IT5008: Tutorial — BEFORE/AFTER Triggers & Cursors in PostgreSQL

Pratik Karmakar

School of Computing,
National University of Singapore

AY25/26 S1

Agenda

- Database context (borrowers, copies, loans)
- Trigger timing: BEFORE vs AFTER
  - Local (row-level) constraint via BEFORE
  - Global invariant via AFTER (+rollback on exception)
- Step-by-step demo timeline
- Cursors: what/why/how, with example
- Takeaways & pitfalls

Database Context (Minimal)

- `copy(owner, book, copy)`: physical copies of books (book ≈ ISBN13)
- `loan(borrower, owner, book, copy, borrowed, returned)`
  - Active loan: `returned IS NULL`

Goal

Enforce: a borrower may have at most 3 active loans.

BEFORE Trigger (Local Loan Limit)

Intent: Prevent inserting a loan row if the borrower already has 3 active loans.

```
1  CREATE OR REPLACE FUNCTION check_local_loan_limit()
2  RETURNS TRIGGER AS $$
3  DECLARE active_loan_count INT;
4  BEGIN
5    SELECT COUNT(*) INTO active_loan_count
6    FROM loan
7    WHERE borrower = NEW.borrower
8      AND returned IS NULL;
9
10   IF active_loan_count >= 3 THEN
11     RAISE NOTICE 'Loan limit reached for %', NEW.borrower;
12     RETURN NULL;              -- Block the insert
13   END IF;
14
15   RETURN NEW;                 -- Allow insert as-is (or modified)
16 END;
17 $$ LANGUAGE plpgsql;
18
19 CREATE TRIGGER enforce_local_limit
20 BEFORE INSERT ON loan
21 FOR EACH ROW
22 EXECUTE FUNCTION check_local_loan_limit();
```

BEFORE Trigger: Behaviour

- Fires **before** the row is written.
- `RETURN NULL` $\Rightarrow$ cancels the event (no row added, no rollback needed).
- `RETURN NEW` $\Rightarrow$ proceed, possibly with edited values.
- Perfect for **local, per-row** constraints.

Example

`INSERT INTO loan(...)` $\rightarrow$ `NOTICE: Loan limit reached for alice@example.com` $\rightarrow$ blocked.

AFTER Trigger (Global Loan Limit)

Intent: Ensure *no* borrower in the whole system exceeds 3 active loans (detect after write).

```
1  CREATE OR REPLACE FUNCTION check_global_loan_limit()
2  RETURNS TRIGGER AS $$
3  DECLARE violator RECORD;
4  BEGIN
5    SELECT borrower INTO violator
6    FROM loan
7    WHERE returned IS NULL
8    GROUP BY borrower
9    HAVING COUNT(*) > 3;
10
11   IF violator IS NOT NULL THEN
12     RAISE EXCEPTION 'Borrower % exceeds global limit', violator.borrower;
13   END IF;
14
15   RETURN NEW;  -- Required, though AFTER cannot change the row
16 END;
17 $$ LANGUAGE plpgsql;
18
19 CREATE TRIGGER enforce_global_limit
20 AFTER INSERT OR UPDATE ON loan
21 FOR EACH ROW
22 EXECUTE FUNCTION check_global_loan_limit();
```

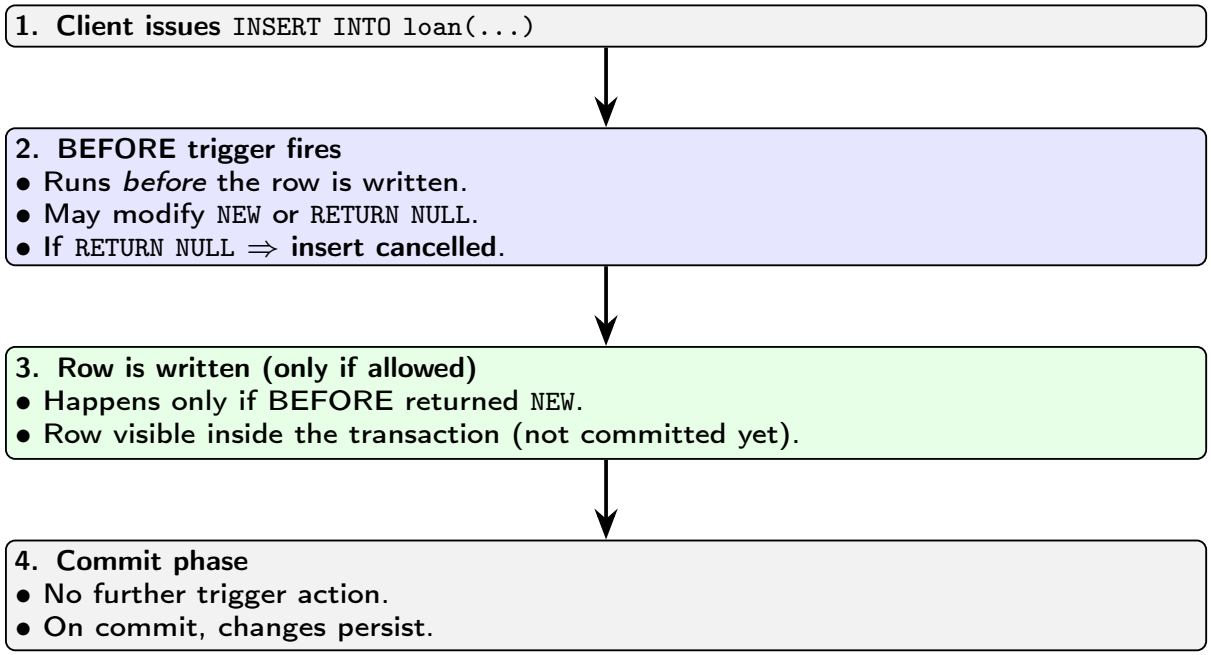
AFTER Trigger: Behaviour & Rollback

- Fires **after** the row is written (but before commit).
- If it `RAISE EXCEPTION`:
  - PostgreSQL **aborts the transaction** automatically.
  - All changes in the current transaction are **rolled back**.
- Ideal for **global invariants**, auditing, notifications.

Effect

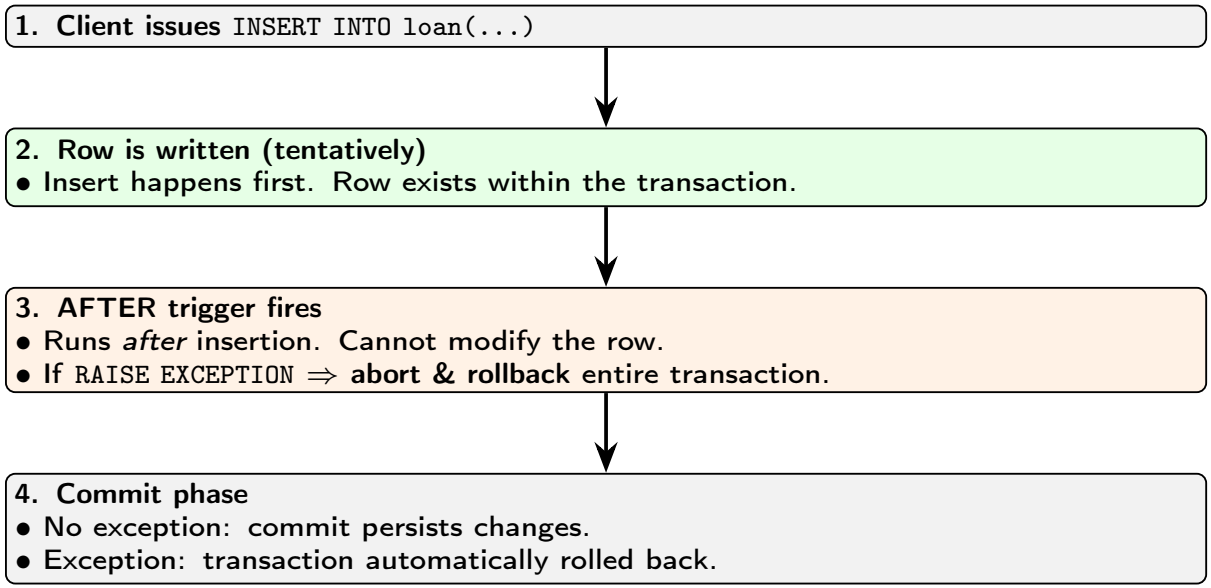
Insert appears to succeed $\rightarrow$ AFTER trigger runs $\rightarrow$ exception $\rightarrow$ transaction rolled back $\Rightarrow$ no net change.

Execution Timeline (INSERT) — **BEFORE** Trigger



Summary: BEFORE can directly **block** the insert by returning `NULL`; otherwise the row proceeds to insert and commit.

Execution Timeline (INSERT) — **AFTER** Trigger



Summary: AFTER cannot block the insert directly; an exception causes an **automatic rollback**.

Story: When we need ONLY a BEFORE trigger

Imagine a library counter on a busy morning. Before writing a loan record, the librarian checks: "Has this person already borrowed three books?"
If yes — she smiles and says, *"Sorry, you've reached your limit."* No new record is written at all.
That's a **BEFORE trigger** — it stops invalid actions *at the door*. It's perfect for quick, local checks or automatic fixes (e.g., fill a missing date or cap a value).
Why not only BEFORE? Because a BEFORE trigger sees just **one row at a time**. It can't reliably check global rules like "no borrower in the system has over 3 active loans," especially when multiple users insert simultaneously.
That's where AFTER comes in.

Story: When we need ONLY an AFTER trigger

Later that day, the librarian reviews the entire logbook. "Hmm... someone now has four active loans!"
She crosses out the last entry and declares, *"This shouldn't have happened — cancel it!"*
That's an **AFTER trigger**. The change was written first, then verified in context. If a rule is violated, PostgreSQL raises an exception and rolls back the entire transaction automatically.
Why AFTER? Use it when correctness depends on the **final state of the table**: aggregates, totals, or system-wide constraints that a BEFORE trigger simply cannot see.

Why AFTER when BEFORE already exists?

Suppose two librarians are on two counters, both helping borrowers at once. Each one checks her own list: "Does this borrower have three books already?" Both see less than three — so both approve a new loan. Now the borrower walks away with four.

That's what happens in a database when two **transactions run concurrently**. Each **BEFORE trigger** only sees its own snapshot — it can't see the other uncommitted insert happening in parallel.

An **AFTER trigger**, on the other hand, fires after all the inserts are actually written. It can finally look at the **true global state** and say, *"Someone now has four loans — roll it back!"*

Moral: BEFORE can stop local mistakes; AFTER is the safety net that catches what concurrency hides.

Story: When we need BOTH — BEFORE and AFTER

On busy days, two librarians share duties.
Librarian A (BEFORE) checks borrowers instantly — fast local guard, no invalid rows even reach the shelf.
Librarian B (AFTER) reviews all records at closing — if any rule still breaks, she cancels the day's batch.
Together they ensure both **speed and global integrity**.
Why combine? BEFORE handles quick per-row validation and normalization; AFTER confirms global consistency and aborts if something slipped through.
It's the safe pattern for concurrent systems — *catch early, verify finally.*

BEFORE DELETE Trigger — Prevent Unsafe Deletion

Intent: Stop deletion of a `loan` record if it is still active (not yet returned).

```
1  CREATE OR REPLACE FUNCTION prevent_active_loan_delete()
2  RETURNS TRIGGER AS $$
3  BEGIN
4    IF OLD.returned IS NULL THEN
5      RAISE EXCEPTION 'Cannot delete active loan for borrower %', OLD.borrower;
6    END IF;
7    RETURN OLD;  -- Allow deletion of already returned loans
8  END;
9  $$ LANGUAGE plpgsql;
10
11 CREATE TRIGGER block_active_loan_delete
12 BEFORE DELETE ON loan
13 FOR EACH ROW
14 EXECUTE FUNCTION prevent_active_loan_delete();
```

Use case: When you want to **block accidental or unsafe deletes** — the BEFORE trigger runs before the row is removed, so you can inspect its old data and decide whether deletion is allowed.

AFTER DELETE Trigger — Roll Back Unsafe Deletion

Intent: Allow the deletion of a `loan` record to happen first, but if it was still **active** (`returned IS NULL`), the system raises an exception and **automatically rolls back** the deletion.

```
1  CREATE OR REPLACE FUNCTION rollback_active_loan_delete()
2  RETURNS TRIGGER AS $$
3  BEGIN
4    IF OLD.returned IS NULL THEN
5      RAISE EXCEPTION 'Cannot delete active loan for borrower %', OLD.borrower;
6    END IF;
7    RETURN OLD;  -- Allow deletion of already returned loans
8  END;
9  $$ LANGUAGE plpgsql;
10
11 CREATE TRIGGER rollback_if_active_delete
12 AFTER DELETE ON loan
13 FOR EACH ROW
14 EXECUTE FUNCTION rollback_active_loan_delete();
```

Behavior:

- The loan row is first deleted (tentatively, inside the transaction).
- The AFTER trigger runs and checks the deleted row using `OLD`.
- If the loan was still active (`returned IS NULL`), it raises an exception — PostgreSQL **automatically rolls back** the entire transaction, restoring the deleted row.
- If the loan had already been returned, the trigger finds no issue, and the deletion **commits successfully**.

Cursors: What & Why

- A **cursor** is a pointer over a query result set.
- Process rows **incrementally** (not all-at-once).
- Useful for large results, row-wise side effects, batched work.
- Transaction-scoped by default (lifetime = session/transaction).

Cursor Example: Iterate Active Loans

```
1  DO $$
2  DECLARE
3    rec RECORD;
4    cur CURSOR FOR
5      SELECT borrower, owner, book, copy
6      FROM loan
7      WHERE returned IS NULL
8      ORDER BY borrower;
9  BEGIN
10   OPEN cur;
11   LOOP
12     FETCH cur INTO rec;
13     EXIT WHEN NOT FOUND;
14     RAISE NOTICE 'Borrower % has % (owner=% copy=%)',
15       rec.borrower, rec.book, rec.owner, rec.copy;
16     -- e.g., perform per-row updates/logging here
17   END LOOP;
18   CLOSE cur;
19 END;
20 $$;
```

Cursor Behaviour & Pitfalls

- `OPEN` executes the query plan and readies the cursor.
- `FETCH` retrieves rows; `NOT FOUND` ends the loop.
- **Holdability**: default cursors close at transaction end; use `WITH HOLD` for post-commit usage.
- **Updates via cursor**: use `FOR UPDATE` and `WHERE CURRENT OF` for safe per-row updates.
- For pure set operations, **prefer SQL**; cursors shine when side effects/order matter.

Cursor Holdability: Keeping a Cursor Alive After Commit

By default, a cursor is bound to the current **transaction**. When you commit, it is automatically closed — this is called **non-holdable**.

Default Behaviour

```
1 BEGIN;
2 DECLARE cur CURSOR FOR SELECT * FROM loan;
3 FETCH NEXT FROM cur;    -- works
4 COMMIT;                 -- closes cursor automatically
5 FETCH NEXT FROM cur;    -- ERROR: cursor "cur" does not exist
```

To keep a cursor alive even after a commit, declare it **WITH HOLD**:

```
1 BEGIN;
2 DECLARE cur CURSOR WITH HOLD FOR SELECT * FROM loan;
3 FETCH NEXT FROM cur;
4 COMMIT;                 -- cursor still usable
5 FETCH NEXT FROM cur;    -- works even after commit
```

Summary: `WITH HOLD` = cursor survives commits (read-only snapshot). Useful for reporting or long read sessions without re-querying.

Updating Rows Safely Through a Cursor

Cursors can also be used to **update or delete** rows one-by-one. To do this, the query must be declared `FOR UPDATE` (or `FOR SHARE`) so PostgreSQL locks each row as it is fetched.

Example: Increment return date safely

```
1  BEGIN;
2  DECLARE cur CURSOR FOR
3    SELECT borrower, book, copy, returned
4    FROM loan
5    WHERE returned IS NULL
6    FOR UPDATE;
7
8  LOOP
9    FETCH cur INTO rec;
10   EXIT WHEN NOT FOUND;
11   UPDATE loan
12     SET returned = CURRENT_DATE
13     WHERE CURRENT OF cur;  -- affects the row just fetched
14 END LOOP;
15
16 CLOSE cur;
17 COMMIT;
```

Updating Rows Safely Through a Cursor

How it works:

- `FOR UPDATE` locks each fetched row, preventing concurrent edits.
- `WHERE CURRENT OF` ensures only that exact row is modified.
- Safe for controlled batch updates; avoids race conditions.

Takeaways

- **BEFORE** triggers can modify/deny the row: `RETURN NULL` blocks the event.
- **AFTER** triggers validate/react post-write: `RAISE EXCEPTION` aborts and **rolls back**.
- Use **BEFORE** for local constraints; **AFTER** for global invariants/auditing.
- **Cursors** enable controlled, row-wise processing for large or side-effect-heavy tasks.

Questions?
Drop a mail at: pratik.karmakar@u.nus.edu