CompSci 516
Database Systems

Lecture 21
Data Warehousing and Data Cube

Instructor: Sudeepa Roy

Reading Material

- [RG] — Chapter 25
- Gray-Chaudhuri-Genest-McLaughlin-Verdieros-Pirahesh, ICDE 1994 "Data Cube: A 
  Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals"
- Harinarayan-Rajaraman-Ullman, SIGMOD 1996 "Implementing data cubes efficiently"

Acknowledgement:
- The following slides have been created adapting the instructor material of the [RG] book provided by 
  the authors Dr. Ramakrishnan and Dr. Gehrke.
- Some slides have been prepared by Prof. Shivnath Babu

Data Warehousing

- Growing industry: $8 billion way back in 1998
- Data warehouse vendor like Teradata
  - big “Petabyte scale” customers
  - Apple, Walmart (2008-2.5PB), eBay (2013-primary DW 9.2 PB, other big data 40PB, single table with 1 trillion rows), 
    Verizon, AT&T, Bank of America
  - supports data into and out of Hadoop
- Lots of buzzwords, hype
  - slice & dice, rollup, MOLAP, pivot, ...

Motivating Examples

- Forecasting
- Comparing performance of units
- Monitoring, detecting fraud
- Visualization

Introduction

- Organizations analyze current and historical data 
  - to identify useful patterns
  - to support business strategies
- Emphasis is on complex, interactive, exploratory analysis of very large datasets
- Created by integrating data from across all parts of an enterprise
- Data is fairly static
- Relevant once again for the recent "Big Data analysis"
  - to figure out what we can reuse, what we cannot


Ack: Slide by Prof. Shivnath Babu
Three Complementary Trends

- **Data Warehousing (DW):**
  - Consolidate data from many sources in one large repository
  - Loading, periodic synchronization of replicas
  - Semantic integration

- **OLAP:**
  - Complex SQL queries and views.
  - Queries based on spreadsheet-style operations and "multidimensional" view of data.
  - Interactive and "online" queries.

- **Data Mining:**
  - Exploratory search for interesting trends and anomalies
  - Next lecture!

Data Warehousing

- A collection of decision support technologies
- To enable people in industry/organizations to make better decisions
  - Supports OLAP (On-Line Analytical Processing)
- Applications in
  - Manufacturing
  - Retail
  - Finance
  - Transportation
  - Healthcare
  - ...
- Typically maintained separately from "Operational Databases"
  - Operational Databases support OLTP (On-Line Transaction Processing)

Why a Warehouse?

- **Two Approaches:**
  - Query-Driven (Lazy)
  - Warehouse (Eager)

Advantages of Warehousing

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

Query-Driven Approach

Advantages of Query-Driven

- No need to copy data
  - less storage
  - no need to purchase data
- More up-to-date data
- Query needs can be unknown
- Only query interface needed at sources
- May be less draining on sources
OLTP | Data Warehousing/OLAP
--- | ---
Mostly updates | Mostly reads
Applications: Order entry, sales update, banking transactions | Applications: Decision support in industry/organization
Detailed, up-to-date data | Summarized, historical data (from multiple operational db, grows over time)
Structured, repetitive, short tasks | Query intensive, ad hoc, complex queries
Each transaction reads/updates only a few tuples (tens of) | Each query can access many records, and perform many joins, scans, aggregates
MB-GB data | GB-TB data
Typically clerical users | Decision makers, analysts as users
Important: Consistency, recoverability, Maximizing tr. throughput | Important: Query throughput Response times

Data Marts
- smaller datawarehouse
- subsets of data on selected subjects
- e.g. Marketing data mart can include customer, product, sales
- Department-focused, no enterprise-wide consensus needed
- But may lead to complex integration problems in the long run

ROLAP and MOLAP
- Relational OLAP (ROLAP)
  - On top of standard relational DBMS
  - Data is stored in relational DBMS
  - Supports extensions to SQL to access multi-dimensional data
- Multidimensional OLAP (MOLAP)
  - Directly stores multidimensional data in special data structures (e.g. arrays)

Data Warehousing to Mining
- Integrated data spanning long time periods, often augmented with summary information
- Several gigabytes to terabytes common
- Interactive response times expected for complex queries; ad-hoc updates uncommon

Warehousing Issues
- Semantic Integration: When getting data from multiple sources, must eliminate mismatches
  - e.g., different currencies, schemas
- Heterogeneous Sources: Must access data from a variety of source formats and repositories
  - Replication capabilities can be exploited here
- Load, Refresh, Purge: Must load data, periodically refresh it, and purge too-old data
- Metadata Management: Must keep track of source, loading time, and other information for all data in the warehouse

DW Architecture
- Extract data from multiple operational DB and external sources
- Clean/integrate/transform/store
- Refresh periodically
  - update base and derived data
  - admin decides when and how
- Main DW and several data marts (possibly)
- Managed by one or more servers and front end tools
- Additional meta data and monitoring/admin tools
ROLAP: Star Schema

- To reflect multi-dimensional views of data
- Single fact table
- Single table for every dimension
- Each tuple in the fact table consists of:
  - pointers (foreign key) to each of the dimensions (multi-dimensional coordinates)
  - numeric value for those coordinates
- Each dimension table contains attributes of that dimension

Dimension Hierarchies

- For each dimension, the set of values can be organized in a hierarchy:

ROLAP: Snowflake Schema

- Refines star-schema
- Dimensional hierarchy is explicitly represented
- (+) Dimension tables easier to maintain
  - suppose the category description is being changed
- (-) Need additional joins
- Fact Constellations
  - Multiple fact tables share some dimensional tables
  - e.g. Projected and Actual Expenses may share many dimensions

Motivation: OLAP Queries

- Data analysts are interested in OLAP Queries
  - Possibly by visualization (Excel) - 2D or 3D plots
  - "Dimensionality Reduction" by summarizing data and computing aggregates
  - Influenced by SQL and by spreadsheets.
  - A common operation is to aggregate a measure over one or more dimensions.

- Find total unit sales for each
  1. Model
  2. Model, broken into years
  3. Year, broken into colors
  4. Year
  5. Model, broken into color, ….

OLAP and Data Cube

Histograms

A tabulated frequency of computed values

```
SELECT Year, COUNT(Units) as total
FROM Sales
GROUP BY Year
ORDER BY Year
```

May require a nested SELECT to compute
**Roll-Ups**

• Analysis reports start at a coarse level, go to finer levels
• Order of attribute matters
• Not relational data (empty cells no keys)

---

**Sales (Model, Year, Color, Units)**

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chevy</td>
<td>1994</td>
<td>Black</td>
<td>50</td>
</tr>
<tr>
<td>Chevy</td>
<td>1994</td>
<td>White</td>
<td>40</td>
</tr>
<tr>
<td>Chevy</td>
<td>1995</td>
<td>Black</td>
<td>115</td>
</tr>
<tr>
<td>Chevy</td>
<td>1995</td>
<td>White</td>
<td>85</td>
</tr>
</tbody>
</table>

---

**Roll-Ups**

• Another representation (Chris Date’96)
• Relational, but
  – long attribute names
  – hard to express in SQL and repetition

---

**‘ALL’ Construct**

Easier to visualize roll-up if allow ALL to fill in the super-aggregates

---

**Sales (Model, Year, Color, Units)**

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chevy</td>
<td>1994</td>
<td>Black</td>
<td>50</td>
</tr>
<tr>
<td>Chevy</td>
<td>1994</td>
<td>White</td>
<td>40</td>
</tr>
<tr>
<td>Chevy</td>
<td>1995</td>
<td>Black</td>
<td>115</td>
</tr>
<tr>
<td>Chevy</td>
<td>1995</td>
<td>White</td>
<td>85</td>
</tr>
</tbody>
</table>

---

**Naïve Approach**

• Data cube generalizes Histogram, Roll-Ups, Cross-Tab
• More complex to do these with GROUP-BY

---

**Naïve Approach**

• Run a number of queries
  
  ```
  SELECT Model, Year, Color, SUM(Units) 
  FROM Sales 
  WHERE Model = 'Chevy' 
  GROUP BY Model, Year, Color 
  UNION 
  SELECT Model, Year, 'ALL', SUM(Units) 
  FROM Sales 
  WHERE Model = 'Chevy' 
  GROUP BY Model, Year 
  UNION 
  SELECT 'ALL', 'ALL', 'ALL', SUM(Units) 
  FROM Sales 
  WHERE Model = 'Chevy';
  ```

---

**Cross Tabulation**

• If we made the roll-up symmetric, we would get a cross-tabulation
• Generalizes to higher dimensions

```sql
SELECT Model, Year, Color, SUM(Units) 
FROM Sales 
WHERE Model = 'Chevy' 
GROUP BY Model, Year, Color 
UNION 
SELECT Model, Year, 'ALL', SUM(Units) 
FROM Sales 
WHERE Model = 'Chevy' 
GROUP BY Model, Year 
UNION 
SELECT 'ALL', 'ALL', 'ALL', SUM(Units) 
FROM Sales 
WHERE Model = 'Chevy';
```

---

**Naïve Approach**

Is the problem solved with Cross Tab and GROUP-BYs with ‘ALL’?
• Requires a lot of GROUP-BYs (64 for 6-dimension)
• Too complex to optimize (64 scans, 64 sort/hash, slow)
### Data Cube: Intuition

```sql
SELECT 'ALL', 'ALL', 'ALL', sum(units)
FROM Sales
UNION
SELECT 'ALL', 'ALL', Color, sum(units)
FROM Sales
GROUP BY Color
UNION
SELECT 'ALL', Year, 'ALL', sum(units)
FROM Sales
GROUP BY Year
UNION
SELECT Model, Year, 'ALL', sum(units)
FROM Sales
GROUP BY Model, Year
UNION
```

### Data Cube

- Computes the aggregate on all possible combinations of group by columns.
- If there are N attributes, there are $2^N - 1$ super-aggregates.
- If the cardinality of the N attributes are $C_1, ..., C_N$, then there are a total of $(C_1+1)...(C_N+1)$ values in the cube.
- ROLL-UP is similar but just looks at N aggregates

### Types of Aggregates

- **Distributive**: input can be partitioned into disjoint sets and aggregated separately
  - COUNT, SUM, MIN
- **Algebraic**: can be composed of distributive aggregates
  - AVG
- **Holistic**: aggregate must be computed over the entire input set
  - MEDIAN
- Efficient computation of the CUBE operator depends on the type of aggregate
  - Distributive and Algebraic aggregates motivate optimizations

### Data Cube Syntax

- SQL Server

```sql
SELECT Model, Year, Color, sum(units)
FROM Sales
GROUP BY Model, Year, Color
WITH CUBE
```

### View Materialization and Maintenance

[RG] Chapters 25.8-25.10
Views (revisiting)

• Motivation (example)
  – Different groups of analysts within an organization are typically concerned with different aspects of a business
  – It is convenient to define “views” that give each group insight into the relevant business details
  – Other views can be defined or queries can be written using these views
  – Convenient and Efficient

Views and OLAP/Warehousing

• OLAP queries are typically aggregate queries
  – Precomputation is essential for interactive response times
  – The CUBE is in fact a collection of aggregate queries, and precomputation is especially important
  – Lots of work on what is best to precompute given a limited amount of space to store precomputed results.

• Warehouses can be thought of as a collection of asynchronously replicated tables and periodically maintained views
  – Factors: size, number of tables involved, many are from external independent databases
  – Has renewed interest in (asynchronous) view maintenance (more later)

Index on Materialized Views: Examples

CREATE VIEW RegionalSales AS
  SELECT R.category, R.state, SUM(R.sales) AS sales
  FROM Products P, Sales S, Locations L
  WHERE P.pid=S.pid AND S.locid=L.locid
GROUP BY R.category, R.state

SELECT R.category, R.state, SUM(R.sales) AS sales
  FROM RegionalSales R
  WHERE R.category="Laptop"
GROUP BY R.category

Index on precomputed view is great!

SELECT R.asu, SUM(R.sales) AS sales
  FROM RegionalSales R
  WHERE R.state="Wisconsin"
GROUP BY R.asu

Index is less useful (must scan entire leaf level)

(Research) Issues in View Materialization

1. What views should we materialize, and what indexes should we build on the precomputed results?

2. Given a query and a set of materialized views, can we use the materialized views to answer the query?
   – Related to the first question (workload dependent)
   – Try to materialize a small, carefully chosen set of views that can be utilized to quickly answer most of the important queries

3. How frequently should we refresh materialized views to make them consistent with the underlying tables?
   – And how can we do this incrementally?
View Maintenance

- Two steps:
  - Propagate: Compute changes to view when data changes
  - Refresh: Apply changes to the materialized view table

- Maintenance policy: Controls when we do refresh
  - Immediate: As part of the transaction that modifies the underlying data tables
    - Materialized view is always consistent
    - Updates are slowed
  - Deferred: Some time later, in a separate transaction
    - View becomes inconsistent
    - Can scale to maintain many views without slowing updates

Types of Deferred Maintenance

Three flavors:

- Lazy:
  - Delay refresh until next query on view; then refresh before answering the query (slows down queries than updates)

- Periodic (Snapshot):
  - Refresh periodically (e.g., once in a day). Queries possibly answered using outdated version of view tuples. Widely used, especially for asynchronous replication in distributed databases, and for warehouse applications
  - E.g., Refresh after a fixed number of updates to underlying data tables
    - E.g. Snapshot in Oracle 7
      - Periodically refreshed by entirely recomputing the view
      - Incremental "fast refresh" or "simple snapshots" for simpler views (no aggregate, group by, join, distinct etc.)

- Event-based or Forced:
  - E.g., Refresh after a fixed number of updates to underlying data tables
    - E.g. Snapshot in Oracle 7
      - Periodically refreshed by entirely recomputing the view
      - Incremental "fast refresh" or "simple snapshots" for simpler views (no aggregate, group by, join, distinct etc.)

Basic Ideas

- Need to compute all group-by-s:
  - ABCD, ABC, ABD, BCD, AB, AC, AD, BC, BD, CD, A, B, C, D

- Compute GROUP-BYs from previously computed GROUP-BYs
  - E.g., first ABCD
    - Then ABC or ACD
    - Then AB or AC ...

- Which order ABCD is sorted, matters for subsequent computations
  - If (ABCD) is the sorted order, ABC is cheap, ACD or BCD is expensive

Notations

- ABCD
  - Group-by on attributes A, B, C, D
  - No guarantee on the order of tuples

- (ABCD)
  - Sorted according to A -> B -> C -> D

- ABCD and (ABCD) and (BCDA)
  - All contain the same results
  - But in different sorted order

Optimization 1: Smallest Parent

- Compute GROUP-BY from the smallest (size) previously computed GROUP-BY as a parent
  - AB can be computed from
    - ABC, ABD, or ABCD
  - ABC or ABD better than ABCD
  - Even ABC or ABD may have different sizes, try to choose the smaller parent

LATTICE STRUCTURE of data cube.
### Optimization 2: Cache Results

- Cache result of one GROUP-BY in memory to reduce disk I/O
  - Compute AB from ABC while ABC is still in memory

![Diagram](https://example.com/diagram2.png)

- Optimization 3: Amortize Disk Reads
  - Amortize disk reads for multiple GROUP-BYS
    - Suppose the result for ABCD is stored on disk
    - Compute all of ABC, ABD, ACD, BCD simultaneously in one scan of ABCD

![Diagram](https://example.com/diagram3.png)

### Optimization 4, 5 (next)

- **4. Share-sort**
  - for sort-based algorithms
  - pipe-sort algorithm
  - covered in class

- **5. Shared-partition**
  - for hash-based algorithms
  - pipe-hash algorithm
    - Uses hash tables to compute smaller GROUP-BYS
    - If the hash tables for AB and AC fit in memory, compute both in one scan of ABCD
    - Otherwise can partition on A, and can compute HTs of AB and AC in different partitions
  - not covered (see paper)

![Diagram](https://example.com/diagram4.png)

### PipeSort: Idea

- Combines two optimizations: “shared-sorts” and “smallest-parent”

- Also includes “cache-results” and “amortized-scans”

### PipeSort: Share-sort optimization

- Data sorted in one order
- Compute all GROUP-BYS prefixed in that order
- Compute one tuple of ABCD, propagate upward in the pipeline by a single scan

- Example:
  - GROUP-BY over attributes ABCD
  - Sort raw data by (ABCD)
  - Compute (ABCD) -> (ABC) -> (AB) -> (A) in pipelined fashion
  - No additional sort needed

- BUT, may have a conflict with “smallest-parent” optimization
  - (ABD) -> (A) could be a better choice
  - Figure out the best parent choice by running a weighted-matching algorithm layer by layer

### Search Lattice

- Directed edge => one attribute less and possible computation
  - Level k contains k attributes
  - Two possible costs for each edge e:
    - (BOA) -> (BC)
    - e.g. ABCD > (BOA) > (BC) or hash
  - S(A) is NOT sorted for j
  - Sorted Not Sorted

![Diagram](https://example.com/diagram5.png)
PipeSort Output

- Outputs a subgraph $O$
  - each node has a single parent
  - each node has a sorted order of attributes
- If parent’s sorted order is a prefix, cost = $A(e_{ij})$, else $S(e_{ij})$
  - Mark by $A$ or $S$
  - At most one $A$-out-edge
  - Note: for some nodes, there may be no green $A$-out-edge

Goal: Find $O$ with min total cost

Outline: PipeSort Algorithm (1)

- Go from level 0 to $N-1$
  - here $N = 4$
- For each level $k$, find the best way to construct it from level $k+1$
- uses “min-cost weighted bipartite matching”
- creates $k$ new copies of nodes at level $k+1$
  - edges from original copy
    - cost $A(e_{ij})$
  - edges from new copies
    - cost $S(e_{ij})$

Outline: PipeSort Algorithm (2)

- Illustration with a smaller example
- Level $k = 1$ from level $k+1 = 2$
  - one new copy (dotted edges)
  - one existing copy (solid edge)
- Assumption for simplicity
  - same cost for all outgoing edges
  - $A(e_{ij}) = A(e_{ij})$ for all $i, j$
  - $S(e_{ij}) = S(e_{ij})$ for all $i, j$