Advanced Data Management Technologies
Unit 13 — DW Pre-aggregation and View Maintenance

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Acknowledgements: I am indebted to M. Böhlen for providing me the lecture notes.
Outline

1. Pre-Aggregates
2. Lattice Framework
3. Greedy Algorithm
4. View Maintenance
Outline

1 Pre-Aggregates

2 Lattice Framework

3 Greedy Algorithm

4 View Maintenance
Aggregates/1

- **Observations**
  - DW queries are simple, follow the same "schema"
  - Aggregate measure per dim_attr_1, dim_attr_2, ...

- **Idea**
  - Compute and store query results in advance (preaggregation)

- **Example**: Store "total sales per month and product"
  - Yields large performance improvements (factor 100,1000, ...).
  - No need to store everything: re-use is possible.
    - e.g., quarterly total can be computed from monthly total.

- **Prerequisites for pre-aggregation**
  - Tree-structured dimensions.
  - Many-to-one relationships from fact to dimensions.
  - Facts mapped to bottom level in all dimensions.
  - Otherwise, re-use is not possible.
Pre-Aggregation Example

- Imagine 1 bio. sales rows, 1000 products, 100 locations
- Create a materialized view
  - `CREATE VIEW TotalSales (pid, locid, total) AS`
  - `SELECT s.pid, s.locid, SUM(s.sales)`
  - `FROM Sales s`
  - `GROUP BY s.pid, s.locid`
  - The materialized view has 100’000 rows.
- Query rewritten to use the view
  - `SELECT p.category, SUM(s.sales)`
  - `FROM Products p, Sales s`
  - `WHERE p.pid=s.pid`
  - `GROUP BY p.category`
  - Rewritten to
  - `SELECT p.category, SUM(t.total)`
  - `FROM Products p, TotalSales t`
  - `WHERE p.pid=t.pid`
  - `GROUP BY p.category`
- Query becomes 10’000 times faster!
Pre-Aggregation Choices

- **Full** pre-aggregation: all combinations of levels
  - Fast query response
  - Takes a lot of space/update time (200-500 times raw data)

- **No** pre-aggregation:
  - Slow query response (for terabytes)

- **Practical** pre-aggregation: chosen combinations
  - A good compromise between response time and space use

- Most (R)OLAP tools *today* support practical pre-aggregation
  - IBM DB2 UDB
  - Oracle 9iR2
  - MS Analysis Services
  - Hyperion Essbase (DB2 OLAP Services)
Using Aggregates

- Given a query, the best pre-aggregate must be found.
  - Should be done by the system, not by the user.

- The four design goals for aggregate usage:
  - Aggregates are stored separately from detail data.
  - “Shrunken” dimensions (i.e., subset of a dimension’s attributes that apply to the aggregation) are mapped to aggregate facts.
  - Connection between aggregates and detail data known by the system.
  - All queries (SQL) refer to detail data only.

- Aggregates are used via aggregate navigator
  - For a query, the best aggregate is found by the system, and the query is rewritten to use it.
  - Traditionally done in middleware, e.g., ODBC.
  - Can now (most often) be performed directly by the DBMS.

- SUM, MIN, MAX, COUNT, AVG can all be handled.
Choosing Aggregates

- Using practical pre-aggregation, it must be decided what aggregates to store.
- This is a non-trivial (NP-complete) optimization problem.
- Many influencing factors
  - Space use
  - Update speed
  - Response time demands
  - Actual queries
  - Prioritization of queries
  - Index and/or aggregates
- Only choose an aggregate if it is considerably smaller than available, usable aggregates (factor 3-5-10).
- Often supported (semi-)automatically by tools/DBMSs
  - Oracle, DB2, MS SQL Server
MS Analysis Aggregate Choice

- Can also log and use knowledge of actual queries.
Outline

1 Pre-Aggregates

2 Lattice Framework

3 Greedy Algorithm

4 View Maintenance
Implementing Data Cubes Efficiently

- The data cube stores multidimensional GROUP BY relations of tables in data warehouses.

- Classic SIGMOD 1996 paper
  - Harinarayan, Rajaraman, and Ullman: *Implementing Data Cubes Efficiently*.

- Simple but effective approach.

- Almost all DBMSes (ROLAP + MOLAP) now use similar, but more advanced, techniques for determining best aggregates to materialize.
Example: Sales fact table with dimensions part (p), supplier (s), customer (c)
- 8 possible groupings of attributes (or views) with 3 dimensions.
- Each grouping gives the total sales as per that grouping.

- Groupings
  - part, supplier, customer (6M rows)
  - part, customer (6M)
  - part, supplier (0.8M)
  - supplier, customer (6M)
  - part (0.2M)
  - supplier (0.01M)
  - customer (0.1M)
  - none (1)

- 8 views organized into a lattice
Picking the right views to materialize improves the query performance.

**Query:** What are the sales of a part?
- If view \( pc \) is available, will need to process about 6M rows.
- If view \( p \) is available, will need to process about 0.2M rows.

Questions
- How many views to materialize to get good performance?
- Given that we have space \( S \), what views to materialize to minimize average query costs?

View \( pc \) and \( sc \) are not needed!
- This reduces effective rows needed from 19M to 7M – a reduction of 60%. 

---

ADMT 2016/17 — Unit 13 J. Gamper
Lattice Framework

- **Lattice**: A pair \( (L, \leq) \), where \( L \) is a set of queries and \( \leq \) is a dependence relation.
  - \( Q_1 \leq Q_2 \) if query \( Q_1 \) can be answered using only the results of query \( Q_2 \).
  - In other words, \( Q_1 \) is dependent on \( Q_2 \).

- The \( \leq \) operator imposes a partial ordering on the queries.
- Partial ordering imposes strict requirements as to what is a lattice.
- However, in practice, we only need to assume there is a top view in which every view is dependent upon.
- Essentially, the lattice models dependencies among queries/views and can be represented by a lattice graph.
Hierarchies and the Lattice Framework

- Hierarchies are important as they underlay two commonly used query operations, drill-down and roll-up.

A common hierarchy

\[ \text{none} \rightarrow \text{Year} \rightarrow \text{Month} \rightarrow \text{Day} \rightarrow \text{Week} \]

... and its dependency relations

- Year \( \leq \) Month \( \leq \) Day
- Week \( \leq \) Day
- but Month \( \nless \) Week and Week \( \nless \) Month

BUT: hierarchies introduce query dependencies that must be accounted for when determining which queries to materialize; and this can be complex.
Composite Lattices

- Dependencies caused by different dimensions and attribute hierarchies can be combined into a **direct product lattice**.
- Assume views can be created by independently grouping any or no member of the hierarchy for each of the $n$ dimensions.
Applicability of Lattice Framework

- The lattice framework is advantageous for several reasons
  - It provides a clean framework to reason with dimensional hierarchies, since hierarchies are themselves lattices.
  - Able to model common queries better as users don’t jump between unconnected elements in the lattice, instead, they move along edges of the lattice.
  - A simple descending-order topological sort on the $\leq$ operator gives the required order of materialization.
  - A framework to calculate the cost of answering a query based on other queries.
Important assumptions

- Time to answer a query is equal to the space occupied by the query (view) from which the query is answered.
- All queries are identical to some queries in the given lattice.
- The clustering of the materialized query and indexes have not been considered.

Example:

- To answer query $Q$, we choose an ancestor of $Q$, say $Q_a$, that has been materialized.
- We thus need to process the table of $Q_a$.
- The cost of answering $Q$ is a function of the size of the table $Q_a$.
- Thus, the cost of answering $Q$ is the number of rows present in the table for that query $Q_a$ used to answer $Q$. 
An experimental validation of the cost model found almost a linear relationship between size and running time.

**Query:** Total sales for a supplier, using different views.

<table>
<thead>
<tr>
<th>Source</th>
<th>Size $S$</th>
<th>Time $T$</th>
<th>Ratio $m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>From cell itself</td>
<td>1</td>
<td>2.07</td>
<td>-</td>
</tr>
<tr>
<td>From view $s$</td>
<td>10,000</td>
<td>2.38</td>
<td>.000031</td>
</tr>
<tr>
<td>From view $ps$</td>
<td>0.8M</td>
<td>20.77</td>
<td>.000023</td>
</tr>
<tr>
<td>From view $psc$</td>
<td>6M</td>
<td>226.23</td>
<td>.000037</td>
</tr>
</tbody>
</table>

This relationship can be expressed by $T = m \times S + c$, where $c$ is the fixed cost and $m$ is the ratio of the query time to the size of the view (i.e., $m = (T - c)/S$).

**Assumption:** The number of rows present in each view is known (not simple, but many ways of estimating the size are available, e.g., sampling, use statistically representative subset).
Outline

1. Pre-Aggregates
2. Lattice Framework
3. Greedy Algorithm
4. View Maintenance
Greedy Algorithm

Given a data cube lattice with space costs associated with each view, the Greedy algorithm selects a set of $k$ views to materialize.

**Algorithm:** The Greedy algorithm

$S = \{\text{top view}\}$;

for $i = 1$ to $k$ do

- Select view $v$ not in $S$ such that the benefit $B(v, S)$ is maximized;
- $S = S \cup \{v\}$;

return $S$;

The algorithm optimizes the space-time trade-off.

- The top view should always be included because it cannot be generated from other views.
- Suppose we may only select $k$ number of views in addition to the top view.
- After selecting set $S$ of views, the benefit $B(v, S)$ of view $v$ relative to $S$, is based on how $v$ can improve the costs of evaluating views, including itself.
- The total benefit of $v$ is the sum over all views $w$ of the benefit of using $v$ to evaluate $w$, providing that benefit is positive.
Greedy Algorithm/2

The benefit $B(v, S)$ of view $v$ relative to $S$ is defined as follows:

- For each view $w \leq v$, define the quantity $B_w$ as follows:
  - Let $u$ be the view of least cost in $S$ such that $w \leq u$.
  - $B_w = \begin{cases} 
  C(u) - C(v) & \text{if } C(v) \leq C(u) \\
  0 & \text{otherwise}
  \end{cases}$

- Then, the benefit is $B(v, S) = \sum_{w \leq v} B_w$. 

[Diagram of a network with nodes and edges labeled with costs, including nodes u, a, b, c, d, v, f, g, h, and w]
Greedy Algorithm: Example/1

- Consider the following lattice with the indicated space costs, which are used for calculating the benefit.
- Top view a must be chosen.
- We want to choose 3 other views.
- At each round, we pick the view that will result in the most benefits after accounting for results of previous rounds.
- In round 1, view b can answer 5 queries (d, e, g, h and itself) at a cost of 50 each.
- This represents a cost reduction of 250 as compared to if view b, d, e, g, h were to be answered by using view a at a cost of 100 each.
- Thus, view b gives the biggest benefit of 250.

Benefits of possible choices at each round

<table>
<thead>
<tr>
<th>View</th>
<th>Choice 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>50 \times 5 = 250</td>
</tr>
<tr>
<td>c</td>
<td>25 \times 5 = 125</td>
</tr>
<tr>
<td>d</td>
<td>80 \times 2 = 160</td>
</tr>
<tr>
<td>e</td>
<td>70 \times 3 = 210</td>
</tr>
<tr>
<td>f</td>
<td>60 \times 2 = 120</td>
</tr>
<tr>
<td>g</td>
<td>99 \times 1 = 99</td>
</tr>
<tr>
<td>h</td>
<td>90 \times 1 = 90</td>
</tr>
</tbody>
</table>
In round 2, the cost of view $a$ of 100 applies only to certain views.

- $b$, $d$, $e$, $g$ and $h$ would have a cost of 50.
- Thus, the benefit of view $f$ wrt view $h$ is the difference between 50 and 40.
- After 3 rounds, the total costs of evaluating all views can be reduced to 420 from the initial 800.

### Benefits of possible choices at each round

<table>
<thead>
<tr>
<th></th>
<th>Choice 1</th>
<th>Choice 2</th>
<th>Choice 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>$50 \times 5 = 250$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td>$25 \times 5 = 125$</td>
<td>$25 \times 2 = 50$</td>
<td>$25 \times 1 = 25$</td>
</tr>
<tr>
<td>$d$</td>
<td>$80 \times 2 = 160$</td>
<td>$30 \times 2 = 60$</td>
<td>$30 \times 2 = 60$</td>
</tr>
<tr>
<td>$e$</td>
<td>$70 \times 3 = 210$</td>
<td>$20 \times 3 = 60$</td>
<td>$2 \times 20 + 10 = 50$</td>
</tr>
<tr>
<td>$f$</td>
<td>$60 \times 2 = 120$</td>
<td>$60 + 10 = 70$</td>
<td></td>
</tr>
<tr>
<td>$g$</td>
<td>$99 \times 1 = 99$</td>
<td>$49 \times 1 = 49$</td>
<td>$49 \times 1 = 49$</td>
</tr>
<tr>
<td>$h$</td>
<td>$90 \times 1 = 90$</td>
<td>$40 \times 1 = 40$</td>
<td>$30 \times 1 = 30$</td>
</tr>
</tbody>
</table>
Greedy Algorithm vs. Optimal Choice

There will be situations where the algorithm does poorly.

- **Round 1**: Picks c whose benefit is 4141.
- **Round 2**: Can pick b or d with benefits of 2100 each.
- **Greedy results in benefit of** \(4141 + 2100 = 6241\).
- **But, the optimal choice is to pick** b and d.
- **b and d would improve by 100 for itself and all 80 nodes below resulting in total benefits of 8200.**
- **Ratio of greedy/optimal = 6241/8200 = 76%**
- **But: the benefit of the greedy algorithm is at least 63% of the benefit of the optimal algorithm (shown by the authors).**
Greedy Algorithm – Space vs. Time

- Experiment with composite lattice shows that it is important to materialize some views but not all.
- Performance increases at first, but after 5 views, increase of performance gets small even as more space is used.

Greedy order of view selection for TPC-D based example.

<table>
<thead>
<tr>
<th>Selection</th>
<th>Benefit</th>
<th>TotTime</th>
<th>TotSpace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cp</td>
<td>infinite</td>
<td>72M rows</td>
</tr>
<tr>
<td>2</td>
<td>ns</td>
<td>24M rows</td>
<td>48M</td>
</tr>
<tr>
<td>3</td>
<td>nt</td>
<td>12M</td>
<td>36M</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>5.9M</td>
<td>30.1M</td>
</tr>
<tr>
<td>5</td>
<td>p</td>
<td>5.8M</td>
<td>24.3M</td>
</tr>
<tr>
<td>6</td>
<td>cs</td>
<td>1M</td>
<td>23.3M</td>
</tr>
<tr>
<td>7</td>
<td>np</td>
<td>1M</td>
<td>23.3M</td>
</tr>
<tr>
<td>8</td>
<td>ct</td>
<td>0.01M</td>
<td>23.3M</td>
</tr>
<tr>
<td>9</td>
<td>t</td>
<td>small</td>
<td>23.3M</td>
</tr>
<tr>
<td>10</td>
<td>n</td>
<td>small</td>
<td>23.3M</td>
</tr>
<tr>
<td>11</td>
<td>s</td>
<td>small</td>
<td>23.3M</td>
</tr>
<tr>
<td>12</td>
<td>none</td>
<td>small</td>
<td>23.3M</td>
</tr>
</tbody>
</table>
Optimal Cases and Anomalies

- Two situations where the algorithm is optimal.
  - If the benefit of the first view is much larger than the other benefits, the greedy is close to optimal.
  - If all the benefits are equal then greedy is optimal.

- But there are also two situations where the algorithm is not realistic.
  - Views in a lattice are unlikely to have the same probability of being requested in a query; hence, probabilities should be associated to each view.
  - Instead of asking for some fixed number of views to materialize, should instead allocate a fixed amount of space to views.
Hypercube Lattices – Observations

- The size of views grows exponentially, until it reaches the size of the raw data at rank \( \lceil \log_r m \rceil \) (i.e., the “cliff”).

![Graph showing size of views vs. log, m](image)

- Assumptions and basis of reasoning
  - Each domain size is \( r \).
  - Top element has \( m \) cells appearing in raw data.
  - If group on \( i \) attributes, cube has \( r^i \) cells.
  - If \( r^i \geq m \), then each cell will have at most one data point. Space cost is \( m \).
  - If \( r^i < m \), then almost all \( r^i \) cells will have at least one data point. Space cost is \( r^i \) as several data points can be collapsed into one aggregate.

- This explains why grouping of 2 attributes (p,c), (s,c) have the same size as (p,s,c) at 6M rows.
Space- and Time-optimal Solutions

- Inevitably, questions will be raised about space and time optimality of hypercubes.
- What is the average time for a query when the space is optimal?
  - Space is minimized when only the top view is materialized.
  - Every query would take time \( m \).
  - Total time cost for all \( 2^n \) queries is \( m2^n \).
- Is there sense to minimize time by materializing all views?
  - No gain past the cliff.
  - No point to do so.
  - Nature of time-optimal solution is to get as close to the cliff as possible.
Outline

1. Pre-Aggregates
2. Lattice Framework
3. Greedy Algorithm
4. View Maintenance
View Maintenance

- Views (pre-aggregates) are used to speed up querying.
- **How** and **when** should we refresh materialized views?
- **Total re-computation**
  - Most often too expensive
- **Incremental view maintenance**
  - Apply only changes since last refresh to view.
  - $r_i =$ inserted rows into relation $r$
  - $r_d =$ deleted rows from relation $r$
- Additional info must be stored to make views self-maintainable
  - Number of derivations $c$ (count) along with each row in view $v$
  - Thus, tuples in view have the form $(a_1, \ldots, a_k, c)$
Projection View Maintenance

- **Projection** views with DISTINCT
- View \( v = \pi_{A_1, \ldots, A_k}(r) \)
- **Insertion** of tuples \( r_i \)

```plaintext
foreach tuple \((a_1, \ldots, a_k) \in \pi_{A_1, \ldots, A_k}(r_i)\) do
  Let \( c_i \) be \# occurrences of the tuple;
  if \((a_1, \ldots, a_k, c_i) \in v\) then
    \( c = c + c_i \)
  else
    Insert \((r, c_i)\) into \( V \)
```

- **Deletion** of tuples \( r_d \)

```plaintext
foreach \((a_1, \ldots, a_k) \in \pi_{A_1, \ldots, A_k}(r_d)\) do
  Let \( c_d \) be \# of occurrences of the tuple;
  if \((a_1, \ldots, a_k, c) \in v\) then
    \( c = c - c_d \)
  if \( c = 0 \) then
    Delete \((a_1, \ldots, a_k, c)\) from \( v \)
```
# Projection View Maintenance Example

Relation $r$, view $v$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>c</td>
</tr>
</tbody>
</table>

$\mathbf{v} = \pi_A(r)$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Insert tuple $(b, 4)$

$\mathbf{r}$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>c</td>
</tr>
</tbody>
</table>

$\mathbf{v} = \pi_A(r)$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Delete tuples $\{(c, 3), (a, 2)\}$

$\mathbf{r}$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

$\mathbf{v} = \pi_A(r)$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Join View Maintenance

- Join views
- View \( v = r \bowtie s \)
- Insertion of \( r_i \):
  - Compute \( r_i \bowtie s \) and add to \( v \), update counts.
- Deletion of \( r_d \):
  - Compute \( r_d \bowtie s \) and subtract from \( v \), update counts.
COUNT/SUM/AVG Aggregation View Maintenance

**COUNT**
- Maintain tuples of the form \((g_1, \ldots, g_m, c)\)
  - \(g_1, \ldots, g_m\) are the grouping attribute values
  - \(c\) is a counter
- Update count \(c\) based on inserts \((r_i)\) and deletes \((r_d)\)
- Insert row \((g_1, \ldots, g_m, 1)\) for new groups
- Delete row \((g_1, \ldots, g_m, c)\) from \(v\) if \(c = 0\)

**SUM**
- Maintain tuples of the form \((g_1, \ldots, g_m, sum, c)\)
- Update count \((c)\) and sum \((sum)\) based on inserts \((r_i)\) and deletes \((r_d)\)
- Insert row \((g_1, \ldots, g_m, val, 1)\) for new grouping attribute values (\(val\) is the value of attribute over which SUM is applied)
- Delete row \((g_1, \ldots, g_m, sum, c)\) from \(v\) if \(c = 0\).

**AVG**
- Computed as pair SUM/COUNT
MIN (MAX works similar)

- Maintain tuples \( x = (g_1, \ldots, g_m, min, c) \)
- Update \( min \) and \( c \) based on inserts \( (r_i) \) and deletes \( (r_d) \) and whether \( val \in \{=, <, >\} \ min \)
- Insert tuple \( (g_1, \ldots, g_m, val) \)
  
  \[
  \begin{align*}
  &\text{if } val < \text{min} \text{ then} \\
  &\quad x = (g_1, \ldots, g_m, val, 1) \\
  \text{else if } val = \text{min} \text{ then} \\
  &\quad x = (g_1, \ldots, g_m, \text{min}, c + 1)
  \end{align*}
  \]

- Delete tuple \( (g_1, \ldots, g_m, val) \)
  
  \[
  \begin{align*}
  &\text{if } val = \text{min} \text{ then} \\
  &\quad x = (g_1, \ldots, g_m, \text{min}, c - 1); \\
  &\quad \text{if } c = 0 \text{ then} \\
  &\quad \quad \text{Scan table for new values for min and c (expensive!)}
  \end{align*}
  \]
Determine a view for MIN using SQL
- Input: relation \( r \) with schema \((A, B)\)
- Output: relation with schema \((A, MIN(B), \text{count of } MIN(B))\)

**Solution 1**

```sql
SELECT t.*, ( SELECT COUNT(*)
    FROM r
    WHERE A = t.A AND B = t.B )
FROM ( SELECT A, min(B) B
    FROM r
    GROUP BY A ) t;
```

<table>
<thead>
<tr>
<th>r</th>
<th>t</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Aggregation View Maintenance Example/2

Solution 2

```sql
SELECT A, B, COUNT(*)
FROM r
GROUP BY A, B
HAVING (A, B) IN ( SELECT A, MIN(B)
    FROM r
    GROUP BY A );
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>MIN(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>MIN(B)</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
### Solution 3

```sql
SELECT A, B, COUNT(*)
FROM r AS t
WHERE B = ( SELECT MIN(B)
               FROM r
               WHERE A = t.A )
GROUP BY A, B;
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>MIN(B)</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>MIN(B)</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**result**
Solution 4 using GMD-join

\[ x = \text{MD}( r/b, \]
\[ r, \]
\[ ( (\text{MIN}(B)/\text{Min}), (\text{COUNT}(\ast)/\text{Cnt}) ), \]
\[ ( (r.A = b.A), (r.A = b.A \text{ AND } r.B = b.B) ) ) \]

result = \( \pi_{a,\text{min},\text{cnt}}(\sigma_{b=\text{min}}(x)) \)
Practical View Maintenance

**When** to synchronize views?
- **Immediate** - in same transaction as base changes.
- **Lazy** - when view is used for the first time after base updates.
- **Periodic** – e.g., once a day, often together with base load.
- **Forced** - after a certain number of changes.

**Updating aggregates**
- Computation outside DBMS in flat files (no longer very relevant!).
- Built by loader.
- Computation in DBMS using SQL.
- Can be expensive: DBMS must be tuned for this.

**Supported by tool/DBMS**
- Oracle, SQL Server, DB2
**Summary**

- **Pre-aggregation** is a key technique to boost performance.
- Data warehouses **automatically determine** views to materialize and when to use them.
- Problems in deciding which set of views to materialize to improve query performance.
- **Lattice framework**: views are organized in a lattice.
- Notion of linear cost in query processing.
- **Greedy algorithm** that picks the right views.
- Some observations about hypercubes and time-space trade-off.
- Views have to be **maintained**.
- **Incremental view maintenance** is state-of-the-art
  - Needs to store a count to trace the number of supporting tuples.