Introduction to Database Systems

CSE 444

Lecture #2
Jan 8 2001

Enrollment Closed

Announcement: Homework

HW#1 is being handed out
Due: Wed Jan 17
Requires use of SQL Server
Homework is individual work
Even when you are asked to share an account
No late submission
You will lose entire credit
In the future, we will only announce availability of homework/solutions
Download from the website

Announcement: Course Project

Goal: Build end to end database application with web front-end
Tasks
Find a database application
Model the data and define application requirements
Design and implement relational schema
Populate database
Build a web-based front end
Your application should be nontrivial
Sample applications and other details available in course web pages

Announcement: Course Project

Group Project
Important: Must work in Team of 3
Each member must have well-defined contribution
Send yanali@cs team information ASAP
Latest by Jan 12 by email
Stages
Formation of Group
Informal Proposal and ASP Programming
Formal design (graded)
Project Report (graded)
Interview and Demo (graded)
March 7, 9
Requires significant design and implementation
Start now!
Get familiar with software

The Relational Data Model

Reading: 3.1, 3.5.1-3.5.3
Data Models

A **data model** is a collection of concepts for describing data. The **relational model of data** is the most widely used model today.

- Main concept: **relation**, basically a table with rows and columns.
- Every relation has a **schema**, which describes the columns, or fields.

The Relational Data Model

- **Database Model**: (ODL, E/R)
- **Relational Schema**: ODL definitions, Diagrams (E/R)
- **Physical storage**: Complex file organization and index structures.

Terminology

<table>
<thead>
<tr>
<th>Products</th>
<th>Attribute names</th>
<th>Name</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tuple</td>
<td>gizmo</td>
<td>$19.99</td>
<td>gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td></td>
<td>Tuple</td>
<td>Power gizmo</td>
<td>$29.99</td>
<td>gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td></td>
<td>Tuple</td>
<td>SingleTouch</td>
<td>$149.99</td>
<td>photography</td>
<td>Canon</td>
</tr>
<tr>
<td></td>
<td>Tuple</td>
<td>MultiTouch</td>
<td>$203.99</td>
<td>household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Domains

- Each attribute has a type
- Must be atomic type called **domain**
- Examples:
  - Integer
  - String
  - Real
  - …

Schemas

- **Relational Schema**: Relation name plus attribute names
  - E.g. Product(Name, Price, Category, Manufacturer)
  - In practice we add the domain for each attribute

- **Database Schema**
  - Set of relational schemas
  - E.g. Product(Name, Price, Category, Manufacturer)
  - Vendor(Name, Address, Phone)

Instances

- An instance of a relational schema R(A1, ..., Ak), is a relation with k attributes with values of corresponding domains
- An instance of a database schema R1(...), R2(...), ..., Rn(...), consists of n relations, each an instance of the corresponding relational schema.
Example

Relational schema: Product(Name, Price, Category, Manufacturer)
Instance:

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>$19.99</td>
<td>gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Power gizmo</td>
<td>$29.99</td>
<td>gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Schemas and Instances

Analogy with programming languages:
- Schema = type
- Instance = value

Important distinction:
- Database Schema = stable over long periods of time
- Database Instance = changes constantly, as data is inserted/updated/deleted

Integrity Constraints (ICs)

IC: condition that must be true for any instance of the database; e.g., domain constraints.
- ICs are specified when schema is defined.
- ICs are checked when relations are modified.

A legal instance of a relation is one that satisfies all specified ICs.
- DBMS should allow only legal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
- Avoids many data entry errors, too!

Keys

Examples:
- “For a given student and course, there is a single grade.”
- “No two students have the same sid and no two students have the same login. Furthermore, any other table wishing to reference a student should reference the sid field if possible.”

Foreign Keys

Only students listed in the Students relation should be allowed to enroll for courses.

Foreign Keys, Referential Integrity

Foreign key: Set of fields in one relation that is used to ‘refer’ to a tuple in another relation.
- Must correspond to primary key of the second relation
- Like a ‘logical pointer’
- If all foreign key constraints are enforced, referential integrity is achieved
- No dangling references
Integrity Constraints and Semantics

- ICs are based upon the semantics of the real-world enterprise.
- We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.
- Key and foreign key ICs are the most common; more general ICs supported too.

Relational Operators and Relational Algebra

Reading: 4.1, 4.5-4.8

Set-Oriented Operations: Relational Algebra

- Basic operations:
  - Selection ( ) Selects a subset of rows
  - Projection ( ) Deletes unwanted columns
  - Set-difference ( ) Tuples in reln. 1, but not in reln. 2.
  - Union ( ) Tuples in reln. 1 and in reln. 2.
  - Cross-product ( ) Allows us to combine two relations
- Since each operation returns a relation, operations can be composed! (Algebra is "closed")

Example

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>38</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

`Sailors` and `Reserves` relations for our examples.
Assume that names of fields in query results are "inherited" from names of fields in query input relations.

Projection

- Retains only attributes that are in projection list.
- Schema of result contains exactly the fields in the projection list, with the same names that they had in the input relation.
- Projection operator has to eliminate duplicates! (Why??)
- Note: real systems typically don’t do duplicate elimination unless the user explicitly asks for it. (Why not?)

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>puppy</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

σ `rating > 8` (S2)

Selection

- Selects rows that satisfy selection condition.
- No duplicates in result! (Why?)
- Schema of result identical to schema of input relation.
- Result relation can be the input for another relational algebra operation! (Operator composition)

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>puppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

σ `rating > 8` (S2)

π `sname, rating` (σ `rating > 8` (S2))
Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be union-compatible:
  - Same number of fields.
  - 'Corresponding' fields have the same type.
- What is the schema of result? \( S_1 \cup S_2 \)
- \( S_1 \setminus S_2 \)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Cross-Product

- Each row of \( S_1 \) is paired with each row of \( R_1 \).

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
</tr>
</tbody>
</table>

Joins

- **Condition Join**: \( R \bowtie_c S = \sigma_c (R \times S) \)
- **Equi-Join**: A special case of condition join where the condition \( c \) contains only equalities

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- Result schema similar to cross-product, but only one copy of fields for which equality is specified.
- **Natural Join**: Equi-join on all common fields (fields with the same name).

Example of Composition and Equivalence

- Find names of sailors who've reserved boat #103
  - Solution:
    \[ \pi_{\text{name}} (\sigma_{\text{bid}=103} \left( \text{Sailors} \bowtie \text{Reserves} \right)) \]
  
  A more efficient solution:
  \[ \pi_{\text{name}} (\sigma_{\text{bid}=103} \left( \text{Reserves} \bowtie \text{Sailors} \right)) \]

SQL

- Information about boat color only available in Boats; so need an extra join:
  \[ \pi_{\text{name}} (\sigma_{\text{color}=\text{red}} \left( \text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors} \right)) \]
- A more efficient solution:
  \[ \pi_{\text{name}} (\pi_{\text{sid}} (\sigma_{\text{bid}=103, \text{color}=\text{red}} \left( \text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors} \right))) \]

- Reading: Sec 5 (all subsections, except 5.10)
Why yet another Language?

- Built-in support for set-oriented retrieval of data from a “large” database.
- Query Languages != programming languages!
  - QLs not expected to be “Turing complete”
  - QLs not intended to be used for complex computation

Basic SQL Query

- relation-list: A list of relation names (possibly with a range-variable after each name).
- target-list: A list of attributes of relations in relation-list
- qualification: Comparisons: Attr op const or Attr1 op Attr2, where op is one of =, !=, >=, <=, <, >
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates.
  - Default is that duplicates are not eliminated!

Selections

Company(sticker, name, country, stockPrice)

Find all US companies whose stock is > 50:

```sql
SELECT * FROM Company WHERE country="USA" AND stockPrice > 50
```

Output schema: R(sticker, name, country, stockPrice)

The LIKE operator

- LIKE p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Company(sticker, name, address, country, stockPrice)

Find all US companies whose address has prefix "Mountain":

```sql
SELECT * FROM Company WHERE country="USA" AND address LIKE "Mountain%"
```

Projections

Select only a subset of the attributes

```sql
SELECT name, stockPrice FROM Company WHERE country="USA" AND stockPrice > 50
```

Input schema: Company(sticker, name, country, stockPrice)

Output schema: R(name, stock price)
**Projections**

Rename the attributes in the resulting table

```
SELECT  name AS company, stockprice AS price
FROM    Company
WHERE   country="USA" AND stockPrice > 50
```

Input schema: Company(sticker, name, country, stockPrice)
Output schema: R(company, price)

**Ordering the Results**

```
SELECT  name, stockPrice
FROM    Company
WHERE   country="USA" AND stockPrice > 50
ORDERBY country, name
```

Ordering is ascending, unless you specify the DESC keyword.

Ties are broken by the second attribute on the ORDERBY list, etc.

**Removing Duplicates**

```
SELECT  DISTINCT category
FROM    Product
WHERE   price > 100
```

**Aggregation**

```
SELECT  Sum(price)
FROM    Product
WHERE   maker="Toyota"
```

SQL supports several aggregation operations:

- SUM, MIN, MAX, AVG, COUNT

**Aggregation: Count**

Except COUNT, all aggregations apply to a single attribute

```
SELECT  Count(*)
FROM    Product
WHERE   year > 1995
```

Better:

```
SELECT  Count(DISTINCT name, category)
FROM    Product
WHERE   year > 1995
```
### Simple Aggregation

**Purchase** (product, date, price, quantity)

**Example 1:** find total sales for the entire database

```sql
SELECT Sum(price * quantity)
FROM Purchase
```

**Example 1': find total sales of bagels**

```sql
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

### Grouping and Aggregation

Usually, we want aggregations on certain parts of the relation.

**Purchase** (product, date, price, quantity)

**Example 2:** find total sales after 9/1 per product.

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '9/1'
GROUPBY product
```

### Grouping and Aggregation

1. Compute the relation (i.e., the FROM and WHERE).
2. Group by the attributes in the GROUPBY
3. Select one tuple for every group (and apply aggregation)

SELECT can have (1) grouped attributes or (2) aggregates.

### First compute the relation

(\(\text{date} > "9/1"\)) then group by product:

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
</tbody>
</table>

### Then, aggregate

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>$29.75</td>
</tr>
<tr>
<td>Banana</td>
<td>$12.48</td>
</tr>
</tbody>
</table>

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '9/1'
GROUPBY product
```

### Example

<table>
<thead>
<tr>
<th>Product</th>
<th>SumSales</th>
<th>MaxQuantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>$12.48</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>$29.75</td>
<td>20</td>
</tr>
</tbody>
</table>

For every product, what is the total sales and max quantity sold?

```sql
SELECT product, Sum(price*quantity) AS SumSales, Max(quantity) AS MaxQuantity
FROM Purchase
GROUPBY product
```
Example

```
SELECT name, max(stockPrice)
FROM Company
WHERE country="USA" AND stockPrice > 50
GROUP BY name
HAVING Min(stockprice) > 25
```

- Partition by stockname
- One aggregation per partition
- Number of output tuples = Number of Partitions