CMSC724: Data Warehouses

Amol Deshpande

University of Maryland, College Park

April 19, 2011
Outline

1. Overview
2. Variant Indexes
3. Data Cubes (Slides borrowed from Nick)
Data Warehouses

- A (usually) stand-alone system that integrates data from everywhere
  - Read-only, updated at night
  - Geared toward business analytics, data mining etc...
- Heavily used and heavily optimized
  1. Materialized views (summary tables, data cubes)
  2. New types of indexes
  3. New join techniques geared toward “star” (or “snowflake”) schemas
  4. Compressed storage techniques
- Key observation: Read-only, so updating not a critical issue
  - However this may not be true in many application domains
Data Warehouses: Overview

Figure 1. Data Warehousing Architecture

Figure: Overview (From Chaudhuri, Dayal; SIGMOD Rec., 1997)
Data Warehouses

- Extract-Transform-Load (ETL)
  - Data cleaning, auditing, integrity constraints
  - Semantic heterogeneity
    - Issues like entity resolution, schema mapping/matching, cleaning etc..

- Load/Refresh:
  - Typically done periodically
  - Batch loading, so can heavily optimize the indexes
    - E.g. If using a B+-tree, bulk-loading can result in much better indexes, than inserting one at a time
  - Refresh:
    - Usually done incrementally, at night or something

- Real-time analysis?
From MAD Skills: New Analysis Practices for Big Data

Claim: Business Analytics increasingly need to be much more agile, and accommodating of dirty data

Experiences from a Fox Audience Network Data Warehouse on Greenplum

- Serves ads across Fox online publishers (e.g. Myspace)
- 200 TB of unique production data: growing rapidly

Need to support: sales acct managers to research scientists

- No precomputed set of statistics will be enough
- Query: *How many female WWF enthusiasts under the age of 30 visited the Toyota community over the last four days and saw a standard-sized web ad?*
- Analysis: *How are these similar to those who visited Nissan?*
- First can be done in SQL, the second requires R (or Matlab, SAS, SPSS etc)
MAD Skills: New Analysis Practices for Big Data

Traditional View: There is no point in bringing data ... into the data warehouse environment without integrating it. If the data arrives at the data warehouse in an unintegrated state, it cannot be used to support a corporate view of data. And a corporate view of data is one of the essences of the architected environment.

- Can hold up access to data for months: not acceptable

Advocate a three-layer approach:

- **Staging schema**: loading raw fact tables or logs
  - Access only to Analysts and Engineers

- **Production Data Warehouse schema**
  - Aggregates that serve most users
  - Also SQL command-line access to some users

- **Reporting schema**
  - Specialized, static aggregates for supporting reporting tools and casual users
  - Data Cubes perhaps fit best here
MAD Skills: New Analysis Practices for Big Data

The paper also discusses how many statistical analysis tasks can be written in SQL

- Worth reading

Greenplum supports a variety of storage formats

- External tables through wrappers
- Specialized append-only stores for data that is not updated

- ...

Also supports Map-Reduce using User Defined Functions
The multidimensional data model grew out of the view of business data popularized by PC spreadsheet programs that were extensively used by business analysts. The spreadsheet applications often used raw data access tools and database server. In addition, there are query environments that provide its multidimensional coordinates, and stores the numeric measures for those coordinates. Each dimension uses a generated key for efficiency) to each of the dimensions. Figure 3 shows an example of a star schema.

The other popular operators include slice_and_dice sales data for a specific product to create a projection of the data on a subset of dimensions for selected values of the other dimensions. For example, we can reducing the dimensionality of the data, i.e., taking a converse of rollup.

The converse of rollup is still the most compelling front-end application for OLAP. Indeed, the Essbase product of Arbor Corporation uses these briefly in the next section. However, when a relational database server is used, the multidimensional model and its operations have to be mapped into relations and SQL queries. When a relational OLAP server is used, the multidimensional model and its operations have to be mapped into relations and SQL queries. We will describe these briefly in the next section. However, when a relational OLAP server is used, the multidimensional model and its operations have to be mapped into relations and SQL queries.

Figure 3. A Star Schema.

**Figure: A Star Schema (From Chaudhuri, Dayal; SIGMOD Record, 1997)**
Data Warehouses: Snowflake Schema

Figure 4. A Snowflake Schema.

Figure: A Snowflake Schema (From Chaudhuri, Dayal; SIGMOD Record, 1997)
Star and Snowflake Schemas

- The Facts table is HUGE
  - Dimension tables relatively small
- Strong key-foreign key dependencies
  - Each fact table tuple joins with exactly one tuple from each dimension table
  - Critical in optimizations

Many queries are of the form:
- Join the Facts table with some of the dimension tables
- Selections on the dimension table attributes (e.g. state = 'MD')
- Possibly selection on the fact table
- Group by on some of the dimension table attributes (e.g. ProdName)
- Aggregate on a main Facts table attribute (e.g. quantity)
Star and Snowflake Schemas

- The Facts table is HUGE
  - Dimension tables relatively small
- Strong key-foreign key dependencies
  - Each fact table tuple joins with exactly one tuple from each dimension table
  - Critical in optimizations
- Many queries are of the form:
  - Join the Facts table with some of the dimension tables
  - Selections on the dimension table attributes (e.g. state = 'MD')
  - Possibly selection on the fact table
  - Group by on some of the dimension table attributes (e.g. ProdName)
  - Aggregate on a main Facts table attribute (e.g. quantity)
Data Warehouses: Multi-dimensional Data

Figure 2. Multidimensional data

Figure: Multi-dimensional Data (From Chaudhuri, Dayal; SIGMOD Record, 1997)
Data Warehouses: Multi-dimensional Data

- OLAP: On-line Analytical Processing
  - Contrast with: OLTP (transaction processing)
- ROLAP: Relational OLAP
  - OLAP built on top of relational databases (standard now)
- MOLAP: Multi-dimensional OLAP
  - Specialized database that stores data in multi-dimensional arrays
  - Makes it easier to support Data Cube type queries
  - Updates need to be done at both places
- HOLAP: A hybrid between the two commonly used
  - Compute part of the cube at higher levels of aggregation, for rest go down to the relations
- See SAS OLAP Server + Teradata for a nice discussion
Data Warehouses: Today and Future?

- A Recent Article by Stonebraker
- Column stores are likely going to be the underlying storage engine for data warehouses
  - But are not optimized for OLTP workloads
  - Several specialized systems are likely the only way to go
- What about Map-Reduce?
  - Most data warehouses will likely evolve toward support some form of MapReduce
  - See Greenplum’s take on it
- Parallelism is pretty much required
- Increasingly need sophisticated statistical analysis
  - Many companies in this space: SAS, SPSS (bought by IBM), Tableau
  - Can’t always scale to very large volumes
  - See recent work at the SystemML project at IBM
Outline

1. Overview
2. Variant Indexes
3. Data Cubes (Slides borrowed from Nick)
Variant Indexes (O’Neil, Quass; SIGMOD’97)

- Specialized indexes for supporting summary aggregate queries
- Different types of indexes:
  - Traditional Value-List Indexes
  - Bitmap Indexes
  - Projection Indexes
    - Very similar to Column-based storage (much research last few years)
  - Bit-sliced Indexes
  - Join Indexes
- Key observation: Read-only database, so can build as many indexes as you want
- Many of the ideas are present in today’s column-stores
Key idea: Given a property over a domain, the following two are interchangeable and complementary:
- a list of values
- a bitmap over the domain
Key idea: Given a property over a domain, the following two are interchangeable and complementary
- a list of values
- a bitmap over the domain

In our case:
- Domain: The set of all RIDs
- Property: A predicate \( R.a = \text{'}Sports\text{'} \)

If the number of RID that satisfy the property is:
- small: store as a list of RIDs
- large: store as a bitmap over the RIDs

Figure 2.1. Example of a Bitmap Index on department, a column of the SALES table

Figure: Bitmap Index
Value-List/Bitmap Index: Segmentation

- Each disk page can store 48K bits, so must partition the Facts table into 48K row partitions
- So
  - Each B+-Tree page contains a portion of the bitmap over the RIDs

If the number of 1's is small, convert to an RID-list. The tipping point is when the number of 1's is < 1/32 of the size. At that point, the RID-list exactly fits in the disk page (48000/32 = 1500, 1500 * 4 = 6K). This is always true regardless of the page size. Segmentation also helps with space storage... if an entire segment is all 0's, don't store it.
Each disk page can store 48K bits, so must partition the Facts table into 48K row partitions.

So

Each B+-Tree page contains a portion of the bitmap over the RIDs.

If the number of 1’s is small, convert to an RID-list.

The tipping point is when the number of 1’s is < 1/32 of the size.

At that point, the RID-list exactly fits in the disk page.

\[(48000/32 = 1500, 1500 \times 4 = 6K)\]

This is always true regardless of the page size.

Segmentation also helps with space storage... if an entire segment is all 0’s, don’t store it.
Selections on the table return bitmaps
- AND, OR, NOT very fast on bitmaps
- Result called a Foundset: $B_f$ (the domain is the Facts Table)

Next step: Aggregate (recall almost all queries compute aggregates)
- Can perform directly on the bitmap in some cases (COUNT)
- Otherwise use projection indexes
- OR use a bit-sliced index
Value-List/Bitmap Index: COUNT

- shcount: count the number of ones in the binary representation
  - $\text{shcount}[3] = 2$, $\text{shcount}[10] = 2$, $\text{shcount}[15] = 4$ etc...

Algorithm 2.1. Performing COUNT with a Bitmap

```c
/* Assume B1[ ] is a short int array overlaying a Foundset Bitmap */
count = 0;
for (i = 0; i < SHNUM; i++)
    count += shcount[B1[i]];
/* add count of bits for next short int */
```

Figure: Bitmap Index
Other Indexes

- Projection Indexes
  - Simply a list of the values for one attribute for all values
  - Like the leaf-level in a B+-tree (except no pointers)
  - Each page contains same number of values (so easier to identify the value for a specific tuple)
Other Indexes

- **Projection Indexes**
  - Simply a list of the values for one attribute for all values
  - Like the leaf-level in a B+-tree (except no pointers)
  - Each page contains same number of values (so easier to identify the value for a specific tuple)

- **Bit-sliced Index**
  - A set of bitmaps, one for each “position” in the binary representation of the values
    - Makes more sense for numerical/ordinal attributes
  - Can be used for computing aggregates like SUM

### Table 3.5
<table>
<thead>
<tr>
<th>Aggregate</th>
<th>Value-List Index</th>
<th>Projection Index</th>
<th>Bit-Sliced Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>Not needed</td>
<td>Not needed</td>
<td>Not needed</td>
</tr>
<tr>
<td>SUM</td>
<td>Not bad</td>
<td>Good</td>
<td>Best</td>
</tr>
<tr>
<td>AVG (SUM/COUNT)</td>
<td>Not bad</td>
<td>Good</td>
<td>Best</td>
</tr>
<tr>
<td>MAX and MIN</td>
<td>Best</td>
<td>Slow</td>
<td>Slow</td>
</tr>
<tr>
<td>MEDIAN, N-TILE</td>
<td>Usually Best</td>
<td>Not Useful</td>
<td>Sometimes Best²</td>
</tr>
<tr>
<td>Column-Product</td>
<td>Very Slow</td>
<td>Best</td>
<td>Very Slow</td>
</tr>
</tbody>
</table>

**Table 3.5.** Tabulation of Performance by Index Type for Evaluating Aggregate Functions
Other Indexes

- **Bitmap Join Index**
  - A bitmap index on the Facts table on a \textit{dimension attribute}
  - Recall that each Facts table tuple joins with exactly one tuple from any dimension table
  - So each Facts table tuple has a unique value for a dimension attribute
    - In the example, might build a bitmap index on attribute "STATE" on Fact table
  - Using Bitmap Join Index, can evaluate predicates directly on the Fact table
  - \textit{Grouping} also more efficient
Other Indexes

- Clustering
  - Can get better performance by appropriately clustering the Fact table

- Groupset Indexes
  - For better grouping performance
Other Indexes

- **Clustering**
  - Can get better performance by appropriately clustering the Fact table

- **Groupset Indexes**
  - For better grouping performance

- Many other optimizations commonly found in Data Warehouses
Other issues

- Using monetary cost as the metric
  - Useful in the real-world (see the 5-minute rule paper by Gray, Graefe in the Redbook)

- Bloom Filters
  - Can’t use them here... there is no “check” at the end to make sure the results are accurate

- Data Mining
  - Typically done outside the database, so the indexes don’t really help

- Applicability to operational databases
  - Not much... these techniques require too many indexes
  - An update would require changing all of those... prohibitive
Outline

1. Overview
2. Variant Indexes
3. Data Cubes (Slides borrowed from Nick)
OLAP-The Data Analysis Cycle

- User extracts data from database with query

- Then visualizes, analyzes data with desktop tools
The Data Cube
[Gray, Bosworth, Layman, Pirahesh ICDE 96]

- summarize multidimensional data for trend analysis

<table>
<thead>
<tr>
<th>Time (UCT)</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Altitude (m)</th>
<th>Temp (°C)</th>
<th>Pres (mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27/11/94:1500</td>
<td>34:16:18N</td>
<td>27:05:55W</td>
<td>10</td>
<td>23</td>
<td>1024</td>
</tr>
</tbody>
</table>

- groupby with statistical functions (avg, min, max, count, sum) aggregates over table sub-groups

```sql
select avg(temp) from weather
select time, altitude from weather
groupby time, altitude
```

- results in a new table

```sql
select location, sum(units) from inventory
group by location
having nation = "USA";
```
Problems with SQL Groupbys

- Histograms (aggregation over computed categories)

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

<table>
<thead>
<tr>
<th>Time (UCT)</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Altitude (m)</th>
<th>Temp. (c)</th>
<th>Pres. (mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>96/6/1:1500</td>
<td>37:58:33N</td>
<td>122:45:28W</td>
<td>102</td>
<td>21</td>
<td>1009</td>
</tr>
<tr>
<td>96/6/7:1500</td>
<td>34:16:18N</td>
<td>27:05:55W</td>
<td>10</td>
<td>23</td>
<td>1024</td>
</tr>
</tbody>
</table>

Many more rows like the ones above and below
Problems with SQL Groupbys

• drill-down and roll-up

Table 3: Sales Roll Up by Model by Year by Color

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Sales by Model by Year by Color</th>
<th>Sales by Model by Year</th>
<th>Sales by Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chevy</td>
<td>1994</td>
<td>black</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>white</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>black</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>white</td>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>black</td>
<td>115</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>white</td>
<td></td>
<td>200</td>
<td>290</td>
</tr>
</tbody>
</table>

Not relational (null values in the keys)

```
SELECT Model, ALL, ALL, SUM(Sales) FROM Sales
WHERE Model = 'Chevy'
GROUP BY Model
UNION
SELECT Model, Year, ALL, SUM(Sales) FROM Sales
WHERE Model = 'Chevy'
GROUP BY Model, Year
UNION
SELECT Model, Year, Color, SUM(Sales) FROM Sales
WHERE Model = 'Chevy'
GROUP BY Model, Year, Color;
```
More problems with Groupbys

• roll-up is asymmetric (e.g. does not aggregate by year or by color alone
• cross-tabulation (spreadsheets)

![Table 5: Chevy Sales Cross Tab](image)

<table>
<thead>
<tr>
<th></th>
<th>1994</th>
<th>1995</th>
<th>total (ALL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td>50</td>
<td>85</td>
<td>135</td>
</tr>
<tr>
<td>white</td>
<td>40</td>
<td>115</td>
<td>155</td>
</tr>
<tr>
<td>total</td>
<td>90</td>
<td>200</td>
<td>290</td>
</tr>
</tbody>
</table>

• even if SQL syntax can be devised, a 6D cross-tab requires 64 groupby queries to generate it and 64 scans and sorts of the data

◆ most of these are not relational expressions but are in many report writers
CUBE: A Relational Aggregate Operator Generalizing Group By

The Data Cube and The Sub-Space Aggregates

© N. Roussopoulos 2007
Idea: N-dimensional Cube
Each Attribute is a Dimension

• N-dimensional Aggregate (sum(), max(), ...)
  ◆ fits relational model exactly:
    ➔ a₁, a₂, ..., aₙ, f(*)

• Super-aggregate over N-1 Dimensional sub-cubes
  ➔ ALL, a₂, ..., aₙ, f(*)
  ➔ a₃, ALL, a₃, ..., aₙ, f(*)
  ➔ ...
  ➔ a₁, a₂, ..., ALL, f(*)
  ◆ this is the N-1 Dimensional cross-tab.

• Super-aggregate over N-2 Dimensional sub-cubes
  ➔ ALL, ALL, a₃, ..., aₙ, f(*)
  ➔ ...
  ➔ a₁, a₂, ..., ALL, ALL, f(*)
Division of labor
Computation vs Visualization

• Relational system builds CUBE relation
  ◆ aggregation best done close to data
  ◆ filtering of data is possible
  ◆ Cube computation may be recursive
    ➔ (e.g., percent of total, quartile, ....)

• Visualization System displays/explores the cube
An Example

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chevy</td>
<td>1990</td>
<td>red</td>
<td>5</td>
</tr>
<tr>
<td>Chevy</td>
<td>1990</td>
<td>white</td>
<td>87</td>
</tr>
<tr>
<td>Chevy</td>
<td>1990</td>
<td>blue</td>
<td>62</td>
</tr>
<tr>
<td>Chevy</td>
<td>1991</td>
<td>red</td>
<td>54</td>
</tr>
<tr>
<td>Chevy</td>
<td>1991</td>
<td>white</td>
<td>95</td>
</tr>
<tr>
<td>Chevy</td>
<td>1991</td>
<td>blue</td>
<td>49</td>
</tr>
<tr>
<td>Chevy</td>
<td>1992</td>
<td>red</td>
<td>31</td>
</tr>
<tr>
<td>Chevy</td>
<td>1992</td>
<td>white</td>
<td>54</td>
</tr>
<tr>
<td>Chevy</td>
<td>1992</td>
<td>blue</td>
<td>71</td>
</tr>
<tr>
<td>Ford</td>
<td>1990</td>
<td>red</td>
<td>64</td>
</tr>
<tr>
<td>Ford</td>
<td>1990</td>
<td>white</td>
<td>62</td>
</tr>
<tr>
<td>Ford</td>
<td>1990</td>
<td>blue</td>
<td>63</td>
</tr>
<tr>
<td>Ford</td>
<td>1991</td>
<td>red</td>
<td>52</td>
</tr>
<tr>
<td>Ford</td>
<td>1991</td>
<td>white</td>
<td>9</td>
</tr>
<tr>
<td>Ford</td>
<td>1991</td>
<td>blue</td>
<td>55</td>
</tr>
<tr>
<td>Ford</td>
<td>1992</td>
<td>red</td>
<td>27</td>
</tr>
<tr>
<td>Ford</td>
<td>1992</td>
<td>white</td>
<td>62</td>
</tr>
<tr>
<td>Ford</td>
<td>1992</td>
<td>blue</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA CUBE</th>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Year</td>
<td>Color</td>
<td>Sales</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>ALL</td>
<td>942</td>
</tr>
<tr>
<td>chevy</td>
<td>ALL</td>
<td>ALL</td>
<td>510</td>
</tr>
<tr>
<td>ford</td>
<td>ALL</td>
<td>ALL</td>
<td>432</td>
</tr>
<tr>
<td>ALL</td>
<td>1990</td>
<td>ALL</td>
<td>343</td>
</tr>
<tr>
<td>ALL</td>
<td>1991</td>
<td>ALL</td>
<td>314</td>
</tr>
<tr>
<td>ALL</td>
<td>1992</td>
<td>ALL</td>
<td>285</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>red</td>
<td>165</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>white</td>
<td>273</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>blue</td>
<td>339</td>
</tr>
<tr>
<td>chevy</td>
<td>1990</td>
<td>ALL</td>
<td>154</td>
</tr>
<tr>
<td>chevy</td>
<td>1991</td>
<td>ALL</td>
<td>199</td>
</tr>
<tr>
<td>chevy</td>
<td>1992</td>
<td>ALL</td>
<td>157</td>
</tr>
<tr>
<td>ford</td>
<td>1990</td>
<td>ALL</td>
<td>189</td>
</tr>
<tr>
<td>ford</td>
<td>1991</td>
<td>ALL</td>
<td>116</td>
</tr>
<tr>
<td>ford</td>
<td>1992</td>
<td>ALL</td>
<td>128</td>
</tr>
<tr>
<td>chevy</td>
<td>ALL</td>
<td>red</td>
<td>91</td>
</tr>
<tr>
<td>chevy</td>
<td>ALL</td>
<td>white</td>
<td>236</td>
</tr>
<tr>
<td>chevy</td>
<td>ALL</td>
<td>blue</td>
<td>183</td>
</tr>
<tr>
<td>ford</td>
<td>ALL</td>
<td>red</td>
<td>144</td>
</tr>
<tr>
<td>ford</td>
<td>ALL</td>
<td>white</td>
<td>133</td>
</tr>
<tr>
<td>ford</td>
<td>ALL</td>
<td>blue</td>
<td>156</td>
</tr>
<tr>
<td>ALL</td>
<td>1990</td>
<td>red</td>
<td>69</td>
</tr>
<tr>
<td>ALL</td>
<td>1990</td>
<td>white</td>
<td>149</td>
</tr>
<tr>
<td>ALL</td>
<td>1990</td>
<td>blue</td>
<td>125</td>
</tr>
<tr>
<td>ALL</td>
<td>1991</td>
<td>red</td>
<td>107</td>
</tr>
<tr>
<td>ALL</td>
<td>1991</td>
<td>white</td>
<td>104</td>
</tr>
<tr>
<td>ALL</td>
<td>1991</td>
<td>blue</td>
<td>104</td>
</tr>
<tr>
<td>ALL</td>
<td>1992</td>
<td>red</td>
<td>59</td>
</tr>
<tr>
<td>ALL</td>
<td>1992</td>
<td>white</td>
<td>116</td>
</tr>
<tr>
<td>ALL</td>
<td>1992</td>
<td>blue</td>
<td>110</td>
</tr>
</tbody>
</table>
Summary of the Cube

- CUBE operator generalizes relational aggregates
- Needs ALL value to denote sub-cubes
  - ALL values represent aggregation sets
- Needs generalization of user-defined aggregates
- Decorations and abstractions are interesting
- Computation has interesting optimizations
- Relationship to “rest of SQL” not fully worked out.
How to "Query" Cubes?

- Over the years, a language has been developed
- Called MDX: MultiDimensional eXpressions
- Example:

```
SELECT
([Geography].[Geo].[Country].members) ON 0,
([Time].[Year].members) ON 1
FROM
[Sales]
```

<table>
<thead>
<tr>
<th></th>
<th>Spain</th>
<th>Switzerland</th>
<th>France</th>
<th>United States</th>
<th>Canada</th>
<th>Mexico</th>
<th>Venezuela</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>3.0</td>
<td>248.0</td>
<td>4</td>
<td>768.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- More details at: [MDX: A Gentle Introduction](#)
Materialized Views

- CUBE is a user abstraction
- How do we store/maintain it?
  - Option 1: Compute the entire CUBE over all possible attributes
    - Not feasible... too large
    - Maintenance cost will be formidable
CUBE is a user abstraction

How do we store/maintain it?

Option 1: Compute the entire CUBE over all possible attributes
   - Not feasible... too large
   - Maintenance cost will be formidable

Option 2: Compute from scratch
   - Don’t materialize anything... go straight to the source table for every query
   - Too much query latency
Materialized Views

- CUBE is a user abstraction
- How do we store/maintain it?
  - Option 1: Compute the entire CUBE over all possible attributes
    - Not feasible... too large
    - Maintenance cost will be formidable
  - Option 2: Compute from scratch
    - Don’t materialize anything... go straight to the source table for every query
    - Too much query latency
  - Option 3: Materialize a portion of the CUBE
    - Keep the most commonly required portions around
    - Try to answer all queries using that
Cube={Materialized Views}
[Harinarayan, Rajaraman, Ullman 96]

- each groupby creates a “summary table” which is a materialized view with some dressing
- storing these summary tables speed up cube queries
- what to store and what not
- TPC-D example for sale analysis

1. part, supplier, customer (6M, i.e., 6 million rows)
2. part, customer (6M)
3. part, supplier (0.8M)
4. supplier, customer (6M)
5. part (0.2M)
6. supplier (0.01M)
7. customer (0.1M)
8. none (1)
The Lattice Organization

- the query sales groupby part will be answered at
  - p - cost of scanning 0.2M records
  - pc - 6.0M
  - psc - 6.0M

- select the views that minimize overall query performance
  - need a good query model
  - need a good optimization criterion
Views grow exponentially

• in general $2^{**N}$ subspaces
DynaMat
Yannis Kotidis, Nick Roussopoulos (Sigmod 1999)

• Conventional Data Warehouse
  ▪ pre-computed set view is static (too hard to select and adjust)
  ▪ usually selected by an administrator

• DynaMat proposed a framework for automatic management of views
  ▪ Unifies view selection & view refresh
  ▪ Amortizes generation and maintenance cost over multiple uses of cached results

• Techniques
  ▪ DynMat caches the results of every query
  ▪ Each incoming query is evaluated against the cached results to see if any of those can be used
  ▪ The captured set is updated within an update cycle to the extent possible
Online Operation

• Try to match each query from the view pool (Fragment Locator)
  - Fragments are either single value predicates or complete ranges
  - A Directory Index is maintained for efficient searches
• On the fly decide whether to cache the result in the pool (Admission Control Entity)
Materialized Range Fragments

- Materialized Results are restricted to one of:
  a) a full Range $R_i = \{\text{min}_d, \text{max}_d\}$
  b) a single value for $d_i$
  c) an empty range denotes a dimension that is not present in the query
- SQL queries are mapped to MR queries that are answered by cached MRFs
- MRFs are coarser than query results (expanded when necessary)
- No combination of MRFs are used to answer a query (more costly especially when MRFs are too small and/or overlap)
- An R-tree based index is used to identify possible MRFs that can answer the query—among those, the best fit is chosen
- The use of MRFs makes matching efficient.

Figure 5: Querying stored MRFs

Figure 6: Directory for view $(\text{product, customer})$