By Marina Barsky

Advanced features of relational databases

Lecture 12

TRANSACTIONS

Integrity or correctness of data

We would like data to be “accurate” or “correct” at all times

EMP table

Name	Age
White	52
Green	3421
Gray	1

Integrity or consistency constraints

- Predicates that data must satisfy

For example:

- x is key of relation R
- Domain(x) = {Red, Blue, Green}
- no employee should make more than twice the average salary

Definition:

- *Consistent state*: satisfies all constraints
- *Consistent DB*: DB in consistent state

Integrity constraints may not capture "full correctness"

Implicit (business) constraints:

- When salary is updated,

  new salary > old salary

- When account record is deleted,

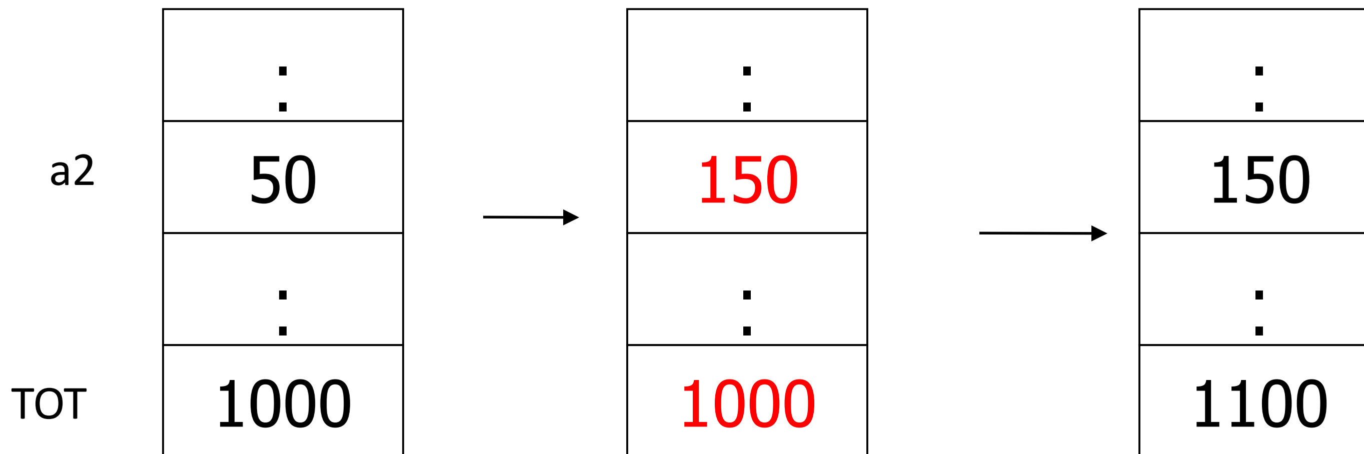
  balance = 0

Observation:
DB cannot be consistent always

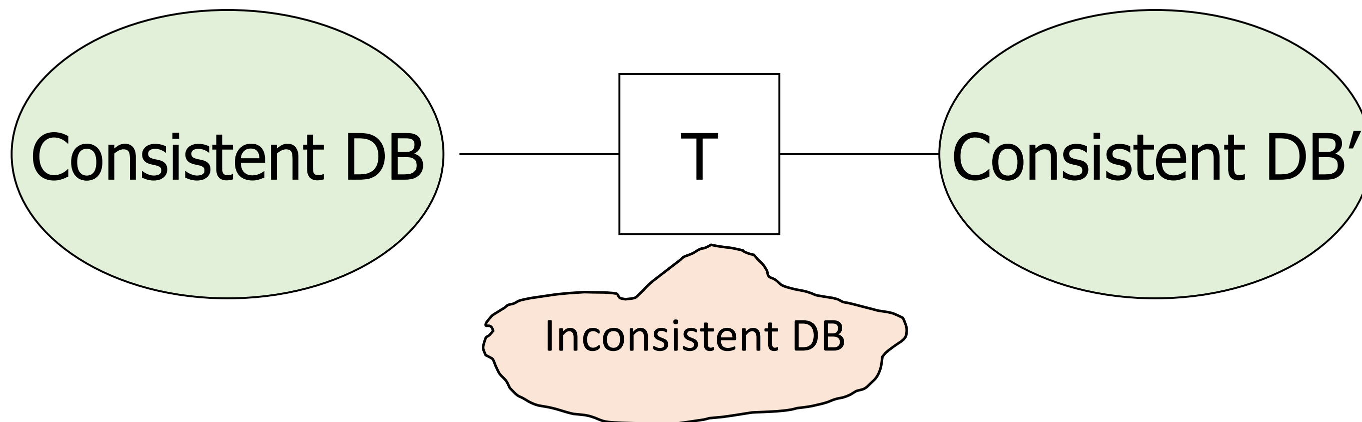
Example: a1 + a2 +…. an = TOT (constraint)

Deposit $100 in a2: $a2 \leftarrow a2 + 100$

$TOT \leftarrow TOT + 100$



Transaction: collection of actions that bring DB from one consistent state to another



If T starts with consistent state + T executes in isolation

$\Rightarrow$T leaves consistent state

Concurrent transactions

- In production environments, it is unlikely that we can limit our system to just one user at a time.
  - Consequently, it is possible for multiple queries to be submitted at approximately the same time.

- If all of the queries were very small (i.e., in terms of time), we could probably just execute them serially, on a first-come-first-served basis.

SERIALLY – ONE AFTER ANOTHER

Queries are executed “simultaneously”

- However, many queries are both complex and time consuming.
  - Executing these queries would make other queries wait a long time for a chance to execute.
  - Disk usage can be optimized for several queries running in parallel
- So, in practice, the DBMS may be running many different queries at about the same time.

INTERLEAVING QUERY PROCESSING

Concurrent Transactions

- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

- Even when there is **no "failure", several** transactions can **interact** to turn a
  consistent state
  into an
  inconsistent state.

Example: two people - one bank account

- Before withdrawing money, each needs to check if the balance is sufficient
- Initially there is **100**$ on the account

Ryan	Monica
READ(X, b)	
b=100	
	READ(X, c)
	c = 100
	c - = 50
	WRITE (X, c)
b - = **100**	
WRITE (X, Ryan)	

Monica: thinks 50 $ left

Ryan: thinks 0 $ left

In fact, the withdrawn amount is 150$

Example: two people - one bank account

- Before withdrawing money, each needs to check if the balance is sufficient
- Initially there is **100**$ on the account

Ryan	Monica
READ(X, b)	
b=100	
	READ(X, c)
	c = 100
	c - = 50
	WRITE (X, c)
b - = **100**	
WRITE (X, b)	

The problem is that the reading and writing operations should be performed as one ***transaction***, their combination should be **atomic**

Transaction

- DBMS groups your SQL statements into ***transactions***.
- The ***transaction*** is the atomic unit of execution of database operations
- By default, each query or DML statement is a transaction
- User can group multiple SQL statements into a single transaction

Transactions with SQL

START TRANSACTION; (BEGIN;)

…SQL statements

COMMIT; (END;)

End of a transaction

- The transaction ends when one of the following occurs:
  - A COMMIT or ROLLBACK are issued
  - A DDL (CREATE, ALTER, DROP …) or DCL (GRANT, REVOKE) statement is issued
  - A user properly exits (COMMIT)
  - System crashes (ROLLBACK)

COMMIT and ROLLBACK

- The SQL statement COMMIT causes a transaction to complete.
  - It's database modifications are now permanent in the database.

- 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 did not request it.

Banking example: DB terminal

Assuming we defined a CHECK constraint on balance >=0

Ryan

```
BEGIN;
 SELECT balance
 FROM accounts
 WHERE
 account_name = "Monica and Ryan";
 UPDATE accounts
 SET balance = balance – 100
 WHERE
 account_name = "Monica and Ryan";
COMMIT;
```

Monica

```
BEGIN;
 SELECT balance
 FROM accounts
 WHERE
 account_name = "Monica and Ryan";
 UPDATE accounts
 SET balance = balance – 50
 WHERE
 account_name = "Monica and Ryan";
COMMIT;
```

Failure – constraint violated

JDBC syntax for transaction

*Basic **JDBC** transaction pattern*

```
Connection conn = ...;

conn.setAutoCommit(false);
try {
  ... //JDBC statements
  conn.commit();
} catch (Exception e) {
  conn.rollback();
}
```

Banking example: JDBC

```
PreparedStatement updateBalance = null;
PreparedStatement selectBalance = null;

String updateSQL =
        "UPDATE " + schema + ".accounts " +
        "set BALANCE = ? WHERE account_name = ?";

String selectSQL =
        "SELECT balance FROM  " + schema + ".accounts " +
        "WHERE account_name = ?";
```

```
try {
    con.setAutoCommit(false);
    int balance;
    updateBalance = con.prepareStatement (updateSQL);
    selectBalance= con.prepareStatement (selectSQL);

    selectBalance.setString (1, accountName);
    ResultSet rs =  selectBalance.executeQuery(query);
    if (rs.next())
        balance = rs.getInt("balance");
    if (balance < withdrawal) { conn.rollback(); return false; }

    updateBalance.setString (2, accountName);
    updateBalance.setInt (1, balance - withdrawal);
    updateBalance.executeUpdate();

    con.commit();
    con.setAutoCommit(true);
    return true;
}
```

No need in DB check constraints here

Proper exception handling with *rollback*

```
} catch (SQLException e ) {
    if (con != null) {
        try {
            con.rollback();
            con.setAutoCommit(true);
            return false;
        } catch(SQLException excep) {
            printException();
            return false;
        }
    }
} finally {
    // close statements updateBalance, selectBalance
}
```

Transaction should have ACID

- ***A**tomicity*: Whole transaction or none is done.
- ***C**onsistency*: Database constraints preserved. Transaction, executed completely, takes database from one *consistent* state to another
- ***I**solation*: It appears to the user as if only one process executes at a time.
    - That is, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after another in some **serial order**.
- ***D**urability*: Effects of a process survive a crash.

Interleaving different transactions

Interleaving of transactions may lead to anomalies even if all the database constraints are preserved

Anomalies of interleaving transactions: example 1

- Consider two transactions T1 and T2, each of which, when running alone preserves database consistency:
  - T1 transfers $100 from A to B (e.g. from checking to saving account)
  - T2 increments both A and B by 1% (e.g. daily interest)

Anomalies of interleaving transactions: example 1

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
r(B)	
w(B)	
commit	

T1 deducted $100 from A

T2 incremented both A and B by 1%

T1 added $100 to B

What is the problem?

Anomaly 1: reading uncommitted data

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
r(B)	
w(B)	
commit	

T1 deducted $100 from A

T2 incremented both A and B by 1%

T1 added $100 to B

The problem is that the bank didn't pay interest on the $100 that was being transferred. This happened because T2 was **reading uncommitted** values.

Anomalies of interleaving transactions: example 2

- Suppose that A is the number of copies available for a book.
- Transactions T1 and T2 both place an order for this book. First they check the availability of the book.
- Consider the following scenario:
  1. T1 checks whether A is greater than 1.
     Suppose T1 sees (reads) value 1.
  2. T2 also reads A and sees 1.
  3. T2 decrements A to 0.
  4. T2 commits.
  5. T1 tries to decrement A, which is now 0, and gets an error because some integrity check doesn't allow it.

Anomaly 2: unrepeatable reads

1. T1 checks whether A is greater than 1.
   Suppose T1 sees (reads) value 1.
2. T2 also reads A and sees 1.
3. T2 decrements A to 0.
4. T2 commits.
5. T1 tries to decrement A, which is now 0, and gets an error because some integrity check doesn't allow it.

The problem is that because value of A has been changed by T1, when T2 reads A for the second time, before updating it, the value is different from that when T2 started.

Anomalies of interleaving transactions: example 3

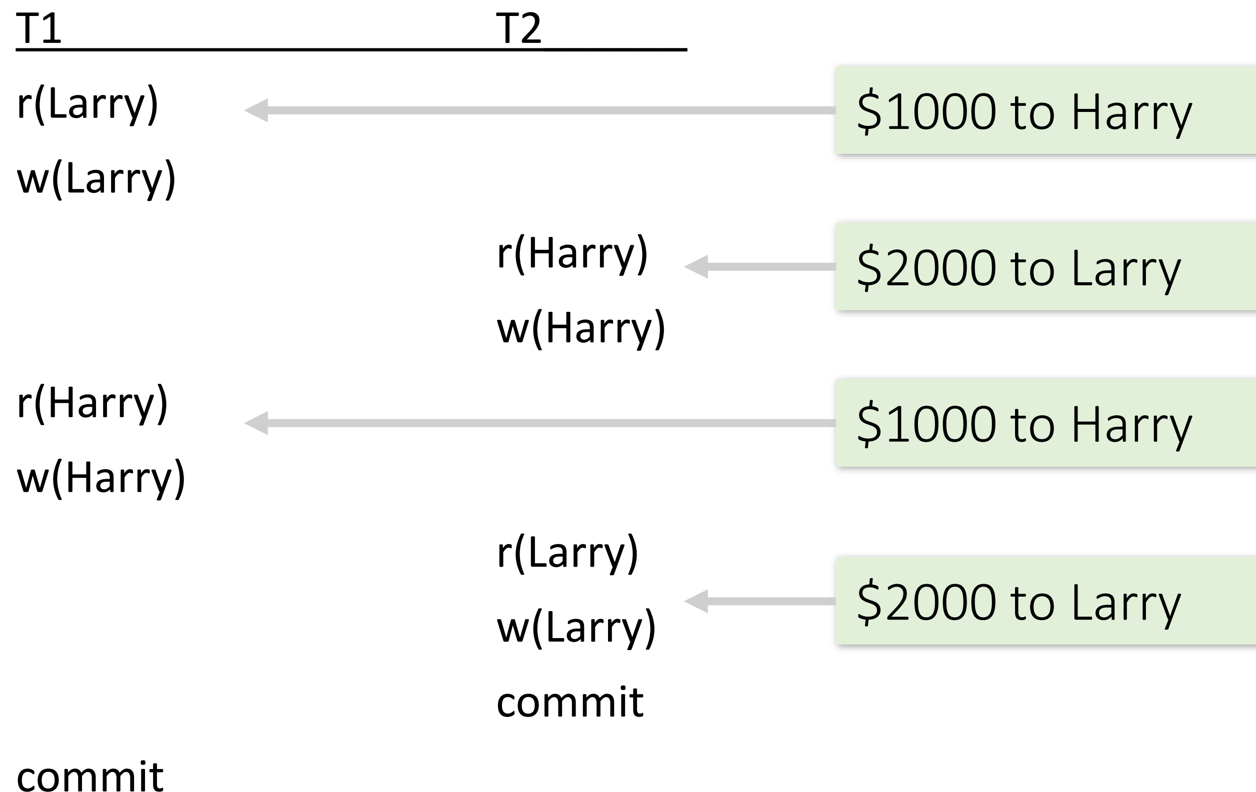
- Suppose that Larry and Harry are two employees, and their salaries must be kept equal. T1 sets their salaries to $1000 and T2 sets their salaries to $2000.
- Now consider the following schedule:

T1	T2
r(Larry)	
w(Larry)	
	r(Harry)
	w(Harry)
r(Harry)	
w(Harry)	
	r(Larry)
	w(Larry)
	commit
commit	

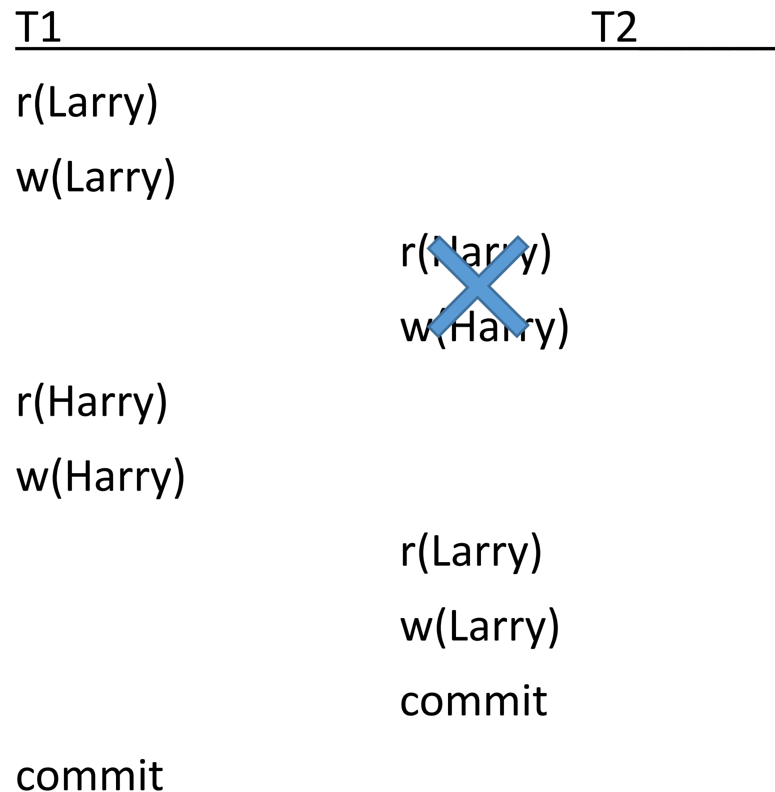
What is the problem?

Anomalies of interleaving transactions: example 3

- Suppose that Larry and Harry are two employees, and their salaries must be kept equal. T1 sets their salaries to $1000 and T2 sets their salaries to $2000.
- Now consider the following schedule:



Anomaly 3:
overwriting uncommitted data



The problem is that T1 has overridden the result of T2, while T2 has not yet been committed.

Anomalies of interleaving

- Reading uncommitted (dirty) data
- Unrepeatable reads
- Overriding uncommitted data

None of these would happen if we were executing transactions one after another: **serial schedules**

A schedule is ***serializable*** if it interleaves transactions but has the same effect as a serial schedule

Enforcing serializability by locks

- If scheduler allows multiple transactions access the same element, this may result in non-serializable schedule
- To prevent this, before reading or writing an element X, a transaction T_i requests a lock on X from the scheduler.
- The scheduler can either grant the lock to T_i or make T_i wait for the lock.
- If granted, T_i should eventually unlock (release) the lock on X.

Possibility of Deadlocks

Example:T1 and T2 each reads X and Y and later writes X and Y.

T1	T2
L1(X)	
	L2(Y)
L1(Y) denied	
	L2(X) denied

This problem is resolved by DBMS by using different types of locks

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Locking granularity

- Should we lock the whole table or just a single row? What's a database object? What should be the locking granularity?

```
SELECT min(year)
FROM Movies;
```

- What happens if we insert a new tuple with the smallest year?

- **Phantom** problem: A transaction retrieves a collection of tuples twice and sees different results, even though it doesn't modify those tuples itself.

Summary of problems

- Dirty read
  - A transaction reads data written by a concurrent uncommitted transaction
- Nonrepeatable read
  - A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).
- Phantom read
  - A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

Transaction tuning in SQL

Gives control over the locking overhead

- Access mode:
  - READ ONLY
  - READ WRITE
- Isolation level (to which extent transaction is exposed to actions of other transactions):
  - SERIALIZABLE (Default)
  - REPEATABLE READ
  - READ COMMITED
  - READ UNCOMMITED

Transaction Isolation Levels

SET TRANSACTION ISOLATION LEVEL **X**

Where X can be

SERIALIZABLE (Default)

REPEATABLE READ

READ COMMITED

READ UNCOMMITED

With a scheduler based on locks:

- A *SERIALIZABLE* transaction obtains locks before reading and writing objects, including locks on sets (e.g. table) of objects that it requires to be unchangeable and holds them until the end.
- A *REPEATABLE READ* transaction sets the same locks as a *SERIALIZABLE* transaction, except that it doesn't lock sets of objects, but only individual objects.

Transaction Isolation Levels

- A *READ COMMITED* transaction T obtains exclusive locks before writing objects and keeps them until the end.

  That is to ensure that the transaction that last modified the values is complete.

  - T reads only the changes made by committed transactions.
  - No value written by T is changed by any other transaction until T is completed.
  - However, a value read by T may well be modified by another transaction (which eventually commits) while T is still in progress.
  - T is also exposed to the *phantom* problem.

- A *READ UNCOMMITED* transaction doesn't obtain any lock at all. So, it can read data that is being modified. Such transactions are allowed to be READ ONLY, or used in cases when reading dirty data does not matter

Examples – renting movies

```
csc343h-mgbarsky=> create table rent (movie varchar(2), rented INT);
CREATE TABLE
csc343h-mgbarsky=> insert into rent values ('A', 0);
INSERT 0 1
csc343h-mgbarsky=> insert into rent values ('B', 0);
INSERT 0 1
csc343h-mgbarsky=> insert into rent values ('C', 0);
INSERT 0 1
```

Isolation level – serializable (default)

csc343h-mgbarsky=> begin;

BEGIN

csc343h-mgbarsky=> **set transaction isolation level serializable**;

SET

csc343h-mgbarsky=> select rented from rent where movie ='A';

rented

0

(1 row)

csc343h-mgbarsky=> update rent set rented=1 where movie ='A';

UPDATE 1

csc343h-mgbarsky=> select rented from rent where m

ovie ='A';

rented

1

(1 row)

csc343h-mgbarsky=> commit;

COMMIT

csc343h-mgbarsky=> begin;

BEGIN

csc343h-mgbarsky=> set transaction isolation level serializable;

SET

csc343h-mgbarsky=> select rented from rent where movie='A';

rented

0

(1 row)

csc343h-mgbarsky=> select rented from rent where movie='A';

rented

0

(1 row)

csc343h-mgbarsky=> update rent set rented=1 where movie ='A';

ERROR: could not serialize access due to concurrent update

Isolation level – repeatable reads

```
update rent set rented=1 where movie='B';

UPDATE 1
```

```
begin;

BEGIN

csc343h-mgbarsky=> set transaction isolation level
repeatable read;

SET

csc343h-mgbarsky=> select * from rent;

 movie | rented

-------+--------

 B    |     0

 C    |     0

 A    |     1

(3 rows)


csc343h-mgbarsky=> select * from rent;

 movie | rented

-------+--------

 B    |     0

 C    |     0

 A    |     1

(3 rows)
```

Reads the same values as before – as expected

Isolation level – repeatable reads

```
insert into rent values ('D', 0);
INSERT 0 1
```

```
begin;
BEGIN
csc343h-mgbarsky=> set transaction isolation level repeatable read;
SET
csc343h-mgbarsky=> select * from rent;
 movie | rented
-------+--------
 C     |      0
 A     |      1
 B     |      1
(3 rows)

csc343h-mgbarsky=> select * from rent;
 movie | rented
-------+--------
 C     |      0
 A     |      1
 B     |      1
(3 rows)
```

No phantom problem – not as expected!

Postgre repeatable reads – no phantoms

From PostgreSQl documentation:

- In PostgreSQL the Repeatable Read isolation level only sees data committed before the transaction began
- It never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.)
- This is a stronger guarantee than is required by the SQL standard for this isolation level, and prevents all of the phenomena including phantoms. This is specifically allowed by the standard, which only describes the minimum protections each isolation level must provide.
- This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction. Thus, successive SELECT commands within a single transaction see the same data, i.e., they do not see changes made by other transactions that committed after their own transaction started.
- Applications using this level must be prepared to retry transactions due to serialization failures.

Isolation level – read committed

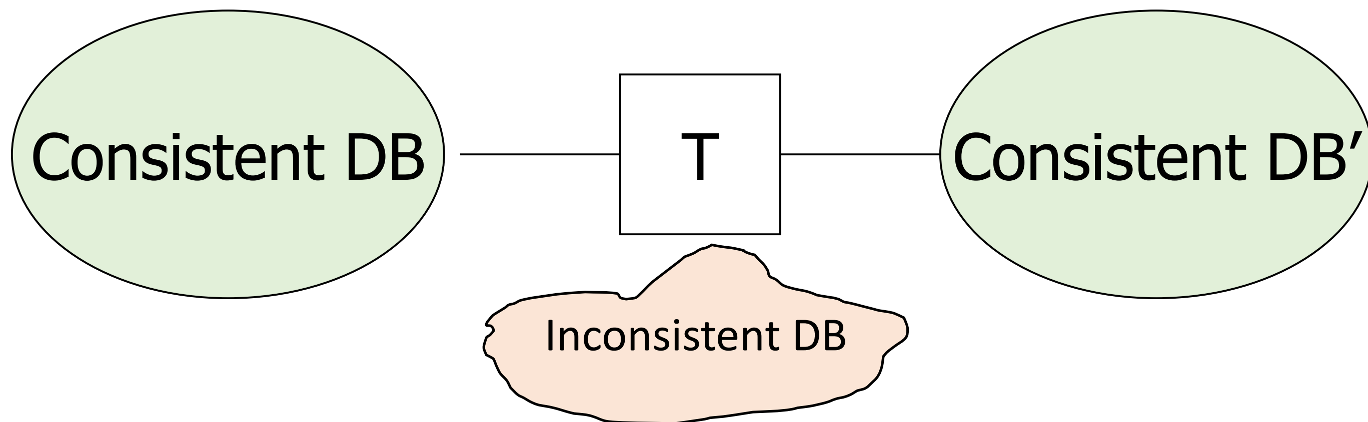
```
begin;
BEGIN
csc343h-mgbarsky=> update rent set rented =1 where movie ='D';
UPDATE 1
csc343h-mgbarsky=> commit;
COMMIT
```

```
set transaction isolation level read committed;
SET
csc343h-mgbarsky=> select * from rent;
 movie | rented
-------+--------
 C     |      0
 A     |      1
 B     |      1
 D     |      0
(4 rows)

csc343h-mgbarsky=> select * from rent;
 movie | rented
-------+--------
 C     |      0
 A     |      1
 B     |      1
 D     |      0
(4 rows)

csc343h-mgbarsky=> select * from rent;
 movie | rented
-------+--------
 C     |      0
 A     |      1
 B     |      1
 D     |      1
(4 rows)
```

How about read uncommitted?

- In PostgreSQL, you can request any of the four standard transaction isolation levels.
- But internally, there are only three distinct isolation levels, which correspond to the levels Read Committed, Repeatable Read, and Serializable.
- When you select the level Read Uncommitted you really get Read Committed, and phantom reads are not possible in the PostgreSQL implementation of Repeatable Read, so the actual isolation level might be stricter than what you select.
- This is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, they do not define which phenomena must happen. The reason that PostgreSQL only provides three isolation levels is that this is the only sensible way to map the standard isolation levels to the multiversion concurrency control architecture.

Transaction: collection of actions that bring DB from one consistent state to another



If T starts with consistent state + T executes in isolation

$\Rightarrow$T leaves consistent state

We learned how to ensure that concurrent (interleaving) actions appear as if each transaction runs in isolation

We still may end up with an inconsistent DB:

- Erroneous data entry
- Transaction bug (application programmer error)
- DBMS bug (DBMS programmer error)
- Other program bug
- System and media failures
  - power loss
  - memory failure
  - processor stop
  - disk crash
  - catastrophic failure: earthquake, flood, end of world

Summary: ACID transactions

- ***C**onsistency*: Database constraints preserved. Transaction, executed completely, takes database from one *consistent* state to another: **serializable schedules**
- ***I**solation*: It appears to the user as if only one process executes at a time: **locking**
- ***A**tomicity*: Whole transaction or none is done: **logging**
- ***D**urability*: Effects of a process survive a crash: **logging**, **recovery**, **RAID**