A Project Report

on

Wonka Labs
Experiments in Food Discovery

Presented By

Pratik Pundlik Sayanekar

(pratik.sayanekar@ucdconnect.ie)

Table of Contents

List of Figures

1. Introduction

The project visions to design an efficient data model that is loosely coupled and highly cohesive to manage food innovations in multiple culinary realms of *Wonka Labs.* Being a creator of the world's most delicious candies, *Wonka Labs* is known for whimsical experiments on food products. This modus operandi makes Wonka Labs one of the finest candy producers. Now Wonka is planning to extend its services beyond the children's market by introducing pizzas, cocktails, and wines. To embrace this new inning, Wonka Labs envisioned building a database schema that manages the innovation pipeline of experiments in food discovery.

The experiments in food discovery backed by DBMS operations such as suggesting the name-lets that are based on the ingredients used in pizza and cocktails help the marketing team to finalize the innovative names for each product to catch the buyer's attention. To upscale the baked offerings such as pizza, Wonka Labs has a collection of fine wines from around the world. As per the domain knowledge of ingredients used in wine and pizza, we have a 'pairing' attribute of wine that complements the 'restrictions' of pizzas. Using a well-normalized database model, and calculating a numerical entity that defines the confidence or weight of each mapped wine and pizza help to prioritize the recommended wine options.

The database model is designed to embrace the requirements of visionary Wonka Labs. The experiments of pizza and cocktails with several available ingredients make the innovative final product. In the development phase, multiple associated name-lets and wine suggestions are generated and the marketing team is responsible to approve one of the striking names for the product. In the production phase, the newly created pizzas and cocktails are released in production and ready to be served.

This type of database model can be utilized for food-making chains with minor modifications as per the requirements. There are several operations such as suggesting impressive name-lets and recommending compatible servings that can be handled by the database. By implementing some complex algorithms, the compatibility can be represented in numeric. In this report, we present an algorithm based on the matching attribute of two products for a recommendation. This highly scalable database is implemented by considering several database design approaches that satisfy the third normal form.

2. Database Plan: A Schematic View

This section describes the detailed illustration of the database that is intended to be designed for the application.

The database schema is nothing but the blueprint to store a large amount of data so that it can be queried efficiently as per the application requirements. It is a strategy to construct the relationship between the logical entities in the data. There are several approaches to design a database schema, Entity-Relationship model is one of the finest strategies where the data model is divided into entities and relationships. Entities are classes or real-time objects and these objects can be mapped together with multiple relationships such as one to one, one to many, many to one, and many to many.

As discussed in the white paper, the identified prime entities are as follows,

i. Pizza
ii. Cocktail
iii. Wine

We will discuss the attributes related to each of the above entities in detail, as per the relationship dependency we will further segregate the correlated attributes into smaller derived entities. In this way, high cohesion and low coupling in a database management system can be achieved.

i. **The Pizza Oven:**

   Wonka Labs sells the series of traditional pizzas and also plans to release some of their own invented pizza which uses specified toppings and base combinations. Every pizza has multiple interesting name suggestions based on the elements used in the pizza. Successful pizzas will move to production with an apt name selected from provided recommendations. The data is stored in development and production tables. So Let's analyze what all attributes are there related to Pizza.

As per the provided data, we can assume that a pizza is made up of 5 integral categories of elements as meat, vegetables, sauce, cheese, and base. One pizza can have multiple elements of each of the above categories. The namelets are related to the elements used in the pizza and one element may have multiple namelets. These namelets are further divided as first_word and second_word. Elements also have some restrictions as properties that can be used to map the respective wine. As a pizza is made up of multiple elements, there will be multiple namelets and restrictions associated with pizza and these attributes are useful to provide name suggestions and complimentary wine suggestions.

The below figure illustrated the high level of view of all entities and associated attributes. We will break down these attributes further as per the relationship.

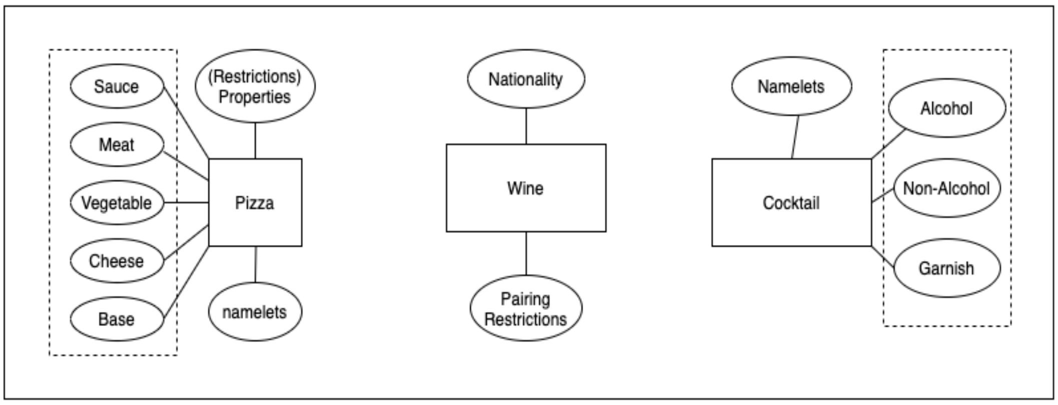


Figure 1: High-level view of Entities and associated attribute

ii. The Cocktail Lounge

The cocktails are treated the same as the pizza, cocktails can be prepared out of a list of ingredients such as alcoholic beverages, non-alcoholic beverages, and garnishes. We can have a glass as well as ice as elements but these categories are not considered in the initial stage. We might add these categories in the later stage of database design to see the impact of the addition of new elements in an existing database. Like pizza, these ingredients may have one or multiple striking namelets associated with them. The task here is to allow Wonka experts to create new cocktails with provided ingredients and suggest multiple namelets for a newly created cocktail as per the ingredients used in it using database functionalities. Cocktails also follow the development and production lifecycle as stated for pizza.

iii. **The Wine Cellar**

To upsell the food offerings, Wonka Labs have a collection of world's finest wines. There are two attributes that are associated with wine, that are *nationality* and *pairing* specifications. One wine may have any number of pairing specifications and in regards with nationality wine may have one or more nationalities associated with it. As per the provided data, we witnessed some wines with no nationality information. The goal here to provide multiple wine suggestions that goes well with pizza as per the nationality preference and pairing specifications, these attribute can be mapped to the *restrictions* of *elements* used in the pizzas.

We have explored the each domain of Wonka's offerings, let's build a relationship between these entities. As per (Atzeni, 1999), 'Relationships represent logical links between two or more entities.'

In relational databases, the relationship among entities can be of 4 type as follows,

a) One to One Relationship
b) One to Many Relationship
c) Many to One Relationship
d) Many to Many Relationship

Relationship between entities play a pivotal role in database schema design. The appropriate design improves the query efficiency. Relationships can be defined by using primary key and foreign key constraints.

The below E-R diagram shows the schematic view of the database design.

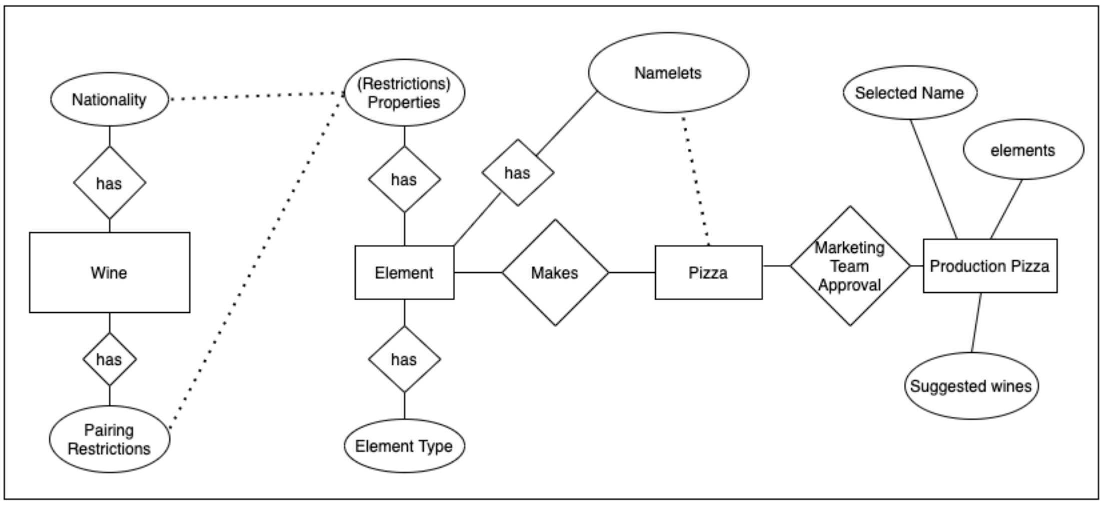


Figure 2: E-R Diagram (Schematic View)

Let's begin with the wine entity, wine has related attributes such as *nationality* and *pairings*. As per the provided data, we have analyzed that this is a one-to-many relationship, where one wine may have one or more *nationalities* and *pairings*. As we have to find pair of wine that goes with the respective pizza, the nationality and pairing attribute can be mapped with the *restriction* attribute of pizza. We will discuss this relationship and the logic behind mapping later.

The pizza entity is made up of elements i.e., ingredients present in pizza. These elements are categorized as meat, vegetable, cheese, sauce, and base. One element belongs to one *category*, so there is one to one relationship between element and category. Restrictions are nothing but the properties related to pizza, the values of this attribute can be mapped with the pairings and nationality of wine attribute. One element may have multiple restrictions. So, element and restriction share one to many relationships. Each element has some predefined namelets, defined as before and after in provided dataset, we have renamed these attributes to the *first_word* and *second_word* for simplicity. Every element has one or many *first_word* and *second_word*. Namelets can be generated by combining both attributes and all possible combinations are considered as valid namelets. We have also observed multiple first_word and second_word pairs for each element. Many to many relationships are observed here, i.e., every word of the first_word attribute can be mapped with every word of the second_word attribute of the respective element. As pizza is made up of multiple elements, pizza can have multiple such namelets. So, like pizza and elements, pizza and namelets shares the one-to-many relationship. So ultimately, we can conclude that one pizza may have any number of elements, and accordingly, any number of restrictions and namelets are associated with the respective pizza.

As per the pizza release process defined in the white paper, database operations are supposed to provide namelets and wine suggestions for experimental pizzas in development. The namelets can be suggested as per the usage of ingredients in the pizza, we assumed that the marketing team will choose one of the suggested namelets that catches the attention of the buyer and the respective pizza will be released in production. Once the pizza is moved to production, the pizza will have only one catchy name selected by a marketing team. We can consider this selected name as one of the entities as production pizza, it has few more related attributes such as series of suggested wines, we can also include the list of elements used to prepare this pizza for simplicity.

Regarding pizza and wine, the relation can be established by using *restrictions* of pizza with *nationality* and *pairings* of wine. As per provided data, we can infer that the restrictions contain the nationality of pizza and pairing values, for example, 'Manchego cheese' has the restriction as Spanish and that can be mapped with the nationality of pizza for a match. Similarly, in some cases, restrictions also contain the pairing value of pizza, for example, 'Pinot Grigio' has 'seafood' as one of the pairings, and many elements have this as restrictions, so we should map pairings and restriction together for wine suggestions.

we also witnessed the element name of pizza in the pairing attribute of wine, for example, 'olives' is found as pairing for some wines, and this value is not as a restriction of any element, instead, it is one of the word of the element itself. To make the wine suggestion more precise we have to consider this scenario as well.

In this way, all the relationships have been defined successfully and we can move forward to design a schema, while designing schema we have explored several combinations of approaches, the goal here is to create a separate table that holds a single relationship to achieve the normalization process. The detailed view of a database is described as an EER diagram in the next section.

3. Database Structure: A Normalized View

In this section, we describe the main tables of designed schema. Role of each table and the thought behind the normalization is elaborated in detail as below.

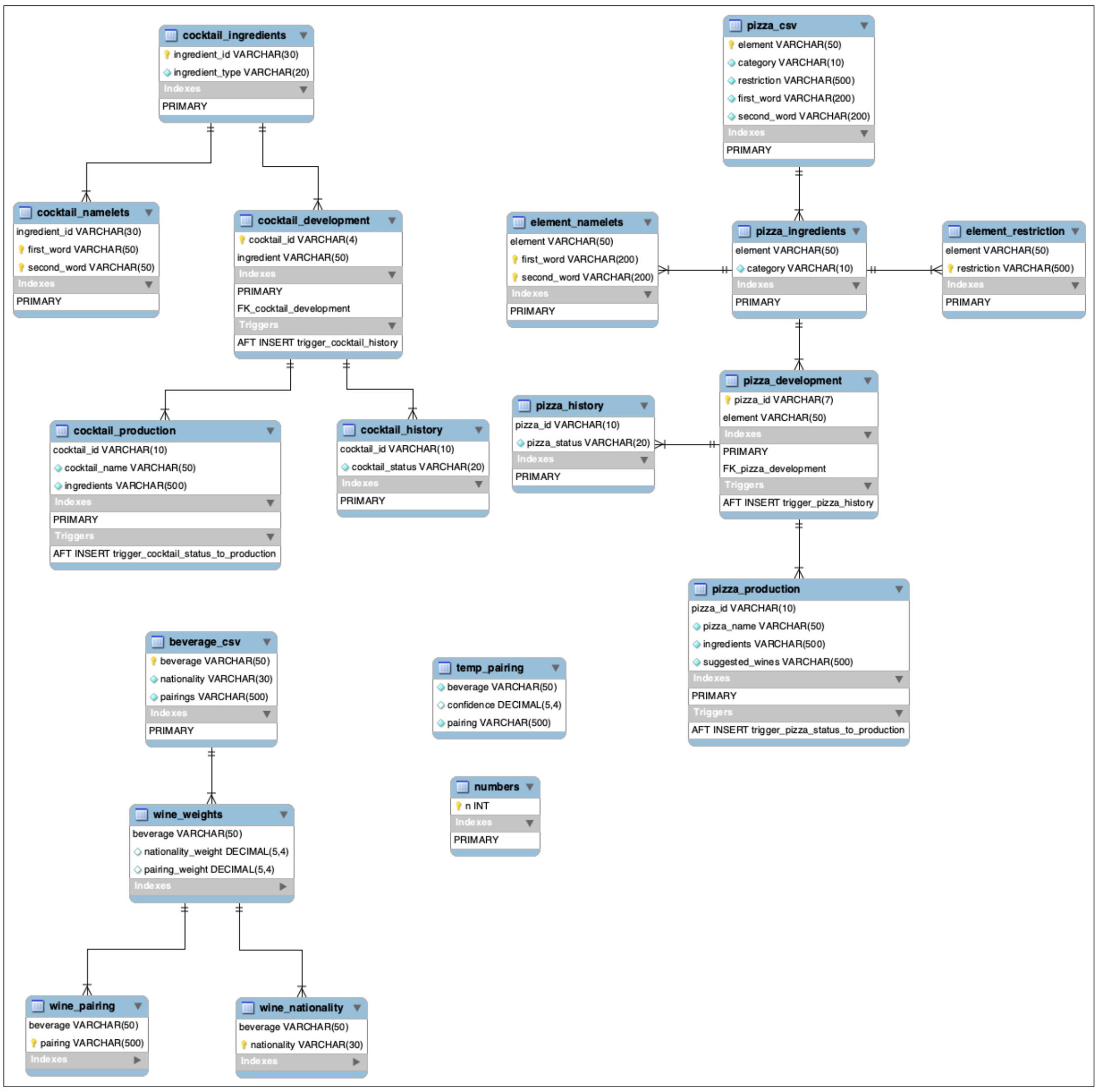


Figure 3: EER Diagram (Database Schema)

Database schema design initiated with the loading of the CSV file of pizza and beverages. As per the ER diagram described in the previous section, we have created this schema. Cocktails, Pizzas, and Wines are the main pillars of this schema. Each of these pillars is backed by development and production tables. As mentioned in a white paper, development tables are used for query purposes and the production tables are intended for a marketing team, so the structure is designed accordingly.

Before diving deep into database schema and strategy, let's discuss what exactly is the normalization and what all normalization forms need to be satisfied. Normalization is a strategy to organize the data into a database that reduces the redundancy in data and eliminates anomalies of several database operations such as insertion, update, and deletion. It basically divides the complex tables into smaller tables with respect to a direct relationship between attributes. The smaller tables are further connected with each other through primary and foreign key constraints. We have studied various normalization forms and they are as follows,

i. **1NF: First Normal Form**

   In 1NF, the table contains an atomic value, i.e., an attribute of a table can have only one atomic value related to it. It should not have multiple values associated with it. In provided data, few comma-separated values are observed.

ii. **2NF: Second Normal Form**

   In 2NF, the table has to follow 1NF and all non-key attributes should directly be dependent on the primary key. If a non-key attribute is dependent on any other key that is not primary, a table should be either altered with a composite primary key or refactored into separate table to be in 2NF form.

iii. **3NF: Third Normal Form**

   In 3NF, the table has to follow 2NF with no transitive dependency between the attributes. Transitive dependency means X is dependent on Y while Y is not dependent on X and Y has a relationship with Z then we can say that X has a transitive dependency on Z. In order to make the table or relation in 3NF, as per the relationship and dependency the main table can be divided into multiple tables.

iv. **BCNF: Boyce Codd Normal Form**

   This can be also referred to as 3.5NF, this is an advanced version of 3NF that has no multivalued dependency. Multivalued dependency means there is more than one independent attribute in the same table. We will discuss this form in detail with respect to our data model.

Let’s explore the data model created for Wonka with respect to normalization. We will discuss each realm of Wonka labs one by one as follows,

A. **Pizza:**

We analyzed the relationships between several attributes attached to the pizza entity. Let’s normalize this entity as per the normalization process as discussed above,

Element	Category	Restrictions	First_word	Second_word
Chicken Tikka	Meat	meat, chicken, spicy, Indian, Asian	Bollywood, Taj, Maharaja's	Palace
raised crust	Base	thick, gluten	Puffy	Elevator, High Rise

Table: pizza_csv

The pizza entity has a *'category'* like an atomic attribute, but the rest of the attributes are not atomic as it holds multiple values associated with each row that makes this table not even satisfy the first normal form.

To normalize the above data to the first normal form, we have to maintain a separate row for each value of non-atomic attribute such as ‘Restriction’.

Element	Restrictions
Chicken Tikka	meat
Chicken Tikka	chicken
Chicken Tikka	spicy
Chicken Tikka	Indian
Chicken Tikka	Asian
raised crust	thick
raised crust	gluten

Table: elements_restriction

The *first_word* and *second_word* attributes are directly depending on the *element* at the same time *first_word* and *second_word* together form a pair of namelets, so as per the second form of normalization we will create a table with a composite primary key of *element*, *first_word* and *Second_word* as below.

Element	**First_word**	**Second_word**
Chicken Tikka	Bollywood	Palace
Chicken Tikka	Taj	Palace
Chicken Tikka	Maharaja's	Palace
raised crust	Puffy	Elevator
raised crust	Puffy	High Rise

Table: element_namelets

Now we have separate tables that follow 2 levels of normalization. It's a time to introduce pizza to the schema, As described in the E-R diagram. Pizza is dependent on elements, but elements do not depend on pizza, they are just a part of the pizza, however, elements have some restrictions and namelets associated with it. So, since pizza depends on elements, pizza has some restrictions and namelets as per the elements used to make the pizza. Here we can confirm the transitive dependency and as mentioned in the 3NF definition, to make the table third level normalized, we will create another table that stores the pizza and elements with composite primary as below.

Pizza_id	**Elements**
PZ001	Chicken Tikka
PZ001	baby corn
PZ001	Alfredo sauce
PZ001	Jarlsberg cheese

Table: pizza_development

After designing a normalized schema till the 3rd level of normalization, to make the data model more efficient and robust for future requirements, we can look for any multivalued dependency as mentioned in BCNF form. As per our pizza data, multivalued dependency has been observed in restrictions and namelets. Both restrictions and namelets are not dependent on each other but they are dependent on the elements. So, to make the data model Boyce Codd normalized we have to maintain this relationship in a separate table.

The detailed description of normalized tables related to the pizza domain is as follows,

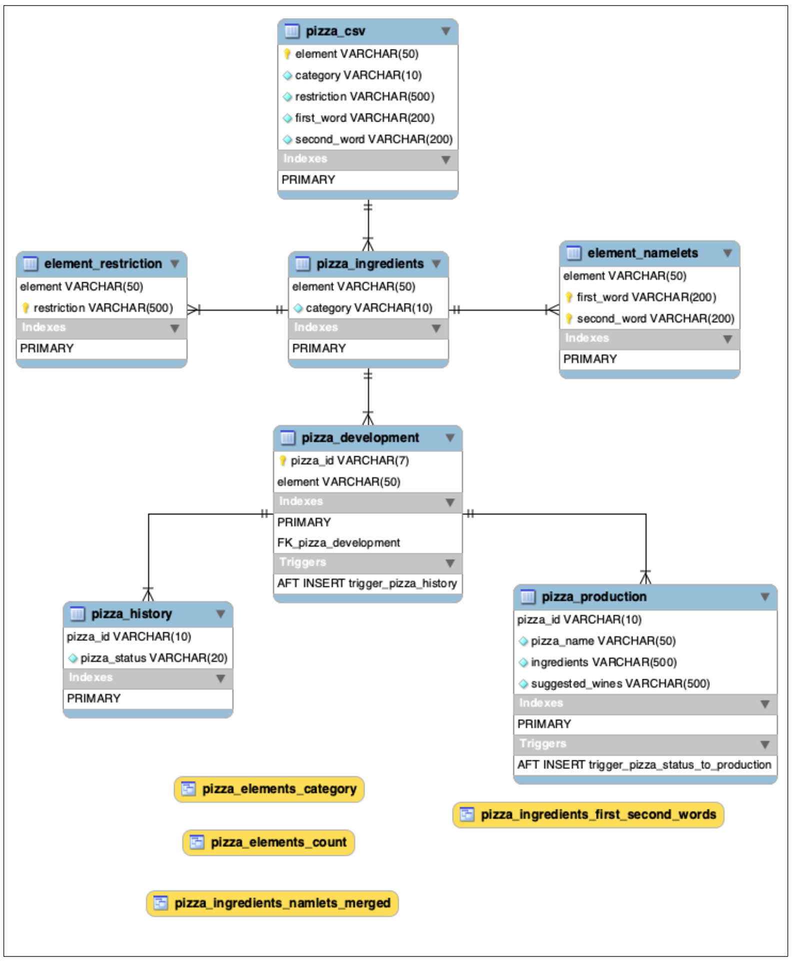


Figure 4: Detailed ER model for Pizza

B. Cocktail

We have analyzed the relationships between the several attributes available in cocktail entity while discussing the ER diagram. As per Wonka's white paper, cocktail entity resembles with the pizza in many ways. Cocktail has ingredients along with category and namelets associated with each ingredient as separate attribute such as *first_word*, *second_word* as below,

Ingredient	Category	First_word	Second_word
Apricot Brandy	Alcoholic	Casio, Desert	Brandy, Rose
Grenadine Syrup	Non-Alcoholic		
raspberries	Garnish	tropical	Jam, berries
Cream	Non-Alcoholic	milky	

Table: All attribute of cocktails

As mentioned in the white paper, ingredients may have any number of names and it could be the condition where an ingredient does not have *first_name* and *last_name*. So, considering all these scenarios we have generated a data for cocktails as described in the above source table.

To normalize the cocktail table, we will follow the same approach as discussed in case of pizza. Let's begin with first normal form, this can be achieved by converting non-atomic values as a separate row associated to each ingredient.

First step is to create a master table, to store all ingredients and their category as atomic values. This table has one entry of each ingredient and associated data type. As this is primary master table for ingredients, the unique set of ingredients are chosen to perform experiments on pizza.

Ingredient	Category
Apricot Brandy	Alcoholic
Grenadine Syrup	Non-Alcoholic
raspberries	Garnish
Cream	Non-Alcoholic

Table: cocktail_ingredients

In the above tables of all attributes, both *first_word* and *second_word* is related to ingredients and as per business logic both these words form a namelets so the combined entity would be in 2NF form, as each non key attribute is directly dependent on the primary key.

Ingredients	First_word	Second_word
Apricot Brandy	Casio	Brandy
Apricot Brandy	Casio	Rose
Apricot Brandy	Desert	Brandy
Apricot Brandy	Desert	Rose
Grenadine Syrup		
raspberries	tropical	Jam
raspberries	tropical	berries
Cream	milky	

Table: cocktail_namelets

In cocktail, there is no transitive and multivalued dependency is observed as we are dealing with only ingredients and namelets. Hence, we can conclude the normalization process for cocktail here.

C. Wine

Wine entity has attributes such as beverage, nationality, and pairing. We have already explored the relationships among these attributes. The data is imported as per the provided CSV to design a precise relationship model. Let's analyze the wine data first and strategize the normalization process.

Beverage	Nationality	Pairings
Diet Coke	Bollywood	American, sweet
Zinfandel Rosé	Californian, American	chicken, Piri Piri, Parma, spicy, seafood, Puttenesca, aioli, garlic, sardines, avocado, crab
Malbec	Maharaja's	duck, venison, salmon, tuna, ham, spicy, ginger, salty, berries, Asian, pork, pastrami, cherry, Cheddar

Table: beverage_csv

Beverages and Nationality can be uniquely identified as one row. Wine entity shows one to many relationships with Nationality and Pairings. So, to satisfy 2NF constraints these table are separated as below.

Beverage	**Pairings**
Diet Coke	American
Diet Coke	sweet
Zinfandel Rosé	chicken
Zinfandel Rosé	Piri Piri
Zinfandel Rosé	Parma
Zinfandel Rosé	Spicy
Zinfandel Rosé	Seafood
Zinfandel Rosé	Puttenesca
Zinfandel Rosé	aioli
Zinfandel Rosé	……..

Table: wine pairing

To convert the wine data model into 1st normalization form, we have to assign a separate row for each nationality value associated with wine. A similar operation should be performed on the pairings attribute. The new entity with the composite primary key is as described below,

Beverage	**Nationality**
Diet Coke	Bollywood
Zinfandel Rosé	Californian
Zinfandel Rosé	American
Malbec	Maharaja's

Table: wine_nationality

For wine, to map the nationality and pairings to the experimental pizza, we have implemented an algorithm that calculates the number of pairings and nationality of respective beverage and gets the numerical entity as the weight of each pair value present in a beverage that can be utilized as confidence behind every mapping. This approach takes all the pairings into consideration that introduce the precision in preferences as compared to suggesting random wine on only one pair and restriction mapping.

Beverage	**Nationality Weight**	**Pairing Weight**
Diet Coke	1	0.5
Zinfandel Rosé	0.5	0.0909
Malbec	1	0.0714

Table: wine_weights

This model has no transitive multivalued dependency present at the moment and the data model is well normalized for in further advancements.
The detailed E-R model is as shown below,

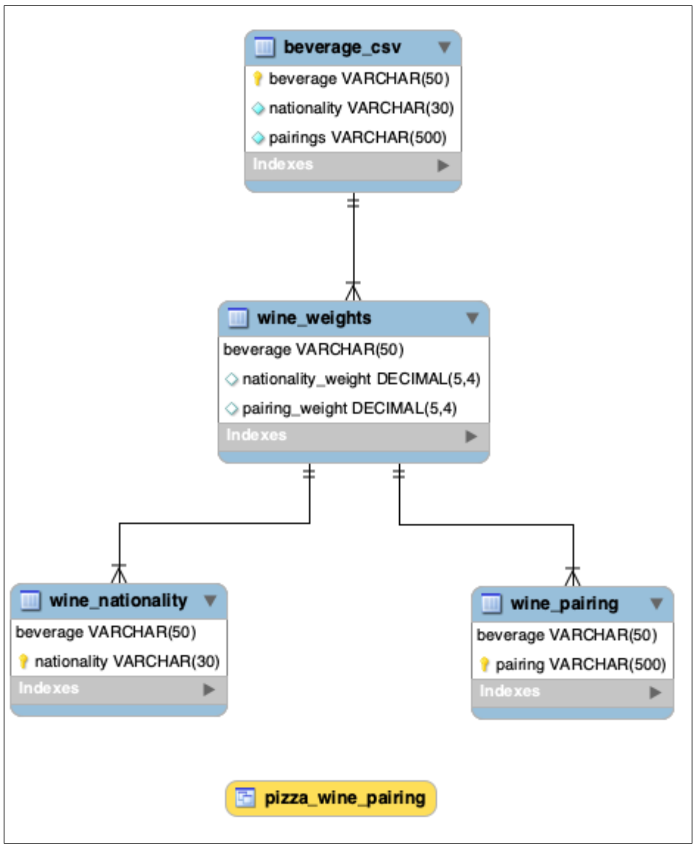


Figure 5: Detailed ER model for wine

4. Database Views

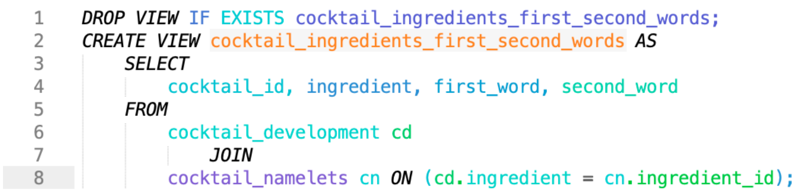
Views are virtual tables, unlike table view doesn't store the data rather it is a result of complex queries. In a relational database, the data is sparsely distributed in multiple tables, so to get the abstract information of multiple tables views can be created. Views provide accessibility to query the data just similar to tables. Views are updated with incremental data, for example, we have started this implementation with few numbers of pizzas, cocktails, and wines and as we move forward and when this database is live on production, views provide an abstraction of data without writing a complex query every time.

In real-world applications, views are influential to integrate several cross-functional systems as it provides the abstraction by exposing only results that are retrieved by multiple domain-specific tables. (Atzeni, 1999) The author of this book has given a prompt clarification about views as 'query language expression'.

This database is accessed by the marketing as well as a development team. So, in order to achieve abstraction of multiple normalized tables, we have created a couple of views as below,

I. cocktail_ingredients_first_second_words

This view provides the first word and second word for cocktails as per the ingredients used in the cocktail using the join query in *cocktail_development* and *cocktail_name*. This view may have multiple rows associated with each cocktail as suggested namelets.

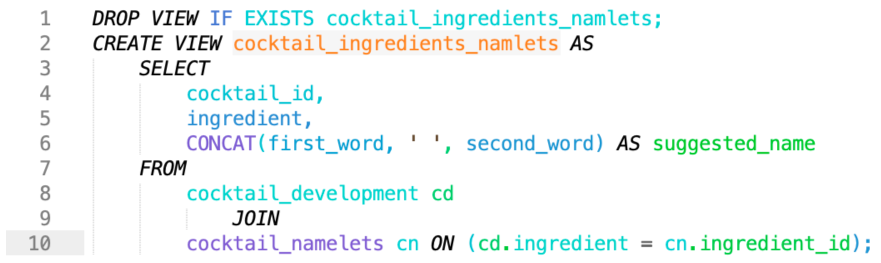


```sql
DROP VIEW IF EXISTS cocktail_ingredients_first_second_words;
CREATE VIEW cocktail_ingredients_first_second_words AS
    SELECT
        cocktail_id, ingredient, first_word, second_word
    FROM
        cocktail_development cd
            JOIN
        cocktail_namelets cn ON (cd.ingredient = cn.ingredient_id);
```

Code Snippet: View - cocktail_ingredients_first_second_words

As these views are created to suggest the namelets for a marketing team, few more versions are created to make ease. To view the respective ingredient for suggested namelets we have created few additional views.

cocktail_ingredients_namlets, here first name and last name are concatenated form along with ingredient name, for the ingredients with no first or second name assigned will show empty string.

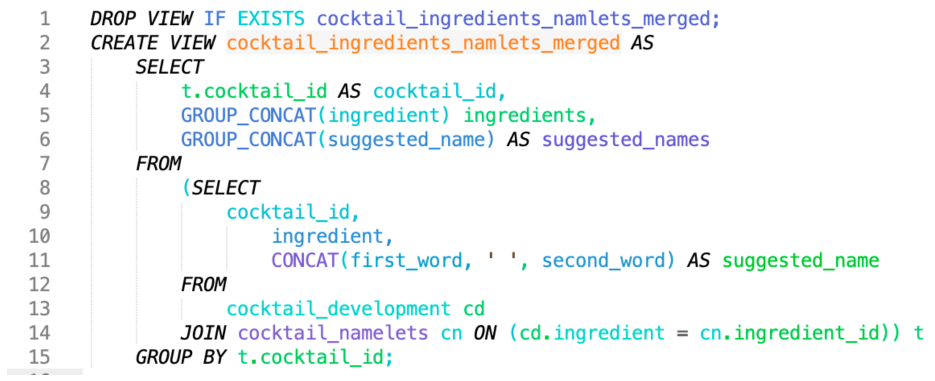


```sql
DROP VIEW IF EXISTS cocktail_ingredients_namlets;
CREATE VIEW cocktail_ingredients_namlets AS
    SELECT
        cocktail_id,
        ingredient,
        CONCAT(first_word, ' ', second_word) AS suggested_name
    FROM
        cocktail_development cd
            JOIN
        cocktail_namelets cn ON (cd.ingredient = cn.ingredient_id);
```

Code Snippet: View - cocktail_ingredients_namlets

The above views have multiple rows associated with a single cocktail, to give more abstract information of the above data, we can use group_caoncat functionality which provides a comma-separated list of ingredients and suggested namelets for each cocktail.

Cocktail_ingredients_namelets_merged has only one row associated with each cocktail with all ingredients and suggested names.



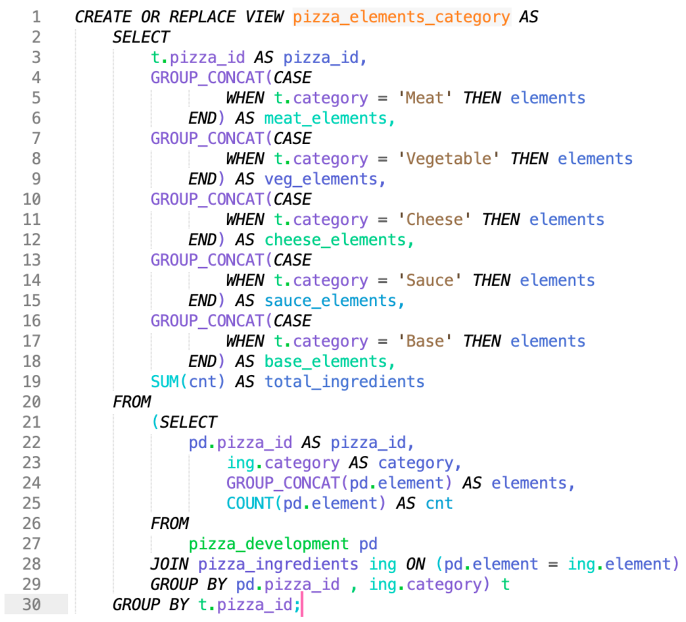
```sql
DROP VIEW IF EXISTS cocktail_ingredients_namlets_merged;
CREATE VIEW cocktail_ingredients_namlets_merged AS
    SELECT
        t.cocktail_id AS cocktail_id,
        GROUP_CONCAT(ingredient) ingredients,
        GROUP_CONCAT(suggested_name) AS suggested_names
    FROM
        (SELECT
            cocktail_id,
                ingredient,
                CONCAT(first_word, ' ', second_word) AS suggested_name
        FROM
            cocktail_development cd
        JOIN cocktail_namelets cn ON (cd.ingredient = cn.ingredient_id)) t
    GROUP BY t.cocktail_id;
```

Code Snippet: View- cocktail_ingredients_namelets_merged

II. pizza_elements_category

This view provides the details about each experimental pizza on development, as per Wonka lab’s initiative of creating their own pizza offerings by selecting a wide range of elements, this view identifies the category of each element and stores the elements of each category. This view also provides the numerical count of elements used in the respective pizza.

The query is as below,



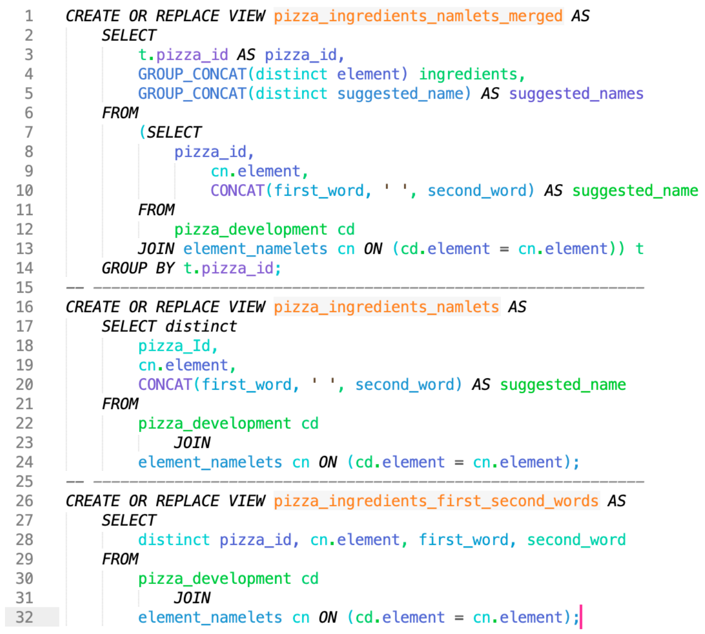
```sql
CREATE OR REPLACE VIEW pizza_elements_category AS
    SELECT
        t.pizza_id AS pizza_id,
        GROUP_CONCAT(CASE
                WHEN t.category = 'Meat' THEN elements
            END) AS meat_elements,
        GROUP_CONCAT(CASE
                WHEN t.category = 'Vegetable' THEN elements
            END) AS veg_elements,
        GROUP_CONCAT(CASE
                WHEN t.category = 'Cheese' THEN elements
            END) AS cheese_elements,
        GROUP_CONCAT(CASE
                WHEN t.category = 'Sauce' THEN elements
            END) AS sauce_elements,
        GROUP_CONCAT(CASE
                WHEN t.category = 'Base' THEN elements
            END) AS base_elements,
        SUM(cnt) AS total_ingredients
    FROM
        (SELECT
            pd.pizza_id AS pizza_id,
                ing.category AS category,
                GROUP_CONCAT(pd.element) AS elements,
                COUNT(pd.element) AS cnt
        FROM
            pizza_development pd
        JOIN pizza_ingredients ing ON (pd.element = ing.element)
        GROUP BY pd.pizza_id , ing.category) t
    GROUP BY t.pizza_id;
```

Code Snippet: View - pizza_element_category

Here, there are 2 queries, the inner query retrieves all grouped elements as per category that is used in the respective pizza and the count of total elements. The outer query converts the multiple rows associated with each pizza into a single row for one pizza.

III. pizza_ingredients_first_second_words

Similar to cocktail, we have created below views for suggested namelets for pizza, these versions are similar to a cocktail as discussed above.

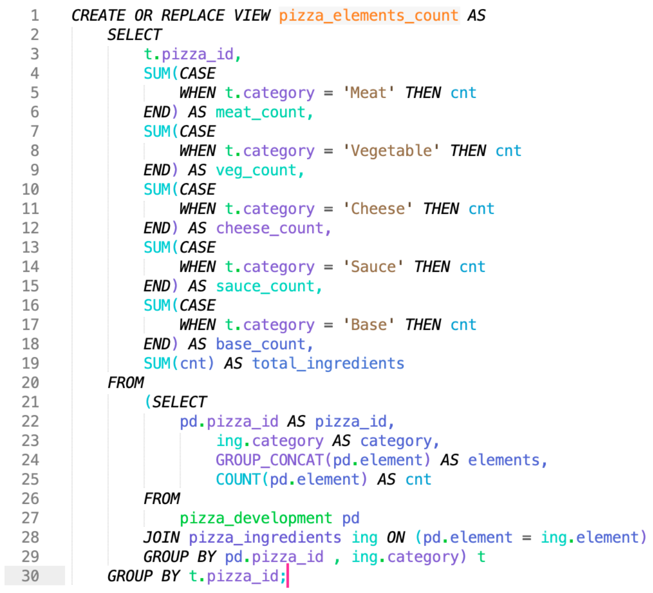


```sql
CREATE OR REPLACE VIEW pizza_ingredients_namlets_merged AS
    SELECT
        t.pizza_id AS pizza_id,
        GROUP_CONCAT(distinct element) ingredients,
        GROUP_CONCAT(distinct suggested_name) AS suggested_names
    FROM
        (SELECT
            pizza_id,
                cn.element,
                CONCAT(first_word, ' ', second_word) AS suggested_name
        FROM
            pizza_development cd
        JOIN element_namelets cn ON (cd.element = cn.element)) t
    GROUP BY t.pizza_id;
-- ------------------------------------------------------------
CREATE OR REPLACE VIEW pizza_ingredients_namlets AS
    SELECT distinct
        pizza_Id,
        cn.element,
        CONCAT(first_word, ' ', second_word) AS suggested_name
    FROM
        pizza_development cd
            JOIN
        element_namelets cn ON (cd.element = cn.element);
-- ------------------------------------------------------------
CREATE OR REPLACE VIEW pizza_ingredients_first_second_words AS
    SELECT
        distinct pizza_id, cn.element, first_word, second_word
    FROM
        pizza_development cd
            JOIN
        element_namelets cn ON (cd.element = cn.element);
```

Code Snippet: View - pizza_element_category

IV. pizza_elements_count

As pizza contains several categories this view provides the count of elements used in each category of pizza. This view can be useful to limit the count of ingredients for a specific category.

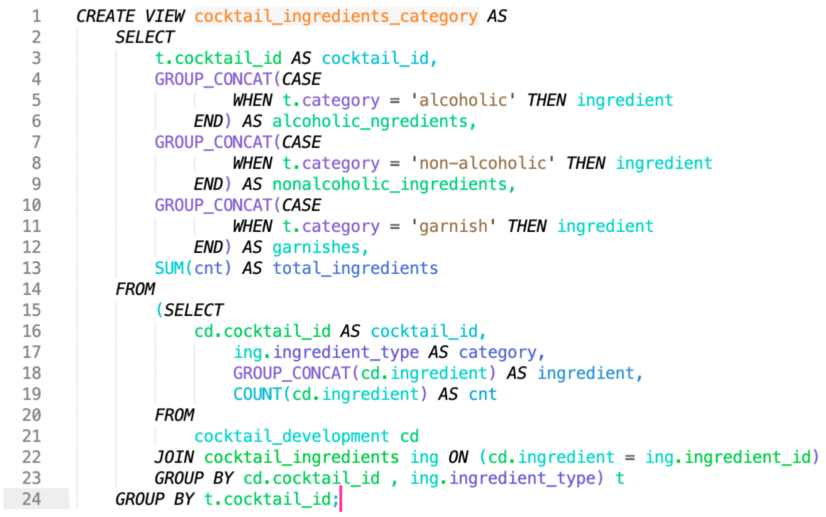


```sql
CREATE OR REPLACE VIEW pizza_elements_count AS
    SELECT
        t.pizza_id,
        SUM(CASE
            WHEN t.category = 'Meat' THEN cnt
        END) AS meat_count,
        SUM(CASE
            WHEN t.category = 'Vegetable' THEN cnt
        END) AS veg_count,
        SUM(CASE
            WHEN t.category = 'Cheese' THEN cnt
        END) AS cheese_count,
        SUM(CASE
            WHEN t.category = 'Sauce' THEN cnt
        END) AS sauce_count,
        SUM(CASE
            WHEN t.category = 'Base' THEN cnt
        END) AS base_count,
        SUM(cnt) AS total_ingredients
    FROM
        (SELECT
            pd.pizza_id AS pizza_id,
                ing.category AS category,
                GROUP_CONCAT(pd.element) AS elements,
                COUNT(pd.element) AS cnt
        FROM
            pizza_development pd
        JOIN pizza_ingredients ing ON (pd.element = ing.element)
        GROUP BY pd.pizza_id , ing.category) t
    GROUP BY t.pizza_id;
```

Code Snippet: View - cocktail_ingredients_first_second_words

Here, there are 2 queries, the inner query retrieves the count of all elements as per category that is used in the respective pizza and the total count. The outer query converts the multiple rows associated with each pizza into a single row for one pizza.

V. Cocktail_ingredients_category



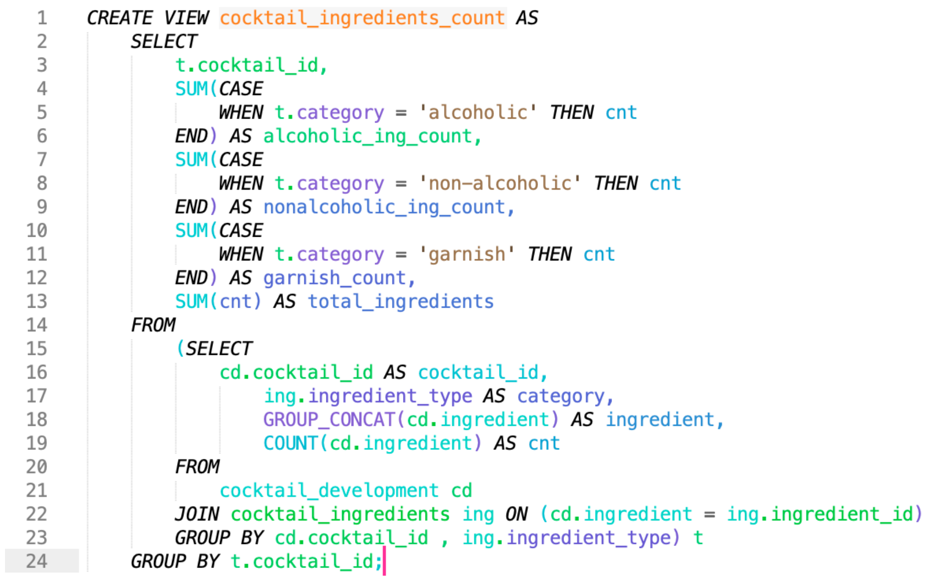
```sql
CREATE VIEW cocktail_ingredients_category AS
    SELECT
        t.cocktail_id AS cocktail_id,
        GROUP_CONCAT(CASE
                WHEN t.category = 'alcoholic' THEN ingredient
            END) AS alcoholic_ngredients,
        GROUP_CONCAT(CASE
                WHEN t.category = 'non-alcoholic' THEN ingredient
            END) AS nonalcoholic_ingredients,
        GROUP_CONCAT(CASE
                WHEN t.category = 'garnish' THEN ingredient
            END) AS garnishes,
        SUM(cnt) AS total_ingredients
    FROM
        (SELECT
            cd.cocktail_id AS cocktail_id,
                ing.ingredient_type AS category,
                GROUP_CONCAT(cd.ingredient) AS ingredient,
                COUNT(cd.ingredient) AS cnt
        FROM
            cocktail_development cd
        JOIN cocktail_ingredients ing ON (cd.ingredient = ing.ingredient_id)
        GROUP BY cd.cocktail_id , ing.ingredient_type) t
    GROUP BY t.cocktail_id;
```

Code Snippet: View - cocktail_ingredients_first_second_words

Here, as the pizza and ingredients data model are similar in nature, we have implemented the same logic to retrieve grouped elements as per category and the count of total elements for cocktails as well.

VI. cocktail_ingredients_count

This view provides the count of each category of elements used in the respective cocktail. This view can be useful to keep a track of a number of alcoholics, non-alcoholic, and garnishes used in the cocktail. In this view, each row represents individual cocktail information.

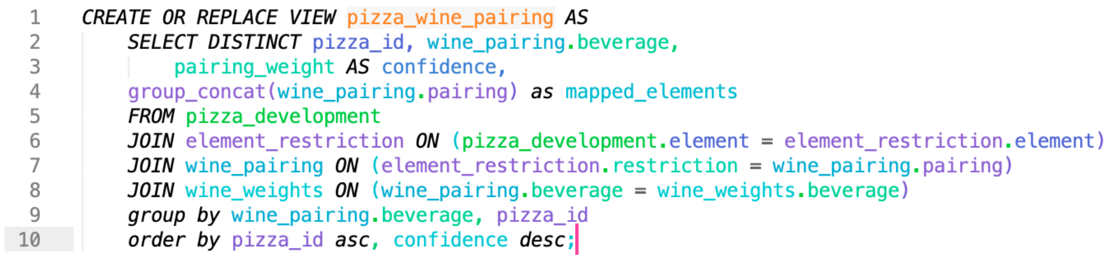


```sql
CREATE VIEW cocktail_ingredients_count AS
    SELECT
        t.cocktail_id,
        SUM(CASE
            WHEN t.category = 'alcoholic' THEN cnt
        END) AS alcoholic_ing_count,
        SUM(CASE
            WHEN t.category = 'non-alcoholic' THEN cnt
        END) AS nonalcoholic_ing_count,
        SUM(CASE
            WHEN t.category = 'garnish' THEN cnt
        END) AS garnish_count,
        SUM(cnt) AS total_ingredients
    FROM
        (SELECT
            cd.cocktail_id AS cocktail_id,
                ing.ingredient_type AS category,
                GROUP_CONCAT(cd.ingredient) AS ingredient,
                COUNT(cd.ingredient) AS cnt
        FROM
            cocktail_development cd
        JOIN cocktail_ingredients ing ON (cd.ingredient = ing.ingredient_id)
        GROUP BY cd.cocktail_id , ing.ingredient_type) t
    GROUP BY t.cocktail_id;
```

Code Snippet: View - cocktail_ingredients_first_second_words

VII. pizza_wine_pair_confidence

This view is defined to provide the complementary wine suggestions along with mapped pairs and the confidence of each mapping. This view plays an important role while selecting the preferences of suggested wine.



```sql
CREATE OR REPLACE VIEW pizza_wine_pairing AS
    SELECT DISTINCT pizza_id, wine_pairing.beverage,
        pairing_weight AS confidence,
    group_concat(wine_pairing.pairing) as mapped_elements
    FROM pizza_development
    JOIN element_restriction ON (pizza_development.element = element_restriction.element)
    JOIN wine_pairing ON (element_restriction.restriction = wine_pairing.pairing)
    JOIN wine_weights ON (wine_pairing.beverage = wine_weights.beverage)
    group by wine_pairing.beverage, pizza_id
    order by pizza_id asc, confidence desc;
```

Code Snippet: View – pizza_wine_pairing

5. Procedural Elements

To adhere to the processing pipelines of Wonka Labs, we have implemented a few stored procedures and triggers at the database level to automate the functional aspect of the application.

5.1 Stored Procedures

Stored procedures are implemented to process database-level operations to fulfill the business requirement. A stored procedure can accept the input to implement series of operations to get the results and as this is stored at the database level, in a real-time application, the stored procedure helps to distribute the computing power from application server to database server.

I. suggest_pizza_name

This store procedure accepts the *pizza_id* as input and provides suggestions of names as per the elements used in the pizza. This stored procedure is useful in the development phase to validate the suggested name functionality.

```sql
DROP PROCEDURE IF EXISTS suggest_pizza_name;
DELIMITER //
CREATE PROCEDURE suggest_pizza_name(IN pizza_id VARCHAR(7))
BEGIN
SELECT
    pizza_development.element, first_word, second_word
FROM
    pizza_development
        JOIN
    element_namelets ON (pizza_development.element = element_namelets.element)
WHERE
    pizza_development.pizza_id = pizza_id;

END //
DELIMITER ;
```

Code Snippet: Stored Procedure – suggest_pizza_name

II. suggest_cocktail_name

Similar to pizza, this stored procedure accepts the *cocktail_id* as input and suggests the name for the cocktails as per the ingredients used in the provided cocktail.

```sql
DROP PROCEDURE IF EXISTS suggest_cocktail_name;
DELIMITER //
CREATE PROCEDURE suggest_cocktail_name(IN cocktail_id VARCHAR(7))
BEGIN
SELECT
    ingredient, first_word, second_word
FROM
    cocktail_development
        JOIN
    cocktail_namelets
        ON (cocktail_development.ingredient = cocktail_namelets.ingredient_id)
WHERE
    cocktail_id = cocktail_id;
END //
DELIMITER ;
```

Code Snippet: Stored Procedure – suggest_cocktail_name

III. suggest_wine_confidence

Suggesting complimentary wines as per the elements used in the pizza is one of the Wonka lab's strategies to upscale the offerings. We have implemented several strategies to calculate the numerical value as the confidence of each suggested wine. This calculation is based on the number of matches between restriction and pairing attributes of pizza and wine entities.

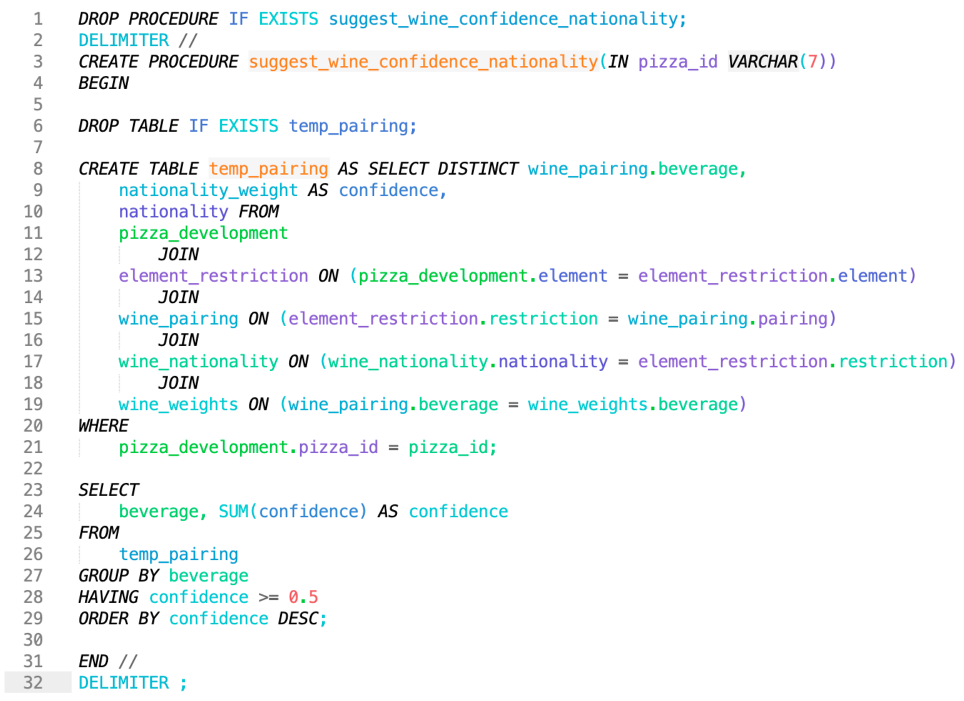
The below-stored procedure accepts the *pizza_id* as input and calculates the pairing weight as the *confidence* of each wine and filters the wine suggestions with a confidence less than *0.50*. By this, we are limiting the less relevant results. This store procedure creates a temporary table to show the desired output.

```sql
DROP PROCEDURE IF EXISTS suggest_wine_confidence;
DELIMITER //
CREATE PROCEDURE suggest_wine_confidence(IN pizza_id VARCHAR(7))
BEGIN

DROP TABLE IF EXISTS temp_pairing;

CREATE TABLE temp_pairing AS SELECT DISTINCT wine_pairing.beverage,
    pairing_weight AS confidence,
    wine_pairing.pairing FROM
    pizza_development
        JOIN
    element_restriction ON (pizza_development.element = element_restriction.element)
        JOIN
    wine_pairing ON (element_restriction.restriction = wine_pairing.pairing)
        JOIN
    wine_weights ON (wine_pairing.beverage = wine_weights.beverage)
WHERE
    pizza_development.pizza_id = pizza_id;

SELECT
    beverage, SUM(confidence) AS confidence
FROM
    temp_pairing
GROUP BY beverage
HAVING confidence >= 0.5
ORDER BY confidence DESC;

END //
DELIMITER ;
```

Code Snippet: Stored Procedure – suggest_wine_confidence

IV. suggest_wine_confidence_nationality

As per the provided data, we have observed some relationship with *nationality* while pairing wine to the experimental pizzas. In order to take *nationality* into consideration while suggesting complimentary wines, we have maintained one attribute as *nationality_weight* that signifies the number of nationalities associated with each wine, and this weight is considered as the confidence to suggest the wine. We have also filtered the wines with confidence of less than 0.5.



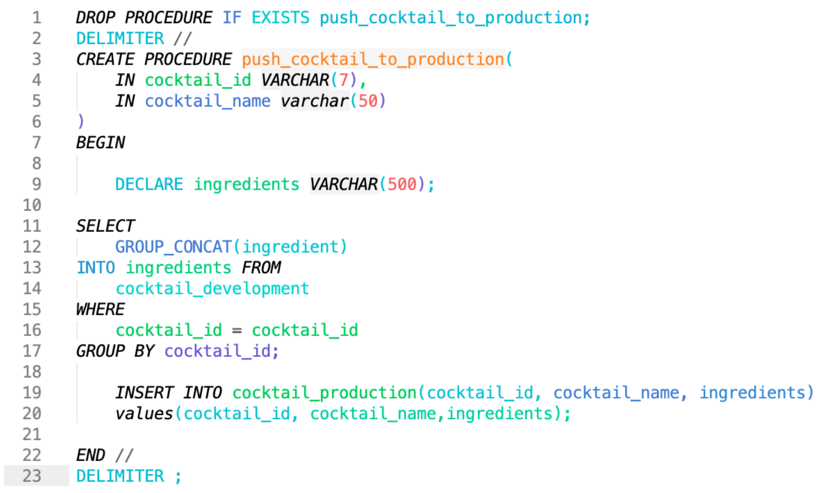
```
DROP PROCEDURE IF EXISTS suggest_wine_confidence_nationality;
DELIMITER //
CREATE PROCEDURE suggest_wine_confidence_nationality(IN pizza_id VARCHAR(7))
BEGIN

DROP TABLE IF EXISTS temp_pairing;

CREATE TABLE temp_pairing AS SELECT DISTINCT wine_pairing.beverage,
    nationality_weight AS confidence,
    nationality FROM
    pizza_development
        JOIN
    element_restriction ON (pizza_development.element = element_restriction.element)
        JOIN
    wine_pairing ON (element_restriction.restriction = wine_pairing.pairing)
        JOIN
    wine_nationality ON (wine_nationality.nationality = element_restriction.restriction)
        JOIN
    wine_weights ON (wine_pairing.beverage = wine_weights.beverage)
WHERE
    pizza_development.pizza_id = pizza_id;

SELECT
    beverage, SUM(confidence) AS confidence
FROM
    temp_pairing
GROUP BY beverage
HAVING confidence >= 0.5
ORDER BY confidence DESC;

END //
DELIMITER ;
```

Code Snippet: Stored Procedure – suggest_wine_confidence_nationality

As mentioned in the white paper, the experimental pizza and cocktail will move to production after the selection of one of the striking names by marketing people. This functionality is implemented using a stored procedure as below,

V. push_cocktail_to_production

This stored procedure accepts the input as experimental cocktail_id and selected cocktail name. These inputs are processed to retrieve a grouped version of the ingredients used in the cocktail preparation and this information is inserted into the production table. The production table has the cocktails released in the production with a list of ingredients and one of the catchy names selected by marketing people.

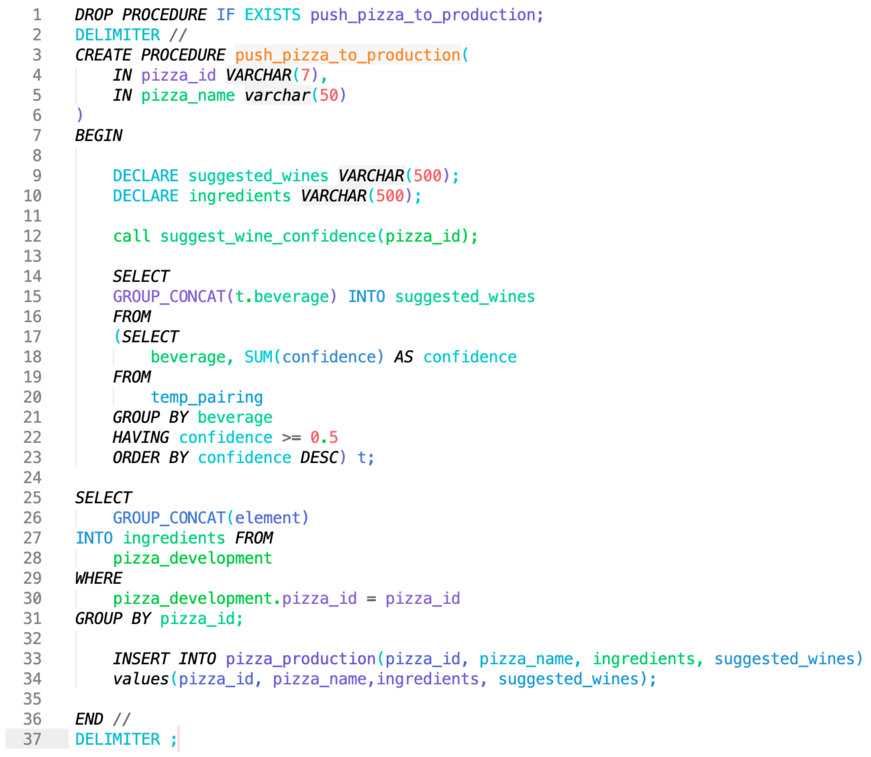


```sql
DROP PROCEDURE IF EXISTS push_cocktail_to_production;
DELIMITER //
CREATE PROCEDURE push_cocktail_to_production(
    IN cocktail_id VARCHAR(7),
    IN cocktail_name varchar(50)
)
BEGIN

    DECLARE ingredients VARCHAR(500);

SELECT
    GROUP_CONCAT(ingredient)
INTO ingredients FROM
    cocktail_development
WHERE
    cocktail_id = cocktail_id
GROUP BY cocktail_id;

    INSERT INTO cocktail_production(cocktail_id, cocktail_name, ingredients)
    values(cocktail_id, cocktail_name,ingredients);

END //
DELIMITER ;
```

Code Snippet: Stored Procedure – suggest_wine_confidence_nationality

VI. push_pizza_to_production

This stored procedure accepts the input as pizza_id from the development table and selects a pizza name. This input is processed to retrieve a grouped version of the elements used in the pizza preparation and this information is inserted into the production table. The production table has the pizzas released in the production with a list of elements and one of the catchy names selected by marketing people.



```sql
DROP PROCEDURE IF EXISTS push_pizza_to_production;
DELIMITER //
CREATE PROCEDURE push_pizza_to_production(
    IN pizza_id VARCHAR(7),
    IN pizza_name varchar(50)
)
BEGIN

    DECLARE suggested_wines VARCHAR(500);
    DECLARE ingredients VARCHAR(500);

    call suggest_wine_confidence(pizza_id);

    SELECT
    GROUP_CONCAT(t.beverage) INTO suggested_wines
    FROM
    (SELECT
        beverage, SUM(confidence) AS confidence
    FROM
        temp_pairing
    GROUP BY beverage
    HAVING confidence >= 0.5
    ORDER BY confidence DESC) t;

SELECT
    GROUP_CONCAT(element)
INTO ingredients FROM
    pizza_development
WHERE
    pizza_development.pizza_id = pizza_id
GROUP BY pizza_id;

    INSERT INTO pizza_production(pizza_id, pizza_name, ingredients, suggested_wines)
    values(pizza_id, pizza_name,ingredients, suggested_wines);

END //
DELIMITER ;
```

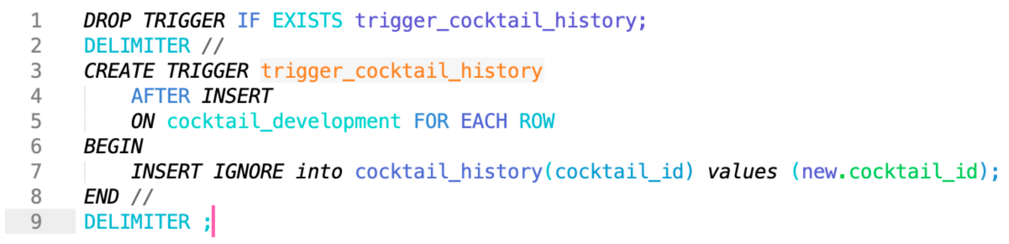
Code Snippet: Stored Procedure – suggest_wine_confidence_nationality

5.2 Triggers

Triggers are special types of stored procedures that are stored at a database server and automatically invoked for a specified event. The event could be anything such as insert, update and delete on a specified table. There are several benefits of a trigger, to enforce the business logic and maintain the history table for the development process we have implemented the below triggers.

I. trigger_cocktail_history:

This trigger is activated after any insertion operation on the *cocktail_development* table. The purpose of this trigger is to maintain the *cocktail_id* of experimental cocktails, every time a new cocktail is inserted into the development table, the trigger will insert the respective cocktail_id in a history table. *cocktail_history* table has *cocktail_id* and status attribute, the default status is *development*.

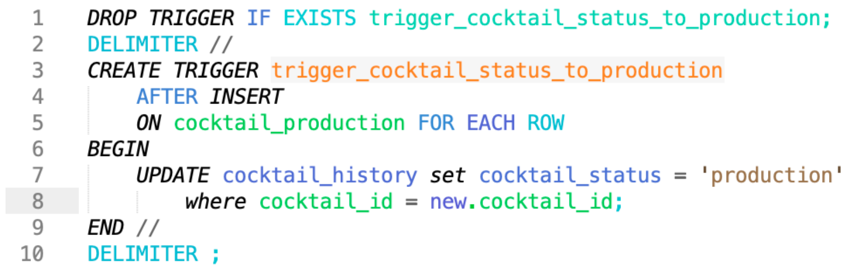


```sql
DROP TRIGGER IF EXISTS trigger_cocktail_history;
DELIMITER //
CREATE TRIGGER trigger_cocktail_history
    AFTER INSERT
    ON cocktail_development FOR EACH ROW
BEGIN
    INSERT IGNORE into cocktail_history(cocktail_id) values (new.cocktail_id);
END //
DELIMITER ;
```

Code Snippet: Trigger - cocktail_history

II. trigger_cocktail_status_to_production:

This trigger is activated after any insertion operation on the *cocktail_production* table. The purpose of this trigger is to change the status of the respective cocktail when it is approved by a marketing team. On approval, the cocktail is moved to the production table. To keep the track of development and production cocktails, the *cocktail_history* table is created and will be updated through these triggers.



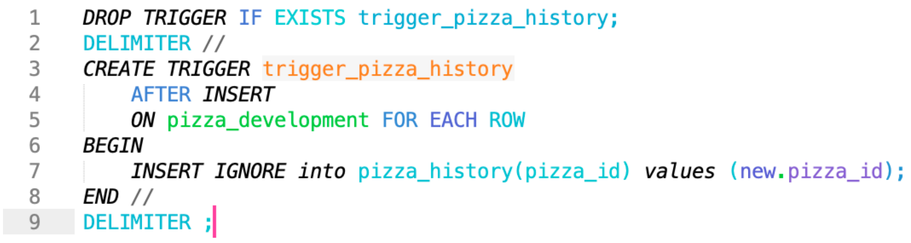
```sql
DROP TRIGGER IF EXISTS trigger_cocktail_status_to_production;
DELIMITER //
CREATE TRIGGER trigger_cocktail_status_to_production
    AFTER INSERT
    ON cocktail_production FOR EACH ROW
BEGIN
    UPDATE cocktail_history set cocktail_status = 'production'
        where cocktail_id = new.cocktail_id;
END //
DELIMITER ;
```

Code Snippet: Trigger – cocktail_status_to_production

Similar to cocktail, we have created 2 more triggers for the pizza as well.

III. trigger_pizza_history:

This trigger is activated when a new pizza is inserted into the *pizza_development* table. This trigger helps to maintain the development pizza *status* in the dedicated history table. *Pizza_history* tables store the pizza_id and status, the default status is *development.*

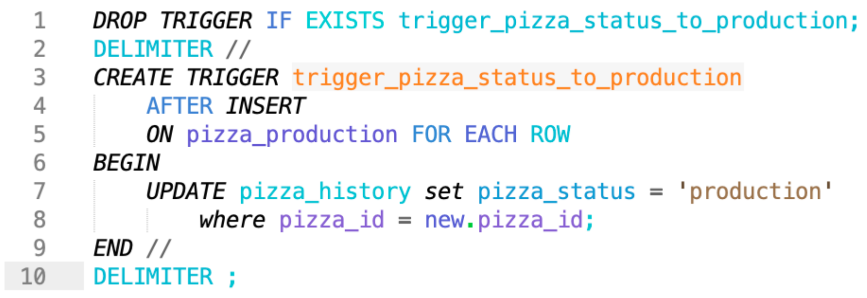


```sql
DROP TRIGGER IF EXISTS trigger_pizza_history;
DELIMITER //
CREATE TRIGGER trigger_pizza_history
    AFTER INSERT
    ON pizza_development FOR EACH ROW
BEGIN
    INSERT IGNORE into pizza_history(pizza_id) values (new.pizza_id);
END //
DELIMITER ;
```

Code Snippet: Trigger – pizza_history

IV. trigger_pizza_status_to_production:

This trigger is activated when a development pizza is moved to production after the approval of a marketing team. Once the pizza is moved to production, this trigger marks the status of a respective pizza in a history table. The status will be updated to production.



```sql
DROP TRIGGER IF EXISTS trigger_pizza_status_to_production;
DELIMITER //
CREATE TRIGGER trigger_pizza_status_to_production
    AFTER INSERT
    ON pizza_production FOR EACH ROW
BEGIN
    UPDATE pizza_history set pizza_status = 'production'
        where pizza_id = new.pizza_id;
END //
DELIMITER ;
```

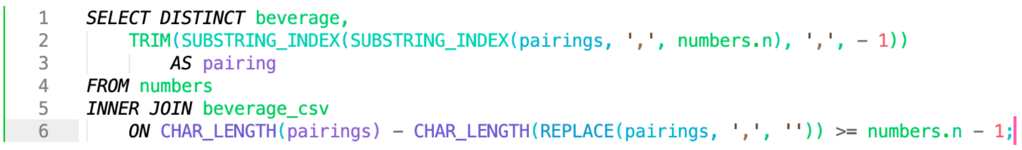
Code Snippet: Trigger – pizza_status_to_production

6. Example Queries: Your Database in Action

The database schema is created by taking the reference of provided data, so in order to create the normalized tables, there are several queries created for data generation purposes. These queries helped to understand the data and their relationships with other tables.

1. In the initial phase of development, the data was in combined form, i.e., comma-separated form, to map these attributes as an individual row of table was a big task and this is achieved by using one additional table that stores the ordered numbers and implementing join statement with a respective combined column as below,

   Query:



```sql
SELECT DISTINCT beverage,
    TRIM(SUBSTRING_INDEX(SUBSTRING_INDEX(pairings, ',', numbers.n), ',', - 1))
        AS pairing
FROM numbers
INNER JOIN beverage_csv
    ON CHAR_LENGTH(pairings) - CHAR_LENGTH(REPLACE(pairings, ',', '')) >= numbers.n - 1;
```

Input

Beverages	Pairings
Barolo	beef, venison, duck
Beaujolais	Feta
Bordeaux	venison, duck

Output

Beverages	Pairings
Barolo	beef
Barolo	Feta
Barolo	duck
Beaujolais	Feta
Bordeaux	venison
Bordeaux	duck

Code Snippet: Query – convert comma-separated values as rows

   The above query pattern is utilized for multiple attributes such as *restrictions* and *nationality* and *namelets* pairs as these attributes were present in comma separated form.

2. To recommend the wine as per the nationality, pairing and restriction mapping, we have decided to calculate the numerical measure of each value, for example particular wine has 5 pairing properties, so each property has the weight of 0.2 and for another wine with 10 pairing properties then the weight becomes 0.1, So we have calculated the nationality and pairing weight associated with each wine. For this

purpose, we have maintained a table with beverage as primary key and nationality and pairing weight as non-key attributes.

Query:

```sql
SELECT distinct beverage,
    1/count(distinct TRIM(SUBSTRING_INDEX(SUBSTRING_INDEX(nationality, ',', n1.n), ',', - 1)))
        AS nationality_weight,
    1/count(distinct TRIM(SUBSTRING_INDEX(SUBSTRING_INDEX(pairings, ',', n2.n), ',', - 1)))
        AS pairing_weight
FROM numbers n1
JOIN beverage_csv
    ON CHAR_LENGTH(nationality) - CHAR_LENGTH(REPLACE(nationality, ',', '')) >= n1.n - 1
JOIN numbers n2
    on CHAR_LENGTH(pairings) - CHAR_LENGTH(REPLACE(pairings, ',', '')) >= n2.n - 1
group by beverage;
```

Input

Beverages	Nationality	Pairings
Barolo	Italian	beef, venison, duck
Zinfandel Rosé	Californian, American	chicken, Piri Piri, Parma, spicy, seafood, Puttenesca, aioli, garlic, sardines, avocado, crab
Bordeaux	French	venison, duck

Output

Beverages	Nationality Weight	Pairing Weight
Barolo	1	0.3333
Zinfandel Rosé	0.5	0.0909
Bordeaux	1	0.5

Code Snippet: Query – convert comma-separated values as rows

The main motive of creating this database is to enable Wonka Labs to carry out experiments with the pizzas and cocktail with series of ingredients and suggest the catchy name for developed pizza and cocktail and to give wine preferences as per the ingredients mapping of pizza and wine. To demonstrate the whole pipeline of development to production, we will execute series of queries and show the results accordingly in this subsection.

6.1 Pizza – Development to Production

I. First and foremost, lets create a pizza, to create a pizza, we need to perform an insert query in *pizza_development* table. This table accepts a unique *pizza_id* and *elements*, Here the elements are referenced to the master table *pizza_ingredients*, so as per the provided constraint, pizza can be created with available ingredients or new ingredients should be added to *pizza_ingredients* table.

Check existing pizzas, and create a new pizza as below,

pizza_id	pizza_status
PZ001	production
PZ002	development
PZ003	development
PZ004	development
PZ005	production

Table: pizza_history

```
INSERT INTO pizza_development values
('PZ006', 'baby corn'),('PZ006', 'cherry tomatoes'),('PZ006', 'kimchi'),
('PZ006', 'pistachios'),('PZ006', "haloumi cheese"),('PZ006', "Mozzarella"),
('PZ006', "Queso Fresco"),('PZ006', "smoked tofu"),
('PZ006', 'Bechamel sauce'),('PZ006', 'Chimichurri sauce'),
('PZ006', 'wholewheat crust');
```

Code Snippet: Insert Query – Pizza_developement

II. Once pizza is in development mode, several views are created to know more about configuration of pizza as shown in below code snippet.

pizza_id	beverages	confidence	mapped elements
PZ006	Sol beer	1	Mexican
PZ006	Diet Coke	0.5	sweet
PZ006	Carlsberg beer	0.33	salty,spicy
PZ006	Chenin Blanc	0.33	salty,sour,spicy
...	...	...	...

Table: View – pizza_wine_pairing

pizza_id	element	first_word	second_word
PZ006	kimchi	Korean	M*A*S*H
PZ006	kimchi	Sour	M*A*S*H
PZ006	Mozzarella	Buffalo	
PZ006	baby corn	Jolly Green	
...	...	...	...

Table: View – pizza_ingredients_first_second_words

pizza_id	meat_count	veg_count	cheese_count	sauce_count	base_count	Total_count
PZ006	NULL	4	4	2	1	11

Table: View – pizza_elements_count

pizza_id	meat	veg	cheese	sauce	base	Total_count
PZ006	NULL	baby corn, cherry tomatoes, kimchi, pistachios	haloumi cheese, Mozzarella, Queso Fresco, smoked tofu	Bechamel sauce, Chimichurri sauce	wholewheat crust	11

Table: View – pizza_elements_category

All the above view can be queried with a simple select query with where clause.

III. In development phase, stored procedures are created to get the complementary wines that would be preferred with respective pizza.

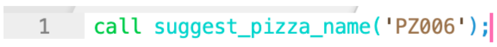
```
call suggest_wine_confidence('PZ006');
```

beverage	confidence
Sol beer	1
Gewürztraminer	0.99
Muscat Blanc	0.99
Carlsberg beer	0.66
Guinness beer	0.66
Heineken beer	0.66
Diet Coke	0.5

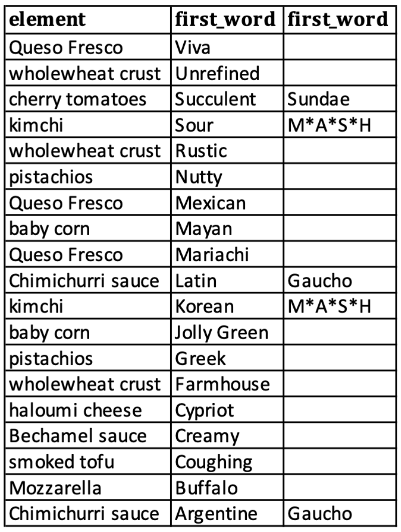
```
call suggest_wine_confidence_nationality('PZ006');
```

beverage	confidence
Sol beer	1

IV. For marketing people, a stored procedure is implemented that provide the striking namelets for respective pizza.

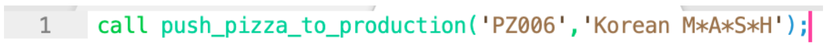


```
call suggest_pizza_name('PZ006');
```



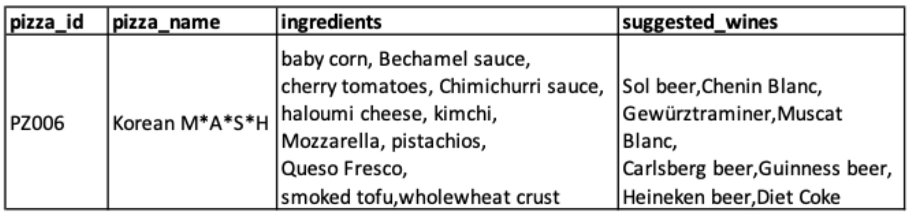
element	first_word	first_word
Queso Fresco	Viva	
wholewheat crust	Unrefined	
cherry tomatoes	Succulent	Sundae
kimchi	Sour	M*A*S*H
wholewheat crust	Rustic	
pistachios	Nutty	
Queso Fresco	Mexican	
baby corn	Mayan	
Queso Fresco	Mariachi	
Chimichurri sauce	Latin	Gaucho
kimchi	Korean	M*A*S*H
baby corn	Jolly Green	
pistachios	Greek	
wholewheat crust	Farmhouse	
haloumi cheese	Cypriot	
Bechamel sauce	Creamy	
smoked tofu	Coughing	
Mozzarella	Buffalo	
Chimichurri sauce	Argentine	Gaucho

V. Marketing team can decide the final name and release the pizza in production.



```
call push_pizza_to_production('PZ006','Korean M*A*S*H');
```

VI. In production table, the pizza with catchy name along with wine preferences and ingredients details are stored.



pizza_id	pizza_name	ingredients	suggested_wines
PZ006	Korean M*A*S*H	baby corn, Bechamel sauce, cherry tomatoes, Chimichurri sauce, haloumi cheese, kimchi, Mozzarella, pistachios, Queso Fresco, smoked tofu,wholewheat crust	Sol beer,Chenin Blanc, Gewürztraminer,Muscat Blanc, Carlsberg beer,Guinness beer, Heineken beer,Diet Coke

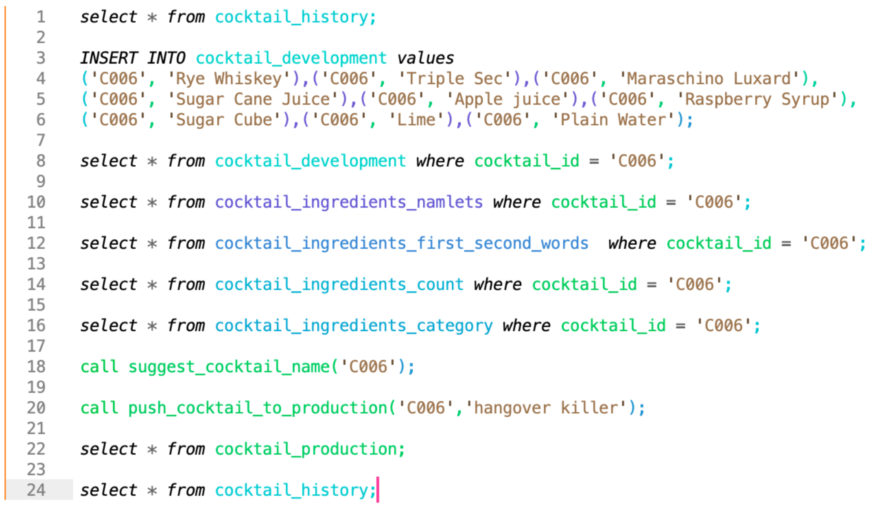
VII. Once the pizza is inserted into the production table, the status of respective pizza is updated at history table as *production*.

6.2 Cocktail - Development to Production

Here, the execution pipeline is similar to the pizza, so to limit the number of screenshots, the only queries are posted for cocktails.

I. Let's create a cocktail, to create a cocktail, we need to perform an insert query in *cocktail_development* table. This table accepts a unique *cocktail_id* and *ingredients*, Here the ingredients are referenced to the master table *cocktail_ingredients*, so as per the provided constraint, cocktail can be created with available ingredients or new ingredients should be added to *cocktail_ingredients* table.

II. Once cocktail is in development mode, several views are created to know more about configuration of cocktail as shown in below code snippet.



```sql
select * from cocktail_history;

INSERT INTO cocktail_development values
('C006', 'Rye Whiskey'),('C006', 'Triple Sec'),('C006', 'Maraschino Luxard'),
('C006', 'Sugar Cane Juice'),('C006', 'Apple juice'),('C006', 'Raspberry Syrup'),
('C006', 'Sugar Cube'),('C006', 'Lime'),('C006', 'Plain Water');

select * from cocktail_development where cocktail_id = 'C006';

select * from cocktail_ingredients_namlets where cocktail_id = 'C006';

select * from cocktail_ingredients_first_second_words  where cocktail_id = 'C006';

select * from cocktail_ingredients_count where cocktail_id = 'C006';

select * from cocktail_ingredients_category where cocktail_id = 'C006';

call suggest_cocktail_name('C006');

call push_cocktail_to_production('C006','hangover killer');

select * from cocktail_production;

select * from cocktail_history;
```

Code Snippet: Cocktail – Development to production

III. For marketing people, a stored procedure is implemented that provide the striking namelets for respective cocktail.

IV. Marketing team can decide the final name and release the cocktail in production.

V. In production table, the cocktail with catchy name and ingredients details are stored.

The detailed execution of queries with few additional queries are provided in the SQL file for further demonstration.

7. Conclusions

The database is designed to adhere to all the specifications provided in the white paper. The data model design is extremely normalized and robust for future additional functionality. The tables are inherited from the actually provided data that ensures the integrity of the data. The relational constraints such as primary key, foreign key references, and data cascade are followed strictly to reduce the redundancy and improve the atomicity of data. Considering the real-world applications, multiple database functionality such as stored procedures, triggers, and views are created. To support the innovatory vision of Wonka Labs, some numerical calculations are introduced that calculates the weights or confidence of each pizza wine pairing to recommend the highly preferred wines to upscale the overall business. To enhance the wine preferences the nationality attribute is also considered wisely. This database is designed in a highly relational manner and the robustness of this design will take care of any logical error that occurred at the application level. Finally, the robust yet normalized data model is designed that can be enhanced on demand as per the requirements in the future with minimal impact.

Acknowledgements

I would like to give special thanks to my professor Tony Veale for his interesting way of teaching. The thorough SQL knowledge through lectures and hands-on experience in practical sessions helped me to develop the intuition about relational databases and practices. A detailed explanation of each phase of development is provided in the above sections. All the database design and functional approaches are defined by me to adhere to the requirements as mentioned in the white paper. The theoretical concepts provided as lecture notes and a book (Atzeni, 1999) was helpful to understand the several approaches in a relational database. To rectify technical challenges of project, MySQL official documentations (MySQL: Documentation 2020) was really helpful. Finally, the continuous guidance of professor and TA's helped me throughout the semester to design and develop a data model from scratch.

References

Atzeni, P., 1999. Database Systems Conceptual, Language & Architectures. London: McGraw-Hill, pp.155-180.

Dev.mysql.com. 2020. Mysql :: Mysql Documentation. [online] Available at: https://dev.mysql.com/doc/

Above mentioned references along with lecture notes are sufficient to create an comprehensive database design. Few research papers were also surveyed to know more about food related database management systems but not included here as references.