CS 457 Database Management Systems

Final Exam Preview
Final Exam Overview

• When: May 7th, 4:30pm – 5:45pm
• 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
- Storage, Architecture, Indexing
- Query execution
- Transactions
Topics

• Data Models
  • SQL
  • Relational Algebra
  • E-R Diagram
  • Design Theory
  • Storage, Architecture, Indexing
  • Query execution
  • Transactions
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
• Storage, Architecture, Indexing
• Query execution
• Transactions
Basic SQL

• Get familiar with the SQL script for Lecture 3
  • Link: https://www.cse.unr.edu/~dfz/teaching/CS457-S19/download/lec03-sql-basics.zip
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

```
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
COUNTing Duplicates

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

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
```

No! Different results if a company has no products

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

```
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 \textbf{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
• Storage, Architecture, Indexing
• Query execution
• Transactions
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

<table>
<thead>
<tr>
<th>Patient P</th>
<th>Voters V</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>name</td>
</tr>
<tr>
<td>54</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>zip</td>
<td>age</td>
</tr>
<tr>
<td>98125</td>
<td></td>
</tr>
<tr>
<td>98120</td>
<td></td>
</tr>
<tr>
<td>disease</td>
<td>zip</td>
</tr>
<tr>
<td>heart</td>
<td></td>
</tr>
<tr>
<td>flu</td>
<td></td>
</tr>
</tbody>
</table>

\[ P \Join V \]

<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>
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”
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

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

\textbf{SELECT} \textit{sname}
\textbf{FROM} \textbf{Supplier} \textit{x, Supply} \textit{y}
\textbf{WHERE} \textit{x.sid} = \textit{y.sid}
\textbf{AND} \textit{y.pno} = 2
\textbf{AND} \textit{x.scity} = ‘Seattle’
\textbf{AND} \textit{x.sstate} = ‘WA’

\textbf{(Hash join)}
\textbf{(On the fly)}
\textbf{(On the fly)}
\textbf{(File scan)}
\textbf{(File scan)}

\textbf{Same logical query plan}
\textbf{Different physical plan}
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Physical Query Plan 3

(On the fly) π sname (d)
(Sort-merge join) sid = sid (c)
(Scan & write to T1)
(a) σ scity = ‘Seattle’ ∧ sstate = ‘WA’
(Scan & write to T2)
(b) σ pno = 2

Supplier (File scan)
Supply (File scan)

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

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Storage, Architecture, Indexing
• Query execution
• Transactions
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
N-N Relationships to Relations

Orders\((prod-ID, cust-ID, date)\)
Shipment\((prod-ID, cust-ID, name, date)\)
Shipping-Co\((name, address)\)

<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 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’

```
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
• Storage, Architecture, Indexing
• Query execution
• Transactions
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

\[ X = \{A_1, \ldots, A_n\} \]

Repeat until \( X \) doesn’t change do:

\[ \text{if } B_1, \ldots, B_n \rightarrow C \text{ is a FD and } B_1, \ldots, B_n \text{ are all in } X \]

\[ \text{then add } C \text{ to } X. \]

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 \(X\):

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

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

\[
\begin{align*}
AB &\rightarrow CD, \quad AD \rightarrow BC, \quad ABC \rightarrow D, \quad ABD \rightarrow C, \quad ACD \rightarrow B \\
\end{align*}
\]
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 ≠ 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 \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 All Attributes
Example BCNF Decomposition

Let’s check anomalies:
- Redundancy ?
- Update ?
- Delete ?

<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>

<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>

SSN → Name, City
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
• Storage, Architecture, Indexing
• Query execution
• Transactions
DBMS Architecture

- Process Manager:
  - Admission Control
  - Connection Mgr

- Query Processor:
  - Parser
  - Query Rewrite
  - Optimizer
  - Executor

- Storage Manager:
  - Access Methods
  - Buffer Manager
  - Lock Manager
  - Log Manager

- Shared Utilities:
  - Memory Mgr
  - Disk Space Mgr
  - Replication Services
  - Admin Utilities

Buffer Manager

- Page requests from higher-level code
- Access methods
  - Buffer pool manager
- Main memory
- Disk
  - Disk is a collection of blocks
  - 1 page corresponds to 1 disk block
Alternate Storage Manager
Design: Column Store

Rows stored contiguously on disk

Columns stored contiguously on disk
More Detailed Example

Row-based (4 pages)

Page

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

Column-based (4 pages)

Page

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

C-Store also avoids large tuple headers
Different Types of Files

• For the data inside base relations:
  • Heap file (tuples stored without any order)
  • Sequential file (tuples sorted some attribute(s))
  • Indexed file (tuples organized following an index)

• Then we can have additional index files that store (key,rid) pairs

• Index can also be a “covering index”
  • Index contains (search key + other attributes, rid)
  • Index suffices to answer some queries
Primary Index

- **Primary index** determines location of indexed records
- **Dense index**: sequence of (key,rid) pairs (record level)
Primary Index

• *Sparse* index (page level)
Secondary Indexes

• To index **other attributes than primary key**
• Always dense
Hash-Based Index

Good for point queries but not range queries

Secondary hash-based index

Primary hash-based index
B+ Trees Basics

• Parameter $d =$ the **degree**

• Each node has $d \leq m \leq 2d$ keys (except root)

Each node also has **m+1 pointers**

• Each leaf has $d \leq m \leq 2d$ keys:

Data records

Next leaf
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Storage, Architecture, Indexing
• Query execution
• Transactions
Cost Parameters

• **Cost = total number of I/Os**
  • This is a simplification that ignores CPU, network

• **Parameters:**
  • $B(R) = \# \text{ of blocks (i.e., pages) for relation } R$
  • $T(R) = \# \text{ of tuples in relation } R$
  • $V(R, a) = \# \text{ of distinct values of attribute } a$
    • When $a$ is a key, $V(R,a) = T(R)$
    • When $a$ is not a key, $V(R,a)$ can be anything $< T(R)$
Join Algorithms

• Hash join

• Nested loop join

• Sort-merge join
Hash Join

Hash join:  \( R \bowtie S \)

- Scan \( R \), build buckets in main memory
- Then scan \( S \) and join
- Cost: \( B(R) + B(S) \)

- One-pass algorithm when \( B(R) \leq M \)
Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- $R$ is the outer relation, $S$ is the inner relation

```plaintext
for each tuple $t_1$ in $R$ do
    for each tuple $t_2$ in $S$ do
        if $t_1$ and $t_2$ join then output $(t_1, t_2)$
```

- **Cost**: $B(R) + T(R) B(S)$
- Multiple-pass since $S$ is read many times

What is the Cost?
Two-Pass Algorithms

• What if data does not fit in memory?
  • Need to process it in multiple passes

• Two key techniques
  • Sorting
  • Hashing
Sort-Merge-Join

Join $R \bowtie S$

• Step 1a: initial runs for $R$
• Step 1b: initial runs for $S$
• Step 2: merge and join
Cost of External Merge Sort

• Read+write+read = 3B(R)+3B(S)

• Assumption: B(R) <= M^2
Topics

• Data Models
• SQL
• Relational Algebra
• E-R Diagram
• Design Theory
• Storage, Architecture, Indexing
• Query execution
• Transactions
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>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>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</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

<table>
<thead>
<tr>
<th>READ(A, t)</th>
<th>WRITE(A, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
</tbody>
</table>

T2

<table>
<thead>
<tr>
<th>READ(A,s)</th>
<th>WRITE(A, s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READ(B, t)</th>
<th>WRITE(B, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$; $L_1(B)$; READ(A, t)</td>
<td>$L_2(A)$; READ(A,s)</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(A, t); $U_1(A)$</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td>$L_2(B)$; DENIED…</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>$t := t+100$</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t); $U_1(B)$;</td>
<td>...GRANTED; READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>$s := s*2$</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s); $U_2(A)$; $U_2(B)$;</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

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...

L₂(B); DENIED...

…GRANTED; READ(A)
A := A * 2
WRITE(A);
L₂(B); READ(B)
B := B * 2
WRITE(B);
U₂(A), U₂(B); Commit
Best of Luck!