CS 457 Database Management Systems

Lecture 14 Midterm Exam Preview
Overview

• Location: SEM 326
• Date: 10/16, Tuesday
• Time: 12:00pm – 1:10pm
• 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

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

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

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

Company(\texttt{cname}, \texttt{country})
Product(\texttt{pname}, price, category, manufacturer)
– manufacturer is foreign key

\begin{verbatim}
SELECT DISTINCT cname
FROM Product, Company
WHERE country = 'USA' AND category = 'gadget' AND
manufacturer = cname
\end{verbatim}
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

- `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'
```
# Natural Join 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>

**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

<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>

<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>

Patient P \( \bowtie_{P.age = V.age} \) Voters 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>
Relational Algebra

Supplier\((\text{sid}, \text{sname}, \text{scity}, \text{sstate})\)
Supply\((\text{sid}, \text{pno}, \text{quantity})\)

SELECT \text{sname} \\
FROM Supplier \(x\), Supply \(y\) \\
WHERE \(x.\text{sid} = y.\text{sid}\) \\
and \(y.\text{pno} = 2\) \\
and \(x.\text{scity} = \text{’Seattle’}\) \\
and \(x.\text{sstate} = \text{’WA’}\)

Relational algebra expression is also called the “logical query plan”
A physical query plan is a logical query plan annotated with physical implementation details.

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

Physical Query Plan 1

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

(On the fly) \( \sigma_{\text{scity} = 'Seattle' \land \text{sstate} = 'WA' \land \text{pno}=2} \)

(Nested loop) \( \text{sid} = \text{sid} \)

Supplier (File scan)  Supply (File scan)

Supplier\((\text{sid, sname, scity, sstate})\)
Supply\((\text{sid, pno, quantity})\)
Physical Query Plan 2

\[
\begin{align*}
\text{(On the fly) } & \quad \pi_{\text{snname}} \\
\text{(On the fly) } & \quad \sigma_{\text{scity}=‘Seattle’ \land \text{sstate}=‘WA’ \land \text{pno}=2} \\
\text{(Hash join) } & \quad \sigma_{\text{sid} = \text{sid}} \\
\text{Supplier (File scan) } & \quad \text{Supply (File scan)}
\end{align*}
\]

Same logical query plan
Different physical plan

\[
\begin{align*}
\text{SELECT } \text{snname} \\
\text{FROM Supplier } x, \text{ Supply } y \\
\text{WHERE } x.\text{sid} = y.\text{sid} \\
\text{and } y.\text{pno} = 2 \\
\text{and } x.\text{scity} = ‘Seattle’ \\
\text{and } x.\text{sstate} = ‘WA’
\end{align*}
\]

Supplier(\text{sid, sname, scity, sstate})
Supply(\text{sid, pno, quantity})
Physical Query Plan 3

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>Pokemn19</td>
<td>Toy</td>
<td>29.99</td>
</tr>
</tbody>
</table>
Represent this in relations
N-N Relationships to Relations

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

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

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

<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 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’

```
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:

name$^+ = \{\text{name, color}\}$

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

color$^+ = \{\text{color}\}$
Closure Algorithm

X={A1, ..., An}.

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

Example:

1. name → color
2. category → department
3. color, category → price

{name, category}+ = 
   { name, category, color, department, price }

Hence: name, category → 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 \( X \):

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

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

\[
\begin{align*}
AB & \rightarrow CD, \quad AD \rightarrow BC, \quad BC \rightarrow D, \quad ABC \rightarrow D, \quad ABD \rightarrow C, \quad ACD \rightarrow B
\end{align*}
\]
Boyce-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: \{\text{SSN, PhoneNumber}\}

Hence \text{SSN} \rightarrow \text{Name, City} is a “bad” dependency

In other words:
\text{SSN+} = \text{SSN, Name, City} and is neither \text{SSN} nor \text{All Attributes}
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>

Let’s check anomalies:
- Redundancy?
- Update?
- Delete?
Find $X$ s.t.: $X \neq X^+$ and $X^+ \neq [\text{all attributes}]$

**Example BCNF Decomposition**

Person(name, SSN, age, hairColor, phoneNumber)

- SSN $\rightarrow$ name, age
- age $\rightarrow$ hairColor

**Iteration 1:**

**Person:** SSN$^+$ = SSN, name, age, hairColor

Decompose into:
- $P$(SSN, name, age, hairColor)
- Phone(SSN, phoneNumber)

**Iteration 2:**

**P:** age$^+$ = age, hairColor

Decompose:
- People(SSN, name, age)
- Hair(age, hairColor)
- Phone(SSN, phoneNumber)

Note the keys!
A Simple View

Create a view that returns for each store the prices of products purchased at that store

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

This is like a new table StorePrice(store, price)
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