# **Laporan Tutorial Database — Trigger, Views & Materialized Views**

**Judul:** Implementasi Trigger, Views, dan Materialized Views pada Studi Kasus E‑Commerce

**Nama:** Raihan Muhammad Riswandi

**NIM:** 24110500001

**Mata Kuliah:** advanced database

**Tanggal:** 10/10/2025

## **1. Deskripsi Studi Kasus**

Domain: **E‑Commerce (TokoElektronik)**

Skenario singkat:

* TokoElektronik menjual produk elektronik (headphone, keyboard, mouse, dll.).
* Sistem menyimpan data customers, products, orders, dan order\_items.
* Ketentuan bisnis:  
  + Saat item dipesan, stok produk harus berkurang otomatis.
  + Jika stok tidak mencukupi, transaksi gagal (dibatalkan).
  + Semua perubahan stok dicatat (audit/log).
  + Laporan penjualan harian perlu dibuat cepat (agregasi besar) — di sini materialized view berguna.

Alasan memilih domain ini: mudah dimengerti, relevan, dan mencakup kebutuhan trigger (business rule), views (abstraksi & keamanan), serta materialized views (laporan agregat yang berat).

## **2. Penjelasan Singkat Konsep (dengan bahasa sendiri)**

**Trigger**: potongan kode yang otomatis dieksekusi oleh database saat peristiwa (INSERT/UPDATE/DELETE) terjadi pada tabel. Berguna untuk menegakkan aturan bisnis, melakukan auditing, atau memelihara data turunan.

**View**: query tersimpan yang berfungsi sebagai lapisan abstraksi/virtual table. Tidak menyimpan data (pada RDBMS biasa) sehingga selalu up‑to‑date, cocok untuk menyederhanakan query kompleks atau memberi hak akses terbatas.

**Materialized View**: seperti view tapi menyimpan hasil query (persistent). Berguna untuk mempercepat query agregat yang berat. Kelemahannya: harus di‑refresh agar data kembali fresh.

Perbedaan kunci:

* View = tidak ada penyimpanan data (hasil dihitung saat query) — selalu fresh.
* Materialized view = menyimpan hasil; baca sangat cepat tapi perlu refresh.

## **3. Skema Database (DDL)**

Catatan: contoh berikut menggunakan **PostgreSQL** (karena mendukung CREATE MATERIALIZED VIEW). Jika pakai MySQL, bagian materialized view perlu di‑simulasikan (lampiran ada metode alternatif).

-- Tabel master

CREATE TABLE customers (

customer\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(150) UNIQUE NOT NULL,

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE products (

product\_id SERIAL PRIMARY KEY,

sku VARCHAR(50) UNIQUE NOT NULL,

name VARCHAR(200) NOT NULL,

price NUMERIC(12,2) NOT NULL CHECK (price >= 0),

stock INTEGER NOT NULL CHECK (stock >= 0),

created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE orders (

order\_id SERIAL PRIMARY KEY,

customer\_id INTEGER NOT NULL REFERENCES customers(customer\_id),

order\_date TIMESTAMP DEFAULT now(),

status VARCHAR(30) NOT NULL DEFAULT 'pending',

total NUMERIC(12,2) DEFAULT 0

);

CREATE TABLE order\_items (

order\_item\_id SERIAL PRIMARY KEY,

order\_id INTEGER NOT NULL REFERENCES orders(order\_id) ON DELETE CASCADE,

product\_id INTEGER NOT NULL REFERENCES products(product\_id),

unit\_price NUMERIC(12,2) NOT NULL CHECK (unit\_price >= 0),

quantity INTEGER NOT NULL CHECK (quantity > 0)

);

-- Tabel audit stock

CREATE TABLE stock\_log (

log\_id SERIAL PRIMARY KEY,

product\_id INTEGER NOT NULL REFERENCES products(product\_id),

change INTEGER NOT NULL,

reason VARCHAR(200),

log\_time TIMESTAMP DEFAULT now()

);

## **4. Data Contoh (sample inserts)**

INSERT INTO customers(name,email) VALUES

('Budi','budi@example.com'),

('Siti','siti@example.com');

INSERT INTO products(sku,name,price,stock) VALUES

('SKU-001','Headphone',150.00,50),

('SKU-002','Keyboard',80.00,30),

('SKU-003','Mouse',25.00,100);

-- contoh order (buat dulu order, nanti item ditambahkan sehingga trigger berjalan)

INSERT INTO orders(customer\_id,status) VALUES (1,'pending'); -- order\_id = 1

## **5. Implementasi Trigger (kode lengkap + penjelasan)**

### **5.1. Tujuan trigger**

* Pastikan stok tidak menjadi negatif saat order\_items dimasukkan.
* Kurangi stok produk saat item ditambahkan.
* Catat perubahan stok ke stock\_log.
* Perbarui total pada tabel orders setiap ada perubahan order\_items.

### **5.2. Fungsi & Triggers (PostgreSQL)**

-- 1) Fungsi yang menurunkan stok saat ada insert ke order\_items

CREATE OR REPLACE FUNCTION fn\_decrease\_stock() RETURNS TRIGGER AS $$

DECLARE

current\_stock INTEGER;

BEGIN

-- ambil & lock baris produk agar race condition terhindar

SELECT stock INTO current\_stock FROM products WHERE product\_id = NEW.product\_id FOR UPDATE;

IF current\_stock IS NULL THEN

RAISE EXCEPTION 'Product % does not exist', NEW.product\_id;

END IF;

IF current\_stock < NEW.quantity THEN

RAISE EXCEPTION 'Not enough stock for product\_id % (available %, required %)', NEW.product\_id, current\_stock, NEW.quantity;

END IF;

UPDATE products SET stock = current\_stock - NEW.quantity WHERE product\_id = NEW.product\_id;

INSERT INTO stock\_log(product\_id, change, reason) VALUES (NEW.product\_id, -NEW.quantity, concat('order\_item inserted, order\_id=', NEW.order\_id));

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trg\_decrease\_stock

BEFORE INSERT ON order\_items

FOR EACH ROW EXECUTE FUNCTION fn\_decrease\_stock();

-- 2) Fungsi yang memperbarui total order setelah insert/update/delete order\_items

CREATE OR REPLACE FUNCTION fn\_update\_order\_total() RETURNS TRIGGER AS $$

DECLARE p\_order\_id INTEGER;

BEGIN

IF (TG\_OP = 'INSERT') THEN

p\_order\_id := NEW.order\_id;

ELSIF (TG\_OP = 'UPDATE') THEN

p\_order\_id := NEW.order\_id;

ELSIF (TG\_OP = 'DELETE') THEN

p\_order\_id := OLD.order\_id;

END IF;

UPDATE orders

SET total = COALESCE((SELECT SUM(unit\_price \* quantity) FROM order\_items WHERE order\_id = p\_order\_id), 0)

WHERE order\_id = p\_order\_id;

IF (TG\_OP = 'DELETE') THEN

RETURN OLD;

ELSE

RETURN NEW;

END IF;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trg\_update\_order\_total

AFTER INSERT OR UPDATE OR DELETE ON order\_items

FOR EACH ROW EXECUTE FUNCTION fn\_update\_order\_total();

-- 3) Fungsi untuk mencatat perubahan stok saat admin update stock di products

CREATE OR REPLACE FUNCTION fn\_log\_stock\_change() RETURNS TRIGGER AS $$

DECLARE diff INTEGER;

BEGIN

diff := NEW.stock - OLD.stock;

IF diff <> 0 THEN

INSERT INTO stock\_log(product\_id, change, reason) VALUES (NEW.product\_id, diff, concat('manual stock update, op=', TG\_OP));

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trg\_log\_stock\_change

AFTER UPDATE OF stock ON products

FOR EACH ROW EXECUTE FUNCTION fn\_log\_stock\_change();

**Penjelasan singkat**:

* fn\_decrease\_stock menggunakan SELECT ... FOR UPDATE untuk lock baris produk sehingga dua transaksi bersamaan tidak membuat stok negatif.
* Bila stok tidak cukup, fungsi me-RAISE EXCEPTION — transaksi akan rollback.
* fn\_update\_order\_total menjaga kolom orders.total selalu sinkron dengan data order\_items.
* fn\_log\_stock\_change memberi jejak audit bila ada perubahan stok manual.

## **6. Implementasi Views (kode + penggunaan)**

### **6.1. Contoh Views**

-- View sederhana: ringkasan order

CREATE VIEW vw\_order\_summary AS

SELECT o.order\_id, c.name AS customer\_name, o.order\_date, o.status, o.total

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id;

-- View detail tiap order item yang memudahkan UI/backend

CREATE VIEW vw\_order\_items AS

SELECT oi.order\_item\_id, oi.order\_id, p.name AS product\_name, oi.unit\_price, oi.quantity, (oi.unit\_price \* oi.quantity) AS line\_total

FROM order\_items oi

JOIN products p ON oi.product\_id = p.product\_id;

### **6.2. Kapan dan kenapa pakai views?**

* Menyederhanakan query kompleks untuk front-end.
* Memberi lapisan abstraksi sehingga struktur tabel internal bisa berubah tanpa mengubah aplikasi.
* Keamanan — dapat memberi hak akses SELECT hanya ke view dan menyembunyikan kolom sensitif.
* Views selalu menampilkan data up‑to‑date (tidak menyimpan hasil).

## **7. Implementasi Materialized Views (kode + penggunaan)**

### **7.1. Contoh Materialized View — Laporan Penjualan Harian**

CREATE MATERIALIZED VIEW mv\_daily\_sales AS

SELECT date\_trunc('day', o.order\_date) AS day,

COUNT(DISTINCT o.order\_id) AS orders\_count,

SUM(oi.quantity) AS total\_quantity,

SUM(oi.quantity \* oi.unit\_price) AS total\_amount

FROM orders o

JOIN order\_items oi ON o.order\_id = oi.order\_id

WHERE o.status <> 'canceled'

GROUP BY day

ORDER BY day;

-- buat unique index supaya bisa REFRESH CONCURRENTLY nanti

CREATE UNIQUE INDEX idx\_mv\_daily\_sales\_day ON mv\_daily\_sales (day);

### **7.2. Cara refresh & catatan**

**Manual refresh (lock selama refresh):** REFRESH MATERIALIZED VIEW mv\_daily\_sales;

**Refresh tanpa blokir pembaca (concurrent):** REFRESH MATERIALIZED VIEW CONCURRENTLY mv\_daily\_sales;

* *Catatan:* REFRESH ... CONCURRENTLY membutuhkan unique index pada materialized view dan tidak dapat dijalankan dalam transaksi (harus autocommit).

### **7.3. Keuntungan materialized view**

* Membaca agregat besar jauh lebih cepat karena hasil sudah tersimpan.
* Cocok untuk reporting/dashboard.

### **7.4. Kelemahan**

* Perlu strategi refresh (real‑time vs interval). Jika data butuh real‑time, materialized view mungkin tidak cocok.
* Menambah storage dan pemeliharaan (refresh biaya IO/CPU).

## **8. Contoh Skema Pengujian / Langkah Eksekusi (apa yang harus di‑screenshot)**

**Langkah 1 — Buat skema & data sample**

* Jalankan DDL di atas. Screenshot output sukses (psql/pgAdmin).

**Langkah 2 — Test trigger stok**

Buat order dan tambahkan item: contoh:  
  
 INSERT INTO orders(customer\_id,status) VALUES (1,'pending') RETURNING order\_id;

INSERT INTO order\_items(order\_id, product\_id, unit\_price, quantity) VALUES (1, 1, 150.00, 2);

* Screenshot hasil: SELECT stock FROM products WHERE product\_id=1; (stok berkurang), SELECT \* FROM stock\_log WHERE product\_id=1 ORDER BY log\_time DESC LIMIT 1;, dan SELECT total FROM orders WHERE order\_id=1;.

**Langkah 3 — Test error saat stok tidak cukup**

* Coba insert quantity jauh lebih besar dari stok (mis. 1000) lalu screenshot pesan error di console.

**Langkah 4 — Test views**

* Jalankan SELECT \* FROM vw\_order\_summary; dan SELECT \* FROM vw\_order\_items; serta screenshot hasilnya.

**Langkah 5 — Test materialized view**

* Jalankan REFRESH MATERIALIZED VIEW mv\_daily\_sales; lalu SELECT \* FROM mv\_daily\_sales ORDER BY day DESC LIMIT 10; dan screenshot hasilnya.
* Jika ingin menunjukkan perbedaan performa, jalankan query agregat langsung vs membaca dari materialized view dan bandingkan runtime (EXPLAIN ANALYZE).

**Langkah 6 — Dokumentasikan error dan solusi**

* Jika ada error (contoh: exception saat insert karena stok tidak cukup), screenshot error dan tuliskan di laporan bagaimana memperbaikinya.

## **9. Analisis Singkat — Kenapa pendekatan ini?**

* Trigger menangani business rule stok secara atomik di sisi database sehingga aplikasi tidak perlu mengingatkan banyak logic (centralized business rules).
* Views memberikan lapisan presentasi sederhana untuk UI dan memudahkan developer frontend.
* Materialized view digunakan untuk laporan agregat (daily sales) yang mahal dihitung tiap kali—memberi trade‑off antara konsistensi data dan performa baca.

Resiko & mitigasi:

* Trigger yang kompleks dapat menyulitkan debugging; selalu tambahkan logging dan dokumentasi.
* Materialized view perlu kebijakan refresh (mis. cron job setiap 10 menit atau manual saat batch selesai).

## **10. Kesimpulan Pembelajaran**

* Trigger, view, dan materialized view tiap‑tiap punya peran berbeda: integrity & automation (trigger), abstraksi & keamanan (view), dan performa reporting (materialized view).
* Pilih tool yang sesuai kebutuhan: jika butuh real‑time konsistensi pilih trigger/view; jika butuh performa baca untuk agregasi besar pilih materialized view dengan jadwal refresh.