CSE 544
Principles of Database Management Systems

Fall 2016
Lecture 14 - Data Warehousing and Column Stores
References

• Data Cube: A Relational Aggregation Operator Generalizing Group By, Cross-Tab, and Sub-Totals.
  Jim Gray et. al. Data Mining and Knowledge Discovery 1, 29-53. 1997

• Database management systems.
  Ramakrishnan and Gehrke.
  Third Ed. **Chapter 25**
Why Data Warehouses?

• Production DBMSs designed to manage operational data
  – Goal: support every day activities
  – Online transaction processing (OLTP)
  – Ex: Tracking sales and inventory of each Wal-mart store

• Data Warehouse designed to analyze and explore data
  – Goal: summarize and discover trends to support decision making
  – Online analytical processing (OLAP)

• Data warehouse usually updated overnight from production databases
The Origin of Data Warehouses

Amazon, 00s

Sales DB

Operational DB

Nightly Backups

Sale transactions

Users
Data Warehouse Overview

• Consolidated data from many sources
  – Must create a single unified schema
  – The warehouse is like a materialized view

• Very large size: terabytes of data are common

• Complex read-only queries (no updates)

• Fast response time is (not as) important
  – Compared to transaction processing
Star Schema

• Central table, e.g.
  – SALES(saleID, time, price, storeID, productID, ...)

• Dimension tables, e.g.
  – Store(storeID, sname, location, ...),
  – Product(productID, pname, weight, ...)
  – SalesPerson(personID, name, ...)
  – ...
OLAP queries

- ETL pipeline load data into a data warehouse

- Operators:
  - Rollup
  - Drill down
  - Pivoting
  - Cube
The ETL Pipeline

- **Extract** data from distributed operational databases
- **Clean** to minimize errors and fill in missing information
- **Transform** to reconcile semantic mismatches
  - Performed by defining views over the data sources
- **Load** to materialize the above defined views
  - Build indexes and additional materialized views
- **Refresh** to propagate updates to warehouse periodically
Back to Warehouses: Outline

- Multidimensional data model and operations
- Data cube & rollup operators
- Data warehouse implementation issues
- Other extensions for data analysis
Multidimensional Data Model

- Focus of the analysis is a collection of measures
  - Example: Wal-mart sales

- Each measure depends on a set of dimensions
  - Example: product (pid), location (lid), and time of the sale (timeid)

**Slicing**: equality selection on one or more dimensions

**Dicing**: range selection
Star Schema

Representing multidimensional data as relations (ROLAP)

Facts table: Sales
In BCNF

Dimensions tables
- Product
- Location
- Times
Not necessarily normalized

Product
- pid
- pname
- category
- price

Location
- locid
- city
- state
- country

Sales
- sid
- pid
- timeid
- locid
- amount

Times
- timeid
- date
- week
- month
- quarter
- year
Dimension Hierarchies

Dimension values can form a hierarchy described by attributes

- **Product**
  - category
  - pname

- **Time**
  - year
  - quarter
  - week
  - month
  - date

- **Location**
  - country
  - state
  - city
Desired Operations

- Histograms (agg. over computed categories) (paper p.34)

- Summarize at different levels: roll-up and drill-down
  - Ex: total sales by day, week, quarter, and year

- Pivoting
  - Ex: pivot on location and time
  - Result of pivot is a cross-tabulation
  - Column values become labels

<table>
<thead>
<tr>
<th></th>
<th>WI</th>
<th>CA</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>500</td>
<td>200</td>
<td>700</td>
</tr>
<tr>
<td>2006</td>
<td>150</td>
<td>850</td>
<td>1000</td>
</tr>
<tr>
<td>2007</td>
<td>250</td>
<td>400</td>
<td>650</td>
</tr>
<tr>
<td>Total</td>
<td>900</td>
<td>1450</td>
<td>2350</td>
</tr>
</tbody>
</table>
Challenge 1: Representation

• Problem: How to represent multi-level aggregation?
  – Ex: Table 3 in the paper need $2^N$ columns for N dimensions!
  – Ex: Table 4 has even more columns!
The representation suggested by Table 5 and unioned GROUP BYs “solve” the problem of representing aggregate data in a relational data model. The problem remains that expressing roll-up, and cross-tab queries with conventional SQL is daunting. A six dimension cross-tab requires a 64-way union of 64 different GROUP BY operators to build the underlying representation.

There is another very important reason why it is inadequate to use GROUP BYs. The resulting representation of aggregation is too complex to analyze for optimization. On most SQL systems this will result in 64 scans of the data, 64 sorts or hashes, and a long wait.

3. CUBE and ROLLUP operators

The generalization of group by, roll-up and cross-tab ideas seems obvious: Figure 3 shows the concept for aggregation up to 3-dimensions. The traditional GROUP BY generates the \( N \)-dimensional data cube core. The \( N \) lower-dimensional aggregates appear as points, lines, planes, cubes, or hyper-cubes hanging off the data cube core.

The data cube operator builds a table containing all these aggregate values. The total aggregate using function \( f() \) is represented as the tuple:

\[
\text{ALL, ALL, ALL, } (\ldots, \text{ALL, ALL, } f(\ast))
\]

Points in higher dimensional planes or cubes have fewer ALL values.

Figure 3. The CUBE operator is the \( N \)-dimensional generalization of simple aggregate functions. The 0D data cube is a point. The 1D data cube is a line with a point. The 2D data cube is a cross tabulation, a plane, two lines, and a point. The 3D data cube is a cube with three intersecting 2D cross tabs.
Creating a data cube requires generating the power set (set of all subsets) of the aggregation columns. Since the \textit{CUBE} is an aggregation operation, it makes sense to externalize it by overloading the SQL \textit{GROUP BY} operator. In fact, the cube is a relational operator, with \textit{GROUP BY} and \textit{ROLL UP} as degenerate forms of the operator. This can be conveniently specified by overloading the SQL \textit{GROUP BY}.

Figure 4 has an example of the cube syntax. To give another, here follows a statement to aggregate the set of temperature observations:

\begin{verbatim}
SELECT day, nation, MAX(Temp) FROM Weather GROUP BY CUBE Day(Time) AS day, Country(Latitude, Longitude) AS nation;
\end{verbatim}

The semantics of the \textit{CUBE} operator are that it first aggregates over all the \textit{SELECT list} attributes in the \textit{GROUP BY} clause as in a standard \textit{GROUP BY}. Then, it \textit{UNIONs} in each super-aggregate of the global cube—substituting \textit{ALL} for the aggregation columns. If there are \( N \) attributes in the \textit{SELECT list}, there will be \( 2^{N-1} \) super-aggregate values. If the cardinality of the \( N \) attributes are \( C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_N \), then the cardinality of the...
Challenge 1: Representation

• Problem: How to represent multi-level aggregation?
  – Ex: Table 3 in the paper need $2^N$ columns for N dimensions!
  – Ex: Table 4 has even more columns!
  – And that’s without considering any hierarchy on the dimensions!

• Solution: special “all” value

<table>
<thead>
<tr>
<th>T.year</th>
<th>L.state</th>
<th>SUM(S.sales)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>WI</td>
<td>500</td>
</tr>
<tr>
<td>2005</td>
<td>CA</td>
<td>200</td>
</tr>
<tr>
<td>2005</td>
<td>ALL</td>
<td>700</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>2350</td>
</tr>
</tbody>
</table>

Note: SQL-1999 standard uses NULL values instead of ALL
Challenge 2: Computing Aggregations

• Need \(2^N\) different SQL queries to compute all aggregates
  – Expressing roll-up of a single column and cross-table queries is thus daunting
  – Cannot optimize all these independent queries

• Solution: CUBE and ROLLUP operators
Outline

• Multidimensional data model and operations

• Data cube & rollup operators

• Data warehouse implementation issues

• Other extensions for data analysis
Data Cube

• CUBE is the N-dimensional generalization of aggregate

• Cube in SQL-1999

SELECT T.year, L.state, SUM(S.sales)
FROM Sales S, Times T, Locations L
WHERE S.timeid=T.timeid and S.locid=L.locid
GROUP BY CUBE (T.year,L.state)

• Creating a data cube requires generating the power set of the aggregation columns
Rollup

- Rollup produces a subset of a cube

- Rollup in SQL-1999

  ```sql
  SELECT T.year, T.quarter, SUM(S.sales)
  FROM Sales S, Times T
  WHERE S.timeid = T.timeid
  GROUP BY ROLLUP (T.year, T.quarter)
  ```

- Will aggregate over each pair of (year, quarter), each year, and total, but will **not** aggregate over each quarter
Computing Cubes and Rollups

• Naive algorithm
  – For each new tuple, update each of $2^N$ matching cells

• More efficient algorithm
  – Use intermediate aggregates to compute others
  – Relatively easy for distributive and algebraic functions

• Updating a cube in response to updates is more challenging
Outline

• Multidimensional data model and operations
• Data cube & rollup operators
• Data warehouse implementation issues
• Other extensions for data analysis
Indexes

- **Bitmap indexes**: good for sparse attributes (few values)
  - Example: Join fact table F with dimension tables D1 and D2
  - Index contain triples of rids $<r_1, r_2, r>$ from D1, D2, and F that join
  - Alternatively, two indexes, each one with pairs $<v_1, r>$ or $<v_2, r>$ where $v_1, v_2$ are values of tuples from D1, D2 that join with r

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>custid</th>
<th>name</th>
<th>gender</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>Alice</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>Bob</td>
<td>M</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12</td>
<td>Chuck</td>
<td>M</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Materialized Views

• How to choose views to materialize?
  – Physical database tuning

• How to keep view up-to-date?
  – Could recompute entire view for each update: expensive
  – Better approach: incremental view maintenance
  – Example: recompute only affected partition

  – How often to synchronize? Periodic updates (at night) are typical
    • Think back in the case of Walmart
Outline

• Multidimensional data model and operations

• Data cube & rollup operators

• Data warehouse implementation issues

• Other extensions for data analysis
Additional Extensions for Decision Support

• Window queries

SELECT L.state, T.month, AVG(S.sales) over W AS movavg
FROM Sales S, Times T, Locations L
WHERE S.timeid = T.timeid AND S.locid = L.locid
WINDOW W AS (PARTITION BY L.State
ORDER BY T.month
RANGE BETWEEN INTERVAL ‘1’ MONTH PRECEDING
AND INTERVAL ‘1’ MONTH FOLLOWING)

• Top-k queries: optimize queries to return top k results

• Online aggregation: produce results incrementally
Leveraging Column Stores
References

Column-Oriented Databases

• Main idea:
  – Physical storage: complete vertical partition; each column stored separately: R.A, R.B, R.A
  – Logical schema: remains the same R(A,B,C)

• Main advantage:
  – Improved transfer rate: disk to memory, memory to CPU, better cache locality
  – Other advantages (next)
Basic tradeoffs:

- Reading all attributes of one records, v.s.
- Reading some attributes of many records
Key Architectural Trends (Sec. 1)

- Virtual IDs
- Block-oriented and vertical processing
- Late materialization
- Column-specific compression
Key Architectural Trends (Sec. 1)

- Virtual IDs
  - Offsets (arrays) instead of keys
- Block-oriented and vertical processing
  - Iterator model: one tuple → one block of tuples
- Late materialization
  - Postpone tuple reconstruction in query plan
- Column-specific compression
  - Much better than row-compression (why?)
Figure 1.2: Performance of C-Store versus a commercial database system on the SSBM benchmark, with different column-oriented optimizations enabled.
Vectorized Processing

Review:
- Volcano-style iterator model
  - Next() method
  - Pipelining
- Materialization of all intermediate results
- Discuss in class:

```sql
select avg(A) from R where A < 100
```
Vectorized Processing

• Vectorized processing:
  – Next() returns a block of tuples (e.g. N=1000) instead of single tuple

• Pros:
  – No more large intermediate results
  – Tight inner loop for selection and/or avg

• Discuss in class:

```
select avg(A) from R where A < 100
```
Compression (Sec. 4)

• What is the advantage of compression in databases?

• Discuss main column-at-a-time compression techniques
Compression (Sec. 4)

• What is the advantage of compression in databases?

• Discuss main column-at-a-time compression techniques
  – Bit-vector (see also bit-map indexes)
Compression (Sec. 4)

Row-based (4 pages)

<table>
<thead>
<tr>
<th>Page</th>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

Column-based (4 pages)

<table>
<thead>
<tr>
<th>Page</th>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

Compressed (2 pages)

<table>
<thead>
<tr>
<th>Page</th>
<th>4XA</th>
<th>1X1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2XB</td>
<td>4X1</td>
</tr>
<tr>
<td></td>
<td>2XC</td>
<td>4X2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5X4</td>
</tr>
</tbody>
</table>

CSE 544 - Fall 2016
Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A='a' \land D='d'}(R(A,B,C,D,...)))$
Late Materialization (Sec. 4)

- **What is it?**
- Discuss $\Pi_B(\sigma_{A='a' \land D='d'}(R(A,B,C,D,...)))$

**Early materialization:**
- Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
- Retain only positions with ‘d’: 4, 9, …

**Late materialization**
- Retrieve positions with ‘a’ in column A: 2, 4, 5, 9, 25…
- Retrieve positions with ‘d’ in column D: 3, 4, 7, 9, 12, …
- Intersect: 4, 9, …
**Late Materialization (Sec. 4)**

**Ex:** \( \text{SELECT } R.b \text{ from } R \text{ where } R.a = X \text{ and } R.d = Y \)

Early materialization

![Early Materialization Diagram]

Late materialization

![Late Materialization Diagram]
The result of a join $R.A \bowtie S.A$ is an array of positions in $R.A$ and $S.A$. Note: sorted on $R.A$ only.

<table>
<thead>
<tr>
<th>R.A</th>
<th>S.A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Value42</td>
</tr>
<tr>
<td>2</td>
<td>Value36</td>
</tr>
<tr>
<td>3</td>
<td>Value42</td>
</tr>
<tr>
<td>4</td>
<td>Value44</td>
</tr>
<tr>
<td>5</td>
<td>Value38</td>
</tr>
<tr>
<td>1</td>
<td>Value38</td>
</tr>
<tr>
<td>2</td>
<td>Value42</td>
</tr>
<tr>
<td>3</td>
<td>Value46</td>
</tr>
<tr>
<td>4</td>
<td>Value36</td>
</tr>
</tbody>
</table>

Positions in $R.A$ (sorted):
- 1
- 2
- 3
- 4

Positions in $S.A$ (unsorted):
- 1
- 2
- 3
- 4
Problem: accessing the values in the second table has poor memory locality
Solution: re-sort by the second column, fetch, sort back
E.g. $\Pi_{S.C}(R(A,\ldots)) \bowtie S(B,C,\ldots)$
Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality
Solution: re-sort by the second column, fetch, sort back

E.g. \( \Pi_{S.C}(R(A, \ldots) \bowtie S(B, C, \ldots)) \)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Sort on positions in S.B

Lookup S.C (this is a merge-join; why?)

1 | Smith
2 | Johnson
3 | Williams
4 | Jones
Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality
Solution: re-sort by the second coljun, fetch, sort back
E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

=  

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

\[ \bowtie \]

Sort on positions in S.B

Lookup S.C (this is a merge-join; why?)
Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality
Solution: re-sort by the second coljun, fetch, sort back
E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$

1 1 2 4 5
2 2 4 1 1
3 3 2 3 3
4 5 1 2 2

= 1 1 1 2 2
  3 3 2 3 3
  4 4 4 4 4

Smith 4 5
Johnson 1 1
Williams 3 3
Jones 2 2

Sort on positions in $S.B$

Lookup $S.C$ (this is a merge-join; why?)

Re-sort on positions in $R.A$

= 1 1
  2 2
  4 5
  1 1
  2 2
  4 4
  1 1
  2 2
  4 4

1 1
2 2
3 3
4 4

1 Smith
2 Johnson
3 Johnson
4 Jones
select sum(R.a) from R, S
where R.c = S.b
  and 5<R.a<20 and 40<R.b<50
  and 30<S.a<40
Late Materialization

```
select sum(R.a) from R, S 
where R.c = S.b 
    and 5 < R.a < 20 and 40 < R.b < 50 
    and 30 < S.a < 40
```
Late Materialization

```sql
select sum(R.a) from R, S
where R.c = S.b
and 5 < R.a < 20 and 40 < R.b < 50
and 30 < S.a < 40
```
Late Materialization

\[
\text{select sum}(R.a) \text{ from } R, S \\
\text{where } R.c = S.b \\
\text{and } 5 < R.a < 20 \text{ and } 40 < R.b < 50 \\
\text{and } 30 < S.a < 40
\]
Simulating a Column-Store DBMS in a Row-Store DBMS

- **Vertical partitioning**
  - Two-column tables: (key, attribute)

- **Index-only plans**
  - Create a B+ tree index on each attribute
  - Answer queries using indexes only, without reading actual data

- **Materialized views**
  - Each view contains a subset of columns
Conclusion

• **Column-store DBMS outperforms row-store DBMS**
  – Measured on a data warehousing benchmark (SSBM)

• **Late materialization and compression are key factors**

• **Difficult to simulate a column-store in a row-store**
  – Tuple overheads cause data blow-up
  – Column joins are expensive
  – Hard to get the DBMS to “do the right thing” (e.g., index plans)

• **Not the end of the story, however, … see CIDR’09 paper**
ABSTRACT

In recent years, column stores (or C-stores for short) have emerged as a novel approach to deal with read-mostly data warehousing applications. Experimental evidence suggests that, for certain types of queries, the new features of C-stores result in orders of magnitude improvement over traditional relational engines. At the same time, some C-store proponents argue that C-stores are fundamentally different from traditional engines, and therefore their benefits cannot be incorporated into a relational engine short of a complete rewrite. In this paper we challenge this claim and show that many of the benefits of C-stores can indeed be simulated in traditional engines with no changes whatsoever. We then identify some limitations of our “pure-simulation” approach for the case of more complex queries. Finally, we predict that traditional relational engines will eventually leverage most of the benefits of C-stores natively, as is currently happening in other domains such as XML data.