IN3020/4020 – Database Systems
Spring 2021, Week 9.2

Isolation Levels
Part 2 (Examples, Postgres & MySQL)

Dr. M. Naci Akkøk

CEO, In-Virtualis, Assoc. Prof. UiO/Ifi, Assoc. Prof. OsloMet/CEET

Based upon slides by E. Thorstensen from Spring 2019

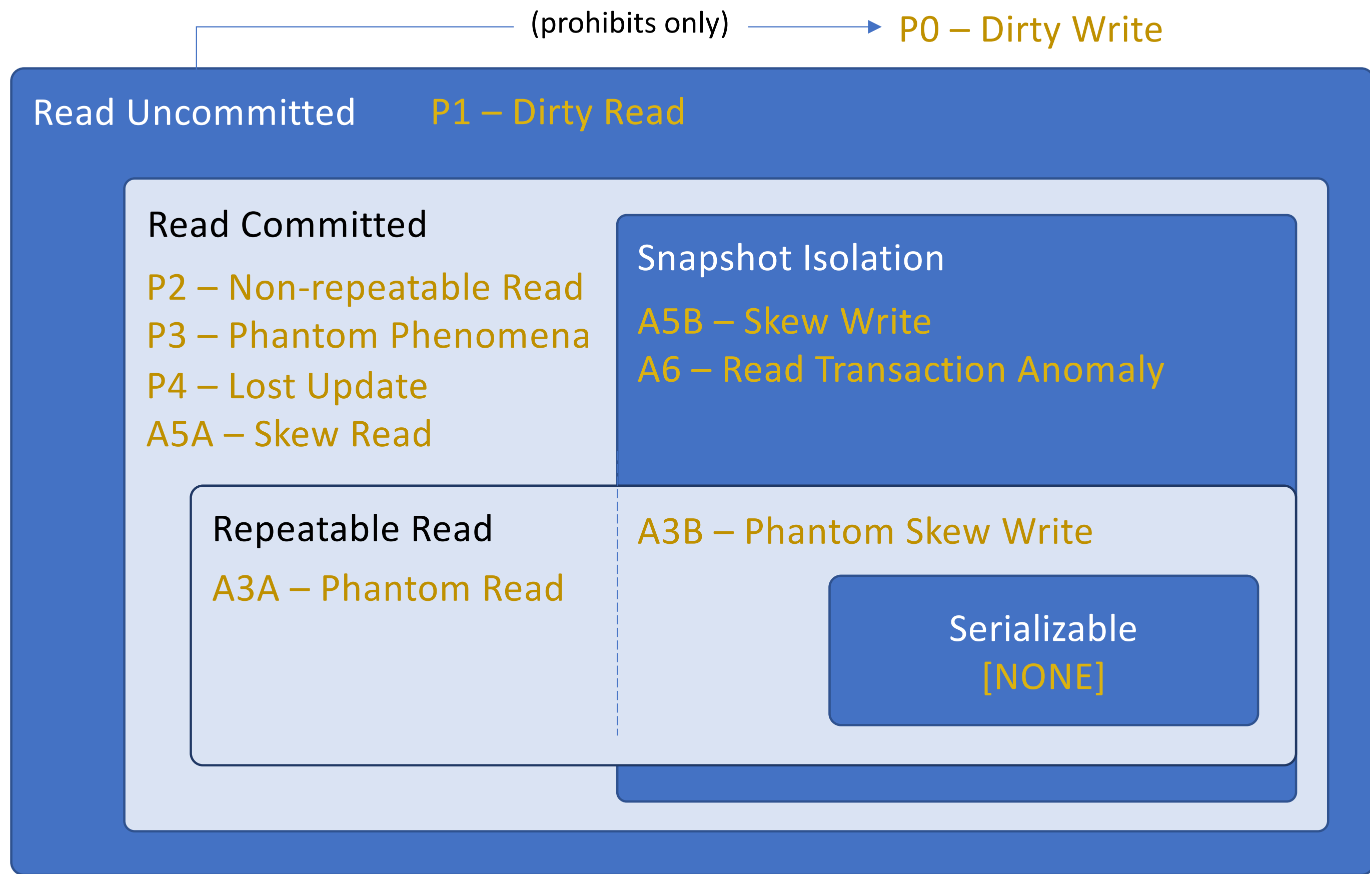
UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Isolation levels

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

- `SELECT ... FOR UPDATE;`

Isolation levels allow for




```
SELECT ... FOR UPDATE;
```

The standard

Focus upon three anomalies:

- Dirty read
- Nonrepeatable read
- Phantom read

- The reality is more complex
- Remember: Not all DBMSs implement everything, and not in the same manner.

Queries in transactions

- `SELECT` (read)
- `UPDATE`
- `INSERT` and `DELETE`

- For each of these, the isolation level specifies what can happen in the transaction

Main idea: Postgres, Oracle and MySQL

- Common: Multiversion control (MVCC) with locking.
- Serializable is handled somewhat different (closer to its intent) in Postgres.
- The default isolation level for InnoDB (MySQL) is REPEATABLE READ. Serializable is treated like repeatable read.
- Oracle DB 20c supports all isolation levels. Note that there are 2 major versions of the DB in the cloud: Oracle Autonomous Transaction Processing (ATP) database and Oracle Autonomous Data Warehouse (ADW)

- https://www.postgresql.org/docs/current/transaction-iso.html
- https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html
- https://docs.oracle.com/en/database/oracle/oracle-database/20/adfns/sql-processing-for-application-developers.html#GUID-E79D04D4-6543-49B3-BAEB-D6585EE67AD7

IN3020/4020 – Database Systems
Spring 2020, Week 11.2

Isolation in Practice – For group sessions
(Postgres, MySQL & Oracle)

Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

MVCC, main features

- Reading does not block writing, and writing does not block reading.
- Only writing will lock, and only other writing may have to wait.
- This is because the system stores multiple versions of tuples.

Background: Data replication in MySQL

- Primary index cluster index, B-tree.
- Leaf nodes are data blocks, everything is in the same file.
- This affects the locking mechanisms.

Read uncommitted (MySQL)

- `SELECT` sees the latest versions of tuples, including non-commits, as expected.

- `UPDATE` and `INSERT` / `DELETE` as in READ COMMITTED.

Read committed (Postgres)

- Default in Postgres.
- Each SELECT sees the latest version of tuples committed before SELECT starts.
- UPDATE / DELETE etc. sees the same version. What about simultaneous updating?

Read committed (Postgres), continued

Simultaneous updating:

- If a transaction (say U) wrote a tuple while T was active, the following happens:
  - T waits for U to finish. In case of abort, T can continue
  - When commit(U), double-checks the tuple and refreshes to new value.
- Waiting via locks to ensure atomic proximity - no missed updates.

Read committed (MySQL)

- Almost the same as in postgres.
- But, in MySQL, index records are locked and not table tuples!
- By default, a cluster index and everything it points to is temporarily locked for UPDATE.
- The locks that are not needed are released after WHERE is evaluated, but there still is some more locking.
- Again, tuples that others have updated have to be waited for.

Read committed example

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session: DELETE FROM website
      WHERE hits = 10;
COMMIT;
```

- `DELETE` finds t with `hits (t) = 10` and waits for lock.
- When `UPDATE` is complete, `DELETE` gets a lock, but then `hits (t) = 11`.

  It does not see the new tuple with `hits = 10`.

Repeatable read, Postgres

- Every `SELECT` in T only sees data committed before T began.
- It won´t see updates and not the inserts (phantoms) that happen along the way.

Repeatable read, Postgres (continued)

- `UPDATE` sees the same as `SELECT` but waiting is FUW (First Update Wins).
- If a transaction (U) wrote a tuple while T was active, the following happens:
- T waits for U to finish; in case of abort, T can continue
- In case of commit(U), T is rolled back - because it doesn´t get to see the new value!

Repeatable read, MySQL

- Pessimistic strategy!
- Each query only sees data committed before running (unlike postgres)
- UPDATE locks all duplicates it matches / waits to lock them (in the read phase).
- `UPDATE` with range condition (`x <100`) locks the «gap» for INSERT.
- But ... no rollback!

Repeatable read, Postgres vs. MySQL

- Postgres: Writing does not block as much – lock on updating. But rollbacks possible.
- MySQL: No rollback, lock when reading for update. And locking of gaps - simultaneous insert not allowed!
- MySQL locks on index, if UPDATE condition is unindexed, locks «index range scanned».
- Thus, simultaneous UPDATE or INSERT / DELETE cannot happen in MySQL - but a concurrent insert can occur between queries in a transaction (oooops!)

https://blog.pythian.com/understanding-mysql-isolation-levels-repeatable-read/

Serializable, MySQL

- Not quite the same implementation of serializable: different interpretations of the standard.
- Every `SELECT` becomes as if they are `UPDATE`, i.e., locks on gaps[1] and rows.
- But where is the equivalent of a serial plan?
  (MySQL does guarantee Serializable – but be careful, it has statement version SESSION level serializability!)
  https://stackoverflow.com/questions/49414519/mysql-interprets-serializable-less-strenuously-than-postgresql-is-it-correct

(1) **INNODB, Gap Locks**. A **gap lock** is a **lock** on a **gap** between index records, or a **lock** on the **gap** before the first or after the last index record.

More on MySQL Serializable

- Not quite the same implementation of serializable: different interpretations of the standard.
- Concurrent `INSERT`s can succeed without being caught.
- Simultaneously updating different tuples fails occasionally ...
- So, MySQL is not really serializable according to the standard.

Read the following interesting thread: https://stackoverflow.com/questions/6269471/does-mysql-innodb-implement-true-serializable-isolation

Serializable, Postgres

- Equivalent to a serial plan.
- Implemented using some kind of precedence graph (somewhat intelligent).
- Can lead to rollback.
- Rollback can occur even if two transactions update each of their own tolls (mandatory statement).

SELECT FOR UPDATE

This is a fine-grained locking mechanism.

```
BEGIN;
SELECT * FROM purchases WHERE processed = false
FOR UPDATE;
- * application is now processing the purchases *
UPDATE purchases SET ...;
COMMIT;
```

The rows selected are locked for other transaction´s updates and for others´ `select for update`.

Isolation levels summarized

- In case of Read Committed, simultaneous updating on the same tuple will continue if the tuple matches, with new value available.
- This goes well if the update is x = f (x), but not if it is based on other info.
- In case of Repeatable Read it will fail / wait for all locks.
- Serializable prevents all interactions (but not in MySQL).

When to use what?

- In Postgres: Read committed vs. repeatable read.
- In MySQL: Repeatable read locks a lot.
- Serializable costs but can guarantee the really big and important transactions.
- For simple select-process-write, `SELECT FOR UPDATE` is good.

IN3020/4020 – Database Systems
Spring 2020, Week 11.2

Other uses of logging & logs

Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Other uses of Undo, Redo, Undo/Redo logs

- As we saw earlier, **DB logs are used to ensure database consistency**, either by "undoing" whatever may cause inconsistency (taking the DB back to the previous consistent state), or by redoing to ensure all intended changes are in place (securing next consistent state), or a mix.

- But logs can be used for more.

Replication and synchronization

Central or "Master" data

"Proxy" data 1

"Proxy" data 2

. . .

"Proxy" data *n*

Replicate (whole/partial)

Change

Synch

Change

Re-replicate (whole/partial)

Replication

- Can be whole or partial:
  - Whole DB
  - Some tables
  - Even parts of tables
- Depends upon the need and kind of usage

- Initial: More like a “move”
- Later (re-replication): Incremental replication

Synchronization

- Pretty much like a "cache" – update propagated when data changed (i.e., when "cache" or replica becomes "dirty") on some location
- Very challenging in terms of concurrency
- Discuss how concurrency and isolation could be implemented in such a case

REMEMBER GAP LOCK? A gap lock (in INNODB) is a lock on the gap between index records. Thanks to this gap lock, when you run the same query twice, you get the same result, regardless other session modifications on that table. This makes reads consistent and therefore makes the replication between servers consistent. If you execute SELECT * FROM id > 1000 FOR UPDATE twice, you expect to get the same value twice. To accomplish that, InnoDB locks all index records found by the WHERE clause with an exclusive lock and the gaps between them with a shared gap lock.

Logs used in replication/synchronization

- Logs on all sides can be used to capture and compare all operations
- And then to synchronize (also re-replicate)

- Most commercial tools also offer ETL (or ELT) capabilities for ingestion, including varying degrees of data transformation capabilities
- They are often used as real-time ETL (ELT) and data-integration tools

Logs used in high-availability architectures

- Logs can also be used to keep an “active” or “passive” copy (often called a “clone”) of the database and the server for ensuring
  - high-availability,
  - disaster recovery,
  - live backups etc.

Oracle GoldenGate