Database Security and Forensics — MASTER 1 CS

LAB ASSESSMENT 2/2 — Logging and Auditing in PostgreSQL

Full Name		Group	
Date	Tuesday 13 May 2026 — Lecture Hall 2	Duration	45 minutes

Instructions

Each question has exactly one correct answer. Read every option carefully before answering — some choices are intentionally close. Circle or tick the corresponding letter on your answer sheet. A correct answer scores 1 point; an incorrect or blank answer scores 0. No penalty is applied. Illegible or ambiguous answers will be marked as null.

Part A — Lab Work (Q1–Q14) → 14 points / 70 %	Part B — Conceptual Understanding (Q15–Q20) → 6 points / 30 %

Part A — Lab Work

Questions based directly on the code and tests performed during the lab session (Q1 to Q14 — 14 points)

Q1. A student defines the trigger as follows. What fundamental error prevents access to the OLD and NEW variables inside the function?

```
CREATE TRIGGER trg_audit
AFTER INSERT OR UPDATE OR DELETE ON bank_accounts
FOR EACH STATEMENT
EXECUTE FUNCTION audit_fn();
```

□ **A.** The keyword AFTER must be replaced by BEFORE in order to access OLD and NEW

□ **B.** FOR EACH STATEMENT means the trigger fires only once per statement and OLD / NEW are not available

□ **C.** Using OR between INSERT, UPDATE and DELETE is invalid — three separate triggers are required

□ **D.** EXECUTE FUNCTION must be replaced by EXECUTE PROCEDURE in PostgreSQL

Q2. A student runs the following statement on a row whose current balance is 4,200:

```
UPDATE bank_accounts SET balance = balance WHERE id = 1;
```

What happens in audit_accounts?

□ **A.** One row is inserted: the AFTER UPDATE trigger always fires regardless of the value

□ **B.** No row: PostgreSQL detects that the value has not changed and does not execute the trigger

□ **C.** No row: the trigger does fire, but the condition OLD.balance <> NEW.balance is false

□ **D.** An exception is raised because balance = balance is a circular expression that is invalid in SQL

Q3. In an AFTER FOR EACH ROW trigger, what happens if the function returns NULL instead of RETURN NEW?

- □ **A.** The original operation (INSERT / UPDATE / DELETE) is automatically cancelled
- □ **B.** The return value is ignored: an AFTER trigger can neither cancel nor modify the operation already applied
- □ **C.** PostgreSQL raises an exception because NULL is not a permitted return value in a trigger
- □ **D.** The row is not inserted into audit_accounts because RETURN NULL short-circuits the rest of the function

Q4. In the lab's trigger function, if current_user = 'admin' performs a DELETE, the function immediately executes RETURN NEW. What subtle issue does this raise?

- □ **A.** NEW holds the row before deletion, so the audit mistakenly records it anyway
- □ **B.** On a DELETE, NEW is NULL; but in an AFTER trigger the return value is ignored, so no error occurs
- □ **C.** RETURN NEW inside a DELETE automatically triggers a ROLLBACK of the transaction
- □ **D.** PostgreSQL raises an exception because NEW cannot be referenced during a DELETE

Q5. A student forgets the ELSIF block for UPDATE and writes only the following:

```
IF TG_OP = 'INSERT' THEN
  INSERT INTO audit_accounts ... VALUES (current_user, TG_OP, NULL,
NEW.balance);
ELSIF TG_OP = 'DELETE' THEN
  INSERT INTO audit_accounts ... VALUES (current_user, TG_OP,
OLD.balance, NULL);
END IF;
RETURN NEW;
```

What happens when a balance UPDATE is performed?

- □ **A.** PostgreSQL raises an exception because TG_OP = 'UPDATE' is not handled
- □ **B.** The UPDATE is not audited: the IF/ELSIF block completes without any INSERT into audit_accounts
- □ **C.** RETURN NEW causes an error because both OLD and NEW are NULL in this context
- □ **D.** A row is inserted with old_balance = NULL and new_balance = NULL

Q6. If the trigger had been defined as BEFORE instead of AFTER, what specific problem would occur on a DELETE?

- □ **A.** OLD.balance would be NULL because the row does not yet exist at the time of the BEFORE trigger
- □ **B.** A BEFORE trigger cannot perform an INSERT into a third-party table
- □ **C.** OLD would hold correct values, but the audit would record a DELETE that might never complete if the operation subsequently fails
- □ **D.** TG_OP would equal 'BEFORE_DELETE' instead of 'DELETE', causing a mismatch in the logic

Q7. The table bank_accounts contains 3 rows with different balances. The following statement is executed:

```
UPDATE bank_accounts SET balance = balance * 1.10;
```

How many rows are inserted into audit_accounts (FOR EACH ROW trigger active, condition OLD.balance <> NEW.balance present)?

- □ **A.** 0 rows: the condition OLD.balance <> NEW.balance is false because the same column is reused
- □ **B.** 1 row: the AFTER UPDATE trigger fires only once for the entire statement
- □ **C.** 3 rows: FOR EACH ROW calls the function once per affected row, and balance * 1.10 ≠ balance for each
- □ **D.** 3 rows only if each UPDATE is executed in a separate transaction

Q8. Why use current_user rather than session_user in the trigger function to record the author of the action?

- □ **A.** current_user and session_user always return the same result in a trigger context
- □ **B.** session_user returns the client's IP address, while current_user returns the active role name
- □ **C.** current_user reflects any active SET ROLE, identifying the role actually in use; session_user remains the originally authenticated user
- □ **D.** current_user is the only one available inside a trigger function; session_user raises an exception in that context

Q9. A student executes the following within a single transaction:

```
BEGIN;
  INSERT INTO bank_accounts(holder, balance, branch) VALUES ('Alice',
1000, 'Tunis');
  INSERT INTO bank_accounts(holder, balance, branch) VALUES ('Bob',
-500, 'Sfax');
COMMIT;
```

The bonus (RAISE EXCEPTION if balance < 0) is active. How many rows does audit_accounts contain after execution?

- □ **A.** 1 row: Alice's row is committed before the exception is raised for Bob
- □ **B.** 2 rows: both INSERTs are recorded in the audit before the exception is handled
- □ **C.** 0 rows: RAISE EXCEPTION rolls back the entire transaction, including Alice's audit row already inserted
- □ **D.** 1 row with an error flag for Bob, automatically inserted by the exception handler

Q10. An UPDATE with a WHERE clause that matches no rows is executed:

```
UPDATE bank_accounts SET balance = -100 WHERE id = 999;  -- id 999
does not exist
```

The bonus (RAISE EXCEPTION if balance < 0) is active. What happens?

- □ **A.** RAISE EXCEPTION is triggered because balance = -100 < 0, even though no row is affected
- □ **B.** No exception: the trigger does not fire if no rows are modified
- □ **C.** A 'row not found' error is raised by PostgreSQL before the trigger even executes

□ **D.** The trigger runs with NEW.balance = -100 and raises the exception, rolling back the transaction

Q11. Three operations are executed in order on a row whose initial balance is 5,000:

```
UPDATE bank_accounts SET balance = 5000 WHERE id = 1;  -- same value
UPDATE bank_accounts SET balance = 4500 WHERE id = 1;
UPDATE bank_accounts SET balance = 4500 WHERE id = 1;  -- same value
```

How many rows appear in audit_accounts in total after these three statements?

□ **A.** 3 rows: every UPDATE fires the AFTER UPDATE trigger
□ **B.** 1 row: only the second UPDATE actually changes the balance
□ **C.** 2 rows: the first two UPDATEs are audited, the third is not
□ **D.** 0 rows: PostgreSQL optimises and skips the trigger if the final value is identical

Q12. The statement DROP TABLE bank_accounts CASCADE is executed. What happens exactly?

□ **A.** PostgreSQL forbids DROP TABLE on a table that has an active trigger
□ **B.** The trigger trg_audit is dropped (CASCADE), but audit_accounts is preserved with all its rows
□ **C.** audit_accounts is automatically truncated (CASCADE), but its structure is kept
□ **D.** Both the trigger and audit_accounts are dropped because audit_accounts depends on bank_accounts

Q13. The trigger function inserts into audit_accounts (AFTER trigger). If that INSERT fails (e.g. NOT NULL constraint violation), what happens?

□ **A.** The failure is silently ignored: the modification to bank_accounts is preserved normally
□ **B.** Only the INSERT into audit_accounts is rolled back; the modification to bank_accounts is kept
□ **C.** The exception propagates and rolls back the entire transaction, including the operation on bank_accounts
□ **D.** PostgreSQL automatically generates a default value to bypass the constraint

Q14. (Bonus) The table contains id = 1 with balance = 500. The following is executed:

```
UPDATE bank_accounts SET balance = 500 - 600 WHERE id = 1;  -- result:
-100
```

The bonus RAISE EXCEPTION is active in the AFTER trigger. What values of OLD.balance and NEW.balance are visible inside the trigger function at the moment the exception is raised?

□ **A.** OLD.balance = 500, NEW.balance = -100 : the AFTER trigger sees the definitive values after the update is applied
□ **B.** OLD.balance = 500, NEW.balance = 500 : the modification is undone before the AFTER trigger runs
□ **C.** OLD.balance = NULL, NEW.balance = -100 : OLD is not available in an AFTER UPDATE trigger
□ **D.** OLD.balance = -100, NEW.balance = 500 : the values are swapped in an AFTER trigger

Part B — Conceptual Understanding

Questions on the fundamental concepts introduced at the beginning of the lab (Q15 to Q20 — 6 points)

Q15. An administrator sets fsync = off in postgresql.conf to improve write performance. What consequence does this have on the guarantee provided by the WAL?

- □ **A.** WAL is no longer written at all; the database becomes entirely vulnerable to crashes
- □ **B.** The durability guarantee is lost: in the event of a system crash, committed transactions can be lost even though WAL is active
- □ **C.** WAL files are written in human-readable text instead of binary, reducing performance
- □ **D.** Only DDL transactions lose their guarantee; DML transactions remain protected

Q16. With log_statement = 'mod', which statements appear in the application logs?

- □ **A.** INSERT and UPDATE only; DELETE and TRUNCATE are excluded from the 'mod' setting
- □ **B.** INSERT, UPDATE and DELETE only; TRUNCATE is not considered DML in this context
- □ **C.** INSERT, UPDATE, DELETE and TRUNCATE; the 'mod' value covers all data-modifying operations
- □ **D.** All statements including SELECT; 'mod' is an alias of 'all' in recent PostgreSQL versions

Q17. A security officer requests auditing of all CREATE TABLE, DROP INDEX and ALTER TABLE attempts by developers. Which solution is most appropriate?

- □ **A.** PL/pgSQL trigger-based audit: it captures all DML and DDL operations on monitored tables
- □ **B.** pgAudit: triggers cannot intercept DDL operations, unlike pgAudit which operates at the engine level
- □ **C.** log_statement = 'mod': this value also covers DDL in addition to DML write operations
- □ **D.** WAL: it records all physical modifications, including structural changes caused by DDL

Q18. Point-In-Time Recovery (PITR) uses WAL files. What additional element is absolutely required?

- □ **A.** A saved postgresql.conf file, to reconfigure the recovery server
- □ **B.** A base backup taken before the target recovery point, from which the WAL files are replayed
- □ **C.** Application logs for the relevant period, to identify which queries must be reversed
- □ **D.** A PL/pgSQL script generated by pg_waldump that reverses each recorded operation

Q19. An external auditor asks: "Can you extract from the WAL the exact list of SQL queries executed yesterday?" Which answer is technically correct?

- □ **A.** Yes, pg_waldump can read and display the original SQL queries from WAL files
- □ **B.** No: the WAL records physical block-level changes, not the text of SQL statements
- □ **C.** Yes, but only for DDL queries; DML queries are not readable in the WAL
- □ **D.** No, because WAL files are automatically deleted after each CHECKPOINT operation

Q20. Three mechanisms are available: WAL, application logs, trigger-based audit. A security incident reveals that a balance was fraudulently modified. Which mechanism answers "who / what value before / what value after / at what exact time"?

□ **A.** WAL only: it holds physical row-level changes with timestamps

□ **B.** Application logs with log_statement = 'mod': they record the full SQL statement along with the user

□ **C.** PL/pgSQL trigger-based audit: the only mechanism that explicitly preserves before/after values per row together with the user identity

□ **D.** pgAudit combined with application logs: pgAudit appends per-row values to the existing logs

ANSWER KEY

For instructor use only — Do not distribute to students

Part A — Lab Work

Question	Answer	Justification
Q1	**B**	FOR EACH STATEMENT → OLD and NEW are unavailable
Q2	**C**	Trigger fires but OLD.balance = NEW.balance → condition false → no INSERT
Q3	**B**	In an AFTER trigger the return value is ignored — RETURN NULL has no effect
Q4	**B**	NEW = NULL on DELETE, but AFTER trigger ignores the return value — no error
Q5	**B**	No ELSIF for UPDATE → the case is silently skipped, no audit row inserted
Q6	**C**	BEFORE: audit records a DELETE that may never actually complete
Q7	**C**	balance * 1.10 ≠ balance for each row → condition true → 3 rows inserted
Q8	**C**	current_user reflects SET ROLE; session_user = originally authenticated user
Q9	**C**	RAISE EXCEPTION rolls back the whole transaction, including Alice's audit row
Q10	**B**	No rows matched → trigger does not fire at all
Q11	**B**	Only the 2nd UPDATE changes the balance (5000→4500) → 1 row inserted
Q12	**B**	CASCADE drops the trigger; audit_accounts is an independent table — preserved
Q13	**C**	Failure inside the trigger propagates and rolls back the entire transaction
Q14	**A**	AFTER trigger: OLD = value before, NEW = applied value (−100)

Part B — Conceptual Understanding

Question	Answer	Justification
Q15	**B**	Without fsync, WAL may stay in memory and be lost on a system crash
Q16	**C**	'mod' covers INSERT, UPDATE, DELETE and also TRUNCATE
Q17	**B**	Triggers cannot intercept DDL → pgAudit is required
Q18	**B**	PITR = base backup + WAL files replayed from that backup point
Q19	**B**	WAL = physical block changes, not SQL text
Q20	**C**	Only the trigger preserves who / before value / after value / timestamp per row

Part A: Q1-B Q2-C Q3-B Q4-B Q5-B Q6-C Q7-C

Part B: Q15-B Q16-C Q17-B Q18-B Q19-B Q20-C

Q8-C Q9-C Q10-B Q11-B Q12-B Q13-C Q14-A	

✦ **Good luck** ✦