E-Commerce Platform Database

Reflection on Advanced SQL and DevOps Practices

Sajid
Database Management Systems
Option 3: Coding-Heavy DBA / DevOps Project

October 9, 2025

Abstract

This report reflects on the design, implementation, and operationalisation of a PostgreSQL database that models a contemporary e-commerce platform. The project satisfied and surpassed the Option 3 requirements by delivering eleven normalized tables, realistic transactional datasets, advanced SQL features, rigorous security, and thorough documentation. I highlight engineering decisions, debugging lessons, and the DevOps mindset that shaped the final deliverable.

1 Why E-Commerce?

I selected the e-commerce domain because it demands breadth and depth: hierarchical catalogs, multi-channel inventory, customer lifecycle management, complex financial flows, and rigorous auditing. Building this database offered a hands-on way to explore problems faced by marketplaces such as Amazon or Shopify while applying the PostgreSQL features covered in class.

2 Design Objectives

- Model the entire purchasing journey—from browsing to payment settlement—using normalized schemas.
- Demonstrate advanced SQL through window functions, recursive CTEs, and analytics-ready views.
- Automate business logic with triggers that protect data integrity and capture operational events.
- Implement defense-in-depth security, including Row-Level Security (RLS) and detailed audit logging.
- Produce professional documentation and maintenance scripts to support real operations.

3 Schema Highlights

The final schema spans eleven tables and more than 300 rows of curated data. Notable design choices include:

- **Categories** form a self-referencing tree, enabling multi-level navigation (Electronics → Smartphones → Android).
- **Products** capture merchandising metadata, cost vs. price, stock levels, and aggregated review statistics.
- **Orders** and **order_items** model complex carts with multi-item purchases, discounts, taxes, and shipping.
- **Payment transactions** represent integration with external processors and track settlement outcomes.
- **Audit log** records before/after JSON payloads for sensitive operations, supporting compliance and forensics.

Realistic data was essential. Product pricing aligns with actual market ranges, shipping addresses map to Boston neighbourhoods, and reviews reflect sentiment typical of consumer platforms. This realism makes analytical outputs trustworthy during demos and testing.

4 Advanced SQL in Practice

Window Functions

I relied heavily on window functions for analytics:

- ‘NTILE(4)‘ segments customers into loyalty quartiles based on lifetime spend.
- ‘LAG‘ compares current month revenue against the previous month to identify growth trends.
- ‘ROW$_NUMBER$‘$ranksbest-sellingproductswithineachcategorytodrivemerchandisingdecision$

Recursive CTEs

Recursive CTEs powered the dynamic category menu. Early iterations suffered from duplicate rows; ‘DISTINCT ON‘ combined with depth tracking ultimately produced a predictable tree.

Materialized Views

Two materialized views aggregate sales KPIs and product performance metrics. They refresh via a stored procedure:

```
CREATE OR REPLACE FUNCTION refresh_commerce_analytics()
RETURNS void LANGUAGE plpgsql AS $$
BEGIN
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_analytics;
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_product_performance;
END;
$$;
```

Running these after nightly ETL loads keeps executive dashboards responsive without burdening transaction tables.

5 Automation and Triggers

Six triggers orchestrate business rules:

- Inventory decrements when orders are placed and replenishes on cancellation or refund.
- ‘cart$_{i}$*tems*‘*maintains*‘*updated*$_{a}$*t*‘*timestampsautomatically*.
- Review inserts recalculate average product ratings and review counts.
- Payment status changes push structured entries into the audit log.

Debugging triggers revealed the importance of order-of-operations. For example, subtracting stock before validating availability created negative inventory edge cases. Wrapping logic in ‘RAISE EXCEPTION‘ blocks and writing targeted tests eliminated those issues.

6 Security and Governance

Protecting customer data required layered controls:

- Four roles (‘commerce$_{a}$*dmin*‘, ‘*commerce*$_{m}$*anager*‘, ‘*commerce*$_{c}$*ustomer*‘, ‘*commerce*$_{r}$*eadonly*‘)*ma*

- RLS policies ensure customers only view their own carts, orders, addresses, and reviews. Staff roles retain oversight for support and analytics.
- Session helper functions (‘set$_{u}$*ser*$_{c}$*ontext*‘, ‘*clear*$_{u}$*ser*$_{c}$*ontext*‘)*enforcethateveryconnectionhasanex*

- The audit log, populated via triggers, preserves immutable records that would satisfy PCI-style investigations.

Implementing RLS forced me to think like an attacker. I validated policies by attempting to bypass them with crafted SQL and confirmed that unauthorized rows remain inaccessible.

7 Operational Readiness

The project extended beyond schema design:

- I scripted deterministic CSV imports with “ and documented absolute vs. relative path usage.
- The dataset README explains provenance so future changes maintain realism.
- I rehearsed restore scenarios with ‘pg$_{d}$*ump*‘/‘*pg*$_{r}$*estore*‘*and*‘*pg*$_{b}$*asebackup*‘, *documentinglessonsint*

- Query catalogues in ‘sql/queries.sql‘ double as automated tests—running them validates that constraints and triggers still behave as expected.

8 Key Lessons

- **Iterative modelling:** Several table designs changed mid-project after query requirements revealed missing relationships or denormalization opportunities.
- **Explain plans matter:** Using 'EXPLAIN ANALYZE' early prevented slow queries later, especially around the order history reports.
- **Documentation is a deliverable:** Writing the README and inline comments clarified intent and surfaced edge cases I had overlooked.
- **Automation pays off:** After building refresh and backup routines, I spent less time on manual chores and more time on analysis.

9 Future Enhancements

- Introduce PostgreSQL full-text search for product discovery.
- Add currency conversion tables to support international storefronts.
- Publish change events to a message queue so external services can react to order updates.
- Implement anomaly detection on payment transactions to flag potential fraud.
- Benchmark partitioning strategies for large historical tables (orders, audit_log).

10 Conclusion

Building this e-commerce database demanded the full spectrum of skills expected from a database engineer: data modelling, SQL mastery, automation, security, and operational awareness. The experience confirmed that delivering a database is as much about disciplined process as it is about schema design. I now feel equipped to contribute to real-world data platforms and to continue exploring advanced PostgreSQL capabilities.