Smart Kitchen Helper: SQL and NoSQL Database Design

Amit Christian

2024-08-05

# Introduction

This document provides an in-depth overview of the SQL and NoSQL database design for the Smart Kitchen Helper project. The system leverages both SQL for structured data and MongoDB (NoSQL) for unstructured or semi-structured data, such as recipe details and multimedia. The design includes detailed schema diagrams, table definitions, MongoDB collections, thorough explanations of constraints, business rules, logical validations, joins, use case scenarios, and access control details. The goal is to ensure that the system is scalable, maintainable, and capable of efficiently managing the data required for the application’s features.

**Author:** Amit Christian  
**Course:** Cloud Computing, Durham College  
**Student ID:** 100937637

# SQL Database Design

## Conceptual Design

The SQL database is structured to manage the core, highly structured data for the Smart Kitchen Helper project. The design follows relational database principles to ensure:

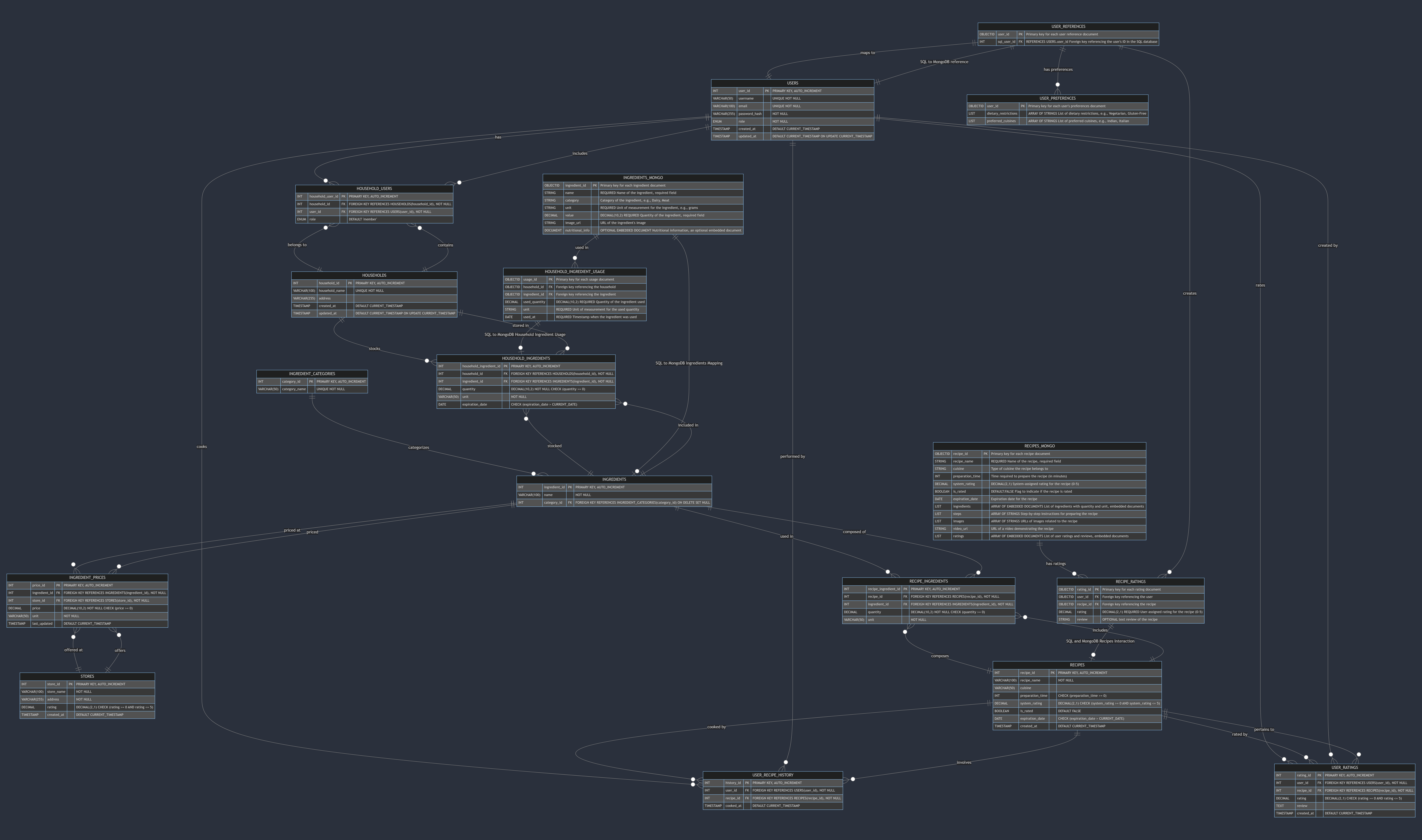
* **Data Integrity**: Ensuring the accuracy and consistency of data throughout its lifecycle. Referential integrity is maintained through the use of foreign keys, and data accuracy is enforced through constraints such as NOT NULL, UNIQUE, and CHECK constraints.
* **Consistency**: Data is consistent across the database. Transactions are designed to move the database from one consistent state to another, ensuring that any changes to the data are valid according to all defined rules.
* **Efficient Querying**: The schema is optimized for SQL queries, allowing for complex joins, aggregations, and subqueries. Indexes are applied to frequently queried fields to improve performance.
* **Complex Relationships**: The design supports intricate relationships between entities, such as one-to-many and many-to-many relationships. This is crucial for modeling real-world scenarios like users belonging to multiple households or ingredients being part of multiple recipes.
* **Business Logic Enforcement**: Business rules are embedded within the schema through the use of constraints, triggers, and stored procedures. This ensures that all operations on the data comply with the business requirements of the Smart Kitchen Helper system.
* **ACID Transactions**: The database is designed to support transactions that are Atomic, Consistent, Isolated, and Durable. This ensures that all database operations are reliable, even in the face of errors, power failures, or other issues.

### Key Aspects of the Conceptual Design

1. **Entities**: The database includes core entities like Users, Households, Ingredients, Recipes, and Stores. Each entity represents a fundamental aspect of the Smart Kitchen Helper system and is designed to hold all necessary information while ensuring referential integrity.
2. **Relationships**: Relationships between entities are clearly defined, supporting the business logic of the application. For example:
   * **Users and Households**: A user can belong to multiple households, and a household can have multiple users, each with different roles.
   * **Ingredients and Recipes**: Ingredients are associated with recipes, indicating the required quantity and unit of measurement.
   * **Households and Ingredients**: Ingredients are managed within households, allowing for tracking of inventory and expiration dates.
3. **Constraints**: Various constraints are used to enforce rules at the database level, such as:
   * **Primary Keys**: Ensure that each record is unique and can be reliably referenced.
   * **Foreign Keys**: Maintain referential integrity between related entities.
   * **Unique Constraints**: Enforce the uniqueness of fields like email addresses and usernames.
   * **Check Constraints**: Ensure that field values meet specific criteria (e.g., ingredient quantities must be non-negative).
4. **Indexes**: Indexes are applied to frequently searched fields to improve query performance. For example, indexes on email in the Users table or recipe\_name in the Recipes table help speed up searches and data retrieval.
5. **Normalization**: The database is normalized to reduce redundancy and ensure data integrity. Normalization also helps in maintaining the database by reducing the complexity of update operations.
6. **Scalability**: The design takes into account future growth, ensuring that the system can handle an increasing amount of data without significant performance degradation. This includes the potential use of partitioning and sharding strategies for very large datasets.
7. **Security**: The database includes role-based access controls, ensuring that only authorized users can perform certain operations. Sensitive data, such as passwords, is stored securely using hashing and encryption techniques.
8. **Backup and Recovery**: The database design supports regular backups and recovery strategies to protect against data loss. Transaction logs and snapshots are used to ensure that data can be restored to a consistent state in case of failure.

By adhering to these principles, the SQL database for the Smart Kitchen Helper project is robust, scalable, and capable of supporting the complex needs of the application. The conceptual design forms the foundation upon which the logical and physical designs are built, ensuring that the system remains reliable and efficient as it grows.

### ER Diagram



### Entities and Relationships

Below are detailed markdown tables for each SQL entity, including field descriptions, constraints, and use case scenarios.

#### **1. Users Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| user\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each user. |
| username | VARCHAR(50) | UNIQUE, NOT NULL | Unique username for user login. |
| email | VARCHAR(100) | UNIQUE, NOT NULL, CHECK (email LIKE '%@%.%') | Unique email for communication, must be valid. |
| password\_hash | VARCHAR(255) | NOT NULL | Encrypted password for security. |
| role | ENUM | NOT NULL | User role, determines permissions and access levels. |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the user account was created. |
| updated\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP | Timestamp of the last update to the user account. |

**Use Case Scenario:** - **When:** A new user registers for an account. - **How:** The user provides their username, email, and password. The system validates the uniqueness of username and email, encrypts the password, and assigns a default role of Guest upon successful registration. The system also tracks the creation and update timestamps to ensure that all changes are logged.

#### **2. Households Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| household\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each household. |
| household\_name | VARCHAR(100) | UNIQUE, NOT NULL | Unique name for the household. |
| address | VARCHAR(255) |  | Optional address of the household. |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the household was created. |
| updated\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP | Timestamp of the last update to the household. |

**Use Case Scenario:** - **When:** A user with the Owner role creates a new household. - **How:** The Owner provides a unique household\_name and optional address. The system ensures the household\_name is unique across all households and assigns a unique household\_id upon successful creation. The creation and update timestamps are logged to track changes.

#### **3. Household\_Users Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| household\_user\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each household-user relation. |
| household\_id | INT | FOREIGN KEY REFERENCES HOUSEHOLDS(household\_id), NOT NULL | Links to the Households table. |
| user\_id | INT | FOREIGN KEY REFERENCES USERS(user\_id), NOT NULL | Links to the Users table. |
| role | ENUM | DEFAULT 'member' | Defines the user’s role within the household. |

**Use Case Scenario:** - **When:** An Owner adds a new Member to their household. - **How:** The Owner selects an existing User and assigns them to their household with a role of either Member or Owner. The system ensures that each User can only have one role per household. The timestamps in the Households table are updated accordingly.

#### **4. Ingredient\_Categories Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| category\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each ingredient category. |
| category\_name | VARCHAR(50) | UNIQUE, NOT NULL | Name of the category (e.g., Veggie, Dairy). |

**Use Case Scenario:** - **When:** A system administrator adds a new category of ingredients. - **How:** The administrator enters a unique category\_name, which the system stores in the database. This category can then be associated with ingredients to facilitate organization and search. Categories help in filtering and sorting ingredients based on types.

#### **5. Ingredients Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| ingredient\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each ingredient. |
| name | VARCHAR(100) | NOT NULL | Name of the ingredient (e.g., Tomato, Butter). |
| category\_id | INT | FOREIGN KEY REFERENCES INGREDIENT\_CATEGORIES(category\_id) ON DELETE SET NULL | Links to the Ingredient\_Categories table. |

**Use Case Scenario:** - **When:** A Member adds or updates an ingredient in their household’s inventory. - **How:** The user specifies the ingredient name and category\_id. The system validates the data to ensure the name is provided and the category\_id exists in the Ingredient\_Categories table. This information helps categorize and manage the household’s inventory efficiently.

#### **6. Stores Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| store\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each store. |
| store\_name | VARCHAR(100) | NOT NULL | Name of the store. |
| address | VARCHAR(255) | NOT NULL | Physical address of the store. |
| rating | DECIMAL(2,1) | CHECK (rating >= 0 AND rating <= 5) | Rating of the store (0 to 5). |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the store was added. |

**Use Case Scenario:** - **When:** An

Owner or Administrator adds a new store to the system. - **How:** The user provides the store\_name, address, and rating. The system validates the rating to ensure it falls within the 0 to 5 range. The store is then added to the system for future reference in ingredient pricing. This information helps in tracking where ingredients are purchased from and their quality.

#### **7. Ingredient\_Prices Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| price\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each price entry. |
| ingredient\_id | INT | FOREIGN KEY REFERENCES INGREDIENTS(ingredient\_id), NOT NULL | Links to the Ingredients table. |
| store\_id | INT | FOREIGN KEY REFERENCES STORES(store\_id), NOT NULL | Links to the Stores table. |
| price | DECIMAL(10,2) | NOT NULL CHECK (price >= 0) | Price of the ingredient at the store. |
| unit | VARCHAR(50) | NOT NULL | Unit of the price (e.g., per kg, per liter). |
| last\_updated | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the price was last updated. |

**Use Case Scenario:** - **When:** An Owner or Member updates the price of an ingredient at a specific store. - **How:** The user selects the ingredient and store, then enters the price and unit. The system validates that the price is non-negative before updating the price information. This table allows tracking of ingredient prices across different stores, helping in cost comparison and budgeting.

#### **8. Household\_Ingredients Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| household\_ingredient\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each household-ingredient relation. |
| household\_id | INT | FOREIGN KEY REFERENCES HOUSEHOLDS(household\_id), NOT NULL | Links to the Households table. |
| ingredient\_id | INT | FOREIGN KEY REFERENCES INGREDIENTS(ingredient\_id), NOT NULL | Links to the Ingredients table. |
| quantity | DECIMAL(10,2) | NOT NULL CHECK (quantity >= 0) | Quantity of the ingredient in the household. |
| unit | VARCHAR(50) | NOT NULL | Unit of measurement for the ingredient. |
| expiration\_date | DATE | CHECK (expiration\_date > CURRENT\_DATE) | Expiration date of the ingredient. |
| is\_expired | BOOLEAN | DEFAULT FALSE | Indicates if the ingredient is expired (based on expiration\_date). |

**Use Case Scenario:** - **When:** A Member manages the ingredients in their household’s inventory. - **How:** The user can add, update, or delete ingredients, ensuring the quantity is non-negative and the expiration\_date is in the future. The system updates the household’s inventory accordingly and automatically flags ingredients as expired if the expiration\_date has passed. This table ensures that households have an accurate and up-to-date record of their ingredients.

#### **9. Recipes Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| recipe\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each recipe. |
| recipe\_name | VARCHAR(100) | NOT NULL | Name of the recipe (e.g., Butter Chicken). |
| cuisine | VARCHAR(50) |  | Type of cuisine (e.g., Indian, Italian). |
| preparation\_time | INT | CHECK (preparation\_time >= 0) | Time required to prepare the recipe (in minutes). |
| system\_rating | DECIMAL(2,1) | CHECK (system\_rating >= 0 AND system\_rating <= 5) | System-assigned rating for the recipe. |
| is\_rated | BOOLEAN | DEFAULT FALSE | Indicates if the recipe has been rated by users. |
| expiration\_date | DATE | CHECK (expiration\_date > CURRENT\_DATE) | Expiration date of the recipe (if applicable). |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the recipe was added. |

**Use Case Scenario:** - **When:** A Content Creator submits a new recipe for approval. - **How:** The user provides the recipe details, and the system validates the data, ensuring the preparation time is non-negative and the expiration date is valid. The recipe is then marked as Pending Approval and awaits review by an Owner or Moderator. This table is critical for managing the recipes available in the system.

#### **10. Recipe\_Ingredients Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| recipe\_ingredient\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each recipe-ingredient relation. |
| recipe\_id | INT | FOREIGN KEY REFERENCES RECIPES(recipe\_id), NOT NULL | Links to the Recipes table. |
| ingredient\_id | INT | FOREIGN KEY REFERENCES INGREDIENTS(ingredient\_id), NOT NULL | Links to the Ingredients table. |
| quantity | DECIMAL(10,2) | NOT NULL CHECK (quantity >= 0) | Quantity of the ingredient required for the recipe. |
| unit | VARCHAR(50) | NOT NULL | Unit of measurement for the ingredient. |

**Use Case Scenario:** - **When:** A Content Creator associates ingredients with a new recipe. - **How:** The user selects ingredients from the database and specifies the required quantity and unit for each ingredient. The system ensures that all quantities are non-negative and the units are valid. This table ensures that the recipes are accurately represented with the correct ingredients and measurements.

#### **11. User\_Recipe\_History Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| history\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each user-recipe interaction. |
| user\_id | INT | FOREIGN KEY REFERENCES USERS(user\_id), NOT NULL | Links to the Users table. |
| recipe\_id | INT | FOREIGN KEY REFERENCES RECIPES(recipe\_id), NOT NULL | Links to the Recipes table. |
| cooked\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the user cooked the recipe. |

**Use Case Scenario:** - **When:** A Member views their cooking history. - **How:** The system records each time a user cooks a recipe, storing the user\_id, recipe\_id, and cooked\_at timestamp. Users can later review their cooking history and rate or review recipes. This table is essential for tracking user activity and interactions with recipes.

#### **12. User\_Ratings Table**

| **Field Name** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| rating\_id | INT | PRIMARY KEY, AUTO\_INCREMENT | Unique identifier for each user rating. |
| user\_id | INT | FOREIGN KEY REFERENCES USERS(user\_id), NOT NULL | Links to the Users table. |
| recipe\_id | INT | FOREIGN KEY REFERENCES RECIPES(recipe\_id), NOT NULL | Links to the Recipes table. |
| rating | DECIMAL(2,1) | CHECK (rating >= 0 AND rating <= 5) | User-assigned rating for the recipe (0-5). |
| review | TEXT |  | Optional text review of the recipe. |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the rating was created. |

**Use Case Scenario:** - **When:** A Member rates and reviews a recipe they have cooked

. - **How:** The user provides a rating and an optional review after cooking a recipe. The system validates the rating and stores it along with the user\_id and recipe\_id. This table is crucial for capturing user feedback and influencing the overall rating of recipes.

### Logical Joins and Validations

The following examples illustrate how different entities in the SQL database can be joined to retrieve and manage data efficiently.

#### **1. Viewing Household Ingredients**

SELECT   
 h.household\_name,   
 i.name AS ingredient\_name,   
 hi.quantity,   
 hi.unit,   
 hi.expiration\_date,  
 hi.is\_expired  
FROM   
 Households h  
JOIN   
 Household\_Ingredients hi ON h.household\_id = hi.household\_id  
JOIN   
 Ingredients i ON hi.ingredient\_id = i.ingredient\_id  
WHERE   
 h.household\_id = ?;

**Explanation:**  
This query retrieves all ingredients in a specific household, including their quantities, units, expiration dates, and whether they are expired. Logical joins ensure that the household, ingredients, and their quantities are correctly linked, and null values are handled by excluding records where necessary.

#### **2. Fetching Store Prices for Ingredients**

SELECT   
 s.store\_name,   
 i.name AS ingredient\_name,   
 ip.price,   
 ip.unit,   
 ip.last\_updated  
FROM   
 Stores s  
JOIN   
 Ingredient\_Prices ip ON s.store\_id = ip.store\_id  
JOIN   
 Ingredients i ON ip.ingredient\_id = i.ingredient\_id  
WHERE   
 i.ingredient\_id = ?;

**Explanation:**  
This query retrieves the prices of a specific ingredient across all stores. Logical joins link stores, prices, and ingredients, ensuring that only valid data is returned.

#### **3. Total Ingredients in a Household**

SELECT   
 COUNT(hi.ingredient\_id) AS total\_ingredients  
FROM   
 Household\_Ingredients hi  
WHERE   
 hi.household\_id = ?;

**Explanation:**  
This query returns the total number of different ingredients in a household, ensuring that only valid records are counted and nulls are excluded.

#### **4. Average User Rating for a Recipe**

SELECT   
 r.recipe\_name,   
 AVG(ur.rating) AS average\_rating  
FROM   
 Recipes r  
JOIN   
 User\_Ratings ur ON r.recipe\_id = ur.recipe\_id  
WHERE   
 r.recipe\_id = ?;

**Explanation:**  
This query calculates the average user rating for a specific recipe, ensuring that only valid ratings are included in the calculation.

#### **5. Fetching Expiring Ingredients**

SELECT   
 i.name AS ingredient\_name,   
 hi.expiration\_date  
FROM   
 Ingredients i  
JOIN   
 Household\_Ingredients hi ON i.ingredient\_id = hi.ingredient\_id  
WHERE   
 hi.is\_expired = TRUE AND hi.household\_id = ?;

**Explanation:**  
This query retrieves all ingredients in a household that are marked as expired. This helps users identify and remove expired items from their inventory.

### Access Control Details

Access Control in the Smart Kitchen Helper system is based on Role-Based Access Control (RBAC), where users are assigned specific roles, and their permissions are determined by these roles. This ensures that sensitive operations are restricted to authorized users, maintaining the system’s security and integrity.

#### **Roles and Their Permissions**

| Role | Access Level | Permissions | Restrictions |
| --- | --- | --- | --- |
| Guest | Limited | View public recipes, explore the application | Cannot modify data, access private household data |
| Member | Standard | Manage household ingredients, view and rate recipes | Cannot manage household settings or approve content |
| Owner | Elevated | Full household management, approve or reject content | Cannot alter system-wide settings or other households |
| Moderator | Elevated (Global) | Review and moderate user-generated content, approve recipes | Cannot manage household settings |
| Administrator | Highest | Full system access, manage users, roles, global settings | Reserved for system developers or IT personnel |
| Content Creator | Specialized | Submit new recipes for approval, update or delete own recipes | Cannot approve or publish recipes |
| Viewer | Read-Only | View all public content, including recipes and reviews | Cannot modify or interact with data beyond viewing |

#### **Scenarios Demonstrating Access Control**

1. **Scenario 1: Adding a New Household Member**
   * **Role:** Owner
   * **Action:** An Owner adds a new member to their household. The system ensures that only the Owner can perform this action, and the new member is assigned the Member role by default.
2. **Scenario 2: Submitting a Recipe**
   * **Role:** Content Creator
   * **Action:** A Content Creator submits a new recipe. The recipe is automatically marked as Pending Approval, and the system restricts the Content Creator from publishing it directly. Only a Moderator or Owner can approve the recipe.
3. **Scenario 3: Accessing User Reviews**
   * **Role:** Guest
   * **Action:** A Guest user tries to access user reviews for a recipe. The system allows this action since reading reviews is permitted for all roles. However, the Guest cannot submit a review.
4. **Scenario 4: Approving a Recipe**
   * **Role:** Moderator
   * **Action:** A Moderator logs in and reviews a recipe submission. After checking the details, they approve the recipe, making it publicly visible. The system records this action in the audit log.
5. **Scenario 5: Viewing Household Ingredients**
   * **Role:** Member
   * **Action:** A Member wants to view and update the ingredients in their household. The system grants access to the household inventory and allows the member to make changes, such as updating quantities or removing expired items.

# MongoDB (NoSQL) Database Design

## Conceptual Design

The NoSQL database, implemented using MongoDB, is used to store unstructured or semi-structured data such as detailed recipe instructions, ingredient images, and media URLs. MongoDB’s flexible schema allows for efficient management of this data type, accommodating varying data models and structures. This design complements the SQL database by handling more dynamic and content-rich data, enabling the system to offer a richer user experience.

### Collections Overview

1. **recipes\_mongo:** Stores detailed information about recipes, including preparation steps, images, and YouTube video URLs.
2. **ingredients\_mongo:** Manages images and detailed information for ingredients.
3. **household\_ingredient\_usage:** Tracks the usage of ingredients within households, including historical usage data.
4. **recipe\_ratings:** Collects user ratings and reviews for recipes, allowing for community feedback and recipe improvement.
5. **user\_preferences:** Stores user-specific preferences and settings, such as dietary restrictions and preferred cuisines.

### Collections with Constraints, Rules, Logical Validations, and Relationships

#### **1. Recipes Collection**

{  
 "recipe\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d67"),  
 "recipe\_name": "Butter Chicken",  
 "cuisine": "Indian",  
 "preparation\_time": 45,  
 "system\_rating": 4.5,  
 "is\_rated": true,  
 "expiration\_date": ISODate("2023-12-31T00:00:00Z"),  
 "ingredients": [  
 {  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d68"),  
 "name": "Chicken",  
 "quantity": 500,  
 "unit": "grams"  
 },  
 {  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d69"),  
 "name": "Butter",  
 "quantity": 100,  
 "unit": "grams"  
 }  
 ],  
 "steps": [  
 "Marinate the chicken with spices and yogurt.",  
 "Cook the marinated chicken in butter until done.",  
 "Prepare the sauce and simmer with chicken."  
 ],  
 "images": [  
 "https://example.com/butter\_chicken\_step1.jpg",  
 "https://example.com/butter\_chicken\_step2.jpg"  
 ],  
 "video\_url": "https://youtube.com/example\_butter\_chicken",  
 "ratings": [  
 {  
 "user\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d61"),  
 "rating": 4.5,  
 "review": "Delicious and easy to make!"  
 }  
 ]  
}

**Constraints and Rules:** - **Schema Flexibility:** MongoDB allows for dynamic schemas, so documents can vary within the same collection. - **Validation:** MongoDB can enforce JSON Schema validation for required fields and data types. - **Logical Expiration:** Recipes include an expiration date, which could be used to ensure ingredients are still valid or the recipe is still relevant.

**Use Case Scenario:** - **When:** A Content Creator adds a new recipe to the MongoDB collection. - **How:** The user specifies the recipe details, including ingredients, preparation steps, and an optional video URL. The system validates the JSON document and stores it in the recipes collection, where it awaits approval from a Moderator or Owner.

#### **2. Ingredients Collection**

{  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d68"),  
 "name": "Chicken",  
 "category": "Meat",  
 "unit": "grams",  
 "value": 500,  
 "image\_url": "https://example.com/chicken.jpg",  
 "nutritional\_info": {  
 "calories": 200,  
 "protein": "20g",  
 "fat": "10g",  
 "carbohydrates": "0g"  
 }  
}

**Constraints and Rules:** - **Category Validation:** Ingredients are categorized, which helps in filtering and querying. - **Unit and Value:** Each ingredient has a unit and a value, helping in accurate inventory management. - **Optional Fields:** MongoDB allows fields like nutritional\_info to be optional, providing flexibility.

**Use Case Scenario:** - **When:** A Member adds or updates an ingredient in their household inventory. - **How:** The user specifies the ingredient details, including category, unit, and value. The system stores this information in the ingredients collection, where it can be retrieved for inventory management.

#### **3. Household Ingredient Usage Collection**

{  
 "usage\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d70"),  
 "household\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d60"),  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d68"),  
 "used\_quantity": 250,  
 "unit": "grams",  
 "used\_at": ISODate("2023-08-04T14:00:00Z")  
}

**Constraints and Rules:** - **Usage Tracking:** Tracks the quantity of ingredients used in households, allowing for detailed inventory management.

**Use Case Scenario:** - **When:** A Member logs the usage of an ingredient in their household. - **How:** The user records the quantity of the ingredient used and the date of usage. The system updates the household\_ingredient\_usage collection, providing a historical log of ingredient usage.

#### **4. Recipe Ratings Collection**

{  
 "rating\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d71"),  
 "user\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d61"),  
 "recipe\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d67"),  
 "rating": 4.5,  
 "review": "This recipe was fantastic! The butter chicken turned out perfect."  
}

**Constraints and Rules:** - **Rating Validation:** Ratings should be within a 0 to 5 range, which can be enforced through application logic or validation.

**Use Case Scenario:** - **When:** A Member rates and reviews a recipe they have cooked. - **How:** The user provides a rating and an optional text review. The system validates and stores this information in the recipe\_ratings collection, contributing to the overall rating of the recipe.

#### **5. User Preferences Collection**

{  
 "user\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d61"),  
 "dietary\_restrictions": ["Vegetarian", "Gluten-Free"],  
 "preferred\_cuisines": ["Indian", "Italian"]  
}

**Constraints and Rules:** - **Schema Flexibility:** Allows for varying preferences and settings per user. - **Array Fields:** Supports lists of preferences, providing flexibility in how user data is stored and accessed.

**Use Case Scenario:** - **When:** A Member updates their dietary preferences or preferred cuisines. - **How:** The user selects or updates their preferences in the system, which stores this data in the user\_preferences collection. This information can later be used to personalize recipe suggestions.

### Summary of Rules, Validations, and System Capabilities

#### **Logical Validations**

1. **Unique Constraints:**
   * Ensure that fields like username, email, and category\_name are unique, preventing duplicate entries.
   * **Edge Case Handling:** If a duplicate entry is attempted, the system should throw a validation error and prompt the user to enter a unique value.
2. **Positive Values and Quantities:**
   * Enforce that numerical values, such as ingredient quantities and prices, are non-negative.
   * **Edge Case Handling:** Input validation should prevent the submission of negative values, and an error message should be displayed to guide the user.
3. **Valid Dates:**
   * Validate that expiration dates for ingredients and recipes are set in the future, ensuring the use of fresh ingredients.
   * **Edge Case Handling:** If a past date is entered, the system should automatically flag the entry as invalid and request a correction.
4. **Role-Based Access Control (RBAC):**
   * Assign and enforce roles for users (e.g., Owner, Member, Guest), controlling their actions within the system.
   * **Edge Case Handling:** If a user attempts an action outside their assigned role, the system should deny the action and log the attempt for security auditing.
5. **Content Moderation:**
   * Implement an approval workflow for recipes and reviews, ensuring that only high-quality content is visible to users.
   * **Edge Case Handling:** If content is flagged as inappropriate, it should be reviewed by a Moderator, and if necessary, rejected or edited before publication.
6. **Ingredient Expiry Check:**
   * Automatically flag ingredients as expired if their expiration date has passed, using the is\_expired field.
   * **Edge Case Handling:** Expired ingredients should not be removed but marked, allowing users to manage or discard them based on the system’s recommendations.

#### **Logical Joins and Relationships**

1. **Household Ingredients:**
   * Join Households, Household\_Ingredients, and Ingredients to retrieve and manage household inventories.
   * **Scenario:** A user views their household’s current inventory, including ingredient quantities and expiration dates.
2. **Store Prices:**
   * Join Stores, Ingredient\_Prices, and Ingredients to track and compare prices across different stores.
   * **Scenario:** A user checks the current prices of ingredients across multiple stores to find the best deal.
3. **Recipe Ingredients:**
   * Join Recipes, Recipe\_Ingredients, and Ingredients to manage and update recipes.
   * **Scenario:** When a recipe is viewed, the system fetches all related ingredients and their quantities, allowing the user to verify if they have everything needed.
4. **User-Recipe Interaction:**
   * Join Users, User\_Recipe\_History, and Recipes to track which recipes users have cooked.
   * **Scenario:** A user reviews their cooking history to find recipes they’ve tried in the past and decide what to cook next.
5. **User Ratings:**
   * Join Users, User\_Ratings, and Recipes to manage and display user feedback on recipes.
   * **Scenario:** A user checks ratings and reviews for a recipe before deciding to cook it, ensuring it meets their expectations.

#### **System Capabilities and Constraints**

### **What the System Can Do:**

1. **Manage Users and Roles:**
   * Create, update, and delete users, assign roles, and enforce RBAC to control access.
   * **Details:** Users can be assigned roles such as Guest, Member, Owner, Moderator, or Administrator, each with specific permissions.
2. **Track Ingredients:**
   * Manage household inventories, track expiration dates, and notify users of expiring items.
   * **Details:** The system automatically flags expired ingredients and provides reminders for upcoming expirations.
3. **Moderate and Approve Content:**
   * Manage an approval workflow for user-generated content, ensuring quality and accuracy.
   * **Details:** Content submitted by users, such as recipes and reviews, is moderated by higher roles before being published.
4. **Fetch and Store Prices:**
   * Track ingredient prices across multiple stores, providing users with price comparisons.
   * **Details:** The system allows users to view price histories and make informed decisions on where to purchase ingredients.
5. **Integrate with External APIs:**
   * Fetch cooking videos from YouTube and link them to recipes.
   * **Details:** Recipes can include video tutorials, enhancing the user experience by providing step-by-step visual instructions.

### **What the System Cannot Do:**

1. **Auto-Approve Content:**
   * Content such as recipes and reviews must be manually approved by authorized users before becoming public.
   * **Details:** This ensures that all content meets quality standards and adheres to community guidelines.
2. **Process Invalid Data:**
   * The system cannot handle invalid or corrupted data entries, such as negative ingredient quantities or past expiration dates.
   * **Details:** Input validation is enforced at the data entry point to prevent such data from being saved in the database.
3. **Override RBAC:**
   * Users cannot perform actions outside their assigned roles, ensuring that sensitive operations are restricted.
   * **Details:** For example, a Guest user cannot modify household data or access restricted content meant for higher roles.

### Conclusion

This document provides a comprehensive and detailed overview of the Smart Kitchen Helper’s SQL and NoSQL database setup, along with logical rules, use case scenarios, access control details, and system capabilities. It serves as a professional design document for developers, architects, and stakeholders involved in the project, ensuring that all data is well-organized, accessible, and logically managed. The system is designed to handle current requirements and future scalability needs, providing a robust backend for the application’s features.

### Explanation:

* **Detailed Use Case Scenarios:** This section outlines specific scenarios in which the system’s features are used, providing practical examples of how the system operates in real-world situations.
* **Access Control Details:** The roles and permissions are clearly defined, ensuring that each user interacts with the system in a secure and controlled manner. The scenarios demonstrate how the system enforces these roles, maintaining data integrity and security.
* **Logical Validations, Joins, and Constraints:** Detailed examples and explanations ensure that the data is handled correctly, avoiding common pitfalls such as data corruption or unauthorized access.

This enhanced document provides a comprehensive and thorough guide for implementing the Smart Kitchen Helper’s SQL and NoSQL databases, ensuring that the system is both robust and scalable.