CS 457 Database Management Systems

Final Exam Preview
Final Exam Overview

• When: December 10th, 3:00pm – 4:15pm
• Where: in class
• Format:
  • Close book, no paper, no cellphone, ...
  • 25 questions: Multiple choice (10); Short answers (15)
• Topics: all lectures
  • With emphasis on today’s slides
Topics (or, chapters)

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
Data Models

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

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

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

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
Basic SQL

• Get familiar with the SQL script for Lecture 3
  • Link: https://www.cse.unr.edu/~dfz/teaching/CS457-F19/download/lec03-sql-basics.zip
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(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

- `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:

\[
\text{SELECT Count(product) FROM Purchase WHERE price > 4.99}
\]

same as Count(*) if no nulls

We probably want:

\[
\text{SELECT Count(DISTINCT product) FROM Purchase WHERE price > 4.99}
\]
Ordering Results

```
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?

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

No! Different results if a company has no products

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

```sql
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 \textit{ALL}:

\begin{verbatim}
SELECT DISTINCT C.cname
FROM Company C
WHERE 200 >= ALL (SELECT price
FROM Product P
WHERE P.cid = C.cid)
\end{verbatim}
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
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!
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

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>

\[ P \bowtie_{P.age=V.age} V \]

<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>
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”
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 2

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

Same logical query plan
Different physical plan

\[
\text{SELECT sname}
\text{FROM Supplier x, Supply y}
\text{WHERE x.sid = y.sid}
\text{and y.pno = 2}
\text{and x.scity = 'Seattle'}
\text{and x.sstate = 'WA'}
\]
Physical Query Plan 3

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

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'
```

(a) $\sigma_{\text{scity}=\text{'Seattle'}} \land \text{sstate}=\text{'WA'}$

(b) $\sigma_{\text{pno}=2}$

(Sort-merge join)

(Scan & write to T1)

(Scan & write to T2)

(On the fly)
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
Entity Set to Relation

Product(prod-ID, category, price)

<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>
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:

- 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)
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

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
Database Design Process

Conceptual Model:
- Name
- Product
- Price
- Company
- Name
- Address

Relational Model:
- Tables + constraints
- And also functional dep.

Normalization:
- Eliminates anomalies

Conceptual Schema

Physical Schema
- Physical storage details
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:**

<table>
<thead>
<tr>
<th>Closure</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>name$^+$</td>
<td>{name, color}</td>
</tr>
<tr>
<td>{name, category}$^+$</td>
<td>{name, category, color, department, price}</td>
</tr>
<tr>
<td>color$^+$</td>
<td>{color}</td>
</tr>
</tbody>
</table>

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

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

Hence: name, category → color, department, price

Example:
1. name → color
2. category → department
3. color, category → price
Practice at Home

Find all FD’s implied by:

- A, B → C
- A, D → B
- B → D

**Step 1: Compute X⁺, for every X:**

- \( A⁺ = A \), \( B⁺ = BD \), \( C⁺ = C \), \( D⁺ = D \)
- \( AB⁺ = ABCD \), \( AC⁺ = AC \), \( AD⁺ = ABCD \),
  \( BC⁺ = BCD \), \( BD⁺ = BD \), \( CD⁺ = CD \)
- \( ABC⁺ = ABD⁺ = ACD⁺ = ABCD \) (no need to compute– why ?)
- \( BCD⁺ = BCD \), \( ABCD⁺ = ABCD \)

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

- \( AB → CD \), \( AD → BC \), \( ABC → D \), \( ABD → C \), \( ACD → B \)
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:

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

Normalize(R)
find $X$ s.t.: $X \neq X^+$ and $X^+ \neq$ [all attributes]
if (not found) then “$R$ is in BCNF”
let $Y = X^+ - X$; $Z = \text{[all attributes]} - X^+$
decompose $R$ into $R_1(X \cup Y)$ and $R_2(X \cup Z)$
Normalize($R_1$); Normalize($R_2$);
Example

<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>

The only key is: \{SSN, PhoneNumber\}

Hence SSN → Name, City is a “bad” dependency.

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

SSN → Name, City

Let's check anomalies:
- Redundancy?
- Update?
- Delete?
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
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions
• Blockchains
ACID Properties

• **Atomicity**: Either all changes performed by transaction occur or none occurs

• **Consistency**: A transaction as a whole does not violate integrity constraints

• **Isolation**: Transactions appear to execute one after the other in sequence

• **Durability**: If a transaction commits, its changes will survive failures
A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(B,t)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,s)</td>
<td></td>
</tr>
<tr>
<td>READ(A,s)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A,s)</td>
<td></td>
</tr>
<tr>
<td>READ(B,s)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,s)</td>
<td></td>
</tr>
</tbody>
</table>
Serializable Schedule

A schedule is **serializable** if it is equivalent to a serial schedule.
A Serializable Schedule

This is a *serializable* schedule.
This is NOT a serial schedule

T1
---
READ(A, t)
t := t+100
WRITE(A, t)

READ(B, t)
t := t+100
WRITE(B, t)

T2
---
READ(A, s)
s := s*2
WRITE(A, s)

READ(B, s)
s := s*2
WRITE(B, s)
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></td>
</tr>
</tbody>
</table>
Serializable Schedules

• The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules? I.e. run one transaction after the other?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases *may* schedule TXNs serially
Conflict Serializability

**Definition** A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions.

- Every *conflict-serializable* schedule is serializable
  - The converse is not true in general
  - Not true when phantom problem exists
Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must precede all unlock requests

- This ensures conflict serializability! (will prove this shortly)
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability
A New Problem: Non-recoverable Schedule

T1

\[ L_1(A); L_1(B); \text{READ}(A, t) \]
\[ t := t+100 \]
\[ \text{WRITE}(A, t); \text{U}_1(A) \]

T2

\[ \text{READ}(B, t) \]
\[ t := t+100 \]
\[ \text{WRITE}(B, t); \text{U}_1(B); \]

\[ \text{DENIED…} \]

\[ \text{READ}(B, t) \]
\[ t := t+100 \]
\[ \text{WRITE}(B, t); \text{U}_1(B); \]

\[ \text{GRANTED}; \text{READ}(B, s) \]
\[ s := s*2 \]
\[ \text{WRITE}(B, s); \text{U}_2(A); \text{U}_2(B); \]

\[ \text{Commit} \]

Abort
Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK
- Schedule is serializable
- Schedule is recoverable
- Schedule avoids cascading aborts
Strict 2PL

T1

L₁(A); READ(A)
A := A+100
WRITE(A);

L₁(B); READ(B)
B := B+100
WRITE(B);

U₁(A), U₁(B); Rollback

T2

L₂(A); DENIED…

…GRANTED; READ(A)
A := A*2
WRITE(A);
L₂(B); READ(B)
B := B*2
WRITE(B);

U₂(A); U₂(B); Commit
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Transactions

• Blockchains
Blockchains

• Categorization (in terms of openness)
  • Permissionless blockchains
    • Bitcoin, Ethereum, Litecoin, ....
    • Bitcoin has about 10,000 mining sites worldwide
  • Permissioned
    • Privated blockchains built upon Hyperledger Fabric (mainly driven by IBM)
Blockchains

• Key data structures and algorithm
  • Replicated over nodes
  • On each node
    • A full copy of linked list of blocks
      • Each block comprises multiple transactions
    • The blocks are *uniquely* linked through cryptography, e.g., hashing
  • Two main jobs:
    • Verification of new blocks (e.g., check hashing values)
    • Reaching consensus with others (see “Between nodes” below)

• Between nodes
  • For permissionless blockchains: competition for mining new blocks (e.g., solving the puzzle to append the new block with correct hash value)
  • For permissioned blockchains: consensus protocols for all non-faulty nodes to reach agreement (e.g., Practical Byzantine Fault Tolerance)
Best of Luck!