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

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)

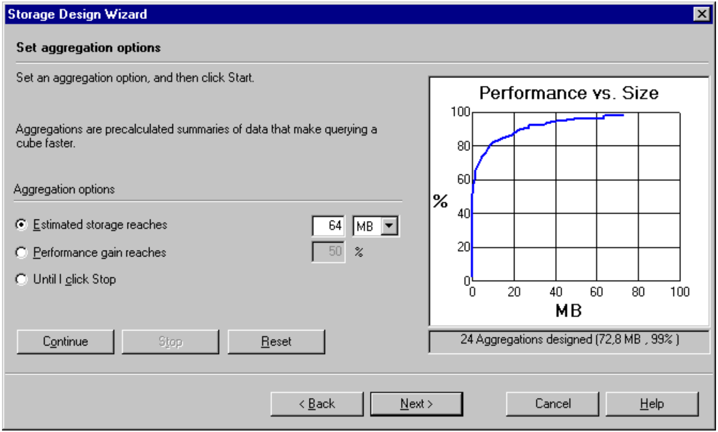
Using Aggregates

- 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.
  - "Shrunk" 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



Storage Design Wizard

Set aggregation options

Set an aggregation option, and then click Start.

Aggregations are precalculated summaries of data that make querying a cube faster.

Aggregation options

● Estimated storage reaches 64 MB

○ Performance gain reaches 50 %

○ Until I click Stop

24 Aggregations designed (72,8 MB , 99%)

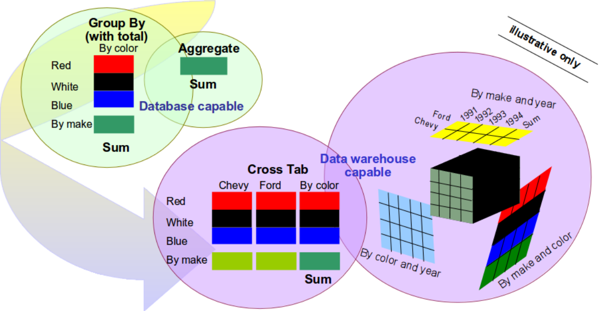
- Can also log and use knowledge of actual queries.

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.



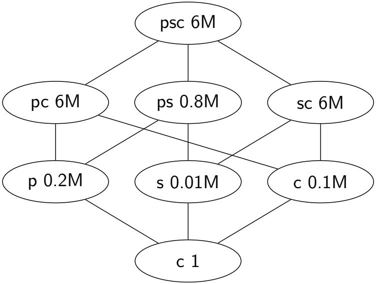
- Classic SIGMOD 1996 paper
  - Harinarayan, Rajaraman, and Ullman: *Implementing Data Cubes Efficiently*.
- Simple but effective approach.
- Almost all DBMSes (ROLAP + MOLAP) now use similar, but more advanced, techniques for determining best aggregates to materialize.

A Data Cube Example/1

Example: Sales fact table with dimensions part (p), supplier (s), customer (c)

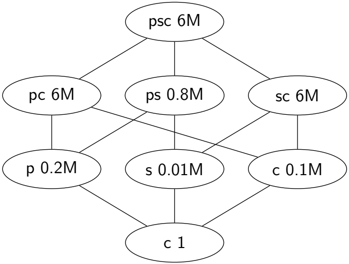
- 8 possible groupings of attributes (or views) with 3 dimensions.
- Each grouping gives the total sales as per that grouping.

- 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**



A Data Cube Example/2

- 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.

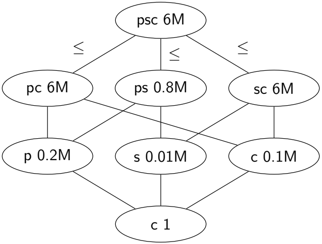


- Questions
  - How many views to materialize to get good performance?
  - Given that we have space S, what views to materialize to minimize average query costs?
- View pc and sc are not needed!
  - This reduces effective rows needed from 19M to 7M – a reduction of 60%.

Lattice Framework

- **Lattice:** A pair $(L, \leq)$, where L is a set of queries and $\leq$ is a dependence relation.
  - $Q1 \leq Q2$ if query $Q1$ can be answered using only the results of query $Q2$.
  - In other words, Q1 is dependent on Q2.

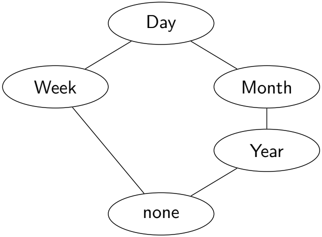
- The $\leq$ operator imposes a partial ordering on the queries.
- Partial ordering imposes strict requirements as to what is a lattice.
- However, in practice, we only need to assume there is a top view in which every view is dependent upon.
- Essentially, the lattice models dependencies among queries/views 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



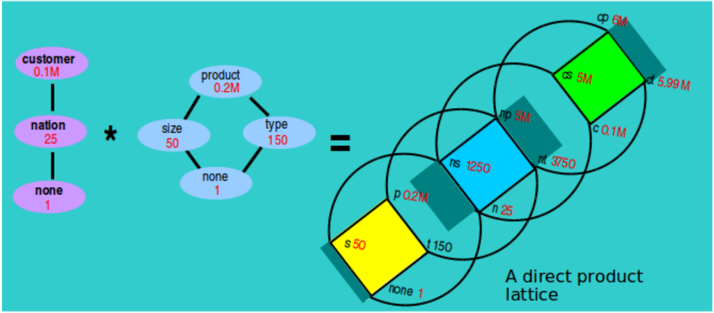
. . . and its dependency relations

- $Year \leq Month \leq Day$
- $Week \leq Day$
- but $Month \not\leq Week$ and $Week \not\leq Month$

- 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.

Cost Model/1

- 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.

Cost Model/2

- 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.

Source	**Size S**	**Time T**	**Ratio m**
From cell itself	1	2.07	-
From view s	10,000	2.38	.000031
From view ps	0.8M	20.77	.000023
From view psc	6M	226.23	.000037

- This relationship can be expressed by $T = m * S + c$, 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/1

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

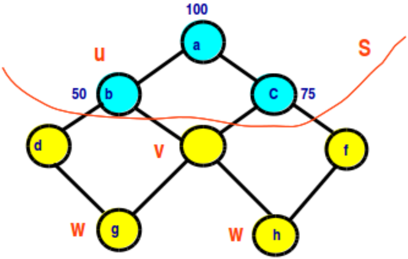
  Algorithm: The Greedy algorithm

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

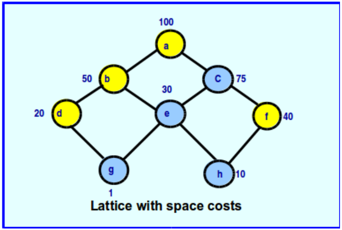
Greedy Algorithm/2

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



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.



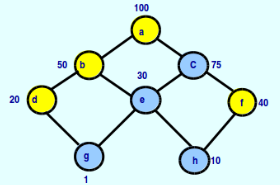
Lattice with space costs

Benefits of possible choices at each round

View	Choice 1
b	**50 x 5 = 250**
c	25 x 5 = 125
d	80 x 2 = 160
e	70 x 3 = 210
f	60 x 2 = 120
g	99 x 1 = 99
h	90 x 1 = 90

Greedy Algorithm: Example/2

- 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.



Lattice with space costs

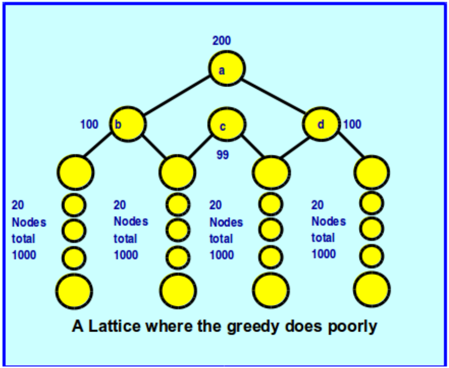
Benefits of possible choices at each round

	Choice 1	**Choice 2**	**Choice 3**
b	**50 x 5 = 250**		
c	25 x 5 = 125	25 x 2 = 50	25 x 1 = 25
d	80 x 2 = 160	30 x 2 = 60	**30 x 2 = 60**
e	70 x 3 =210	20 x 3 = 60	2 x 20 + 10 =50
f	60 x 2 =120	**60 + 10 = 70**	
g	99 x 1 = 99	49 x 1 = 49	49 x 1 = 49
h	90 x 1 = 90	40 x 1 = 40	30 x 1 = 30

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.



A Lattice where the greedy does poorly

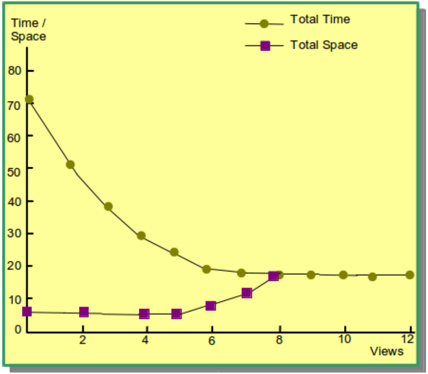
- Ratio of $\textit{greedy}/\textit{optimal} = 6241/8200 = 76\%$
- But: the benefit of the greedy algorithm is at least 63% of the benefit of the optimal algorithm (shown by the authors).

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.

	Selection	Benefit	TotTime	TotSpace
1	cp	infinite	72M rows	6nM rows
2	ns	24M rows	48M	6M
3	nt	12M	36M	6M
4	c	5.9M	30.1M	6.1M
5	p	5.8M	24.3M	6.3M
6	cs	1M	23.3M	11.3M
7	np	1M	23.3M	16.3M
8	ct	0.01M	23.3M	23.3M
9	t	small	23.3M	23.3M
10	n	small	23.3M	23.3M
11	s	small	23.3M	23.3M
12	none	small	23.3M	23.3M

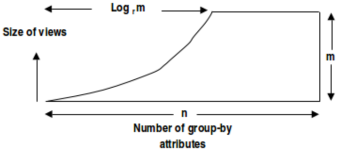


Optimal Cases and Anomalies

- Two situations where the algorithm is optimal.
  - If the benfit 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”).



- 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 atmost 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.
  - $\mathbf{r}_i =$ inserted rows into relation $\mathbf{r}$
  - $\mathbf{r}_d =$ deleted rows from relation $\mathbf{r}$
- Additional info must be stored to make views self-maintainable
  - Number of derivations c (count) along with each row in view $\mathbf{v}$
  - Thus, tuples in view have the form $(a_1, \dots, a_k, c)$

Projection View Maintenance

- Projection views with `DISTINCT`
- View $\mathbf{v} = \pi_{A_1,\ldots,A_k}(\mathbf{r})$
- Insertion of tuples $\mathbf{r}_i$

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

- Deletion of tuples $\mathbf{r}_d$

```
foreach $(a_1, \ldots, a_k) \in \pi_{A_1,\ldots,A_k}(\mathbf{r}_d)$ do
    Let $c_d$ be # of occurrences of the tuple;
    if $(a_1, \ldots, a_k, c) \in \mathbf{v}$ then
        $c = c - c_d$
    if $c = 0$ then
        Delete $(a_1, \ldots, a_k, c)$ from $\mathbf{v}$
```

Projection View Maintenance Example

Relation **r**, view **v**

r

A	B
a	1
a	2
b	2
c	3

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

A	C
a	2
b	1
c	1

Insert tuple $(b, 4)$

r

A	B
a	1
a	2
b	2
c	3
b	4

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

A	C
a	2
b	2
c	1

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

r

A	B
a	1
b	2
b	4

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

A	C
a	1
b	2

Join View Maintenance

- Join views
- View $\mathbf{v} = \mathbf{r} \bowtie \mathbf{s}$
- Insertion of $\mathbf{r}_i$
    - Compute $\mathbf{r}_i \bowtie \mathbf{s}$ and add to $\mathbf{v}$, update counts.
- Deletion of $\mathbf{r}_d$
    - Compute $\mathbf{r}_d \bowtie \mathbf{s}$ and subtract from $\mathbf{v}$, 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 $(\mathbf{r}_i)$ and deletes $(\mathbf{r}_d)$
  - Insert row $(g_1, \ldots, g_m, 1)$ for new groups
  - Delete row $(g_1, \ldots, g_m, c)$ from $\mathbf{v}$ 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 $(\mathbf{r}_i)$ and deletes $(\mathbf{r}_d)$
  - 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 $\mathbf{v}$ if $c = 0$.
- **AVG**
  - Computed as pair SUM/COUNT

MIN/MAX Aggregation View Maintenance

- **MIN** (**MAX** works similar)
    - Maintain tuples $x = (g_1, \ldots, g_m, min, c)$
    - Update *min* and *c* based on inserts ($\mathbf{r}_i$) and deletes ($\mathbf{r}_d$) and whether val $\{=, <, >\}$ min
    - Insert tuple $(g_1, \ldots, g_m, val)$

```
if $val < min$ then
    $x = (g_1, \ldots, g_m, val, 1)$
else if $val = min$ then
    $x = (g_1, \ldots, g_m, min, c + 1)$
```

-
    - Delete tuple $(g_1, \ldots, g_m, val)$

```
if $val = min$ then
    $x = (g_1, \ldots, g_m, min, c - 1)$;
    if $c = 0$ then
        Scan table for new values for $min$ and $c$ (expensive!)
```

Aggregation View Maintenance Example/1

- 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

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

r

A	B
1	2
1	2
1	3
2	3

t

A	MinB
1	2
2	3

result

A	MinB	Cnt
1	2	2
2	3	1

Aggregation View Maintenance Example/2

Solution 2

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

r

A	B
1	2
1	2
1	3
2	3

A	MIN(B)
1	2
2	3

result

A	MIN(B)	Cnt
1	2	2
2	3	1

Aggregation View Maintenance Example/3

Solution 3

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

r

A	B
1	2
1	2
1	3
2	3

a=1

MIN(B)
2

a=2

MIN(B)
3

result

A	MIN(B)	Cnt
1	2	2
2	3	1

Aggregation View Maintenance Example/4

Solution 4 using GMD-join

```
x = MD( r/b,
        r,
        ( (MIN(B)/Min), (COUNT(*)/Cnt) ),
        ( (r.A = b.A), (r.A = b.A AND r.B = b.B) ) )
```

`result =` $\pi_{a,min,cnt}(\sigma_{b=min}(x))$

r

a	b
1	2
1	2
1	3
2	3

x

a	b	min	cnt
1	2	2	2
1	2	2	2
1	3	2	1
2	3	3	1

result

a	min(b)	cnt
1	2	2
2	3	1

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.