Where We Are

• Data models
  – Relational
  – IMS / Codasyl
  – Unstructured

• Query processing
  – Algorithms for relational operators
  – Indexing and physical design

• Queries that real-world users write
  – Data warehousing
  – Transaction processing
Where We Are

• What queries do real people write?
  – Data warehousing
    • Column stores (today)
    • Parallel databases (Thursday)
    • Programming models (next week)
  – Transaction processing
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?

• **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

• **Enterprises also need to analyze and explore their data**
  – Goal: summarize and discover trends to support decision making
  – Online analytical processing (OLAP)

• **To support OLAP consolidate all data into a warehouse**
The Origin of Data Warehouses

Walmart, 90s

Sale transactions

Nightly Backups

Sales DB

Cashiers
The Origin of Data Warehouses

Amazon, 00s

Operational DB

Sales transactions

Nightly Backups

Sales DB

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
Creating a Data Warehouse

• **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

• This is known as the **ETL pipeline**
Alternative: Distributed DBMS

• User submits a query at one site

• Query is defined over data located at different sites
  – Different physical locations
  – Different types of DBMSs

• Query optimizer finds the best distributed query plan
  – Query executes across all the locations
  – Results shipped to query site and returned to user

• (Stay tuned for the next lecture!)
Distributed DBMS Limitations

• Top-down
  – Global, \textit{a priori} data placement
  – Global query optimization
    • One query at a time; no notion of load balance
  – Distributed transactions, tight coupling
• Assumes full \textit{cooperation} of all sites
• Assumes \textit{uniform} sites
• Assumes \textit{short-duration} operations

• Limited scalability
Back to Warehouses: Outline

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

• Formulate query that extracts data from the database
  – Typically ad-hoc complex query with group by and aggregate

• Visualize the data (e.g., spreadsheet)
  – Dataset is an N-dimensional space

• Analyze the data
  – Identify “interesting” subspace by aggregating other dimensions
  – Categorize the data and compare categories with each other
  – Roll-up and drill-down on the data
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
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)
  – Problem: awkward to express in SQL (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:

ALL, ALL, ALL, (((

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 `CUBE` is an aggregation operation, it makes sense to externalize it by overloading the SQL `GROUP BY` operator. In fact, the cube is a relational operator, with `GROUP BY` and `ROLL UP` as degenerate forms of the operator. This can be conveniently specified by overloading the SQL `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:

```sql
SELECT day, nation, MAX(Temp) FROM Weather GROUP BY CUBE Day(Time) AS day, Country(Latitude, Longitude) AS nation;
```

The semantics of the `CUBE` operator are that it first aggregates over all the `<select list>` attributes in the `GROUP BY` clause as in a standard `GROUP BY`. Then, it `UNIONs` in each super-aggregate of the global cube—substituting `ALL` for the aggregation columns.

If there are `N` attributes in the `<select list>`, there will be `2^N - 1` super-aggregate values. If the cardinality of the `N` attributes are `C_1 ↪ C_2 ↪ ⋯ ↪ 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)
  
<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>

- **Join indexes**: to speed-up specific join queries
  - Example: Join fact table F with dimension tables D1 and D2
  - Index contain triples of rids <r_1,r_2,r> from D_1, D_2, 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 D_1, D_2 that join with r
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**

  ```sql
  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

Main Idea

• Most DBMS (up till now) store each tuple using row-major order

• Store tuples by column-major order instead?

Why?
From Row-Store to Column-Store

Rows stored contiguously on disk (+ tuple headers)

Columns stored contiguously on disk (no headers needed)
Recall Record Formats in Row Stores

Variable length records

Record header
## More Detailed Example

### 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>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
</tbody>
</table>

C-Store also avoids large tuple headers

### Page

<table>
<thead>
<tr>
<th>Page</th>
<th>B</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>
Column-Store Optimizations

Numbers from earlier paper and **C-Store system**: “Column-Stores vs. Row-Stores: How Different Are they Really?” Abadi et. al. SIGMOD’08.

- **Vectorized processing / Block iteration** (1.5X)
  - Pass blocks of values between ops instead of individual tuples
- **Compression**: e.g., run-length encoding of columns (10X)
- **Late tuple materialization** (3X improvement)
  - Process individual columns as long as possible
  - Merge columns into complete tuples as late as possible
- **Invisible joins** (1.5X)
### Compression Example

<table>
<thead>
<tr>
<th>Row-based (4 pages)</th>
<th>Column-based (4 pages)</th>
<th>Compressed (2 pages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page A 1</td>
<td>A 1</td>
<td>4XA</td>
</tr>
<tr>
<td>Page A 2</td>
<td>A 2</td>
<td>1X1</td>
</tr>
<tr>
<td>Page A 2</td>
<td>A 2</td>
<td>2XB</td>
</tr>
<tr>
<td>Page A 2</td>
<td>A 2</td>
<td>4X2</td>
</tr>
<tr>
<td>Page B 2</td>
<td>B 2</td>
<td>2XC</td>
</tr>
<tr>
<td>Page B 4</td>
<td>B 4</td>
<td>5X4</td>
</tr>
<tr>
<td>Page C 4</td>
<td>C 4</td>
<td></td>
</tr>
<tr>
<td>Page C 4</td>
<td>C 4</td>
<td></td>
</tr>
</tbody>
</table>
Late Tuple Materialization

Ex: SELECT R.b from R where R.a=X and R.d=Y

Early materialization

Late materialization

CSE 544 - Fall 2016
Late Tuple Materialization

Figure 4.1: An example of a select-project-join query with late materialization.
**Figure 4.2:** An example multi-column block containing values for the SHIPDATE, RETFLAG, and LINENUM columns. The block spans positions 47 to 53; within this range, positions 48, 49, 52, and 53 are active (i.e., they have passed all selection predicates).
Joins

```
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```
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
Conclusion

Teaching an Old Elephant New Tricks

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.