Functional Dependencies, Normalization, and Constraints

CSC 440 — Inventory Management System

Miles Hollifield
Claire Jeffries

1. Functional Dependencies

The schema was primarily shaped by functional dependencies (FDs) derived from business rules and the structure of the required entities. In most base tables, a simple dependency exists in which the primary key determines all non key attributes. Examples include

product_id → {name, category_id, manufacturer_id, standard_batch_size}
and
ingredient_id → {name, supplier_id, type}.

These dependencies ensure that each entity is free from redundancy.

Role specialization dependencies influenced the design of MANUFACTURER and SUPPLIER, both of which rely on the *FD user_id → role-specific attributes*, enforcing a one-role-per-user rule. This justified the use of one-to-one extension tables instead of combining all roles into a single relation.

Version-controlled entities required additional constraints. In *RECIPE_PLAN*, the FD *(product_id, version_number) → {created_date, is_active}* ensures that recipe versions for each product are uniquely identifiable. This led to enforcing a *UNIQUE(product_id, version_number)* constraint. Similarly, in *FORMULATION*, the FD *(ingredient_id, effective_start_date) → {pack_size, unit_price, effective_end_date}* ensures non-overlapping formulation periods over time. These dependencies guided the decomposition decisions and prevented partial and transitive dependencies.

Intersection tables such as *RECIPE_INGREDIENT* and *FORMULATION_MATERIAL* were designed around composite-key dependencies: *(plan_id, ingredient_id) → quantity_required* and *(formulation_id, material_ingredient_id) → quantity_required*. These FDs ensured that each ingredient appears at most once within a given recipe plan or formulation.

---

2. Normalization

The final schema satisfies at least Third Normal Form (3NF) across all relations, with most tables achieving Boyce–Codd Normal Form (BCNF).

2.1 Tables in BCNF

Most tables in the design qualify for BCNF because all functional dependencies have determinants that are candidate keys. These include:

- USER
- MANUFACTURER
- SUPPLIER
- CATEGORY
- PRODUCT
- INGREDIENT
- RECIPE_PLAN (with the unique recipe-version constraint)
- FORMULATION
- INGREDIENT_BATCH
- PRODUCT_BATCH
- RECIPE_INGREDIENT
- FORMULATION_MATERIAL
- BATCH_CONSUMPTION

In each of these relations, every non-key attribute is fully dependent on the primary key, and no partial or transitive dependencies exist.

2.2 Tables in 3NF

A small number of relations are technically in 3NF rather than BCNF. This occurs primarily in role-extension tables such as *MANUFACTURER* and *SUPPLIER*, where *user_id* serves simultaneously as a primary key and foreign key. Although the dependency originates from a key in another table, all non-key attributes still depend directly and fully on the primary key of the relation. This satisfies 3NF requirements and introduces no anomalies. These structures are standard in relational designs involving specialization.

2.3 Summary

The schema contains no multi-valued dependencies, no transitive dependencies, and no violations of either BCNF or 3NF. The resulting database is minimal, non-redundant, and stable.

---

3. Constraints

Integrity constraints were distributed between the database and the application layer based on their nature and technical feasibility. SQL enforceable constraints were implemented in the database for reliability and consistency across clients.

3.1 Constraints Enforced in the Database

Several categories of constraints were captured directly in the database:

Foreign Key Constraints - Referential integrity was enforced among all core relations, including requiring each product to reference an existing manufacturer and each ingredient to reference a supplier.

Unique Constraints - Keys requiring uniqueness across combinations of attributes – such as *(product_id, version_number)* in *RECIPE_PLAN* and *(ingredient_id, effective_start_date)* in *FORMULATION* – were enforced using unique indexes.

CHECK Constraints - Positive-value checks were applied to numeric fields such as quantities and unit prices, preventing negative or illogical inputs.

Triggers - Four triggers enforce key dynamic rules:

- automatic lot number generation based on user-supplied identifiers,
- initialization of *on_hand_oz* for new ingredient batches,
- automatic decrementing of on-hand inventory during consumption,
- prevention of consumption of expired ingredient lots.

Stored Procedure Logic - The multi-step process of creating a production batch, calculating costs, and decrementing ingredient lots is implemented in the stored procedure *RecordProductionBatch*, ensuring atomicity and consistency of complex updates.

3.2 Constraints Enforced in the Application Layer

Some rules could not be implemented at the database level due to MySQL limitations or complexity:

90-Day Expiration Rule for Manufacturer Intake - This rule depends on the role of the user performing the operation; since SQL constraints cannot reference session-level role context, this check was implemented in Python.

Single-Level Composition for Compound Ingredients - Preventing nested compound ingredients requires recursive validation, which is not expressible in standard SQL. The application checks that all materials in a compound formulation are atomic.

Only One Active Recipe per Product - Activating a recipe requires ensuring that no other recipe for the same product is active. This cannot be enforced by a simple *CHECK* constraint, so the application ensures proper state transitions.

Batch Size and Inventory Sufficiency - Validating correct batch size multiples and checking ingredient availability require procedural and aggregate logic, which are implemented in application code and partially in the stored procedure.

3.3 Rationale for Application-Level Enforcement

These constraints required:

- conditional logic dependent on user role,
- cross-row or cross-table aggregation,
- recursive reasoning, which are not directly supported by MySQL constraint declarations. Implementing them in the application ensures correctness without overcomplicating the database layer.