

CS 6400 A

Database Systems Concepts and Design

Lecture 14

10/16/24

Announcement

- Assignment 2 due next Monday (Oct 21)

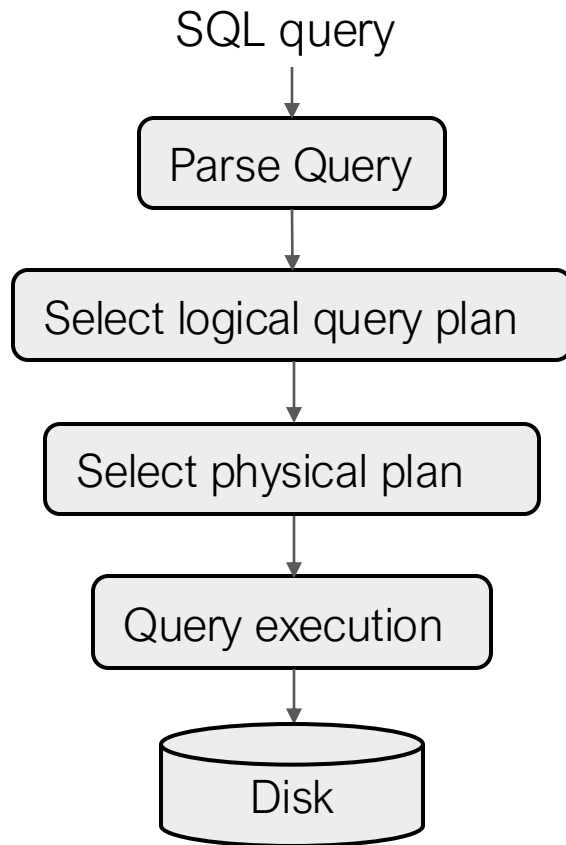
Leaderboard

Search

◆ Rank	◆ Submission Name	▲ Time
In-Memory Data Layout	<u>Ignacio Di Leva</u>	298.2678573796529
2	<u>Zihao</u>	947.8287718555131
3	<u>Puxuan</u>	1148.984366648172

Recap: RDBMS Architecture

How does a SQL engine work ?



Translate to RA expression and find logically equivalent but more efficient plans (Lecture 12)

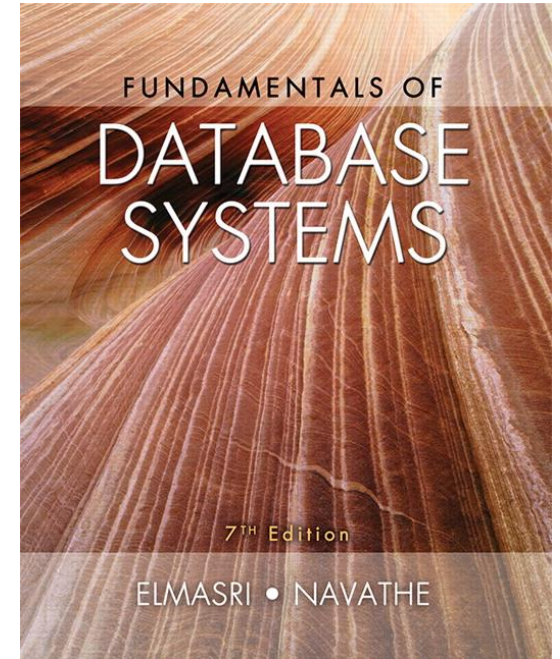
Cost-based query optimization: estimate cost and select physical plan with the smallest cost (Lecture 13)

Query execution (e.g., run join algorithms against tuples on disk) (Lecture 11)

Reading Materials

Fundamental of Database Systems (7th Edition)

- Chapter 20 - Introduction to Transaction Processing Concepts and Theory



Acknowledgement:

The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Agenda

1. Transaction Basics
2. ACID properties
3. Using transactions in SQL
4. Schedule

1. Transaction Basics

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
```

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

Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

Transactions in SQL

- In “ad-hoc” SQL:
 - Default: each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
```

```
UPDATE Bank SET amount = amount - 100
```

```
WHERE name = 'Bob'
```

```
UPDATE Bank SET amount = amount + 100
```

```
WHERE name = 'Joe'
```

```
COMMIT
```

Model of Transaction in this class

We assume that the DBMS is only concerned about reads and writes to data

- A user's program may carry out many operations on the data retrieved from the database

A **transaction** is the DBMS's abstract view of a user program

- The same program executed multiple times would be considered as different transactions
- The DBMS does not really understand the “semantics” of the data, it only cares about read and write sequences

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

Motivation

1. Recovery & Durability of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full, or not at all**; keep log to be able to “roll-back” TXNs

Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99
```

Crash / abort!

```
DELETE Product
WHERE price <=0.99
```

What goes wrong?

Protection against crashes / aborts

Client 1:

START TRANSACTION

INSERT INTO SmallProduct(name, price)

SELECT pname, price

FROM Product

WHERE price <= 0.99

DELETE Product

WHERE price <=0.99

COMMIT OR ROLLBACK

Now we'd be fine!

Motivation

2. Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs busy...

Multiple users: single statements

Client 1: **UPDATE** Product
SET Price = Price – 1.99
WHERE pname = 'Gizmo'

Client 2: **UPDATE** Product
SET Price = Price*0.5
WHERE pname='Gizmo'

Two managers attempt to discount products *concurrently*-
What could go wrong?

Multiple users: single statements

Client 1: START TRANSACTION

UPDATE Product

SET Price = Price - 1.99

WHERE pname = 'Gizmo'

COMMIT

Client 2: START TRANSACTION

UPDATE Product

SET Price = Price*0.5

WHERE pname='Gizmo'

COMMIT

Now works like a charm - we'll see how / why next lecture...

2. ACID Properties

Desirable Properties of Transactions: ACID

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed.
- **Durability**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

ACID: Atomicity

TXN's activities are atomic: **all or nothing**

- Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*

Two possible outcomes for a TXN

- It *commits*: all the changes are made
- It *aborts*: no changes are made

ACID: Consistency

The tables must always satisfy user-specified *integrity constraints*

- *Examples:*
 - Account number is unique
 - Stock amount can't be negative
 - Sum of *debits* and of *credits* is 0

How consistency is achieved:

- Programmer makes sure a txn takes a consistent state to a consistent state
- *System* makes sure that the txn is **atomic**

ACID: Isolation

A transaction executes concurrently with other transactions

Isolation: the effect is as if each transaction executes in *isolation* of the others.

- e.g. should not be able to observe changes from other transactions during the run

ACID: Durability

The effect of a TXN must continue to exist (“*persist*”) after the TXN

- And after the whole program has terminated
- And even if there are power failures, crashes, etc.
- And etc...

- Means: Write data to **disk**

Change on the horizon?
Non-Volatile Ram (NVRam).
Byte addressable.

Ensuring Consistency

User's responsibility to maintain the integrity constraints, as the DBMS may not be able to catch such errors in user program's logic

- e.g., if you transfer money from the savings account to the checking account, the total amount still remains the same

However, the DBMS may be in inconsistent state “during a transaction” between actions

- which is ok, but it should leave the database at a consistent state when it commits or aborts

Ensuring Atomicity

Transactions can be incomplete due to several reasons

- Aborted (terminated) by the DBMS because of some anomalies during execution
 - in that case automatically restarted and executed anew
- The system may crash (e.g., no power supply)
- A transaction may decide to abort itself encountering an unexpected situation
 - e.g., read an unexpected data value or unable to access disks

Ensuring Atomicity

A transaction interrupted in the middle can leave the database in an inconsistent state

- DBMS has to remove the effects of partial transactions from the database

DBMS ensures atomicity by “undoing” the actions of incomplete transactions

DBMS maintains a “log” of all changes to do so

Ensuring Durability

The **log** also ensures durability

If the system crashes before the changes made by a completed transactions are written to the disk, the log is used to remember and restore these changes when the system restarts

“**recovery manager**”

- takes care of atomicity and durability

Ensuring Isolation

DBMS guarantees isolation

- If T1 and T2 are executed concurrently, either the effect would be T1->T2 or T2->T1 (and from a consistent state to a consistent state)

But DBMS provides no guarantee on which of these order is chosen

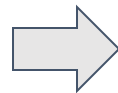
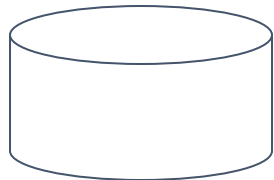
Often ensured by “locks” but there are other methods too

The Correctness Principle

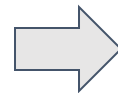
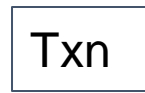
A fundamental assumption about transaction is:

If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transactions ends.

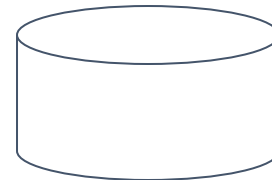
DB in consistent state



Run in isolation



DB in consistent state



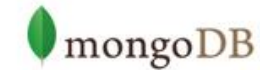
A Note: ACID is contentious!

Many debates over ACID, both **historically** and **currently**



Many newer “NoSQL” DBMSs relax ACID

In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs...

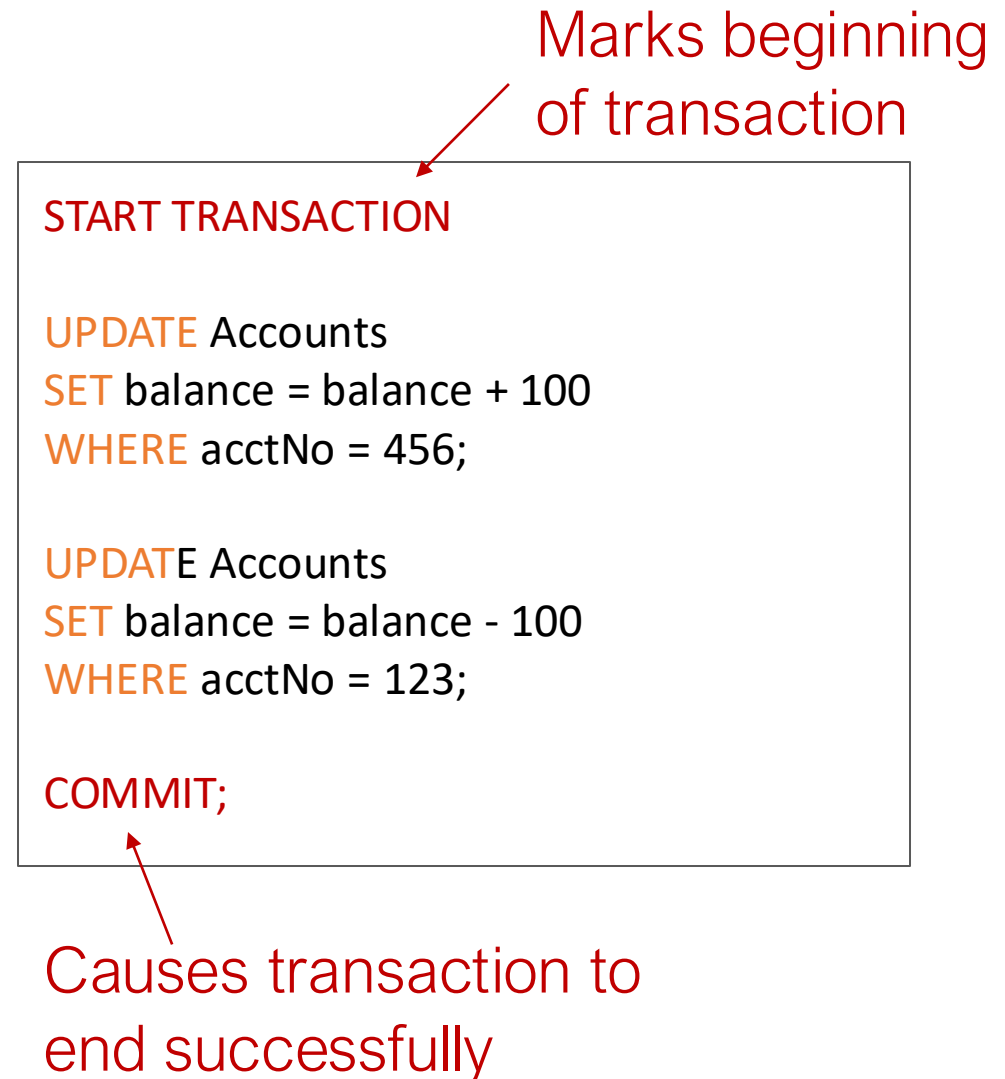


ACID is an extremely important & successful paradigm, but still debated!

3. Using Transactions in SQL

Using Transactions in SQL

- SQL allows the programmer to group several statements in a single *transaction*
- Either all operations are performed or none are
- A single SQL statement is always considered to be **atomic**.



Using Transactions in SQL

- ROLLBACK causes the transaction to abort and undo any changes

We find that there are insufficient funds to make transfer

```
START TRANSACTION
```

```
UPDATE Accounts
```

```
SET balance = balance + 100
```

```
WHERE acctNo = 456;
```

```
ROLLBACK;
```

Using Transactions in SQL

```
SET TRANSACTION transaction_mode [, ...]
```

where *transaction_mode* is one of:

- ISOLATION LEVEL {
 SERIALIZABLE
 | REPEATABLE READ
 | READ COMMITTED
 | READ UNCOMMITTED }
- READ WRITE | READ ONLY

Isolation Levels

- With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
- However, if any transaction executes at a lower level, then serializability may be violated.

Access Mode

- The default is READ WRITE unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed.

Read-only transactions

Transactions that only read data and do not write can be executed in parallel

Tell DBMS before running transaction:

```
SET TRANSACTION READ ONLY;
```

Dirty reads

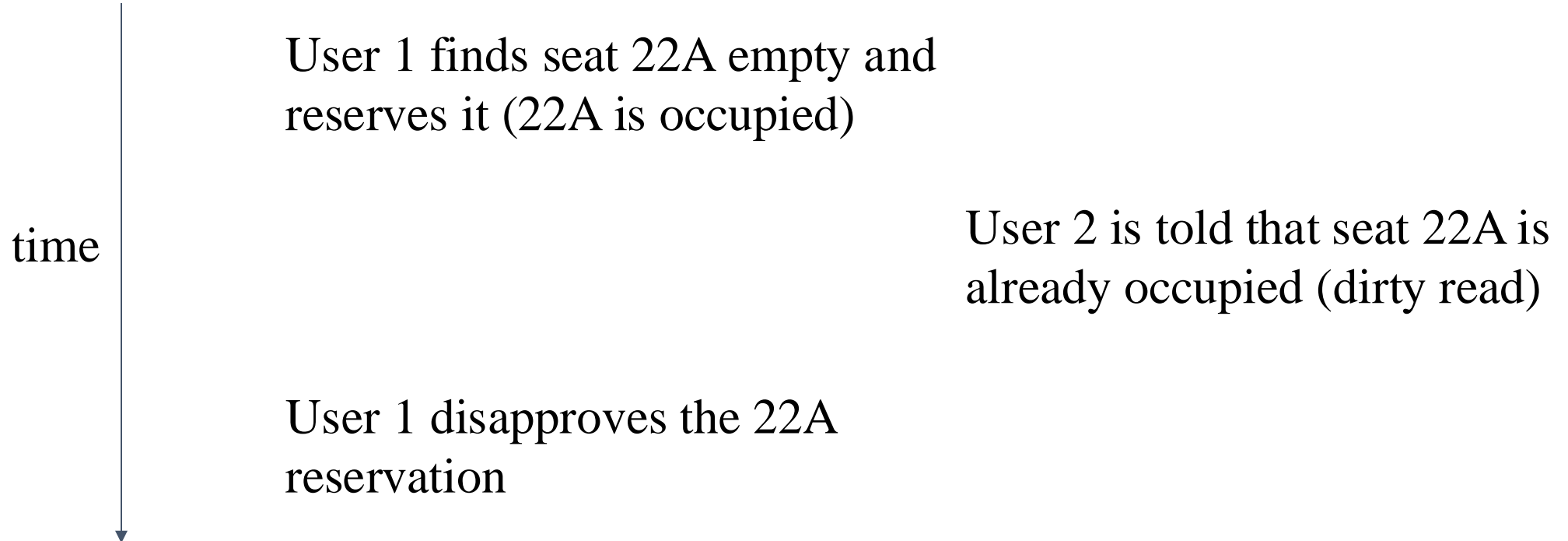
Reading data written by a transaction that has not yet committed

Consider this seat selection example:

1. Find available seat and reserve by setting *seatStatus* to 'occupied'
2. Ask customer for approval of seat
 - a. If so, commit
 - b. If not, release seat by setting *seatStatus* to 'available' and repeat Step (1)

Dirty read

If we allow dirty reads, this can happen



Dirty reads

If this result is acceptable, the transaction processing can be done faster

- DBMS does not have to prevent dirty reads
- Allows more parallelism

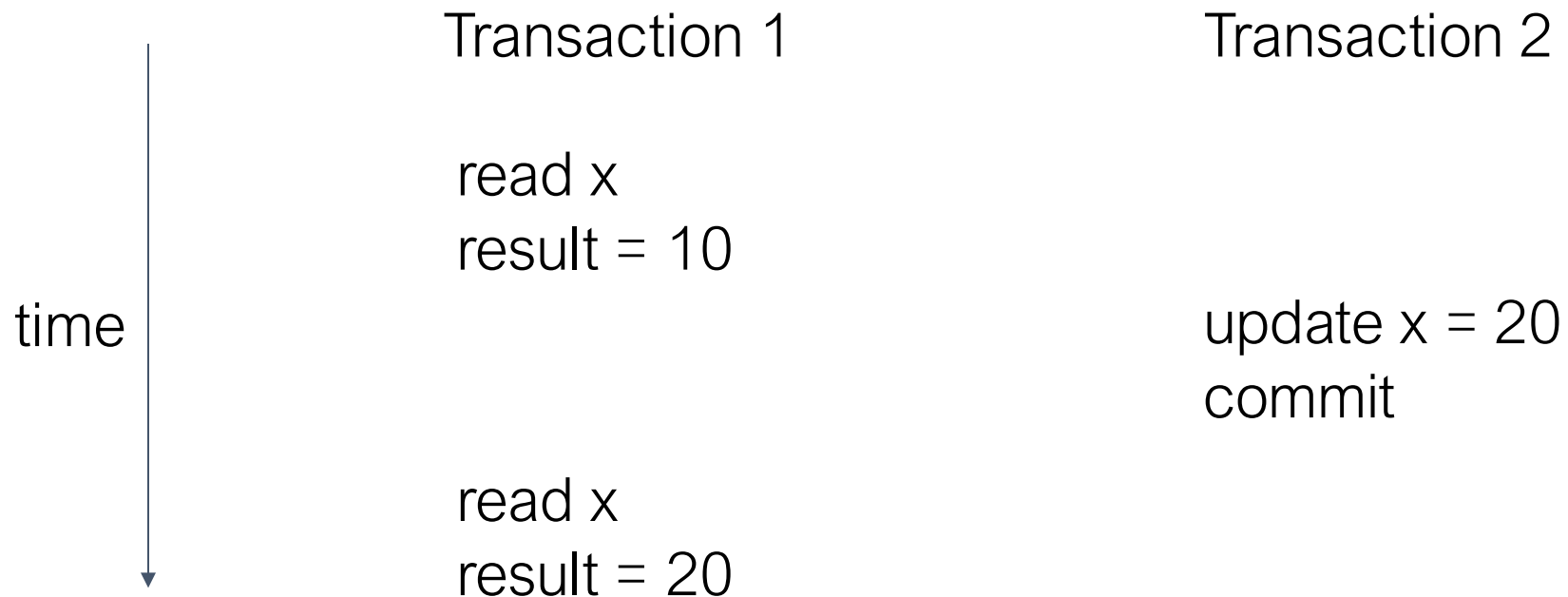
Tell DBMS before running transaction:

```
SET TRANSACTION READ WRITE  
ISOLATION LEVEL READ UNCOMMITTED;
```

Read committed

Only allow reads from committed data, but same query may get different answers

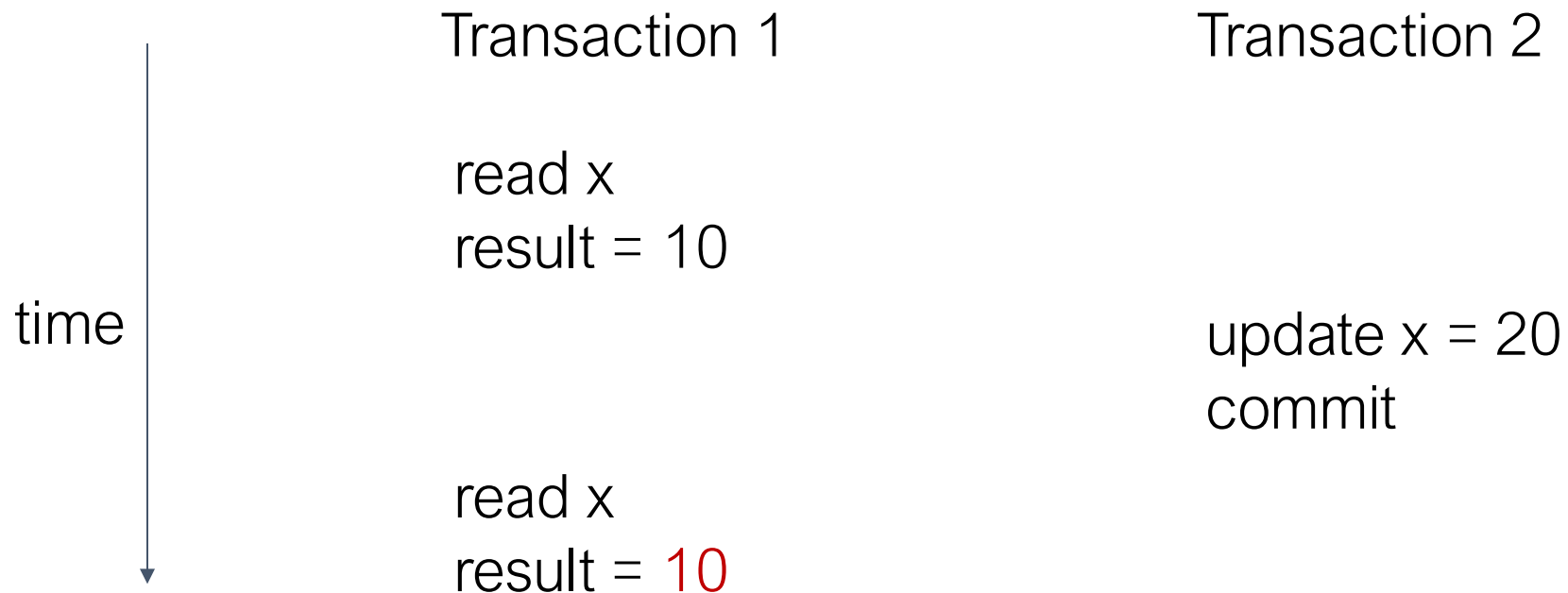
```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```



Repeatable read

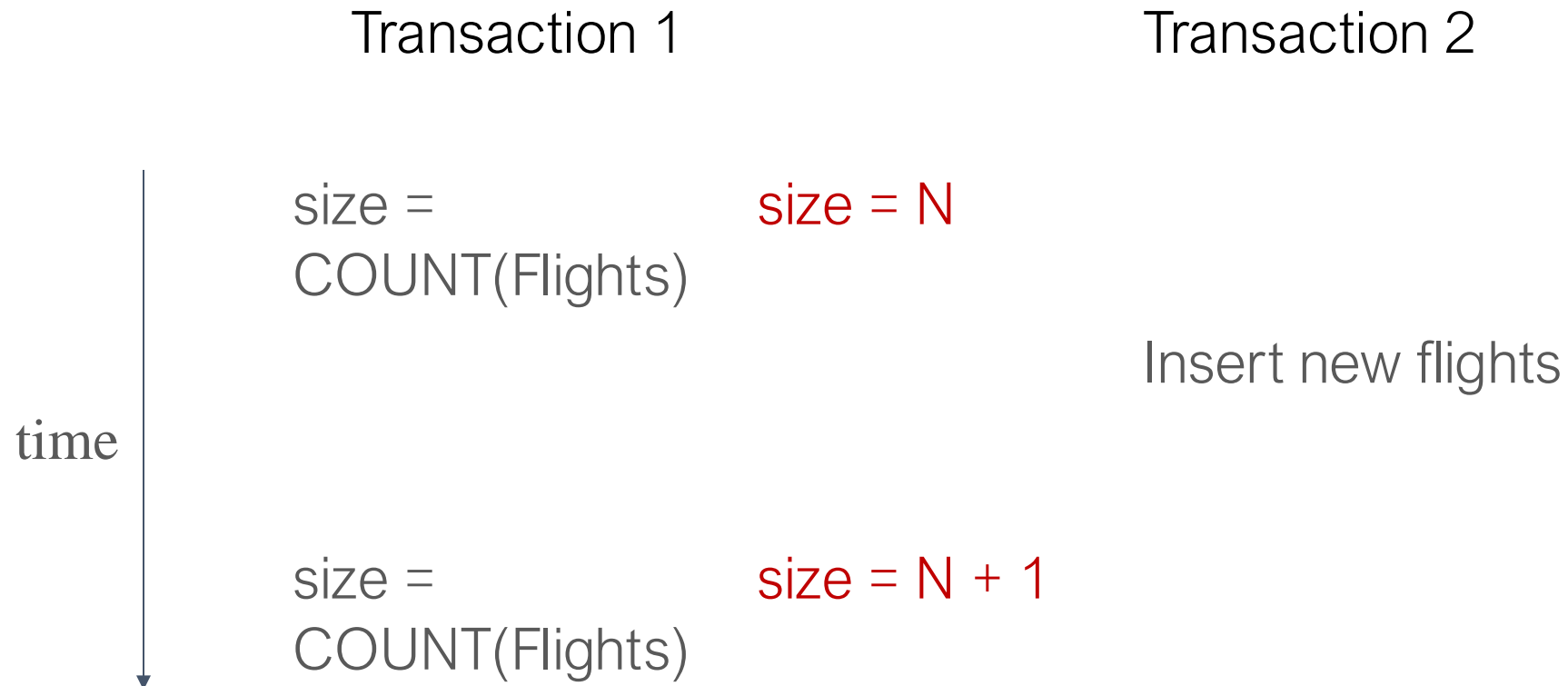
Any tuple that was retrieved will be retrieved again if the same query is repeated, even though other transactions may modify the individual rows that were read.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```



Repeatable read

May allow “phantom” tuples, which are new tuples inserted between queries



Clarifications on Repeatable Read

Guarantee: rows read by a transaction will not change if read again in that transaction.

- Doesn't guarantee anything about rows that weren't originally read.

Why Phantom Reads Can Occur

- Locking: Repeatable read typically locks the rows it reads, but not the gaps between rows.
- New Inserts: Without gap locking, new rows could be inserted that match your WHERE clause.

Comparison of SQL isolation levels

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

Comparison of SQL isolation levels

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

- Rarely used in practice, as the performance is not much better than other levels
- In fact, PostgreSQL doesn't support this isolation level
- No lock on data

Comparison of SQL isolation levels

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

- Fast and simple to use; adequate for many applications
- Shared lock (read lock) on rows when they are read, exclusive lock (write lock) on rows when they are being modified

Comparison of SQL isolation levels

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

- Good for reporting, data warehousing types of workload
- Shared locks on all rows read by a transaction

Comparison of SQL isolation levels

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

- Recommended only when updating transactions contain logic sufficiently complex that they might give wrong answers in READ COMMITTED mode
- Locking the entire range of rows that could potentially be accessed by a transaction's queries

3. Schedule

Schedule

A transaction is seen by DBMS as a list of actions.

- READ, WRITE of database objects
- ABORT, COMMIT

Assumption: Transactions communicate only through READ and WRITE

Schedule is a list of actions from a set of transactions as seen by the DBMS

- Two actions from the same transaction T MUST appear in the schedule in the same order that they appear in T
- Intuitive, a schedule represents an actual or potential execution sequence

Transaction primitives

- INPUT(X): copy block X from disk to memory
- READ(X, t): copy X to transaction's local variable t
(run INPUT(X) if X is not in memory)
- WRITE(X, t): copy value of t to X (run INPUT(X) if X is not in memory)
- OUTPUT(X): copy X from memory to disk

Schedule

- Actions taken by one or more transactions

<i>T1</i>	<i>T2</i>
READ(<i>A</i> , <i>t</i>)	READ(<i>A</i> , <i>s</i>)
<i>t</i> := <i>t</i> +100	<i>s</i> := <i>s</i> *2
WRITE(<i>A</i> , <i>t</i>)	WRITE(<i>A</i> , <i>s</i>)
READ(<i>B</i> , <i>t</i>)	READ(<i>B</i> , <i>s</i>)
<i>t</i> := <i>t</i> +100	<i>s</i> := <i>s</i> *2
WRITE(<i>B</i> , <i>t</i>)	WRITE(<i>B</i> , <i>s</i>)

Characterizing Schedules based on Serializability (1)

Serial schedule

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
 - Otherwise, the schedule is called nonserial schedule.

Serializable schedule

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Serial and serializable schedules are guaranteed to preserve the consistency of database states

Serial schedule

- One transaction is executed at a time

<i>T1</i>	<i>T2</i>	<i>A</i>	<i>B</i>
READ(<i>A</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>A</i> , <i>t</i>) READ(<i>B</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>B</i> , <i>t</i>)		25	25
		125	
			125
	READ(<i>A</i> , <i>s</i>) <i>s</i> := <i>s</i> *2 WRITE(<i>A</i> , <i>s</i>) READ(<i>B</i> , <i>s</i>) <i>s</i> := <i>s</i> *2 WRITE(<i>B</i> , <i>s</i>)	250	
			250

Schedule: (T1, T2)

Q: Do serial schedules allow for high throughput?

Serializable schedule

- There exists a serial schedule with the same effect

<i>T1</i>	<i>T2</i>	<i>A</i>	<i>B</i>
		25	25
READ(<i>A</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>A</i> , <i>t</i>)		125	
	READ(<i>A</i> , <i>s</i>) <i>s</i> := <i>s</i> *2 WRITE(<i>A</i> , <i>s</i>)	250	
READ(<i>B</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>B</i> , <i>t</i>)			125
	READ(<i>B</i> , <i>s</i>) <i>s</i> := <i>s</i> *2 WRITE(<i>B</i> , <i>s</i>)		250

Same effect as (T1, T2)

Serializable schedule

- This is not serializable

<i>T1</i>	<i>T2</i>	<i>A</i>	<i>B</i>
		25	25
READ(<i>A</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>A</i> , <i>t</i>)		125	
	READ(<i>A</i> , <i>s</i>) <i>s</i> := <i>s</i> *2 WRITE(<i>A</i> , <i>s</i>)	250	
	READ(<i>B</i> , <i>s</i>) <i>s</i> := <i>s</i> *2 WRITE(<i>B</i> , <i>s</i>)		50
READ(<i>B</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>B</i> , <i>t</i>)			150

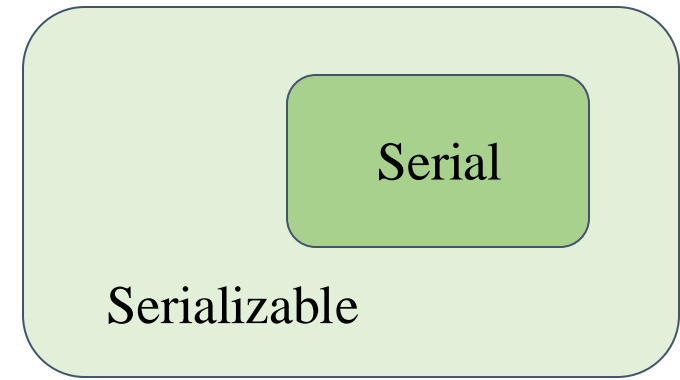
Serializable schedule

- Serializable, but only due to the detailed transaction behavior

<i>T1</i>	<i>T2</i>	<i>A</i>	<i>B</i>
		25	25
READ(<i>A</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>A</i> , <i>t</i>)		125	
	READ(<i>A</i> , <i>s</i>) <i>s</i> := <i>s</i> +200 WRITE(<i>A</i> , <i>s</i>)	325	
	READ(<i>B</i> , <i>s</i>) <i>s</i> := <i>s</i> +200 WRITE(<i>B</i> , <i>s</i>)		225
READ(<i>B</i> , <i>t</i>) <i>t</i> := <i>t</i> +100 WRITE(<i>B</i> , <i>t</i>)			325

Same effect as (T1, T2)

Serial vs Serializable Schedule



Being serializable is not the same as being serial

Being serializable implies that the schedule is a correct schedule.

- It will leave the database in a consistent state.

Interleaving improves efficiency due to concurrent execution, e.g.,

- While one transaction is blocked on I/O, the CPU can process another transaction
- Interleaving short and long transactions might allow the short transaction to finish sooner (otherwise it need to wait until the long transaction is done)

Abstract view of TXNs: reads and writes

Serializability is hard to check - cannot always know detailed behaviors

DBMS's abstract view of transactions:

$r_i(X)$: T_i reads X
 $w_i(X)$: T_i writes X

$T_1: r_1(A); w_1(A); r_1(B); w_1(B)$

$T_2: r_2(A); w_2(A); r_2(B); w_2(B)$

Serializable schedule: $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

Conflicts: Anomalies with Interleaved Execution

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

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

* No conflict with “RR” if no write is involved

WR Conflict

T1: R(A), W(A),	R(B), W(B), Abort
T2: R(A), W(A),	Commit

T1: R(A), W(A),	R(B), W(B), Commit
T2: R(A), W(A),	R(B), W(B), Commit

Reading Uncommitted Data (WR Conflicts, “dirty reads”):

- transaction T2 reads an object that has been modified by T1 but not yet committed
- or T2 reads an object from an inconsistent database state (like fund is being transferred between two accounts by T1 while T2 adds interests to both)

RW Conflict

T1: R(A),	R(A), W(A), C
T2: R(A), W(A), C	

Unrepeatable Reads (RW Conflicts):

- T2 changes the value of an object A that has been read by transaction T1, which is still in progress
- If T1 tries to read A again, it will get a different result
- Suppose two customers are trying to buy the last copy of a book simultaneously

WW Conflict

T1: W(A),	W(B), C
T2: W(A),	W(B), C

Overwriting Uncommitted Data (WW Conflicts, “lost update”):

- T2 overwrites the value of A, which has been modified by T1, still in progress
- Suppose we need the salaries of two employees (A and B) to be the same
 - T1 sets them to \$1000
 - T2 sets them to \$2000

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' .

Conflict-serializable schedule

- Conflict-equivalent to serial schedule

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); r_2(A); w_1(B); w_2(A); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

Serial

Conflict-serializable schedule

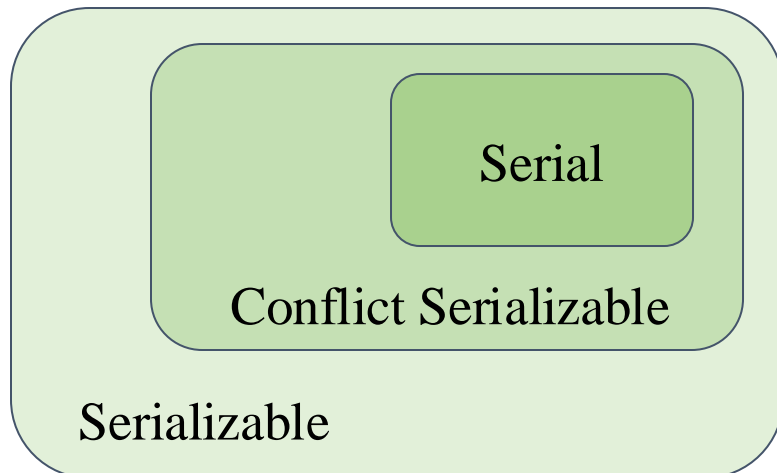
- A conflict-serializable schedule is always serializable
- But not vice versa (e.g., serializable schedule due to detailed transaction behavior)

S1: $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$;

Serial

S2: $w_1(Y)$; $w_2(Y)$; $w_2(X)$; $w_1(X)$; $w_3(X)$;

Serializable, but
not conflict
serializable



In-class Exercise

- What are schedules that are conflict-equivalent to (T1, T2)?

T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

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 T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is serializable if and only if the precedence graph has no cycles.

Precedence graph

Can use to decide conflict serializability

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

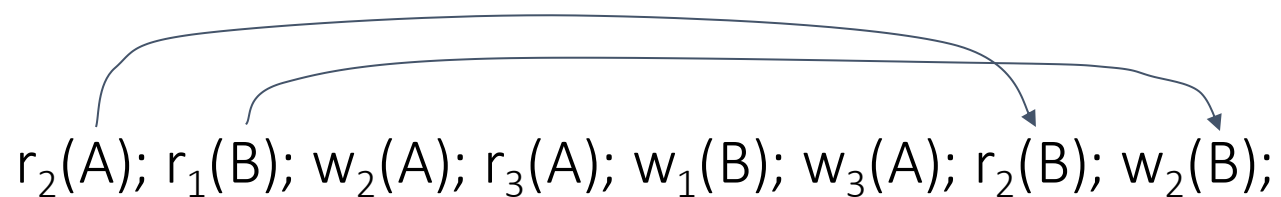
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

** Also called dependency graph, conflict graph, or serializability graph*

Precedence graph

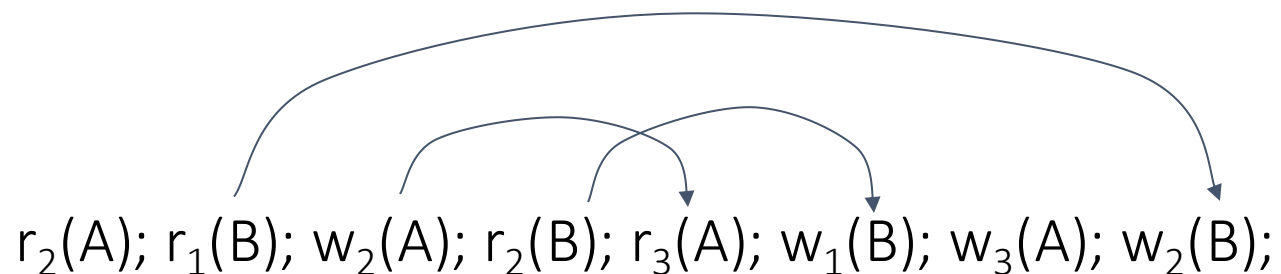
Can use to decide conflict serializability

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$



$T1 \rightarrow T2 \rightarrow T3$

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

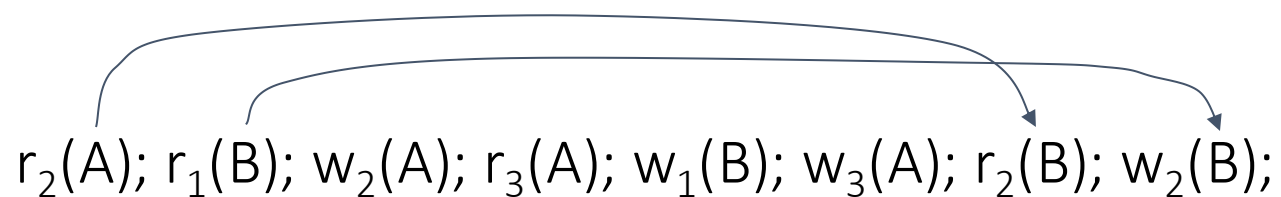


$T1 \rightarrow T2 \rightarrow T3$

- One node per committed transaction
- Edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions
 - $W_i(A) \text{ --- } R_j(A)$, or $R_i(A) \text{ --- } W_j(A)$, or $W_i(A) \text{ --- } W_j(A)$

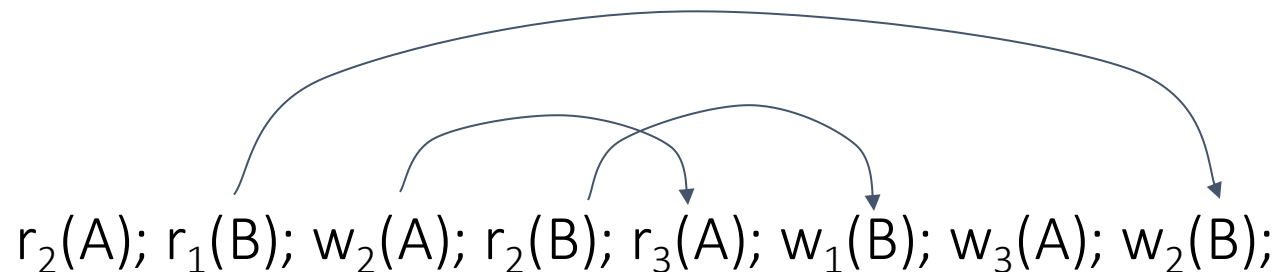
Precedence graph

Can use to decide conflict serializability



This is conflict serializable

$T1 \rightarrow T2 \rightarrow T3$



This is not because of cycle

$T1 \rightarrow T2 \rightarrow T3$

- One node per committed transaction
- Edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions
 - $W_i(A) \text{ --- } R_j(A)$, or $R_i(A) \text{ --- } W_j(A)$, or $W_i(A) \text{ --- } W_j(A)$

In-class Exercise

- What is the precedence graph for the schedule:

$r_1(A); r_2(A); r_1(B); r_2(B); r_3(A); r_4(B); w_1(A); w_2(B);$

- One node per committed transaction
- Edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions
 - $W_i(A) \text{ --- } R_j(A)$, or $R_i(A) \text{ --- } W_j(A)$, or $W_i(A) \text{ --- } W_j(A)$