CSE462/562: Database Systems (Spring 24)

Lecture 13: Transaction

4/29/2024

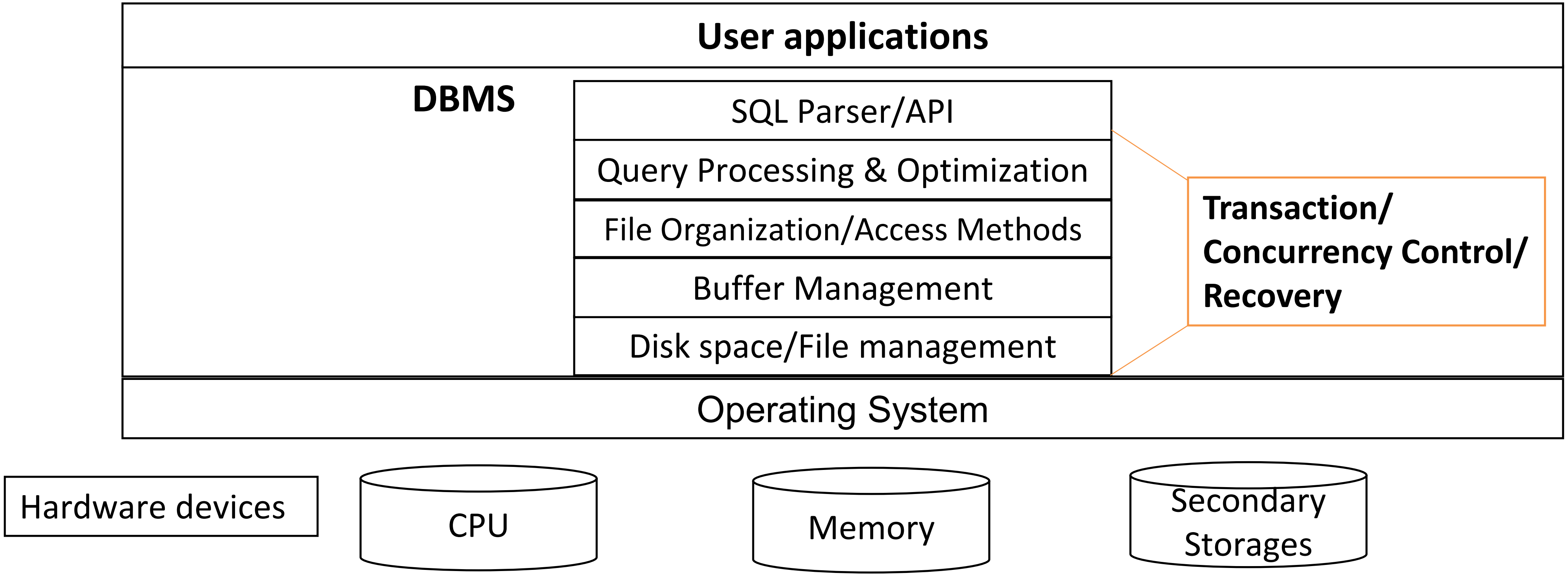
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Last updated: 3/19/2024

Reminders

- HW5 due today, 23:59 PM EDT
- HW6 released today, due on 5/13, 23:59 PM EDT
- Project 5 due on 5/20, 23:59 PM EDT

Big picture


User applications
DBMS
SQL Parser/API
Query Processing & Optimization
File Organization/Access Methods
Buffer Management
Disk space/File management
Transaction/
Concurrency Control/
Recovery
Operating System
Hardware devices
CPU
Memory
Secondary
Storages

What is a transaction?

Transaction:

```
BEGIN;
INSERT INTO A VALUES (…)
SELECT * from A;
DELETE FROM A WHERE …;
COMMIT;
```

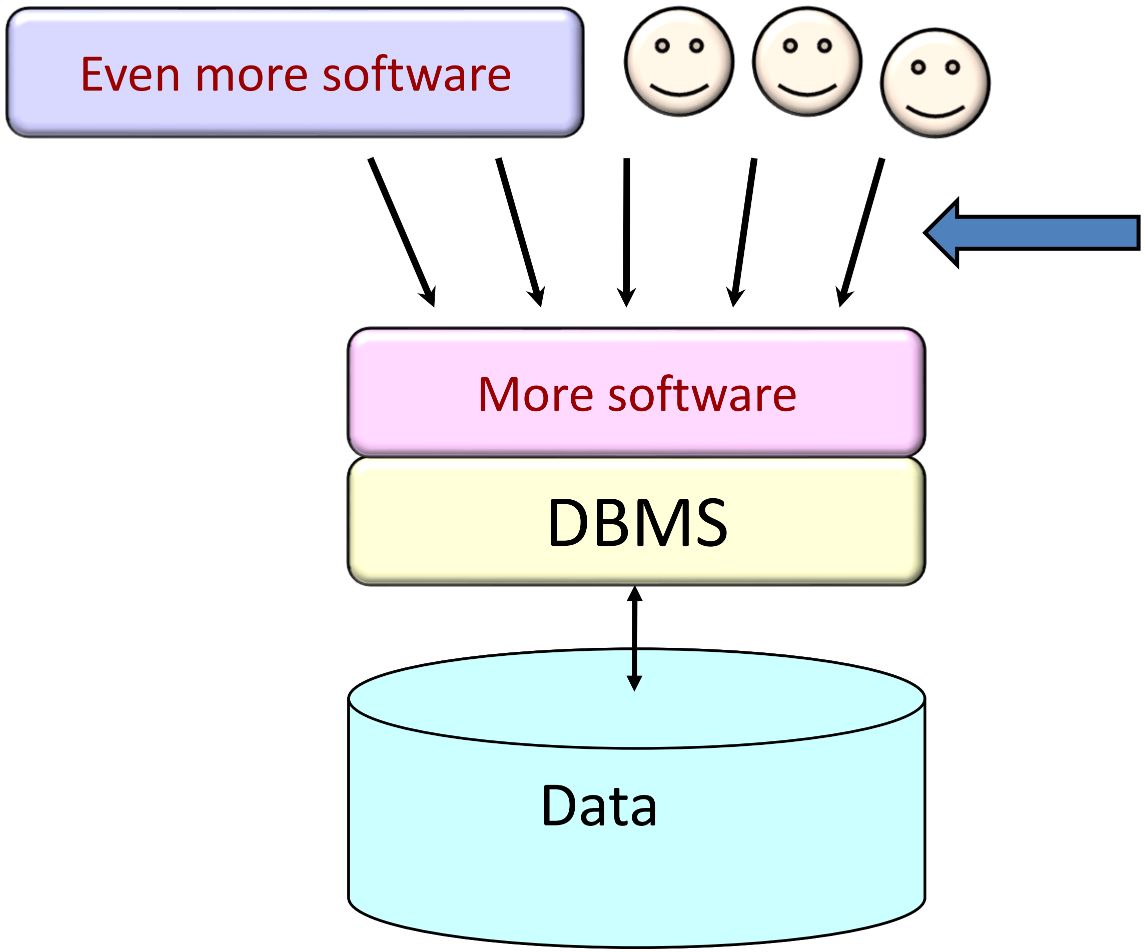
- A transaction is a sequence of one or more SQL operations treated as a unit
  - START/BEGIN [TRANSACTION] to start a new transaction
  - COMMIT: make all the changes by the current transaction permanent and visible
  - ROLLBACK/ABORT: revert all the changes by the current transaction

  - *Autocommit* turns each statement into a transaction
    - often enabled by default

Two independent motivation for transactions

- Concurrent database access
- Resilience to system failures

Motivation 1: concurrent Database Access



Select…
Update…
Create Table…
Drop Index…
Help…
Delete…

Concurrent access: attribute-level inconsistency

```
Update Account Set balance = balance + 1000
Where month(birthday) = 4
```

concurrent with …

```
Update Account Set balance = balance - 500
Where month(birthday) = 4
```

Actions involved: Get, Modify, Put. They may be interleaved!

```
Account(acctno, birthday, balance)
Sales(saleid, sale_date, acctno, amount, status)
```

Concurrent access: tuple-level inconsistency

```
Update Sales Set status = 'processing' Where saleid = 87654321
```

concurrent with …

```
Update Sales Set amt  = amt * 0.8 Where saleid = 87654321
```

Actions involved: Get, Modify, Put. They may be interleaved! Maybe only one of changes survives in the end.

```
Account(acctno, birthday, balance)
Sales(saleid, sale_date, acctno, amount, status)
```

Concurrent access: table-level inconsistency

```
Update Sales S Set status = 'processing'
Where exists (Select * From Account A
              Where S.acctno = A.acctno AND A.balance > S.amount)
```

concurrent with …

```
Update Account Set balance = balance + 1000 where month(birthday) = 4;
```

Actions involved: Get, Modify, Put. They may be interleaved!

```
Account(acctno, birthday, balance)
Sales(saleid, sale_date, acctno, amount, status)
```

Concurrent access: multi-statement inconsistency

```
Insert Into Archive
  Select * From Sales Where status=‘paid’;
Delete From Sales Where decision=‘paid’;
```

concurrent with …

```
Select Count(*) From Sales;
Select Count(*) From Archive;
```

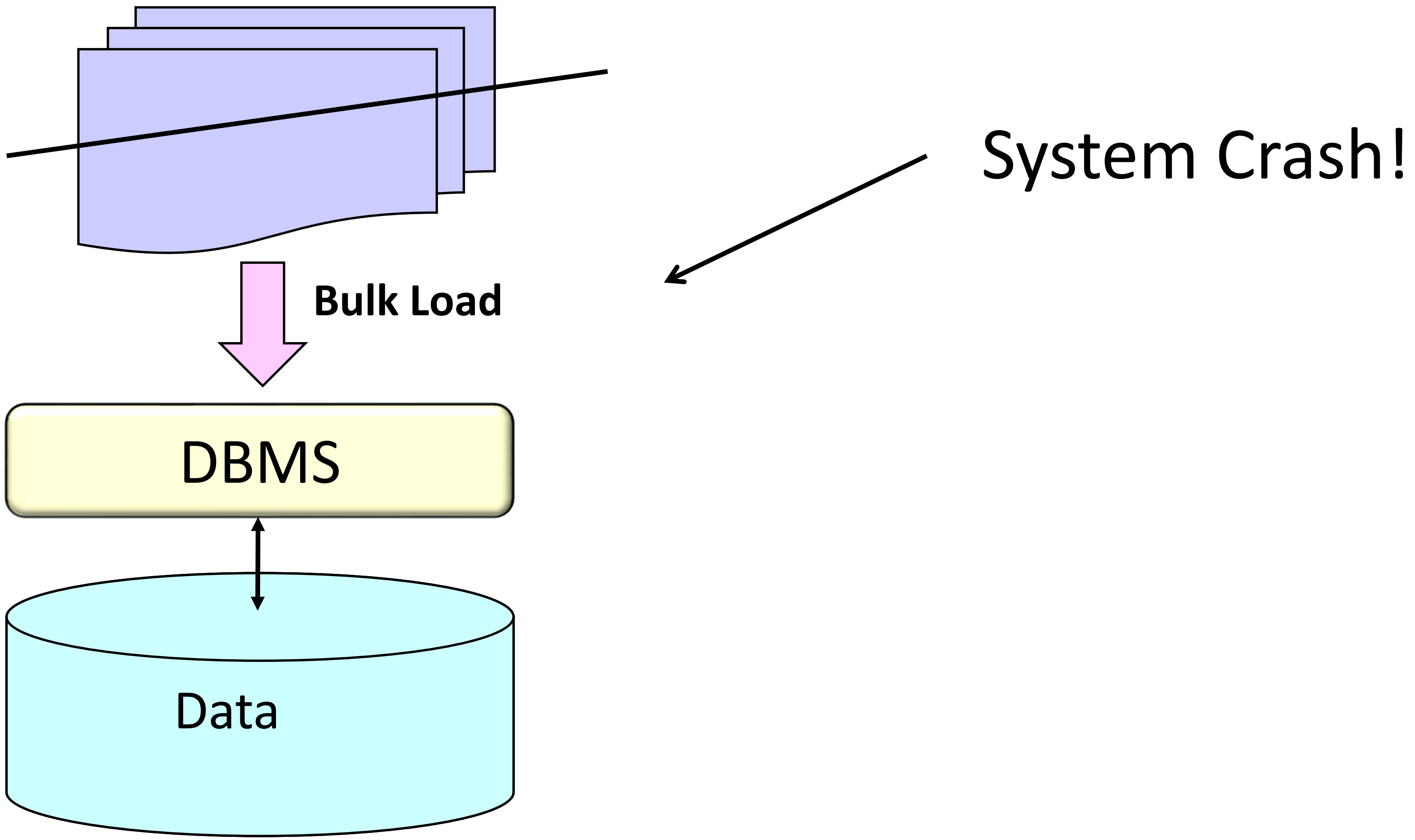
```
Account(acctno, birthday, balance)
Sales(saleid, sale_date, acctno, amount, status)
Archive(saleid, sale_data, acctno, amount, status)
```

Concurrency goal

- Execute sequence of SQL statements so that they appear to run in isolation
  - Simple solution?
    - Run them serially and in isolation.
    - But it’s inefficient when they are accessing different objects.

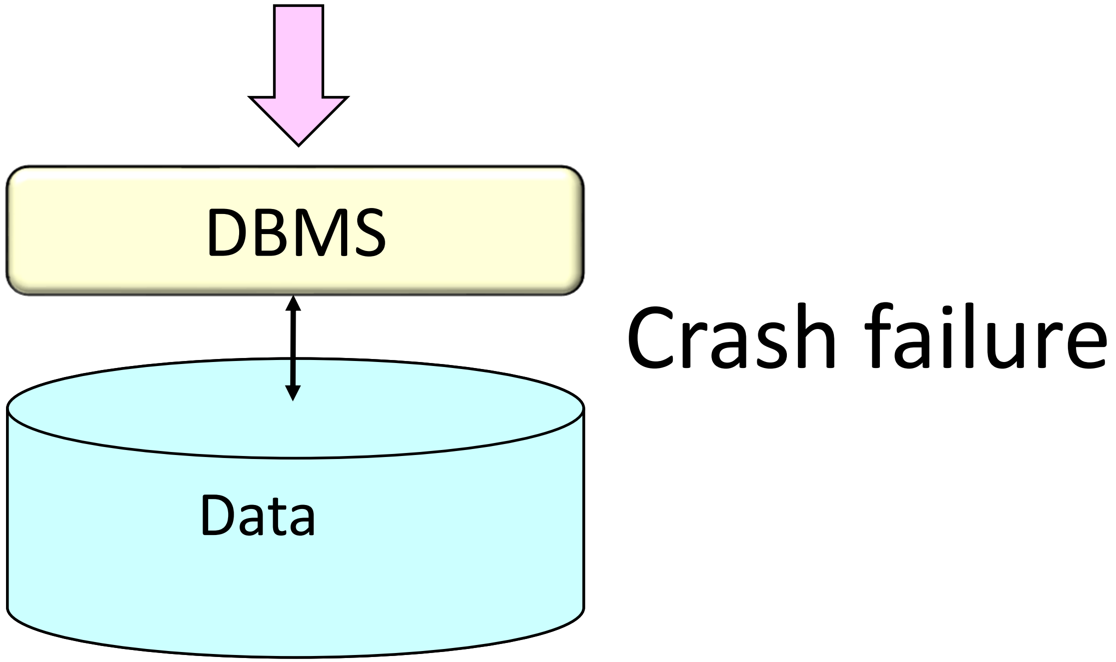
  - Need to enable concurrency whenever it is safe to do so.
    - Interleaving actions from two transactions to improve the overall performance

Motivation 2: resilience to system failures

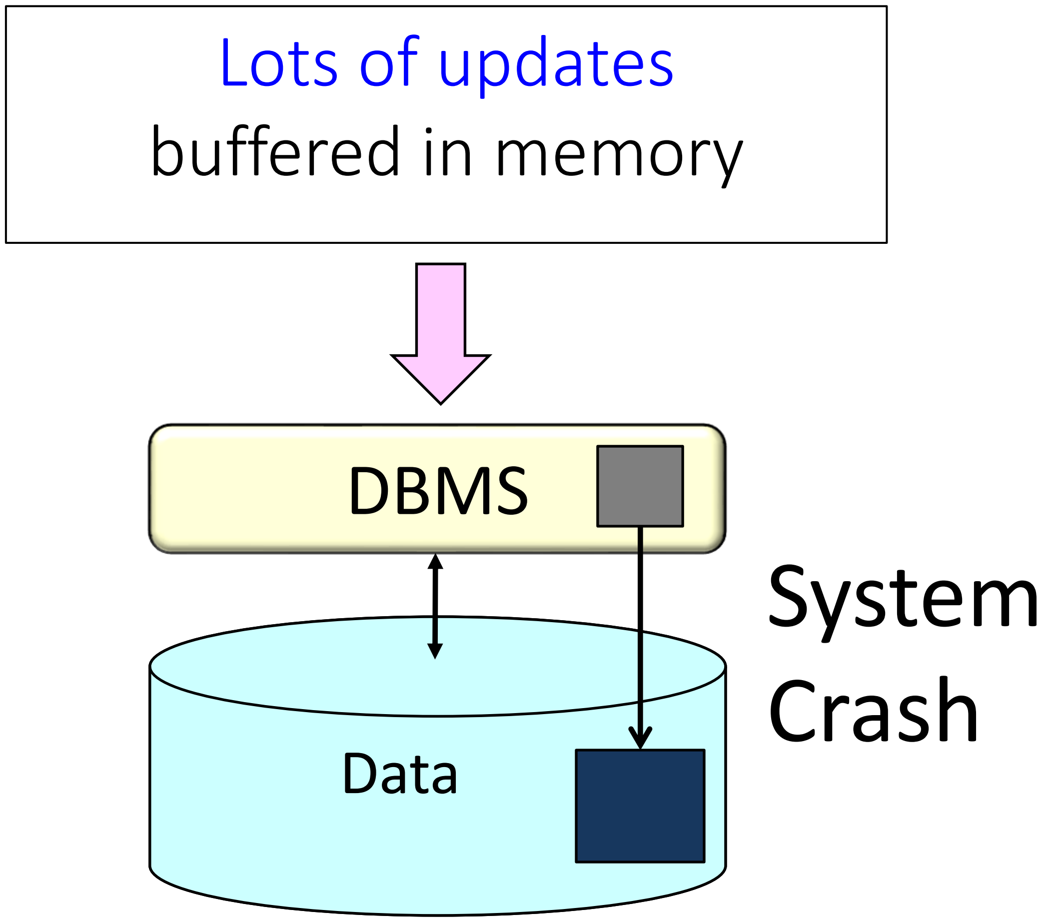


Example: system crash leads to data loss

```
Insert Into Archive
  Select * From Sales Where status = 'paid';
Delete From Sales Where status = 'paid';
```

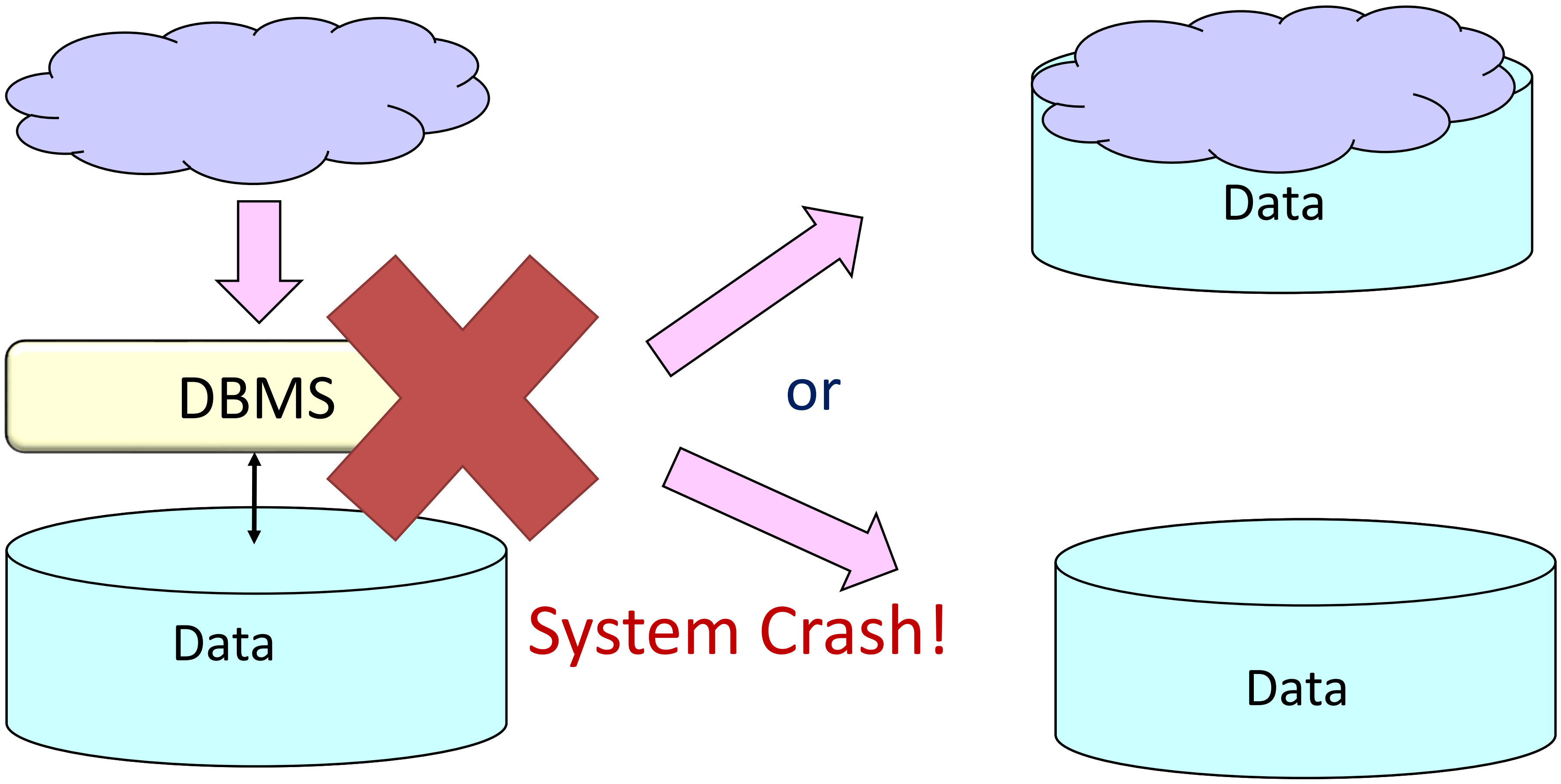


Example: system crash leads to data loss



System-failure goal

- Guarantee all-or-nothing execution, regardless of failures



Why using transaction?

Transaction:

```
BEGIN;
INSERT INTO A VALUES (…)
SELECT * from A;
DELETE FROM A WHERE …;
COMMIT;
```

- Transaction: a solution for both concurrency and failures
  - Transaction appear to run in isolation in the eye of the user
  - If the system fails, each transaction’s changes appear in DB either entirely or not at all.

ACID Properties

- The desirable properties of transaction processing in DBMS.
  - Two important components
    - Concurrency control
    - Logging

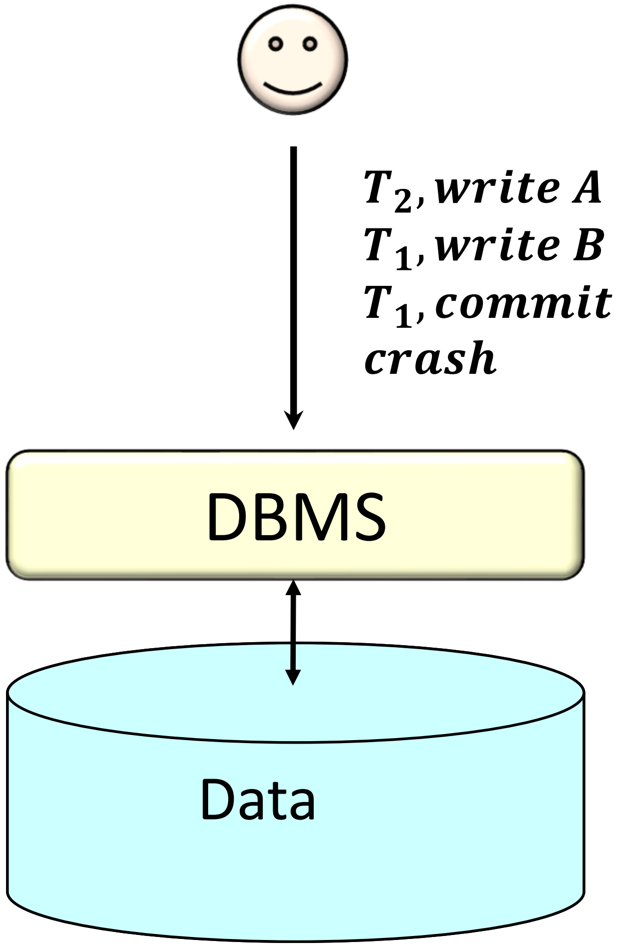
Atomicity

Consistency

Isolation

Durability

Atomicity in ACID properties



Each transaction is "all-or-nothing," never left half done

Achieved by Logging!

System needs to UNDO T2 in this case since it has NOT "Committed" at the time of crash.

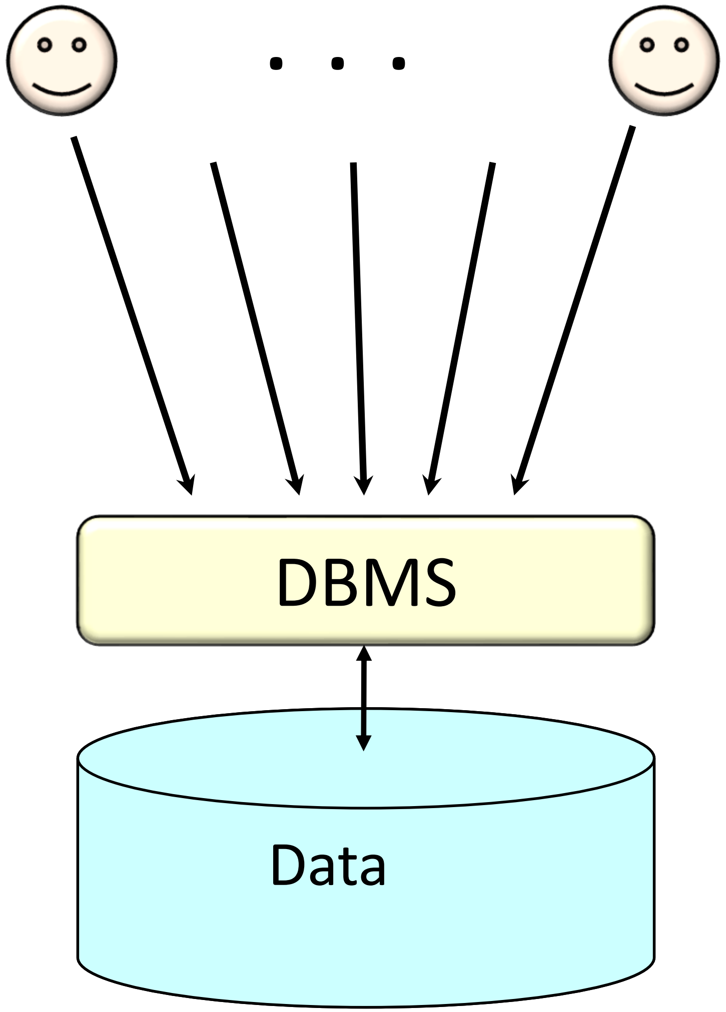
Transaction Abort

- Undoes partial effects of transaction
- Can be system or user initiated
  - System: crash recovery, serialization failure
  - User: calling ROLLBACK or ABORT, SQL errors (e.g., division by zero)

Each transaction is "all-or-nothing," never left half done

```
Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans='ok' Then Commit; Else Rollback;
```

Consistency in ACID properties

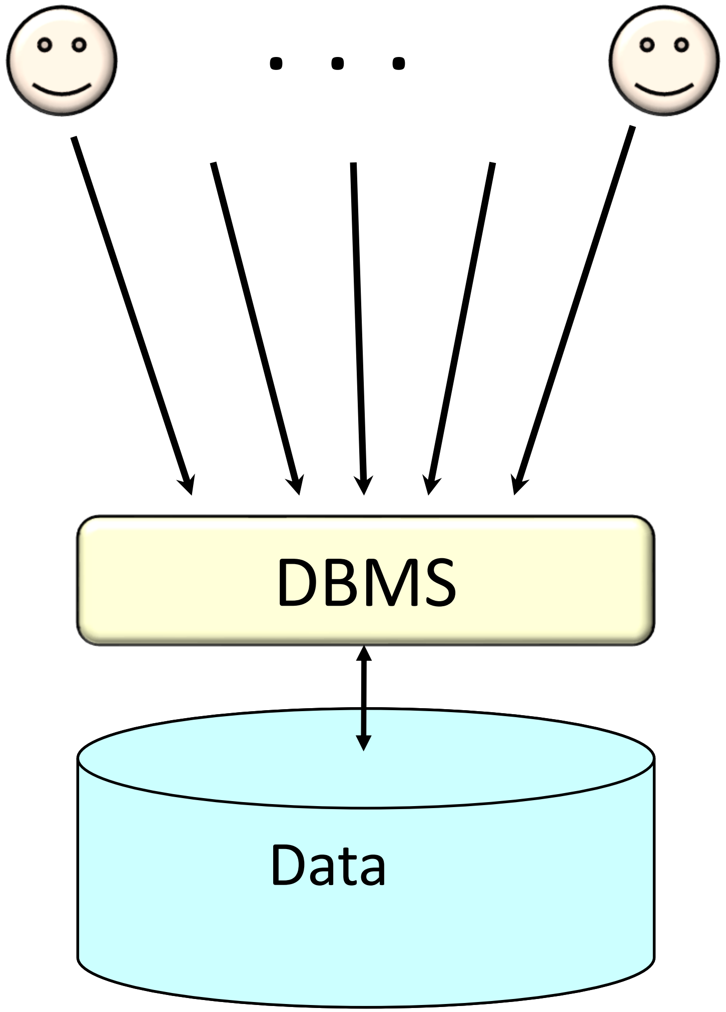


Each client, each transaction:

- Can assume all constraints hold when transaction begins
- Must guarantee all constraints hold when transaction ends

Serializability
+ Integrity constraint check for individual statements/transactions
$\Rightarrow$ constraints always hold

Isolation in ACID properties

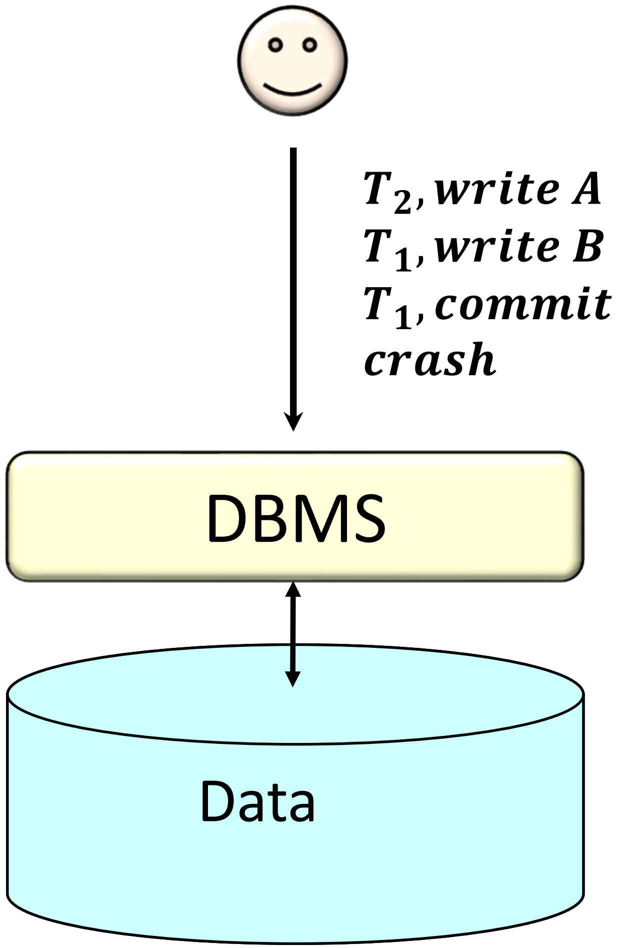


Serializability

Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

Achieved by Concurrency Control!
e.g., Locking.

Durability in ACID properties



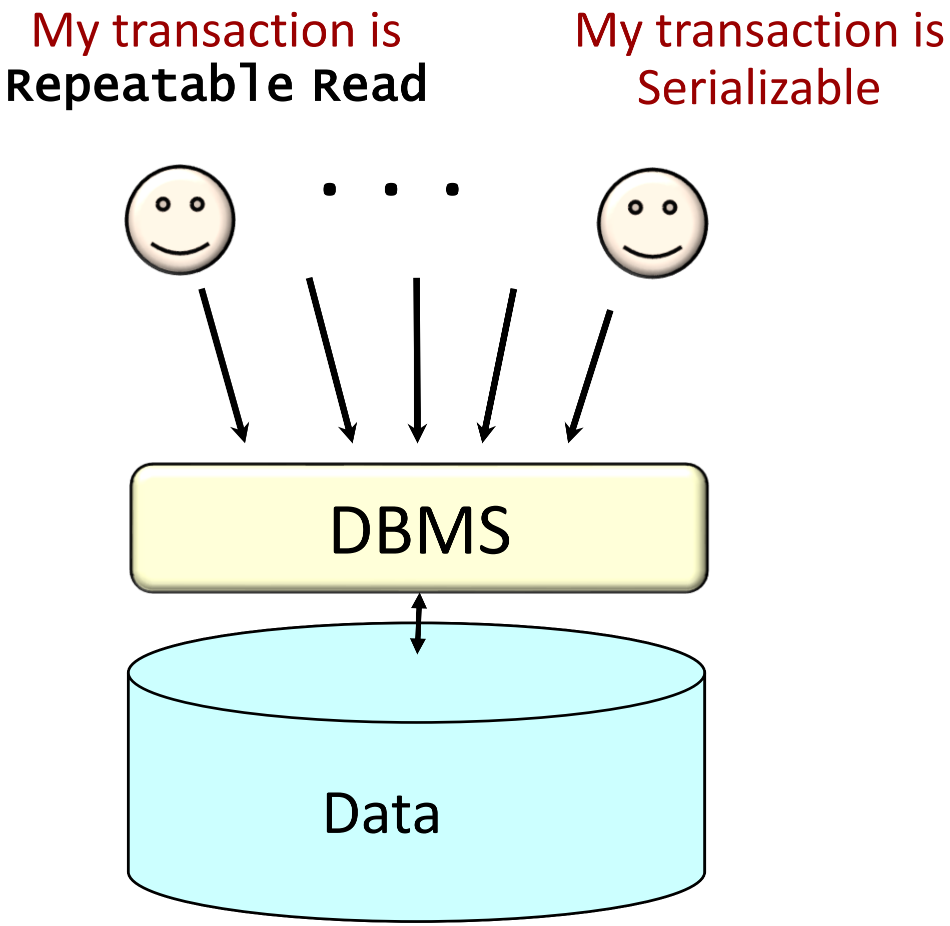
If system crashes
after transaction commits,
all effects of transaction
remain in database

Achieved by Logging!

System may need to REDO T1 in this case since it has "Committed".

Isolation levels

- Isolation Levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
- Per transaction
  - "In the eye of the beholder"
- All except serializable are defined by a few common anomalies
  - *Dirty read*
  - *Non-repeatable read*
  - *Phantom read*



Anomaly 1: Dirty read

- “Dirty” data item: written by an uncommitted transaction

```
Update Account Set balance = balance + 1000
Where month(birthday) = 4
```

concurrent with …

```
Select Avg(balance) From Account
```

- Dirty Reads: if read this value before the 1st Transaction has committed
- What happens if the 1st T rolls back after 2nd T has read this value?
  - non-serializable schedule

Anomaly 2: Non-repeatable read

- Two reads to the same item emit different values in the same transaction.

Transaction 1

Transaction 2

```
Select balance From Account
Where acctno=12345678
```

```
Update Account Set balance= balance - 1000
Where acctno = 12345678;
COMMIT;
```

```
Select balance From Account
Where acctno = 12345678
```

Two reads in Xact 1 returns different values!
Note: it is allowed to return the value previously set in the same transaction.

Anomaly 3: phantom read

- A transaction
  - that might have avoided all dirty reads and non-repeatable reads
  - still does not guarantee serializability: because of the phantom read

Transaction 1

Transaction 2

```
Select balance From Account
Where month(birthday) = 4
```

```
INSERT INTO ACCOUNT VALUES
  (87654321, ‘1992-04-01’, 6000);
COMMIT;
```

```
Select balance From Account
Where month(birthday) = 4
```

Xact 1 queries the accounts whose owner were born in April twice. The second time includes something non-existent in the first time.

ANSI isolation levels

Anomalies / Isolatoin Level	Dirty Read	Non-repeatable read	Phantom Read
Read Uncommited	Possible	Possible	Possible
Read Committed	Impossible	Possible	Possible
Repeatable Read	Impossible	Impossible	Possible
Serializable *	Impossible	Impossible	Impossible

- *DBMS is allowed to provide stronger isolation level even if a weaker one is specified*
  - The table only describes the minimum requirements (i.e., the set of anomalies to prevent)
  - e.g., it is allowed to always provide serializable regardless of which isolation level is set
- *Serializable is defined by equivalence with serial schedule instead of anomalies!*
  - Same as free of the three anomalies with locking (2PL, discussed in next lecture)
  - Does cause issues for snapshot isolation (which admits additional anomalies, e.g., write skew)
    - Further reading: Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10

READ ONLY transactions

- Helps system optimize performance
- Independent of isolation level

```
Set Transaction Read Only;
Set Transaction Isolation Level Repeatable Read;
Select Avg(balance) From Account;
Select Max(balance) From Account;
```

Isolation levels: summary

- Strongest isolation level: serializable
  - Worst performance but easiest to reason about
  - Note: serializable is often not the default isolation level in DBMS
    - for performance consideration
      - cause less performance surprise for novice users
      - looks better on benchmarks (if they are not careful)
      - but the implication is you have to be carefully reason about the program
        - or encounter weird bugs in production.
    - ***Takeaway: Always Read the Documentation of Transaction Behaviors!***
- Weaker isolation levels
  - Increased concurrency + decreased overhead = increased performance
  - Weaker consistency guarantees
  - Some systems have default Repeatable Read or even read committed
- Isolation level per transaction and "eye of the beholder"
  - Each transaction's reads must conform to its isolation level

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions, regardless of whether the DB is single-threaded or multi-threaded.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- Issues: Effect of interleaving transactions, and crashes.

Example, a banking database

- Consider two transactions (*Xacts*):

```
T1:    BEGIN   A=A+100,   B=B-100   END
T2:    BEGIN   A=1.06*A,   B=1.06*B   END
```

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (cont'd)

- Consider the possible interleaving schedules

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

The DBMS's view of the second schedule:

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

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

  (Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

- When we discuss schedules, we only consider reads/writes/commit/abort
  - Ignores computation
- Two forms of (restricted) serializability
  - conflict serializable
  - view serializability

Anomalies with interleaved execution

- Dirty reads (WR conflict)

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

- Unrepeatable reads (RW conflict)

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

Anomalies with interleaved execution

- Phantom read (RW conflict w/ predicate)

T1:	R(t: P(t))		R(t: P(t)) C
T2:		W(A' , s.t. $A' \in \boldsymbol{P}$) C	

- Dirty write (WW conflict)

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

Conflict serializability

- Two operations of two different transactions conflict if
  - Performed on the same object
  - At least one of them is a write

T1:	R_1 (A), W_1(A), R_1(B), W_1(B)
T2:	R_2(A), W_2(A)

Conflicts:
$R_1(A), W_2(A)$
$W_1(A), R_2(A)$
$W_1(A), W_2(A)$

- We can swap two adjacent nonconflicting operations without changing the final state

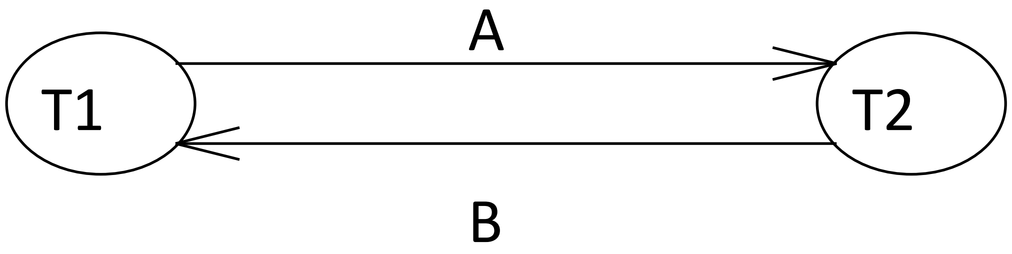
T1:	R_1 (A), W_1(A), R_1(B), W_1(B)
T2:	R_2(A), W_2(A)

- Two schedules are conflict equivalent if one can be transformed into the other through swaps
  - Involve the same actions of the same transactions in the same order
  - Every pair of conflicting operations are ordered the same way
- Schedule S is said to be conflict serializable if it is *conflict equivalent* to some *serial* schedule S'

Determining conflict serializability

- Dependency graph
  - One node per Xact
    - edge from *Ti* to *Tj* if
      - an operation of Ti conflicts with an operation of Tj and
      - Ti's operation appears earlier in the schedule than the conflicting operation of Tj.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

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



Dependency graph

View serializability

- View serializability is based on view equivalence
- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

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

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

View equivalent but not conflict equivalent

- View serializability is "weaker" than conflict serializability!
  - Every conflict serializable schedule is view serializable, but not vice versa!
  - I.e. admits more serializable schedules

Transaction aborts

- So far, we have not considered transaction aborts in conflict serializability
- If a transaction Ti is aborted, all its actions must be undone
  - Not only that, if Tj reads an object last written by Ti, Tj must be aborted as well!
- Many systems avoid such cascading aborts by disallowing reading an object until it is committed
  - If Ti writes an object, Tj can read this only after Ti commits.
  - Avoids non-recoverable schedules
    - where Tj reads an object previously written by Ti and Tj commits before Ti does
    - If there's a crash, the system is in a non-recoverable state
    - *Recoverable does not mean no cascading abort*
- In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded (to be discussed in more details later)
- This mechanism is also used to recover from system crashes
  - all active Xacts at the time of the crash are aborted when the system comes back up.

Summary

- This lecture
  - ACID properties in Database Transactions
  - Isolation levels
  - Serializable transaction scheduling under concurrency

- Next lecture
  - Pessimistic concurrency control
  - Crash recovery

- Reminders
  - HW5 due today, 23:59 PM EDT
  - HW6 released today, due on 5/13, 23:59 PM EDT
  - Project 5 due on 5/20, 23:59 PM EDT