CS 4604: Introduction to Database Management Systems

Transactions 1: Intro. to ACID

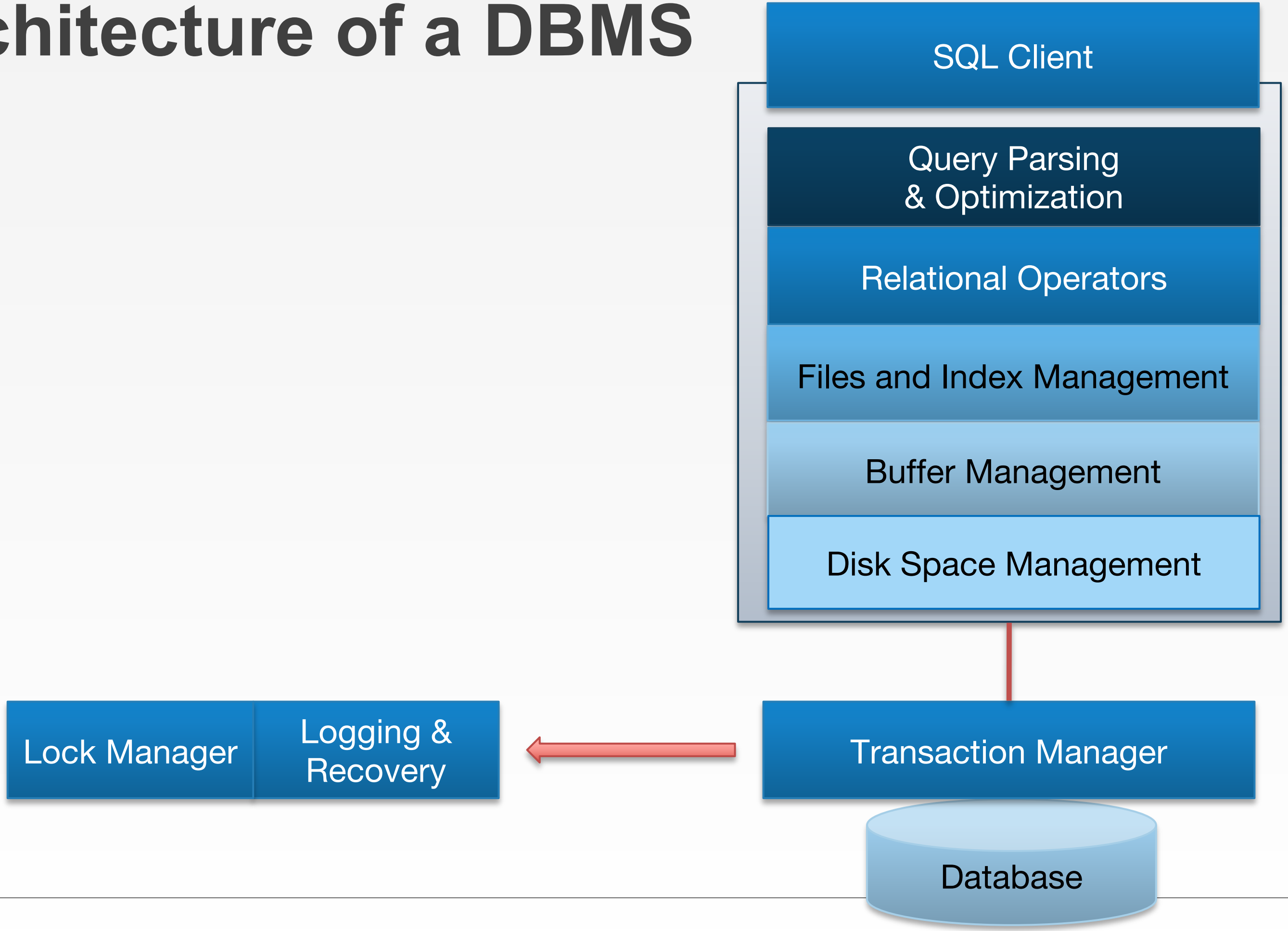
Virginia Tech CS 4604 Sprint 2021

Instructor: Yinlin Chen

Today’s Topics

- ACID
- Transaction management

Architecture of a DBMS



Concurrency Control & Recovery

- **Part 1: Concurrency Control**
  - Correct/fast data access in the presence of concurrent work by many users
  - Disorderly processing that provides the illusion of order
- **Part 2: Recovery**
  - Ensure database is fault tolerant
  - Not corrupted by software, system or media failure
  - Storage guarantees for mission-critical data
- **It's all about the programmer!**
  - Systems provide guarantees
  - These guarantees lighten the load of app writers

What is a Transaction?

- A *sequence* of *multiple actions* to be executed as an *atomic* unit
  - a sequence of read and write operations (read(A), write(B), …)
  - DBMS’s abstract view of a user program
- Application View (SQL View):
  - Begin transaction
  - Sequence of SQL statements
  - End transaction
- Examples
  - Transfer money between accounts
  - Book a flight, a hotel and a car together on Expedia

Transaction

- Transaction (“Xact”):
  - A sequence of reads and writes of database objects
  - Batch of work that must commit or abort as an atomic unit
- **Xact Manager** controls execution of transactions
- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike operating systems, which support interaction of processes, a DMBS needs to keep processes from troublesome interactions.
- Program logic is **invisible** to DBMS!
  - Arbitrary computation possible on data fetched from the DB
  - The DBMS **only sees data read/written from/to the DB**

Transaction Example

- Transaction to transfer $100 from account R to account S

Not seen by the DBMS transaction manager!

1. start transaction
2. read(R)
3. R = R – 100
4. write(R)
5. read(S)
6. S = S + 100
7. write(S)
8. end transaction

ACID: Properties of Transactions

- **A**tomicity: Either all actions in the transaction happened or none happen
- **C**onsistency: If the DB starts out consistent, it ends up consistent at the end of the transaction
- **I**solation: It appears to the user as if only one process executes at a time. Each transaction is isolated from that of others
- **D**urability: If a transaction is completed, its effects should persist even if the system survives a crash

Atomicity of Transactions

Two possible outcomes of executing a transaction:

– Transaction might *commit* after completing all its actions

– or it could *abort* (or be aborted by the DBMS) after executing some actions

  - Or system crash while the transaction is in progress; treat as abort

DBMS guarantees that transaction are *atomic.*

– From user's point of view: transaction always either executes all its actions, or executes no actions at all

COMMIT

- The SQL statement COMMIT causes a transaction to complete
  - It is database modifications are now permanent in the database
  - The effects of a committed transaction must survive failures
- DBMS typically ensures the above by logging all actions:
  - **Redo** actions of committed transactions not yet propagated to disk when system crashes

ROLLBACK

The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*

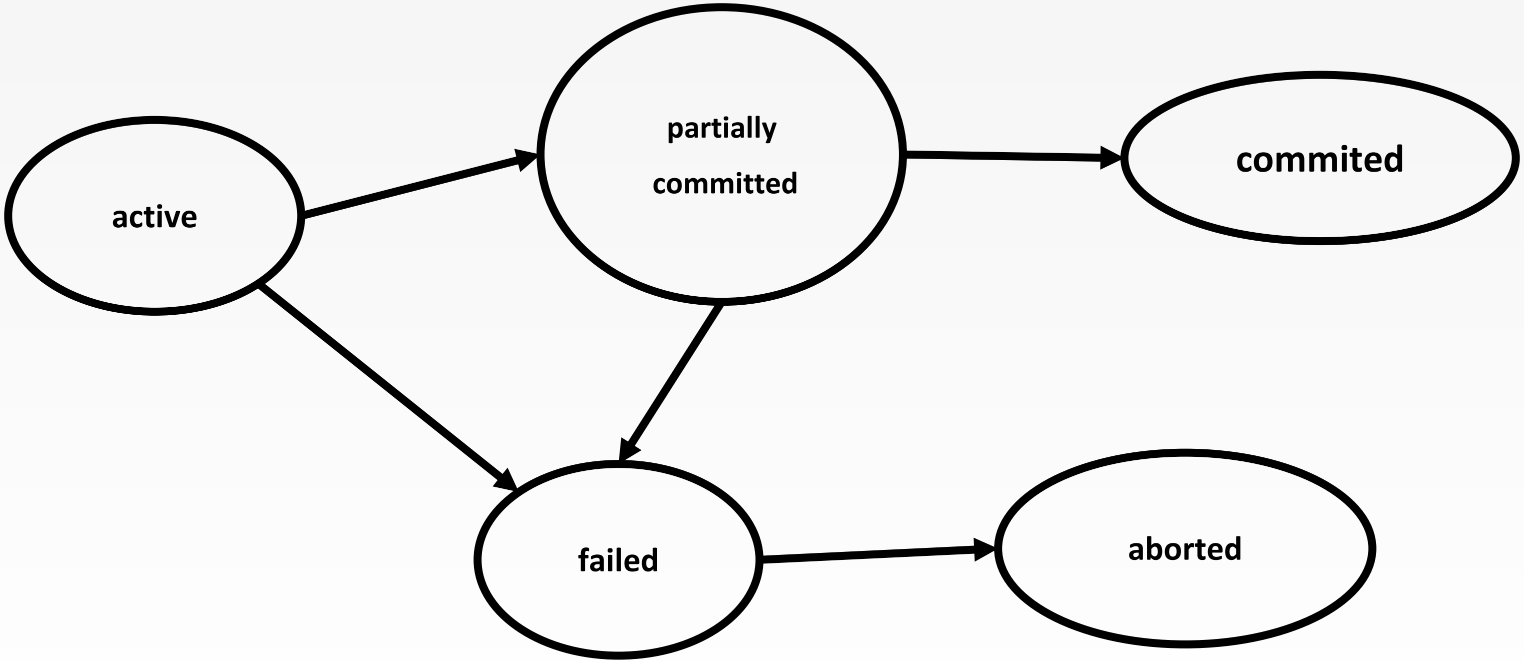
- No effects on the database

Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it

DBMS typically ensures the above by logging all actions:

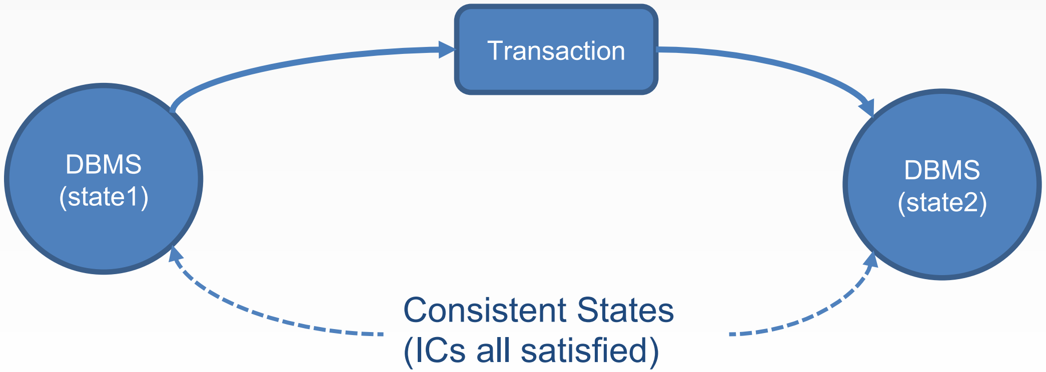
- **Undo** the actions of aborted/failed transactions

Transaction states



Transaction Consistency

- **Transactions preserve DB consistency**
  - Given a consistent DB state, produce another consistent DB state
- DB consistency expressed as a set of **declarative integrity constraints**
  - CREATE TABLE/ASSERTION statements
- **Transactions that violate integrity are aborted**
  - That's all the DBMS can automatically check!



Isolation (Concurrency)

- DBMS interleaves actions of many transactions
  - Actions = reads/writes of DB objects

- DBMS ensures 2 transactions do not "interfere"

- Each transaction executes as if it ran by itself
  - Concurrent accesses have no effect on transaction's behavior
  - Net effect must be identical to executing all transactions in some serial order
  - Users & programmers think about transactions in isolation
    - Without considering effects of other concurrent transaction!

Isolation: An Example

- Think about avoiding problems due to concurrency
  - If another transaction T2 accesses R and S between steps 4 and 5 of T1, it will see a lower value for R+S.

T1

1. start transaction
2. read(R)
3. R = R – 100
4. write(R)
5. read(S)
6. S = S + 100
7. write(S)
8. end transaction

T2

1. start transaction
2. read(R)
3. print(R+S)
4. end transaction

- Isolation easy to achieve by running one Xact at a time
  - However, recall that serial execution is not desirable

Durability

- The effects of a committed transaction must survive failures
- We will talk more about this in logging and recovery

ACID properties

Atomicity (all or none)
Consistency
Isolation (as if alone)
Durability

Logging and recovery
Concurrency control

- ACID Transactions make guarantees that
  - Improve performance (via concurrency)
  - Relieve programmers of correctness concerns
    - Hide concurrency and failure handling!

- Two key issues to consider, and mechanisms
  - **Concurrency control** (via two-phase locking)
  - Recovery (via write-ahead logging WAL)

Concurrent Execution

- Multiple transactions are allowed to run concurrently in the system.
- *Throughput* (transactions per second):
  - Increase processor/disk utilization → more transactions per second (TPS) completed
    - Single core: can use the CPU while another is reading to/writing from the disk
    - Multicore: ideally, scale throughput in the number of processors
- *Latency* (response time per transaction):
  - Multiple transactions can run at the same time rather than waiting for earlier ones to finish
  - So one transaction's latency need not be dependent on another unrelated transaction
  - Lightweight transactions are not bottlenecked on more time-consuming ones to finish
  - Or that's the hope

Statement of problems

Arbitrary interleaving can lead to

– Temporary inconsistency (ok, unavoidable)

– “Permanent” inconsistency (bad!)

Inconsistent Reads: A user reads only part of what was updated

Lost Update: Two users try to update the same record so one of the updates gets lost

Dirty Reads: One user reads an update that was never committed

Example: 'Inconsistent Reads' problem

User 1

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

DELETE Product
WHERE price <= 0.99
```

User 2

```
SELECT count(*)
FROM Product

SELECT count(*)
FROM DollarProducts
```

Example: ‘Lost-update’ problem

time

T1	T2
Read(N)	
	Read(N)
N=N-1	
	N= N-1
Write(N)	
	Write(N)

Example: ‘Dirty Reads’ problem

User 1

```
UPDATE Account
SET amount = 1000000
WHERE number = “my-account”
```

Aborted by the system

User 2

```
SELECT amount
FROM Account
WHERE number = “my-account”
```

Concurrency Control: Providing Isolation

- **Naïve approach - serial execution**
  - One transaction runs at a time
  - Safe but slow
- **Execution must be interleaved for better performance**
- With concurrent executions, how does one **define** and **ensure** correctness?

Transaction Schedules

T1	T2
begin	
read(A)	
write(A)	
read(B)	
write(B)	
commit	
	begin
	read(A)
	write(A)
	read(B)
	write(B)
	commit

A **schedule** is a sequence of actions on data from **one or more** transactions.

Actions: Begin, Read, Write, Commit and Abort.

$$R_1(A)\ W_1(A)\ R_1(B)\ W_1(B)\ R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)$$

By convention we only include committed transactions, and omit
Begin and Commit.

Serial Equivalence

- Concept for correct behavior

- **Definition**: **Serial schedule**
  - Each transaction runs from start to finish **without any intervening** actions from other transactions

- **Definition**: two schedules are **equivalent** if they:
  - involve the same transactions
  - each individual transaction's actions **are ordered the same**
  - both schedules leave the DB in the **same final state**
  - For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule

Serializability

- **Definition**: Schedule S is **serializable** if:
  - S is equivalent to some serial schedule
  - Results are equivalent to some serial execution of the transactions
- Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Serializable Schedule

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

- Let T1 transfer $100 from A to B
- Let T2 add 10% interest to A & B

- Final outcome:
  - A = 1.1*(A-100)
  - B = 1.1*(B+100)

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

Schedule 1

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

- Let T1 transfer $100 from A to B
- Let T2 add 10% interest to A & B
- Serial schedule in which T1 is followed by T2
  - Final outcome:
    - A := 1.1*(A-100)
    - B := 1.1*(B+100)

Schedule 2

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	

- Serial schedule in which T2 is followed by T1
  - Final outcome:
    - A := (1.1*A)-100
    - B := (1.1*B)+100
  - Different!
    - But still understandable

Schedule 3

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

- Schedule in which actions of T1 and T2 are interleaved.
- This is not a serial schedule
- But it is **equivalent to schedule 1**
  - A := (A-100)*1.1
  - B := (B+100)*1.1
- Hence **serializable**!

Conflicting Operations

- Tricky to check property **"leaves the DB in the same final state"**
- Need an easier equivalence test!
  - Settle for a "conservative" test: always true positives, but some false negatives
  - I.e., sacrifice some concurrency for easier correctness check

- **Use notion of "conflicting" operations (read/write)**

- **Definition: Two operations conflict if they:**
  - Are by different transactions,
  - Are on the same object,
  - At least one of them is a write.

- The order of non-conflicting operations has no effect on the final state of the database!
  - Focus our attention on the order of **conflicting operations**

Anomalies with interleaved execution:

- **Two operations conflict if they:**
  - Are by different transactions,
  - Are on the same object,
  - At least one of them is a write.

- WR conflicts
- RW conflicts
- WW conflicts

Anomalies with Interleaved Execution

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

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

Anomalies with Interleaved Execution

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

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

Anomalies with Interleaved Execution

Unrepeatable Reads (RW Conflicts):

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

Anomalies with Interleaved Execution

Unrepeatable Reads (RW Conflicts):

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

Anomalies (Continued)

Overwriting Uncommitted Data (WW Conflicts):

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

Anomalies (Continued)

Overwriting Uncommitted Data (WW Conflicts):

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

Serializability

Objective: find non-serial schedules, which allow transactions to execute concurrently without interfering, thereby producing a DB state that could be produced by a serial execution

BUT

– Trying to find schedules equivalent to serial execution is too slow!

Conflict Serializable Schedules

- **Definition: Two schedules are *conflict equivalent* if:**
  - They involve the same actions of the same transactions, and
  - Every pair of conflicting actions is ordered the same way

- **Definition: Schedule S is *conflict serializable* if:**
  - S is conflict equivalent to some serial schedule
  - Implies S is also Serializable

Note: some serializable schedules are NOT conflict serializable

- Conflict serializability gives false negatives as a test for serializability!
- The cost of a conservative test
- A price we pay to achieve efficient enforcement

Conflict Serializability - Intuition

- **A schedule S is conflict serializable if**
  - You are able to transform S into a **serial schedule** by swapping **consecutive non-conflicting** operations of different transactions

- ***Example***

R(A) W(A)		R(B) W(B)	
	R(A) W(A)		R(B) W(B)

Conflict Serializability – Intuition, Part 2

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***

R(A)	W(A)			R(B)	W(B)		
		R(A)	W(A)			R(B)	W(B)

R(A)	W(A)			R(B)	W(B)		
		R(A)	W(A)			R(B)	W(B)

Conflict Serializability – Intuition, Part 3

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- *Example*

R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)

Conflict Serializability – Intuition, Part 4

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***

R(A)	W(A)			R(B)	W(B)		
		R(A)	W(A)			R(B)	W(B)

R(A)	W(A)		R(B)	W(B)			
		R(A)			W(A)	R(B)	W(B)

Conflict Serializability – Intuition, Part 5

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***

R(A)	W(A)			R(B)	W(B)		
		R(A)	W(A)			R(B)	W(B)

R(A)	W(A)	R(B)		W(B)			
			R(A)		W(A)	R(B)	W(B)

VIRGINIA TECH™

Conflict Serializability – Intuition, Part 6

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- *Example*

R(A) W(A) R(B) W(B)
R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)
R(A)W(A) R(B) W(B)

Conflict Serializability (Continued)

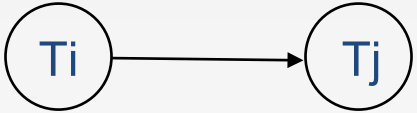
- Here's another example:

R(A)			W(A)
	R(A)	W(A)	

- Conflict Serializable or not?

NOT!

Conflict Dependency Graph



- **Dependency Graph:**
  - One node per Xact
  - Edge from Ti to Tj if:
    - An operation Oi of Ti conflicts with an operation Oj of Tj and
    - Oi appears earlier in the schedule than Oj

- **Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.**

  Proof Sketch: Conflicting operations prevent us from “swapping” operations into a serial schedule

Example

- **A schedule that is not conflict serializable**

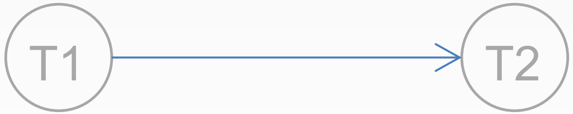
T1:   R(A), W(A)

Dependency graph

Example, pt 2

- **A schedule that is not conflict serializable**

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

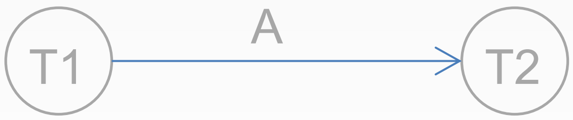


Dependency graph

Example, pt 3

- **A schedule that is not conflict serializable**

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

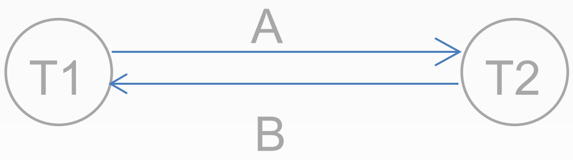


Dependency graph

Example, pt 4

- **A schedule that is not conflict serializable**

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



Dependency graph

View Serializability

- **Alternative notion of serializability: fewer false negatives**
- **Schedules S1 and S2 are view equivalent if:**
  - *Same initial reads*:
    - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - *Same dependent reads*:
    - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - *Same winning writes*:
    - If Ti writes final value of A in S1, then Ti also writes final value of A in S2
- Basically, allows all **conflict serializable schedules + "blind writes"**

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

view ≡

T1:	R(A)	W(A)		
T2:			W(A)	
T3:				W(A)

Notes on Serializability Definitions

- **View Serializability allows (a few) more schedules than conflict serializability**
  - But V.S. is difficult to enforce efficiently.
- **Neither definition allows all schedules that are actually serializable.**
  - Because they don't understand the meanings of the operations or the data
- **Conflict Serializability is what gets used, because it can be enforced efficiently**
  - To allow more concurrency, some special cases do get handled separately.
  - (Search the web for "Escrow Transactions" for example)

Serializability in Practice

- One solution for “conflict serializable” schedules is Two Phase Locking (2PL)
- Use locks; keep them until commit
  - Strict Two Phase Locking (strict 2PL)

Summary

Concurrency control and recovery are among the most important functions provided by a DBMS.

Concurrency control is automatic

– System automatically inserts lock/unlock requests and schedules actions of different Xacts
– Property ensured: resulting execution is equivalent to executing the Xacts one after the other in some order.

Reading and Next Class

- ACID and Transactions: Ch 16.1 – 16.6
- Next: 2PL/2PLC and Deadlocks: Ch 17