Data Warehousing

CPS 216
Advanced Database Systems

Review

Data warehousing: integrating data for OLAP
- OLAP versus OLTP
- Warehousing versus mediation
- Warehouse maintenance
  - Warehouse data as materialized views
  - Recomputation versus incremental maintenance
  - Self-maintenance

Today

- Star, snowflake, and cube
- ROLAP and MOLAP algorithms

Star schema

<table>
<thead>
<tr>
<th>Product</th>
<th>Store</th>
<th>Fact table</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>city</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>Durham</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>Chapel Hill</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>RTP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>105</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
</tr>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
</tr>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
</tr>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
</tr>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product Brand</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
</tr>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Brand</th>
</tr>
</thead>
<tbody>
<tr>
<td>brandID</td>
</tr>
<tr>
<td>b1</td>
</tr>
<tr>
<td>b2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product type</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeID</td>
</tr>
<tr>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
</tr>
<tr>
<td>t3</td>
</tr>
<tr>
<td>t4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product category</th>
</tr>
</thead>
<tbody>
<tr>
<td>catID</td>
</tr>
<tr>
<td>cat7</td>
</tr>
<tr>
<td>cat9</td>
</tr>
</tbody>
</table>

Dimension hierarchies

Product ➔ Brand ➔ Product type ➔ Product category

Star join indexes

- Queries frequently join fact table with dimension tables
  - Materialize the join result to speed up queries
- For each combination of dimension attribute values, store the list of tuple ID’s in the fact table
  - Brand name, store city, customer city ➔ sales records;
    Product type, store city ➔ sales records; etc.
  - Conceptually, multi-attribute indexes on the join result
- One index to support each combination of selection conditions on attributes?
  - Too many indexes!
**Bitmap join indexes**

» O’Neil & Quass, SIGMOD 1997

- Bitmap and projection indexes for each dimension attribute
  - Value of the dimension attribute ↔ tuple ID’s in the fact table
- To process an arbitrary combination of selection conditions, use bitmap indexes
  - Bitmaps can be combined efficiently
- To retrieve attribute values for output, use projection indexes

**Data cube**

Completig the cube (slide 1)

Total quantity of sales for each product in each store

```
SELECT SUM(qty) FROM Sale
GROUP BY PID, SID;
```

**Completing the cube (slide 2)**

Total quantity of sales for each product

```
SELECT SUM(qty) FROM Sale GROUP BY PID;
```

**Completing the cube (slide 3)**

Total quantity of sales

```
SELECT SUM(qty) FROM Sale;
```

**CUBE operator**

» Gray et al., ICDE 1996

- Sale(CID, PID, SID, qty)
  - Proposed SQL extension:
    - `SELECT SUM(qty) FROM Sale GROUP BY CUBE CID, PID, SID;
  - Output contains:
    - Normal groups produced by GROUP BY
      - `(c1, p1, s1, sum), (c1, p2, s3, sum), etc.
    - Groups with one or more ALL’s
      - `(ALL, p1, s1, sum), (c2, ALL, ALL, sum), (ALL, ALL, ALL, sum), etc.
  - Can you write a CUBE query using only GROUP BY's?
ROLLUP operator

- Sometimes CUBE is too much
  - (…, state, city, street, …, age, DOB, …)
  - CUBE state, city, street returns meaningless groups
- Proposed SQL extension:
  GROUP BY ROLLUP state, city, street;
- Output contains groups with ALL’s only as suffix
    (‘NC’, ALL, ALL), (ALL, ALL, ALL)
  - But not (ALL, ALL, ‘Main Street’) or (ALL, ‘Durham’, ALL)

Computing GROUP BY

- ROLAP (Relational OLAP)
  - Use standard relational engine
  - Sorting and clustering
  - Using indexes
  - Automatic summary tables
- MOLAP (Multidimensional OLAP)
  - Use a sparse multidimensional array

Sorting and clustering

- Sort (or cluster, e.g., using hashing) tuples according to GROUP BY attributes
  - Tuples in the same group are processed together
  - Only one intermediate aggregate result needs to be kept—low memory requirement
- What if GROUP BY attributes ≠ sort attributes?
  - Still fine if GROUP BY attributes form a prefix of the sort order
  - Otherwise, need to keep intermediate aggregate results around

More on sort order

- Sort by the order in which GROUP BY attributes appear?
  - Not necessary; e.g., GROUP BY PID, SID can be processed just as efficiently by sorting on SID, PID
- Sort by the order in which GROUP BY ROLLUP attributes appear?
  - Useful; e.g., GROUP BY ROLLUP state, city, street can be processed efficiently by sorting on state, city, street, but not by sorting on street, city, state

Using bitmap join indexes

- O’Neil & Quass, SIGMOD 1997
- Use the bitmap join indexes on GB1, GB2, …, GBk
  - For each value v1 of GB1 in order:
    - For each value v2 of GB2 in order: …
      - Intersect bitmaps to locate tuples;
        - Retrieve their measures;
        - Calculate aggregate for group (v1, v2, …, vk);
  - Helps if data is sorted by GB1, GB2, …, GBk
    - So measures in the same group are clustered

Automatic summary tables

- Computing GROUP BY aggregates is expensive
- OLAP queries perform GROUP BY all the time
- Idea: precompute and store the aggregates!
- Automatic summary tables
  - Maintained automatically as base data changes
  - Just another index/materialized view
Aggregation view lattice

GROUP BY CID, PID, SID
GROUP BY CID, SID
GROUP BY PID, SID
GROUP BY CID
GROUP BY PID
GROUP BY SID
GROUP BY ∅

A child can be computed from any parent

Selecting views to materialize

- Factors in deciding what to materialize
  - What is its storage cost?
  - What is its update cost?
  - Which queries can benefit from it?
  - How much can a query benefit from it?
- Example
  - GROUP BY ∅ is small, but not useful to most queries
  - GROUP BY CID, PID, SID is useful to any query, but too large to be beneficial
  » Harinarayan et al., SIGMOD 1996; Gupta & Mumick, ICDE 1999

Interlude: TPC-D, -H, and -R

- TPC-D: standard OLAP benchmark until 1999
  - With aggressive use of precomputation techniques (materialized views, automatic summary tables), vendors were able to “cheat” and achieve amazing performance
- Now, TPC-D has been replaced by
  - TPC-H: ad hoc OLAP queries
    - Cannot use materialized views
  - TPC-R: business-reporting OLAP queries
    - Can use materialized views
  » http://www.tpc.org/

From tables to arrays

- Zhao et al., SIGMOD 1997
- “Chunk” an $n$-dimensional cube into $n$-dimensional subcubes
  - For a dense chunk (>40% full), store it as is
  - For a sparse chunk (<40% full), compress it using $<coordinate, value>$ pairs
- To convert a table into chunks
  - Pass 1: Partition table into memory-size partitions, each of which contains a number of chunks
  - Pass 2: Read partitions back in one at a time, and chunk each partition in memory

Basic array cubing

Memory required:
1 chunk
(could be more)
Minimal spanning tree

- Recall the aggregation lattice
- MST of the lattice: parent is always chosen to be the one with minimum size
- Compute each node from its parent in the MST

Multiway array cubing

- Goal: compute all aggregates at the same time in a single pass over the array, using minimum amount of memory
- Group by $B, C$ requires 1 chunk
- Group by $A, C$ requires 4 chunks
- Group by $A, B$ requires 16 chunks

Memory requirement

- Dimension order is $D_1, D_2, \ldots, D_n$
- Aggregate to compute projects out $D_p$ (i.e., Group by $D_1, \ldots, D_{p-1}, D_{p+1}, \ldots, D_n$)
- The memory required is roughly $|D_1| \cdot |D_1| \cdot \ldots \cdot |D_{p-1}|$ chunks
  - Where $|D_i|$ denotes the number of chunks along $D_i$
  - It is harder to aggregate away dimensions that come later in the dimension order

Minimum-memory spanning tree

- MMST of the aggregation lattice
  - Parent is always chosen to be the one that makes the child require the minimum memory to compute
  - Note that results are produced in dimension order too, so computation of the entire MMST can be pipelined
- Choose an optimal dimension order to minimize the total amount of memory required by MMST
  - It turns out that this optimal order is $D_1, D_2, \ldots, D_n$, where $|D_1| \leq |D_2| \leq \ldots \leq |D_n|

ROLAP versus MOLAP

- Multiway array cubing algorithm (MOLAP) beats sorting-based ROLAP algorithms
  - Compressed array representation is more compact than table representation
  - Sorting-based ROLAP spends too much time on comparing and copying
  - In MOLAP, order is implied by the array positions
- An alternative ROLAP technique
  - Convert table to array
  - Do MOLAP processing
  - Dump the result cube to a table

Next time

Data mining