CMPT 321
FALL 2017

Control of transactions in SQL

Lecture 05.05

By Marina Barsky

Not all transactions are created equal

- Full locking of database objects can prevent other transactions from completing and make users wait for a long time
- Depending on the nature and meaning of a transaction, programmer can choose to sacrifice some consistency in order to increase overall efficiency

- Examples:
  - Getting available seats in the web app for flight selection
  - Getting total number of your followers on twitter
- Here getting an inconsistent snapshot of data is better than locking all the affected objects

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

Levels of increasing isolation

Level	Reading Uncommitted Data (Dirty Read)	Unrepeatable Reads (different values in the same rows)	Phantom (different collections of rows)
READ UNCOMMITED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

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 on all the required objects, including locks on sets (e.g. table) of objects 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 only 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 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 in PostgreSQL – renting movies

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

Isolation level – serializable (default)

```
mbarsky=> BEGIN;
BEGIN
mbarsky=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
```

```
mbarsky=> BEGIN;
BEGIN
mbarsky=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
```

```
mbarsky=> SELECT rented FROM rent WHERE movie='A';
 rented
--------
      0
(1 row)
```

Checked that movie is available

```
mbarsky=> UPDATE rent SET rented = 1 WHERE movie = 'A';
UPDATE 1
```

Rented movie A

```
mbarsky=> SELECT rented FROM rent WHERE movie = 'A';
 rented
--------
      0
(1 row)
```

Checked that movie is available
The change is not committed yet
NO READ UNCOMMITTED

```
mbarsky=>  SELECT rented FROM rent WHERE movie = 'A';
 rented
--------
      1
(1 row)
```

Sees an update

```
mbarsky=> UPDATE rent SET rented = 1 WHERE movie = 'A’;
```

Does not see an update
Tries to update: row is locked
Waits

```
mbarsky=> COMMIT;
COMMIT
```

```
ERROR:  could not serialize access due to concurrent update
```

Cannot finish transaction. Aborts

Transaction isolation levels

Level	Reading Uncommitted Data (Dirty Read)	Unrepeatable Read (different values in the same rows)	Phantom (different collections of rows)
READ UNCOMMITED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Isolation level – repeatable reads

```
mbarsky=> BEGIN;
BEGIN
```

```
mbarsky=> BEGIN;
mbarsky=> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 B     |      0
 C     |      0
 A     |      1
(3 rows)
```

```
mbarsky=> UPDATE rent SET rented = 1 WHERE movie = 'B';
UPDATE 1
```

Rented movie B

```
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 B     |      0
 C     |      0
 A     |      1
(3 rows)
(1 row)
```

Reads the same values as when it started – **repeatable reads guaranteed**

```
mbarsky=> COMMIT;
COMMIT
```

```
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 B     |      0
 C     |      0
```

Reads the same values as when it started – repeatable reads guaranteed – even after T1 commits

Isolation level – repeatable reads

```
mbarsky=> BEGIN;
BEGIN
```

```
mbarsky=> BEGIN;
BEGIN
mbarsky=> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 B     |     0
 C     |     0
 A     |     1
(3 rows)
```

```
mbarsky=> INSERT INTO rent VALUES('D', 0);
INSERT 0 1
```

New row

```
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 B     |     0
 C     |     0
 A     |     1
(3 rows)
```

Does not see the new row: no Phantom problem **which was expected for this isolation level**

```
mbarsky=> COMMIT;
COMMIT
```

```
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 B     |     0
 C     |     0
 A     |     1
(3 rows)
```

Does not see Phantom tuple even after T1 commits

Transaction isolation levels

Level	Reading Uncommitted Data (Dirty Read)	Unrepeatable Read (different values in the same rows)	Phantom (different collections of rows)
READ UNCOMMITED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe (No in PG)
SERIALIZABLE	No	No	No

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.
- This is a stronger guarantee than is required by the SQL standard for this isolation level, and prevents all of the phenomena including phantoms.
- 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

```
mbarsky=> BEGIN;
BEGIN
```

```
mbarsky=> BEGIN;
BEGIN
mbarsky=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
```

```
mbarsky=> UPDATE rent SET rented = 1 WHERE movie = 'D';
UPDATE 1
```

Update by T1

```
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 C     |     0
 A     |     1
 B     |     1
 D     |     0
(4 rows)
```

Does not see the change – T1 did not commit yet

```
mbarsky=> COMMIT;
COMMIT
```

```
mbarsky=> SELECT * FROM rent;
 movie | rented
-------+--------
 C     |     0
 A     |     1
 B     |     1
 D     |     1
(4 rows)
```

Sees the change after T1 commits: so 2 reads are different within the same transaction – **non-repeatable reads**

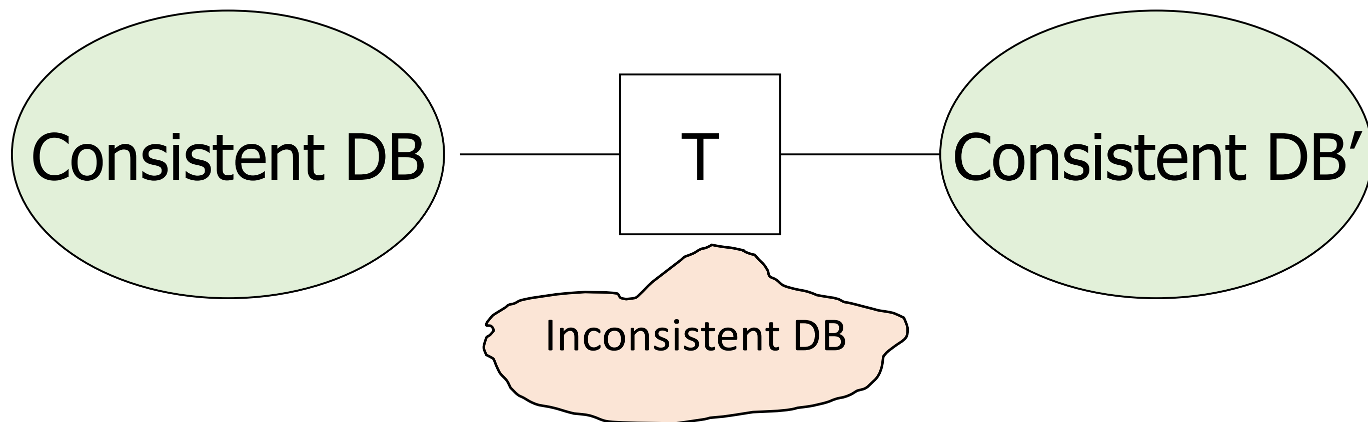
Transaction isolation levels

Level	Reading Uncommitted Data (Dirty Read)	Unrepeatable Read (different values in the same rows)	Phantom (different collections of rows)
READ UNCOMMITED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe (Yes)	Maybe (Yes)
REPEATABLE READ	No	No	Maybe (No in PG)
SERIALIZABLE	No	No	No

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.

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



If T starts with consistent state + T executes in isolation

⇒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**