Warehouses & Virtual Databases offer Integrated & Added-Value Views

Focus: Sources are relational DBs. Integration specified by distributed view definition(s). Clients issue queries on views.
Virtual View -> Mediator is a Distributed Query Processor
Materialized view (warehouse) -> Mediator actually stores integrated view
Distributed Query Processing in Mediators

Distributed Query processing

Translate subplans to SQL

+ How will the join happen?

- What if the source offers multiple data services instead of JDBC access?
Distributed Join Types

- **Mediator-based Join**
  - Ship results of queries at mediator

- **Parameterized Join**
  - Right subquery is enhanced with selection on join attribute
  - For each join value of left hand side, execute another right subquery

- **Data Ship Join**
  - Insert the result of left hand side (lhs) in the db of right hand side (rhs).
  - Execute join at db of right hand side

- **Semijoin Reduction Join**
  - Send rhs parameters to lhs
  - (Data ship alike variation) Lhs sends to rhs the semijoin of its subquery with the parameters set.
  - Execute join at db of rhs
  - Also, variation that looks like mediator-based join

Virtual Views Vs Materialized Views

<table>
<thead>
<tr>
<th>View kind</th>
<th>CREATE VIEW V AS</th>
<th>CREATE MATERIALIZED VIEW V AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE VIEW V AS</td>
<td>SELECT G, SUM(A) AS S FROM R GROUP BY G</td>
<td>SELECT G, SUM(A) AS S FROM R GROUP BY G</td>
</tr>
</tbody>
</table>

Upon Updating R
- Database does nothing
  - (Ideally) Database must refresh V to reflect changes on R

Upon Querying V
- Optimize & run
  - \( \sigma_{G=5} \) R
  - \( \sigma_{G=5} \) V

Upon Updating R
- Database does nothing
  - (Ideally) Database must refresh V to reflect changes on R

---

7. Also, variation that looks like mediator-based join

8. Upon Querying V
- Optimize & run
  - \( \sigma_{G=5} \) R
  - \( \sigma_{G=5} \) V

---

4
Recompute Vs Incremental (Materialized View) Maintenance – Informal Example

CREATE MATERIALIZED VIEW V AS
SELECT G, SUM(A) AS S
FROM R
GROUP BY G

(start)
INSERT INTO R (...);
...
INSERT INTO R (...);
...
INSERT INTO R (...);
commit

DELETE FROM V WHERE true;
INSERT INTO V
(SELECT G, SUM(A) AS S FROM R
GROUP BY G);

At the end of the transaction, we want V to reflect the new state of R
Option 1: Delete and Recompute V
Option 2: Incrementally maintain V

\( \Delta R^+ \): the set of tuples inserted in R
( obtained by log or other mechanism )

UPDATE V
SET S = S +
(SELECT SUM(A)
FROM \( \Delta R^+ \)
WHERE \( \Delta R^+.G = V.G \))
WHERE V.G IN (SELECT G FROM \( \Delta R^+ \));
INSERT INTO V
(SELECT G, SUM(A) AS S FROM \( \Delta R^+ \)
WHERE NOT G IN (SELECT G FROM V)
GROUP BY G);

Capturing IVM as computation of \( \Delta V^+ \), \( \Delta V^- \)

• ignore (just for simplicity) update commands
  • think of update as delete – insert combo
• input is \( \Delta R^+ \), \( \Delta R^- \)
• compute “tuples to be deleted from the view” \( \Delta V^- \)
• compute “tuples to be inserted in the view” \( \Delta V^+ \)
• delete \( \Delta V^- \) from V, insert \( \Delta V^+ \) in V

CREATE MATERIALIZED VIEW V AS
SELECT G, SUM(A) AS S
FROM R
GROUP BY G

\( \text{choose}(a,b) \) returns \( a \) if \( a \) is NOT NULL,
returns \( b \) if \( a \) is NULL
\( n0(a) \) returns \( a \) if \( a \) is NOT NULL, 0 otherwise

WITH RGplus AS
(SELECT G, SUM(A) AS S
FROM \( \Delta R^+ \)
GROUP BY G ),
RGminus AS
(SELECT G, SUM(A) AS S
FROM \( \Delta R^- \)
GROUP BY G),
RGnet AS
(SELECT choose(p.G, m.G) AS G,
n0(p.S) – n0(m.S) AS S
FROM RGplus AS p FULL OUTER JOIN
RGminus AS m ON p.G=m.G
\( \Delta V^- \) AS
(SELECT * FROM V
WHERE G IN
(SELECT G FROM RGnet))
\( \Delta V^+ \) AS
(SELECT r.G AS G,
n0(V.S) + r.S AS S
FROM (V RIGHT OUTER JOIN
RGnet AS r ON V.G=r.G)

Snapshot 0

Problem: Find efficient view updates

\[ \Delta V^+ = f^+(\Delta R_1^+, \ldots, \Delta R_n^+ \mid \Delta R_1^-, \ldots, \Delta R_n^- \mid V^0 \mid R_1^0, \ldots, R_n^0) \]

View

\[ V^0 = V(R_1^0, \ldots, R_n^0) \]

\[ \Delta V^+ = f(\) \]

\[ \rightarrow V^1 = V(R_1^1, \ldots, R_n^1) \]

Database tables

\[ R_1^0, \ldots, R_n^0 \]

\[ \rightarrow R_1^1, \ldots, R_n^1 \]

Table Updates

- \( \Delta R_1^+, \ldots, \Delta R_n^+ \)
- \( \Delta R_1^-, \ldots, \Delta R_n^- \)

From logs or intercepted by triggers

IVM: Deferred version

Snapshot 0

Problem: Find efficient view updates

\[ \Delta V^+ = f^+(\Delta R_1^+, \ldots, \Delta R_n^+ \mid \Delta R_1^-, \ldots, \Delta R_n^- \mid V^0 \mid R_1^1, \ldots, R_n^1) \]

View

\[ V^0 = V(R_1^0, \ldots, R_n^0) \]

\[ \Delta V^+ = f(\) \]

\[ \rightarrow V^1 = V(R_1^1, \ldots, R_n^1) \]

Database tables

\[ R_1^0, \ldots, R_n^0 \]

\[ \rightarrow R_1^1, \ldots, R_n^1 \]

Table Updates

- \( \Delta R_1^+, \ldots, \Delta R_n^+ \)
- \( \Delta R_1^-, \ldots, \Delta R_n^- \)

From logs or intercepted by triggers
IVM: Self-maintaining version
(not always possible)

Snapshot 0

Problem: Find efficient view updates

\[ \Delta V^* = f^*(\Delta R_1^+, \ldots, \Delta R_n^+, \Delta R_1^-, \ldots, \Delta R_n^-, V^0) \]

\[ \Delta V^- = f^-(\ldots) \]

\[ V^1 = V(R_1^1, \ldots, R_n^1) \]

Snapshot 1

View

\[ V^0 = V(R_1^0, \ldots, R_n^0) \]

Database tables

\[ R_1^0, \ldots, R_n^0 \rightarrow R_1^1, \ldots, R_n^1 \]

Table Updates

\[ \Delta R_1^+, \ldots, \Delta R_n^+ \]

\[ \Delta R_1^-, \ldots, \Delta R_n^- \]

From logs or intercepted by triggers

Basic IVM Algorithm:
Compose operator IVM rules

Example (wlog deferred, i.e., R means \( R^1 \) and S means \( S^1 \))

- Rule for \( V = R \bowtie S \)
  - \( \Delta V^* = (\Delta R^+ \bowtie S) \cup (R \bowtie \Delta S^+) \) - \( (\Delta R^+ \bowtie \Delta S^+) \)
  - \( \Delta V^- = ??? \)

- Rule for \( V = \sigma_c R \)
  - \( \Delta V^* = \sigma_c \Delta R^+ \)
  - \( \Delta V^- = ??? \)

- Composition of rules leads to solutions for
  \( V = T \bowtie \sigma_{A>5} W \)

\[ \Delta V^* = \ldots \]

\[ \Delta V^- = \ldots \]

- May rewrite initial expression
IVM with Caching

- May associate intermediate views (caches) with subexpressions
- Bottom-up: From updating caches to reaching the materialized view
- Caches will typically need indices
- Caches may or may not pay off as they incur cost for maintaining them (and their indices)

Generalizations

- Multiple views
  - Self maintenance may involve a view utilizing the other views in its computation
- Genuine updates
  - Not simulated via insertions/deletions
- Insertions, deletions, updates on tables and views expressed as DML statements
Comparisons

Materialized View

- High query performance
- Queries not visible outside warehouse
- Local processing at sources unaffected
- Can operate when sources unavailable
- Extra information at warehouse
  - Modify, summarize (store aggregates)
  - Add historical information

Virtual View

- No need for yet another database
- More up-to-date data
  - Depending on specifics of IVM
- Query needs can be unknown
- Only query interface needed at sources
  => Lower Total Cost of Ownership

Performance revisited: What if indices are not enough for decent online performance?

- Buy RAM
- Use a column database
  - In analytics queries can give a 10x easily
- Scalable, parallel processing
  - Mostly via no SQL
- Precompute
  - Fast answers!
  - Penalty: Cost of maintaining precomputed results
  - Applicability depends on schema and queries
  - Star schemas and summation are a good (but not the only) target of precomputation
Precomputation problems

Steps:
1. Choose what data to precompute
2. Use the precomputed data smartly in your queries
3. Update smartly the precomputed data as the database changes (IVM)

Tradeoff:
- Precomputed data accelerate analytics => faster queries
- But need to be updated => cost

Example: Precomputation and its Use

Database has huge table \textit{Sales}(product, store, date, amt)

Application issues often this slow query and displays the results
\begin{verbatim}
SELECT product, SUM(amt) AS sumamt
FROM Sales
GROUP BY product
\end{verbatim}

To improve performance we precompute table
\begin{verbatim}
ProductSales(product, sumamt)
\end{verbatim}

and insert in it the precomputed data by
\begin{verbatim}
INSERT INTO ProductSales ( product, sumamt )
SELECT product, SUM(amt) AS sumamt
FROM Sales
GROUP BY product
\end{verbatim}

Now the application issues instead this fast query below
\begin{verbatim}
SELECT *
FROM ProductSales
\end{verbatim}
Example (cont’d)

Now we have to keep up to date the ProductSales(product, sumamt) as new sales happen. E.g., if another $10 of product 23 were just sold:

```
UPDATE ProductSales
SET sumamt = sumamt + 10
WHERE product = 23
```

(in actual code it will use prepared queries)

You do not need the “exact” view

- Consider V1(Product, Customer, Sales) and V2(Product, Customer, Date, Sales) are precomputed
- ProductSales is not precomputed
- You need to answer the query
  ```
  SELECT product, SUM(amt)
  FROM SALES
  GROUP BY product
  ```
- Write it in an alternate way, using one of the views in the most efficient way
Star Schemas

<table>
<thead>
<tr>
<th>prodID</th>
<th>name</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>bolt</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>nut</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>storeID</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>nyc</td>
</tr>
<tr>
<td>c2</td>
<td>sfo</td>
</tr>
<tr>
<td>c3</td>
<td>la</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>orderID</th>
<th>date</th>
<th>custID</th>
<th>prodID</th>
<th>storeID</th>
<th>qty</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>o100</td>
<td>1/7/97</td>
<td>53</td>
<td>p1</td>
<td>c1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>o102</td>
<td>2/7/97</td>
<td>53</td>
<td>p2</td>
<td>c1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>105</td>
<td>3/8/97</td>
<td>111</td>
<td>p1</td>
<td>c3</td>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>custID</th>
<th>name</th>
<th>address</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>joe</td>
<td>10 main</td>
<td>sfo</td>
</tr>
<tr>
<td>81</td>
<td>fred</td>
<td>12 main</td>
<td>sfo</td>
</tr>
<tr>
<td>111</td>
<td>sally</td>
<td>80 willow</td>
<td>la</td>
</tr>
</tbody>
</table>

Star Schema
Terms

- Fact table
- Dimension tables
- Measures

### Dimension Hierarchies

- sType
- store
- city
- region

- Snowflake Schema
- Constellations
**Cube**

**Fact table view**

<table>
<thead>
<tr>
<th>sale</th>
<th>prodId</th>
<th>storeId</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>c1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>p2</td>
<td>c1</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>p1</td>
<td>c3</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>c2</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

**Multi-dimensional cube**

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>12</td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

Dimensions = 2

**3-D Cube**

**Fact table view**

<table>
<thead>
<tr>
<th>sale</th>
<th>prodId</th>
<th>storeId</th>
<th>date</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>c1</td>
<td></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>p2</td>
<td>c1</td>
<td></td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>p1</td>
<td>c3</td>
<td></td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>c2</td>
<td></td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>p1</td>
<td>c1</td>
<td></td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>p1</td>
<td>c2</td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Multi-dimensional cube**

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>44</td>
<td>4</td>
</tr>
<tr>
<td>p1</td>
<td>12</td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

Dimensions = 3
Aggregates on Slices

- Add up amounts for day 1
  - SELECT sum(amt) FROM SALE
  - WHERE date = 1

<table>
<thead>
<tr>
<th>sale</th>
<th>prodId</th>
<th>storeId</th>
<th>date</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>c1</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>p2</td>
<td>c1</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>p1</td>
<td>c3</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>c2</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>p1</td>
<td>c1</td>
<td>2</td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>p1</td>
<td>c2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Aggregates

- Add up amounts by day
  - SELECT date, sum(amt) FROM SALE
  - GROUP BY date

<table>
<thead>
<tr>
<th>sale</th>
<th>prodId</th>
<th>storeId</th>
<th>date</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>c1</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>p2</td>
<td>c1</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>p1</td>
<td>c3</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>c2</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>p1</td>
<td>c1</td>
<td>2</td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>p1</td>
<td>c2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ans</th>
<th>date</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>
Another Example

- Add up amounts by day, product
  - `SELECT date, sum(amt) FROM SALE`
  - `GROUP BY date, prodId`

<table>
<thead>
<tr>
<th>sale</th>
<th>prodId</th>
<th>storeId</th>
<th>date</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>c1</td>
<td>1</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>c1</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>c3</td>
<td>1</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>c2</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>c1</td>
<td>2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>c2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Aggregates

- Operators: sum, count, max, min, median, avg
- “Having” clause
- Using dimension hierarchy
  - average by region (within store)
  - maximum by month (within date)
Cube Aggregation

Date, product, store dimensions

Example: computing sums

Cube Operators

sale(c1,*,*)
sale(c2,p2,*)
sale(*,*,*)
Extended Cube

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>56</td>
<td>4</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td>19</td>
<td>129</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>44</td>
<td>4</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>12</td>
<td>50</td>
<td>62</td>
<td>48</td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>56</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Aggregation Using Hierarchies

Region A

- (customer c1 in Region A; customers c2, c3 in Region B)
What to Materialize?

- Store in warehouse results useful for common queries

Example:

<table>
<thead>
<tr>
<th></th>
<th>day 1</th>
<th>day 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>p1 44</td>
<td>p1 44</td>
</tr>
<tr>
<td>c2</td>
<td>p1 4</td>
<td>p2 4</td>
</tr>
<tr>
<td>c3</td>
<td>p1 50</td>
<td>p2 50</td>
</tr>
<tr>
<td>p1</td>
<td>p1 56</td>
<td>p1 56</td>
</tr>
<tr>
<td>p2</td>
<td>p1 11</td>
<td>p2 11</td>
</tr>
<tr>
<td>p2</td>
<td>p2 8</td>
<td></td>
</tr>
</tbody>
</table>

Cube Aggregates Lattice

Example assumes fact table is sales(city, product, date, amt)
Cube Aggregates Lattice

Example assumes fact table is sales(city, product, amt) and cities c

Should one precompute joins?

- Notice that we have featured foreign keys, not printable values. Why?
- Why (city product) and not (city region product)?
- Minor penalty to find the cities of a particular region
- Probably larger penalty by having a larger table
  - Think space in storage and time to scan it