VinUniversity

Smart Inventory & Procurement Management System

Tran Quang Khai
Thai Huu Tri
Nguyen Ngoc Han

COMP3030 - Databases and Database Systems
Dec 15, 2025

1 Introduction & System Overview

Small retail stores often struggle to maintain accurate and timely information about their inventory. Typical problems include stockouts, overstocking, inconsistent records between different locations and a lack of visibility into which products are actually selling well.

The goal of the **Store Inventory and Product Management System (SIPMS)** is to provide a lightweight but realistic database-backed web application that helps store managers:

- Maintain a central catalogue of products and categories,
- Track stock levels across multiple locations (warehouses and stores),
- Record stock movements caused by purchases, sales, transfers and manual adjustments,
- Monitor low-stock products and generate simple reports,
- Control access using user roles and keep an audit trail of critical actions.

This design document focuses on the database aspects of SIPMS: conceptual and logical modelling, physical schema definition, and project planning. Application implementation details (Django backend and React frontend) will be covered in later milestones.

2 Conceptual & Logical Design

2.1 Functional Requirements

The main functional requirements of SIPMS are:

1. The system shall allow administrators to manage **product categories** (create, update, deactivate).
2. The system shall store a **product catalogue** including name, SKU, barcode, unit price, unit of measure and reorder level for each product.
3. The system shall maintain a list of **locations** (warehouses and stores) where stock is stored.
4. For each pair `(product, location)`, the system shall store the current **quantity on hand**.
5. The system shall support **purchase orders** (PO) from suppliers, with line items and status (draft, approved, received, cancelled).
6. The system shall support **sales orders** (SO) from customers, with line items and status (draft, confirmed, refunded, cancelled).
7. The system shall record **stock movements** for all inventory changes, including purchases, sales, internal transfers and manual adjustments.
8. The system shall support **internal transfers** by moving stock from one location to another.

9. The system shall provide a basic **dashboard** showing total stock per location, low-stock alerts and top-selling products.
10. The system shall manage **users and roles** and restrict access to certain actions (for example only managers may approve POs or SOs).
11. The system shall maintain an **audit log** for important operations (logins, order status changes, stock adjustments).

2.2 Non-functional Requirements

Non-functional requirements focus on quality attributes of the system:

1. **Usability:** The web interface should be easy to use for non-technical store staff with minimal training.
2. **Performance:** Typical queries (viewing stock per location, searching products by name, loading dashboards) should respond within a few seconds for the expected data size in this project.
3. **Reliability:** The database should preserve data integrity using primary keys, foreign keys and constraints.
4. **Security:** Users must authenticate with a username and password, and their permissions depend on their role.
5. **Consistency:** Stock quantities shown on dashboards and reports must be consistent with the underlying stock movement history.
6. **Extensibility:** The schema should be flexible enough to add new locations, product attributes or reports with minimal structural change.
7. **Portability:** The database shall run on MySQL, and the design should be largely transferable to other relational DBMS.

2.3 Conceptual Data Model

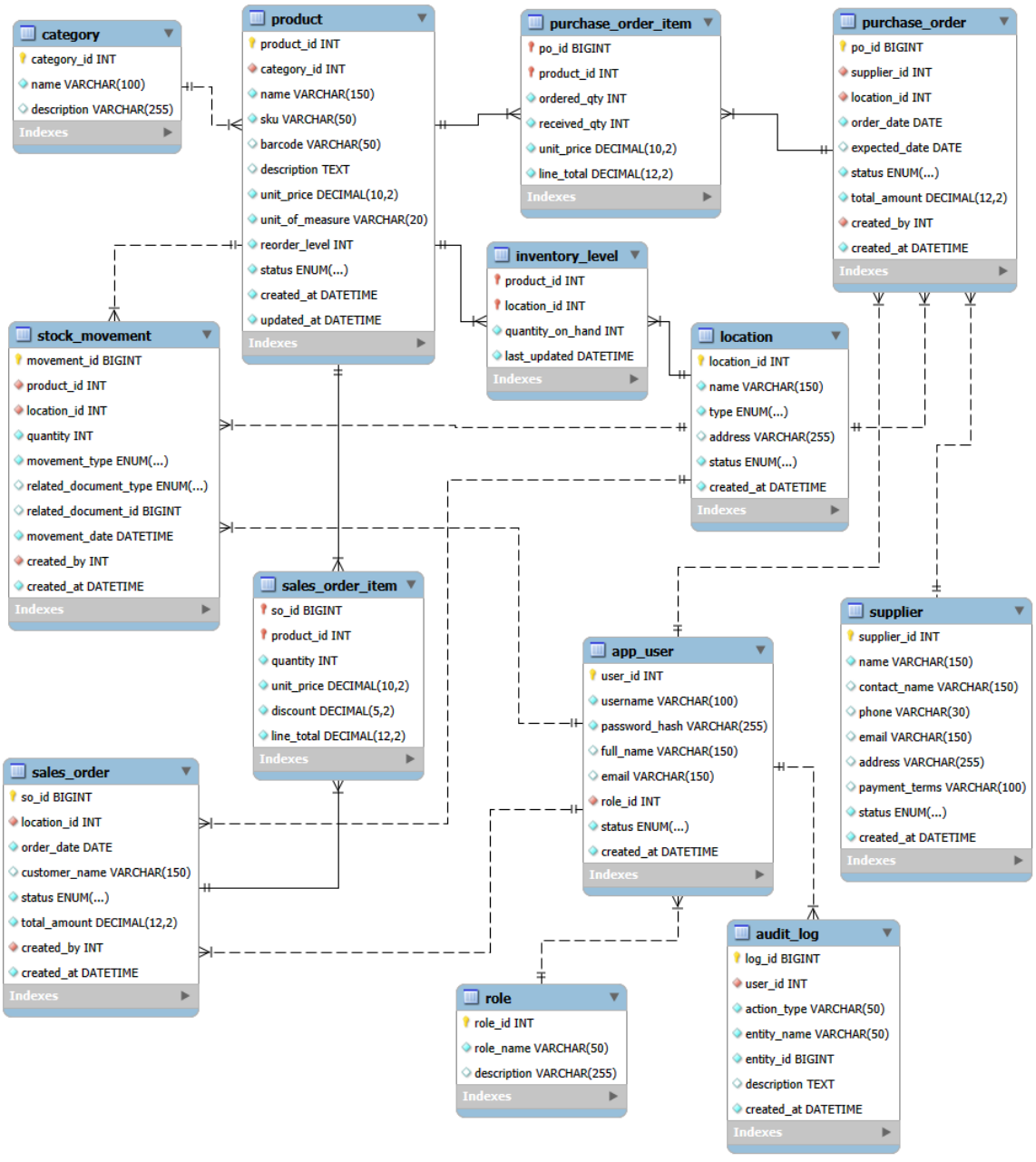
Entities Overview

At the conceptual level, SIPMS includes the following main entities:

- **Category:** groups similar products.
- **Product:** individual items that can be purchased or sold.
- **Supplier:** vendors who supply products.
- **Location:** warehouses or stores where stock is held.
- **InventoryLevel:** current quantity of a product at a specific location; identified by the pair (product, location).
- **PurchaseOrder / PurchaseOrderItem:** represent goods ordered from suppliers and the individual line items.

- **SalesOrder / SalesOrderItem:** represent goods sold to customers and the individual line items.
- **StockMovement:** atomic changes in stock quantity, characterised by movement type and signed quantity.
- **Role:** defines a set of permissions.
- **User:** application users, each assigned exactly one role.
- **AuditLog:** records important user actions.

Entity–Relationship Diagram



The conceptual Entity–Relationship Diagram (ERD) shows the entities, primary keys and relationships between them. Key relationships include:

- Each product belongs to exactly one category.
- Each product can appear in many purchase order items and sales order items.
- InventoryLevel records the quantity for each (product, location) pair.
- PurchaseOrder and SalesOrder have one-to-many relationships to their line items.
- StockMovement links to exactly one product, one location and one user who created the movement.
- Each user is assigned exactly one role, and each audit log entry references a user.

The complete ERD for SIPMS is shown in Figure 2.3 in Appendix A.

Business Rules

Some key business rules captured by the conceptual model are:

- **BR1 – No negative stock:** stock on hand must not become negative at any location.
- **BR2 – Unique stock record:** for each product and location there is at most one InventoryLevel record.
- **BR3 – Product classification:** every product must belong to exactly one category.
- **BR4 – Purchase order life cycle:** a purchase order goes through states (draft, approved, partially received, closed, cancelled) and only approved orders can create stock movements.
- **BR5 – Sales order life cycle:** a sales order goes through states (draft, confirmed, refunded, cancelled) and only confirmed orders can decrease stock.
- **BR6 – Internal transfer consistency:** each transfer between locations is represented as two stock movements (outbound and inbound) with equal absolute quantities.
- **BR7 – Role-based actions:** only users with a manager-like role may approve or cancel orders or perform manual adjustments.
- **BR8 – Auditability:** important actions (login, approval, cancellation, adjustment) must be recorded in the AuditLog table with user, time and description.

2.4 Normalization to Third Normal Form (3NF)

The SIPMS database schema was designed directly with normalization principles in mind. All major entities were analysed for functional dependencies (FDs), partial dependencies, and transitive dependencies. This section explains the normalization process and demonstrates that all relations satisfy Third Normal Form (3NF).

Approach

Rather than decomposing a single large unnormalized table, the SIPMS schema was constructed by modelling real-world entities: `Product`, `Category`, `Location`, `Supplier`, `PurchaseOrder`, `SalesOrder`, `InventoryLevel`, `StockMovement`, and supporting entities such as `App_User`, `Role`, and `AuditLog`.

For each entity, we identify:

- its primary key,
- its attributes and functional dependencies,
- foreign key constraints,
- and whether any non-key attributes depend on other non-key attributes.

A table is in 3NF if:

- it is in 2NF, and
- every non-key attribute depends only on the key, the whole key, and nothing but the key.

Below we justify 3NF compliance for each core component.

Product

```
Product(product_id, category_id, name, sku, barcode,
description, unit_price, unit_of_measure, reorder_level,
status, created_at, updated_at)
```

Functional dependencies:

- `product_id` → all other attributes.
- `category_id` is a foreign key and does not determine any product attributes; it only specifies product category.

All non-key attributes depend only on `product_id`. There are no partial or transitive dependencies.
Therefore, `Product` is in 3NF.

Category

```
Category(category_id, name, description)
```

Functional dependency:

- `category_id` → `name`, `description`.

All attributes depend on the primary key `category_id` and there are no non-key attributes determining others.
Therefore, `Category` is in 3NF.

Supplier

```
Supplier(supplier_id, name, contact_name, phone, email,
address, payment_terms, status, created_at)
```

Functional dependency:

- `supplier_id` → all other attributes.

All non-key attributes describe a supplier and depend solely on `supplier_id`. No transitive dependencies exist.
Therefore, `Supplier` is in 3NF.

Location

```
Location(location_id, name, type, address, status, created_at)
```

Functional dependency:

- `location_id` → all other attributes.

All attributes depend on the primary key `location_id`; there are no dependencies among non-key attributes.
Therefore, `Location` is in 3NF.

InventoryLevel

```
InventoryLevel(product_id, location_id, quantity_on_hand,
last_updated)
```

Composite primary key: (`product_id`, `location_id`).

Functional dependency:

- (`product_id`, `location_id`) → `quantity_on_hand`, `last_updated`.

No attribute depends solely on one component of the composite key. No transitive dependencies are present.
Therefore, `InventoryLevel` is in 3NF.

PurchaseOrder

```
PurchaseOrder(po_id, supplier_id, location_id, order_date,
expected_date, status, total_amount,
created_by, created_at)
```

Functional dependency:

- `po_id` → all other attributes.

All non-key attributes describe a single purchase order and depend solely on `po_id`.
Therefore, `PurchaseOrder` is in 3NF.

PurchaseOrderItem

```
PurchaseOrderItem(po_id, product_id, ordered_qty, received_qty,
unit_price, line_total)
```

Composite primary key: (`po_id`, `product_id`).

Functional dependency:

- (`po_id`, `product_id`) → `ordered_qty`, `received_qty`, `unit_price`, `line_total`.

All numeric values belong to a specific line of a purchase order, identified by the composite key. There are no partial or transitive dependencies.
Therefore, `PurchaseOrderItem` is in 3NF.

SalesOrder

```
SalesOrder(so_id, location_id, order_date, customer_name,
status, total_amount, created_by, created_at)
```

Functional dependency:

- `so_id` → all other attributes.

All non-key attributes describe a single sales order and depend only on `so_id`.
Therefore, `SalesOrder` is in 3NF.

SalesOrderItem

```
SalesOrderItem(so_id, product_id, quantity, unit_price,
discount, line_total)
```

Composite primary key: (`so_id`, `product_id`).

Functional dependency:

- (`so_id`, `product_id`) → `quantity`, `unit_price`, `discount`, `line_total`.

All attributes refer to a single sales order line and depend on the full composite key.
Therefore, `SalesOrderItem` is in 3NF.

StockMovement

```
StockMovement(movement_id, product_id, location_id, quantity,
movement_type, related_document_type,
related_document_id, movement_date,
created_by, created_at)
```

Functional dependency:

- `movement_id` → all other attributes.

Each record describes a single inventory change; all attributes describe that movement and none depend on another non-key attribute.
Therefore, `StockMovement` is in 3NF.

Role

```
Role(role_id, role_name, description)
```

Functional dependency:

- `role_id` → `role_name`, `description`.

Therefore, `Role` is in 3NF.

App_User

```
App_User(user_id, username, password_hash, full_name,
email, role_id, status, created_at)
```

Functional dependency:

- `user_id` → all other attributes.

No non-key attribute (e.g. `email`) determines other attributes; all depend on `user_id`.
Therefore, `App_User` is in 3NF.

AuditLog

```
AuditLog(log_id, user_id, action_type, entity_name,
entity_id, description, created_at)
```

Functional dependency:

- `log_id` → all other attributes.

Each log entry is identified by `log_id` and all details depend on this key.
Therefore, `AuditLog` is in 3NF.

Conclusion

All tables in the SIPMS schema satisfy:

- no repeating groups (1NF),
- no partial dependencies on composite keys (2NF),
- no transitive dependencies involving non-key attributes (3NF).

Therefore, the logical schema of SIPMS is fully normalized to Third Normal Form (3NF).

3 Physical Schema Definition

The physical schema implements the conceptual design using MySQL. It consists of tables, foreign keys, indexes, views, and constraints defined in `database/schema.sql`.

3.1 Overview of tables and keys

Each conceptual entity corresponds to a physical table with appropriate primary and foreign keys:

- **Category** – PK: `category_id`.
- **Product** – PK: `product_id`; FK: `category_id` → `Category`.
- **Supplier** – PK: `supplier_id`.
- **Location** – PK: `location_id`.
- **InventoryLevel** – PK: (`product_id`, `location_id`); FK to `Product` and `Location`.
- **PurchaseOrder** – PK: `po_id`; FK: `supplier_id`, `location_id`, `created_by`.
- **PurchaseOrderItem** – PK: (`po_id`, `product_id`); FK to `PurchaseOrder` and `Product`.
- **SalesOrder** – PK: `so_id`; FK: `location_id`, `created_by`.
- **SalesOrderItem** – PK: (`so_id`, `product_id`); FK to `SalesOrder` and `Product`.
- **StockMovement** – PK: `movement_id`; FK: `product_id`, `location_id`, `created_by`.
- **Role** – PK: `role_id`.
- **App_User** – PK: `user_id`; FK: `role_id` → `Role`.
- **AuditLog** – PK: `log_id`; FK: `user_id` → `App_User`.

3.2 DDL scripts

All `CREATE TABLE`, `ALTER TABLE`, index definitions and constraint declarations are located in `database/schema.sql`. Executing the file initializes the entire SIPMS database from scratch. This script acts as the single source of truth for the physical schema.

3.3 Views

The following MySQL views were added to simplify dashboard queries:

- **v_inventory_summary**: joins `InventoryLevel`, `Product` and `Location` to show real-time stock.
- **v_low_stock**: filters products where `quantity_on_hand` < `reorder_level`.
- **v_top_selling_products**: aggregates `SalesOrderItem` by product and computes revenue.

3.4 Indexes and partitioning

Secondary indexes improve query performance on:

- foreign keys (`product_id`, `location_id`, `supplier_id`, `created_by`),
- frequently searched fields such as `product.name`,
- date fields like `movement_date` and `order_date`.

Partitioning is not required for the course scale, but in real deployments tables such as `StockMovement` may be range-partitioned by date for improved performance.

4 Task Division & Project Plan

4.1 Team member responsibilities

The project team consists of three members. Each member acts as a lead for one area and also contributes to documentation and testing for their part of the system.

- **Database & Documentation Lead - Tran Quang Khai**
  Designs the conceptual and logical schema, creates the ERD, performs normalization, prepares the DDL scripts and views and writes the main database-related sections of the documentation.
- **Backend Lead - Thai Huu Tri**
  Implements the REST API using Django, connects to the MySQL database, integrates with views and stored procedures, and documents and tests the backend endpoints.
- **Frontend Lead - Nguyen Ngoc Han**
  Builds the user interface (CRUD pages, order workflows, dashboards), integrates with the backend API, and documents and tests the user-facing features.

4.2 Timeline and activities

The overall project is organised into five activities aligned with the course milestones. Table 1 summarises the main deliverables, responsibilities and deadlines.

Main deliverable	**Responsible**	**Deadline**
Conceptual & logical database design (requirements, ERD, 3NF, design document)	Khai	Dec 15, 2025
Backend API (Django REST + DB integration)	Tri	Dec 19, 2025
Frontend UI (React + API integration)	Han	Dec 19, 2025
Reporting & dashboards (analytics and exporting features)	Han & Tri	Dec 21, 2025
Final testing, demo and report (integration of all components)	All members	Dec 22, 2025

Table 1: Project activities, responsibilities and deadlines.

5 Supporting documentation

5.1 Rationale for design decisions

The schema of SIPMS was designed to keep the core business concepts clearly separated while still supporting efficient queries.

- **Separation of product, category, supplier and location.** Products, categories, suppliers and locations are modelled as separate tables to avoid duplication of descriptive attributes (for example supplier contact information or warehouse addresses) and to simplify future changes. This also makes it easy to reuse the same product across multiple purchase or sales orders.
- **Composite keys for inventory and order line items.** The tables `InventoryLevel`, `PurchaseOrderItem` and `SalesOrderItem` use composite primary keys (`product_id` with `location_id` or order identifiers). This reflects the natural key of these concepts: a quantity always belongs to a specific product at a specific location or in a specific order. Using composite keys prevents duplicate lines for the same product and keeps the schema in 3NF.
- **Status and auditability.** Order and product tables include a `status` attribute instead of deleting rows when they are cancelled or disabled. Together with the `AuditLog` table this allows the system to keep a complete history of important changes and to support future reporting on user actions.
- **Role-based access control.** The separation between `Role` and `App_User` makes it possible to assign permissions by role (for example *Admin*, *Inventory Clerk*, *Viewer*) instead of hard-coding privileges per user. This design is extensible if new roles are needed later.
- **Indexing strategy.** All foreign key columns are indexed (such as `product_id`, `location_id`, `supplier_id`, `created_by`) to speed up joins between orders, inventory levels and master data. In addition, frequently searched attributes such as `product.name` and date fields like `movement_date` and `order_date` are indexed to support dashboard queries and time-based reports.

Overall, these decisions aim to balance normalisation (to reduce redundancy and anomalies) with practical performance considerations for typical inventory queries.

5.2 Sample data loading approach

For demonstration and testing purposes the project includes a simple sample data loading strategy.

- **Seed script.** A separate SQL file `database/seed.sql` will contain `INSERT` statements to populate core tables with a small, consistent dataset: a few categories, products, suppliers, locations, users and roles. The script is designed to be idempotent so that it can be rerun on a fresh database.

- **Referential integrity.** The seed data is ordered to respect foreign key dependencies: master tables (`Category`, `Product`, `Supplier`, `Location`, `Role`, `App_User`) are inserted first, followed by transactional tables (`PurchaseOrder`, `PurchaseOrderItem`, `SalesOrder`, `SalesOrderItem`, `InventoryLevel`, `StockMovement`, `AuditLog`).
- **Realistic scenarios.** The sample data will cover typical business scenarios such as: initial stock levels for each location, purchase orders from different suppliers, sales orders to customers, and stock movements generated from those orders. This allows the team to verify that the schema and queries behave as expected before integrating the backend and frontend.
- **Reproducibility.** Both the schema script `schema.sql` and the seed script `seed.sql` are version-controlled in the repository so that any team member (or the instructor) can create the same database state locally with a few commands.