mod rpb \]

**Real approach**

- estimate the average length \( L \) of a row in a table
- compute the number of slots in a DB block: \( \text{BS}ize/L \)
- allocate more slots than \( \text{BS}ize/L \)
- real row length differs from \( L \) \( \Rightarrow \) some slots are wasted \( \Rightarrow \) BI is larger than it could be
Mapping bits to ROWIDs

<table>
<thead>
<tr>
<th>B: red</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>red</td>
</tr>
<tr>
<td>0</td>
<td>green</td>
</tr>
<tr>
<td>0</td>
<td>blue</td>
</tr>
<tr>
<td>0</td>
<td>blue</td>
</tr>
<tr>
<td>0</td>
<td>blue</td>
</tr>
<tr>
<td>1</td>
<td>red</td>
</tr>
<tr>
<td>1</td>
<td>red</td>
</tr>
<tr>
<td>0</td>
<td>black</td>
</tr>
<tr>
<td>0</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>red</td>
</tr>
<tr>
<td>0</td>
<td>green</td>
</tr>
</tbody>
</table>

nb of bits allocated per data block

Mapping bits to ROWIDs: Oracle

```sql
create table TEST_BI1
(id number(7),
 no number(7),
 txt1 varchar2(20),
 txt2 varchar2(20),
 txt3 varchar2(20),
 txt4 varchar2(20),
 txt5 varchar2(20));

create table TEST_BI2
(id number(7) not null,
 no number(7) not null,
 txt1 varchar2(20) not null,
 txt2 varchar2(20) not null,
 txt3 varchar2(20) not null,
 txt4 varchar2(20) not null,
 txt5 varchar2(20) not null);
```

```sql
select o.name, o.obj#, t.spare1
from sys.obj$ o, sys.tab$ t
where o.obj#=t.obj# and o.name in ('TEST_BI1', 'TEST_BI2')
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>OBJ#</th>
<th>SPARE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_BI1</td>
<td>73450</td>
<td>736</td>
</tr>
<tr>
<td>TEST_BI2</td>
<td>73452</td>
<td>506</td>
</tr>
</tbody>
</table>

max number of rows per data block
Mapping bits to ROWIDs: Oracle

- Populating table TEST_BI1 with $10^6$ of rows of length 113B
- Computing column statistics

```sql
create bitmap index NO_BI_INDX on TEST_BI1(NO) pctfree 0;
```

```sql
select index_name, leaf_blocks
from dba_indexes
where index_name = 'NO_BI_INDEX';
```

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>LEAF_BLOCKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_BI_INDEX</td>
<td>350</td>
</tr>
</tbody>
</table>
```

Mapping bits to ROWIDs: Oracle

```sql
drop index NO_BI_INDEX;
```

```sql
alter table TEST_BI1 minimize records_per_block;
```

```sql
create bitmap index NO_BI_INDX on TEST_BI1(NO) pctfree 0;
```

```sql
select index_name, leaf_blocks
from dba_indexes
where index_name = 'NO_BI_INDEX';
```

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>LEAF.Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_BI_INDEX</td>
<td>150</td>
</tr>
</tbody>
</table>
```
BI characteristics (1)

✦ Reasonably small size for attributes of low cardinality

✦ Example
  - #rows = 1,000,000
  - card(A) = 4
  - ROWID = 10B (Oracle)
  - bitmap index on attribute A
    - 4 bitmaps: 4 x (1,000,000 / 8) = 4 x 125kB = 500kB
  - B+*-tree on attribute A
    - 1,000,000 x 10B = 10MB

BI characteristics (2)

✦ Efficient processing of bitmaps
  - logical operations AND, OR, NOT, COUNT
  - 64 bits processed in one CPU clock cycle

✦ Size
  - small index ⇒ small #I/O
  - processing in RAM

✦ Not applicable to LIKE
BI characteristics (3)

- **Large size** for attributes of large cardinality
- **Example**
  - #rows = 1 000 000
  - card(A) = 1024
  - ROWID = 10B (Oracle)
  - bitmap index on attribute A
    - 1024 bitmaps: $1024 \times (1,000,000 / 8) = 1024 \times 125\text{kB} = 128\text{MB}$
  - B*-tree on A
    - $1,000,000 \times 10B = 10\text{MB}$

BI characteristics (3)

- **Index maintenance**
  - inserting rows ⇒ increasing length of bitmaps
  - updating rows ⇒ updating 2 bitmaps
  - deleting rows ⇒
    - decreasing length of bitmaps
    - OR bitmap of deleted rows
  - locking contiguous segments of bitmaps ⇒ concurrency decreasing
Experiment (1)

Oracle11g
- data cache: 1.7GB, SGA: 3.4GB
- #rows: 250 000 000 (DB size: 10.8GB)

Experiment (2)
Experiment (3)

Cardinality of indexed attribute: 1024

Select count from ... where K1024 <= n

Experiment (4)

WHERE K1024 <= 128

Bitmap index

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>4</td>
<td>11315 (1)</td>
<td>00:02:16</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>BMP_5K1024</td>
<td>31M</td>
<td>119M</td>
<td>11315 (1)</td>
<td>00:02:16</td>
</tr>
</tbody>
</table>

B*-tree index

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>4</td>
<td>65441 (1)</td>
<td>00:13:06</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX RANGE SCAN</td>
<td>BMP_5K1024</td>
<td>31M</td>
<td>119M</td>
<td>65441 (1)</td>
<td>00:13:06</td>
</tr>
</tbody>
</table>
**Experiment (5)**

```
select sum(...) from ... where K1024<=n
```

![Graph](image1)

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>10</td>
<td>149K (1)</td>
<td>00:29:51</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>BMTEST</td>
<td>977K</td>
<td>9543K</td>
<td>149K (1)</td>
<td>00:29:51</td>
</tr>
<tr>
<td>3</td>
<td>BITMAP CONVERSION TO ROWIDS</td>
<td></td>
<td>1</td>
<td>10</td>
<td>149K (1)</td>
<td>00:29:51</td>
</tr>
<tr>
<td>4</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>BMP_5K1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Experiment (7)**

```
WHERE K1024 <= 4
```

**bitmap index**

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>10</td>
<td>387K (2)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td></td>
<td></td>
<td>149K (1)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>BMTEST</td>
<td>977K</td>
<td>9543K</td>
<td>149K (1)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>4</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>BMP_5K1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**B*-tree index**

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>10</td>
<td>387K (2)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td></td>
<td></td>
<td>149K (1)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>BMTEST</td>
<td>977K</td>
<td>9543K</td>
<td>149K (1)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>4</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>BMP_5K1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Experiment (8)

**WHERE K1024 <= 32**

#### Bitmap Index

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>10</td>
<td>344K (1)</td>
<td>01:08:57</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>BMTEST</td>
<td>7819K</td>
<td>74M</td>
<td>344K (1)</td>
<td>01:08:57</td>
</tr>
<tr>
<td></td>
<td>BITMAP CONVERSION TO ROWIDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 4</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>BMP_5K1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**WHERE K1024 <= 64**

#### Bitmap Index

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>10</td>
<td>387K (2)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>BMTEST</td>
<td>15M</td>
<td>149M</td>
<td>387K (2)</td>
<td>01:17:26</td>
</tr>
</tbody>
</table>

---

**B*-tree Index

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>10</td>
<td>387K (2)</td>
<td>01:17:26</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>BMTEST</td>
<td>15M</td>
<td>149M</td>
<td>387K (2)</td>
<td>01:17:26</td>
</tr>
</tbody>
</table>

---

R.Wrembel - Poznan University of Technology
Decreasing size of BI

- Range-based bitmap index
- Encoding
- Compression

Range-based BI (1)

- Domain of indexed attribute is divided into ranges
  - e.g., temperature: <0, 20), <20, 40), <40, 60), <60, 80), <80, 100)

<table>
<thead>
<tr>
<th>Indexed attribute</th>
<th>B4 (100, 80&gt;]</th>
<th>B3 (80, 60&gt;]</th>
<th>B2 (60, 40&gt;]</th>
<th>B1 (40, 20&gt;]</th>
<th>B0 (20, 0&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tempC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>39.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>51.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>98.8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>71</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>68.8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50.4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Query: count records for which 10<=temp<45
**Range-based BI (2)**

- Bitmaps can represent also sets of values
  - e.g., B1: \{yellow, orange, red\}, B2: \{light blue, blue, navy blue\}

- Characteristics
  - the number of bitmaps depends less on the attribute cardinality \(\propto\) depends on the range/set width
  - border bitmaps may point to rows that do not fulfill selection criteria \(\Rightarrow\) additional row filtering after fetching

**Encoding (1)**

- Replacing the value of an indexed attribute by another value whose bitmap representation is more compact

- Example
  - \(\text{card(productName)}: 50000 \Rightarrow\) typical number of products in a supermarket
  - standard bitmap index \(\Rightarrow 50000\) bitmaps
  - 50000 distinct values can be encoded on 16 bits
    - \([\log_{2}50000] = 16\)
  - a mapping data structure is required for mapping the encoded values into their real values
Encoding (2)

- query: select * from T where product = 'pecorino d'Abruzzo'
- apply mask: \(~B_{15} \ldots ~B_{4} ~B_{3} ~B_{2} ~B_{1} ~B_{0}\)

<table>
<thead>
<tr>
<th>indexed attribute</th>
<th>(B_{15})</th>
<th>(B_{14})</th>
<th>(B_{13})</th>
<th>(B_{12})</th>
<th>(B_{11})</th>
<th>(B_{10})</th>
</tr>
</thead>
<tbody>
<tr>
<td>queso Manchego</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>queso de Burgos</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>queso Cerrato</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>queso Serrat</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>tipi</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>queso de Urbasa</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pecorino barcellone</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pecorino d'Abruzzo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pecorino del Belice</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>pecorino di Farindola</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>pecorino lucano</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>pecorino rosso</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pecorino sardo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pecorino sense</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Compression (1)

- Byte-aligned Bitmap Compression (BBC)
- Word-Aligned Hybrid (WAH)
- Run Length Huffman
- Based on the run-length encoding
  - homogeneous vectors of bits are replaced with a bit value (0 or 1) and the vector length
  - \(0000000 1111111111 000 \Rightarrow 07 \ 110 \ 03\)
- A bitmap is divided into words
  - BBC uses 8-bit words
  - WAH uses 31-bit words
  - RLH uses \(n\)-bit words (\(n\) - parameter)
**Compression (2)**

- WAH-compressed bitmaps are larger than BBC-compressed ones
- Operations on WAH-compressed bitmaps are faster than on BBC-compressed ones
  - Wu, K. and Otoo, E. J. and Shoshani, A.: Compressing Bitmap Indexes for Faster Search Operations, SSBDM, 2002
- Types of words in BBC and WAH
  - fill word \( \Rightarrow \) represents a compressed segment of a bitmap (composed either of all 0s or all 1s)
  - tail word \( \Rightarrow \) represents non-compressable segment of a bitmap (composed of interchanged 0 and 1 bits)

**WAH (1)**

- Example: 32-bit processor, bitmap composed of 5456 bits

```
100000...0000011100001110000000000000000000...0000000000001111111111111111111111
```

- Step 1: divide the bitmap into groups including 31 bits each
Step 2: merge adjacent homogeneous groups (having the same values of all bits, i.e., groups 2-175)

<table>
<thead>
<tr>
<th>31 bits</th>
<th>5394 bits having value “0”</th>
<th>31 bits</th>
</tr>
</thead>
</table>

Heterogeneous group

<table>
<thead>
<tr>
<th>31 bits</th>
<th>174 * 31 bits</th>
<th>31 bits</th>
</tr>
</thead>
</table>

Group 1

Group 2-175

Group 176

Step 3: group encoding
- run: fill + tail
- run 1: tail
- run 2: fill + tail
- run 3: tail

The number of 31-bit groups

<table>
<thead>
<tr>
<th>31 bits of the first group</th>
<th>fill length 174 * 31 bits</th>
<th>31 bits of the last group</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit=0: tail word</td>
<td>bit=0: fill value</td>
<td>bit=0: tail word</td>
</tr>
<tr>
<td>run 1</td>
<td>run 2</td>
<td></td>
</tr>
</tbody>
</table>
WAH (4)

+ Unsorted data
+ For low cardinality attributes bitmaps are dense
  - many homogeneous 31-bit words filled with 1
+ For high cardinality attributes bitmaps are sparse
  - many homogeneous 31-bit words filled with 0
+ For medium cardinality attributes
  - the number of homogeneous 31-bit words is lower

RLH

+ RLH - the Run-Length Huffman Compression (M. Stahno and R. Wrembel. Information Systems, 34(4-5), 2009)
+ Based on
  - the Huffman encoding
  - a modified run-length encoding
Huffman Encoding (1)

Concept
- original symbols from a compressed file are replaced with bit strings
- the more frequently a given symbol appears in the compressed file the shorter bit string for representing the symbol
- encoded symbols and their corresponding bit strings are represented as a Huffman tree
- the Huffman tree is used for both compressing and decompressing

Huffman Encoding (2)

Example: encoding text "this_is_a_test"
Step 1: frequencies of the symbols in the encoded string

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>3</td>
</tr>
<tr>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
</tbody>
</table>
Huffman Encoding (3)

Steps:
1. **Step 2: building Huffman tree**
   - Start with nodes of the lowest frequency

2. **Step 3: getting the Huffman codes of the symbols from the tree**

Huffman Encoding (4)

Steps:
1. **Step 4: replacing original symbols with their Huffman codes**
   - Original text: 14B
   - Compressed text: 37b → 5B
**RLH (1)**

- **Modified run-length encoding**
  - encodes distances between bits of value 1

![Bitmaps](image)

**Modified run-length encoding**

- female: 100303001000
- male: 030020033

---

**RLH (2)**

- **Huffman encoding**
  - step 1: computing frequencies of symbols (distances) in encoded bitmaps

<table>
<thead>
<tr>
<th>distance</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Huffman encoding**

- female: 100303001000
- male: 030020033

---
RLH (3)

Huffman encoding

- step 2: building a Huffman tree

- an encoded symbol is represented by a path from the root to a leaf

RLH (4)

Huffman encoding

- step 3: replacing distances with their Huffman codes
RLH-N

Dividing a bitmap into $N$-bit sections
- constructing one Huffman tree based on frequencies of distances from all $N$-bit sections

Including in the HT all possible distances that may appear in a $N$-bit section
- non-existing distances have assigned the frequency of 1

Experimental Evaluation

Comparing RLH, WAH, and uncompressed bitmaps (UBI) with respect to
- bitmap sizes
- query response times

Implementation in Java
- data and bitmap indexes stored on disk in OS files

Experiments run on
- PC, AMD Athlon XP 2500+; 768 MB RAM; Windows XP

Data
- 100 000 000 indexed rows
- indexed attribute of type integer
  - cardinality from 2 to 20 000
  - randomly distributed values
WAH and RLH: index sizes

- RLH, RLH-N, WAH, and UBI with respect to the size of a bitmap index (N = \{256, 512, 1024, 2048\}) for RLH-N

![Graph showing index size vs. cardinality of indexed attribute]

WAH and RLH: response times

- Query: select ... from ... where ind_attribute in (v1, v2, ..., v100)
- Randomly ordered rows wrt. the value of the indexed attribute

![Graph showing response time vs. cardinality of indexed attribute]
Updating RLH Bitmaps

✈ Costly process
  ▪ decompressing the whole bitmap
  ▪ modifying the bitmap
  ▪ compressing the bitmap
  ▪ changes frequencies of distances between 1 bits
  ▪ creates new distances between 1 bits
✈ In a DW environment index structures
  ▪ are dropped before loading a DW
  ▪ are recreated after loading is finished

BIs in Oracle

✈ Defined explicitly by DBA
✈ Compressed automatically
✈ Bitmap join index available
✈ Used for optimizing star queries
**Bitmap Join Index (1)**

```
create bitmap index Sales_JBI
on Sales(Products.ProdName)
from Sales s, Products p
where s.ProductID=p.ProductID;
```
BJI (3)

Star query optimization with the support of BJI

```sql
SELECT SUM(sa.SalesPrice), p.ProdName, sh.ShopID
FROM Sales sa, Shops sh, Products p
WHERE sh.country IN ('Poland', 'Slovakia')
AND p.Category = 'cheese'
AND sa.ShopID = sh.ShopID
AND sa.ProductID = p.ProductID
GROUP BY p.ProdName, sh.ShopID;
```

BJIs defined on attributes

- Shops.Country
- Products.Category

BJI (4)
The Oracle case

select sum(SalesPrice) 
from Sales, Products, Customers, Time 
where Sales.ProductID=Products.ProductID 
and Sales.CustomerID=Customers.CustomerID 
and Sales.TimeKey=Time.TimeKey 
and ProdName in ('ThinkPad Edge', 'Sony Vaio', 'Dell Vostro') 
and Town='London' 
and Year=2009;

create bitmap index BI_Pr_Sales 
on Sales(Products.ProdName) 
from Sales s, Products p 
where s.ProductID=p.ProductID;

create bitmap index BI_Cu_Sales 
on Sales(Customers.Town) 
from Sales s, Customers c 
where s.CustomerID=c.CustomerID;

create bitmap index BI_Ti_Sales 
on Sales(Time.Year) 
from Sales s, Time t 
where s.TimeKey=t.TimeKey;
create bitmap index BI_Pr_Cu_Ti_Sales
on Sales(Products.ProdName, Customers.Town, Time.Year)
from Sales, Products, Customers, Time
where Sales.ProductID=Products.ProductID
and Sales.CustomerID=Customers.CustomerID
and Sales.TimeKey=Time.TimeKey;

**BIs in DB2 (1)**

- Created and managed implicitly by the system
- Applied to join optimization
  - Every dim table is independently semi-joined with a fact table
  - The semi-joins use B-trees on foreign keys
  - ROWIDs of every semi-join result are transformed into a separate bitmap
  - Bitmaps $B_i$ are constructed by means of a hash function on ROWID
    - the hash value points to a bit in $B_i$
  - Final bitmap is computed by AND-ing $B_i$'s
**BIs in DB2 (2)**

Diagram showing the relationship between Shops, Sales, Products, and the condition `Shops.Country IN ('Poland', 'Slovakia')` and `Products.Category='cheese'`.

**BIs in SQL Server (1)**

- Created and managed implicitly by the system
- Applied to join optimization
  - join of a dim table with a fact table by means of hash join
  - table with a PK (dim table) ⇒ external table
  - table with a FK (fact table) ⇒ internal table
**BIs in SQL Server**

- Hashing PK values into a bitmap
  - HashFunction(PK) → bit no of value 1
- Hashing FK values into a bitmap
  - HashFunction(PK) → bit no of value 1

The rows from both tables that hash to the same bit → join result

**BIs in SybaseIQ**

- Defined explicitly by DBA
- Low Fast ⇒ for attributes of low cardinalities
  - max cardinality: 10 000
  - the highest performance for cardinality up to 1000
- High Non Group ⇒ for attributes of high cardinalities
  - for aggregate queries with range predicates
**BI in SAS**

- **SAS Scalable Performance Data (SPD) Server**
  - hybrid index
  - table is divided into segments (e.g., 8192 rows)
  - every segment is indexed independently by
    - bitmap index or
    - B-tree
  - index type selected automatically by the system taking into account the distribution of values in a segment

![Diagram showing segmentation and indexing](image)

**DB2: Clustering index** (1)

- Clustering index determines how rows are physically ordered (clustered) on disk
- After defining the index, rows are inserted in the order determined by the index
- Only one index can be a clustering index (one physical order of rows on disk)
- By default the first index created is the clustering one (unless you explicitly define another index to be the clustering index)
**DB2: Clustering index (2)**

CREATE INDEX cityID_Indx ON Auctions(cityID) CLUSTER

<table>
<thead>
<tr>
<th>cityID</th>
<th>dateID</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>POZ</td>
<td>03-07-2013</td>
<td>10</td>
</tr>
<tr>
<td>WRO</td>
<td>05-07-2013</td>
<td>13</td>
</tr>
<tr>
<td>WAWS</td>
<td>03-07-2013</td>
<td>7</td>
</tr>
<tr>
<td>WRO</td>
<td>09-07-2013</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cityID</th>
<th>dateID</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDA</td>
<td>08-07-2013</td>
<td>17</td>
</tr>
<tr>
<td>POZ</td>
<td>05-07-2013</td>
<td>9</td>
</tr>
<tr>
<td>POZ</td>
<td>08-07-2013</td>
<td>2</td>
</tr>
<tr>
<td>POZ</td>
<td>09-07-2013</td>
<td>13</td>
</tr>
</tbody>
</table>

**DB2: Clustering index (3)**

- Eliminates sorting
- Operations that benefit from clustering indexes include:
  - grouping
  - ordering
  - comparisons other than equal
  - distinct
MultiDimensional Cluster - MDC
- groups data based on values of multiple dimension attributes
- a physical region (block) is associated with each unique combination of dimension attribute values
- a block stores records with the same values of dimension attributes

Block Map: a structure that stores information about block states (in use, free, loaded, ...)

CREATE TABLE Auctions
(...) cityID VARCHAR(4), dateID DATE, quantity INT, ...)
ORGANIZE BY (cityID, dateID);

MDC

<table>
<thead>
<tr>
<th>cityID</th>
<th>dateID</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDJ</td>
<td>08-07-2011</td>
<td>17</td>
</tr>
<tr>
<td>GDJ</td>
<td>08-07-2011</td>
<td>7</td>
</tr>
<tr>
<td>GDJ</td>
<td>10-07-2011</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cityID</th>
<th>dateID</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDJ</td>
<td>10-07-2011</td>
<td>11</td>
</tr>
<tr>
<td>POZ</td>
<td>05-07-2011</td>
<td>4</td>
</tr>
<tr>
<td>WRO</td>
<td>05-07-2011</td>
<td>10</td>
</tr>
<tr>
<td>WAW</td>
<td>07-07-2013</td>
<td>2</td>
</tr>
<tr>
<td>GDA</td>
<td>06-07-2013</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cityID</th>
<th>dateID</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>POZ</td>
<td>08-07-2011</td>
<td>2</td>
</tr>
<tr>
<td>GDJ</td>
<td>08-07-2011</td>
<td>7</td>
</tr>
<tr>
<td>GDJ</td>
<td>10-07-2011</td>
<td>17</td>
</tr>
<tr>
<td>WRO</td>
<td>10-07-2013</td>
<td>22</td>
</tr>
<tr>
<td>GDA</td>
<td>10-07-2013</td>
<td>11</td>
</tr>
<tr>
<td>WRO</td>
<td>10-07-2013</td>
<td>4</td>
</tr>
</tbody>
</table>
DB2: MDC (3)

- Block index: B-tree based, points to blocks
  - created automatically for each of the dimensions in MDC

DB2: MDC (4)

- Composite block index: includes all dimension key columns
  - used for insert, update, delete
MDC in queries

```sql
SELECT SUM(quantity), cityID, dateID
FROM Auctions
WHERE cityID = 'GDA' AND dateID = '10-07-2013'
group by cityID, dateID;
```

```
\{block IDs with cityID='GDA'} \cup \{block IDs with dateID='10-07-2013'}
```
**MDC**

- Candidates as dimensions in MDC
  - attributes used in predicates: range, =, IN
  - dimension foreign keys in fact table
  - attributes used in GROUP BY
  - attributes used in ORDER BY

- Summary
  - Data ordered on disk ⇒ less I/O
  - Block index points to a data block ⇒ inserting, updating, deleting may not affect the index structure

---

**MDC case study**

- Source: IBM presentation
- Mobile network operator in USA
- Characteristic
  - 10 billion transactions daily
  - 32 TB raw data
  - thousands concurrent users
  - up to 37000 queries daily
  - DW loading: over 1 billion rows daily (max 1.6 billion)
  - DB2 DWE, 16 x 8 CPU P5 pSeries

- MDC
  - deletion faster by 80% of time
  - I/O lower by 43%
Clustered Based Table (Netezza)

- Clustered Base Table (CBT) ⇒ data are organized by 1 to 4 attributes (organizing keys)
- Data stored in extents (zones)
  - an extent is the smallest unit of disk allocation = 3MB
- Organizing keys are used to group records within the table (store them in one or more nearby extents)
- Netezza creates zone maps for the organizing keys
- Materialized views cannot be build on CBTs

CREATE TABLE tab-name
(...) [ORGANIZE ON (org-key1, ...)]

From IBM original teaching material
CREATE BITMAP INDEX bmi_a_years_countries
ON auctions(yr.year, cr.countryName)
FROM
Auctions a,
Days d, Months m, Years yr,
Cities ci, Regions r, Countries cr
WHERE
a.cityid=ci.cityid AND
r.regionid=ci.regionid AND
r.countryid=cr.countryid AND
a.dateid=d.dateid AND
d.monthid=m.monthid AND
m.yearid=yr.yearid

exec. plan of Q3

- Indexes do not reflect a hierarchical structure of dimensions
- Time dimension is used in most of star queries → joins
  - HOBİ → Hierarchically Organized Bitmap Index
  - TI → Time Index
  - PA → Partial Aggregates
HOBI

- HOBI is composed of bitmap indexes created for every level of a dimension hierarchy
- BI are also organized in a hierarchy
- The BI hierarchy reflects the dimension hierarchy, such that a bitmap index at an upper level aggregates bitmap indexes from a lower level

Time Index

- Assumption: data ordered on disk by time
- TI encodes time in other dimensions
- Shared by all dimensions
- No joins with Time required
Partial Aggregates

PA are computed for:

- selected measures
- selected aggregation functions
- a given dimension
- a given dimension level
- a given dimension level instances
- a given time interval

Time-HOBI

- Eliminates joins of a fact table with the Time dimension ⇒ Time Index
- Eliminates joins of a fact table with other dimensions ⇒ HOBI
- Eliminates or reduces the costs of computing aggregates of measures at various levels of a dimension hierarchy ⇒ Partial Aggregates
**Experiments (1)**

- **Auctions:** 500 000 000, 30GB
- **Competitor:** Oracle indexes
  - concatenated BI on year and countryName that joined tables Years, Months, Days, Countries, Regions, Cities, and Auctions
  - BI on year
  - BI on countryName

**Experiments (2)**

R.Wrembel - Poznan University of Technology
Experiments (3)

Time-HOBI vs. MV

- Time-HOBI takes advantage of materialized partial aggregates ⇒ functionality of MV
- Unlike MV, Time-HOBI may be used to optimize queries that either do not compute aggregates or compute aggregates that have not been materialized in the index ⇒ more flexible structure
References

- Various types of indexes
  - J. Dyke: Bitmap Index Internals. juliandyke.com
  - N. Koudas. Space efficient bitmap indexing. CIKM, 2000
  - P. O’Neil and G. Graefe. Multi-table joins through bitmapped join indices. SIGMOD Record, 1995
  - P. O’Neil and D. Quass. Improved query performance with variant indexes. SIGMOD, 1997

- Indexes in DB2
  - S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, M. Huras: Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. SIGMOD, 2003
  - P. Bates, P. Zikopoulos: Multidimensional Clustering (MDC) Tables in DB2 LUW. Talk, DB2 Night Show, Jan 2011
  - IBM Netezza System Administrator’s Guide. IBM Netezza 7.0 and Later, Oct 2012
  - Clustering indexes. DB2 technical doc
References

- **Binning**
  - Koudas N.: Space Efficient Bitmap Indexing. CIKM, 2000
  - Rotem D., Stockinger K., Wu K.: Optimizing Candidate Check Costs for Bitmap Indices. CIKM, 2005

- **Encoding**
  - Wu M., Buchmann A.P.: Encoded Bitmap Indexing for Data Warehouses. ICDE, 1998
  - Chan C.Y., Ioannidis Y.E.: An Efficient Bitmap Encoding Scheme for Selection Queries. SIGMOD, 1999

- **Compressing**
  - BBC

References

- **Compressing**
  - WAH
    - Stockinger K., Wu K., Shoshani A.: Strategies for Processing ad hoc Queries on Large Data Sets. DOLAP, 2002
  - PL-WAH
    - F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. EDBT, 2010
  - Approximate Compression with Bloom Filters
    - Apaydin T., Canahuate G., Ferhatosmanoglu H., Tosun, A. S.: Approximate encoding for direct access and query processing over compressed bitmaps. VLDB, 2006
References

**Compressing**

- **Reordering**
  - Johnson D., Krishnan S., Chhugani J., Kumar S., Venkatasubramanian S.: Compressing Large Boolean Matrices Using Reordering Techniques. VLDB, 2004
  - Pinar A., Tao T., Ferhatosmanoglu H.: Compressing Bitmap Indices by Data Reorganization. ICDE, 2005

- **WAH vs. BBC**
  - Stockinger K., Wu K., Shoshani A.: Strategies for processing ad hoc queries on large data warehouses. DOLAP, 2002
  - Wu K., Otoo E.J., Shoshani A.: Compressing bitmap indexes for faster search operations. SSDBM, 2002

- **RLH**
  - M. Stahno and R. Wrembel. RLH: Bitmap compression technique based on runlength and Huffman encoding. Information Systems, 34(4-5), 2009
Data Warehouse
Physical Design:
Part II

Robert Wrembel
Poznan University of Technology
Institute of Computing Science
Robert.Wrembel@cs.put.poznan.pl
www.cs.put.poznan.pl/rwrembel

Lecture outline

✦ Row storage vs. Column storage
✦ Data compression
✦ Materialization
  ▪ Small summary data
  ▪ Materialized views and query rewriting
✦ Partitioning
✦ MOLAP
Row storage (standard)

Rows identified by ROWIDs

<table>
<thead>
<tr>
<th>Company</th>
<th>Date</th>
<th>Month</th>
<th>Day</th>
<th>Open</th>
<th>Min</th>
<th>Max</th>
<th>Close</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2VWKB</td>
<td>Mar</td>
<td>31</td>
<td>149.00</td>
<td>147.00</td>
<td>149.00</td>
<td>148.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>B2VWKB</td>
<td>Mar</td>
<td>30</td>
<td>149.00</td>
<td>147.00</td>
<td>149.00</td>
<td>148.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>B2VWKB</td>
<td>Mar</td>
<td>29</td>
<td>149.00</td>
<td>147.00</td>
<td>149.00</td>
<td>148.00</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

Row storage (2)

Database block

<table>
<thead>
<tr>
<th>Company</th>
<th>Date</th>
<th>Month</th>
<th>Day</th>
<th>Open</th>
<th>Min</th>
<th>Max</th>
<th>Close</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2VWKB</td>
<td>Mar</td>
<td>5</td>
<td>108.00</td>
<td>107.00</td>
<td>108.00</td>
<td>107.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>B2VWKB</td>
<td>Mar</td>
<td>3</td>
<td>108.00</td>
<td>107.00</td>
<td>108.00</td>
<td>107.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>B2VWKB</td>
<td>Mar</td>
<td>1</td>
<td>108.00</td>
<td>107.00</td>
<td>108.00</td>
<td>107.00</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>
Column storage (1)

- relational data model
- SQL interface
- every column stored and accessed separately

≠

- key:value data model
- no SQL interface
- columns are clustered ⇒ column family

Model 204 (projection index)
Sybase IQ (Sybase, Inc.)
SADAS (Advanced Systems)
C-Store/Vertica
MonetDB
Infobright
...

Column storage (2)

Rows identified by slot numbers (in a block)

Data blocks

140,00 149,00 140,00 150,30 147,60 148,00 148,00 147,00 145,60
145,60 150,00 150,00 151,70 148,00 150,00 150,00 150,00 150,40
172,40 173,00 159,00 159,00 170,30 166,00
147,60 147,60 146,00 145,60 142,00 147,60 145,60 147,60
147,60 147,60 146,00 145,60 146,00 146,00 146,00 145,60 145,60
105,00 109,00 110,00 109,00 105,00 105,00 105,00 105,00 105,00
Column storage (3)

- Database block
  - no free space
  - better space utilization

Query processing (1)

```
select Open
from StockQuotes
where CompanyID='BZ WBK'
and Year=2006
and Month='Mar'
and Day=8;
```

list of positions (slot numbers) of values fulfilling the predicate represented as:
- array or
- bit vector or
- set of position ranges
R.Wrembel - Poznan University of Technology
**Query processing (4)**

<table>
<thead>
<tr>
<th>Shops</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Poland</td>
</tr>
<tr>
<td>20</td>
<td>Russia</td>
</tr>
<tr>
<td>30</td>
<td>Slovakia</td>
</tr>
<tr>
<td>40</td>
<td>Slovenia</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Products</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>cheese</td>
</tr>
<tr>
<td>500</td>
<td>cheese</td>
</tr>
<tr>
<td>200</td>
<td>cheese</td>
</tr>
<tr>
<td>700</td>
<td>wine</td>
</tr>
</tbody>
</table>

**Hashing**

Poland or Slovakia

![Hashing Sales rows](image)

**Matched Sales rows are represented by bitmaps**

<table>
<thead>
<tr>
<th>ShopID in (10, 30)</th>
<th>ProductID in (102, 500, 200)</th>
<th>SalesID in (S2, S3, S4, S5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Query processing (5)**

**Hashing Sales rows**

- Hashing Sales row with ShopID: 10, 30
- Hashing ProductID: 100, 500, 200

**Matching Sales rows are represented by bitmaps**

<table>
<thead>
<tr>
<th>ShopID in (10, 30)</th>
<th>ProductID in (102, 500, 200)</th>
<th>SalesID in (S2, S3, S4, S5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Query processing** (6)

- Get ProductIDs using the final bitmap
- Join with Products to get 'prodName' using the ProductIDs

- Get ShopIDs using the final bitmap
- Join with Shops to get 'Country' using the ShopIDs

---

**Column storage in DW architecture**

![Diagram showing ETL/ELT, Data Warehouse, and models like Model 204, Sybase IQ, SADAS, C-Store, Vertica, MonetDB, Infobright, etc.](image)
**CS - compression** (1)

Run-length encoding

delta encoding

**CS - compression** (2)

→ Discrete domain encoding

<table>
<thead>
<tr>
<th>Shop</th>
<th>Date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma Old Brewery</td>
<td>01.2006</td>
<td>56 000</td>
</tr>
<tr>
<td>Alma Cie Park</td>
<td>01.2006</td>
<td>13 000</td>
</tr>
<tr>
<td>Alma Focus Park</td>
<td>01.2006</td>
<td>24 000</td>
</tr>
<tr>
<td>Alma Old Brewery</td>
<td>02.2006</td>
<td>52 000</td>
</tr>
<tr>
<td>Alma Cie Park</td>
<td>02.2006</td>
<td>19 000</td>
</tr>
<tr>
<td>Alma Focus Park</td>
<td>02.2006</td>
<td>21 100</td>
</tr>
<tr>
<td>Alma Old Brewery</td>
<td>03.2006</td>
<td>43 200</td>
</tr>
<tr>
<td>Alma Cie Park</td>
<td>03.2006</td>
<td>25 700</td>
</tr>
<tr>
<td>Alma Focus Park</td>
<td>03.2006</td>
<td>14 700</td>
</tr>
<tr>
<td>Alma Old Brewery</td>
<td>04.2006</td>
<td>32 400</td>
</tr>
<tr>
<td>Alma Cie Park</td>
<td>04.2006</td>
<td>19 500</td>
</tr>
<tr>
<td>Alma Focus Park</td>
<td>04.2006</td>
<td>15 000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shop</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma Old Brewery</td>
<td>1</td>
</tr>
<tr>
<td>Alma Cie Park</td>
<td>2</td>
</tr>
<tr>
<td>Alma Focus Park</td>
<td>3</td>
</tr>
</tbody>
</table>
RS - compression (1)

Oracle

<table>
<thead>
<tr>
<th>Eternity Calvin Klein</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polo Ralph Lauren</td>
<td>230</td>
</tr>
<tr>
<td>Polo Ralph Lauren</td>
<td>90</td>
</tr>
<tr>
<td>Polo Ralph Lauren</td>
<td>110</td>
</tr>
<tr>
<td>Eternity Calvin Klein</td>
<td>210</td>
</tr>
<tr>
<td>Eternity Calvin Klein</td>
<td>210</td>
</tr>
<tr>
<td>Eternity Calvin Klein</td>
<td>210</td>
</tr>
</tbody>
</table>

Dictionary compression (DB2)

Like discrete domain encoding
- dictionary stored in
  - a dedicated table
  - data block header

Performance comparison: CS-RS (1)

Source: Sybase talk (KKNTPD, 2005, Poland)

Sybase IQ
0.25 - 0.9 TB

Aggregates: 0.4-1TB
Indexes: 0.05-0.37TB
Source data: 0.2-0.5TB

Source data
1TB

Sybase IQ
0.25 - 0.9 TB

Aggregates: 0.4-1TB
Indexes: 0.05-0.37TB
Source data: 0.2-0.5TB

Index 0.5-3TB

Source data 0.9-1.1TB

Traditional RS system
2.4-6 TB

Aggregates: 1-2 TB
Performance comparison: CS-RS (2)

Source: presentation by Advanced Systems

Performance comparison: CS-RS (3)

Source: D.J. Abadi, S.R. Madden, N. Hachem: Column-stores vs. row-stores: how different are they really?. SIGMOD, 2008

Experimental setup:
- star schema benchmark ⇒ DW benchmark derived from TPC-H (pure star schema)
- 13 queries divided into 4 categories
Performance comparison: CS-RS (4)

<table>
<thead>
<tr>
<th></th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.1</th>
<th>2.2</th>
<th>2.3</th>
<th>3.1</th>
<th>3.2</th>
<th>3.3</th>
<th>3.4</th>
<th>4.1</th>
<th>4.2</th>
<th>4.3</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>2.7</td>
<td>2.0</td>
<td>1.5</td>
<td>43.8</td>
<td>44.1</td>
<td>46.0</td>
<td>43.0</td>
<td>42.8</td>
<td>31.2</td>
<td>6.5</td>
<td>44.4</td>
<td>14.1</td>
<td>12.2</td>
<td>25.7</td>
</tr>
<tr>
<td>RS</td>
<td>1.0</td>
<td>1.0</td>
<td>0.2</td>
<td>15.5</td>
<td>13.5</td>
<td>11.8</td>
<td>16.1</td>
<td>6.9</td>
<td>6.4</td>
<td>3.0</td>
<td>29.2</td>
<td>22.4</td>
<td>6.4</td>
<td>10.2</td>
</tr>
<tr>
<td>RS (MV)</td>
<td>0.4</td>
<td>0.1</td>
<td>0.1</td>
<td>3.7</td>
<td>4.2</td>
<td>3.9</td>
<td>11.0</td>
<td>4.4</td>
<td>7.6</td>
<td>0.6</td>
<td>8.2</td>
<td>3.7</td>
<td>2.6</td>
<td>4.0</td>
</tr>
<tr>
<td>CS (Row-MV)</td>
<td>0.8</td>
<td>9.1</td>
<td>8.4</td>
<td>33.5</td>
<td>23.5</td>
<td>22.3</td>
<td>18.5</td>
<td>21.5</td>
<td>17.6</td>
<td>17.4</td>
<td>48.6</td>
<td>38.4</td>
<td>32.1</td>
<td>25.9</td>
</tr>
</tbody>
</table>

Performance comparison: CS-RS (5)

- RS with various data structures
  - T: traditional RS
  - T(B): traditional + bitmap indexes
  - MV: optimal set of mat. views
  - VP: vertical partitioning (simulated - each column in its own table)
  - AI: B+-tree on each column

Average query times:
- T: 25.7
- T(B): 64.0
- MV: 10.2
- VP: 79.9
- AI: 221.2
Our experiments (1)

- Intel Core 2 Duo P8400 2.27 GHz, 4GB RAM, disc Hitachi Travelstar 5K250 HTS542525K9SA00
- Oracle11g and SybaseIQ 15.4
- DB size 3GB
- Cache
  - Sybase: 1024MB (main cache size)
  - Oracle: 1024MB (data cache)

Our experiments (2)

- Indexes
  - SybaseIQ
    - Fast Projection (default) ⇒ on all columns for projection optimization
    - High Group (default) ⇒ on UNIQUE, PRIMARY KEY, FOREIGN KEY
  - Oracle
    - PRIMARY KEY (default)
    - FOREIGN KEY
Our experiments (3)

Our experiments (4)

- Q1: GROUP BY (productName, regionName)
- Q2: GROUP BY (productName, regionName, monthNr)
- Q3: GROUP BY, 4 dimensions

```sql
SELECT sum(a.price), p.productName, r.regionName, m.monthNr, u.userId
FROM auctions a, products p, cities c, regions r,
    days d, months m, users u
WHERE a.productId = p.productId
  AND a.cityId = c.cityId
  AND c.regionId = r.regionId
  AND a.dateId = d.dateId
  AND d.monthId = m.monthId
  AND a.userId = u.userId
GROUP BY p.productName,
         r.regionName,
         m.monthNr,
         u.userId;
```
Our experiments (5)

- Q4: GROUP BY (productName, regionName, monthNr)
- variable selectivity {5, 10, 20, 30, 40, 50, 80%} on City and Date

![Bar chart showing elapsed time for different selectivities with Sybase and Oracle]

Our experiments (6)

- Q5: GROUP BY ROLLUP (productName, regionName)
- Q6: GROUP BY ROLLUP (productName, countryName, monthNr)

![Bar chart comparing elapsed time for Q5 and Q6 with Sybase and Oracle]
Q7: one table query

```
SELECT sum(a.price), a.productId, a.cityId, a.dateId
FROM Auctions a
GROUP BY productId, cityId, dateId;
```

Materialization - SMA

**SMA - Small Materialized Aggregates** (G. Moerkotte, VLDB, 1998)
- disk data are divided into buckets
- every bucket has associated SMA
SMA

- SMA
  - defined on an ordering attribute
  - used for filtering buckets
  - e.g. select ... from ... where TimeKey > '22-Mar-2006'

<table>
<thead>
<tr>
<th>SMA bucket1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeKey max: 31.03.2006</td>
</tr>
<tr>
<td>TimeKey min: 27.03.2006</td>
</tr>
<tr>
<td>count: 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SMA bucket2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeKey max: 24.03.2006</td>
</tr>
<tr>
<td>TimeKey min: 20.03.2006</td>
</tr>
<tr>
<td>count: 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SMA bucket3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeKey max: 17.03.2006</td>
</tr>
<tr>
<td>TimeKey min: 13.03.2006</td>
</tr>
<tr>
<td>count: 5</td>
</tr>
</tbody>
</table>

Zone Map - IBM Netezza (1)

- ZM - Zone Maps
  - similar to SMA
  - data stored in extents (zones)
    - an extent is the smallest unit of disk allocation = 3MB
  - created automatically, by default for columns of type integer, date, and timestamp
  - created automatically for columns used in the ORDER BY clause of a materialized view
  - for a given attribute store MIN and MAX value of the attribute in an extent
  - created for every extent
  - maintained automatically by the system
Zone Map - IBM Netezza (2)

ZM

<table>
<thead>
<tr>
<th>Index</th>
<th>Date</th>
<th>Open</th>
<th>Close</th>
<th>Index</th>
<th>Date</th>
<th>Open</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZWBK</td>
<td>21-03-2006</td>
<td>149.50</td>
<td>147.50</td>
<td>BZWBK</td>
<td>16-03-2006</td>
<td>148.50</td>
<td>147.50</td>
</tr>
<tr>
<td>BZWBK</td>
<td>20-03-2006</td>
<td>149.50</td>
<td>147.50</td>
<td>BZWBK</td>
<td>15-03-2006</td>
<td>151.50</td>
<td>148.50</td>
</tr>
<tr>
<td>BZWBK</td>
<td>17-03-2006</td>
<td>151.50</td>
<td>148.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BZWBK</td>
<td>16-03-2006</td>
<td>150.00</td>
<td>149.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Zone Filters (1)

Zone filters, Bit vector filters, Zone indexes (G. Graefe, DAWAK, 2009)

ZF - Zone Filter combines SMA i ZM
  - ZF maintained for each zone and attribute
  - ZF stores m consecutive MIN and MAX values
    - if m=1 then ZF equivalent to ZM
    - if m=1 then either MIN or MAX can be NULL (depending on NULLS LAST/FIRST) - not useful for filtering zones
    - if m=2 then in the presence of NULLs the second value of MIN or MAX is NOT NULL
Zone Filters (2)

 Example
  - m=3
  - MIN (TimeKey)=\{'01-Feb-2013', '04-Feb-2013', '07-Feb-2013'\}
  - query: WHERE TimeKey='02-Feb-2013' ⇒ the zone can be skipped

Zone Filters (3)

 BVF - Bit Vector Filter
  - maintained for each zone and for each column
  - provides a synopsis of the actual values
  - the m MIN and MAX values are not represented in the BVF

 ZI - Zone Index
  - supports searching within a zone
  - a dedicated ZI maintained for a zone
Materialization (6)

Application in queries
- select zones by means of BVFs
- find rows within zones by means of ZIs

Materialized query
- The result of a query persistently stored in a database
  - table (naive approach)
  - materialized view (Oracle, IBM Netezza), materialized query table/summary table (DB2), indexed view (SQL Server)
    - additional functionality
      - refreshing
      - query rewriting
create materialized view SalesMV1
build immediate
refresh force
with rowid
as
select ProdName, Category, Country, Month, Quarter,
    sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
    and s.CustomerID=c.CustomerID
    and s.TimeKey=t.TimeKey
group by ProdName, Category, Country, Month, Quarter;

alter materialized view SalesMV1 enable query rewrite;

create materialized view SalesMV1
...application
select Category, Country, Quarter, sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
    and s.CustomerID=c.CustomerID
    and s.TimeKey=t.TimeKey
group by Category, Country, Quarter, Year;

select Category, Country, Quarter, sum(SumSales)
from salesMV1
group by Category, Country, Quarter;
MV - example Oracle (3)

```
select Category, Country, Quarter, sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
group by Category, Country, Quarter;
```

query rewriting

```
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=170 Bytes=7 140)
1 0 HASH (GROUP BY) (Cost=4 Card=170 Bytes=7140)
2 1 MAT_VIEW REWRITE ACCESS (FULL) OF 'SALESMV1' (MAT_VIEW REWRITE)
   (Cost=3 Card=170 Bytes=7140)
```

MV - example Oracle (4)

```
create materialized view SalesMV2
...
select ProductID, Category, Country, Month, Quarter, Year,
sum(SalesPrice) as SumSales
from Sales s, Products p, Customers c, Time t
and s.TimeKey=t.TimeKey
group by ProductID, Category, Country, Month, Quarter, Year;
```

query rewriting

```
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=8 Card=175 Bytes=1 2425)
1 0 HASH (GROUP BY) (Cost=8 Card=175 Bytes=12425)
2 1 HASH JOIN (Cost=7 Card=175 Bytes=12425)
3 2 TABLE ACCESS (FULL) OF 'PRODUCTS' (TABLE) (Cost=3 Card=162 Bytes=4374)
4 2 MAT_VIEW REWRITE ACCESS (FULL) OF 'SALESMV2' (MAT_VIEW REWRITE)
   (Cost=3 Card=175 Bytes=7700)
```

ProductID → ProdName

query rewriting

join-back
MV example DB2 (1)

 análuated

© Maintained by:

- user: maintained by user clause
- system (default): maintained by system clause
  - either automatic or non-automatic refreshing mode is available
  - automatic mode (refresh immediate clause) ⇒ a MQT is refreshed automatically as the result of changes in the content of its base tables; this refreshing mode requires that a unique key from each base table is included in the MQT
  - non-automatic (refresh deferred clause) ⇒ a MQT has to be refreshed by explicit execution of:

```
refresh table TableName {incremental|not incremental}
```

MV - example DB2 (2)

```sql
create table YearlySalesMV2
as
(select ProdID, ProdName, Year,
sum(salesPrice) as SumSales
from Sales s, Products p, Time t
where s.ProductID=p.ProductID
and s.TimeKey=t.TimeKey
and t.Year=2009
group by ProdID, ProdName, Year)
data initially immediate
refresh immediate
maintained by system
enable query optimization;
```
For incremental refreshing

- MV log \( \Rightarrow \) staging table

```sql
create table YearlySalesMV3_ST for YearlySalesMV3
propagate immediate
set integrity for YearlySalesMV3
staging immediate unchecked
```

Used for query rewriting

- Stored as a table
- Divided into data slices that are co-located on the same data slices as the corresponding base table data slices

```sql
CREATE MATERIALIZED VIEW v-name AS
SELECT ... FROM tab-name [ORDER BY ...]
```

Some restrictions

- only one table in the FROM clause
- the WHERE clause cannot be used
- the columns in the projection list must be columns - expressions (aggregates, mathematical operators, SQL functions, DISTINCT, ...) are not allowed
- the columns in the optional ORDER BY clause must be one or more columns in the projection list
Inserting rows into a base table

- new rows are appended to the MV ⇒ two areas in the MV:
  - the sorted records generated when the view was created
  - the unsorted records that have been inserted into the base table after the MV was created
- resorting by manual refreshing

Suspending MV ⇒ making it inactive

Refreshing MV

- manually ⇒ the REFRESH option
- automatically ⇒ setting a refresh threshold
  - the thresholds allows to refresh all the materialized views associated with a base table
  - the threshold specifies the percentage of unsorted data in the materialized view, value from 1 to 99 (default 20)

The system creates zone maps for all columns in a MV that have data types integer, date, and timestamp
MV example - SQL Server

MV is created by creating a unique clustered index on a view (clustering data by the value of the indexed column).

The index causes that the view is materialized.

```
create view YearlySalesMV2
with schemabinding
as
select ProdID, ProdName, Year, sum(salesPrice) as SumSales
from Sales s, Products p, Time t
where s.ProductID=p.ProductID
and s.TimeKey=t.TimeKey
and t.Year=2009
group by ProdID, ProdName, Year
```

prevents from modifying base tables’ schemas as long as the view exists.

```
create unique clustered index Indx_ProdID
on YearlySalesMV(ProdID, ProdName, Year)
```

MV - SQL Server

Query rewriting: MV must be explicitly referenced in a query with `noexpand`.

```
select Column1, Column2, ...
from Table, IndexedView with (noexpand)
where ...
```

Refreshing: immediate and incremental.
MV refreshing

овал Refreshing time
- immediate
- deferred
  - automatic (with a defined frequency)
  - manual

овал Refreshing mode (A. Gupta, I.S. Mumick, MIT Press, 1999)
- full
- incremental
  - detecting changes in source tables
  - propagating the changes into a MV

овал Querying MVs during their refreshing ⇒ assuring data consistency (Zhuge et. al., SIGMOD, 2005)
- compensation algorithm
- versions of data

MV design

овал Designing the optimal set of MVs
oval Typically ⇒ for a given query workload

oval Constraints
- minimizing response time for the largest number of queries
- minimizing response time for the most expensive queries
- minimizing costs of refreshing MVs
- minimizing disk space

oval Physical design advisors
**Greedy algorithm**


**Assumptions**
- the data size of every query is known (estimated)
- a query execution cost is represented by the data volume that is read by the query

**Goal**
- minimize the data volume read by the queries with a given number of materialized views

---

**Execution (1)**

- **Node** ⇒ query
- **Arc** ⇒ the possibility of computing a query based on a lower-level query

---

**Run 1**
- B: $50 \times 5 = 250$
- C: $25 \times 5 = 125$
- D: $80 \times 2 = 160$
- E: $70 \times 3 = 210$
- F: $60 \times 2 = 120$
- G: $99 \times 1$
- H: $90 \times 1$
Execution (2)

Run 2
- C: \(25 \times 2 = 50\) (C and F; H, E, G computed from B)
- D: \(30 \times 2 = 60\)
- E: \(20 \times 3 = 60\)
- F: \(60 + 10 = 70\) (B-F)
- G: 49
- H: 40 (H computed from B)

Execution (3)

Run 3
- C: \(25 \times 2 = 50\) (C; E, G computed from B)
- D: \(30 \times 2 = 60\)
- E: \(20 + 20 + 10 = 50\) (E-F)
- G: 49
- H: 30
Partitioning (1)

- A mechanism of dividing a table or index into smaller parts ⇒ partitions
- The most benefit from partitioning is achieved if every partition is stored on a separate disc ⇒ parallel disc scans

Partitioning (2)

- Horizontal partitioning
- Vertical partitioning
- Mixed partitioning
Partitioning (3)

Correctness criteria

Completeness ⇔ when table $T$ is partitioned into $P_1, P_2, \ldots, P_n$ then every row from $T$ or its fragment must be stored in one of these partitions

- guarantees that after partitioning no data will disappear

Disjointness ⇔ when table $T$ is partitioned into $P_1, P_2, \ldots, P_n$ then every row or its fragment from $T$ must be stored in exactly one partition

- guarantees that partitioning does not create data redundancy

Reconstruction ⇔ there must be a mechanism of reconstructing original table $T$ from its partitions

Partitioning (4)

In horizontal partitioning rows are divided into subsets based on the value of partitioning attribute(s)

Horizontal partitioning techniques

- hash
- range-based
- set-based
- round-robin
```
Example Oracle (1)

**Range partitioning**

```sql
create table Sales_Range_TKey
(ProductID varchar2(8) not null references Products(ProductID),
 TimeKey date not null references time(TimeKey),
 CustomerID varchar2(10) not null references Customers(CustomerID),
 SalesPrice number(6,2))
PARTITION by RANGE (TimeKey)
(partition Sales_1Q_2009
 values less than (TO_DATE('01-04-2009', 'DD-MM-YYYY'))
tablespace Data01,
 partition Sales_2Q_2009
 values less than (TO_DATE('01-07-2009', 'DD-MM-YYYY'))
tablespace Data02,
 partition Sales_3Q_2009
 values less than (TO_DATE('01-10-2009', 'DD-MM-YYYY'))
tablespace Data03,
 partition Sales_4Q_2009
 values less than (TO_DATE('01-01-2010', 'DD-MM-YYYY'))
tablespace Data04,
 partition Sales_Others
 values less than (MAXVALUE) tablespace Data05);
```

Example Oracle (2)

**List partitioning**

```sql
create table Sales_List_PayType
(ProductID varchar2(8) not null references Products(ProductID),
 TimeKey date not null references time(TimeKey),
 CustomerID varchar2(10) not null references Customers(CustomerID),
 SalesPrice number(6,2),
 PaymentType varchar(2))
PARTITION by LIST (PaymentType)
(partition Sales_Credit_Debit values ('Cr','De') tablespace Data01,
 partition Sales_Cash values ('Ca') tablespace Data02,
 partition Sales_Others values (DEFAULT) tablespace Data05);
```

**Hash partitioning**

```sql
... PARTITION by HASH (CustomerID)
(partition Cust1 tablespace Data01,
 partition Cust2 tablespace Data02));
```
Other types of partitioning

- virtual column ⇒ based on expression on partitioning attribute(s)
- system ⇒ records placement in partitions controlled by an application
- reference ⇒ partitioning FK tables according to a partitioning schema of their PK table
- composite ⇒ partitions with subpartitions

Example Oracle

Example DB2

Range partitioning

```sql
create table Sales_Range_TKey
(ProductID varchar2(8), ...)
PARTITION BY RANGE(TimeKey)
(partition Sales_1Q_2009 starting '01-01-2009',
partition Sales_2Q_2009 starting '01-04-2009',
partition Sales_3Q_2009 starting '01-07-2009',
partition Sales_4Q_2009 starting '01-10-2009' ending '31-12-2009')
```
Example SQL Server

Partition function  defines the number of partitions for a table and ranges of values for every partition
Partition scheme  defines storage locations for table partitions

```
create PARTITION FUNCTION [Sales_Range_TKey] (datetime)
as RANGE right for values
('20090401', '20090701', '20091001', '20100101');
```

```
date < 2009-04-01
2009-04-01 <= date < 2009-07-01
2009-07-01 <= date < 2009-10-01
2009-10-01 <= date < 2010-01-01
date >= 2010-01-01
```

```
create PARTITION SCHEME PS_Sales_Range_TKey
as partition Sales_Range_TKey
to (Data01, Data02, Data03, Data04, Data05);
```

```
create table Sales_Range_TKey
(ProductID varchar(8),
TimeKey datetime ...)
on PS_Sales_Range_TKey (TimeKey)
```

Part. tables in queries

```
CREATE TABLE Sales1
(...)
PARTITION by RANGE (TimeKey)
(partition Sales_Jan2009
values less than (TO_DATE('01-02-2009', 'DD-MM-YYYY'))),
partition Sales_Feb2009
values less than (TO_DATE('01-03-2009', 'DD-MM-YYYY'))),
partition Sales_Mar2009
values less than (TO_DATE('01-04-2009', 'DD-MM-YYYY'))),
partition Sales_Apr2009
values less than (TO_DATE('01-05-2009', 'DD-MM-YYYY')));
```

```
select * from sales1
where TimeKey between to_date('01-01-2009', 'DD-MM-YYYY') and
to_date('31-01-2009', 'DD-MM-YYYY');
```

```
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=17 Card=7505 Bytes =562875)
1 0 PARTITION RANGE (SINGLE) (Cost=17 Card=7505 Bytes=562875)
2 1 TABLE ACCESS (FULL) OF 'SALES1' (TABLE) (Cost=17 Card=7505 Bytes=562875)
```
Partition-wise join (1)

```sql
CREATE TABLE orders
(order_id NUMBER(12) NOT NULL PRIMARY KEY,
 order_date DATE NOT NULL, ...)
PARTITION BY RANGE (order_date)
(PARTITION p_2006_jan VALUES LESS THAN (TO_DATE('01-FEB-2006', 'DD-MON-YYYY'))),
PARTITION p_2006_feb VALUES LESS THAN (TO_DATE('01-MAR-2006', 'DD-MON-YYYY'))),
PARTITION p_2006_mar VALUES LESS THAN (TO_DATE('01-APR-2006', 'DD-MON-YYYY'))),
PARTITION p_2006_apr VALUES LESS THAN (TO_DATE('01-MAY-2006', 'DD-MON-YYYY'))),
......
PARTITION p_2006_dec VALUES LESS THAN (TO_DATE('01-JAN-2007', 'DD-MON-YYYY')))
```

```sql
CREATE TABLE order_items
(order_id NUMBER(12) NOT NULL, ...,
 CONSTRAINT order_items_orders_fk FOREIGN KEY (order_id)
 REFERENCES orders(order_id))
PARTITION BY REFERENCE (order_items_orders_fk)
```

Partition-wise join (2)

```sql
SELECT o.order_date, sum(oi.sales_amount) sum_sales
FROM orders o, order_items oi
WHERE o.order_id = oi.order_id
AND o.order_date BETWEEN TO_DATE('01-FEB-2006', 'DD-MON-YYYY')
AND TO_DATE('31-MAR-2006', 'DD-MON-YYYY')
GROUP BY o.order_id, o.order_date
ORDER BY o.order_date;
```
MOLAP (1)

- Operations
  - drill-down / roll-up
  - slice, dice
  - rotate (pivot)
  - drill-across
  - drill-through

MOLAP (2)

- Implementations
  - N-dim array
  - Hash table (SQL Server)
  - BLOB (Oracle)
  - Quad-tree
  - K-D-tree

- Systems
  - Cognos PowerPlay (IBM)
  - Oracle OLAP DML
  - Hyperion Essbase (Oracle)
  - MicroStrategy
  - MS Analysis Services (Microsoft)
  - SAS OLAP Server
MOLAP (3)

Cube chunking

MOLAP (4)

Compression - store cells that contain NOT NULL values
- compress when % of NULL cells reaches a given threshold (e.g., 40%)

[1,4,C: value][4,4,D: value]
MOLAP (5)

✦ More efficient than ROLAP for aggregate computing
✦ Efficient when a cube contains a few dimension
✦ Loading less efficient than ROLAP

Our experiment (1)

✦ Oracle
✦ ROLAP
  ▪ TPC-H benchmark
  ▪ B-tree indexes on PKs and FKs
✦ MOLAP
  ▪ 4 cubes
  ▪ Discounts(Parts, Orders, ShipTime)
  ▪ Discounts(Parts, Orders, ReceiptTime)
  ▪ Quantities(Parts, Orders, ShipTime)
  ▪ Prices(Parts, Orders, ShipTime)
Our experiment (2)

Load time [hh:mm:ss]

![Graph showing load time for different data sizes and methods: SQL*Loader, SQL*Loader (direct load path), ROLAP->MOLAP.]

Our experiment (3)

MOLAP data size [MB]

![Graph showing MOLAP data size for different data sizes and methods: uncompressed ROLAP, compressed MOLAP, source data size.]

R.Wrembel - Poznan University of Technology
Remarks

- MOLAP may not always be more efficient than ROLAP
- ROLAP may take advantage of
  - bitmap idxs, bitmap join idxs, materialized views, partitioning
  - sophisticated query optimizer
- the presented results shouldn't be generalized ⇒ they show the characteristics of MOLAP implementation in a particular version of the system
Other directions

- Parallel and distributed DWs
  - data (partitions, MVs, indexes) allocation in nodes
  - load balancing and data redistribution

- In-memory/main-memory DWs
  - optimization of memory usage
  - compression

- DWs in a cloud
  - assuring scalability
  - load balancing and data redistribution
  - high availability
  - building DWs functionalities on Hadoop/Map Reduce
  - benchmarking

References

- Column storage
  - D.J. Abadi, S.R. Madden, M. Ferreira: Integrating compression and execution in column-oriented database systems. SIGMOD, 2006
  - D.J. Abadi, S.R. Madden, N. Hachem: Column-stores vs. row-stores: how different are they really?. SIGMOD, 2008
  - A. Albano: SADAS - An Innovative Column-Oriented DBMS for Business Intelligence Applications. SADAS Manual
References

Materialized views

- S. Kulkarni, M. Mohania.: Concurrent Maintenance of Views Using Multiple Versions. IDEAS, 1999
- S. Samtani, V. Kumar, M. Mohania: Self Maintenance of Multiple Views in Data Warehousing. CIKM, 1999

Small summary data

- IBM Netezza Database User’s Guide. IBM Netezza 7.0.x, Oct 2012
- Netezza underground: Zone maps and data power. 

Partitioning

- P. Furtado: Experimental evidence on partitioning in parallel data warehouses. DOLAP 2004
- P. Furtado: Algorithms for Efficient Processing of Complex Queries in Node-Partitioned Data Warehouses. IDEAS 2004
- P. Furtado: A Survey of Parallel and Distributed Data Warehouses. IJDWM 5(2), 2009
- L. Bellatreche, R. Bouchakri, A. Cuzzocrea, S. Maabout: Horizontal partitioning of very-large data warehouses under dynamically-changing query workloads via incremental algorithms. SAC, 2013