Transactions and Concurrency in PostgreSQL

Real BEGIN/COMMIT/ROLLBACK in psql, MVCC explained mechanically (xmin/xmax), row-level locking with SELECT ... FOR UPDATE, and a deadlock produced and explained on purpose -- without repeating "Transaction Management"'s isolation coverage. The 4th and FINAL lesson in the Advanced PostgreSQL category -- completing 14/14 topics in the PostgreSQL course.

Advanced 30 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/transactions-and-concurrency-in-postgresql

---

"Transaction Management," back in the Spring Data JPA course, already covered `@Transactional`, the three classic isolation problems (dirty/non-repeatable/phantom reads), and PostgreSQL's own isolation defaults in its "Isolation Levels (A Quick Look)" and "Isolation in PostgreSQL" sections — none of that is repeated here. This final lesson goes underneath `@Transactional` entirely: real `BEGIN` / `COMMIT` / `ROLLBACK` in `psql`, how PostgreSQL's MVCC actually makes concurrent reads and writes coexist, real row-level locking with `SELECT ... FOR UPDATE`, and a deadlock produced and explained on purpose.

ACID in psql: BEGIN, COMMIT, ROLLBACK

Every SQL statement so far in this course has run on its own. A transaction groups several statements so they succeed or fail together:

```
BEGIN;

UPDATE topic SET estimated_minutes = 999 WHERE slug = 'joins';

SELECT estimated_minutes FROM topic WHERE slug = 'joins';
-- 999, visible within this same transaction

ROLLBACK;

SELECT estimated_minutes FROM topic WHERE slug = 'joins';
-- back to its real value -- the UPDATE never happened, as far as any other session is
concerned
```

`BEGIN` starts a transaction; every statement after it is provisional until `COMMIT` makes it permanent or `ROLLBACK` discards it entirely, as if it never ran. This is atomicity — the "A" in

ACID — made directly visible: the `UPDATE` above was real and readable *within* the transaction, then completely undone by `ROLLBACK`, with no partial trace left behind.

Autocommit: What Happens Without an Explicit Transaction

Every single-statement example earlier in this course — every `SELECT`, every `INSERT` — ran without a visible `BEGIN`/`COMMIT` at all, because PostgreSQL wraps any statement not inside an explicit transaction in an implicit one of its own, committing it immediately if it succeeds. This is why "Inserting, Updating, and Deleting Data"'s real migration `INSERT`s never needed an explicit `BEGIN` — each one already ran as its own single-statement transaction. `BEGIN`/`COMMIT`/`ROLLBACK` only become necessary the moment more than one statement genuinely needs to succeed or fail as a unit — precisely what `@Transactional`, already covered in "Transaction Management," achieves from the Java side without any application code ever writing `BEGIN`/`COMMIT` itself.

MVCC: How PostgreSQL Makes Concurrent Reads Possible Without Blocking

"Isolation in PostgreSQL" already named MVCC as the mechanism behind `REPEATABLE_READ`/`SERIALIZABLE` without explaining how it actually works — here's the mechanism itself. **MVCC** (Multi-Version Concurrency Control) means PostgreSQL never overwrites a row in place when it's updated — instead, it writes a *new version* of the row and marks the old version as superseded, keeping both around until nothing could possibly still need the old one. Every row secretly carries two hidden system columns, `xmin` (the id of the transaction that created this row version) and `xmax` (the id of the transaction that deleted or superseded it, if any) — a `SELECT` never sees these by default, but they're what PostgreSQL actually consults to decide which row version a given transaction is allowed to see.

This is precisely why readers never block writers, and writers never block readers, in PostgreSQL — a transaction reading `topic` sees whichever row versions existed as of its own snapshot, regardless of another transaction concurrently writing new versions; there's no queue, no waiting, just two transactions looking at different (but both entirely valid) versions of the same logical row. An old row version only gets physically cleaned up once no running transaction could still need it — the job PostgreSQL's background `VACUUM` process handles, a detail worth knowing exists without needing to operate it directly for this project's own scale.

Row-Level Locking: SELECT ... FOR UPDATE

MVCC handles concurrent reads gracefully, but two transactions genuinely trying to *modify* the same row still need to coordinate — that's what row-level locking is for. `SELECT ... FOR UPDATE` locks the rows it returns, blocking any other transaction from modifying (or also locking) those same rows until the first transaction commits or rolls back:

```
-- Session A
BEGIN;
SELECT * FROM topic WHERE slug = 'joins' FOR UPDATE;
-- topic row for 'joins' is now locked by Session A

-- Session B, run concurrently, while Session A's transaction is still open
UPDATE topic SET estimated_minutes = 20 WHERE slug = 'joins';
-- Session B blocks here -- waits until Session A commits or rolls back
```

Session B's `UPDATE` doesn't fail — it simply waits, genuinely blocked, until Session A's transaction ends one way or the other. This is the SQL-level mechanism underneath what "The Persistence Context and Locking," back in the Spring Data JPA course, already covered as `@Lock(LockModeType.PESSIMISTIC_WRITE)` — Hibernate issues exactly this `FOR UPDATE` clause underneath that annotation, nothing more exotic.

A Real FOR UPDATE Scenario

`SELECT ... FOR UPDATE` earns its place specifically when a read needs to be followed by a write that depends on what was just read, and no other transaction should be able to change that value in between — a classic real case is decrementing a limited quantity:

```
BEGIN;
SELECT sort_order FROM topic WHERE slug = 'joins' FOR UPDATE;
-- read sort_order, do some application-level calculation with it
UPDATE topic SET sort_order = sort_order + 1 WHERE slug = 'joins';
COMMIT;
```

Without `FOR UPDATE`, two concurrent transactions could both read the same `sort_order`, both compute the same "next" value independently, and both write it — a genuine lost update, since the second write silently overwrites the first without either transaction knowing the other happened. `FOR UPDATE` prevents exactly this by making the second transaction's own `SELECT ... FOR UPDATE` wait for the first to finish, rather than letting both proceed on stale information.

Producing and Explaining a Deadlock

A **deadlock** happens when two transactions each hold a lock the other one needs, and each is waiting for the other to release it — neither can ever proceed:

```
-- Session A
BEGIN;
UPDATE topic SET estimated_minutes = 1 WHERE slug = 'joins';
-- Session A now holds a lock on the 'joins' row

-- Session B, concurrently
BEGIN;
UPDATE topic SET estimated_minutes = 1 WHERE slug = 'aggregation-and-group-by';
-- Session B now holds a lock on the 'aggregation-and-group-by' row

-- Session A, next
UPDATE topic SET estimated_minutes = 2 WHERE slug = 'aggregation-and-group-by';
-- Session A blocks, waiting for Session B's lock

-- Session B, next
UPDATE topic SET estimated_minutes = 2 WHERE slug = 'joins';
-- Session B would also block, waiting for Session A's lock --
-- except PostgreSQL detects the cycle first
```

PostgreSQL actively detects this cycle rather than letting both sessions wait forever — one of the two transactions (chosen as the "victim," typically whichever would be cheaper to roll back) gets a real error (`deadlock detected`) and is automatically rolled back, freeing its locks so the other transaction can proceed. The fix isn't a database setting — it's a coding discipline: always acquire locks on multiple rows in the same, consistent order across every transaction that touches them (here, always `joins` before `aggregation-and-group-by` , never the reverse), which makes the cyclic wait this example produced structurally impossible.

Common Misconceptions

"A transaction locks the whole table." Not by default — `SELECT ... FOR UPDATE` locks only the specific rows it returns; MVCC means ordinary reads never lock anything at all, for anyone. **"MVCC means PostgreSQL doesn't need locks."** It reduces how often locks are needed (readers and writers never block each other), but genuine write-write conflicts on the same row still need row-level locking, exactly as this lesson's `FOR UPDATE` examples show. **"A deadlock means the database is broken."** It's the opposite — deadlock detection is PostgreSQL correctly noticing an unsolvable cycle and resolving it automatically, rather than the alternative (both transactions frozen forever), which would be the genuinely broken outcome.

Best Practices

- Reach for `SELECT ... FOR UPDATE` specifically when a read's result will directly inform a write, and no other transaction should be able to change that value in between — the "read, then increment" pattern in this lesson is the canonical case, and it's the same mechanism `@Lock(PESSIMISTIC_WRITE)` , already covered in "The Persistence Context and Locking," relies on underneath.

- Lock multiple rows in a consistent order across every transaction that might touch them together — this lesson's deadlock example exists specifically because two sessions locked `joins` and `aggregation-and-group-by` in opposite order; always picking the same order eliminates the cycle entirely.
- Keep transactions short, especially ones holding row locks — every statement between `BEGIN` and `COMMIT` / `ROLLBACK` is time another transaction might spend waiting on a lock this one holds.
- Recognize that this project's own repositories never use `SELECT ... FOR UPDATE` or explicit row locking anywhere — a genuinely honest reflection of its read-mostly nature (mirroring "Transaction Management"'s own "why is there still no `@Transactional`" observation about parts of this codebase), not an oversight.

Common Mistakes

- Assuming a `ROLLBACK` needs to be requested by name before anything is undone — an uncommitted transaction that a client simply disconnects from also gets rolled back automatically, but relying on that rather than an explicit `ROLLBACK` makes intent far harder to read later.
- Holding a `SELECT ... FOR UPDATE` lock open across a slow operation unrelated to the database (an external API call, a long computation) — every other transaction needing that row waits for the entire operation, not just the database work.
- Assuming a deadlock is rare enough in practice to ignore — any code path where two operations can genuinely lock the same rows in different orders will eventually deadlock under real concurrent load, however rarely; application code that touches multiple rows needs a retry strategy for exactly this error, not an assumption it can't happen.
- Confusing a `SELECT ... FOR UPDATE` block with a `Seq Scan` blocking a query — "Indexes and Query Performance with EXPLAIN" covered scan types as a cost/performance concept; locking, covered here, is a correctness/concurrency concept entirely independent of it.

Summary, Cheat Sheet, and Glossary

Summary

- `BEGIN` / `COMMIT` / `ROLLBACK` make atomicity directly visible in `psql`; every statement outside an explicit transaction already runs inside PostgreSQL's own implicit, auto-committed one.
- MVCC means an update writes a new row version rather than overwriting in place, tracked via hidden `xmin` / `xmax` columns — this is why ordinary reads never block writes, or vice versa, in PostgreSQL, the mechanism "Isolation in PostgreSQL" named without explaining.

- `SELECT ... FOR UPDATE` locks specific rows for the duration of a transaction, blocking other transactions from modifying them until it ends — the exact SQL underneath `@Lock(PESSIMISTIC_WRITE)`, already covered in "The Persistence Context and Locking."
- A deadlock happens when two transactions each hold a lock the other needs; PostgreSQL detects the cycle and rolls back one transaction automatically — avoided entirely by locking multiple rows in a consistent order.
- This project's own repositories never use explicit row locking, a genuine reflection of its read-mostly workload rather than a gap — the same honest pattern "Transaction Management" already applied to `@Transactional` itself.

Cheat Sheet

```
BEGIN;
...statements...
COMMIT;    -- or ROLLBACK;

SELECT ... FROM t WHERE ... FOR UPDATE;  -- locks matching rows until commit/rollback
```

Glossary

- **Atomicity**: a transaction's statements succeed or fail together, as one unit — the "A" in ACID.
- **MVCC (Multi-Version Concurrency Control)**: PostgreSQL's strategy of keeping multiple row versions instead of overwriting in place, letting reads and writes proceed without blocking each other.
- **xmin / xmax**: hidden system columns recording which transaction created and (if applicable) superseded a given row version.
- **Row-level lock**: a lock held on specific rows (via `SELECT ... FOR UPDATE`), blocking other transactions from modifying them until it's released.
- **Deadlock**: a cycle of transactions each waiting on a lock the other holds, detected and resolved automatically by PostgreSQL rolling back one of them.