Logging & Auditing in PostgreSQL

Objectives	• Distinguish WAL, application logs, and auditing in PostgreSQL • Understand the three available auditing approaches • Implement a complete custom audit system using PL/pgSQL triggers • Test and validate the mechanism on a real table

Intro Logging and Auditing Mechanisms

PostgreSQL provides three complementary levels of logging, each addressing a different need: engine reliability, technical monitoring, and traceability of user actions.

1. WAL (Write-Ahead Logging)

WAL is PostgreSQL's internal journal. Before any modification (INSERT, UPDATE, DELETE) is applied to the database, the operation is first written to the WAL file. In the event of a crash, PostgreSQL automatically replays any WAL entries not yet applied, ensuring that no committed data is ever lost. This mechanism is always active, stored in binary format under `$PGDATA/pg_wal/`, and is not human-readable. It also supports server-to-server replication and point-in-time recovery (PITR).

2. Application Logs

Application logs are human-readable text files configurable in `postgresql.conf`. They record connections, disconnections, SQL errors, slow queries, and server events. Unlike the WAL, they are readable by an administrator but cannot restore data. Key parameters: `log_statement = 'mod'` traces all write operations, `log_min_duration_statement = 500` flags queries exceeding 500 ms, and `log_connections = on` records every incoming connection.

3. Auditing

Auditing answers a specific question: *who changed what, when, and what was the value before?* This is a common requirement in banking, healthcare, and government applications. **Native logs** are quick to set up but do not preserve row-level values. The **pgAudit extension** provides finer-grained auditing of DDL statements and specific users. **Custom trigger-based auditing** is the most comprehensive solution: each modification automatically fires a PL/pgSQL function that records the before and after values in a dedicated audit table, offering full traceability tailored to business requirements.

> **Note:** WAL ensures internal engine consistency. Application logs serve monitoring and diagnostics. Auditing addresses traceability and compliance requirements. These three mechanisms are independent and complementary.

Ex. Implementing a Custom Audit System

Consider a database used by a banking agency. Your manager asks you to set up an audit system that automatically tracks all modifications made to the following table:

```
CREATE TABLE bank_accounts (
    id        SERIAL PRIMARY KEY,
    holder    TEXT    NOT NULL,
    balance   NUMERIC NOT NULL,
    branch    TEXT
);
```

Logging & Auditing in PostgreSQL

	Question and instructions
Q1	**Create the audit table.** The table `audit_accounts` must store, for each detected operation: the name of the PostgreSQL user who performed the action (`current_user` returns this value automatically), the operation type as text (INSERT, UPDATE, or DELETE), the old balance before the change (NULL for an INSERT), the new balance after the change (NULL for a DELETE), and the exact date and time. Use `NUMERIC` for balances to preserve precision, and `TIMESTAMP DEFAULT now()` to timestamp each row automatically.
Q2	**Write the trigger function.** A PL/pgSQL trigger function must return the type `trigger`. The variable `TG_OP` holds the operation name. Use an `IF / ELSIF` block to handle each case: on INSERT, only `NEW` is available; on DELETE, only `OLD`; on UPDATE, both. For UPDATE, compare `OLD.balance <> NEW.balance` before inserting — this avoids logging changes that do not affect the balance. At the start, check `current_user`: if the user is 'admin', return `NEW` immediately without recording anything.
Q3	**Create and attach the trigger.** An `AFTER` trigger fires after the modification has been applied, guaranteeing that `OLD` and `NEW` are final. The `FOR EACH ROW` clause means the function is called once per modified row. The trigger must listen to INSERT, UPDATE, and DELETE on `bank_accounts`.
Q4	**Test and validate your solution.** Run the statements below in order and query `audit_accounts` after each one. Verify that: the INSERT produces one row with the new balance and NULL for the old; the first UPDATE (balance change) produces a row with both values; the second UPDATE (branch only) produces no row in the audit; the DELETE produces a row with the old balance and NULL for the new. Comment on each row returned.

```
INSERT INTO bank_accounts (holder, balance, branch)
VALUES ('Ahmed Ben Ali', 5000, 'Tunis Centre');

UPDATE bank_accounts SET balance = 4200 WHERE id = 1;

UPDATE bank_accounts SET branch = 'Sfax';  -- must NOT appear in audit

DELETE FROM bank_accounts WHERE id = 1;

-- Expected: 3 rows (INSERT, balance UPDATE, DELETE)
SELECT * FROM audit_accounts ORDER BY action_time;
```

Reminder	**`NEW`** = row after the change (INSERT, UPDATE) **`OLD`** = row before the change (UPDATE, DELETE) **`TG_OP`** = operation name ('INSERT', 'UPDATE', 'DELETE') **`TG_TABLE_NAME`** = name of the table that fired the trigger.

Bonus +2 pts	Modify the function so that it raises an exception using `RAISE EXCEPTION 'Negative balance not allowed'` if the new balance falls below zero on an INSERT or UPDATE. The transaction must be automatically rolled back in this case.