SQL and transactions

When connecting to the DBMS operations might be autocommit: each SQL statement is a transaction. (default in psql).

Otherwise transactions start with first SQL statement.

- End with { `COMMIT;` / `ROLLBACK;` — aborts trans.

Read Only Transactions.

Read only transactions can never create conflicts to other transactions.

⇒ easier to interleave.

When possible, indicate transaction is read only:

```
SET TRANSACTION READ ONLY;
```

Must be first statement of transaction.

By default transactions are read/write

Isolation Levels

Sometimes we are willing to sacrifice serializability for the sake of performance.

SQL gives 4 options called isolation levels.

Isolation level	Dirty Reads	Non Repeatable Reads	Phantoms
Read Uncommitted	✓	✓	✓
Read Committed	X	✓	✓
Repeatable Read	X	X	✓
Serializable	X	X	X

Type of conflict that the isolation level allows (✓) or not (X)

Use:

```
SET TRANSACTION ISOLATION LEVEL
    <isolation level>;
```

Must be first statement of transaction.

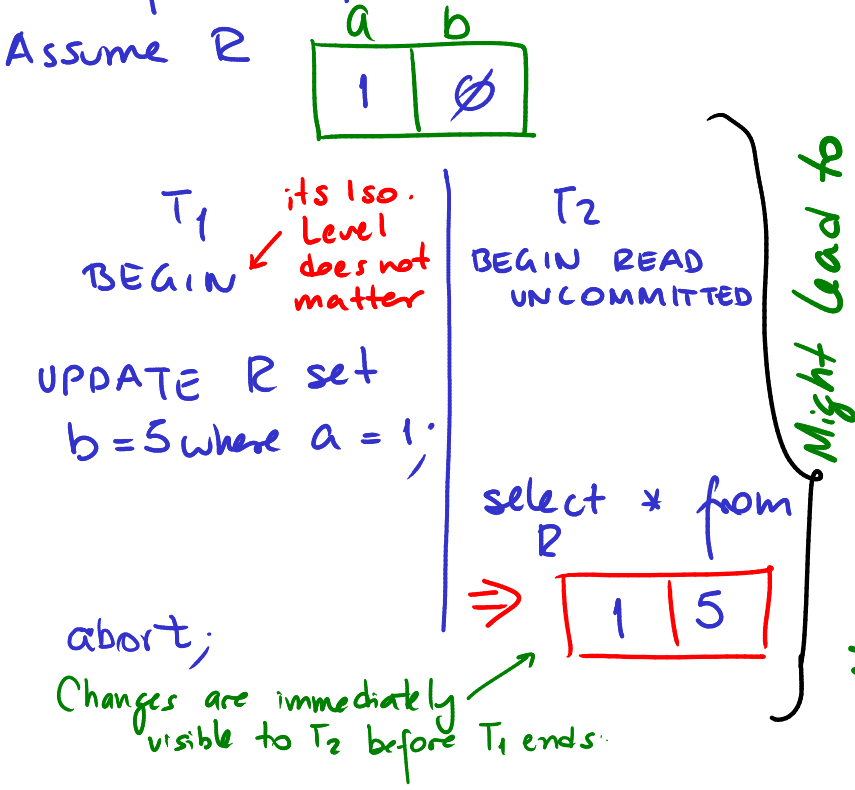
Isolation Levels

We will make the following assumption:

- SQL statements are atomic

READ UNCOMMITTED

- transactions see immediately the effects of other transactions.



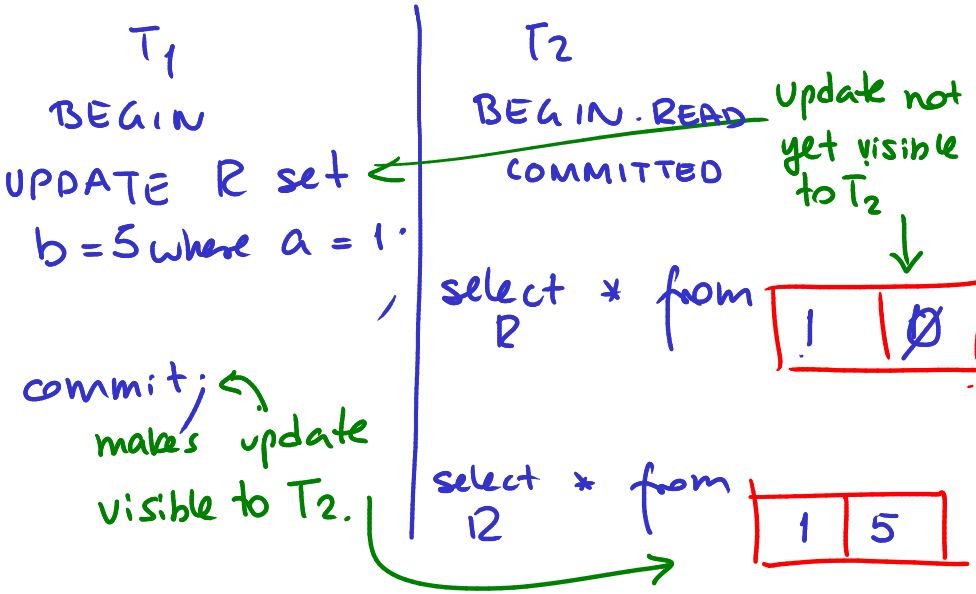
READ COMMITTED

- Transactions see effects of other T_i after T_i has committed.
- As if every T has its own local copy of tuples.
- When a T commits, all local copies updated

Example

R

a	b
1	∅



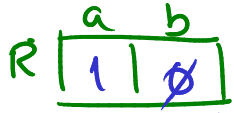
Results in Unrepeatable Read.

Again Might lead to NON SERIALIZABLE SCHEDULE

REPEATABLE READ

- Look as if the T has a local copy of all tuples it accesses.
- Once it reads a tuple it does not see the effects of other transactions on those tuples. (guarantees repeatable read)
- But it might immediately see inserted tuples by other transactions that have been committed (phantoms).

Example:



T_1

BEGIN

update R set b = 5
where a = 1.;

Modifies existing tuple

commit;

T_2.

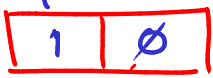
BEGIN REPEATABLE READ;

select ignores update (not yet committed)

select * from R;

Repeatable Read

select * from R



T_2 does not see effects of T_1, even after T_2 commits.

Schedule is SERIALIZABLE

Another example

R

a	b
1	∅

T_1	T_2
BEGIN	BEGIN REPEATABLE READ
UPDATE R SET b = 5 where a = 1;	
	SELECT * from R
	⇒ (1, ∅) It does not see effect of T_1 (no dirty read)
Insert into R values (0, 0); Phantom.	
commit;	
	SELECT * from R
	⇒ (1, ∅), (∅, ∅)

T_2 sees the phantom but not updated tuple.

⇒ Non Serializable schedule.

If T_2 is serializable, T_2 would not see phantom.

WHAT IF T_2 IS READ/WRITE?

- All previous examples T_2 had only Reads.
- If two transactions write to the same object one T waits for the other to commit regardless of isolation level!!
- Can result in deadlock even if READ UNCOMMITTED
- If two Tran. write diff. objects, they proceed as previously described.

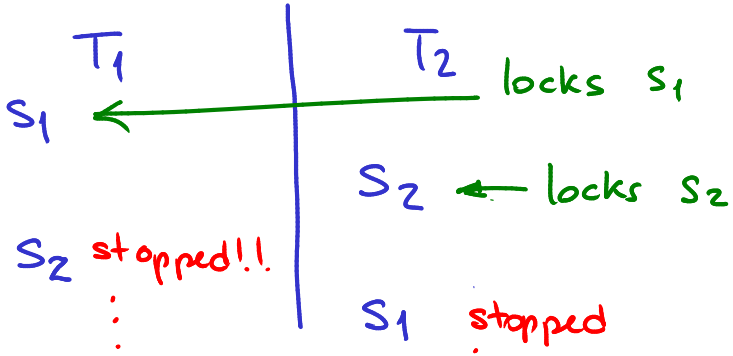
EXAMPLE

Both T are READ UNCOMMITTED

S_1 = Update R set b = 5 where a = 1;

S_2 = Update R set b = ∅ where a = 2

$T_1 = S_1 S_2$ $T_2 = S_2 S_1$



DEADLOCK!! one gets killed, the other continues

- In practice, inserts are allowed to continue
  - No locking of entire table
  - But Transaction might be aborted if T is serializable (and schedule no longer possible to serialize).
- Affects potential interleaving.

EXAMPLE: T_1, T_2 SERIALIZABLE