CS 4440 A

Midterm Review

Lecture 9

02/11/26

Midterm Logistics

- Midterm will be held Monday Feb 16 from 3:30pm - 4:45pm (during class time).
- **Please arrive early** - the exam is going to start at 3:30pm.
- Open notes, no electronic devices. **Please print out your notes.**
- Contents covered: Lec 2 (relational algebra) – Lec 7 (txn cc)
- Past Exam: available on canvas, under Files->Past Exams

Lec 2: Relational Algebra

Relational Algebra (RA)

- Five **basic** operators:
  1. Selection: σ
  2. Projection: Π
  3. Cartesian Product: $\times$
  4. Union: $\cup$
  5. Difference: -
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Grouping: γ

RDBMSs use *multisets*, however in relational algebra formalism we will consider **sets!**

Selection (σ)

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- The condition c can be =, <, >, <>

```
Students(sid,sname,gpa)
```

SQL:

```
SELECT *
FROM Students
WHERE gpa > 3.5;
```

RA:

$$\sigma_{gpa>3.5}(Students)$$

Projection (Π)

- Eliminates columns, then removes duplicates
- Notation: $\Pi_{A1,\ldots,An}(R)$

```
Students(sid,sname,gpa)
```

SQL:

```
SELECT DISTINCT
   sname,
   gpa
FROM Students;
```

RA:

$$\Pi_{sname,gpa}(Students)$$

Renaming (ρ)

- Changes the schema, not the instance
- A 'special' operator- neither basic nor derived
- Notation: $\rho_{B1,\ldots,Bn}(R)$

- **Note: this is shorthand for the proper form (since names, not order matters!):**
  - $\rho_{A1 \rightarrow B1,\ldots,An \rightarrow Bn}(R)$

```
Students(sid,sname,gpa)
```

SQL:

```
SELECT
  sid AS studId,
  sname AS name,
  gpa AS gradePtAvg
FROM Students;
```

RA:

$\rho_{studId,name,gradePtAvg}(Students)$

Natural Join (⋈)

- Notation: $R_1 \bowtie R_2$

- Joins R_1 and R_2 on *equality of all shared attributes*
  - If R_1 has attribute set A, and R_2 has attribute set B, and they share attributes $A \cap B = C$, can also be written: $R_1 \bowtie_C R_2$

- Our first example of a *derived* RA operator:
  - Meaning: $R_1 \bowtie R_2 = \Pi_{A \cup B}(\sigma_{C=D}(\rho_{C \to D}(R_1) \times R_2))$
  - Where:
    - The rename $\rho_{C \to D}$ renames the shared attributes in one of the relations
    - The selection $\sigma_{C=D}$ checks equality of the shared attributes
    - The projection $\Pi_{A \cup B}$ eliminates the duplicate common attributes

```
Students(sid,name,gpa)
People(ssn,name,address)
```

SQL:

```
SELECT DISTINCT
  ssid, S.name, gpa,
  ssn, address
FROM
  Students S,
  People P
WHERE S.name = P.name;
```

RA:

$$Students \bowtie People$$

Theta Join ($\bowtie_\theta$)

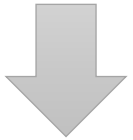
- A join that involves a predicate
- R1 $\bowtie_\theta$ R2 $= \sigma_\theta$ (R1 $\times$ R2)
- Here θ can be any condition

Note that natural join is a theta join + a projection.

```
Students(sid,sname,gpa)
People(ssn,pname,address)
```

SQL:

```
SELECT *
FROM
  Students,People
WHERE θ;
```

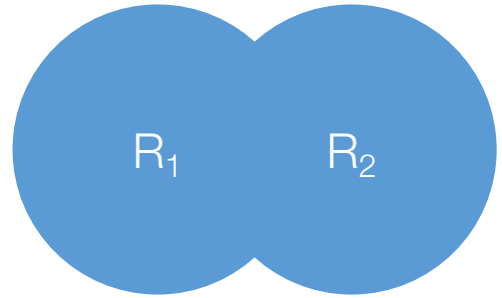


RA:

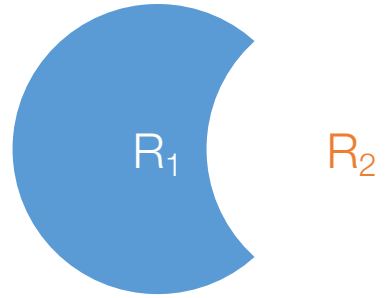
$$Students \bowtie_\theta People$$

Union (∪) and Difference (–)

- R1 ∪ R2
- Example:
  - ActiveEmployees ∪ RetiredEmployees



- R1 – R2
- Example:
  - AllEmployees -- RetiredEmployees



Converting SFW Query -> RA

You should also be able to convert RA -> SQL query.

Remember to add the DISTINCT keyword

```
SELECT DISTINCT A1,…,An
FROM            R1,…,Rm
WHERE           c1 AND … AND ck;
```

$$\Pi_{A_1,\ldots,An}(\sigma_{c_1} \ldots \sigma_{c_k}(R_1 \bowtie \cdots \bowtie R_m))$$

The selections should happen before the projections

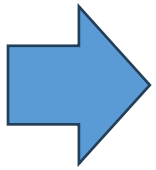
Practice Question

- Exam 1: 1 Relational Algebra

Lec 3-4: Design Theory

Data Anomalies

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..



Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
..	..

Course	Room
CS145	B01
CS229	C12

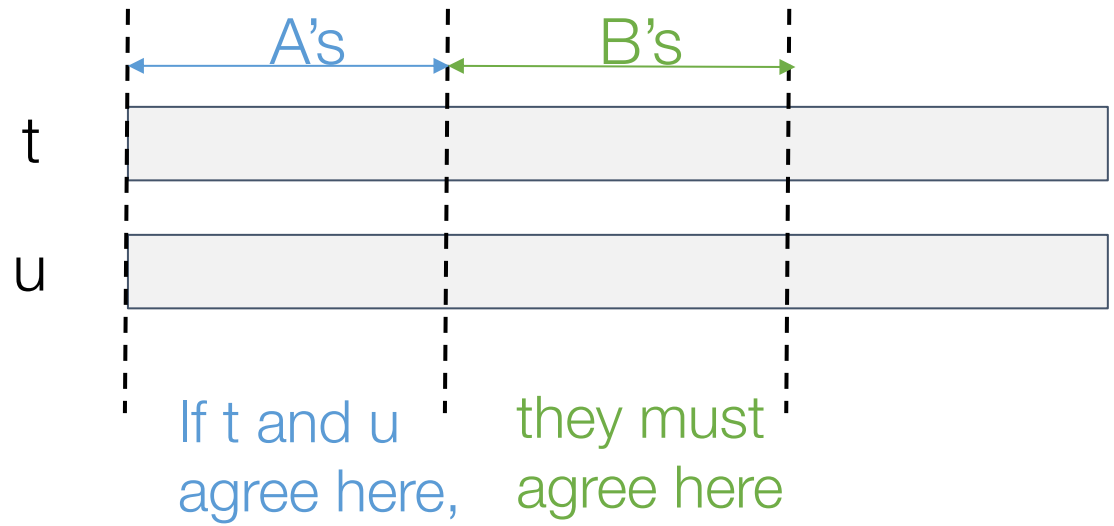
Eliminate anomalies by **decomposing relations.**

- Redundancy
- Update anomaly
- Delete anomaly
- Insert anomaly

Functional dependency (FD)

Definition: if two tuples of R agree on all the attributes $A_1, A_2, \ldots, A_n$, they must also agree on (or functionally determine) $B_1, B_2, \ldots, B_m$

- Denoted as $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$



A->B means that "whenever two tuples agree on A then they agree on B."

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Three simple rules called **Armstrong's Rules.**

1. Reflexivity,
2. Augmentation, and
3. Transitivity

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs **F**:
Then the **closure**, $\{A_1, \ldots, A_n\}^+$ is the set of attributes **B** s.t. $\{A_1, \ldots, A_n\} \rightarrow B$

Example: F =

```
{name} → {color}
{category} → {department}
{color, category} → {price}
```

Example Closures:

```
{name}+ = {name, color}
{name, category}+ =
{name, category, color, dept, price}
{color}+ = {color}
```

Closure algorithm

Start with X = $\{A_1, \ldots, A_n\}$ and set of FDs F.

Repeat until X doesn't change; **do**:

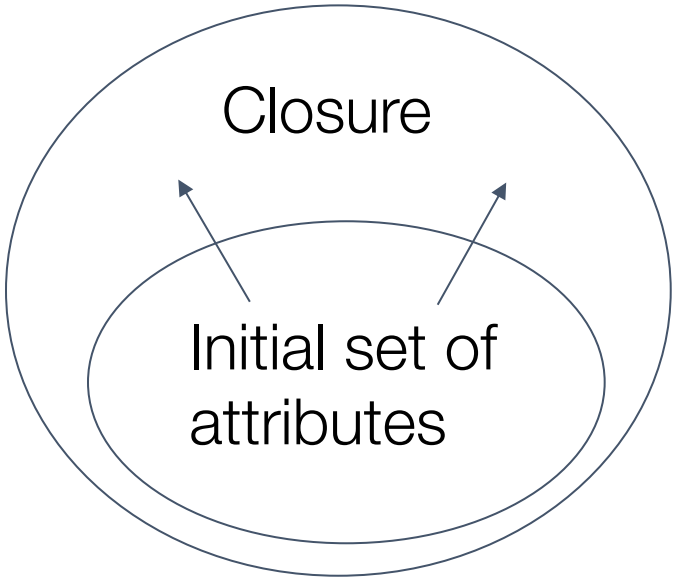
if $\{B_1, \ldots, B_n\} \rightarrow$ C is entailed by F

and $\{B_1, \ldots, B_n\} \subseteq X$

then add C to X.

Return X as X^+

Helps to split the FD's of F so each FD has a single attribute on the right



Keys and Superkeys

A **superkey** is a set of attributes $A_1, \ldots, A_n$ s.t. for *any other* attribute **B** in R, we have $\{A_1, \ldots, A_n\} \rightarrow B$

I.e. all attributes are *functionally determined* by a superkey

A **key** is a *minimal* superkey

Meaning that no subset of a key is also a superkey

Computing Keys and Superkeys

- Superkey?
  - Compute the closure of A
  - See if it = the full set of attributes

- Key?
  - Confirm that A is superkey
  - Make sure that no subset of A is a superkey
    - *Only need to check one 'level' down!*

Let A be a set of attributes, R set of all attributes, F set of FDs:

```
IsSuperkey(A, R, F):
A+ = ComputeClosure(A, F)
Return (A+==R)?
```

```
IsKey(A, R, F):
If not IsSuperkey(A, R, F):
        return False
For B in SubsetsOf(A, size=len(A)-1):
        if IsSuperkey(B, R, F):
                return False
return True
```

Practice Question

- Exam 1: 2.1 FDs, 2.2 Keys

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

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

if $\{A_1, ..., A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, ..., A_n\}$ **is a superkey** for R

Equivalently: ∀ sets of attributes X, either ($X^+ = X$) or (X^+ = all attributes)

Example

BCNFDecomp(R):

- Find an FD $X \rightarrow Y$ that violates BCNF
  (X and Y are sets of attributes)
- Compute the closure X+
- let $Y = X^+ - X$, $Z = (X^+)^C$
  decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$
- Recursively decompose R_1 and R_2

R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}

Note: Projection of FDs

Let F be the set of FDs in the relation R. What FD's hold for $R_1 = \pi_L(R)$?

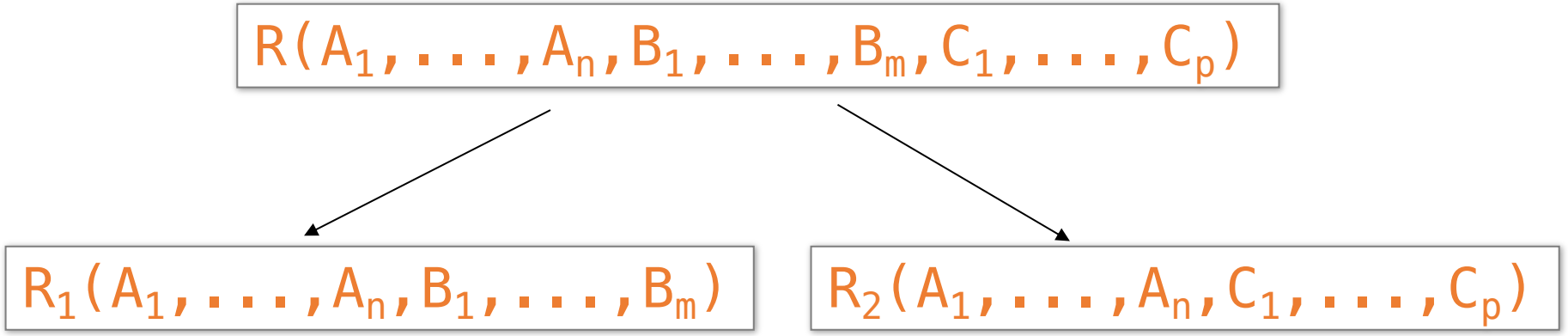
An FD $X \rightarrow Y$ from the original relation R will hold in the project R_1 iff

- Attributes in X and Y are all contained with R_1
- $X \rightarrow Y$ is logical implied by the original set F

Example

- Suppose R(A, B, C, D) has FDs F = {$A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$}
- Then the FD's for R_1(A, C, D) are
  - $A \rightarrow C$: Implied by F
  - $C \rightarrow D$: Inherited from F

Lossless Decompositions

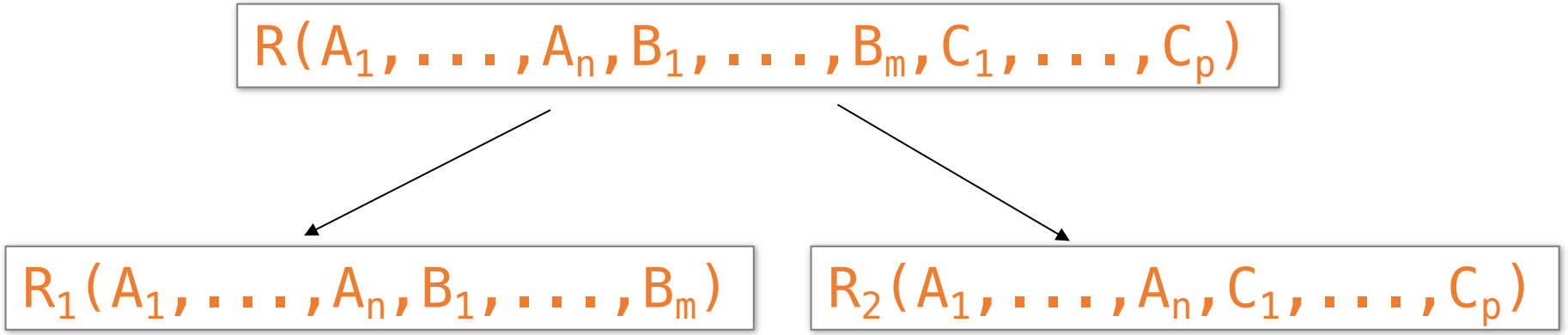


R_1 = the *projection* of R on A_1, ..., A_n, B_1, ..., B_m

R_2 = the *projection* of R on A_1, ..., A_n, C_1, ..., C_p

A decomposition R to (R1, R2) is lossless if R = R1 Join R2

Lossless Decompositions



If $A_1, ..., A_n \rightarrow B_1, ..., B_m$
Then the decomposition is lossless

Note: don't need
$A_1, ..., A_n \rightarrow C_1, ..., C_p$

BCNF decomposition is always lossless.

Lossy vs. Lossless

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	19.99	Camera
OneClick	24.99	Camera
Gizmo	19.99	Camera
Gizmo	24.99	Camera

Lossy vs. Lossless

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Recorder

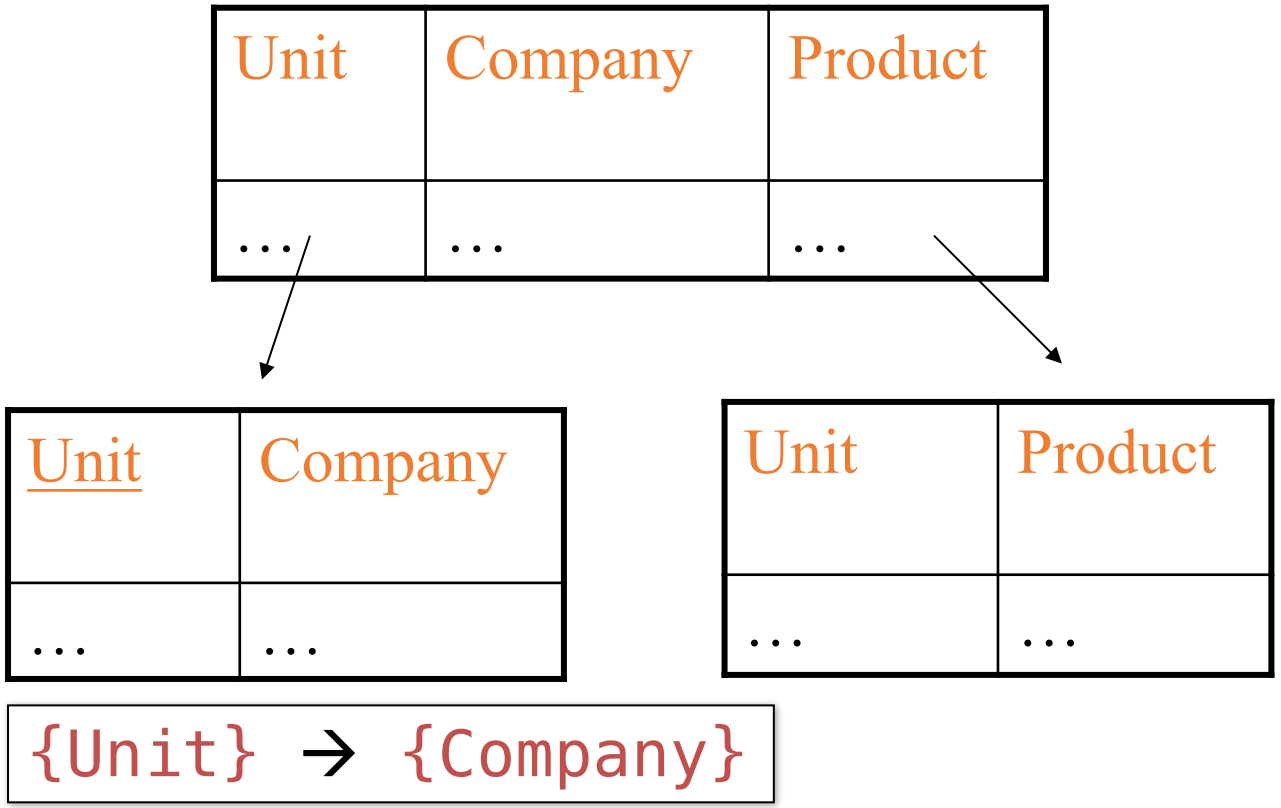
{Category} → {Name}

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Recorder

Price	Category
19.99	Gadget
24.99	Camera
19.99	Recorder

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Recorder

A Problem with BCNF



{Unit} → {Company}
{Company,Product} → {Unit}

We do a BCNF decomposition on a "bad" FD:

$\{Unit\}^+ = \{Unit, Company\}$

We lose the FD {Company,Product} → {Unit}!!

Third normal form (3NF)

A relation R is in 3NF if:

For every non-trivial FD $A_1, ..., A_n \rightarrow B$, either

- $\{A_1, ..., A_n\}$ is a superkey for R
- B is a prime attribute (i.e., B is part of some candidate key of R)

Example:

- The keys are AB and AC
- B → C is a BCNF violation, but not a 3NF violation because C is prime (part of the key AC)

R(A,B,C)

AC → B
B → C

Minimal basis generation

Input: F = {A → AB, AB → C}

1. Split FD's so that they have singleton right sides
   G = {A → B, A → A, AB → C}
2. Remove trivial FDs
   G = {A → B, AB → C}
3. Minimize the left sides of each FD
   G = {A → B, A → C}
4. Remove redundant FDs
   G = {A → B, A → C}

Step 3:

For each FD X → A in F:
For each attribute B in X:
If (X - {B})+ contains A,
remove B from X.

3NF Decomposition Algorithm

3NFDecomp(R, F):

- Find minimal basis for F, say G
- For each FD X → A in G, if there is no relation that contains XA, create a new relation (X, A)
- Eliminate any relation that is a proper subset of another relation.
- If none of the resulting schemas are superkeys, add one more relation whose schema is a key for R

Minimal basis:

AB → C
C → B
A → D

Keys:

ABE,ACE

R(A,B,C,D,E)

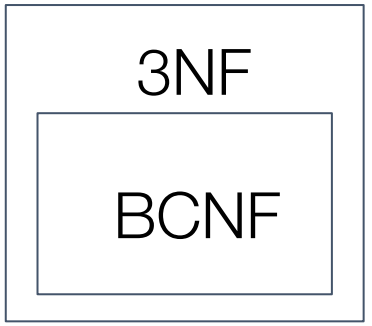
R_1(A,B,C)

R_2(B,C)

R_3(A,D)

R_4(A,B,E)

BCNF vs 3NF



- Given a non-trivial FD X → B (X is a set of attributes)
  - BCNF: X must be a superkey
  - 3NF: X must be a superkey or B is prime
- Use 3NF over BCNF if you need dependency preservation
- However, 3NF may not remove all redundancies and anomalies

Practice Question

- Exam 1: 3 Decomposition

MVD Example

Movie_ Star (A)	Address (B)	Movie (C)
Leonardo DiCaprio	Los Angeles	Titanic
Leonardo DiCaprio	Los Angeles	Inception
Leonardo DiCaprio	New York	Titanic
Leonardo DiCaprio	New York	Inception
Kate Winslet	Los Angeles	Titanic
Kate Winslet	Los Angeles	The Reader
Kate Winslet	London	Titanic
Kate Winslet	London	The Reader

- **Independence**: The set of addresses is independent of the set of movies for a given movie star.
- **Redundancy**: Notice how each movie is repeated for every address that the star lives in, and vice versa.

MVDs: Movie Star Example

	Movie_ Star (A)	Address (B)	Movie (C)
t_1	Leonardo DiCaprio	Los Angeles	Titanic
t_3	Leonardo DiCaprio	Los Angeles	Inception
	Leonardo DiCaprio	New York	Titanic
t_2	Leonardo DiCaprio	New York	Inception
	Kate Winslet	Los Angeles	Titanic
	Kate Winslet	Los Angeles	The Reader
	Kate Winslet	London	Titanic
	Kate Winslet	London	The Reader

More formally, we write **{A}** ↠ **{B}** if for every pair of tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$, there exisits a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R\backslash B] = t_2[R\backslash B]$

Where R\B is “R minus B” i.e. the attributes of R not in B

*There are no restrictions on t1, t2, t3. They can be the same or different.

Multi-Value Dependencies (MVDs)

One less formal, literal way to phrase the definition of an MVD:

The MVD $X \twoheadrightarrow Y$ holds on R if for any pair of tuples with the same X values, the tuples with the same X values, but the other permutations of Y and A\Y values, is also in R

Ex: X = {x}, Y = {y}:

x	y	z
1	0	1
1	1	0

For $X \twoheadrightarrow Y$ to hold must have…

x	y	z
1	0	1
1	1	0
1	0	0
1	1	1

Practice Question

- Exam 1: 2.3 MVDs

Lec 5-7: Transactions

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
      UPDATE Product
      SET Price = Price – 1.99
      WHERE pname = 'Gizmo'
COMMIT
```

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. Recovery & Durability: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. Concurrency: Achieving better performance by parallelizing TXNs *without* creating anomalies

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

Comparison of SQL isolation levels

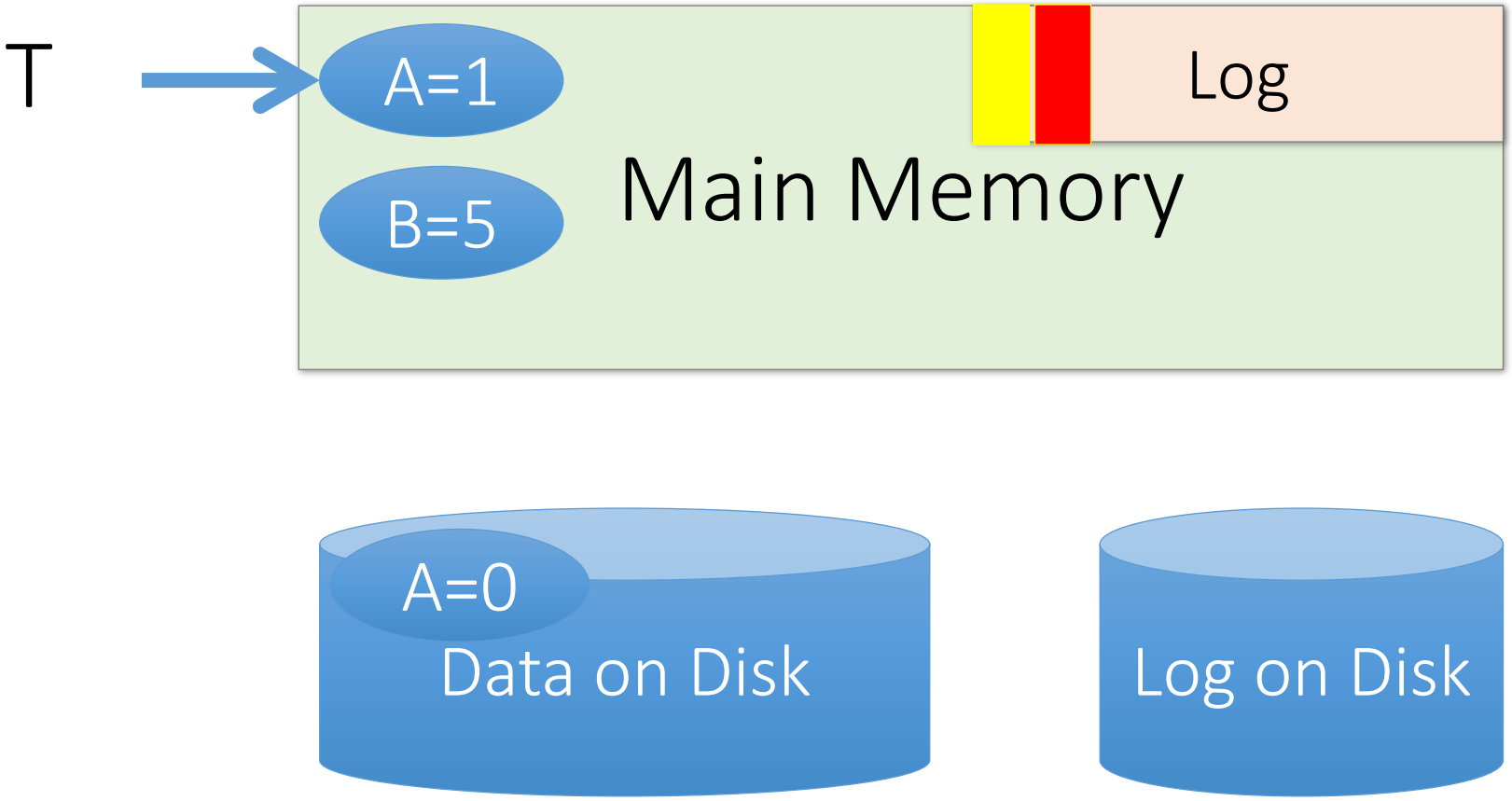
Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
`READ UNCOMMITTED`	✓	✓	✓
`READ COMMITTED`	🚫	✓	✓
`REPEATABLE READ`	🚫	🚫	✓
`SERIALIZABLE`	🚫	🚫	🚫

Logging and Recovery

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

A: 0→1



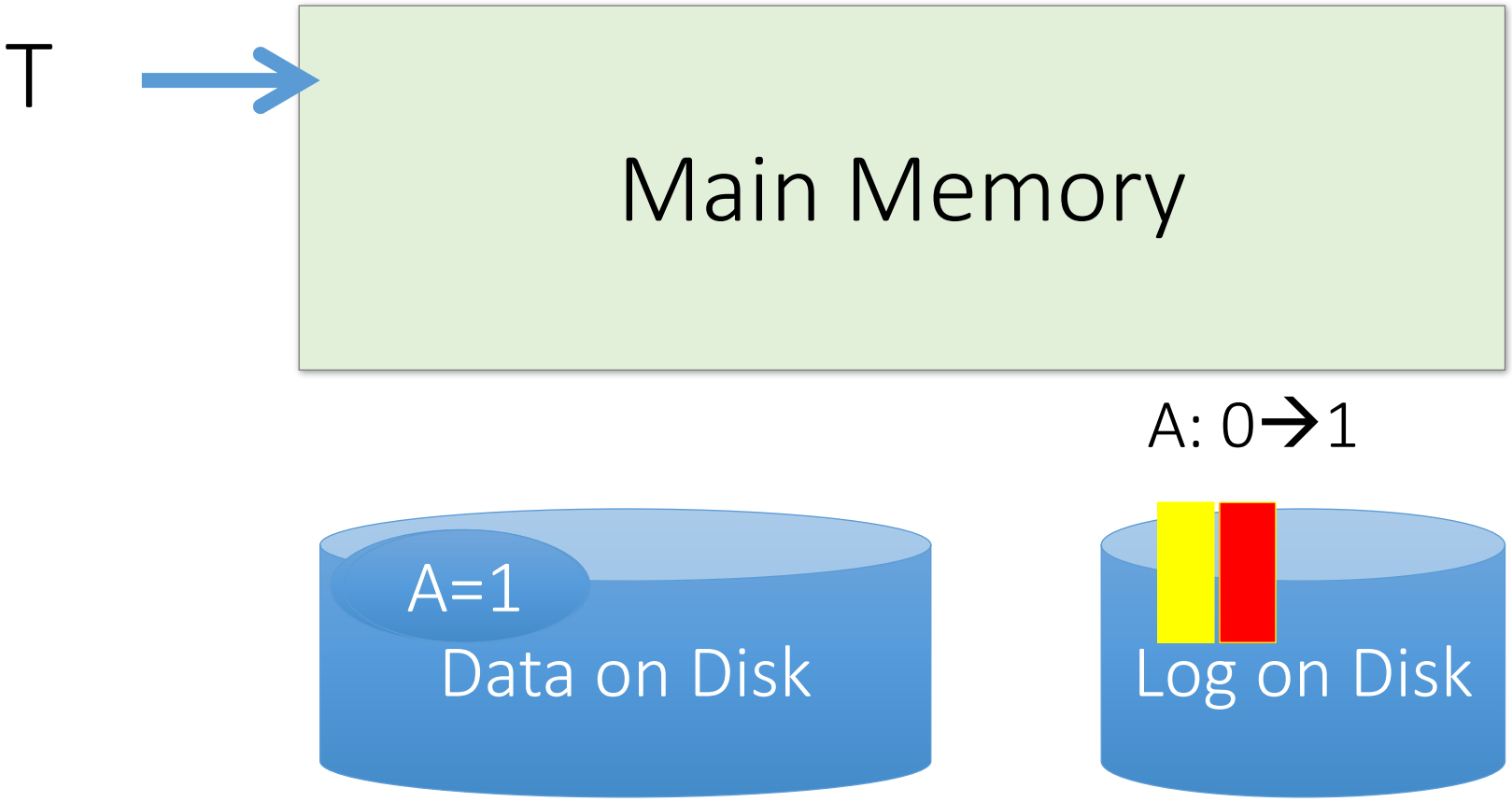
This time, let's try committing ***after*** **we've written log to disk but *before* we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



This time, let's try committing ***after*** **we've written log to disk but *before* we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

DB uses **Write-Ahead Logging (WAL)** Protocol:

1. Log before data: Must *force log record* for an update *before* the corresponding data page goes to storage

→ Atomicity

2. Force log on commit: Must *write all log records* for a TX *before commit*

→ Durability

Transaction is committed *once commit log record is on stable storage*

Undo logging

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Rule 1:
<*T, A,* 8> must be flushed to disk before new *A* is written to disk (same for *B*)

Undo log format:

<*T*, *X*, *v*>: T updated database element X whose old value is v

Rule 2:
<COMMIT *T*> must be flushed to disk after *A* and *B* are written to disk

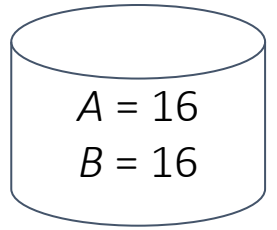
Recovery using undo logging

- Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Ignore (*T* was committed)

Ignore (*T* was committed)

Observe <COMMIT *T*> record

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Crash

If we first meet <END CKPT>, only need to recover until <START CKPT (T1, T2)>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
Crash
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

If we first meet <START CKPT (T1, T2)>, only need to recover until <START T1>

Redo logging

Redo logging ignores incomplete transactions and repeats committed ones

- Undo logging cancels incomplete transactions and ignores committed ones

<*T*, *X*, *v*> now means *T* wrote new value *v* for database element *X*

One rule: all log records (e.g., <*T*, *X*, *v*> and <COMMIT *T*>) must appear on disk before modifying any database element *X* on disk

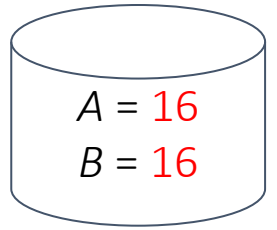
Recovery with redo logging

- Scan log forward and redo committed transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



Nonquiescent checkpointing for redo log

- Write to disk all DB elements modified by committed transactions

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Write to disk all DB elements by transactions that already committed when START CKPT was written to log (i.e., T1)

Nonquiescent checkpointing for redo log

- After crash, redo committed transactions that either started after START CKPT or were active during START CKPT

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Undo/redo logging

More flexible than undo or redo logging in ordering actions

<*T*, *X*, *v*, *w*> : *T* changed value of *X* from *v* to *w*

One rule: <*T*, *X*, *v*, *w*> must appear on disk before modifying *X* on disk

Nonquiescent checkpointing for undo/redo logging

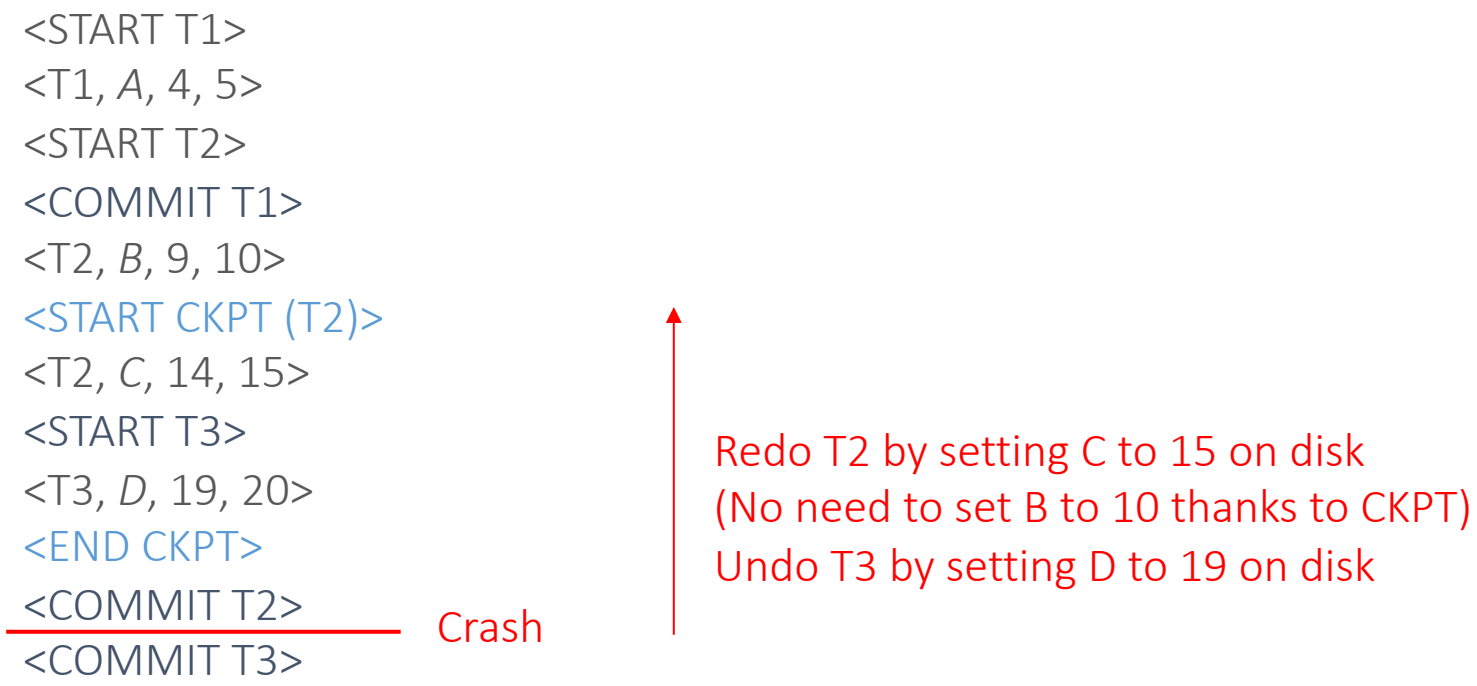
- Simpler than other logging methods

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>

Write to disk all the buffers that are dirty

Nonquiescent checkpointing for undo/redo logging

- After a crash, redo committed transactions, and undo uncommitted ones



Practice Question

- Exam 2: 7.1 Recovery

Schedules and Serializability

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions

- A and B are **equivalent schedules** if, *for any database state*, the effect on DB of executing A **is identical to** the effect of executing B

- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some**" makes this definition powerful & tricky!

Conflicts: Anomalies with Interleaved Execution

Conditions for conflicts:

- The operations must belong to different transactions (no conflict within the same transaction).
- The operations must access the same database object
- At least one of the operations must be a write operation.

Types of conflicts:

- Write-Read (WR)
- Read-Write (RW)
- Write-Write (WW)

Implication for schedules:

A pair of consecutive actions that cannot be interchanged without changing behavior

DB isolation levels define which types of conflicts a database will prevent or allow.

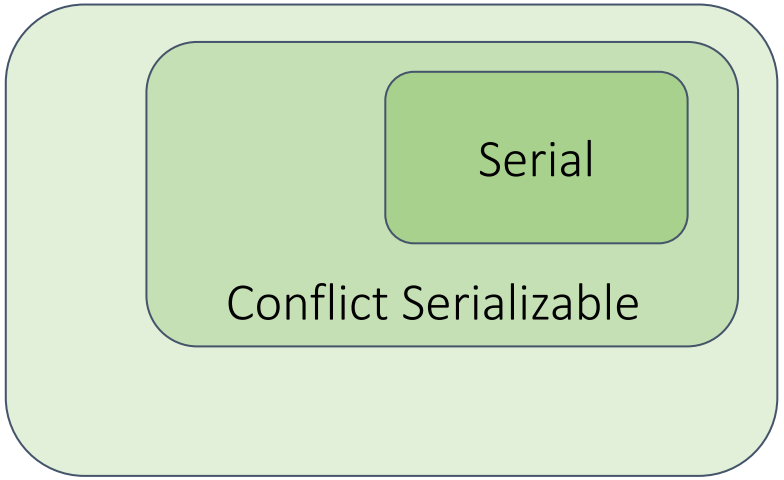
Characterizing Schedules based on Serializability (2)

Conflict equivalent

- Two conflict equivalent schedules have the same effect on a database
- All pairs of conflicting actions are in same order
- one schedule can be obtained from the other by swapping “non-conflicting” actions
  - either on two different objects
  - or both are read on the same object

Conflict serializable

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.



Testing for conflict serializability

Through a precedence graph:

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

Practice Question

- Exam 2: 4. Serializability
  - 4.1 Conflicts and Serializability
  - 4.2 Repairing History