CS 457 Database Management Systems

Lecture 15 Midterm Exam Preview
Announcements

• Please, keep silent in the class
  • Distracting other students...
  • Use your own best judgement

• Programming assignment
  • Again, standard input, demo?
  • [http://www.ee.surrey.ac.uk/Teaching/Unix/unix3.html](http://www.ee.surrey.ac.uk/Teaching/Unix/unix3.html)

• In the Exam
  • If you check your cellphone...
    • I wouldn’t know whether you checked your email or do something not allowed in the exam
Overview

• Location: SEM 101
• Date: 3/15, Thursday
• Time: 4:35pm – 5:45pm
• Close book, no paper, no book, no cellphone
  • We’ll project a clock on the screen
• Format:
  • 22 questions
    • Multiple choice (7)
    • Short answers (15)
• Topics (next page)
Topics

Slides on:
- Relational models
- SQL
- Relational Algebra
- Entity-Relationship Diagram
- Design Theory
- No architecture/storage
- No indexing
- No execution
Topics

• Relational models
  • SQL
  • Relational Algebra
  • Entity-Relationship Diagram
  • Design Theory
• No architecture/storage
• No indexing
• No execution
Data Models

• What are data models?
  • A data model is a general, conceptual way of structuring data
  • Think about it...

• Data models studied in this course:
  • Relational data model (data->relation)
  • Semi-structured data model (XML)
Data Models

• Other popular models:
  • Key-value stores (e.g., NoSQL)
  • Graph data model
  • Object-oriented

• A data model describes both
  • The data
  • And the query language
Data Models

• Schema vs. Instance
  • Schema: the structure of a particular database under a certain data model
  • Instance: the actual data
The Relational Data Model

• Database schema
  • "table name" or "relation name"
  • "column name" or "attribute name"
  • each attribute has a "type"

• Degree (or arity) of relation: number of attributes
The Relational Data Model

• Database instance:
  • "table" or "relation"
  • "column" or "attribute" or "field"
  • "row" or "tuple" or "record"

• Cardinality of relation instance: number of tuples
More about tables

• NOT ordered
  • They represent sets or bags

• NOT prescribe how they should be implemented
  • PHYSICAL DATA INDEPENDENCE!

• FLAT
  • all attributes are base types
Topics

• Relational models
• SQL
• Relational Algebra
• Entity-Relationship Diagram
• Design Theory
• No architecture/storage
• No indexing
• No execution
Basic SQL

• Get familiar with the SQL script for Lecture 3
Altering a table in SQL

```sql
ALTER TABLE Company ADD ceo varchar(20);
SELECT * FROM Company;

UPDATE Company SET ceo='Brown' WHERE cname = 'Canon';

SELECT * FROM Company;
```
(Inner) joins

Company(cname, country)
Product(pname, price, category, manufacturer)
– manufacturer is foreign key

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country = 'USA' AND category = 'gadget' AND manufacturer = cname
```
Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include both left and right tuples even if there’s no match
Simple Aggregations

Five basic aggregate operations in SQL

```sql
select count(*) from Purchase
select sum(quantity) from Purchase
select avg(price) from Purchase
select max(quantity) from Purchase
select min(quantity) from Purchase
```

Except count, all aggregations apply to a single attribute
COUNT applies to duplicates, unless otherwise stated:

```
SELECT Count(product) FROM Purchase WHERE price > 4.99
```

same as Count(*) if no nulls

We probably want:

```
SELECT Count(DISTINCT product) FROM Purchase WHERE price > 4.99
```
Ordering Results

```sql
SELECT product, sum(price*quantity) as rev
FROM purchase
GROUP BY product
ORDER BY rev desc
```
HAVING Clause

Same query as earlier, except that we consider only products that had at least 30 sales.

```
SELECT product, sum(price*quantity)
FROM Purchase
WHERE price > 1
GROUP BY product
HAVING sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
1. Subqueries in SELECT

But are these really equivalent?

```
SELECT DISTINCT C.cname, (SELECT count(*)
FROM Product P
WHERE P.cid=C.cid)
FROM Company C
```

```
SELECT C.cname, count(*)
FROM Company C, Product P
WHERE C.cid=P.cid
GROUP BY C.cname
```

No! Different results if a company has no products

```
SELECT C.cname, count(pname)
FROM Company C LEFT OUTER JOIN Product P
ON C.cid=P.cid
GROUP BY C.cname
```
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Using **ALL**:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE 200 >= ALL (SELECT price
FROM Product P
WHERE P.cid = C.cid)
```
Topics

• Relational models
• SQL

• Relational Algebra
• Entity-Relationship Diagram
• Design Theory
• No architecture/storage
• No indexing
• No execution
Overview: Relational Algebra = HOW

SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = z.cid and
  x.price > 100 and
  z.city = 'Seattle'

Execution order is now clearly specified

Logical plan
Many physical details are still left open!

Final answer

T1(pid,name,price,pid,cid,store)
T2(. . . .)
T3(. . . )
T4(name,name)

Customer
Purchase
Product
Natural Join Example

<table>
<thead>
<tr>
<th>Patient P</th>
<th>Voters V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>age</strong></td>
<td><strong>name</strong></td>
</tr>
<tr>
<td>54</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>p2</td>
</tr>
</tbody>
</table>

P $\bowtie$ V

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
</tr>
</tbody>
</table>
## Equijoin Example

### Patient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

### Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

Equijoin $P \bowtie_{P.age = V.age} V$

<table>
<thead>
<tr>
<th>age</th>
<th>P.zip</th>
<th>disease</th>
<th>name</th>
<th>V.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>98120</td>
</tr>
</tbody>
</table>
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Relational Algebra

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    AND y.pno = 2
    AND x.scity = 'Seattle'
    AND x.sstate = 'WA'
```

Relational algebra expression is also called the "logical query plan"
Physical Query Plan 1

\[
\text{SELECT sname FROM Supplier x, Supply y WHERE } x.\text{sid} = y.\text{sid} \land y.\text{pno} = 2 \land x.\text{scity} = \text{Seattle} \land x.\text{sstate} = \text{WA}
\]

A physical query plan is a logical query plan annotated with physical implementation details.
Physical Query Plan 2

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

### Same logical query plan
Different physical plan

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```

```
(On the fly)  \pi_{sname}
(On the fly)  \sigma_{scity='Seattle' \land sstate='WA' \land pno=2}
(Hash join)  sid = sid
```

Supplier (File scan)
Supply (File scan)
**Physical Query Plan 3**

- **(On the fly)**
  - \( \pi_{\text{sname}} \)

- **(Sort-merge join)**
  - \( \text{sid} = \text{sid} \)

- **(Scan & write to T1)**
  - \( \sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA}} \)
  - **(File scan)**
  - **Supplier**

- **(Scan & write to T2)**
  - \( \sigma_{\text{pno}=2} \)
  - **(File scan)**
  - **Supply**

**Different but equivalent logical query plan; different physical plan**

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
  and y.pno = 2
  and x.scity = 'Seattle'
  and x.sstate = 'WA'
```
Topics

• Relational models
• SQL
• Relational Algebra
• Entity-Relationship Diagram
• Design Theory
• No architecture/storage
• No indexing
• No execution
Entity Set to Relation

Product

<table>
<thead>
<tr>
<th>prod-ID</th>
<th>category</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo55</td>
<td>Camera</td>
<td>99.99</td>
</tr>
<tr>
<td>Poken19</td>
<td>Toy</td>
<td>29.99</td>
</tr>
</tbody>
</table>
N-N Relationships to Relations

Represent this in relations
Orders \((\text{prod-ID}, \text{cust-ID}, \text{date})\)

Shipment \((\text{prod-ID}, \text{cust-ID}, \text{name}, \text{date})\)

Shipping-Co \((\text{name}, \text{address})\)

<table>
<thead>
<tr>
<th>prod-ID</th>
<th>cust-ID</th>
<th>name</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo55</td>
<td>Joe12</td>
<td>UPS</td>
<td>4/10/2011</td>
</tr>
<tr>
<td>Gizmo55</td>
<td>Joe12</td>
<td>FEDEX</td>
<td>4/9/2011</td>
</tr>
</tbody>
</table>
Constraints in SQL

Constraints in SQL:

- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

- The more complex the constraint, the harder it is to check and to enforce
What happens when data changes?

• SQL has three/four policies for maintaining referential integrity:
  • **NO ACTION** reject violating modifications (default)
  • **CASCADE** after delete/update do delete/update
  • **SET NULL** set foreign-key field to NULL
  • **SET DEFAULT** set foreign-key field to default value
    • need to be declared with column, e.g.,
    CREATE TABLE Product (pid INT DEFAULT 42)
Database Triggers Example

When Product.price is updated, if it is decreased then set Product.category = ‘On sale’

```sql
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
  OLD ROW AS OldTuple
  NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
  UPDATE Product
  SET category = 'On sale'
  WHERE productID = OldTuple.productID
```
CREATE TRIGGER ProductCategory
ON Product
AFTER UPDATE
AS
BEGIN
  UPDATE Product
  SET category='sale' WHERE productID IN
  (SELECT i.productID from inserted i, deleted d
   WHERE i.productID = d.productID
   AND i.price < d.price)
END
Topics

• Relational models
• SQL
• Relational Algebra
• Entity-Relationship Diagram

• Design Theory
• No architecture/storage
• No indexing
• No execution
Database Design Process

Conceptual Model:

Relational Model:
Tables + constraints
And also functional dep.

Normalization:
Eliminates anomalies

Conceptual Schema

Physical storage details
Physical Schema
Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$

The closure, $\{A_1, \ldots, A_n\}^+ = \text{the set of attributes } B$ s.t. $A_1, \ldots, A_n \rightarrow B$

Example:
1. name $\rightarrow$ color
2. category $\rightarrow$ department
3. color, category $\rightarrow$ price

Closures:
$\text{name}^+ = \{\text{name, color}\}$
$\{\text{name, category}\}^+ = \{\text{name, category, color, department, price}\}$
$\text{color}^+ = \{\text{color}\}$
Closure Algorithm

X={A1, …, An}.

Repeat until X doesn’t change do:
  if B1, …, Bn \rightarrow C is a FD and B1, …, Bn are all in X
  then add C to X.

Example:

1. name \rightarrow color
2. category \rightarrow department
3. color, category \rightarrow price

{\text{name, category}}^+ = \{ \text{name, category, color, department, price} \}

Hence: name, category \rightarrow color, department, price
Practice at Home

Find all FD’s implied by:

\[
\begin{align*}
A, B & \rightarrow C \\
A, D & \rightarrow B \\
B & \rightarrow D
\end{align*}
\]

Step 1: Compute \(X^+\), for every \(X\):

\[
\begin{align*}
A^+ & = A, \quad B^+ = BD, \quad C^+ = C, \quad D^+ = D \\
AB^+ & = ABCD, \quad AC^+ = AC, \quad AD^+ = ABCD, \\
& \quad BC^+ = BCD, \quad BD^+ = BD, \quad CD^+ = CD \\
ABC^+ & = ABD^+ = ACD^+ = ABCD \quad \text{(no need to compute– why ?)} \\
BCD^+ & = BCD, \quad ABCD^+ = ABCD
\end{align*}
\]

Step 2: Enumerate all FD’s \(X \rightarrow Y\), s.t. \(Y \subseteq X^+\) and \(X \cap Y = \emptyset\):

\[
\begin{align*}
AB & \rightarrow CD, \quad AD \rightarrow BC, \quad ABC \rightarrow D, \quad ABD \rightarrow C, \quad ACD \rightarrow B
\end{align*}
\]
Boyece-Codd Normal Form

There are no “bad” FDs:

**Definition.** A relation R is in BCNF if:

Whenever \( X \rightarrow B \) is a non-trivial dependency, then \( X \) is a superkey.

Equivalently:

**Definition.** A relation R is in BCNF if:

\[ \forall X, \text{ either } X^+ = X \text{ or } X^+ = [\text{all attributes}] \]
BCNF Decomposition Algorithm

Normalize(R)

find X s.t.: X ≠ X+ and X+≠ [all attributes]

if (not found) then “R is in BCNF”

let Y = X+ - X; Z = [all attributes] - X+

decompose R into R1(X ∪ Y) and R2(X ∪ Z)

Normalize(R1); Normalize(R2);
Example

The only key is: \{SSN, PhoneNumber\}
Hence \(SSN \rightarrow Name, City\) is a “bad” dependency

In other words:
\(SSN^+ = SSN, Name, City\) and is neither SSN nor All Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>PhoneNumber</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>206-555-1234</td>
<td>Seattle</td>
</tr>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>206-555-6543</td>
<td>Seattle</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>908-555-2121</td>
<td>Westfield</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>908-555-1234</td>
<td>Westfield</td>
</tr>
</tbody>
</table>
Example BCNF Decomposition

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>Seattle</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>Westfield</td>
</tr>
</tbody>
</table>

SSN → Name, City

<table>
<thead>
<tr>
<th>SSN</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-45-6789</td>
<td>206-555-1234</td>
</tr>
<tr>
<td>123-45-6789</td>
<td>206-555-6543</td>
</tr>
<tr>
<td>987-65-4321</td>
<td>908-555-2121</td>
</tr>
<tr>
<td>987-65-4321</td>
<td>908-555-1234</td>
</tr>
</tbody>
</table>

Let’s check anomalies:
- Redundancy ?
- Update ?
- Delete ?
Create a view that returns for each store the prices of products purchased at that store.

Create view:

```
CREATE VIEW StorePrice AS
SELECT DISTINCT x.store, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname
```
Types of Views

• **Virtual views**
  • Computed only on-demand – slow at runtime
  • Always up to date

• **Materialized views**
  • Pre-computed offline – fast at runtime
  • May have stale data (must recompute or update)
  • Indexes are materialized views

• A key component of physical tuning of databases is the selection of materialized views and indexes
Questions