FINAL PROJECT REPORT DOCUMENT

HOSPITAL MANAGEMENT SYSTEM (HMS)

Submitted for the Term Project of

Course COMP 3030: Databases and Database Systems

by Group 4

Nguyễn Thị Phương Thảo - V202401781

Đỗ Thị Hải Bình - V202401548

Lê Thảo Vy - V202401694

Under the supervision of

Professor Le Duy Dung

TA Ta Quang Hieu

Link Github: [*https://github.com/haibinh-dt/VinUni-healthcare-database-project-skibidi.git*](https://github.com/haibinh-dt/VinUni-healthcare-database-project-skibidi.git)

Table of Contents

1. Database Design: Conceptual and Physical Architecture

1.1 Functional Requirements

1. Manage patient registration and profile information, including personal details and associated clinical attachments.
2. Manage doctor profiles, departmental assignments, and availability schedules.
3. Schedule appointments based on doctor availability and time slots, with automatic conflict detection to prevent double-booking.
4. Track appointment status changes throughout the scheduling lifecycle with full status history.
5. Record clinical visits and consultations, including diagnoses, clinical notes, and uploaded medical documents.
6. Manage diagnosis records and associate multiple diagnoses with a single visit where applicable.
7. Create and manage prescriptions linked to clinical visits.
8. Manage pharmacy inventory by medication batches, including stock quantities, expiry dates, and supplier information.
9. Generate and manage supplier invoices for pharmacy procurement costs.
10. Automatically validate and update pharmacy inventory levels based on prescription records using database-level triggers.
11. Record pharmacy stock movements to support inventory auditing and traceability.
12. Define and manage medical services provided during clinical visits.
13. Associate medical services with visits and generate itemized invoices for services and prescribed medications.
14. Process patient payments and record corresponding financial transactions for income and expense tracking.
15. Provide analytical and statistical reports for hospital operations, pharmacy usage, and financial performance.
16. Maintain audit logs for sensitive operations, including changes to medical records, pharmacy inventory, billing data, and user-related activities, using database-level triggers.
17. Track user login history, including timestamps and access attempts, to support security monitoring and incident investigation.
18. Enforce role-based authentication and authorization for system users, including administrators, medical staff, and pharmacists.
19. Deliver system notifications for key events such as appointment reminders, low-stock alerts, and billing updates.

1.2 Non-functional Requirements

1.2.1 Security

- Ensure secure authentication with hashed credentials and protection of sensitive medical and financial data.
- Enforce role-based access control (RBAC) at both the database and application layers.
- Prevent unauthorized access and injection attacks through controlled database access and prepared statements.

1.2.2 Data Integrity & Consistency

- Maintain data integrity through the use of primary keys, foreign keys, constraints, cascading rules, and transactional control.
- Ensure consistent system behavior by enforcing business rules at the database level using procedures and triggers.

1.2.3 Auditability & Traceability

- Provide comprehensive auditability of system usage and data modifications through database-level logging mechanisms.
- Track user activities and critical state changes to support accountability, monitoring, and incident investigation.

1.2.4 Performance & Reliability

- Optimize system performance using workload-driven indexing strategies and efficient query design on frequently accessed data.
- Support stable and predictable performance under concurrent outpatient hospital workloads.

1.2.5 Scalability & Maintainability

- Enable scalability and modularity in database design to accommodate future expansion of system features.
- Maintain a clean, well-structured, and maintainable codebase with clear documentation and consistent design patterns.

1.2.6 Backup & Recovery

- Support database-level backup and recovery mechanisms to prevent data loss and ensure system reliability.

1.2.7 Usability

- Deliver a responsive and user-friendly web interface tailored to the needs of all supported user roles.

1.3 Entity-Relationship Diagram (ERD)

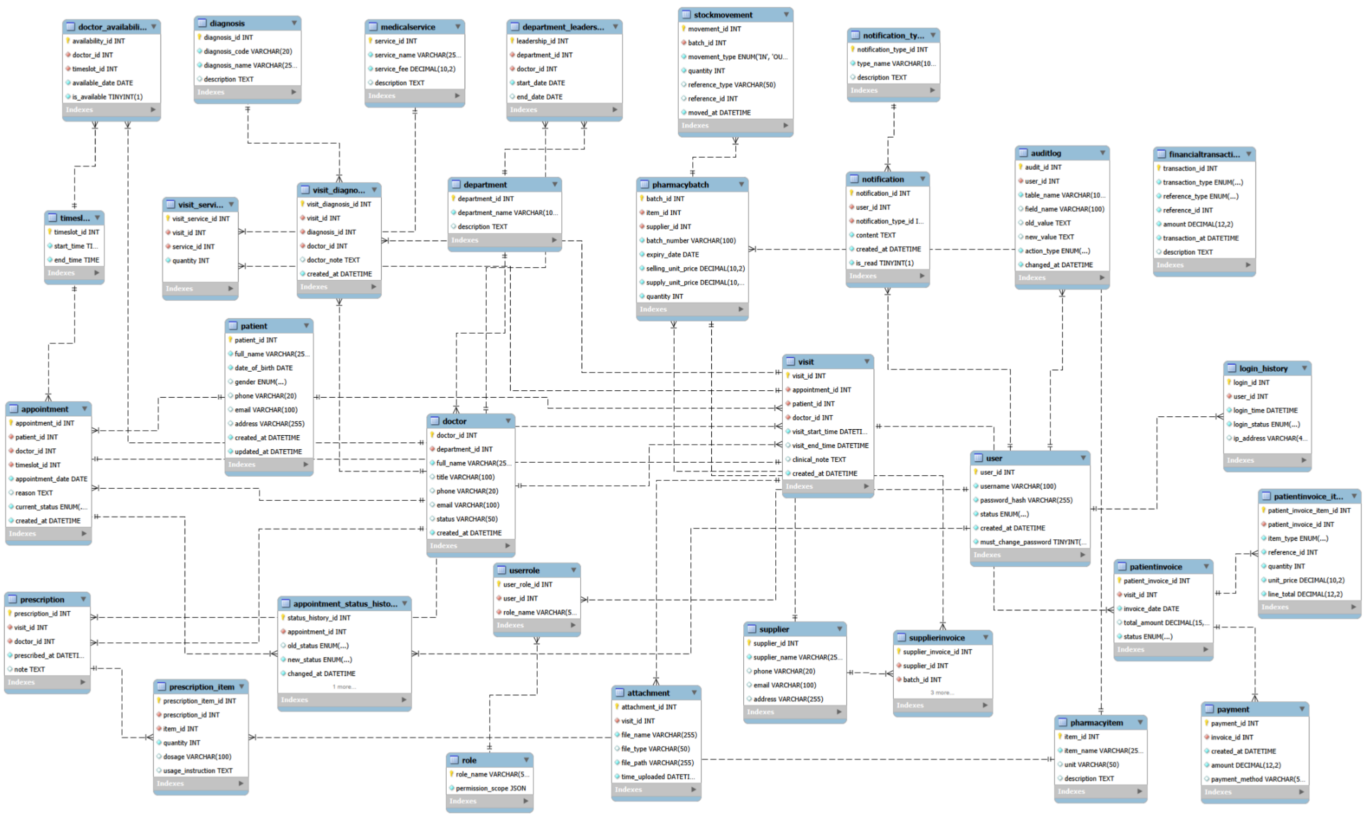

Figure 1. Entity-Relationship Diagram

1.4 Database Normalization (Up to 3NF)

The database schema of the Hospital Management System is designed in accordance with the principles of database normalization to minimize redundancy, prevent update anomalies, and ensure data integrity. All relations are normalized up to **Third Normal Form (3NF)** through a structured and systematic process.

1.4.1 Normalization First Normal Form(1NF)

A relation is in 1NF if all attributes contain atomic values, and there are no repeating rows.

In the Hospital Management System database design:

- All tables use **single-valued, atomic attributes**
- No table contains nested or repeating fields
- Each table has a **clearly defined primary key** to ensure entity uniqueness

Multivalued and many-to-many relationships are resolved through **separate associative tables**, including:

- Visit_Diagnosis
- Visit_Service
- PatientInvoice_Item
- UserRole

  This decomposition eliminates repeating groups and ensures that each table represents a single logical entity or relationship, and establishes a solid foundation for achieving higher levels of database normalization.

1.4.2. Normalization Second Normal From(2NF)

A relation is in 2NF if it is in 1NF and it has no partial dependencies, meaning all non-key attributes are fully dependent on the entire primary key.

- The database design eliminates partial dependencies across all tables.
- Many-to-many relationships, such as *Visit_Diagnosis*, *Visit_Service*, and *Prescription_Item*, are implemented using associative tables with surrogate primary keys.
  In these tables, all non-key attributes depend on the specific relationship instance.
- Core entity tables, including *Patient*, *Doctor*, and *MedicalService*, use single-attribute primary keys to ensure 2NF compliance.

1.4.3 Normalization Third Normal Form(3NF)

A relation is in 3NF if it is in 2NF and contains no transitive dependencies, meaning non-key attributes do not depend on other non-key attributes.

- Descriptive attributes are stored only in their respective entity tables.
  E.g., Department information is stored in Department, not duplicated in Doctor.
- Status history, audit logs, and login tracking are stored in separate tables (Appointment_Status_History, AuditLog, Login_History) to avoid embedding derived or historical data in core entities.
- Financial records distinguish between business events (invoices, payments) and accounting entries (FinancialTransaction), preventing transitive dependencies between financial attributes.

- Role permissions are centralized in the Role table using a JSON-based permission scope, avoiding redundant permission definitions across users.

1.5 Physical Database Schema Design

The physical database schema of the Hospital Management System is implemented in MySQL and consists of 32 normalized tables organized into functional modules, including clinical workflows, scheduling, pharmacy operations, billing and finance, security, auditing, and notifications. All tables use the InnoDB storage engine to support transactions, enforce foreign key relationships, and ensure data reliability.

1.5.1 Tables

The database consists of a total of **32 tables**, which are grouped according to their functional responsibilities to improve clarity, maintainability, and scalability. Each group supports a specific aspect of the system's operations while maintaining strong normalization and clear relationships between entities.

Category Table	Tables included	Functional Description	Relationship Description
1. Core Clinical Workflow Tables	**Patient, Doctor, Department, Appointment, Visit, Diagnosis, Visit_Diagnosis, Visit_Service, Attachment.**	These tables support the core clinical processes of the system, including patient registration, doctor and department management, appointment scheduling, visit recording, and clinical diagnosis documentation. They store essential clinical and operational data required for patient care delivery.	The core clinical workflow is centered on the Patient, Doctor, Department, Appointment, Visit, and Diagnosis tables, which represent the primary clinical entities. Supporting tables such as Visit_Diagnosis and Visit_Service are used to model many-to-many relationships between visits, diagnoses, and medical services, while the Attachment table stores clinical documents associated with each visit.
2. Scheduling and Availability Tables	**TimeSlot, Doctor_Availability, Appointment_Status_History**	These tables manage appointment scheduling, time-based availability, and status tracking. They enable efficient allocation of appointment slots, prevent scheduling conflicts, and maintain a history of appointment status changes	Doctor_Availability links Doctor and TimeSlot to define available periods. Appointment references TimeSlot for scheduling. Appointment_Status_History maintains a one-to-many relationship with Appointment to record all status transitions over time for auditing purposes.
3. Pharmacy and Inventory Tables	**PharmacyItem, PharmacyBatch, Prescription, Prescription_Item, StockMovement,**	These tables manage pharmacy inventory and prescription processing. The design supports batch-level stock tracking, expiry date monitoring,	PharmacyItem has a one-to-many relationship with PharmacyBatch to track stock batches. Prescription is linked to Visit and Patient, while

	Supplier, SupplierInvoice	prescription fulfillment, automated stock adjustments, and supplier procurement workflows.	Prescription_Item models the many-to-many relationship between Prescription and PharmacyItem. StockMovement records inventory changes per PharmacyBatch. Supplier and SupplierInvoice are linked to manage procurement transactions.
4. Billing and Financial Tables	**MedicalService, PatientInvoice, PatientInvoice_Item, Payment, FinancialTransaction**	These tables handle billing and financial management, separating clinical services from invoicing and accounting records. This structure ensures accurate billing, payment tracking, and financial reporting while maintaining normalization	MedicalService is linked to Visit through Visit_Service. PatientInvoice has a one-to-many relationship with PatientInvoice_Item to detail billed services. Payment records are associated with PatientInvoice, while FinancialTransaction provides an accounting-level record linked to payments and invoices.
5. User, Security, and Audit Tables	**User, Role, UserRole, AuditLog, Login_History**	These tables implement authentication, authorization, and system auditing. They control user access through roles and maintain logs of system usage and sensitive operations for security and compliance purposes.	User and Role are connected through the associative table UserRole to support many-to-many role assignments. AuditLog records actions performed by users on system entities. Login_History maintains a one-to-many relationship with User to track authentication attempts and sessions.
6. Notification System Tables	**Notification, Notification_Type**	These tables manage system notifications and alerts, enabling the delivery of reminders, warnings, and status updates to system users.	Notification_Type defines the classification of notifications. Notification references Notification_Type and User to associate each message with its type and intended recipient.

Table 1. Table database description

1.5.2 Primary & Foreign Keys

Each table in the schema is defined with a **primary key** to uniquely identify records. Most tables use **surrogate primary keys** (e.g., *patient_id*, *appointment_id*, *visit_id*) to simplify indexing and improve join performance, while lookup tables such as **Role** use meaningful natural keys where appropriate.

Foreign keys are extensively used to enforce **referential integrity** across system modules and to accurately represent relationships between entities. Examples of key relationships include:

- *Appointment.patient_id* referencing *Patient.patient_id*
- *Appointment.doctor_id* referencing *Doctor.doctor_id*
- *Visit.appointment_id* referencing *Appointment.appointment_id*
- *Prescription.visit_id* referencing *Visit.visit_id*
- *PatientInvoice.visit_id* referencing *Visit.visit_id*

Associative tables such as Visit_Diagnosis, Prescription_Item, Visit_Service, and UserRole rely on foreign keys to model many-to-many relationships while maintaining normalization and data consistency.

1.5.3 Constraints and Integrity Rules

To maintain data integrity and enforce business rules, the physical database schema implements a comprehensive set of constraints and integrity mechanisms defined at both the database and application-support levels.

- **NOT NULL constraints** are applied to mandatory attributes such as primary and foreign keys, entity names, dates, quantities, and status fields to ensure that essential data is always present and incomplete records are prevented.
- **UNIQUE constraints** are enforced on attributes that require uniqueness across the system, including user credentials (e.g., username), reference codes (such as diagnosis codes), and key identifiers, thereby preventing duplicate or conflicting records.
- **ENUM and CHECK constraint**s are used to restrict attribute values for controlled domains, including appointment status, payment status, transaction type, user status, and inventory movement type. These constraints ensure that only valid, predefined values can be stored, reinforcing business logic consistency at the database level.
- **REFRENTAIL INTEGRITY constraints** are implemented through foreign key relationships between parent and child tables. These constraints ensure valid associations between entities such as patients, visits, prescriptions, invoices, and inventory records, and prevent the creation of orphan records. Cascading rules are applied selectively to maintain data consistency during updates and deletions where appropriate.
- **DEFAULT values** are defined for commonly used attributes, including timestamps, boolean flags, and status fields, to ensure consistent data initialization and reduce the likelihood of missing or inconsistent values during record creation.
- **NULLABLE constraints** are applied selectively to optional attributes such as notes, descriptions, end dates, and supporting metadata. This approach provides flexibility for optional information while preserving strict validation for critical data elements.

In addition to declarative constraints, **database triggers** and **scheduled events** are used to enforce complex integrity rules that cannot be fully expressed through standard constraints alone. These mechanisms support automated stock updates, audit logging, and expiry monitoring, further strengthening data accuracy and operational reliability.

Together, these constraints and integrity rules ensure that the physical schema is robust, secure, and consistent, providing a reliable foundation for transactional processing, performance optimization, and seamless integration with the web application layer.

2. Database Implementation and Core Objects

2.1 Database Views

a. Overview:

The table below provides an overview of the database views implemented in the Hospital Management System (HMS), organized by functional category, view name, purpose, and authorized user roles. Each view is designed to support a specific operational or analytical requirement, including appointment scheduling, clinical record access, pharmacy inventory monitoring, billing and financial analysis, and administrative auditing.

#	Category	View Name	Purpose	Target Role
1	Scheduling	v_reception_daily_queue	List of all arrivals for today with status color-coding.	RECEPTIONIST, DOCTOR
2		v_doc_availability_browser	Provides a real-time overview of doctor availability and open time slots.	RECEPTIONIST
3		v_appointment_details	Full context of an appointment (Doctor + Patient + Dept).	RECEPTIONIST, ADMIN
4		v_doctor_schedule_detail	Detailed daily agenda for a specific doctor.	DOCTOR
5		v_doctor_schedule_summary	Counts of Completed/Cancelled/Confirmed per doctor.	ADMIN, RECEPTIONIST
6	Clinical Records	v_patient_master_record	The "Patient Book" with last visit and primary diagnosis.	ALL ROLES
7		v_visit_clinical_summary	Overview of a visit: How many diagnoses and services.	DOCTOR, ADMIN
8		v_visit_diagnoses_detail	Exact details and notes for every diagnosis in a visit.	DOCTOR
9		v_patient_medical_history	The full EMR timeline (Notes + Meds + Doctors).	DOCTOR
10	Pharmacy	v_prescription_for_pharmacy	Prescription details (Meds + Dosage + Instructions).	PHARMACIST, DOCTOR
11		v_inventory_batch_status	Current stock levels per batch with "Expired" flag.	PHARMACIST
12		v_pharmacy_stock_alerts	Low stock and expiry warnings (for Dashboard).	PHARMACIST, ADMIN
13	Finance	v_invoice_payment_tracker	Real-time balance calculations (Paid vs. Remaining).	FINANCE, RECEPTIONIST
14		v_invoice_payment_summary	Displays invoice information, showing total billed amounts and total payments received.	FINANCE

15		v_supplier_procurement	Supplier costs vs. Selling prices (Profit margin).	FINANCE, PHARMACIST
16		v_supplier_invoice_details	Presents detailed supplier invoice data.	FINANCE
17		v_financial_cash_flow	Daily Income (Patient) vs Expenses (Suppliers).	FINANCE, ADMIN
18	Admin & Security	v_audit_readable_log	Human-readable log of who changed what.	ADMIN
19		v_appointment_status_audit	Tracks the historical changes of appointment statuses (e.g., confirmed, cancelled, completed)	ADMIN
20		v_user_role_directory	Provides a centralized directory of system users	ADMIN
21		v_user_security_activity	Login history, IPs, and failed attempt counts.	ADMIN
22		v_user_auth	Supports user authentication	ADMIN

Table 2. Views overview

b. View Details:

The table below provides a more detailed description of selected database views. While the previous table provides an overview of all views, this table focuses on explaining the specific function and usage context of individual views in detail.

#	**Categories**	**View Name**	**Description**
1	Scheduling	**v_reception_daily_queue**	Today's arrivals, check-in status, and UI status colors.
2		**v_doc_availability_browser**	Real-time lookup of bookable slots for the booking page.
3		**v_appointment_details**	Full context of an appointment record (Patient + Dr + Slot).
4		**v_doctor_schedule_detail**	Detailed daily agenda for a specific doctor.
5		**v_doctor_schedule_summary**	Summary counts (Completed/Cancelled) for daily stats.
6	Clinical Records	**v_patient_master_record**	Central directory for searching/listing all patients.

7		**v_visit_clinical_summary**	Overview of visit volume (count of diagnoses/services).
8		**v_visit_diagnoses_detail**	Specific ICD-10 codes and notes for a patient's visit.
9		**v_patient_medical_history**	The "Master EMR": Full timeline of past meds and notes.
10	Pharmacy	**v_prescription_for_pharmacy**	List of meds, dosages, and instructions for dispensing.
11		**v_inventory_batch_status**	Detailed batch-level stock list with expiry flags.
12		**v_pharmacy_stock_alerts**	Dashboard alerts for items near expiry or low quantity.
13	Finance	**v_invoice_payment_tracker**	Real-time balance calculations for patient billing.
14		**v_invoice_payment_summary**	Grouped billing details showing totals vs. paid amounts.
15		**v_supplier_procurement**	Profit margin analysis (Supply Cost vs. Selling Price).
16		**v_supplier_invoice_details**	Deep dive into supplier billing and linked inventory.
17		**v_financial_cash_flow**	Combined ledger of all income and expenses.
18		**v_audit_readable_log**	Human-readable trail of all data changes.
19	Admin	**v_appointment_status_audit**	Targeted history of status changes for appointments.
20		**v_user_security_activity**	Login history, IP tracking, and failed login counts.
21		**v_user_role_directory**	Master list of system users and their assigned roles.
22		**v_user_auth**	Provides user credential and account status data required for authentication and access control.

Table 3. Views Descriptions

Database views are implemented to abstract complex joins, improve query readability, and provide role-specific access to frequently used data. These views are primarily used by the web application to support operational workflows, dashboards, and reporting, while preventing direct access to underlying base tables.

2.2 Stored Procedures

The implementation of 27 stored procedures is appropriate for a Hospital Management System, as the system handles multiple sensitive and interrelated operations across security, scheduling, clinical care, pharmacy, billing, and administration. By encapsulating these operations within stored procedures, the database enforces consistent business rules, improves security by restricting direct table access, and ensures data integrity across complex workflows. This approach also enhances maintainability and scalability by centralizing core logic at the database level while supporting efficient integration with the web application.

#	Section	Stored Procedure	Description	Authorized Users
1	User & Security	**sp_create_user_with_default_password**	Creates system users with hashed passwords and role assignment. Prevents direct INSERT into User table.	ADMIN
2		**sp_change_password**	Allow users to change their own password securely.	ALL
3		**sp_update_user_role**	Changes user roles with authorization checks to prevent privilege escalation.	ADMIN
4		**sp_deactivate_user**	Soft-deactivates users while preserving audit and historical records.	ADMIN
5		**sp_verify_login**	Verify user credentials.	SYSTEM
6	Front desk & Scheduling	**sp_register_patient**	Registers new patients with duplicate detection based on phone/email.	RECEPTIONIST
7		**sp_create_doctor_availability**	Publishes doctor availability slots while preventing overlaps and invalid times.	DOCTOR
8		**sp_book_appointment**	Books appointments with conflict prevention and availability updates in a transaction.	RECEPTIONIST
9		**sp_confirm_appointment**	Confirms a scheduled appointment and updates its status to reflect patient confirmation.	RECEPTIONIST
10		**sp_cancel_appointment**	Cancels appointments and restores doctor availability automatically.	RECEPTIONIST

11	Clinical Operations	**sp_start_visit**	Transitions a confirmed appointment into an active visit, enforcing state rules.	DOCTOR
12		**sp_add_diagnosis**	Adds medical diagnoses linked to visits with role verification and audit logging.	DOCTOR
13		**sp_add_visit_service**	Adds a medical service or procedure performed during a patient visit and links it to the visit record.	DOCTOR
14		**sp_add_visit_attachment**	Attaches clinical documents or files (e.g., reports, images) to a patient visit.	DOCTOR
15		**sp_create_prescription**	Creates prescriptions linked to a visit, enforcing the one-prescription-per-visit rule.	DOCTOR
16		**sp_add_prescription_item**	Adds medication items, dosage, and instructions to an existing prescription.	DOCTOR
17		**sp_end_visit**	Marks a patient visit as completed and finalizes associated clinical records.	DOCTOR
18	Pharmacy & Inventory	**sp_add_pharmacy_batch**	Adds pharmacy inventory batches with expiry date and quantity validation.	PHARMACIST
19		**sp_create_supplier_invoice**	Creates a supplier invoice linked to pharmacy procurement and inventory records.	ADMIN
20		**sp_dispense_medication**	Records the dispensing of prescribed medication and updates pharmacy stock levels accordingly.	PHARMACIST
21	Finance & System	**sp_generate_patient_invoice**	Generates invoices automatically from visits, services, and prescriptions.	FINANCE
22		**sp_record_payment**	Records payments and updates invoice status consistently.	FINANCE

23		**sp_create_department**	Adds new departments with duplicate-name prevention and validation.	ADMIN
24		**sp_assign_department_lead**	Assigns or updates the lead doctor for a specific department with authorization checks.	ADMIN
25		**sp_create_notification**	Creates system notifications targeted to users based on system events.	SYSTEM, ADMIN
26		**sp_get_patient_emr**	Retrieves a patient's complete electronic medical record (EMR) for clinical review	DOCTOR
27		**sp_log_audit_event**	Centralized audit logging for security, compliance, and debugging.	SYSTEM

Table 4. Stored Procedures Description

2.3 Database Triggers

Database triggers are used in the Hospital Management System (HMS) to automatically enforce critical business rules, data validation, auditing, and system consistency at the database level. Triggers are executed in response to specific data modification events (INSERT, UPDATE, or DELETE), ensuring that important actions occur reliably regardless of how the data is modified by the application.

In this system, triggers serve four primary purposes. **Operational triggers** maintain system consistency by synchronizing related data, such as restoring doctor availability when appointments are cancelled and automatically updating pharmacy batch quantities after stock movements. **Financial triggers** ensure accurate accounting by generating financial transaction records and updating invoice statuses when payments or invoice status changes occur. **Validation triggers** prevent invalid data entry, such as blocking the insertion of pharmacy batches with expired dates. **Audit triggers** provide a comprehensive and human-readable audit trail by recording changes to sensitive and critical tables, supporting accountability, traceability, and compliance requirements.

The use of triggers complements declarative constraints and stored procedures by handling rules that cannot be fully enforced through static constraints alone. By embedding these rules directly in the database, the HMS ensures consistent behavior, improved data integrity, and reduced reliance on application-level logic for enforcing critical system rules.

#	**Trigger Name**	**Entity Group**	**Table**	**Timing**	**Event**	**Trigger Purpose**
1	**trg_appointment_after_update**	**Operational**	**Appointment**	AFTER	UPDATE	1. Audit status change. 2. Record

						status history. 3. Restore availability if 'CANCELLED'.
2	**trg_stock_movement_after_insert**	**Operational**	**StockMovement**	AFTER	INSERT	1. Audit movement. 2. Auto-sync current batch quantity.
3	**trg_pharmacy_batch_after_update**	**Operational**	**Pharmacy Batch**	AFTER	UPDATE	1. Audit price/qty changes. 2. Generate Low-Stock & Expiry alerts.
4	**trg_patient_invoice_after_update**	**Financial**	**PatientInvoice**	AFTER	UPDATE	1. Audit invoice data. 2. Log 'INCOME' in FinancialTransaction when status = 'PAID'.
5	**trg_supplier_invoice_after_update**	**Financial**	**SupplierInvoice**	AFTER	UPDATE	Log 'EXPENSE' in FinancialTransaction when status = 'PAID'.
6	**trg_payment_after_insert**	**Financial**	**Payment**	AFTER	INSERT	1. Audit payment. 2. Auto-update Invoice status to 'PAID' if balance is 0.
7	**check_expiry_before_insert**	**Validation**	**Pharmacy Batch**	BEFORE	INSERT	Prevents insertion of pharmacy batches with expiry dates earlier than or equal to the current date.
8	**trg_audit_user_update, trg_audit_patient_update, trg_audit_doctor_update, trg_audit_userrol**	**Audit Only**	**11 Other Tables**	AFTER	I/U/D	One trigger per table (User, UserRole, Doctor, Department_Leadership, MedicalService, Patient, Visit, Prescription,

	e_insert, trg_audit_deptlead_insert, trg_audit_medservice_update, trg_audit_visit_insert, trg_audit_prescription_insert, trg_audit_dept_update, trg_audit_role_update, trg_audit_supplier_update					Department, Role, Supplier).

Table 5. Database Triggers

3. Performance Tuning

Performance Evaluation Methodology

Performance evaluation in this chapter was conducted using synthetically generated data designed to approximate realistic hospital workloads. Because data generation involves randomized values, exact execution times and row distributions may vary between runs. Therefore, evaluation focuses on execution plans, access paths, index usage, and partition pruning behavior rather than absolute timing measurements. Queries were executed multiple times to mitigate caching effects, and results were interpreted qualitatively by comparing structural changes in query execution before and after optimization. This methodology provides a robust assessment of scalability and efficiency under realistic operating conditions.

3.1 Indexing Strategy

The Hospital Management System database adopts a workload-driven indexing strategy, prioritizing indexes that support realistic access patterns rather than indiscriminately indexing all attributes. This approach balances query performance with write efficiency, particularly for a system that includes both transactional and audit-heavy tables.

3.1.1 Primary Key Indexing

Each table in the database uses a single-column primary key, typically an auto-increment integer. Primary keys are automatically indexed by MySQL and serve to:

- Guarantee entity identity
- Simplify join operations across related tables

- Keep foreign key references compact and efficient

This design choice ensures consistent and predictable performance across the schema.

3.1.2 Foreign Key Indexing

Columns frequently used in join operations, such as `patient_id`, `doctor_id`, `visit_id`, and `user_id`, are indexed either explicitly or implicitly through primary or unique constraints. This optimizes joins across core workflows, including:

- Appointments → Visits → Invoices
- Visits → Prescriptions → Pharmacy Batches
- Users → Login History and Audit Logs
- Doctors → Schedules and Availability

As a result, relational queries remain performant as the dataset grows.

3.1.3 Composite Indexes for Time-Based Queries

Several tables include composite indexes that combine an entity identifier with a date or timestamp attribute, such as:

- `(user_id, login_time)`
- `(appointment_id, changed_at)`
- `(batch_id, moved_at)`

These indexes are designed to optimize common query patterns, including:

- Retrieving recent activity for a specific user or entity
- Generating chronological histories (audit logs, status changes, stock movements)

By aligning the index structure with query predicates, the database avoids unnecessary full table scans.

3.1.4 Uniqueness Constraints as Logical Indexes

UNIQUE constraints are intentionally applied where business logic requires natural uniqueness, for example:

- One appointment per doctor–time slot–date combination
- One invoice per clinical visit

In addition to enforcing data correctness, these constraints function as highly selective indexes that further improve query performance.

3.1.5 Avoiding Over-Indexing

Tables that are primarily write-heavy or rarely queried directly, such as historical or logging tables, are indexed conservatively. Indexes are added only on columns that support expected access patterns, reducing overhead during INSERT and UPDATE operations.

The effectiveness of indexing depends on both query structure and predicate selectivity. Queries that filter on indexed attributes consistently enable the optimizer to replace full table scans with index-based access paths such as index range scans or ref lookups. In cases where query ordering aligns with index column order, the optimizer can additionally avoid explicit sorting operations by performing backward or forward index scans. Conversely, when query predicates or ordering conditions do not fully match index definitions, MySQL may still utilize an index for filtering but require additional operations such as filesort. These behaviors reflect the intended trade-off between index design flexibility and optimal access path selection.

3.2 Partitioning Strategy

The partitioning strategy complements indexing and query optimization by reducing the volume of data scanned during time-based queries, maintaining consistent performance as historical data accumulates, and simplifying data lifecycle management for audit and reporting tables. Together, these techniques form a cohesive performance tuning framework tailored to the operational requirements of the hospital management system.

3.2.1 Rationale for Partitioning

Partitioning was selectively applied to tables that satisfy the following criteria:

- Large and continuously growing data volume
- Predominantly time-based access patterns
- Minimal update operations on historical records
- Frequent use in reporting or audit queries

Based on these criteria, partitioning was applied to audit log tables, appointment and visit reporting tables, and financial transaction tables. Smaller, highly transactional tables were intentionally excluded to avoid unnecessary complexity.

3.2.2 Partitioning Design

Range partitioning was implemented using monthly partitions based on date or timestamp attributes. Each partition represents a calendar month, with a `MAXVALUE` partition used to capture future data. Partition keys were chosen to match query filtering predicates, enabling the query optimizer to perform partition pruning automatically for date-restricted queries.

3.2.3 Impact of Partitioning on Performance

Aspect	**Without Partitioning**	**With Partitioning**
Rows scanned	Entire table	Relevant partitions only
Query scalability	Degrades over time	Stable

Historical cleanup	Row-level deletes	Partition drop
Maintenance overhead	High	Reduced

Table 6. Comparison of Query Execution Behavior Before and After Indexing

3.2.4 Partitioning Trade-offs

Partition pruning effectiveness depends on the selectivity of date-based query predicates. Queries restricted to a single time interval, such as a specific month, consistently access only the corresponding partition, thereby minimizing the amount of data scanned. Queries spanning broader time ranges may access multiple partitions; however, they still avoid scanning the entire table, resulting in substantially reduced I/O compared to non-partitioned designs. While partitioning improves read performance and maintainability for large, time-series tables, it also introduces additional design considerations, including increased schema complexity, partition key constraints in queries, and limited benefit for small or frequently updated tables. For these reasons, partitioning was applied conservatively and only where justified by workload characteristics.

3.3 Query Optimization

Query optimization focused on ensuring that SQL statements were written to fully leverage the indexing strategy described in Section 3.1 while minimizing unnecessary computation, I/O overhead, and memory usage.

3.3.1 Optimization Objectives

The primary objectives of query optimization were to:

- Maximize index usage and avoid full table scans
- Reduce the volume of data processed per query
- Improve join efficiency across relational workflows
- Ensure predictable performance under increasing data volume

3.2.2 Query Design Principles

Several design principles were consistently applied throughout the system:

- **Explicit column selection**
  Queries avoid `SELECT *` and retrieve only required attributes, reducing disk I/O and network transfer costs.
- **Predicate simplification**
  Filtering conditions are written in a form that allows the query optimizer to utilize indexes effectively, particularly for date and timestamp columns.
- **Early filtering**
  `WHERE` clause predicates are applied as early as possible to reduce the number of rows participating in join operations.

- **Avoidance of functions on indexed columns**
  Expressions such as `YEAR(date_column)` are avoided in favor of range-based filtering to preserve index usability.

3.3.3 Example Query Optimization

Non-optimized query

```
SELECT *

FROM Appointment a

JOIN Patient p ON a.patient_id = p.patient_id

WHERE YEAR(a.appointment_date) = 2025;
```

Optimized query

```
SELECT a.appointment_id, a.appointment_date, p.full_name

FROM Appointment a

JOIN Patient p ON a.patient_id = p.patient_id

WHERE a.appointment_date BETWEEN '2025-01-01' AND '2025-12-31';
```

Analysis

The optimized query enables index range scans on `appointment_date`. Removing `SELECT *` reduces unnecessary data retrieval, and execution plan analysis (`EXPLAIN`) confirms that full table scans are replaced by indexed access paths.

3.3.4 Use of Views and Stored Procedures

Frequently executed or logically complex queries are encapsulated within database views and stored procedures. This improves query reusability, reduces the risk of inconsistent query logic, and results in cleaner application-level code. Query plans were validated using `EXPLAIN` to ensure that views did not introduce hidden performance penalties or inhibit index usage.

3.3.5 Query Optimization Outcome

As a result of these optimizations:

- Join-heavy queries execute more efficiently
- Time-based queries scale predictably with data growth
- Query performance remains stable under concurrent access

3.4 Performance Testing Results

Performance testing was conducted to empirically evaluate the effectiveness of indexing, query optimization, and partitioning strategies under realistic workloads.

3.4.1 Testing Setup

The testing process involved:

- Generating synthetic but realistic datasets for transactional and reporting tables
- Executing representative queries before and after optimization
- Observing execution behavior using execution plans and response consistency
- Comparing data access patterns rather than relying solely on raw execution time

Tested workloads included patient appointment lookups, doctor schedule retrieval, audit log history queries, and time-based reporting queries.

3.4.2 Observed Performance Improvements

Query Category	Before Optimization	After Optimization
Join operations	Frequent full table scans	Indexed joins
Time-range queries	Large table scans	Index and partition pruning
Reporting queries	Performance degradation	Stable response behavior
Write-heavy tables	Index overhead risk	Controlled indexing

Table 7. Impact of Query Optimization on Access Paths and Data Scanning

3.4.3 Analysis of Results

Indexed queries consistently reduced the number of examined rows, and execution plans confirmed effective usage of both composite and foreign key indexes. Partitioned tables demonstrated predictable query behavior even as historical data volume increased. Although exact execution times and row counts vary across runs due to randomized data generation, the structural behavior of the query optimizer remained consistent across tests.

3.4.4 Summary of Testing Findings

The performance testing results validate that the applied tuning strategies:

- Improve query efficiency
- Enhance system scalability
- Maintain acceptable write throughput
- Support long-term operational stability

3.5 Threats to Validity

Several factors may affect the interpretation of the performance evaluation results. First, synthetic data may not perfectly capture all characteristics of real-world hospital data distributions, such as skewed access patterns or burst workloads. Second, caching effects at the database and operating system level may influence execution behavior despite repeated query execution. Finally, performance analysis focused on execution plans and access paths rather than absolute timing values, which limits direct comparison with production environments. Nevertheless, the observed consistency in optimizer behavior and access strategies provides confidence that the tuning decisions improve scalability and efficiency under realistic conditions.

4. Security Configuration

4.1 Database Users and Roles

The HMS implements user and role management using application-defined database tables instead of direct database account privileges. User identities, roles, and permissions are modeled within the schema through the **User**, **Role**, and **UserRole** tables, providing a flexible and secure role-based access control mechanism.

Each system user is stored in the `User` table with mandatory and validated attributes, including a unique username, encrypted password hash, account status, and audit metadata. **NOT NULL and UNIQUE constraints** ensure that all users have valid identifiers and that duplicate accounts cannot be created. Sensitive authentication information is protected by storing passwords only in hashed form.

System roles are defined in the `Role` table and represent distinct authorization levels within the application, such as administrative, clinical, pharmacy, financial, and scheduling roles. The many-to-many relationship between users and roles is implemented using the `UserRole` junction table, allowing users to hold multiple roles when required. Referential integrity constraints ensure that role assignments always reference valid users and roles.

User-related operations are intentionally controlled through stored procedures rather than direct table manipulation. Procedures responsible for user creation, authentication, password updates, role assignment, and account status changes enforce business rules and authorization checks at the database level. This design prevents unauthorized modifications and reduces the risk of security breaches caused by direct data access.

In addition, database triggers monitor and audit changes to sensitive user-related data, such as account status updates, username changes, and role assignments. These triggers generate detailed audit records, ensuring accountability and traceability of all security-critical actions.

Overall, this security design ensures that user authentication and authorization are consistently enforced at the database layer, supporting secure role-based access control while maintaining strong data integrity and auditability across the system.

Role Name	Description	Key Responsibilities
ADMIN	System administrator with full management privileges.	User and role management, department configuration, financial oversight, audit review.
DOCTOR	Clinical user responsible for patient care.	Managing visits, diagnoses, prescriptions, and clinical documentation.
RECEPTIONIST	Front-desk user responsible for scheduling and coordination.	Appointment booking, confirmation, and patient check-in
PHARMACIST	User responsible for pharmacy operations.	Dispensing medications, managing inventory and batch records.
FINANCE	User responsible for billing and financial management.	Managing invoices, payments, supplier invoices, and financial reporting

Table 8. Application Users and Roles table

4.2 Role-Based Access Control (RBAC)

The Hospital Management System (HMS) implements **Role-Based Access Control (RBAC)** at the database design level to ensure that users can access only the data and operations required for their assigned responsibilities. RBAC is enforced through a combination of schema design, database views, stored procedures, and audit triggers, providing a secure and structured authorization model.

RBAC Implementation Overview:

RBAC in the HMS is achieved through the following mechanisms:

- **Roles definition**
  Roles are defined in the `Role` table and represent different system responsibilities such as ADMIN, DOCTOR, RECEPTIONIST, PHARMACIST, and FINANCE.
- **User–Role assignment**
  Users are assigned roles through the `UserRole` junction table, allowing a user to have one or more roles. Foreign key constraints ensure that all role assignments are valid.
- **Controlled data access using views**
  Database views are used to restrict and simplify data access. Each view exposes only the data required by specific roles (e.g., scheduling views for reception staff, clinical views for doctors, financial views for finance users).
- **Controlled operations using stored procedures**
  Sensitive actions such as appointment confirmation, visit updates, prescription handling, billing, and role assignment are executed through stored procedures. This prevents direct modification of base tables and ensures that business rules are consistently enforced.
- **Auditing and monitoring**
  Triggers record changes to user accounts, role assignments, and other sensitive entities, ensuring accountability and traceability of role-based actions.

Role Name	**Access via Views**	**Key Operations via Stored Procedures**
ADMIN	User, audit, and system configuration views	User creation, role assignment, department leadership management
DOCTOR	Patient records, visits, diagnoses, schedules	Start/end visits, add services, prescriptions, attachments
RECEPTIONIST	Appointment and availability views	Book and confirm appointments
PHARMACIST	Prescription and inventory views	Dispense medication, manage stock and batches
FINANCE	Invoice, payment, and supplier views	Record payments, manage supplier invoices

Table 9. Role-Based Access Control Summary table

Key RBAC Design Benefits:

- Prevents unauthorized data access
- Limits exposure of sensitive tables
- Centralizes authorization logic in the database
- Supports auditing and accountability
- Agns system access with real hospital roles

4.3 Password Encryption and Sensitive Data Protection

4.3.1 Password Security

- Current: SHA256 hashing used for password storage
- Issues: No salt, outdated algorithm, weak policy (6 char minimum)
- Risk: High - vulnerable to rainbow table attacks

4.3.2 Sensitive Data Protection

- Current: No encryption of medical records or PHI data
- Issues: Patient data stored in plain text
- Risk: Critical - violates HIPAA compliance

4.3.3 Session Security

- Current: Proper session management with regeneration
- Status: Secure

4.4 SQL Injection Prevention

4.4.1 Database Security

- Current: PDO with prepared statements throughout
- Status: Secure - all user inputs properly parameterized
- Examples: Stored procedures and parameterized queries used consistently

4.4.2 Input Validation

- Current: htmlspecialchars() for XSS prevention
- Status: Adequate

5. Web Application Integration and System Testing

5.1 Web Application Overview

The Hospital Management System is a comprehensive web-based application built using PHP and MySQL, designed to streamline hospital operations and improve patient care. The system implements a modular architecture with role-based access control, serving five distinct user types: Administrators, Doctors, Receptionists, Pharmacists, and Finance staff.

5.1.1 System Architecture

- Frontend: PHP-based web application with Bootstrap 5 framework for responsive design
- Backend: MySQL database with stored procedures and views for data management
- User Interface: Clean, intuitive interface with role-specific dashboards and navigation
- Security: Session-based authentication with role-based access control (RBAC)
- API Layer: RESTful endpoints for AJAX functionality and real-time updates

5.1.2 Key Features

- Patient Management: Complete patient lifecycle from registration to discharge
- Appointment Scheduling: Automated booking system with conflict resolution
- Medical Records: Comprehensive electronic medical records (EMR) system
- Pharmacy Integration: Medication dispensing and inventory management
- Financial Management: Billing, invoicing, and payment processing
- Analytics Dashboard: Real-time reporting and business intelligence

5.2 CRUD Functionality

The application implements comprehensive Create, Read, Update, and Delete (CRUD) operations across all major entities, ensuring data integrity and user-friendly interfaces.

5.2.1 Patient Management

- **Create**: New patient registration with comprehensive demographic data
- **Read**: Patient search and detailed profile viewing
- **Update**: Patient information modification and medical history updates
- **Delete**: Soft delete functionality with audit trail preservation

5.2.2 Appointment Management\

- **Create**: Automated appointment scheduling with doctor availability checking
- **Read**: Appointment calendar view and detailed appointment information
- **Update**: Rescheduling, status updates (confirmed, in-progress, completed, cancelled)
- **Delete**: Appointment cancellation with notification system

5.2.3 Medical Services

- **Create**: Addition of diagnoses, treatments, and medical services during visits
- **Read**: Comprehensive visit history and medical record access
- **Update**: Modification of treatment plans and prescription adjustments
- **Delete**: Removal of erroneous entries with audit logging

5.2.4 User Management

- **Create**: New user registration with role assignment
- **Read**: User profile management and access level viewing
- **Update**: Password changes, role modifications, and profile updates
- **Delete**: User deactivation with data preservation

5.3 Analytics and Reporting Features

The system provides comprehensive analytics and reporting capabilities to support data-driven decision making.

5.3.1 Dashboard Analytics

- **Real-time KPIs**: Today's appointments, confirmed visits, completed procedures
- **Status Tracking**: Visual representation of appointment statuses
- **Performance Metrics**: Doctor productivity, department utilization
- **Financial Overview**: Revenue tracking and payment status monitoring

5.3.2 Administrative Reports

- **Audit Logs**: Complete system activity tracking with user actions
- **User Analytics**: Login patterns, role distribution, and access statistics
- **System Health**: Database performance, error logs, and maintenance alerts

5.3.3 Clinical Reports

- **Patient Statistics**: Demographic analysis, visit frequency, treatment outcomes
- **Doctor Performance**: Appointment completion rates, patient satisfaction metrics
- **Department Analytics**: Resource utilization, service demand patterns

5.3.4 Financial Reports

- **Revenue Analysis**: Income by department, service type, and time period
- **Payment Tracking**: Outstanding balances, payment methods, collection rates
- **Cost Analysis**: Service costs, medication expenses, operational expenses

5.4 Database–Web Integration

The application employs a robust database integration layer ensuring seamless data flow between the web interface and MySQL database.

5.4.1 Database Abstraction Layer

```
class Database {
  private $host = "localhost";
  private $db_name = "hospital_management_system";
  private $username = "root";
  private $password = "";

  public function getConnection() {
       // PDO connection with error handling
  }

  public function callProcedure($procedureName, $inParams, $outParams) {
       // Stored procedure execution with parameter binding
  }

  public function queryView($viewName, $whereClause, $params) {
       // View querying with dynamic conditions
  }
 }
```

5.4.2 Stored Procedure Integration

- **Data Validation**: Server-side validation through stored procedures
- **Transaction Management**: Atomic operations ensuring data consistency
- **Audit Trail**: Automatic logging of all data modifications
- **Business Logic**: Complex operations encapsulated in database layer

5.4.3 View-Based Data Access

- **Pre-optimized Queries**: Database views for complex data relationships
- **Security**: Controlled data access through view definitions
- **Performance**: Reduced query complexity and improved response times

5.4.4 AJAX Integration

```
function loadNotifications() {
  $.ajax({
       url: '/hospital_management/api/notifications.php',
       method: 'GET',
       dataType: 'json',
       success: function(response) {
```

```
                updateNotificationBadge(response.unread_count);
                displayNotifications(response.notifications);
            }
        });
    }
```

5.5 System Testing and Validation

Comprehensive testing strategies ensure system reliability, security, and performance.

5.5.1 Unit Testing

- **Database Layer**: Testing of stored procedures and view queries
- **API Endpoints**: Validation of RESTful API responses
- **Business Logic**: Testing of core application functions
- **Input Validation**: Sanitization and validation of user inputs

5.5.2 Integration Testing

- **Database-Web Interface**: End-to-end data flow validation
- **API Integration**: Testing of AJAX calls and real-time updates
- **Cross-Module Communication**: Inter-role functionality verification
- **Payment Processing**: Financial transaction validation

5.5.3 User Acceptance Testing (UAT)

- **Role-Based Testing**: Each user role tested by domain experts
- **Workflow Validation**: Complete patient journey testing
- **Performance Testing**: Load testing under various user scenarios
- **Security Testing**: Authentication, authorization, and data protection

5.5.4 Security Validation

- **SQL Injection Prevention**: Prepared statements and parameter binding
- **Cross-Site Scripting (XSS)**: Input sanitization and output encoding
- **Session Security**: Secure session management and timeout handling
- **Access Control**: Role-based permission validation

5.5.5 Performance Testing

- **Database Query Optimization**: Index usage and query performance analysis
- **Page Load Times**: Frontend performance optimization

This comprehensive testing approach ensures the Hospital Management System maintains high standards of reliability, security, and user satisfaction in a critical healthcare environment.

6. System Documentation and Project Presentation

6.1 Project Demonstration Summary

The project demonstration provided an opportunity to present not only the technical implementation of the Hospital Management System (HMS), but also the overall design thinking and learning outcomes achieved throughout the project. Through the live demonstration, we were able to show how the system functions as a complete, integrated solution rather than a collection of isolated features.

During the demonstration, the system was presented from the perspective of real hospital users, including administrators, receptionists, doctors, pharmacists, and finance staff. This helped illustrate how role-based access control shapes each user's interaction with the system and ensures that users can only access data and operations relevant to their responsibilities. Demonstrating these role-based workflows reinforced the importance of security and usability in real-world systems.

6.2 Team Contribution and Task Division

Member name	**Taks**	**Task Division**
Nguyen Thi Phuong Thao	Database Design & Schema Implementation	● Responsible for the overall database design of the System. ● Creating the conceptual and logical ER diagrams, identifying all core entities, attributes, and relationships. ● Ensuring the schema is normalized up to Third Normal Form (3NF). ● Designing the physical database schema and implementing table definitions with appropriate primary keys, foreign keys, constraints, and cascading rules to maintain data integrity and consistency across all system modules.
Le Thao Vy	Database Logic, Triggers, and Performance Optimization	● Responsible for implementing advanced database logic and optimizing system performance. ● Developing stored procedures for core operations. ● Creating triggers for audit logging and automatic pharmacy stock updates ● Defining database views to support reporting and analytics. ● Designing indexing strategies and applying to improve query efficiency on frequently accessed data.
Do Thi Hai Binh	Database Security Configuration & System Testing	● Responsible for securing the database and validating system correctness. ● Including configuring MySQL users, roles, and privileges to enforce role-based access control (RBAC), ● Implementing secure authentication mechanisms such as password hashing. ● Ensuring compliance with data protection requirements.

		● Testing system workflows, verifying integrity constraints and trigger behavior, and identifying and resolving security or data consistency issues.