# Hospital Management System: Advanced Database Design and Administration Reflection

FAIAN

Database Management Systems

Option 3: Coding-Heavy DBA / DevOps Project

October 9, 2025

**Abstract**

This reflection summarizes the design, implementation, and operational lessons learned while building a production-grade Hospital Management System (HMS) database on PostgreSQL. The project covers ten normalized tables, realistic healthcare datasets, analytical SQL, Row-Level Security (RLS), automated business rules, and DevOps practices such as scripted backups. I explain the motivations behind major design choices, discuss technical challenges that required deeper investigation, and outline how this work strengthened my preparation for database engineering roles.

## 1 Introduction

Healthcare data demands high integrity, privacy, and reliability. I selected the HMS domain to explore these requirements within the scope of Option 3, which emphasizes hands-on database engineering. My aim was to produce a database that could plausibly support daily hospital operations while remaining understandable to future maintainers. Throughout the project I balanced academic exploration with production realities: realistic sample data, auditable changes, and documented runbooks.

## 2 Project Objectives

- Model core hospital operations in Third Normal Form (3NF) with clear referential integrity.
- Demonstrate advanced SQL patterns, including window functions, recursive CTEs, and analytical queries.
- Implement fine-grained security via RLS, role-based access control, and audit logging.
- Automate repetitive tasks through triggers, stored procedures, and scripted maintenance.
- Provide comprehensive documentation for deployment, operations, and future enhancements.

# 3 Schema Architecture

The schema consists of ten interrelated tables grouped into clinical, operational, and governance layers. Clinical tables capture departments, doctors, patients, appointments, admissions, prescriptions, and medical tests. Operational tables track billing and staff assignments, while the governance layer provides an immutable `audit_log`.

## 3.1 Normalization Strategy

Every table satisfies 3NF. For example, `billing` references either an appointment or an admission, but never both, enforced through a CHECK constraint. Shared attributes such as contact information are centralised to prevent update anomalies. Enumerated status fields (for example, appointment status) are controlled through domain tables to limit invalid values.

## 3.2 Data Realism

I designed 200+ sample records that align with Boston-area hospital demographics. Insurance providers, medication dosages, and diagnostic codes follow realistic patterns so that analytical queries return plausible scenarios.

# 4 Advanced SQL and Analytics

Analytical requirements drove several design choices:

- Window functions generate rankings for physician performance dashboards and length-of-stay statistics.
- Recursive CTEs produce organisational charts and patient follow-up timelines.
- Materialized views summarise revenue and departmental throughput for fast dashboard refreshes.
- The `sql/queries.sql` catalog offers twenty curated examples with inline documentation to teach key techniques.

# 5 Security and Compliance

Protecting Protected Health Information (PHI) shaped the security implementation:

- Five roles (administrator, doctor, staff, patient, billing) have least-privilege grants.
- RLS policies ensure patients see only their records, while doctors are restricted to their roster.
- A trigger-driven audit log records every insert, update, and delete on sensitive tables, capturing the acting role, IP address, and timestamp.
- Masking functions obfuscate identifiers when data is exported to analytics sandboxes.

# 6 Automation and DevOps Practices

Seven triggers automate operational safeguards, such as preventing double-booked appointments and keeping timestamp columns accurate. PowerShell scripts manage nightly backups with retention rotation, while cron-ready Bash equivalents support Linux deployments. Documentation in `backup_restore_guide.md` records disaster-recovery drills, and runbooks outline verification steps after each restore.

The database can be deployed to on-premises PostgreSQL servers, managed PostgreSQL services (for example, Amazon RDS or Azure Database for PostgreSQL), or containerized environments. Configuration sections highlight parameters for WAL archiving, auto-vacuum tuning, and monitoring with `pg_stat_statements`.

# 7 Challenges and Resolutions

## 7.1 Trigger Ordering

Multiple BEFORE triggers on the `appointments` table initially executed in an unexpected order, causing timestamps to lag validation checks. I resolved this by renaming triggers with numeric prefixes to enforce deterministic sequencing:

```
CREATE TRIGGER t01_validate_appointment
    BEFORE INSERT OR UPDATE ON appointments
    FOR EACH ROW EXECUTE FUNCTION validate_appointment();

CREATE TRIGGER t02_stamp_appointment
    BEFORE INSERT OR UPDATE ON appointments
    FOR EACH ROW EXECUTE FUNCTION set_timestamp_columns();
```

## 7.2 Sequence Synchronisation

Bulk loading CSV data occasionally left sequences behind the actual maximum IDs. A post-load script now executes `SELECT setval(...)` statements for every table, guaranteeing predictable primary key generation for subsequent inserts.

## 7.3 Row-Level Security Testing

Writing 25 RLS policies introduced complexity when testing overlapping conditions. I created helper functions that temporarily assume each role, allowing repeatable regression tests that confirm policy coverage before deployment.

# 8 Skills Strengthened

## 8.1 Technical

1. Advanced SQL: window functions, recursive CTEs, query optimization.
2. Database security: RLS, RBAC, audit logging, and data masking.
3. Performance tuning: index analysis, execution plans, and statistics maintenance.
4. DevOps: scripted backups, restore rehearsals, and configuration management.

5. Cloud readiness: parameter planning for managed PostgreSQL services.

### 8.2 Professional

- Clear documentation that future DBAs can follow without direct coaching.
- Domain literacy in healthcare workflows and regulatory expectations.
- Issue tracking and prioritisation to manage a multi-file deliverable.
- Collaboration readiness through structured repository layout and coding standards.

## 9 Future Enhancements

1. Table partitioning for high-volume history tables such as `appointments` and `billing`.
2. Logical replication to feed downstream analytics warehouses with minimal latency.
3. Integration with a message queue so that application services receive change notifications.
4. Automated data quality dashboards that leverage 'pg$_stat$'$views and custom metrics$.
4. Expanded datasets, including nursing notes and radiology reports, with appropriate access controls.

## 10 Conclusion

The HMS project deepened my appreciation for treating databases as living systems rather than static schemas. Crafting realistic data, enforcing security, and planning for failure required disciplined engineering. PostgreSQL offered rich tooling for every challenge I encountered, from RLS and materialized views to WAL archiving. Most importantly, the project demonstrated that the combination of thoughtful design, accurate documentation, and automation can make even complex healthcare data manageable. I am confident that the skills practiced here translate directly to production database engineering roles.