Third Normal Form (3NF) Justification Report

Project: Global Food & Nutrition Explorer

Team: Akash Ankush Kamble, Nidhi Rajani, Goutham Chengalvala

1. Introduction

The primary goal of our database design was to transform the original Open Food Facts dataset—a single, wide, and denormalized CSV file—into a robust, efficient, and scalable relational database. To achieve this, we designed our schema to adhere to the principles of Third Normal Form (3NF).

This report justifies how our final schema, implemented in `sql/1_schema.sql`, satisfies the requirements of 1NF, 2NF, and 3NF, thereby ensuring data integrity and minimizing redundancy.

---

2. First Normal Form (1NF)

Rule: A table is in 1NF if all its attributes (columns) contain atomic (indivisible) values, and each record is unique (enforced by a primary key). There should be no repeating groups of columns.

Justification:

Our schema satisfies 1NF through the following design choices:

1. **Atomicity of Values:** The original dataset violated 1NF by storing multiple values in single columns, such as `brands`, `categories_en`, and `countries_en`, which contained comma-separated strings. Our Python `clean_data` script explicitly parses these strings into lists. Subsequently, our `normalize_data` function models these as multiple rows in dedicated junction tables (e.g., `product_brands`,

`product_categories`). This ensures that every cell in our final database holds only a single value.

2. **Unique Records:** Every table in our schema has a clearly defined Primary Key to ensure each row is unique:

- **Single-Column Keys:** Tables like `products` (PK: `code`), `brands` (PK: `brand_id`), and `nutrition_facts` (PK: `product_code`) use a single attribute to uniquely identify records.

- **Composite Keys:** Junction tables like `product_brands` use a composite primary key (`product_code` , `brand_id`) to uniquely identify each product-brand relationship.

By ensuring each cell holds a single value and each record is uniquely identifiable, our entire schema adheres to 1NF.

---

3. Second Normal Form (2NF)

Rule: A table is in 2NF if it is in 1NF and all its non-key attributes are fully functionally dependent on the *entire* primary key. This rule is primarily relevant for tables with composite primary keys.

Justification:

Our schema satisfies 2NF:

1. **Tables with Single-Column Primary Keys:** All tables with a single-column primary key (`products` , `brands` , `categories` , `countries` , `labels` , `nutrition_facts`) are automatically in 2NF by definition, as there is no possibility of partial dependency.

2. **Tables with Composite Primary Keys:** Our junction tables (`product_brands` , `product_categories` , `product_countries` , `product_labels`) are the ones where 2NF must be carefully considered. These tables have a composite primary key (e.g., `(product_code, brand_id)`). However, these tables contain **no non-key attributes**. Their only columns are the ones that form the primary key. Since there

are no non-key attributes, there can be no partial dependencies, and thus they trivially satisfy 2NF.

Therefore, every table in our schema is in 2NF.

---

4. Third Normal Form (3NF)

Rule: A table is in 3NF if it is in 2NF and it contains no transitive dependencies. A transitive dependency exists when a non-key attribute is functionally dependent on another non-key attribute.

Justification:

We specifically designed our schema to eliminate transitive dependencies found in the original flat-file dataset.

Example: Eliminating Transitive Dependency with Brands

- **Hypothetical Problem (Violation of 3NF):** If we had designed the `products` table as `(code, product_name, brand_name)`, this would create a transitive dependency. The `brand_name` is dependent on the `code`. If we were to add a `brand_headquarters` column, the dependency would be:

`code → brand_name → brand_headquarters`.

Here, `brand_headquarters` (a non-key attribute) depends on `brand_name` (another non-key attribute), which is a classic transitive dependency. This would lead to update anomalies (changing a brand's headquarters would require updating every product row for that brand).

- **Our 3NF Solution:** We resolved this by creating a separate `brands` table with `(brand_id, brand_name)`.
- In the `brands` table, `brand_name` is fully dependent only on `brand_id` (the primary key).
- The `products` table does not contain any brand information directly.
- The relationship is managed through the `product_brands` junction table.

This same logic was applied to `categories` , `countries` , and `labels` , moving them into their own dimension tables where their name attribute depends solely on their respective ID (the primary key).

Analysis of Main Tables:

- **`products` table:** Attributes like `product_name` , `nutriscore_grade` , and `nova_group` are all facts directly about the product itself. They are dependent only on the `code` (the primary key) and not on any other non-key attribute in the table.

- **`nutrition_facts` table:** All attributes (`energy_kcal_100g` , `fat_100g` , etc.) are facts directly about the product's nutritional content. They are dependent only on the `product_code` (the primary key).

By systematically removing these transitive dependencies, every table in our schema satisfies 3NF.

---

5. Conclusion

Our database schema for the "Global Food & Nutrition Explorer" successfully achieves Third Normal Form. This design provides significant advantages:

- **Data Integrity:** It prevents insertion, update, and deletion anomalies.

- **Reduced Redundancy:** It minimizes data duplication, saving storage space and improving consistency.

- **Flexibility and Scalability:** The normalized structure is easier to maintain and extend as the application's requirements grow.

This solid foundation will enable efficient and powerful querying for the subsequent phases of our project.