MSCIT 5210/MSCBD 5002: Knowledge Discovery and Data Mining

Acknowledgement: Slides modified by Dr. Lei Chen based on the slides provided by Jiawei Han, Micheline Kamber, and Jian Pei

©2012 Han, Kamber & Pei. All rights reserved.
Chapter 4: Data Warehousing, On-line Analytical Processing and Data Cube

- Data Warehouse: Basic Concepts
- Data Warehouse Modeling: Data Cube and OLAP
- Data Cube Computation: Preliminary Concepts
- Data Cube Computation Methods
- Summary
Aspects of SQL

- Most common Query Language – used in all commercial systems
- Discussion is based on the SQL92 Standard. Commercial products have different features of SQL, but the basic structure is the same
- Data Manipulation Language
- Data Definition Language
- Constraint Specification
- Embedded SQL
- Transaction Management
- Security Management .....
Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

  \[
  \text{select } A_1, A_2, \ldots, A_n \text{ from } R_1, R_2, \ldots, R_m \text{ where } P
  \]

  - \(A_i\) represent attributes
  - \(R_i\) represent relations
  - \(P\) is a predicate.

- This query is equivalent to the relational algebra expression:

  \[
  \Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))
  \]

- The result of an SQL query is a relation (but may contain duplicates). SQL statements can be nested.
Projection

- **The select clause** corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.

- Find the names of all branches in the loan relation
  
  ```
  select branch-name
  from loan
  ```

  Equivalent to: \( \Pi_{\text{branch-name}}(\text{loan}) \)

- **An asterisk in the select clause** denotes “all attributes”
  
  ```
  select *
  from loan
  ```

- **Note**: for our examples we use the tables:
  - Branch (branch-name, branch-city, assets)
  - Customer (customer-name, customer-street, customer-city)
  - Loan (loan-number, amount, branch-name)
  - Account (account-number, balance, branch-name)
  - Borrower (customer-name, loan-number)
  - Depositor (customer-name, account-number)
Duplicate Removal

- SQL allows duplicates in relations as well as in query results. Use `select distinct` to force the elimination of duplicates.

Find the names of all branches in the loan relation, and remove duplicates

```
select distinct branch-name
from loan
```

- The keyword `all` specifies that duplicates are not removed.

```
select all branch-name
from loan
```

`force` the DBMS to remove duplicates

`force` the DBMS not to remove duplicates
Arithmetic Operations on Retrieved Results

• The **select** clause can contain arithmetic expressions involving the operators, $$+, -, \div$$ and $$\times$$, and operating on constants or attributes of tuples.

• The query:

  ```sql
  select branch-name, loan-number, amount * 100
  from loan
  ```

  would return a relation which is the same as the loan relations, except that the attribute amount is multiplied by 100
The where Clause

- The *where* clause specifies conditions that tuples in the relations in the *from* clause must satisfy.
- Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than $1200.

```sql
select loan-number
from loan
where branch-name="Perryridge" and amount > 1200
```

- SQL allows logical connectives *and*, *or*, and *not*. Arithmetic expressions can be used in the comparison operators.
- Note: attributes used in a query (both *select* and *where* parts) must be defined in the relations in the *from* clause.
The where Clause (Cont.)

- SQL includes the `between` operator for convenience.
- Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, $\geq 90,000$ and $\leq 100,000$)

```sql
select loan-number
from loan
where amount between 90000 and 100000
```
The from Clause

- The **from** clause corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product borrower × loan

```sql
select *
from borrower, loan
```

It is rarely used without a where clause.

- Find the name and loan number of all customers having a loan at the Perryridge branch.

```sql
select distinct customer-name, borrower.loan-number
from borrower, loan
where borrower.loan-number = loan.loan-number and branch-name = "Perryridge"
```
Aggregate Functions

- Operate on a column of a relation, and return a value
  - `avg`: average value
  - `min`: minimum value
  - `max`: maximum value
  - `sum`: sum of values
  - `count`: number of values

- Note: for our examples we use the tables:
  - Branch (`branch-name`, `branch-city`, `assets`)
  - Customer (`customer-name`, `customer-street`, `customer-city`)
  - Loan (`loan-number`, `amount`, `branch-name`)
  - Account (`account-number`, `balance`, `branch-name`)
  - Borrower (`customer-name`, `loan-number`)
  - Depositor (`customer-name`, `account-number`)
Aggregate Function Computation

- Find the average account balance at the Perryridge branch.

```
select avg(balance)
from account
where branch-name="Perryridge"
```

**Balances of Perryridge accounts**

- **Avg()**: 120,000
Examples of Aggregate Functions

- Find the numbers of tuples in the customer relation.
  
  ```sql
  select count(*)
  from customer
  ```
  - remember * stands for all attributes
  - Same as:
    ```sql
    select count(customer-city)
    from customer
    ```
  - Different from:
    ```sql
    select count(distinct customer-city)
    from customer
    ```
  - Because customer-city is not a key
Group by

- Find the number of accounts for each branch.
  
  \[
  \text{select } \text{branch-name, count(account-number)} \\
  \text{from account} \\
  \text{group by branch-name}
  \]

- For each group of tuples with the same branch-name, apply aggregate function count and distinct to account-number

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>a-102</td>
<td>400</td>
</tr>
<tr>
<td>Brighton</td>
<td>a-217</td>
<td>750</td>
</tr>
<tr>
<td>Perryridge</td>
<td>a-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>a-215</td>
<td>750</td>
</tr>
<tr>
<td>Redwood</td>
<td>a-222</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>a-102</td>
<td>400</td>
</tr>
<tr>
<td>Perryridge</td>
<td>a-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>a-217</td>
<td>750</td>
</tr>
<tr>
<td>Brighton</td>
<td>a-215</td>
<td>750</td>
</tr>
<tr>
<td>Redwood</td>
<td>a-222</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch-name</th>
<th>count-account-no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>2</td>
</tr>
<tr>
<td>Brighton</td>
<td>2</td>
</tr>
<tr>
<td>Redwood</td>
<td>1</td>
</tr>
</tbody>
</table>
Attributes in `select` clause outside of aggregate functions must appear in `group by` list, why?

```
select branch-name, balance, count( distinct account-number)
from account
```  

```correction```

```
select branch-name, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
correct
```

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
from account
```  

```
select branch-name, balance, count(distinct account-number)
Group by with Join

- Find the number of depositors for each branch.

```sql
select branch-name, count( distinct customer-name )
from depositor, account
where depositor.account-number = account.account-number
group by branch-name
```

- Perform Join then group by then count ( distinct () )

```sql
depositor (customer-name, account-number)
account (branch-name, account-number, balance)
Join ⇒ (customer-name, account-number, branch-name, balance)
```

- Group by and aggregate functions apply to the Join result
Group by Evaluation

**Query:**

select branch-name, customer-name 
from depositor, account 
where depositor.account-number = account.account-number

group by branch-name, customer-name

distinct count

<table>
<thead>
<tr>
<th>branch-name</th>
<th>cust-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>John Wong</td>
</tr>
<tr>
<td>Perryridge</td>
<td>Jacky Chan</td>
</tr>
<tr>
<td>Uptown</td>
<td>John Wong</td>
</tr>
<tr>
<td>Uptown</td>
<td>Mary Kwan</td>
</tr>
<tr>
<td>Downtown</td>
<td>John Wong</td>
</tr>
<tr>
<td>Downtown</td>
<td>Pat Lee</td>
</tr>
<tr>
<td>Downtown</td>
<td>May Cheung</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch-name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>2</td>
</tr>
<tr>
<td>Uptown</td>
<td>2</td>
</tr>
<tr>
<td>Downtown</td>
<td>3</td>
</tr>
</tbody>
</table>
Having Clause

• Find the names of all branches where the average account balance is more than $700
  
  \[
  \text{select branch-name, } \text{avg(balance)} \\
  \text{from account} \\
  \text{group by branch-name} \\
  \text{having avg(balance) > 700}
  \]

• predicates in the **having** clause are applied to *each group* after the formation of groups

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>a-102</td>
<td>400</td>
</tr>
<tr>
<td>Perryridge</td>
<td>a-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>a-217</td>
<td>750</td>
</tr>
<tr>
<td>Brighton</td>
<td>a-215</td>
<td>750</td>
</tr>
<tr>
<td>Redwood</td>
<td>a-222</td>
<td>700</td>
</tr>
</tbody>
</table>
Group-by

- Motivation: Group-by permits us to display aggregate results (e.g., max, min, sum) for groups. For instance, if we have GROUP-BY X, we will get a result for every different value of X.
- Recall that aggregate queries without group-by return just a single number.
- If we put an attribute in SELECT, the attribute must also appear in GROUP-BY. The opposite is not true: there may be attributes in GROUP-BY that do not appear in SELECT.
- Any condition that appears in WHERE, is applied before the formation of groups – in other words, records that do not pass the WHERE condition are eliminated before the formation of groups.
- Any condition that appears in HAVING refers to the groups and is applied after the formation of the groups. The condition must involve aggregate functions, or attributes that appear in the SELECT or GROUP-BY lines.
Query 1: Find the total number of copies in stock for each poet

<table>
<thead>
<tr>
<th>poet</th>
<th>book</th>
<th>copies_in_stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>The Skull</td>
<td>21</td>
</tr>
<tr>
<td>Douglas Livingstone</td>
<td>A Littoral Zone</td>
<td>2</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Tstetlo</td>
<td>3</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Must Weep</td>
<td>8</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>A Tough Tale</td>
<td>2</td>
</tr>
</tbody>
</table>

SELECT poet, SUM (copies_in_stock) as sum
FROM writer
GROUP BY poet
Query 2: For each poet, find the max, min, avg and total number of copies in stock

<table>
<thead>
<tr>
<th>poet</th>
<th>book</th>
<th>copies_in_stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>The Skull</td>
<td>21</td>
</tr>
<tr>
<td>Douglas Livingstone</td>
<td>A Littoral Zone</td>
<td>2</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Tstetlo</td>
<td>3</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Must Weep</td>
<td>8</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>A Tough Tale</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>poet</th>
<th>max</th>
<th>min</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>21</td>
<td>2</td>
<td>11.5</td>
<td>23</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>8</td>
<td>2</td>
<td>4.33</td>
<td>13</td>
</tr>
</tbody>
</table>

```
SELECT poet, MAX(copies_in_stock) AS max, MIN(copies_in_stock) AS min, AVG(copies_in_stock) AS avg, SUM(copies_in_stock) AS sum
FROM writer
GROUP BY poet
```
Query 3: For each poet, find the max, min, avg and total number of copies in stock – take into account only books that have > 5 copies in stock

```
SELECT poet, MAX(copies_in_stock) AS max, MIN(copies_in_stock) AS min, AVG(copies_in_stock) AS avg, SUM(copies_in_stock) AS sum
FROM writer
WHERE copies_in_stock > 5
GROUP BY poet
```
Query 4: Find the total number of copies in stock for each poet who has a total of more than 20 copies in stock

```
SELECT poet, SUM(copies_in_stock) AS sum
FROM writer
GROUP BY poet
HAVING sum>20
```

<table>
<thead>
<tr>
<th>poet</th>
<th>book</th>
<th>copies_in_stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>The Skull</td>
<td>21</td>
</tr>
<tr>
<td>Douglas Livingstone</td>
<td>A Littoral Zone</td>
<td>2</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Tsetelo</td>
<td>3</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Must Weep</td>
<td>8</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>A Tough Tale</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>poet</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>23</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>13</td>
</tr>
</tbody>
</table>
Query 5: Find the total number of copies in stock for each poet who has a total of more than 20 copies in stock – take into account only books that have more than 5 copies in stock

<table>
<thead>
<tr>
<th>poet</th>
<th>book</th>
<th>copies_in_stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>The Skull</td>
<td>21</td>
</tr>
<tr>
<td>Douglas Livingstone</td>
<td>A Littoral Zone</td>
<td>2</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Tstetlo</td>
<td>3</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Must Weep</td>
<td>8</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>A Tough Tale</td>
<td>2</td>
</tr>
</tbody>
</table>

SELECT poet, SUM(copies_in_stock) AS sum
FROM writer
WHERE copies_in_stock > 5
GROUP BY poet
HAVING sum > 20
Query 6: Find the total number of copies in stock for each poet whose name starts with any letter after “E”

SELECT poet, SUM(copies_in_stock) as sum
FROM writer
WHERE poet > "E"
GROUP BY poet

SELECT poet, SUM(copies_in_stock) as sum
FROM writer
GROUP BY poet
HAVING poet > "E"
Query 7: Find the total number of copies in stock for each poet who has more than 2 books

<table>
<thead>
<tr>
<th>poet</th>
<th>book</th>
<th>copies_in_stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas Livingstone</td>
<td>The Skull</td>
<td>21</td>
</tr>
<tr>
<td>Douglas Livingstone</td>
<td>A Littoral Zone</td>
<td>2</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Tstetlo</td>
<td>3</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>Must Weep</td>
<td>8</td>
</tr>
<tr>
<td>Mongane Wally</td>
<td>A Tough Tale</td>
<td>2</td>
</tr>
</tbody>
</table>

```
SELECT poet, SUM(copies_in_stock) as sum
FROM writer
GROUP BY poet
HAVING count(*) > 2
```
DATA WAREHOUSES and OLAP

- **On-Line Transaction Processing (OLTP)** Systems manipulate operational data, necessary for day-to-day operations. Most existing database systems belong to this category.

- **On-Line Analytical Processing (OLAP)** Systems support specific types of queries (based on group-bys and aggregation operators) useful for decision making.

- **Data Mining** tools discover interesting patterns in the data.
Why OLTP is not sufficient for Decision Making

Let's say that Welcome supermarket uses a relational database to keep track of sales in all of stores simultaneously.

<table>
<thead>
<tr>
<th>product id</th>
<th>store id</th>
<th>quantity sold</th>
<th>date/time of sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>567</td>
<td>17</td>
<td>1</td>
<td>1997-10-22 09:35:14</td>
</tr>
<tr>
<td>219</td>
<td>16</td>
<td>4</td>
<td>1997-10-22 09:35:14</td>
</tr>
<tr>
<td>219</td>
<td>17</td>
<td>1</td>
<td>1997-10-22 09:35:17</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example (cont.)

#### PRODUCTS table

<table>
<thead>
<tr>
<th>Prod. id</th>
<th>product name</th>
<th>product category</th>
<th>Manufac. t. id</th>
</tr>
</thead>
<tbody>
<tr>
<td>567</td>
<td>Colgate Gel Pump 6.4 oz.</td>
<td>toothpaste</td>
<td>68</td>
</tr>
<tr>
<td>219</td>
<td>Diet Coke 12 oz. can</td>
<td>soda</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### STORES table

<table>
<thead>
<tr>
<th>store id</th>
<th>city id</th>
<th>store location</th>
<th>phone number</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>34</td>
<td>510 Main Street</td>
<td>415-555-1212</td>
</tr>
<tr>
<td>17</td>
<td>58</td>
<td>13 Maple Avenue</td>
<td>914-555-1212</td>
</tr>
</tbody>
</table>

#### CITIES table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>state</th>
<th>Popul.</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>San Francisco</td>
<td>California</td>
<td>700,000</td>
</tr>
<tr>
<td>58</td>
<td>East Fishkill</td>
<td>New York</td>
<td>30,000</td>
</tr>
</tbody>
</table>
An executive asks "I noticed that there was a Colgate promotion recently, directed at people who live in small towns. How much Colgate toothpaste did we sell in those towns yesterday? And how much on the same day a month ago?"

```sql
select sum(sales.quantity_sold)
from sales, products, stores, cities
where products.manufacturer_id = 68  -- restrict to Colgate-
  and products.product_category = 'toothpaste'
  and cities.population < 40000
  and sales.datetime_of_sale::date = 'yesterday'::date
  and sales.product_id = products.product_id
  and sales.store_id = stores.store_id
  and stores.city_id = cities.city_id
```
Example (cont.)

- You have to do a 4-way JOIN of some large tables. Moreover, these tables are being updated as the query is executed.
- Need for a separate RDBMS installation (i.e., a Data Warehouse) to support queries like the previous one.
- The Warehouse can be tailor-made for specific types of queries: if you know that the toothpaste query will occur every day then you can denormalize the data model.
Example (cont.)

- Suppose Welcome acquires ParknShop which is using a different set of OLTP data models and a different brand of RDBMS to support them. But you want to run the toothpaste queries for both divisions.
- Solution: Also copy data from the ParknShop Database into the Welcome Data Warehouse (data integration problems).
- One of the more important functions of a data warehouse in a company that has disparate computing systems is to provide a view for management as though the company were in fact integrated.
Motivation

- In most organizations, data about specific parts of business is there -- lots and lots of data, somewhere, in some form.
- Data is available but *not information* -- and *not the right information at the right time*.
- To bring together information from multiple sources as to provide a consistent database source for decision support queries.
- To off-load decision support applications from the on-line transaction system.
What is a Data Warehouse?

- Defined in many different ways, but not rigorously.
  - A decision support database that is maintained separately from the organization’s operational database
  - Support information processing by providing a solid platform of consolidated, historical data for analysis.
- “A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management’s decision-making process.”—W. H. Inmon
- Data warehousing:
  - The process of constructing and using data warehouses
Data Warehouse—Subject-Oriented

- Organized around major subjects, such as customer, product, sales
- Focusing on the modeling and analysis of data for decision makers, not on daily operations or transaction processing
- Provide a simple and concise view around particular subject issues by excluding data that are not useful in the decision support process
Data Warehouse—Integrated

- Constructed by integrating multiple, heterogeneous data sources
  - relational databases, flat files, on-line transaction records
- Data cleaning and data integration techniques are applied.
  - Ensure consistency in naming conventions, encoding structures, attribute measures, etc. among different data sources
    - E.g., Hotel price: currency, tax, breakfast covered, etc.
- When data is moved to the warehouse, it is converted.
The time horizon for the data warehouse is significantly longer than that of operational systems

- Operational database: current value data
- Data warehouse data: provide information from a historical perspective (e.g., past 5-10 years)

- Every key structure in the data warehouse
  - Contains an element of time, explicitly or implicitly
  - But the key of operational data may or may not contain “time element”
Data Warehouse—Nonvolatile

- A physically separate store of data transformed from the operational environment
- Operational update of data does not occur in the data warehouse environment
  - Does not require transaction processing, recovery, and concurrency control mechanisms
  - Requires only two operations in data accessing:
    - initial loading of data and access of data
## OLTP vs. OLAP

<table>
<thead>
<tr>
<th></th>
<th>OLTP</th>
<th>OLAP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>users</strong></td>
<td>clerk, IT professional</td>
<td>knowledge worker</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>day to day operations</td>
<td>decision support</td>
</tr>
<tr>
<td><strong>DB design</strong></td>
<td>application-oriented</td>
<td>subject-oriented</td>
</tr>
<tr>
<td><strong>data</strong></td>
<td>current, up-to-date detailed, flat relational isolated</td>
<td>historical, summarized, multidimensional integrated, consolidated</td>
</tr>
<tr>
<td><strong>usage</strong></td>
<td>repetitive</td>
<td>ad-hoc</td>
</tr>
<tr>
<td><strong>access</strong></td>
<td>read/write index/hash on prim. key</td>
<td>lots of scans</td>
</tr>
<tr>
<td><strong>unit of work</strong></td>
<td>short, simple transaction</td>
<td>complex query</td>
</tr>
<tr>
<td># records accessed</td>
<td>tens</td>
<td>millions</td>
</tr>
<tr>
<td>#users</td>
<td>thousands</td>
<td>hundreds</td>
</tr>
<tr>
<td><strong>DB size</strong></td>
<td>100MB-GB</td>
<td>100GB-TB</td>
</tr>
<tr>
<td><strong>metric</strong></td>
<td>transaction throughput</td>
<td>query throughput, response</td>
</tr>
</tbody>
</table>
Why a Separate Data Warehouse?

- High performance for both systems
  - DBMS—tuned for OLTP: access methods, indexing, concurrency control, recovery
  - Warehouse—tuned for OLAP: complex OLAP queries, multidimensional view, consolidation
- Different functions and different data:
  - **missing data**: Decision support requires historical data which operational DBs do not typically maintain
  - **data consolidation**: DS requires consolidation (aggregation, summarization) of data from heterogeneous sources
  - **data quality**: different sources typically use inconsistent data representations, codes and formats which have to be reconciled
- Note: There are more and more systems which perform OLAP analysis directly on relational databases
Data Warehouse: A Multi-Tiered Architecture

Data Warehouse: A Multi-Tiered Architecture

Data Sources
- Operational DBs
- Other sources

Data Storage
- Data Warehouse
- Extract Transform Load Refresh
- Data Marts

OLAP Engine
- Monitor & Integrator
- OLAP Server
- OLAP Engine

Front-End Tools
- Analysis
- Query
- Reports
- Data mining

Other sources
- Metadata
Chapter 4: Data Warehousing and On-line Analytical Processing

- Data Warehouse: Basic Concepts
- Data Warehouse Modeling: Data Cube and OLAP
- Data Cube Computation: Preliminary Concepts
- Data Cube Computation Methods
- Summary
From Tables and Spreadsheets to Data Cubes

- A data warehouse is based on a multidimensional data model which views data in the form of a data cube.
- A data cube, such as sales, allows data to be modeled and viewed in multiple dimensions.
  - **Dimension tables**, such as item (item_name, brand, type), or time(day, week, month, quarter, year).
  - **Fact table** contains measures (such as dollars_sold) and keys to each of the related dimension tables.
- In data warehousing literature, an n-D base cube is called a base cuboid. The top most 0-D cuboid, which holds the highest-level of summarization, is called the apex cuboid. The lattice of cuboids forms a data cube.
Data cube

- A data cube, such as sales, allows data to be modeled and viewed in multiple dimensions.
- Suppose ALLELETROLNICS create a sales data warehouse with respect to dimensions:
  - Time
  - Item
  - Location
3D Data cube Example

<table>
<thead>
<tr>
<th>location (cities)</th>
<th>Chicago</th>
<th>New York</th>
<th>Toronto</th>
<th>Vancouver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>605</td>
<td>825</td>
<td>14</td>
<td>400</td>
</tr>
<tr>
<td>Q2</td>
<td>680</td>
<td>952</td>
<td>31</td>
<td>512</td>
</tr>
<tr>
<td>Q3</td>
<td>812</td>
<td>1023</td>
<td>30</td>
<td>501</td>
</tr>
<tr>
<td>Q4</td>
<td>927</td>
<td>1038</td>
<td>38</td>
<td>580</td>
</tr>
</tbody>
</table>

item (types)
A data cube, such as sales, allows data to be modeled and viewed in multiple dimensions.

Suppose ALLELETRONICS create a sales data warehouse with respect to dimensions:
- Time
- Item
- Location
- Supplier
4D Data cube Example

- **Supplier = “SUP1”**
  - Location (cities): Chicago, New York, Toronto, Vancouver
  - Time (quarters): Q1, Q2, Q3, Q4
  - Items: Computer, Security, Home, Phone, Entertainment
  - Values: 605, 825, 14, 400

- **Supplier = “SUP2”**
  - Location (cities): Chicago, New York, Toronto, Vancouver
  - Time (quarters): Q1, Q2, Q3, Q4
  - Items: Computer, Security, Home, Phone, Entertainment

- **Supplier = “SUP3”**
  - Location (cities): Chicago, New York, Toronto, Vancouver
  - Time (quarters): Q1, Q2, Q3, Q4
  - Items: Computer, Security, Home, Phone, Entertainment
Cube: A Lattice of Cuboids

0-D (apex) cuboid

1-D cuboids

2-D cuboids

3-D cuboids

4-D (base) cuboid
Conceptual Modeling of Data Warehouses

- Modeling data warehouses: dimensions & measures
  - **Star schema**: A fact table in the middle connected to a set of dimension tables
  - **Snowflake schema**: A refinement of star schema where some dimensional hierarchy is normalized into a set of smaller dimension tables, forming a shape similar to snowflake
  - **Fact constellations**: Multiple fact tables share dimension tables, viewed as a collection of stars, therefore called galaxy schema or fact constellation
Star Schema: An Example

- **Time**
  - time_key
  - day
  - day_of_the_week
  - month
  - quarter
  - year

- **Location**
  - location_key
  - street
  - city
  - state_or_province
  - country

- **Item**
  - item_key
  - item_name
  - brand
  - type
  - supplier_type

- ** Measures**
  - units_sold
  - dollars_sold
  - avg_sales

- **Sales Fact Table**
  - time_key
  - item_key
  - branch_key
  - location_key

- **Branch**
  - branch_key
  - branch_name
  - branch_type

The diagram illustrates a star schema with a central fact table connected to dimension tables for time, location, item, and measures. The arrows indicate the relationships between these tables.
Snowflake Schema: An Example

Sales Fact Table
- time_key
- item_key
- branch_key
- location_key
- units_sold
- dollars_sold
- avg_sales

Measures
- item_key
- item_name
- brand
- type
- supplier_key

Branch
- branch_key
- branch_name
- branch_type

Location
- location_key
- street
- city_key

City
- city_key
- city
- state_or_province
- country

Supplier
- supplier_key
- supplier_type
Fact Constellation: An Example

### Sales Fact Table
- **time_key**
- **item_key**
- **branch_key**
- **location_key**
- **units_sold**
- **dollars_sold**
- **avg_sales**

### Item Table
- **item_key**
- **item_name**
- **brand**
- **type**
- **supplier_type**

### Branch Table
- **branch_key**
- **branch_name**
- **branch_type**

### Location Table
- **location_key**
- **street**
- **city**
- **province_or_state**
- **country**

### Measures
- **time**
- **day**
- **day_of_the_week**
- **month**
- **quarter**
- **year**

### Shipping Fact Table
- **time_key**
- **item_key**
- **shipper_key**
- **from_location**
- **to_location**
- **dollars_cost**
- **units_shipped**

### Shipper Table
- **shipper_key**
- **shipper_name**
- **location_key**
- **shipper_type**
A **Concept Hierarchy** defines a sequence of mappings from a set of low-level concepts to high-level concepts.

Consider a concept hierarchy for the dimension “Location”.
A Concept Hierarchy for a Dimension (location)

- all
- region
- country
- city
- office

- Europe
  - Germany
  - Spain
  - Canada
    - Mexico
  - North_America

- all
  - Vancouver
    - Toronto
  - L. Chan
    - M. Wind
Many concept hierarchies are implicit within the database system.
Concept Hierarchies

- Concept hierarchies may also be defined by grouping values for a given dimension or attribute, resulting in a set-grouping hierarchy.
So, how are *concept hierarchies* useful in OLAP?

In the multidimensional model, data are organized into multiple dimensions,

And each dimension contains multiple levels of abstraction defined by concept hierarchies.
Data Cube Measures: Three Categories

- **Distributive**: if the result derived by applying the function to $n$ aggregate values is the same as that derived by applying the function on all the data without partitioning
  - E.g., count(), sum(), min(), max()

- **Algebraic**: if it can be computed by an algebraic function with $M$ arguments (where $M$ is a bounded integer), each of which is obtained by applying a distributive aggregate function
  - E.g., avg(), min_N(), standard_deviation()

- **Holistic**: if there is no constant bound on the storage size needed to describe a subaggregate.
  - E.g., median(), mode(), rank()
Multidimensional Data

- Sales volume as a function of product, month, and region

Dimensions: Product, Location, Time
Hierarchical summarization paths

- Industry
- Region
- Year
  - Category
  - Country
  - Quarter
  - Product
  - City
  - Month
  - Office
  - Week
  - Day
  - Month
  - Region
  - Year
### A Sample Data Cube

**Total annual sales of TVs in U.S.A.**

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>1Qtr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2Qtr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3Qtr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4Qtr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Country**

- U.S.A
- Canada
- Mexico

**Summarized view**

- All, All, All
Cuboids Corresponding to the Cube

0-D \((apex)\) cuboid

1-D cuboids

2-D cuboids

3-D \((base)\) cuboid
Typical OLAP Operations

- Roll up (drill-up): summarize data
  - by climbing up hierarchy or by dimension reduction
- Drill down (roll down): reverse of roll-up
  - from higher level summary to lower level summary or detailed data, or introducing new dimensions
- Slice and dice: project and select
- Pivot (rotate):
  - reorient the cube, visualization, 3D to series of 2D planes
- Other operations
  - drill across: involving (across) more than one fact table
  - drill through: through the bottom level of the cube to its back-end relational tables (using SQL)
Fig. 3.10 Typical OLAP Operations
Cube Operators for Roll-up

- **sale(s1,*,*)**
- **sale(s2,p2,*)**
- **sale(*,*,*)**

**Day 1**

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>12</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**Day 2**

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>44</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**Sum**

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>67</td>
<td>12</td>
<td>50</td>
</tr>
</tbody>
</table>

**Total**

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>110</td>
</tr>
<tr>
<td>p2</td>
<td>19</td>
</tr>
</tbody>
</table>
# Extended Cube

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>56</td>
<td>4</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td>50</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>19</td>
<td>53</td>
<td>129</td>
</tr>
</tbody>
</table>

**Day 1**

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>12</td>
<td>50</td>
<td>62</td>
<td>48</td>
</tr>
<tr>
<td>p2</td>
<td>11</td>
<td>8</td>
<td>19</td>
<td>48</td>
</tr>
<tr>
<td>*</td>
<td>23</td>
<td>8</td>
<td>50</td>
<td>81</td>
</tr>
</tbody>
</table>

**Day 2**

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>44</td>
<td>4</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>44</td>
<td>4</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>67</td>
<td>19</td>
<td>53</td>
<td>129</td>
</tr>
</tbody>
</table>

**Sale**

sale(*,p2,*)
Aggregation Using Hierarchies

(store s1 in Region A; stores s2, s3 in Region B)
Slicing

TIME = day 1
### Slicing & Pivoting

#### Sales ($ millions)

<table>
<thead>
<tr>
<th>Products</th>
<th>Store s1</th>
<th>Store s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronics</td>
<td>$5.2</td>
<td>$8.9</td>
</tr>
<tr>
<td>Toys</td>
<td>$1.9</td>
<td>$0.75</td>
</tr>
<tr>
<td>Clothing</td>
<td>$2.3</td>
<td>$4.6</td>
</tr>
<tr>
<td>Cosmetics</td>
<td>$1.1</td>
<td>$1.5</td>
</tr>
</tbody>
</table>

#### Sales ($ millions)

<table>
<thead>
<tr>
<th>Products</th>
<th>d1</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Electronics</td>
<td>$5.2</td>
<td>$8.9</td>
</tr>
<tr>
<td>Toys</td>
<td>$1.9</td>
<td>$0.75</td>
</tr>
<tr>
<td>Clothing</td>
<td>$2.3</td>
<td>$4.6</td>
</tr>
<tr>
<td>Cosmetics</td>
<td>$1.1</td>
<td>$1.5</td>
</tr>
<tr>
<td>Store s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Electronics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clothing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cosmetics</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5: Data Cube Technology

- Data Warehouse: Basic Concepts
- Data Warehouse Modeling: Data Cube and OLAP
- Data Cube Computation: Preliminary Concepts
- Data Cube Computation Methods
Data Cube: A Lattice of Cuboids

0-D(apex) cuboid

1-D cuboids

2-D cuboids

3-D cuboids

4-D(base) cuboid
Data Cube: A Lattice of Cuboids

- Base vs. aggregate cells; ancestor vs. descendant cells; parent vs. child cells
  1. (9/15, milk, Urbana, Dairy_land)
  2. (9/15, milk, Urbana, *)
  3. (*, milk, Urbana, *)
  4. (*, milk, Urbana, *)
  5. (*, milk, Chicago, *)
  6. (*, milk, *, *)

0-D(apex) cuboid
1-D cuboids
2-D cuboids
3-D cuboids
4-D(base) cuboid
Cube Materialization: Full Cube vs. Iceberg Cube

- Full cube vs. iceberg cube
  
  ```sql
  compute cube sales iceberg as
  select month, city, customer group, count(*)
  from salesInfo
  cube by month, city, customer group
  having count(*) >= min support
  ```

- Compute *only* the cuboid cells whose measure satisfies the iceberg condition
- Only a small portion of cells may be “above the water” in a sparse cube
- Ex.: Show only those cells whose count is no less than 100
Why Iceberg Cube?

- Advantages of computing iceberg cubes
  - No need to save nor show those cells whose value is below the threshold (iceberg condition)
  - Efficient methods may even avoid computing the un-needed, intermediate cells
  - Avoid explosive growth

- Example: A cube with 100 dimensions
  - Suppose it contains only 2 base cells: \{(a_1, a_2, a_3, \ldots, a_{100}), (a_1, a_2, b_3, \ldots, b_{100})\}
  - How many aggregate cells if “having count >= 1”? 
    - Answer: \(2^{101} - 4\)
  - What about the iceberg cells, (i.e., with condition: “having count >= 2”)?
    - Answer: 4
Is Iceberg Cube Good Enough? Closed Cube & Cube Shell

- Let cube P have only 2 base cells: \{ (a_1, a_2, a_3 \ldots, a_{100}): 10, (a_1, a_2, b_3, \ldots, b_{100}): 10 \}
  - How many cells will the iceberg cube contain if “having count(*) ≥ 10”?  
    - Answer: $2^{101} - 4$ (still too big!)

- Close cube:
  - A cell $c$ is **closed** if there exists no cell $d$, such that $d$ is a descendant of $c$, and $d$ has the same measure value as $c$
    - Ex. The same cube P has only 3 closed cells:
      - $\{ (a_1, a_2, *, \ldots, *): 20, (a_1, a_2, a_3 \ldots, a_{100}): 10, (a_1, a_2, b_3, \ldots, b_{100}): 10 \}$
  - A **closed cube** is a cube consisting of only closed cells

- Cube Shell: The cuboids involving only a small # of dimensions, e.g., 2
  - Idea: Only compute cube shells, other dimension combinations can be computed on the fly

Q: For $(A_1, A_2, \ldots A_{100})$, how many combinations to compute?
General Heuristics (Agarwal et al. VLDB’96)

- Sorting, hashing, and grouping operations are applied to the dimension attributes in order to reorder and cluster related tuples.
- Aggregates may be computed from previously computed aggregates, rather than from the base fact table.
  - **Smallest-child**: computing a cuboid from the smallest, previously computed cuboid.
  - **Cache-results**: caching results of a cuboid from which other cuboids are computed to reduce disk I/Os.
  - **Amortize-scans**: computing as many as possible cuboids at the same time to amortize disk reads.
  - **Share-sorts**: sharing sorting costs across multiple cuboids when sort-based method is used.
  - **Share-partitions**: sharing the partitioning cost across multiple cuboids when hash-based algorithms are used.
Chapter 5: Data Cube Technology

- Data Cube Computation: Preliminary Concepts
- Data Cube Computation Methods
- Processing Advanced Queries by Exploring Data Cube Technology
- Multidimensional Data Analysis in Cube Space
- Summary
Data Cube Computation Methods

- Multi-Way Array Aggregation
- BUC
Multi-Way Array Aggregation

- Array-based “bottom-up” algorithm
- Using multi-dimensional chunks
- No direct tuple comparisons
- Simultaneous aggregation on multiple dimensions
- Intermediate aggregate values are re-used for computing ancestor cuboids
- Cannot do Apriori pruning: No iceberg optimization
Multi-way Array Aggregation for Cube Computation (MOLAP)

- Partition arrays into chunks (a small subcube which fits in memory).
- Compressed sparse array addressing: (chunk_id, offset)
- Compute aggregates in “multiway” by visiting cube cells in the order which minimizes the # of times to visit each cell, and reduces memory access and storage cost.

What is the best traversing order to do multi-way aggregation?
Multi-way Array Aggregation for Cube Computation (3-D to 2-D)

The best order is the one that minimizes the memory requirement and reduced I/Os
Multi-way Array Aggregation for Cube Computation (2-D to 1-D)
Multi-Way Array Aggregation for Cube Computation (Method Summary)

- Method: the planes should be sorted and computed according to their size in ascending order
  - Idea: keep the smallest plane in the main memory, fetch and compute only one chunk at a time for the largest plane
- Limitation of the method: computing well only for a small number of dimensions
  - If there are a large number of dimensions, “top-down” computation and iceberg cube computation methods can be explored
Data Cube Computation Methods

- Multi-Way Array Aggregation
- BUC
Bottom-Up Computation (BUC)

- BUC (Beyer & Ramakrishnan, SIGMOD'99)
- Bottom-up cube computation
  (Note: top-down in our view!)
- Divides dimensions into partitions and facilitates iceberg pruning
  - If a partition does not satisfy $min\_sup$, its descendants can be pruned
  - If $minsup = 1 \Rightarrow$ compute full CUBE!
- No simultaneous aggregation
BUC: Partitioning

- Usually, entire data set can’t fit in main memory
- Sort *distinct* values
  - partition into blocks that fit
- Continue processing
- Optimizations
  - Partitioning
    - External Sorting, Hashing, Counting Sort
  - Ordering dimensions to encourage pruning
    - Cardinality, Skew, Correlation
  - Collapsing duplicates
    - Can’t do holistic aggregates anymore!
Attribute-Based Data Warehouse

- Problem
  - Data Warehouse
  - NP-hardness
- Algorithm
- Performance Study
Parts are bought from suppliers and then sold to customers at a sale price $SP$

<table>
<thead>
<tr>
<th>part</th>
<th>supplier</th>
<th>customer</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>s1</td>
<td>c1</td>
<td>4</td>
</tr>
<tr>
<td>p3</td>
<td>s1</td>
<td>c2</td>
<td>3</td>
</tr>
<tr>
<td>p2</td>
<td>s3</td>
<td>c1</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Parts are bought from suppliers and then sold to customers at a sale price SP.
Parts are bought from suppliers and then sold to customers at a sale price SP.

### Table T

<table>
<thead>
<tr>
<th>part</th>
<th>supplier</th>
<th>customer</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>s1</td>
<td>c1</td>
<td>4</td>
</tr>
<tr>
<td>p3</td>
<td>s1</td>
<td>c2</td>
<td>3</td>
</tr>
<tr>
<td>p2</td>
<td>s3</td>
<td>c1</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Query 1

- **SELECT** part, customer, SUM(SP)
- **FROM** table T
- **GROUP BY** part, customer

<table>
<thead>
<tr>
<th>part</th>
<th>customer</th>
<th>SUM(SP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>c1</td>
<td>4</td>
</tr>
<tr>
<td>p3</td>
<td>c2</td>
<td>3</td>
</tr>
<tr>
<td>p2</td>
<td>c1</td>
<td>7</td>
</tr>
</tbody>
</table>

### Query 2

- **SELECT** customer, SUM(SP)
- **FROM** table T
- **GROUP BY** customer

<table>
<thead>
<tr>
<th>customer</th>
<th>SUM(SP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>11</td>
</tr>
<tr>
<td>c2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Average**, **Maximum**, **Minimum**, **...**
Parts are bought from suppliers and then sold to customers at a sale price \( SP \).
Suppose we materialize all views. This wastes a lot of space.

Cost for accessing pc = 4M
Cost for accessing ps = 0.8M
Cost for accessing sc = 2M
Cost for accessing p = 0.2M
Cost for accessing s = 0.01M
Cost for accessing c = 0.1M
Suppose we materialize the top view only.

Cost for accessing pc = 6M (not 4M)

Cost for accessing ps = 6M (not 0.8M)

Cost for accessing sc = 6M (not 2M)

Cost for accessing p = 6M (not 0.2M)

Cost for accessing s = 6M (not 0.01M)

Cost for accessing c = 6M (not 0.1M)
Suppose we materialize the top view and the view for “ps” only.

- Cost for accessing pc = 6M (still 6M)
- Cost for accessing p = 0.8M (not 6M previously)
- Cost for accessing s = 0.8M (not 6M previously)
- Cost for accessing c = 6M (still 6M)
- Cost for accessing sc = 6M (still 6M)
- Cost for accessing ps = 0.8M (not 6M previously)
- Cost for accessing ps = 0.8M (not 6M previously)
Suppose we materialize the top view and the view for “ps” only.

<table>
<thead>
<tr>
<th>Accessing</th>
<th>Cost</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>6M</td>
<td>0</td>
</tr>
<tr>
<td>ps</td>
<td>0.8M</td>
<td>5.2M</td>
</tr>
<tr>
<td>sc</td>
<td>2M</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>0.2M</td>
<td>5.2M</td>
</tr>
<tr>
<td>s</td>
<td>0.01M</td>
<td>5.2M</td>
</tr>
<tr>
<td>c</td>
<td>0.1M</td>
<td>0</td>
</tr>
</tbody>
</table>

Selective Materialization Problem:
We can select a set V of k views such that 
Gain(V U {top view}, {top view}) is maximized.

Gain({view for “ps”, top view}, {top view}) = 5.2*3 = 15.6
Attribute-Based Data Warehouse

- Problem
  - Date Warehouse
- Algorithm
Greedy Algorithm

- $k =$ number of views to be materialized

- Given
  - $v$ is a view
  - $S$ is a set of views which are selected to be materialized

- Define the benefit of selecting $v$ for materialization as
  - $B(v, S) = \text{Gain}(S \cup v, S)$
Greedy Algorithm

- \( S \leftarrow \{\text{top view}\}; \)
- For \( i = 1 \) to \( k \) do
  - Select that view \( v \) not in \( S \) such that \( B(v, S) \) is maximized;
  - \( S \leftarrow S \cup \{v\} \)
- Resulting \( S \) is the greedy selection
Benefit from pc = 6M-6M = 0

<table>
<thead>
<tr>
<th>Benefit</th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Benefit from ps = 6M - 0.8M = 5.2M

<table>
<thead>
<tr>
<th>Benefit</th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

k = 2
Benefit from sc = 6M-6M= 0

<table>
<thead>
<tr>
<th>Benefit</th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.1 Data Cube

**Benefit from p = 6M - 0.2M = 5.8M**

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>5.8 x 1 = 5.8</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(k = 2\)
1.1 Data Cube

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>5.8 x 1 = 5.8</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>5.99 x 1 = 5.99</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Benefit from s = 6M - 0.01M = 5.99M

$k = 2$
### Benefit from c = 6M - 0.1M = 5.9M

- **Benefit Table**

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pc</strong></td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td><strong>ps</strong></td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td><strong>sc</strong></td>
<td>0 x 3 = 0</td>
<td></td>
</tr>
<tr>
<td><strong>p</strong></td>
<td>5.8 x 1 = 5.8</td>
<td></td>
</tr>
<tr>
<td><strong>s</strong></td>
<td>5.99 x 1 = 5.99</td>
<td></td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>5.9 x 1 = 5.9</td>
<td></td>
</tr>
</tbody>
</table>

**k = 2**
1.1 Data Cube

| Benefit from pc = 6M-6M = 0 |

<table>
<thead>
<tr>
<th>Benefit</th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>$0 \times 3 = 0$</td>
<td>$0 \times 2 = 0$</td>
</tr>
<tr>
<td>ps</td>
<td>$5.2 \times 3 = 15.6$</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>$0 \times 3 = 0$</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>$5.8 \times 1 = 5.8$</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>$5.99 \times 1 = 5.99$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$5.9 \times 1 = 5.9$</td>
<td></td>
</tr>
</tbody>
</table>

k = 2
<table>
<thead>
<tr>
<th>Benefit</th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td>0 x 2 = 0</td>
</tr>
<tr>
<td>ps</td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>0 x 3 = 0</td>
<td>0 x 2 = 0</td>
</tr>
<tr>
<td>p</td>
<td>5.8 x 1 = 5.8</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>5.99 x 1 = 5.99</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>5.9 x 1 = 5.9</td>
<td></td>
</tr>
</tbody>
</table>

Benefit from sc = 6M-6M = 0

k = 2
1.1 Data Cube

Benefit from $p = 0.8M - 0.2M = 0.6M$

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>$0 \times 3 = 0$</td>
<td>$0 \times 2 = 0$</td>
</tr>
<tr>
<td>ps</td>
<td>$5.2 \times 3 = 15.6$</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>$0 \times 3 = 0$</td>
<td>$0 \times 2 = 0$</td>
</tr>
<tr>
<td>p</td>
<td>$5.8 \times 1 = 5.8$</td>
<td>$0.6 \times 1 = 0.6$</td>
</tr>
<tr>
<td>s</td>
<td>$5.99 \times 1 = 5.99$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$5.9 \times 1 = 5.9$</td>
<td></td>
</tr>
</tbody>
</table>
1.1 Data Cube

Benefit from s = 0.8M - 0.01M = 0.79M

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0 x 3 = 0</td>
<td>0 x 2 = 0</td>
</tr>
<tr>
<td>ps</td>
<td>5.2 x 3 = 15.6</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>0 x 3 = 0</td>
<td>0 x 2 = 0</td>
</tr>
<tr>
<td>p</td>
<td>5.8 x 1 = 5.8</td>
<td>0.6 x 1 = 0.6</td>
</tr>
<tr>
<td>s</td>
<td>5.99 x 1 = 5.99</td>
<td>0.79 x 1 = 0.79</td>
</tr>
<tr>
<td>c</td>
<td>5.9 x 1 = 5.9</td>
<td></td>
</tr>
</tbody>
</table>

k = 2
### 1.1 Data Cube

<table>
<thead>
<tr>
<th>Benefit from $c$</th>
<th>$6M - 0.1M = 5.9M$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc 6M</td>
<td>0 x 3 = 0</td>
</tr>
<tr>
<td>pc 0.2M</td>
<td>0 x 2 = 0</td>
</tr>
<tr>
<td>ps 0.8M</td>
<td>5.2 x 3 = 15.6</td>
</tr>
<tr>
<td>sc 6M</td>
<td>0 x 3 = 0</td>
</tr>
<tr>
<td>sc 0.1M</td>
<td>0 x 2 = 0</td>
</tr>
<tr>
<td>p 0.2M</td>
<td>5.8 x 1 = 5.8</td>
</tr>
<tr>
<td>p 0.01M</td>
<td>0.6 x 1 = 0.6</td>
</tr>
<tr>
<td>s 0.01M</td>
<td>5.99 x 1 = 5.99</td>
</tr>
<tr>
<td>s 0.01M</td>
<td>0.79 x 1 = 0.79</td>
</tr>
<tr>
<td>c 0.1M</td>
<td>5.9 x 1 = 5.9</td>
</tr>
<tr>
<td>c 0.1M</td>
<td>5.9 x 1 = 5.9</td>
</tr>
</tbody>
</table>

$k = 2$
Two views to be materialized are
1. ps
2. c

V = {ps, c}
Gain(V U {top view}, {top view})
= 15.6 + 5.9 = 21.5
Performance Study

- How bad does the Greedy Algorithm perform?
1.1 Data Cube

- Benefit from b = 200 - 100 = 100

- Benefit: 200 - 100 = 100

<table>
<thead>
<tr>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b 41 x 100= 4100</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
1.1 Data Cube

Benefit from c = 200 - 99 = 101

Benefit

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>41 x 100 = 4100</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>41 x 101 = 4141</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Data Cube

- **a**: 200
- **b**: 100
- **c**: 99
- **d**: 100

#### Benefit

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>b</strong></td>
<td>41 x 100 = 4100</td>
<td></td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>41 x 101 = 4141</td>
<td></td>
</tr>
<tr>
<td><strong>d</strong></td>
<td>41 x 100 = 4100</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Notes

- **k = 2**

- 20 nodes

- None: 1

- \( p_1 \): 97
- \( q_1 \): 97
- \( r_1 \): 97
- \( s_1 \): 97
- \( p_{20} \): 97
- \( q_{20} \): 97
- \( r_{20} \): 97
- \( s_{20} \): 97

- Benefit calculation:
  - \( 41 \times 100 = 4100 \)
  - \( 41 \times 101 = 4141 \)
**Division 1: Data Cube**

For node **b**, the benefit is calculated as:

\[ \text{Benefit from } b = 200 - 100 = 100 \]

**Table: Benefit from Division 1**

<table>
<thead>
<tr>
<th>Node</th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>41 x 100 = 4100</td>
<td>21 x 100 = 2100</td>
</tr>
<tr>
<td>c</td>
<td>41 x 101 = 4141</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>41 x 100 = 4100</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
**Greedy:**

\[ V = \{b, c\} \]

\[ \text{Gain}(V \cup \{\text{top view}\}, \{\text{top view}\}) = 4141 + 2100 = 6241 \]
### Greedy:

- **V = \{b, c\}**
- Gain(V U \{top view\}, \{top view\})
  
  \[
  = 4141 + 2100 = 6241
  \]

### Optimal:

- **V = \{b, d\}**
- Gain(V U \{top view\}, \{top view\})
  
  \[
  = 4100 + 4100 = 8200
  \]

---

<table>
<thead>
<tr>
<th></th>
<th>1st Choice (M)</th>
<th>2nd Choice (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>41 x 100 = 4100</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>41 x 101 = 4141</td>
<td>21 x 101 + 20 x 1 = 2141</td>
</tr>
<tr>
<td>d</td>
<td>41 x 100 = 4100</td>
<td>41 x 100 = 4100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**k = 2**
If this ratio = 1, Greedy can give an optimal solution. If this ratio ≈ 0, Greedy may give a “bad” solution.

Greedy:

\[ V = \{b, c\} \]
\[ \text{Gain}(V \cup \{\text{top view}\}, \{\text{top view}\}) = 4141 + 2100 = 6241 \]

Optimal:

\[ V = \{b, d\} \]
\[ \text{Gain}(V \cup \{\text{top view}\}, \{\text{top view}\}) = 4100 + 4100 = 8200 \]

\[
\frac{\text{Greedy}}{\text{Optimal}} = \frac{6241}{8200} = 0.7611
\]

Does this ratio has a “lower” bound?

It is proved that this ratio is at least 0.63.
Performance Study

- This is just an example to show that this greedy algorithm can perform badly.
- A complete proof of the lower bound can be found in the paper.
Summary

- **Data warehousing:** A multi-dimensional model of a data warehouse
  - A data cube consists of *dimensions & measures*
  - Star schema, snowflake schema, fact constellations
  - **OLAP** operations: drilling, rolling, slicing, dicing and pivoting

- **Data Warehouse Architecture, Design, and Usage**
  - Multi-tiered architecture
  - Business analysis design framework
  - Information processing, analytical processing, data mining, **OLAM** (Online Analytical Mining)

- **Implementation:** Efficient computation of data cubes
  - Partial vs. full vs. no materialization
  - Indexing OALP data: Bitmap index and join index
  - OLAP query processing
  - OLAP servers: ROLAP, MOLAP, HOLAP

- **Data generalization:** Attribute-oriented induction
S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. VLDB’96

D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance in data warehouses. SIGMOD’97

R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. ICDE’97


V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. SIGMOD’96
References (II)

- P. O'Neil and D. Quass. Improved query performance with variant indexes. SIGMOD'97
- A. Shoshani. OLAP and statistical databases: Similarities and differences. PODS'00.
- S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. ICDE'94
- J. Widom. Research problems in data warehousing. CIKM'95.