SQL: Transactions

Introduction to Databases

CompSci 316 Spring 2019

DUKE
COMPUTER SCIENCE

Announcements (Tue., Feb. 26)

- Project Milestone #1 due tonight
  - Please submit one report per group
- Homework 2 problems due on Thursday

Motivation: Concurrent Execution

- Concurrent execution of user programs is essential for good DBMS performance.
  - Disk accesses are frequent, and relatively slow
  - it is important to keep the CPU busy by working on several user programs concurrently
  - short transactions may finish early if interleaved with long ones
  - may increase system throughput (avg. #transactions per unit time) and decrease response time (avg. time to complete a transaction)
- A user's program may carry out many operations on the data retrieved from the database
  - but the DBMS is only concerned about what data is read/written from/to the database

Transactions

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

- A transaction is the DBMS's abstract view of a user program
  - a sequence of reads and write
  - the same program executed multiple times would be considered as different transactions
  - 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)

Example

- Consider two transactions:

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

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment
- 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

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

- Consider a possible interleaving (schedule):

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

❖ This is OK. 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)	

Commit and Abort

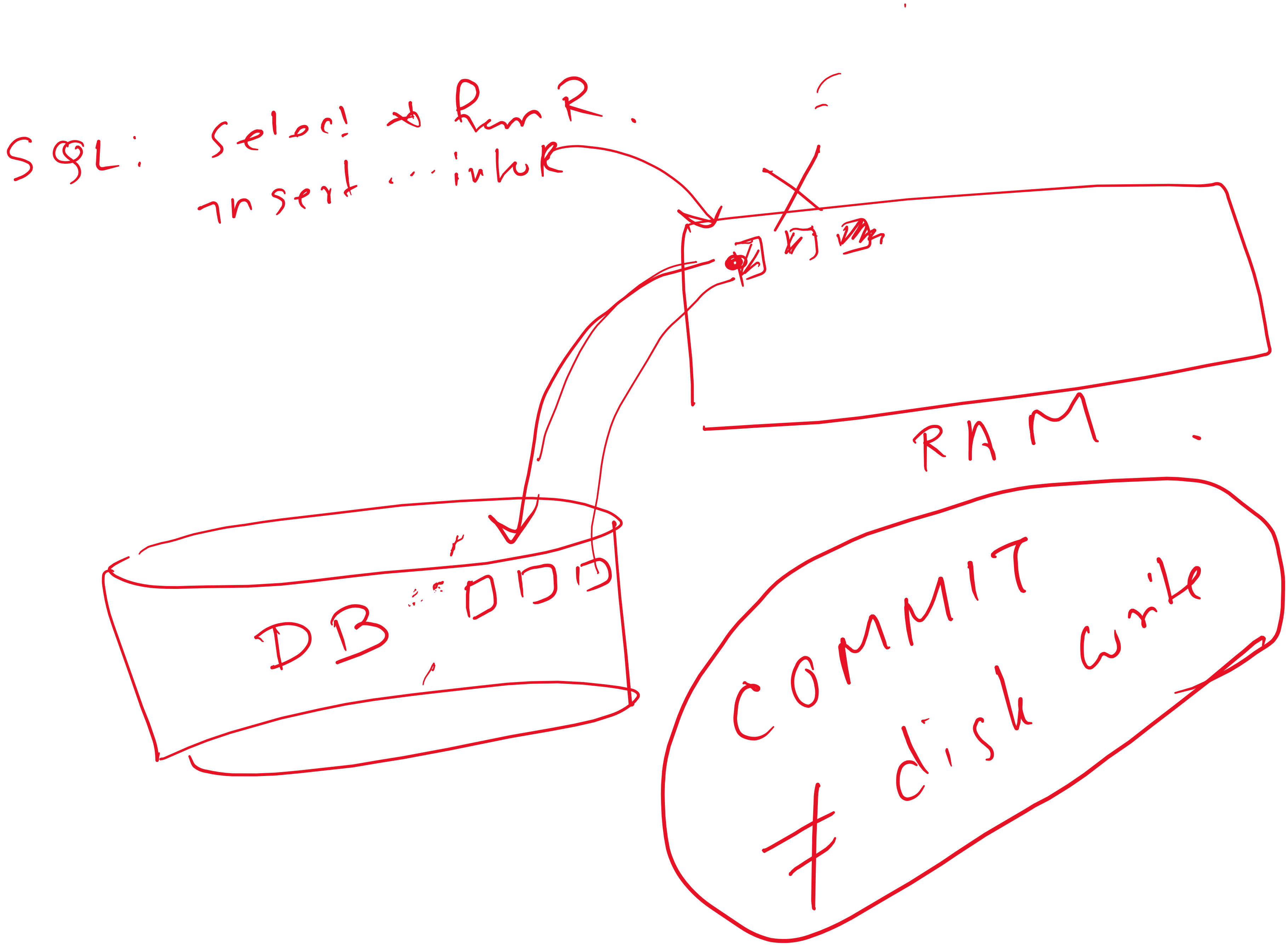
```
T1:     BEGIN   A=A+100,   B=B-100   END
T2:     BEGIN   A=1.06*A,   B=1.06*B   END
```

- A transaction might commit after completing all its actions
- or it could abort (or be aborted by the DBMS) after executing some actions

Concurrency Control and Recovery

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

- Concurrency Control
  - (Multiple) users submit (multiple) transactions
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions
  - user should think of each transaction as executing by itself one-at-a-time
  - The DBMS needs to handle concurrent executions
- Recovery
  - Due to crashes, there can be partial transactions
  - DBMS needs to ensure that they are not visible to other transactions



SQL: select * from R.
insert ... into R
RAM
DB
COMMIT
≠ disk write

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity

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

- A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all
  - Users do not have to worry about the effect of incomplete transactions

Consistency

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

- Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database
  - e.g. if you transfer money from the savings account to the checking account, the total amount still remains the same

Isolation

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

- A user should be able to understand a transaction without considering the effect of any other concurrently running transaction
  - even if the DBMS interleaves their actions
  - transaction are "isolated or protected" from other transactions

Durability

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

- Once the DBMS informs the user that a transaction has been successfully completed, its effect should persist
  - even if the system crashes before all its changes are reflected on disk

Next, how we maintain all these four properties
But, in detail later

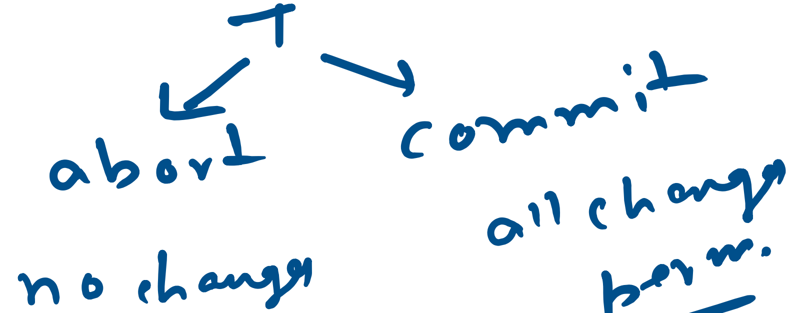
Ensuring Consistency

- e.g. Money debit and credit between accounts
- User's responsibility to maintain the integrity constraints
- DBMS may not be able to catch such errors in user program's logic
  - e.g. if the credit is (debit – 1)
- 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
- Database consistency follows from transaction consistency, isolation, and atomicity

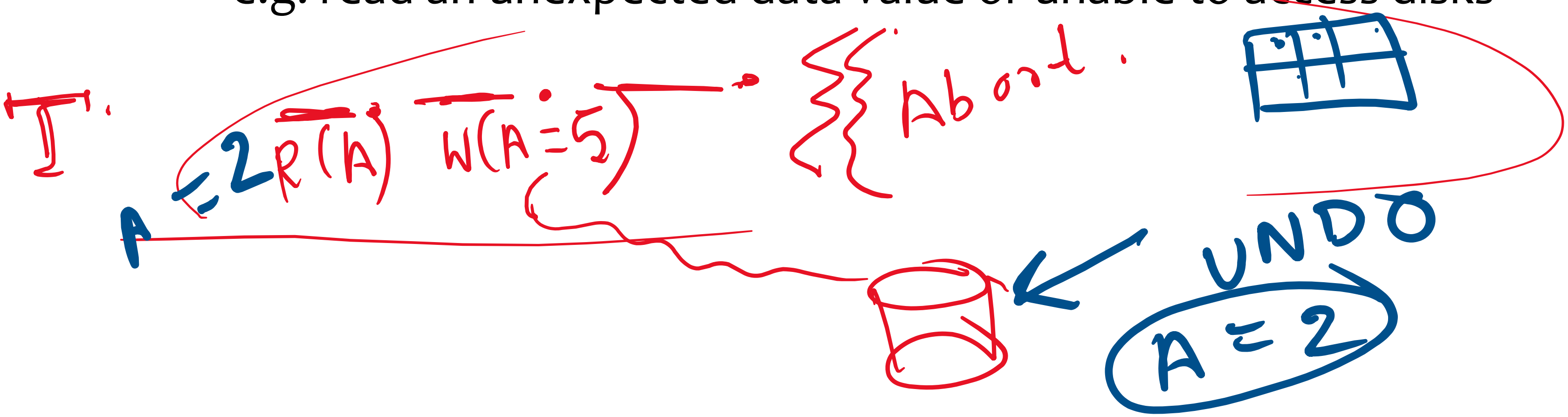
Ensuring Isolation

- DBMS guarantees isolation (later, how)
- 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

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 (say 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

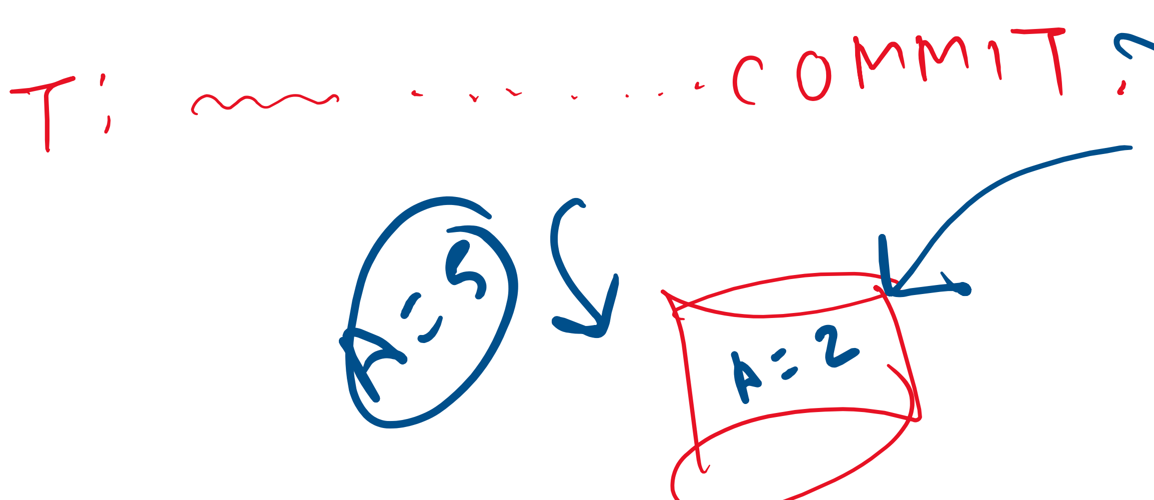


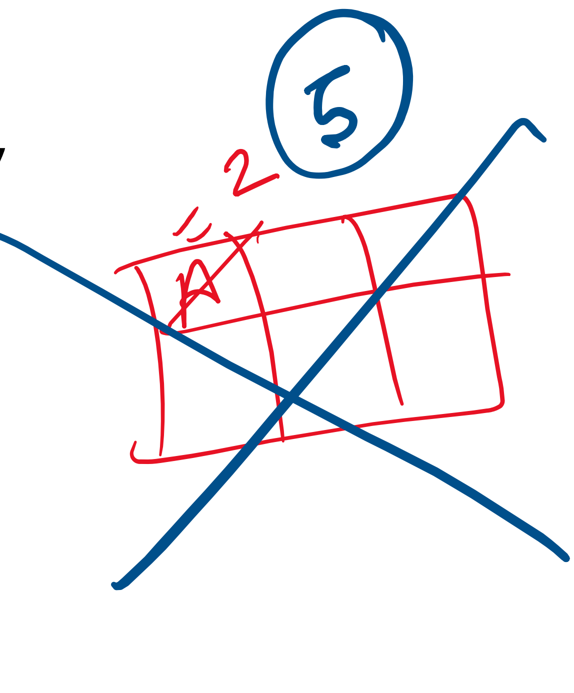
Ensuring Atomicity

- 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” will be discussed later
  - takes care of atomicity and durability





Notations

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

- Transaction is a list of "actions" to the DBMS
  - includes "reads" and "writes"
  - $R_T(O)$: Reading an object O by transaction T
  - $W_T(O)$: Writing an object O by transaction T
  - also should specify $Commit_T$ (C_T) and $Abort_T$ (A_T)
  - T is omitted if the transaction is clear from the context

Assumptions

- Transactions communicate only through READ and WRITE
  - i.e. no exchange of message among them
- A database is a fixed collection of independent objects
  - i.e. objects are not added to or deleted from the database
  - this assumption can be relaxed
    - (dynamic db/phantom problem later)

Schedule

- An actual or potential sequence for executing actions as seen by the DBMS
- A list of actions from a set of transactions
  - includes READ, WRITE, ABORT, COMMIT
- Two actions from the same transaction T MUST appear in the schedule in the same order that they appear in T
  - cannot reorder actions from a given transaction

Serial Schedule

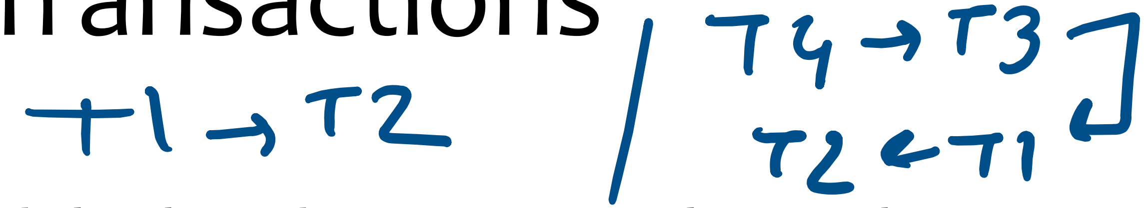
T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

- If the actions of different transactions are not interleaved
  - transactions are executed from start to finish one by one

Problems with a serial schedule

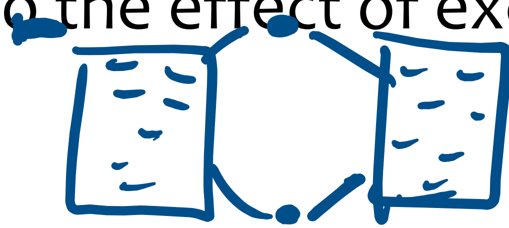
- The same motivation for concurrent executions, e.g.
  - while one transaction is waiting for page I/O from disk, another transaction could use the CPU
  - reduces the time disks and processors are idle
- Increases system throughput
  - average #transactions computed in a given time
- Also improves response time
  - average time taken to complete a transaction
  - since short transactions can be completed with long ones and do not have to wait for them to finish

Scheduling Transactions



- Serial schedule: Schedule that does not interleave the actions of different transactions

- Equivalent schedules: For any database state, the effect (on the set of objects in the database) 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 committed transactions
  - Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Serializable Schedule

- If the effect on any consistent database instance is guaranteed to be identical to that of "some" complete serial schedule for a set of "committed transactions"
- However, no guarantee on T1-> T2 or T2 -> T1

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

serial schedule

T1→T2

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

T2→T1

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

serializable schedules

(Later, how to check for serializability)

Anomalies with Interleaved Execution

- If two consistency-preserving transactions when run interleaved on a consistent database might leave it in inconsistent state

- 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

Schedules with Aborts

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

- Actions of aborted transactions have to be undone completely
  - may be impossible in some situations
    - say T2 reads the fund from an account and adds interest
    - T1 aims to deposit money but aborts
  - if T2 has not committed, we can "cascade aborts" by aborting T2 as well
  - if T2 has committed, we have an "unrecoverable schedule"

Recoverable Schedule

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

- Transaction commits if and only after all transactions they read have committed
  - avoids cascading aborts

ACID: Summary

- A transaction is a sequence of database operations with the following properties (ACID):
  - Atomic: Operations of a transaction are executed all-or-nothing, and are never left "half-done"
  - Consistency: Assume all database constraints are satisfied at the start of a transaction, they should remain satisfied at the end of the transaction
  - Isolation: Transactions must behave as if they were executed in complete isolation from each other
  - Durability: If the DBMS crashes after a transaction commits, all effects of the transaction must remain in the database when DBMS comes back up

SQL transactions

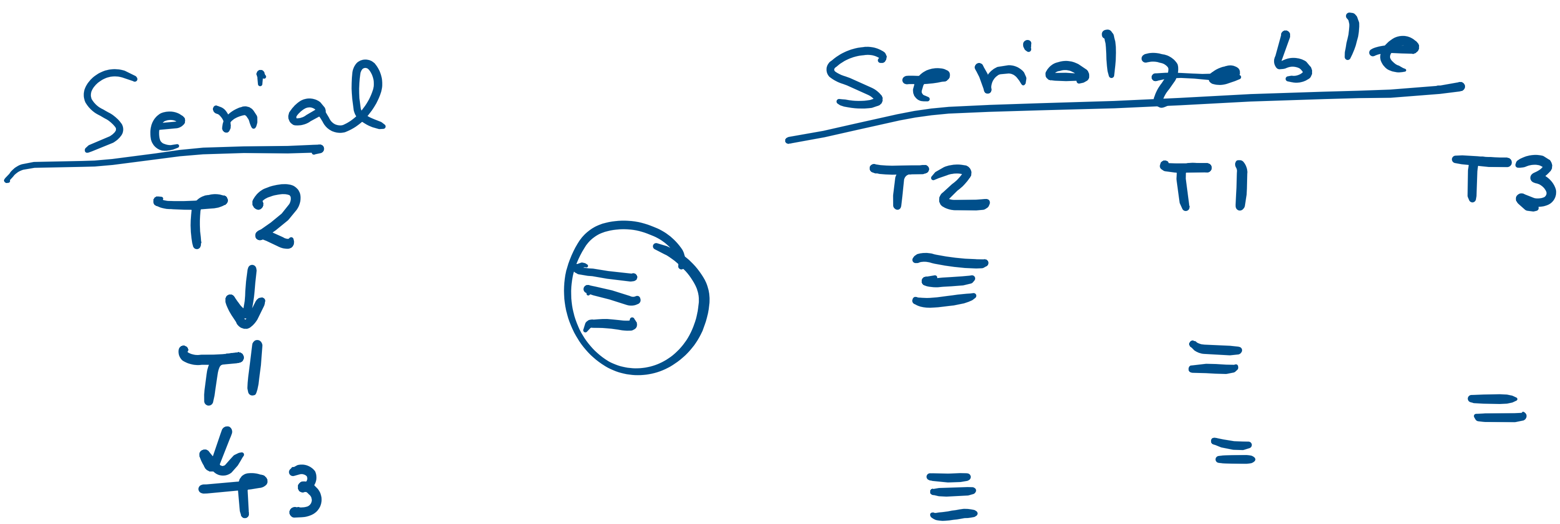
- A transaction is automatically started when a user executes an SQL statement
- Subsequent statements in the same session are executed as part of this transaction
  - Statements see changes made by earlier ones in the same transaction
  - Statements in other concurrently running transactions do not
- COMMIT command commits the transaction
  - Its effects are made final and visible to subsequent transactions
- ROLLBACK command aborts the transaction
  - Its effects are undone

Fine prints

- Schema operations (e.g., CREATE TABLE) implicitly commit the current transaction
  - Because it is often difficult to undo a schema operation
- Many DBMS support an AUTOCOMMIT feature, which automatically commits every single statement
  - You can turn it on/off through the API
  - For PostgreSQL:
    - psql command-line processor turns it on by default
    - You can turn it off at the psql prompt by typing:
      \set AUTOCOMMIT 'off'

SQL isolation levels

- Strongest isolation level: SERIALIZABLE
  - Complete isolation
- Weaker isolation levels: REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED
  - Increase performance by eliminating overhead and allowing higher degrees of concurrency
  - Trade-off: sometimes you get the "wrong" answer



READ UNCOMMITTED

- Can read “dirty” data
  - A data item is dirty if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average
  - ```
    -- T1:                      -- T2:
    UPDATE User
    SET pop = 0.99
    WHERE uid = 142;            SELECT AVG(pop)
                                FROM User;
    ROLLBACK;
                                COMMIT;
    ```
    ```

READ COMMITTED

- No dirty reads, but non-repeatable reads possible
  - Reading the same data item twice can produce different results
- Example: different averages
  - -- T1: -- T2:

```
-- T1:                          -- T2:
                                SELECT AVG(pop)
                                FROM User;
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
                                SELECT AVG(pop)
                                FROM User;
                                COMMIT;
```

REPEATABLE READ

- Reads are repeatable, but may see phantoms
- Example: different average (still!)
  - -- T1: -- T2:

```
-- T1:                              -- T2:
                                    SELECT AVG(pop)
                                    FROM User;
INSERT INTO User
VALUES(789, 'Nelson',
    10, 0.1);
COMMIT;
                                    SELECT AVG(pop)
                              FROM User;
                                    COMMIT;
```

Summary of SQL isolation levels

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

- Syntax: At the beginning of a transaction, SET TRANSACTION ISOLATION LEVEL *isolation_level* [READ ONLY | READ WRITE];
  - READ UNCOMMITTED can only be READ ONLY
- PostgreSQL defaults to READ COMMITTED

Transactions in programming

Using pyscopg2 as an example:

```
conn = psycopg2.connect(dbname='beers')
conn.set_session(isolation_level='SERIALIZABLE',
        ready_only=False,
        autocommit=True)
```

- isolation_level defaults to READ COMMITTED
- read_only defaults to False
- autocommit defaults to False

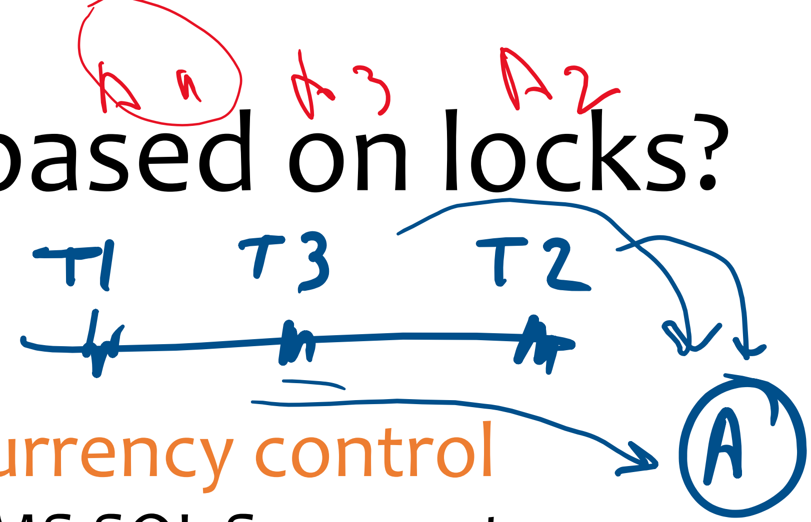
- When autocommit is False, commit/abort current transaction as follows:

```
conn.commit()
conn.rollback()
```

ANSI isolation levels are lock-based

- READ UNCOMMITTED
  - Short-duration locks: lock, access, release immediately
- READ COMMITTED
  - Long-duration write locks: do not release write locks until commit
- REPEATABLE READ
  - Long-duration locks on all data items accessed
- SERIALIZABLE
  - Lock ranges to prevent insertion as well

Isolation levels not based on locks?



Snapshot isolation in Oracle

- Based on multiversion concurrency control
  - Used in Oracle, PostgreSQL, MS SQL Server, etc.
- How it works
  - Transaction X performs its operations on a private snapshot of the database taken at the start of X
  - X can commit only if it does not write any data that has been also written by a transaction committed after the start of X
- Avoids all ANSI anomalies
- But is NOT equivalent to SERIALIZABLE because of write skew anomaly

Write skew example

- Constraint: combined balance $A + B \geq 0$
- $A = 100, B = 100$
- T_1 checks $A + B - 200 \geq 0$, and then proceeds to withdraw 200 from A
- T_2 checks $A + B - 200 \geq 0$, and then proceeds to withdraw 200 from B
- Possible under snapshot isolation because the writes (to A and to B) do not conflict
- But $A + B = -200 < 0$ afterwards!

☞To avoid write skew, when committing, ensure the transaction didn't *read* any object others wrote and committed after this transaction started

Bottom line

- Group reads and dependent writes into a transaction in your applications
  - E.g., enrolling a class, booking a ticket

- Anything less than SERIALABLE is potentially very dangerous
  - Use only when performance is critical
  - READ ONLY makes weaker isolation levels a bit safer