A PROJECT REPORT ON

WONKA LABS
“EXPERIMENTS IN FOOD DISCOVERY”

SUBMITTED
IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

COMP40725 - INTRO TO RD & SQL PROGRAMMING

SUBMITTED BY

PRASHANT UTTAM WAKCHAURE **Student No: 20200126**

prashant.wakchaure@ucdconnect.ie

SCHOOL OF COMPUTER SCIENCE

LECTURER: PROF. TONY VEALE

ABSTRACT

This project report envisions the practical implementation of the whitepaper which was leaked from the laboratories of the notoriously secretive William P. Wonka III, grandson of that most famed Wonka of all, the founder of Wonka industries and the creator of the world's most delectable candies. Wonka has suffered miserable amount of reputational damage, in spite of which, his grandson is about to recast the Wonka brand by amplifying its reach into the food ventures. These ventures extraordinarily encompass the savory baked goods and adult beverages. We will see what these are in the later sections. There are multiple teams handling the various kinds of strands which the Wonka innovation pipeline has defined. The teams include, various scientists, marketing analysts, data analysts, and many more. I have been appointed as a DBA (Database Administrator), wherein I'm responsible for creating the RDBMS bedrock in order to brace the strands offered by the Wonka Laboratories.

After reading the 6-page white paper, I realized I have a lot of constraints, definitions, normalizations, procedural elements, and what not… to take care off; for which I thought about using MySQL Workbench to initiate the design of the Wonka Labs database schema consisting multiple strands. While designing the schema, the most important thing which I kept in mind was interpretability of the SQL queries, defined tables, views, procedural elements, etc. Even though the notion of my database schema might seem gigantic, I have conspicuously taken the data normalization process very whole-heartedly, because of which there are a lot of tables; but they are evident proof of how efficient, and easy-query processing the entities of the database are.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

The project wholesomely demonstrates the design and query processing of the database which is built in SQL for the Wonka Laboratories. It is in structured form, i.e., the entities of the database are related either loosely or tightly, considering the relationship between the various strands of the Wonka innovation pipeline. The database schema majorly focuses on coherency, accuracy, completeness, and organized way of data retrieval whenever needed in the desired format. Although this project does not form a complete high-level application, but its query processing aims at demonstrating efficient data retrieval which is human-readable and can be implemented for a complete end-to-end application. The designed schema also envisages the need for scalability, which could be implemented on top of this project, wherein the major cornerstone of the project is to extend the database vertically, rather than horizontal expansion of columns, which might lead to curse of dimensionality. The project was developed keeping the notion of good underlying database design, which is normalized, interpretable and efficient simultaneously.

Now, talking about the strands of the Wonka Laboratories, they have introduced 3 new strands in addition to the traditional candy development process, which are: the cocktail development for the adult beverage market; pizza development for the savory baked goods and wine cellar provision which are suggested in accordance to the pizza ingredient characteristics. Out of the 3 strands, the pizza and cocktail strands are composite in nature, wherein they both have similar structures, but the only thing missing is the pairings/properties of the ingredients in the cocktail, as there is no pairing with wine for cocktails. Whereas, the pizzas go hand in hand with the wine cellar provision, wherein the wines are paired to the properties/characteristics of the pizza. We will look at this in elaboration in the ahead sections. One thing to note here is that the Wonka company do not produce these wines, although they are hoping to start its production relying upon the existing database schema, whereas Wonka upsells these wines by suggesting suitable pairings with its baked offerings. The pizza and cocktail strands are built upon the phenomenon of production and development. The tables created for production and development along with its ancillary tables does not act as a complete package of a pizza/cocktail/wine delivery application. Instead, the things which are developed in the development table(s) are pushed to the production table with some real confidence value (which is pizza/cocktail name in our case), wherein, as suggested, the marketing team will choose the further options from the production table, is what I assumed after reading the abstract white paper. But, in order to fulfill the scope of the project, I have indeed created stored procedures which picks one row randomly for a unique id from the production table which acts in disguise of the marketing team. But in real world, as mentioned in the white paper too; it will be the marketing teams' job to select the best confidence value (pizza/cocktail name) from the production table in order to attract the buyer's attention.

One thing which can be inferred after reading the white paper is that the scope of the project is somewhere down the line confined to the operations performed in the database by the DBA. The things which are to be done by the scientists and marketing team are completely handed over to the DBA, which is me, which boils down to the things which can be efficiently assumed and built upon in a schematic manner. Even though, the things are assumed, I have tried my level best to keep the strands in the database as coherent and normalized as possible by introducing various views, stored procedures, triggers and other structural query elements intrinsically.

We will look at the 3 strands in a combined way, as they resemble high coherence. Firstly, "The Pizza Oven" of the Wonka Foods already sells traditional pizzas (readymade) under their traditional names. In addition, they are also planning to sell their own made toppings/base combinatory pizzas. These ingredients have namelets, i.e., the *first_name* & the *second_name*. With that said, I have assumed that every ingredient will have a single first name and single second name, rather than making complex assumptions like multiple ingredients may have multiple names, and so on.

These namelets are stored in the respective table. We will look at the table names and other entity-relationships of these tables and coherence of strands in the next sections. Now, as given in the white paper; the pizza development and production will require 2 tables, but henceforth we will refer them as phenomenon's, since there are multiple tables in a SQL database schema, and this could create confusion. So, the Pizza Development process consists of pizzas with their respective ingredients, and the names of these pizzas are in disguise of ids and not proper pizza naming conventions like we see in our nearby pizza outlets. Now, the ingredients which I took into consideration are of 5 types: *Meat, Vegetables, Cheese, Sauce, and Toppings*. Every respective ingredient will have a namelet consisting of first and second name in some other table. The basic thing which we can assume here is that, when we add pizzas from development to production, there might be multiple first names and second names for the ingredients of the pizzas' and that's the reason why the production table(s) will consist of all these first and second names of all the ingredients, from which the marketing team will decide the best first name and second name mapping, while this is randomly handled by a stored procedure in our case. So, rather than selecting a single ingredients complete name for a specific pizza id name, we can give multiple choices to the marketing team to choose the best name for the developed pizza in the production table, which will in turn lead to attract the buyer's attention. The naming conventions which I have chosen for the ingredients are inspired from the "pizza_ingredients.csv" file, which is provided with the white paper.

In the practical labs, we saw the cost to make and average abv ratings of the cocktails, while in this task "The Cocktail Lounge" strand is expected to be analogous to the pizza oven strand. I have considered 4 types of ingredient items for this task, which are: *Alcoholic Drinks, Soft Drinks, Garnishes, and Glasses.* In accordance with the white paper, there will be similar production and development phenomena for the cocktail lounge. Equivalent to the tables in pizza realm, there will be first and second names for the cocktail ingredient items which will be developed in the development table and will be duly added (with a set of first and second names for each ingredient item) in the production table.

Now, for this task, there were no namelets supplied, so I have assumed ridiculous names for the cocktail ingredients! Even though they might sound funny and illogical, I tried my level best in naming all the alcoholic and non-alcoholic ingredients along with the glasses and garnishes, since I'm not at all into alcohols! Now, why did I name the glasses and garnishes too? As suggested in the paper that the composite nature of pizza-cocktail strand also resemble in the way the naming conventions are given, wherein the alcoholic drinks are somewhat analogous to the meat, soft drinks to the vegetables, garnishes to the cheese/sauces, and glasses to the bases. Hence, the cocktail strands' production and development tables are a replica of what we see in the pizza realm. I also engineered the overarching goal of how the database keeps equilibrium between the cocktails and the pizzas, which will be elaborated in the upcoming sections ahead.

Going ahead, we encounter "The Wine Cellar", wherein the Wonka company does not yet produce its own wines, in turn they upsell them with proper pairing (mapping) to the properties/characteristics of the pizzas in the pizza realm. Now, referring the "beverages.csv" which was provided along with the white paper, I mapped the pairings with the restrictions given in the "pizza_ingredients.csv". One most peculiar thing to notice here is that, sometimes the wine pairings are not only dependent on the restrictions, but also on the ingredient names and nationalities. To ease this process in order for easy query retrieval. I created a separate table for pairings which contains all the properties, characteristics, some names of the ingredients which map to the wine (as given in the csv file), and also some nationalities. Now, these pairings will contain a pairing id which will be given to the ingredients with their respective characteristics. Similarly, the pairing ids will be given to the wines comprehensively in accordance to the csv file provided and also some manual additions and denominations as per the appropriate convenience. In this way, the wine suggestions will be displayed in a proper view (virtual table) in my case, as the view cannot be mishandled by the end teams, and will act as a read-only structure. Now, if we consider mapping a single ingredient property with the pairing, then a bunch load of wines will be suggested, since all the wines might be paired with that certain property. So, rather than pairing a single ingredient property, I kept a heuristic threshold of distinct pairing ids greater than or equal to 3, which mean that if the wine consists of more than 3 pairing ids which matches with the ingredients' properties for a given pizza, we return (suggest) only those many sets of wines. This will be better understood in the schematic discussions ahead.

In crux, the development tables of the pizza and cocktail strands will consist of id names along with the ingredients for the particular ids, and the production tables will later consist of the list of names for a particular pizza/cocktail id name generated with the help of the development and other ancillary tables. On the other hand, the wine suggestions will also be displayed along with the pizza id name. Now, to go an extra mile, I randomly generated a complete name for the pizzas and cocktails from their respective production tables' - ingredients' - first and last name. I'm implementing this in disguise of multiple stored procedures, which will return select queries based upon the same things as discussed.

With that said, I briefly discussed about what will be happening in my version of the Wonka Labs database. I know that some things might be blur and unclear. But the interrogatory clouds shall disappear in the upcoming sections, wherein I will discuss about the motivation and scope behind these comprehensive steps taken in order to design the Wonka labs database schema!

2 DATABASE PLAN: A SCHEMATIC VIEW

In this section, we will be discussing about the schematic plan of the database, which consists of the entities and its relationships. An entity is something about which we store data. Entities are not essentially needed to be tangible [1]. As seen in fig. 1., cocktail/pizza development, cocktail/pizza name-lets, cocktail/pizza ingredients are some entities in our database. When we represent entities in a database, we actually store only the attributes. Each group of attributes that describes a single real-world occurrence of an entity acts to represent an instance of an entity [1]. The fig. 1., only showcases how the entities are physically stored. It is certainly not an enhanced version which depicts what is actually there in my database schema, i.e., the main tables, views, routine graphs, etc. It is in fact, just a physical representational data flow of the original schema. The blue part encompasses the cocktail strand, whereas the red and green parts encompass the pizza and wine strand respectively. This indeed demonstrates the denormalized schema of my way of interpreting the Wonka labs new innovations in a brief schematic manner.

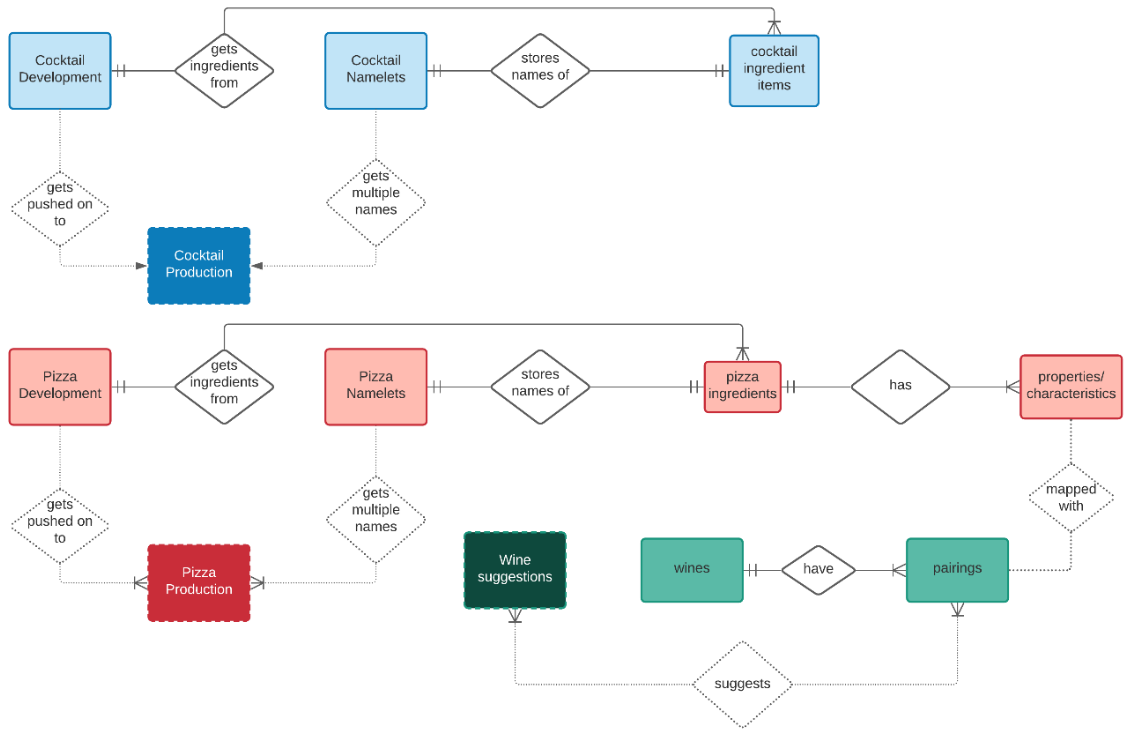


Figure 1 - Conceptual Entity-Relation Diagram

We can evidently see from the fig. 1. as well as from a general understanding of the white paper, that the database consists of 3 paramount entities:

- ❖ Pizza
- ❖ Wine
- ❖ Cocktail

These are recognized as prime entities, since the behavior of the database revolves around these entities. Even though pizzas and wines are sharing primary/foreign keys. We can see high resemblance of similarity between the pizza and cocktail realm. It's just that the wines are mapped with the pairings/properties of the pizza's ingredients. We will look at these entities one by one taking the database schema into consideration:

- ❖ **Pizza –**

  This is one of the prime entities, which comprehensively includes the Wonka labs savory baked products, which is nothing but the pizza topping/base combinations. As seen in fig. 2., "The Pizza Oven" development altogether consist of 5 tables: *meat_ingredients, veg_ingredients, cheese_ingredients,*

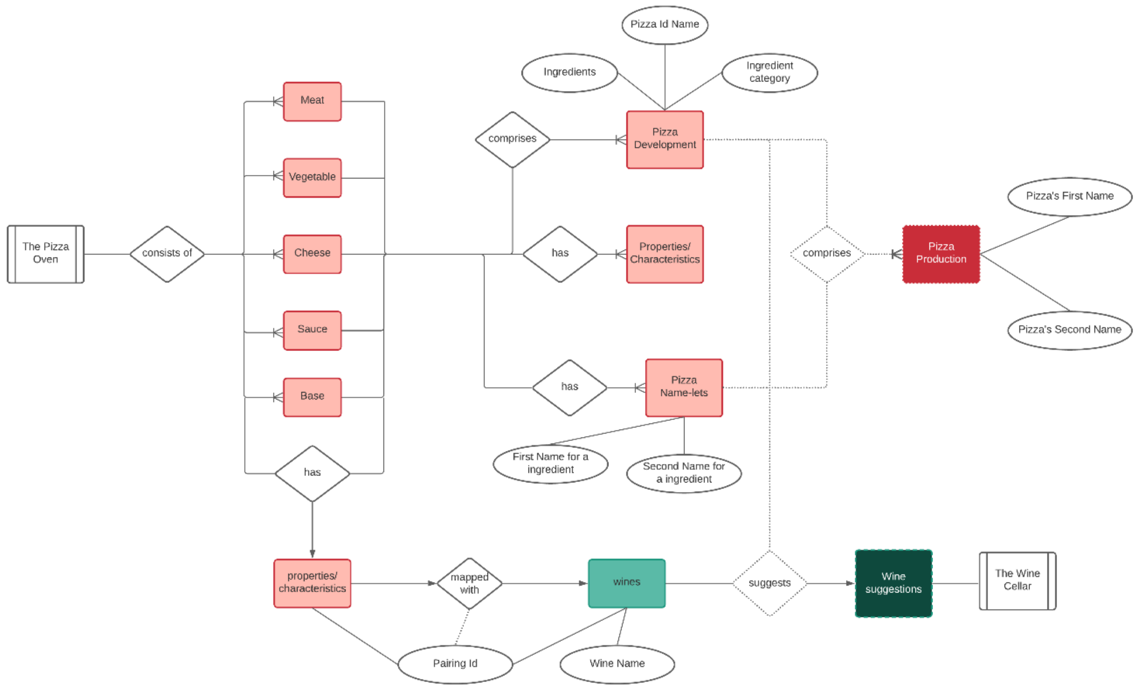


Figure 2 - Pizza & Wine Realm Entity-Relation Diagram

sauce_ingredients, and base_ingredients, along with their 5 "*properties"* tables, which consist of the particular *ingredient_id* along with its mapped properties/characteristics (*pairing_id*) from the "*pairings"* table. As the entries in the ingredients table along with their names in the respective table will be added by a specialist team as suggested in the white paper. However, for this project, the DBA is given the responsibility of adding them into the database, so, I assumed that the entries in the ingredients table, along with the pizza namelets table are basically manually added, which can be used by the other ancillary tables. In crux, these tables are basically normalized look up tables for adding the ingredients having certain names into the development table, and further having deemed as successful innovations, they are being added to the production table, which is a view. I will discuss about the production table shortly. But firstly, we will have a glance through the development side of the pizza strand.

The designated development table (*pizza_dev_table*) contains the ingredients from the particular ingredient table; these ingredients are manually inserted, since it is where the pizza is developed and tested with different combinations of the toppings/bases. Along with the ingredients it contains a unique key, which is basically the primary key, called the "*pizza_name_id"*. Now, in order for the scientists to know the ingredient type, I have also defined a table called "*pizza_ingred_categories"*, which contains the 5 types of ingredients in the pizzas with 1 to 5 ids. These ids are foreign key in the "*pizza_dev_table"*. Now, the pizza names are based upon the ingredients which are present in a particular unique *pizza id name*. One thing which I assumed from the white paper and the provided csv file is that – each pizza name will be a 2-word name and the 2 words will dignify the ingredient in it, so intuitively every ingredient will have a 2-word (*first_name, second_name*) name, which will be present in a table called – *pizza_namelets*. Previously, we added the ingredients (*ingred_name*) into the *pizza_dev_table*, this attribute is a foreign key referencing to the ingredients(*ingred_name*) in the *pizza_namelets*. It might sound confusing, but on a higher cohesion level; this kind of referencing in order to achieve relationships between the entities in the EER diagram is indeed necessary. One more logical reason behind doing this is, now we cannot add any dirty value in the development table, as the *pizza_namelets* table is defined prior to the *pizza_dev_table*, and we can only enter pizza ingredients into the *pizza_dev_table* if it has a namelet in the *pizza_namelets* table. This is a major design issue which I handled quite intuitively here. This is the power of referencing or what we call a foreign key. Foreign keys are the attributes which point to the primary key of another table. They act as a cross-reference between these two tables, thus establishing an entity-relationship [2].

Subsequently, we move the successfully innovated pizzas to the production table. Now, in limitation to a white paper project or we can say – in reality, we don't have any authority or any marketing team to choose a perfect name for a pizza in the development table, so that the pizza id name will be given an official pizza name based upon its lucrative first and second names (which are based upon the ingredients in it). So, we take all the namelets (*first_name, second_name*) of that particular *pizza_name_id,* e.g., PZ001, and store in a view called *pizza_prod,* which in turn is the production table. Now, the marketing team will have multiple namelets for a pizza id along with the ingredient name (*ingred_name*), since they should be aware of which ingredient is corresponding to which namelet. I will demonstrate this in a disguise of a stored procedure in the section ahead.

- **Wine –**

  This is the second prime entity, which partially includes the Wonka labs adult beverages offerings called as the "The Wine Cellar". As discussed in earlier sections too, Wonka labs do not produce the wines, instead they upsell it with perfect amalgamation with the pizzas in accordance to which ingredient name or properties of ingredients compliments which wine. One more important assumption or insight from the white paper is that, the specialist knowledge of which wines complement which ingredients is already given in the *beverages.csv* file. So, we store these pairings (which are nothing but restrictions or as we saw previously in the pizza realm – the properties/characteristics of the pizza ingredients) in a suitable table called "*pairings*". This table consists of the various parings which altogether describe the type, property, any characteristic, nationality, and any other restriction of the pizza ingredients. These pairings are unique, which means they have a unique primary key – *pairing_id*, which is used as a foreign key in various tables for referencing which paring element is mapped to the ingredient or the wine. With that said, we also create a "*wines*" table as seen in fig. 2., which stores the wines which wonka labs upsells. These wines also contain multiple *pairing_id's*. Now, these *pairing_id's* are also used as foreign keys in all the 5 *properties* tables present in the Pizza realm. I created separate properties tables for this same reason, and also to nullify the many to many relationships which was hindering between the pairings and the ingredients tables. I will discuss this in elaboration in the sections ahead.

Now, the *wine_suggestions* are displayed as a view similar to the pizza production table, which consists of the suggested wines for a particular pizza (*pizza_id_name*). These are suggested by joining multiple tables and inner queries. I will discuss this in the designated view section later ahead.

- **Cocktail –**

This is the third prime entity, which completes the Wonka labs adult beverages offerings called as the "The Cocktail Lounge". As seen in fig. 3., this entity is analogous to the pizza entity (production and development). It acts in symmetry thereby establishing the reflection of how the pizza-cocktail entities are a composite product with eye-catching names that attracts the buyer's attention. As discussed in the introduction section, the composite nature is depicted in the ingredients between the pizza and cocktail. However, one thing

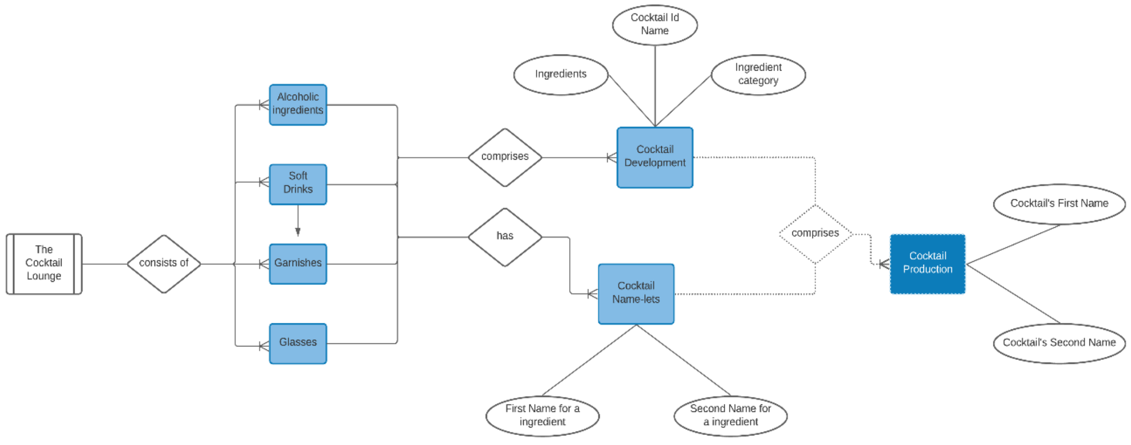


Figure 3 - Cocktail Realm Entity-Relation Diagram

which is not present for the cocktail entity is the properties of these cocktails, since we do not want to map the cocktails with any other entities/tables. It follows the same workflow as the pizza realm. However, to make it more interesting and introduce the concept of triggers coherently, there is one assumption that the *cocktail_namelets* table will be the master table, wherein the 4 *ingredients* tables will contain these *cocktail_namelet_id* as foreign keys. This, however makes the *cocktail_namelets* denormalized, but I'm aware of it and I have purposely planted this in the database for the purpose of this project and in order to introduce the concept of triggers, wherein the trigger will be applied on the *cocktail_namelets* table, and whenever anything is inserted, updated or deleted from the *cocktail_namelets* table, the appropriate ingredient with the appropriate *ingred_item_category_id* will be manipulated based on the manipulations in the *cocktail_namelets* table. I will explain this in depth in the procedural elements section later ahead.

Subsequently, interesting combinations of ingredients will be added to the cocktail development table (*cocktail_dev_table*), comprising of alcoholic ingredients, soft drinks, garnishes and a single glass for a particular *cocktail_name_id,* e.g., CKT001. And in the similar way as the pizza realm, the successfully deemed cocktails will be added to the cocktail production, which is a view called "*cocktail_prod*". Here also, the cocktail production will consist of multiple *first_name* and *second_name* for a particular *cocktail_name_id*, since there should be options made available to choose a perfect pair of first and second name which goes in hand with the pizza realm products to catch the buyer's attention. I have implemented this randomly in disguise of a stored procedure, which I will be discussing in the respective section later ahead.

Hence, in this section I conspicuously elaborated on the schematic view of the wonka labs database and my motivation behind the database design. All the intrinsic entities (tables) within the prime entities are created keeping normalization and efficient storage mechanism (of the data available) into consideration. The E-R diagram also includes the derived data, which are the views (production tables), which will be in turn derived from the existing tables. The schema which I have implemented satiates the rigid constraints elaborated in the white paper in an abstract way with proper procedural elements, views, and table constraints like [3]:

- ❖ **NOT NULL –** Many id columns cannot contain a null value.
- ❖ **UNIQUE –** The columns like *pairing_name, ingredient_name* is set to UNIQUE, wherein, every value is different in that column.
- ❖ **PRIMARY KEY –** It is an amalgamation of not null and unique, which uniquely identifies each row in a table.
- ❖ **FOREGIN KEY –** It avoids any actions on the table which is referenced which would lead to disruption between the relationship of the tables.
- ❖ **DEFAULT –** We set a default value of 'NA' to be inserted for the first & second names in the namelets table, if there are no values inserted.

However, the high-level Enhanced Entity Relationship diagram will be discussed in the next section.

3 DATABASE STRUCTURE: A NORMALIZED VIEW

In this section, we will witness the normalized view of the Wonka Labs database, which is what is expected in the final overview of any relational database schema. Inconsistent data are those that are correct and make sense, but when they are duplicated throughout the database and/or the organizations, they introduce unnecessary redundancy in the schema [1]. We usually normalize relations to help eliminate duplicated data, but in some instances, we have to contain information about the same entities in multiple tables, which is purposely done to offer a bigger picture beyond the norms of data normalization. I have practiced the same in my database schema offering too.

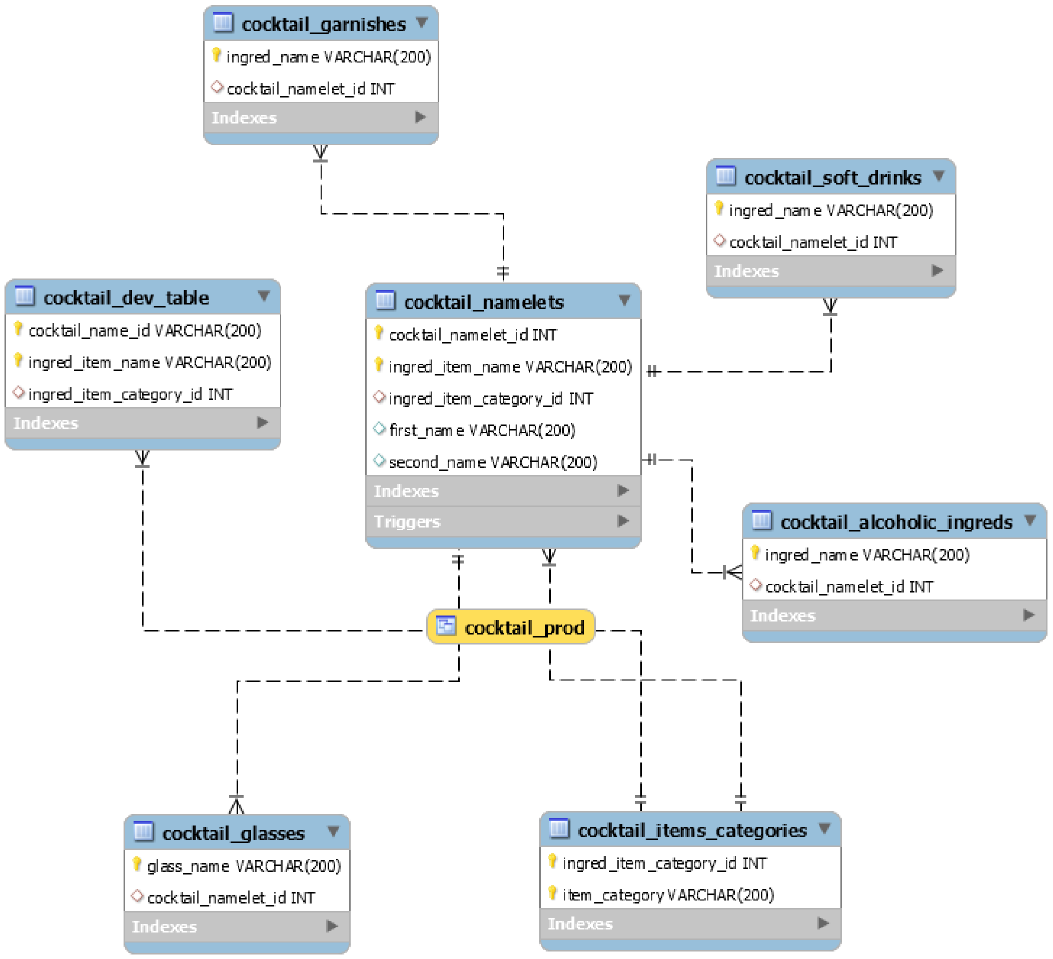


Figure 4 - Cocktail Realm EER Diagram

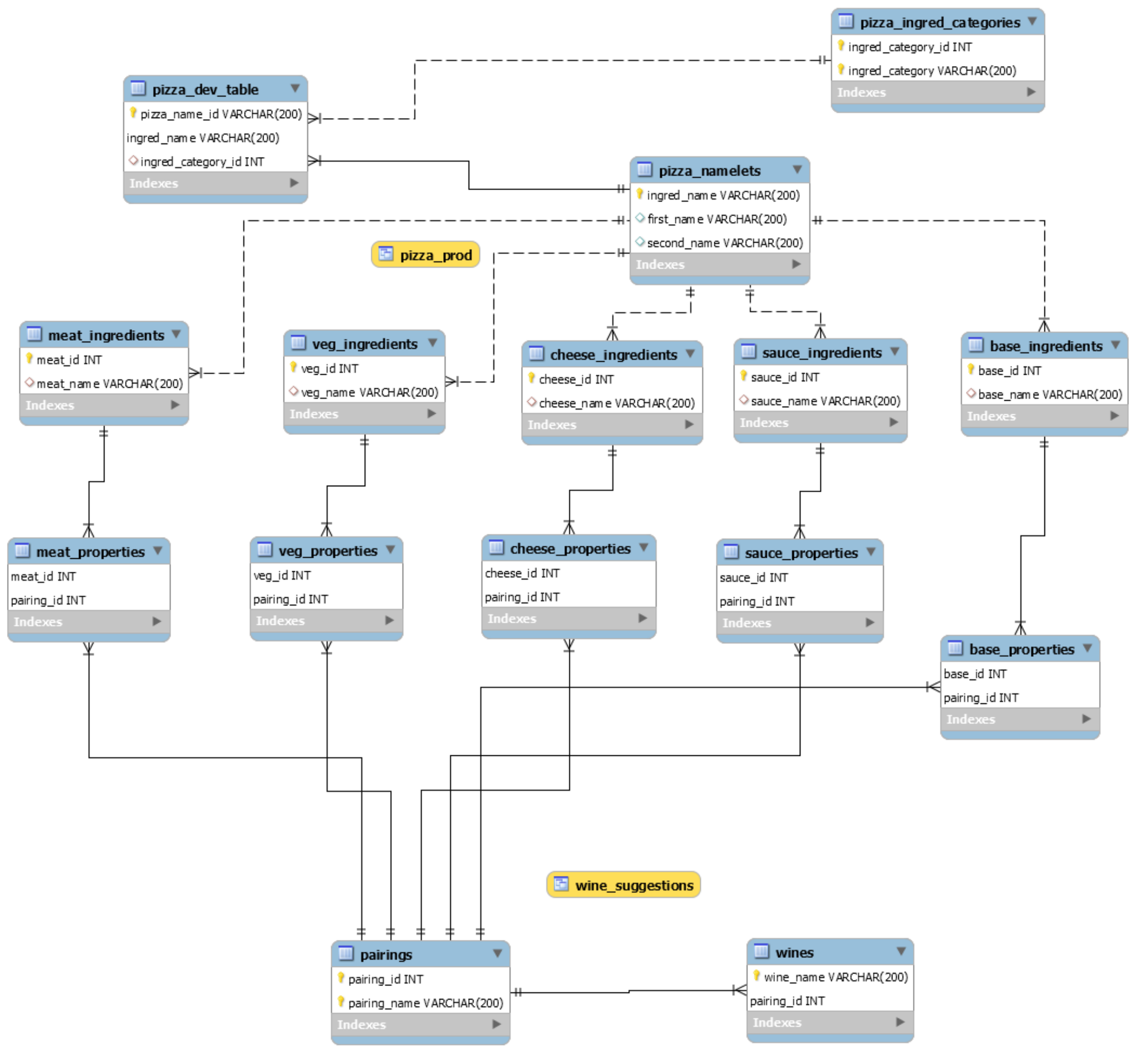


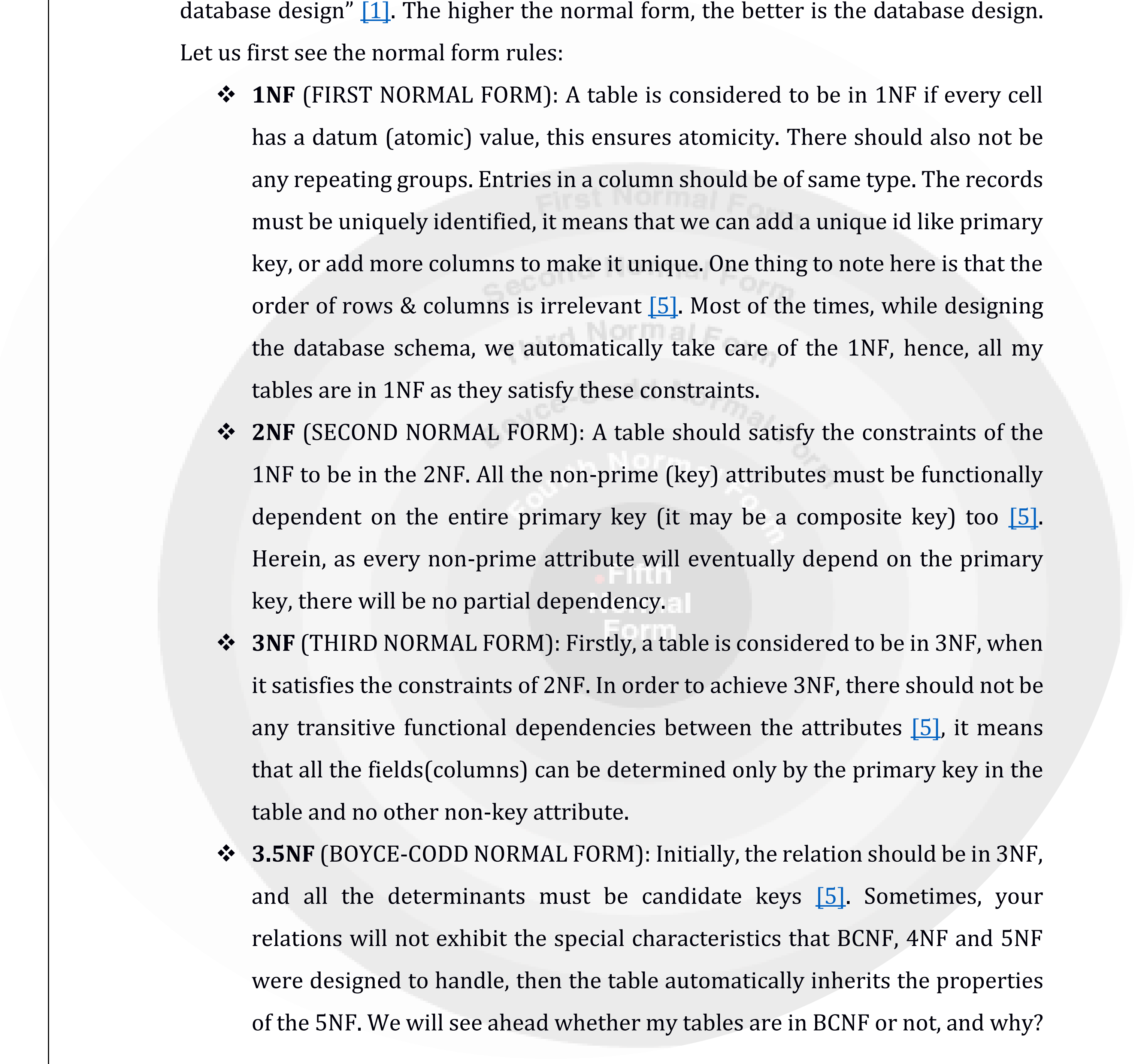
Figure 5 - Pizza & Wine Realm EER Diagram

The Conceptual and prime entity-related ER diagrams as seen in the schematic section were indeed very abstract, which along with a brief gave us a very rough idea about the database schema and the entities in it. As we can see in the above figures, the EER (Enhanced Entity-Relationship Diagram) gives us an elaborated and normalized view of all the entities (tables) included in the database in their most normalized form and also the relationship between those entities. With that said, fig. 4., illustrates the Cocktail Realm EER Diagram, whereas the fig. 5., illustrates Pizza & Wine Realm EER Diagram. I separately mentioned these EER Realm's, since the cocktail realm is not related as a database relationship to the pizza & wine realms.

Database relationships are very similar to how we are related to each other in our family tree. In context of a relational database, there are associations between the tables (entities). Profoundly, there are 3 types of relationships [4], and we will understand these relationships in context to our database schema:

- ❖ **ONE-TO-ONE:** In this relationship type, both the tables can have only one record on either side of the relationship. Every primary key value is related to only one or no records in the related table [4]. Usually, these kinds of relationships are forced by the business rules and don't flow naturally. I have also not used this relationship explicitly, but if we observe carefully, there is a one-to-one relationship between the *pizza_namelets and* all the 5 *ingredients* table, since each ingredient will have only one name-let as discussed in the previous sections.
- ❖ **ONE-TO-MANY:** In this relationship type, the primary key table contains only one record that relates to none, one or many records in the related table [4]. Intuitively, it is similar to you having only one mother, but your mother may have multiple children [4]. In our database schema, we can see many one-to-many entity-relationships, in fact, it is the most common database relationship evident in any database schema. For instance, one *ingredient* can have many *properties.*
- ❖ **MANY-TO-MANY:** In this relationship type, every record in one table can be related to any number or no number of records in the other table [4]. This can lead to data efficiency, storage, and redundancy problems. To avoid this, we introduce a bridging/associate/linking table, which takes a unique key from both the related tables. In my schema, one ingredient was having many properties, and one property was included in many ingredients, so I introduced 5 properties tables, which store the appropriate *ingredient_id* and the corresponding *pairing_id*, in this way the *properties* tables achieved a linking between the *pairings* and the 5 *ingredients* tables. The 2 ids in this linking table become the foreign keys referencing the 2 tables with primary keys.

Now, we will look at the normalization forms and how efficiently they help in query processing for any database schema. What is the difference between Data Normalization & Normal form? Normal form is nothing but a set of theoretical rules to which a relation must conform, whereas "normalization is the process of designing relations to adhere to increasingly stringent sets of rules to avoid problems with poor database design" [1]. The higher the normal form, the better is the database design. Let us first see the normal form rules:



- ❖ **1NF** (FIRST NORMAL FORM): A table is considered to be in 1NF if every cell has a datum (atomic) value, this ensures atomicity. There should also not be any repeating groups. Entries in a column should be of same type. The records must be uniquely identified, it means that we can add a unique id like primary key, or add more columns to make it unique. One thing to note here is that the order of rows & columns is irrelevant [5]. Most of the times, while designing the database schema, we automatically take care of the 1NF, hence, all my tables are in 1NF as they satisfy these constraints.
- ❖ **2NF** (SECOND NORMAL FORM): A table should satisfy the constraints of the 1NF to be in the 2NF. All the non-prime (key) attributes must be functionally dependent on the entire primary key (it may be a composite key) too [5]. Herein, as every non-prime attribute will eventually depend on the primary key, there will be no partial dependency.
- ❖ **3NF** (THIRD NORMAL FORM): Firstly, a table is considered to be in 3NF, when it satisfies the constraints of 2NF. In order to achieve 3NF, there should not be any transitive functional dependencies between the attributes [5], it means that all the fields(columns) can be determined only by the primary key in the table and no other non-key attribute.
- ❖ **3.5NF** (BOYCE-CODD NORMAL FORM): Initially, the relation should be in 3NF, and all the determinants must be candidate keys [5]. Sometimes, your relations will not exhibit the special characteristics that BCNF, 4NF and 5NF were designed to handle, then the table automatically inherits the properties of the 5NF. We will see ahead whether my tables are in BCNF or not, and why?

Now, we will see how I converted the entities in the previous section's ER diagram to achieve a normalized format as seen in the EER diagrams in this section, with some examples. I will be demonstrating all the final tables as seen in the EER diagram.

- **PIZZA & WINE REALM:**

Herein, I created 5 tables for the 5 ingredients in the pizzas (*meat, vegetables, cheese, sauce, base*), for instance, for meat, the table looks like:

- *meat_ingredients:*

meat_id	meat_name
23	sausages
24	tuna

As we can see, this table follows the BCNF form as there is a single record in each cell, there are no partial or transitive dependencies and the table has only one primary key. Although there is a candidate key in the table, it is actually a foreign key (*meat_name*) referencing the *ingred_name* in the *pizza_namelets* table. The other ingredients table look the same.

Now, to keep a record of the names of the pizza ingredients, I created a master table called pizza_namelets:

- *pizza_namelets:*

ingred_name	first_name	second_name
Avocado	Californian	New-Age
Asparagus Tips	Pointy	Arrowhead

As we can see, this table follows the BCNF form as there is a single record in each cell, there are no partial or transitive dependencies and the table has only one primary key. Also, the 2 non-prime attributes are only dependent on the primary key and not between themselves.

Now, to add properties of the ingredients, I created a master *pairings* table:

- *pairings:*

pairing_id	pairing_name
70	Aioli
71	Garlic

As we can see, this table follows the BCNF form as there is a single record in each cell, there are no partial or transitive dependencies and the table has only one primary key. Also, both the *pairing_id and the pairing_name* are composite unique keys. The pairing id is auto incremented and cannot be null.

Now, to map the pairings (properties) with the ingredients in the 5 ingredients table, I created 5 *properties* tables for each *ingredient* table), for instance, for cheese, the table looks like:

- *cheese_properties:*

cheese_id	pairing_id
1	5
29	5

As we can see, this table follows the BCNF form as there is a single record in each cell, there are no partial or transitive dependencies and the table has only one primary key. Also, both the *pairing_id and the cheese_id* are composite unique keys. Both of these attributes are foreign keys too, which reference the *meat_ingredients* and *pairings* tables respectively. I introduced these tables, to nullify the many-to-many relationship between the pairings and ingredients. The other properties table of the other ingredients look the same.

Now, we add ingredients into the pizza development table for a pizza id name, and it looks like:

➢ *pizza_dev_table:*

pizza_name_id	ingred_name	ingred_category_id
PZ0034	Beef Jerky	1
PZ0034	Black Pudding	1
PZ0034	Bell Peppers	2
PZ0034	Edam	3
PZ0034	Chimichurri Sauce	4
PZ0034	Thick Crust	5

Here, we are adding values from the look-up tables like the ingredient table, and the ingred_category_id is added from another table called "*pizza_items_categories*", which contains the 5 categories mapping as, 1 for meat, 2 for veggies, 3 for cheese, 4 for sauce and 5 for base. Even though this table does not satisfy the BCNF form as the non-prime attribute only depends on the ingredient name and not necessarily on the pizza_name_id. This field is purposely kept into the database, for the scientists to easily identify which ingredient is of which category.

Now, we have a look at the wine cellar:

➢ *wines:*

wine_name	pairing_id
Cava	1
Cava	56
Cava	78
Barolo	23
Barolo	56
Barolo	1

Here, we are storing the wines with the appropriate pairing id. This table is also in BCNF form, as intuitively both the columns are unique composite primary keys, since every wine may have multiple pairings, but each pairing will appear once in a wine. This table along with the pizza development table and the ingredient properties allows us to get the wine suggestions for the particular ingredient's pairings and the pizza name id.

- **COCKTAIL REALM:**

  As discussed, multiple times in this section above and in previous sections too & also evident from fig. 4., the cocktail schema is exactly similar to the pizza realm schema. The only difference is that there are no propery/pairings tables, and the master table cocktail_namelets looks like this:

  - *cocktail_namelets:*

cocktail_namelet_id	ingred_item_name	ingred_item_category	first_name	second_name
1	Absinthe	1	Mistress	Lady
2	Apricot Brandy	1	Fruity	Tipsy
3	Angostura Bitters	3	Ticker	Hippo
4	Rocks Glass	4	Dirty	Stinger
5	Cream	2	Mash	Bong

Here, it might look odd enough to encompass so many attributes into the cocktail realm as compared to the pizza namelets, but this is done in order to introduce the concept of a procedural element called triggers, I will discuss this ahead. Inspite of this amalgamation, the table is in BCNF as the the non-prime attributes inreg_item_category, first_name and second_name depend on the primary key attributes and not between themselves. Here, the composite primary key is cocktail_namelet_id and ingred_item_name.

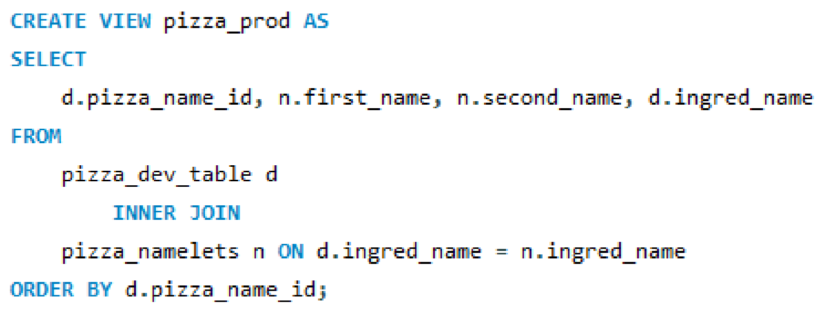
With that said, all the tables in the database are in their best normalized forms as essential and necessary for the operations mentioned in the white paper.

4 DATABASE VIEWS

In this section, I will discuss about the views which are incorporated in my database schema. What is a View? Views provide a way to give users a specific portion of a larger schema with which they can work [1]. They are defined in relational databases with the help of query processing. It is nothing but a virtual table which does not store any data intrinsically, although, it is essential as a read-only table. They are incorporated in the schema to offer security, flexibility and availability [7].

For the Wonka Labs schema, I have incorporated 3 views which are potential elements of the project and should be considered as views because of security and other data protection concerns. The 3 views are as follows:

❖ Pizza Production View (***pizza_prod***):



```
CREATE VIEW pizza_prod AS
SELECT
    d.pizza_name_id, n.first_name, n.second_name, d.ingred_name
FROM
    pizza_dev_table d
        INNER JOIN
    pizza_namelets n ON d.ingred_name = n.ingred_name
ORDER BY d.pizza_name_id;
```

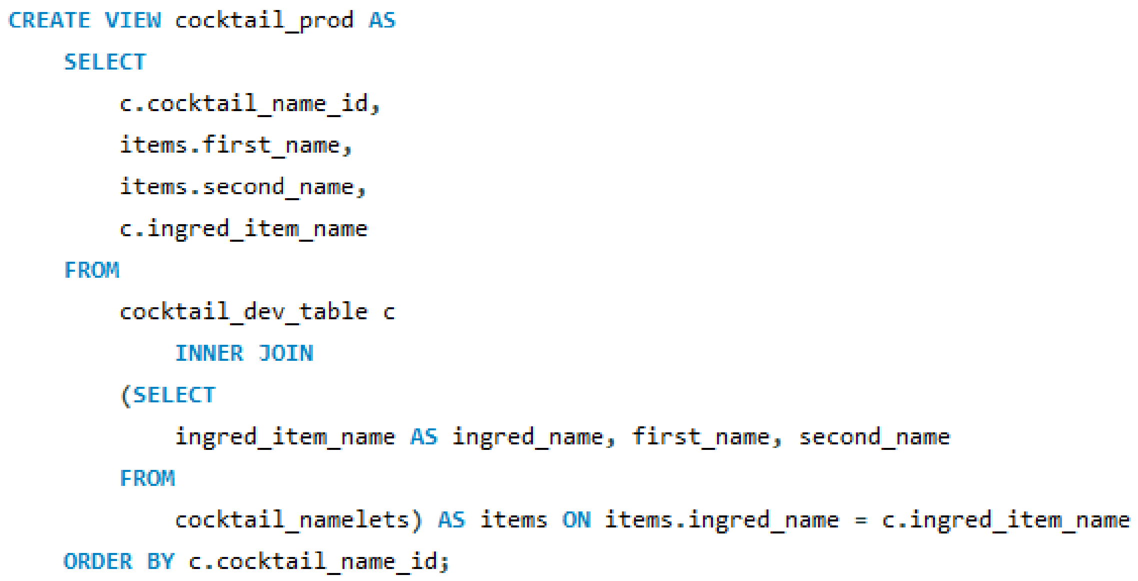
Figure 6 Pizza Production View

Output:

pizza_name_id	first_name	second_name	ingred_name
PZ001	Rubber	Chewtoy	Beef Jerky
PZ001	Ding-Dong	BONG	Bell Peppers
PZ001	Darth Vader's	Blackhole	Black Pudding
PZ001	Caped	Autumn	Capers
PZ001	Latin	Gaucho	Chimichurri Sauce
PZ001	Red Light	Master	Edam

As seen in fig. 6., the pizza production view consists of the particular pizza_name_id along with its suggested first and second names and the associated ingredient with it. I have incorporated the pizza production table as a view, since it is the final product which is into the production with the specific namelets corresponding to the ingredients to choose from. And as it is given to the marketing team for choosing the best namelet, it is the best approach if we display it as a virtual table (view), since we don't want anyone to change the ingredients or the names of the pizzas other than the scientists. So, offering a view for production is a best practice is what I feel. This view is an amalgamation of the *pizza_namelets* table and the *pizza_dev_table* on the unique common primary key *ingred_name.*

❖ Cocktail Production View (***cocktail_prod***):



```
CREATE VIEW cocktail_prod AS
    SELECT
        c.cocktail_name_id,
        items.first_name,
        items.second_name,
        c.ingred_item_name
    FROM
        cocktail_dev_table c
            INNER JOIN
        (SELECT
            ingred_item_name AS ingred_name, first_name, second_name
        FROM
            cocktail_namelets) AS items ON items.ingred_name = c.ingred_item_name
    ORDER BY c.cocktail_name_id;
```

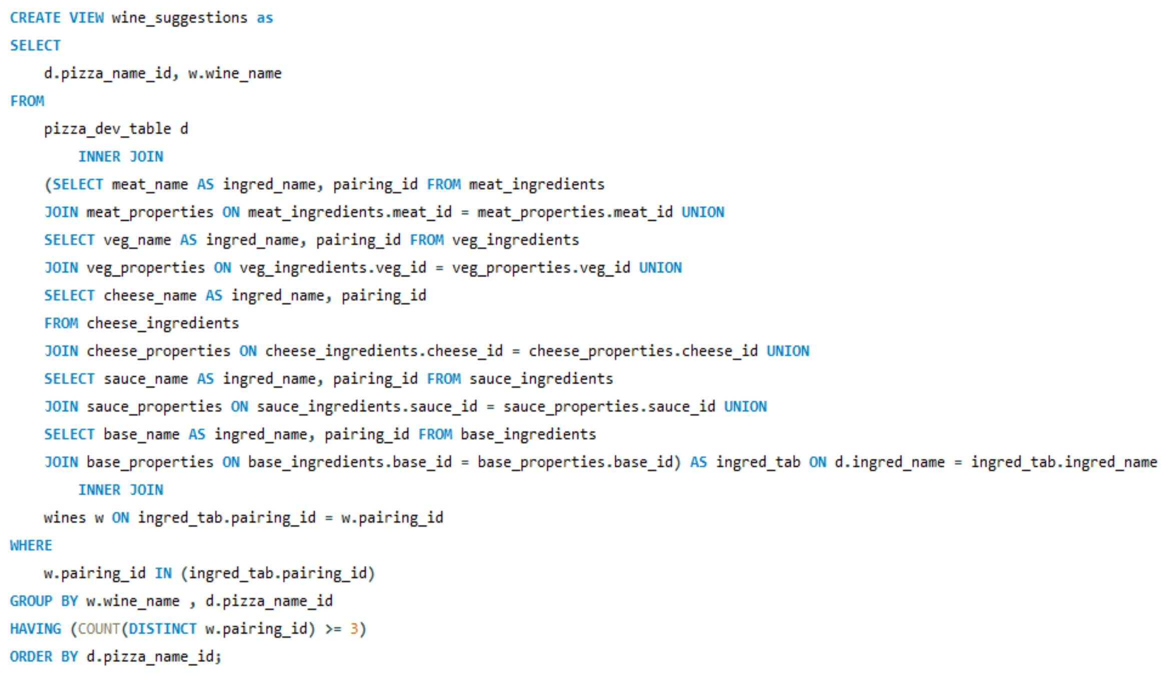
Figure 7 - Cocktail Production View

Output:

cocktail_name_id	first_name	second_name	ingred_item_name
CKT001	Gorilla	Milk	Brandy
CKT001	Flight	Aviation	Maraschino Luxardo
CKT001	Cheeky	Naughty	Cocktail Cherry
CKT001	Flexing	Special	Lemon Slice
CKT001	Great	Haze	Highball Glass
CKT001	Lining	Gimlet	Fresh Lime Juice

As seen in fig. 7., the cocktail production view consists of the particular cocktail_name_id along with its suggested first and second names and the associated ingredient with it. I have incorporated the cocktail production table as a view, since it is the final product which is into the production with the specific namelets corresponding to the ingredients to choose from. And as it is given to the marketing team for choosing the best namelet, it is the best approach if we display it as a virtual table (view), since we don't want anyone to change the ingredients or the names of the pizzas other than the scientists. So, offering a view for production is a best practice is what I feel. This view is an amalgamation of the *cocktail_namelets* table and the *cocktail_dev_table* on the unique common primary key *ingred_item_name.* It is exactly similar to the working as seen in the previous pizza production. I have written a different style of query, which seems complex, but I wanted to show some variations of inner queries.

❖ Wine Suggestions View (***wine_suggestions***):



```
CREATE VIEW wine_suggestions as
SELECT
    d.pizza_name_id, w.wine_name
FROM
    pizza_dev_table d
        INNER JOIN
    (SELECT meat_name AS ingred_name, pairing_id FROM meat_ingredients
    JOIN meat_properties ON meat_ingredients.meat_id = meat_properties.meat_id UNION
    SELECT veg_name AS ingred_name, pairing_id FROM veg_ingredients
    JOIN veg_properties ON veg_ingredients.veg_id = veg_properties.veg_id UNION
    SELECT cheese_name AS ingred_name, pairing_id
    FROM cheese_ingredients
    JOIN cheese_properties ON cheese_ingredients.cheese_id = cheese_properties.cheese_id UNION
    SELECT sauce_name AS ingred_name, pairing_id FROM sauce_ingredients
    JOIN sauce_properties ON sauce_ingredients.sauce_id = sauce_properties.sauce_id UNION
    SELECT base_name AS ingred_name, pairing_id FROM base_ingredients
    JOIN base_properties ON base_ingredients.base_id = base_properties.base_id) AS ingred_tab ON d.ingred_name = ingred_tab.ingred_name
        INNER JOIN
    wines w ON ingred_tab.pairing_id = w.pairing_id
WHERE
    w.pairing_id IN (ingred_tab.pairing_id)
GROUP BY w.wine_name , d.pizza_name_id
HAVING (COUNT(DISTINCT w.pairing_id) >= 3)
ORDER BY d.pizza_name_id;
```

Figure 8 - Wine Suggestions View

Output:

pizza_name_id	wine_name
PZ002	Chenin Blanc
PZ002	Gewurztraminer
PZ002	Muscat Blanc
PZ003	Cabernet Sauvignon
PZ003	Chardonnay
PZ003	Chenin Blanc

As seen in fig. 8., the wine suggestions view consists of the wines for the particular pizza. I have included the suggestions as a view for the same reason, as it is kind of a look up table, which only needs a read-only structure, since we don't want anyone to change the suggested wines for the pizza id manually. So, offering a view for wine suggestions is a best practice is what I feel. This view is rather complex, which is an amalgamation of the pizzas in dev table and wines table along with a query derived table alias as ingred_tab, which contains the pairing id to be mapped with the pairing id in wines table.

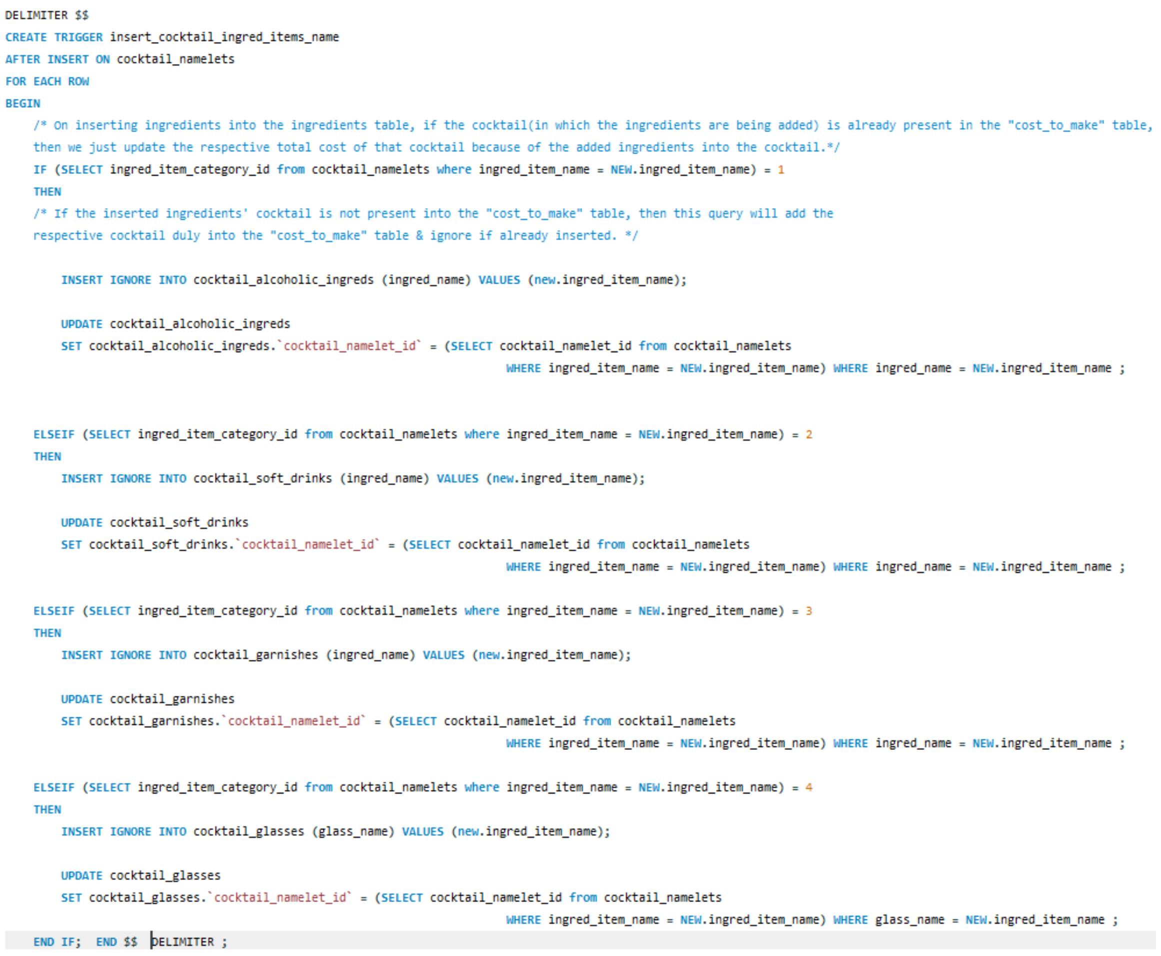
Apart from the 3 views which I have created, we can intuitively create much more multiple views. For example, we could create one view for every base table that is exactly similar to the base table but with a different name. In this way you prevent the end users from seeing the base tables and do not tell the end users the table names for security purposes [1]. In this way, it becomes easy for the users to just see the table contents abstractly, as they are not interested in the logic behind the complex queries [6]. Views also helps us to follow business rules and consistency of business logic. We can also use them to make database changes, like I have used the views for creating stored procedures, which I will discuss in the next section. The biggest advantage is that the views don't take a lot of space, since every time you call a view, you'll run the related query, hence, you don't lose disk space on views [6]. The best thing about views is that, we can mold a table, extract which ever attributes from a table we want and display it in a view for restricted user access. Hence, the views which I created satisfy these notions.

5 PROCEDURAL ELEMENTS

In this section, I will be discussing about the stored procedures and the triggers used in the Wonka Labs database. Stored procedures in SQL are prepared SQL code which may contain multiple tables, temporary tables, any complex queries, views, and many more sql code can be reused over and over again [8]. It can be called like regular programming functions which are called in interpreted and compiled programming languages. We can also pass input parameters to a store procedure, which makes the stored procedure act based on the parameter values which are passed or called. Tiggers on the other hand are also called as procedural sql code which is automatically executed in response to certain events on a specified table [7]. Triggers are essential to maintain integrity of data in a database. A trigger in SQL works similar to real-world trigger, similar to a gun trigger is pulled off to fire a bullet [9]. Triggers are also essential when we need to handle errors from the database layer. One change in a table can be linked to manipulations in multiple tables on a trigger to the master table. We don't need to wait for scheduled events to run [9], since the triggers are invoked automatically before or after a manipulation is made to the data in a table.

For the Wonka Labs database, I have demonstrated a trigger and 3 procedural elements for this project. I would've added more procedural elements, but time was a constraint! I will explain this as we go along:

❖ Inserting cocktail ingredients to the master table, automatically inserts data in the corresponding 4 ingredients tables of the cocktail realm (***insert_cocktail_ingred_items_name***): As discussed earlier in the previous sections, I made the cocktail_namelets the master table, which is kind of a look-up table manually created to push elements to the ancillary tables of the cocktail development realm. It consists of the ingredient item name, along with its ingredient category id (1: Alcohol, 2: Soft Drink, 3: Garnishes, 4: Glasses), and the given first and second namelets for the cocktails in production table (view). Now, the ingredients are pushed to the development by looking up the respective *ingredients'* tables of the 4 types of ingredient items in the cocktail realm. So, we can directly insert the ingredients into these 4 tables, by giving a after insert trigger on the *cocktail_namelets*, as seen in fig. 9.



```
DELIMITER $$
CREATE TRIGGER insert_cocktail_ingred_items_name
AFTER INSERT ON cocktail_namelets
FOR EACH ROW
BEGIN
    /* On inserting ingredients into the ingredients table, if the cocktail(in which the ingredients are being added) is already present in the "cost_to_make" table,
    then we just update the respective total cost of that cocktail because of the added ingredients into the cocktail.*/
    IF (SELECT ingred_item_category_id from cocktail_namelets where ingred_item_name = NEW.ingred_item_name) = 1
    THEN
    /* If the inserted ingredients' cocktail is not present into the "cost_to_make" table, then this query will add the
    respective cocktail duly into the "cost_to_make" table & ignore if already inserted. */

        INSERT IGNORE INTO cocktail_alcoholic_ingreds (ingred_name) VALUES (new.ingred_item_name);

        UPDATE cocktail_alcoholic_ingreds
        SET cocktail_alcoholic_ingreds.`cocktail_namelet_id` = (SELECT cocktail_namelet_id from cocktail_namelets
                                                    WHERE ingred_item_name = NEW.ingred_item_name) WHERE ingred_name = NEW.ingred_item_name ;


    ELSEIF (SELECT ingred_item_category_id from cocktail_namelets where ingred_item_name = NEW.ingred_item_name) = 2
    THEN
        INSERT IGNORE INTO cocktail_soft_drinks (ingred_name) VALUES (new.ingred_item_name);

        UPDATE cocktail_soft_drinks
        SET cocktail_soft_drinks.`cocktail_namelet_id` = (SELECT cocktail_namelet_id from cocktail_namelets
                                                    WHERE ingred_item_name = NEW.ingred_item_name) WHERE ingred_name = NEW.ingred_item_name ;

    ELSEIF (SELECT ingred_item_category_id from cocktail_namelets where ingred_item_name = NEW.ingred_item_name) = 3
    THEN
        INSERT IGNORE INTO cocktail_garnishes (ingred_name) VALUES (new.ingred_item_name);

        UPDATE cocktail_garnishes
        SET cocktail_garnishes.`cocktail_namelet_id` = (SELECT cocktail_namelet_id from cocktail_namelets
                                                    WHERE ingred_item_name = NEW.ingred_item_name) WHERE ingred_name = NEW.ingred_item_name ;

    ELSEIF (SELECT ingred_item_category_id from cocktail_namelets where ingred_item_name = NEW.ingred_item_name) = 4
    THEN
        INSERT IGNORE INTO cocktail_glasses (glass_name) VALUES (new.ingred_item_name);

        UPDATE cocktail_glasses
        SET cocktail_glasses.`cocktail_namelet_id` = (SELECT cocktail_namelet_id from cocktail_namelets
                                                    WHERE ingred_item_name = NEW.ingred_item_name) WHERE glass_name = NEW.ingred_item_name ;
    END IF;  END $$  DELIMITER ;
```

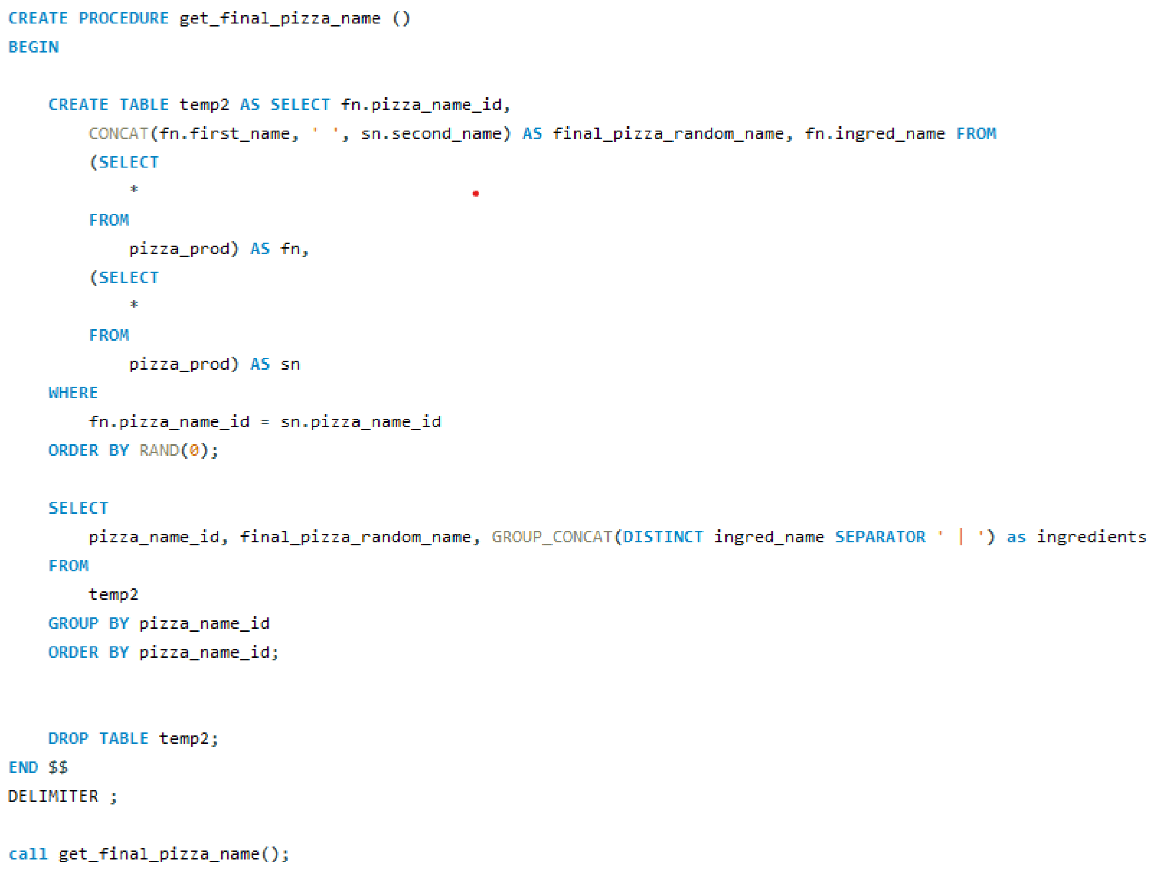
Figure 9 - Trigger for inserting ingredients into the cocktail ingredients' tables

Output:

ingred_name	cocktail_namelet_id
Green Chartreuse	12
Jamaican Rum	13
London Dry Gin	14

The above output table is of the *cocktail_alcoholic_ingreds* table. Similar outputs are in the soft drinks, garnishes and glasses tables with their appropriate cocktail namelet id. These cocktail_namelet_id's along with the corresponding ingred_name and first & second namelets are pushed to the cocktail development table. For handling appropriate entries into the appropriate ingredient item category table, I used if else conditions inside the trigger procedure sql code. I'm also checking whether the inserted ingredient is already present or not, if it is present, it will get updated. We can also add more triggers like after/before delete and update for the same tables, if we want to delete or update the 4 ingredients tables on delete/update of the master cocktail namelets table. For this project scope and page length constraint, I have added only 1 trigger.

- ❖ Retrieving a final random name for the pizza name id, from the list of first and second namelets from the production table (***get_final_pizza_name()***):



```sql
CREATE PROCEDURE get_final_pizza_name ()
BEGIN

    CREATE TABLE temp2 AS SELECT fn.pizza_name_id,
        CONCAT(fn.first_name, ' ', sn.second_name) AS final_pizza_random_name, fn.ingred_name FROM
        (SELECT
            *
        FROM
            pizza_prod) AS fn,
        (SELECT
            *
        FROM
            pizza_prod) AS sn
    WHERE
        fn.pizza_name_id = sn.pizza_name_id
    ORDER BY RAND(0);

    SELECT
        pizza_name_id, final_pizza_random_name, GROUP_CONCAT(DISTINCT ingred_name SEPARATOR ' | ') as ingredients
    FROM
        temp2
    GROUP BY pizza_name_id
    ORDER BY pizza_name_id;


    DROP TABLE temp2;
END $$
DELIMITER ;

call get_final_pizza_name();
```

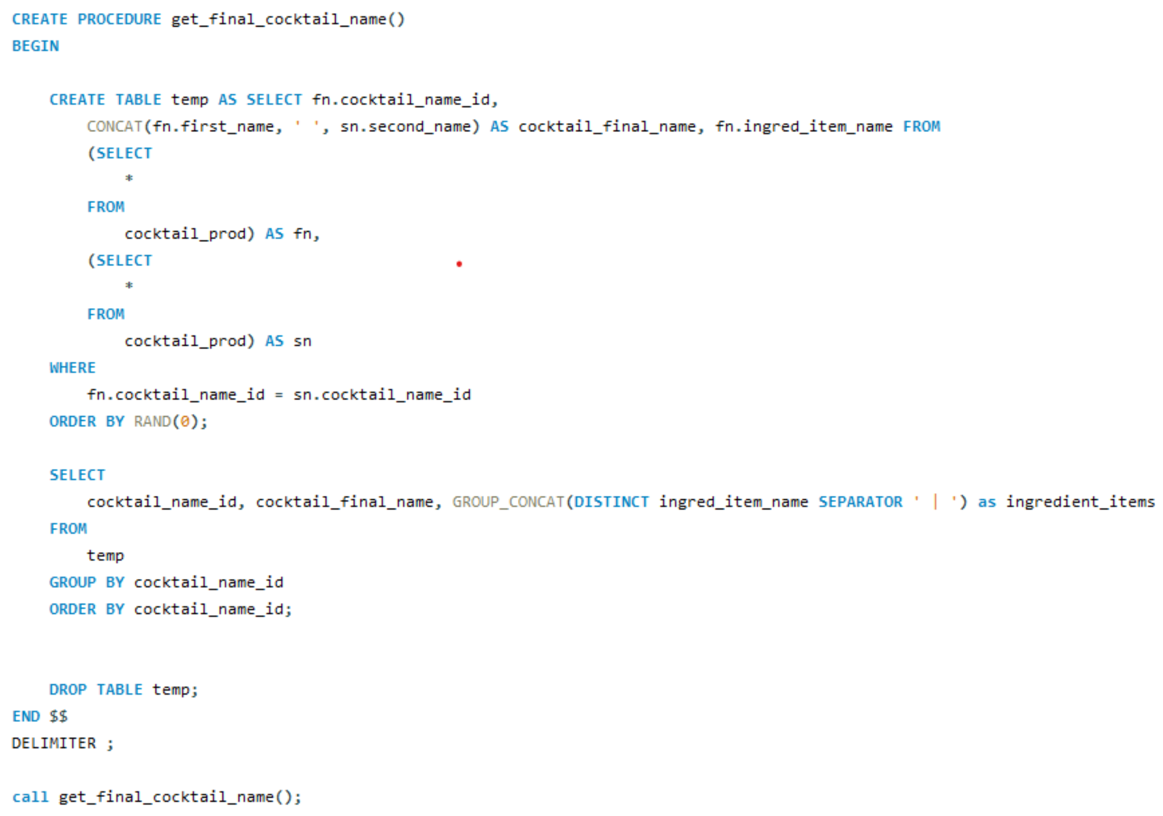
Figure 10 - Stored Proc. for retrieving final pizza name

Output:

pizza_name_id	final_pizza_random_name	ingredients
PZ001	Red Light Autumn	Beef Jerky | Bell Peppers | Black Pudding | Capers | Chimichurri Sauce | Edam | French Onions | Thick Crust

As seen in fig. 10., I have written an efficient and unique stored procedure, which creates a temporary table, and randomly concatenates the first name from the *first_name* column and any second name from the *second_name* column. I achieved this randomness by the help of using the rand() function, which randomly selects a column value. I have also given the seed as 0, to offer reproducibility, i.e., whenever we run the stored proc. the resultset will be the same. I have also used concat() with a separator(|), for getting the ingredient items for the particular pizza. By calling this procedure, I'm doing the work of the marketing team heuristically in a relational programming language. Since, the marketing team will also choose a first name of an ingredient category and some other second name of a different ingredient category. Hence, the stored proc. acts in disguise of the marketing team.

❖ Retrieving a final random name for the cocktail name id, from the list of first and second namelets from the production table (***get_final_cocktail_name()***): This stored proc. is similar to the one present above, since the cocktail and pizza entities are composite in nature in the wonka labs database. As seen in fig. 11., I have written an efficient and unique stored procedure, which creates a temporary table, and randomly concatenates the first name from the *first_name* column and any second name from the *second_name* column. I achieved this randomness by the help of using the rand() function, which randomly selects a column value. I have also given the seed as 0, to offer reproducibility, i.e., whenever we run the stored proc. the resultset will be the same. I have also used concat() with a separator(|), for getting the ingredient items for the particular cocktail. By calling this procedure, I'm doing the work of the marketing team heuristically in a relational programming language. Since, the marketing team will also choose a first name of an ingredient category and some other second name of a different ingredient category. Hence, the stored proc. acts in disguise of the marketing team to choose the official cocktail name for the cocktail name id.



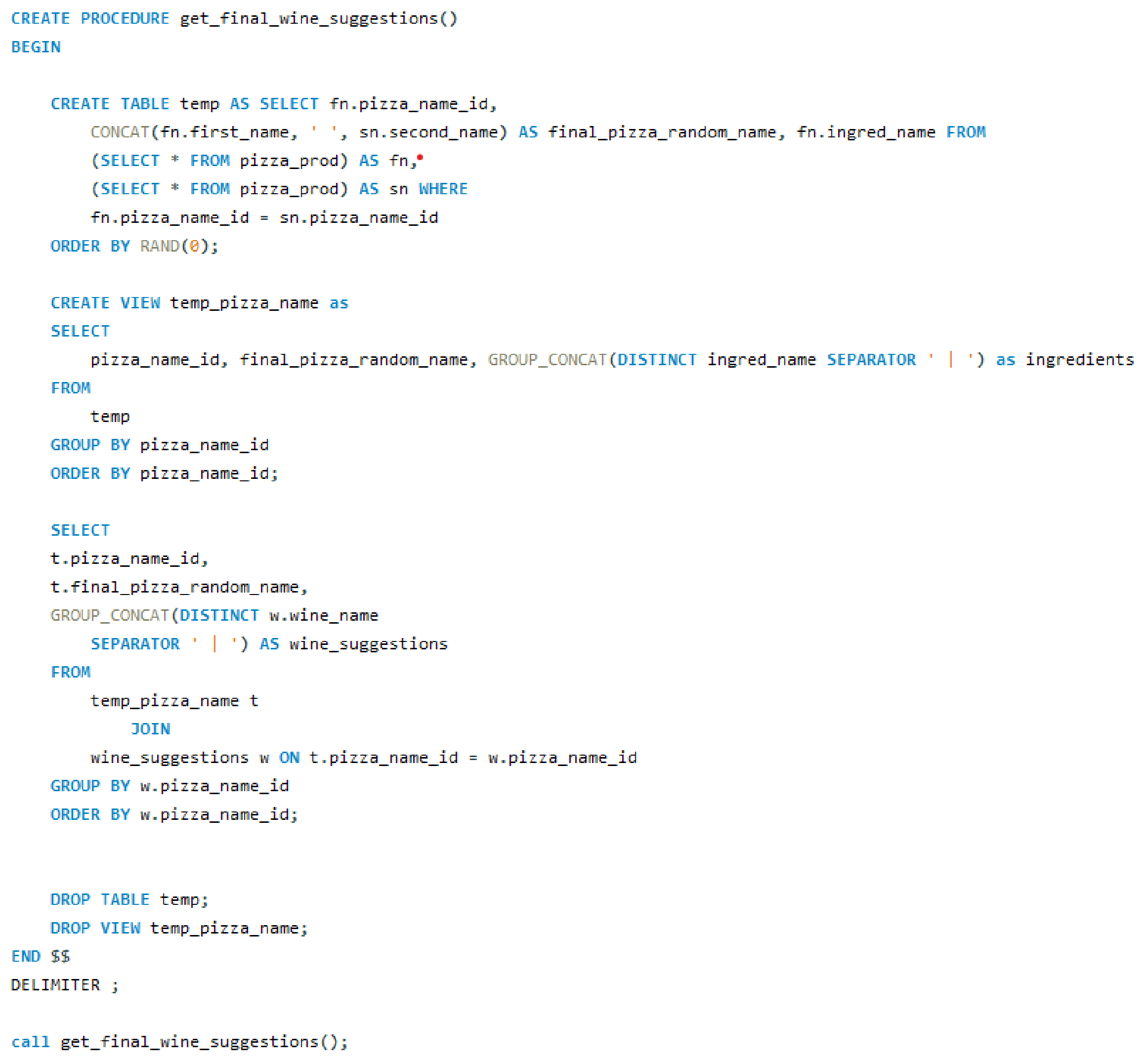
```
CREATE PROCEDURE get_final_cocktail_name()
BEGIN

    CREATE TABLE temp AS SELECT fn.cocktail_name_id,
        CONCAT(fn.first_name, ' ', sn.second_name) AS cocktail_final_name, fn.ingred_item_name FROM
        (SELECT
            *
        FROM
            cocktail_prod) AS fn,
        (SELECT
            *
        FROM
            cocktail_prod) AS sn
    WHERE
        fn.cocktail_name_id = sn.cocktail_name_id
    ORDER BY RAND(0);

    SELECT
        cocktail_name_id, cocktail_final_name, GROUP_CONCAT(DISTINCT ingred_item_name SEPARATOR ' | ') as ingredient_items
    FROM
        temp
    GROUP BY cocktail_name_id
    ORDER BY cocktail_name_id;


    DROP TABLE temp;
END $$
DELIMITER ;

call get_final_cocktail_name();
```

Figure 11 - Stored Proc. for retrieving final cocktail name

Output:

cocktail_name_id	cocktail_final_name	Ingredient_items
CKT001	Flight Gimlet	Brandy | Cocktail Cherry | Fresh Lime Juice | Highball Glass | Lemon Slice | Maraschino Luxardo | Simple Syrup

❖ Retrieving the final list of wine suggestions for the final pizza name (***get_final_wine_suggestions()***): As seen in fig. 12., this is the most wholesome stored procedure in the whole database schema, it creates a temporary table and a view inside the stored proc. in order to retrieve the final pizza name and the wine suggestions respectively. The resultset clearly shows us the relationship between the wines and the pizzas and how for a particular pizza, a list of wines is suggested, which is what expected in the white paper. Here also, I have used the rand and the concat function to retrieve the values.



```
CREATE PROCEDURE get_final_wine_suggestions()
BEGIN

    CREATE TABLE temp AS SELECT fn.pizza_name_id,
        CONCAT(fn.first_name, ' ', sn.second_name) AS final_pizza_random_name, fn.ingred_name FROM
        (SELECT * FROM pizza_prod) AS fn,
        (SELECT * FROM pizza_prod) AS sn WHERE
        fn.pizza_name_id = sn.pizza_name_id
    ORDER BY RAND(0);

    CREATE VIEW temp_pizza_name as
    SELECT
        pizza_name_id, final_pizza_random_name, GROUP_CONCAT(DISTINCT ingred_name SEPARATOR ' | ') as ingredients
    FROM
        temp
    GROUP BY pizza_name_id
    ORDER BY pizza_name_id;

    SELECT
    t.pizza_name_id,
    t.final_pizza_random_name,
    GROUP_CONCAT(DISTINCT w.wine_name
        SEPARATOR ' | ') AS wine_suggestions
    FROM
        temp_pizza_name t
            JOIN
        wine_suggestions w ON t.pizza_name_id = w.pizza_name_id
    GROUP BY w.pizza_name_id
    ORDER BY w.pizza_name_id;


    DROP TABLE temp;
    DROP VIEW temp_pizza_name;
END $$
DELIMITER ;

call get_final_wine_suggestions();
```

Figure 12 - Stored Proc. for retrieving final wine suggestions with final pizza name

Output:

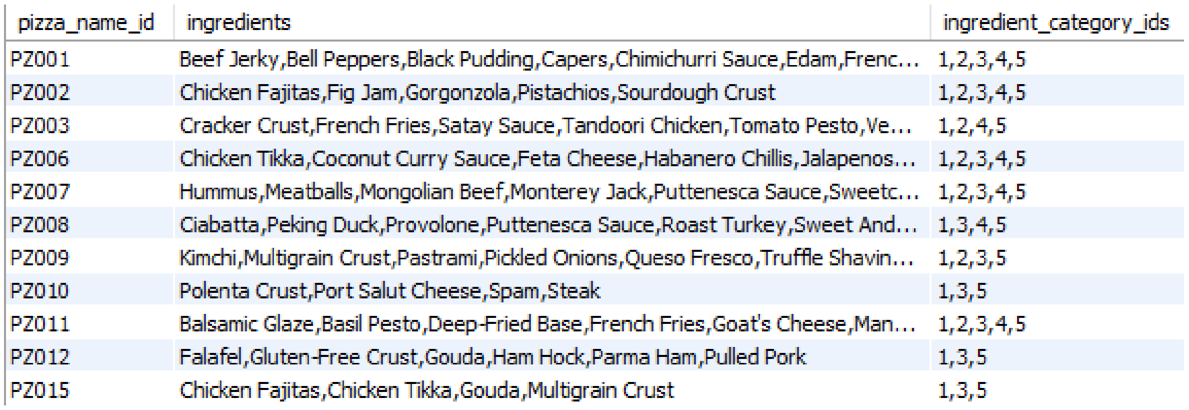
pizza_name_id	final_pizza_random_name	ingredient_items
PZ001	Red Light Autumn	Cabernet Sauvignon | Chardonnay | Chenin Blanc | Gewurztraminer | Malbec | Muscat Blanc | Pinot Noir

6 DEMONSTRATION OF SQL QUERIES

In this section I will be demonstrating some queries which evidently shows the Wonka Labs database into action. Also, there are multiple cocktail ingredients and pizza ingredients (dummy data) being added into the development tables in each realm for sufficient query resultset. There are various DML queries which can be implemented to get wholesome and useful results which can be used for future analysis. Following are some of the queries which returns results on the conditions given in the SQL select queries:

❖ **QUERY 1: Returns the pizzas from the development table which compulsorily have meat in it.**

```
SELECT d.pizza_name_id, GROUP_CONCAT(d.ingred_name) as ingredients,
    GROUP_CONCAT(DISTINCT d.ingred_category_id) as ingredient_category_ids
FROM pizza_dev_table d
WHERE d.pizza_name_id in (SELECT d.pizza_name_id
FROM pizza_dev_table d WHERE d.ingred_category_id = 1)
group by d.pizza_name_id
ORDER BY d.pizza_name_id;
```

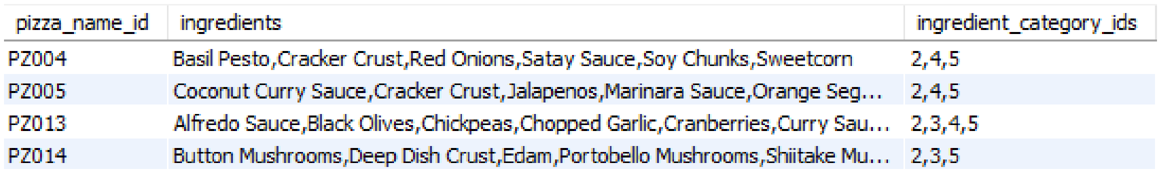


pizza_name_id	ingredients	ingredient_category_ids
PZ001	Beef Jerky,Bell Peppers,Black Pudding,Capers,Chimichurri Sauce,Edam,Frenc...	1,2,3,4,5
PZ002	Chicken Fajitas,Fig Jam,Gorgonzola,Pistachios,Sourdough Crust	1,2,3,4,5
PZ003	Cracker Crust,French Fries,Satay Sauce,Tandoori Chicken,Tomato Pesto,Ve...	1,2,4,5
PZ006	Chicken Tikka,Coconut Curry Sauce,Feta Cheese,Habanero Chillis,Jalapenos...	1,2,3,4,5
PZ007	Hummus,Meatballs,Mongolian Beef,Monterey Jack,Puttenesca Sauce,Sweetc...	1,2,3,4,5
PZ008	Ciabatta,Peking Duck,Provolone,Puttenesca Sauce,Roast Turkey,Sweet And...	1,3,4,5
PZ009	Kimchi,Multigrain Crust,Pastrami,Pickled Onions,Queso Fresco,Truffle Shavin...	1,2,3,5
PZ010	Polenta Crust,Port Salut Cheese,Spam,Steak	1,3,5
PZ011	Balsamic Glaze,Basil Pesto,Deep-Fried Base,French Fries,Goat's Cheese,Man...	1,2,3,4,5
PZ012	Falafel,Gluten-Free Crust,Gouda,Ham Hock,Parma Ham,Pulled Pork	1,3,5
PZ015	Chicken Fajitas,Chicken Tikka,Gouda,Multigrain Crust	1,3,5

Figure 13 - Query 1

❖ **QUERY 2: Returns only the vegetarian pizzas from the development table.**

```
SELECT d.pizza_name_id,  GROUP_CONCAT(d.ingred_name) as ingredients,
    GROUP_CONCAT(DISTINCT d.ingred_category_id) as ingredient_category_ids
FROM pizza_dev_table d
WHERE d.pizza_name_id not in (SELECT d.pizza_name_id
FROM pizza_dev_table d WHERE d.ingred_category_id = 1)
group by d.pizza_name_id
ORDER BY d.pizza_name_id;
```

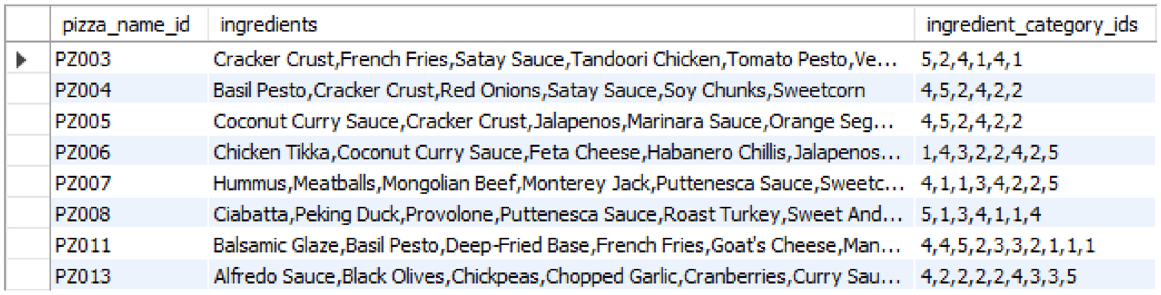


pizza_name_id	ingredients	ingredient_category_ids
PZ004	Basil Pesto,Cracker Crust,Red Onions,Satay Sauce,Soy Chunks,Sweetcorn	2,4,5
PZ005	Coconut Curry Sauce,Cracker Crust,Jalapenos,Marinara Sauce,Orange Seg...	2,4,5
PZ013	Alfredo Sauce,Black Olives,Chickpeas,Chopped Garlic,Cranberries,Curry Sau...	2,3,4,5
PZ014	Button Mushrooms,Deep Dish Crust,Edam,Portobello Mushrooms,Shiitake Mu...	2,3,5

Figure 14 - Query 2

❖ **QUERY 3: Returns the pizzas which have more than 1 sauce.**

```
SELECT d.pizza_name_id,  GROUP_CONCAT(d.ingred_name) as ingredients,
    GROUP_CONCAT(d.ingred_category_id) as ingredient_category_ids
FROM pizza_dev_table d
WHERE d.pizza_name_id in (SELECT d.pizza_name_id
FROM pizza_dev_table d WHERE d.ingred_category_id = 4 group by d.pizza_name_id HAVING COUNT(*) > 1 )
group by d.pizza_name_id
ORDER BY d.pizza_name_id;
```

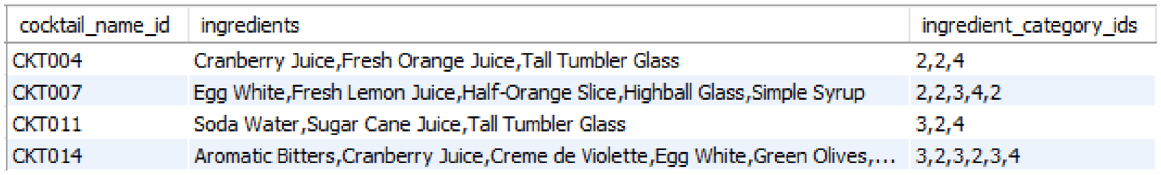


	pizza_name_id	ingredients	ingredient_category_ids
▶	PZ003	Cracker Crust,French Fries,Satay Sauce,Tandoori Chicken,Tomato Pesto,Ve...	5,2,4,1,4,1
	PZ004	Basil Pesto,Cracker Crust,Red Onions,Satay Sauce,Soy Chunks,Sweetcorn	4,5,2,4,2,2
	PZ005	Coconut Curry Sauce,Cracker Crust,Jalapenos,Marinara Sauce,Orange Seg...	4,5,2,4,2,2
	PZ006	Chicken Tikka,Coconut Curry Sauce,Feta Cheese,Habanero Chillis,Jalapenos...	1,4,3,2,2,4,2,5
	PZ007	Hummus,Meatballs,Mongolian Beef,Monterey Jack,Puttenesca Sauce,Sweetc...	4,1,1,3,4,2,2,5
	PZ008	Ciabatta,Peking Duck,Provolone,Puttenesca Sauce,Roast Turkey,Sweet And...	5,1,3,4,1,1,4
	PZ011	Balsamic Glaze,Basil Pesto,Deep-Fried Base,French Fries,Goat's Cheese,Man...	4,4,5,2,3,3,2,1,1,1
	PZ013	Alfredo Sauce,Black Olives,Chickpeas,Chopped Garlic,Cranberries,Curry Sau...	4,2,2,2,2,4,3,3,5

Figure 15 - Query 3

❖ **QUERY 4: Returns the cocktails which only have non-alcoholic ingredients (Mocktails).**

```
SELECT d.cocktail_name_id,  GROUP_CONCAT(d.ingred_item_name) as ingredients,
    GROUP_CONCAT(d.ingred_item_category_id) as ingredient_category_ids
FROM cocktail_dev_table d
WHERE d.cocktail_name_id not in (SELECT d.cocktail_name_id
FROM cocktail_dev_table d WHERE d.ingred_item_category_id = 1  )
group by d.cocktail_name_id
ORDER BY d.cocktail_name_id;
```

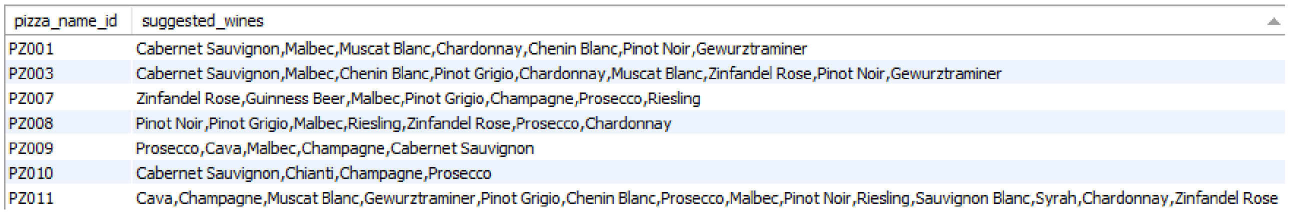


cocktail_name_id	ingredients	ingredient_category_ids
CKT004	Cranberry Juice,Fresh Orange Juice,Tall Tumbler Glass	2,2,4
CKT007	Egg White,Fresh Lemon Juice,Half-Orange Slice,Highball Glass,Simple Syrup	2,2,3,4,2
CKT011	Soda Water,Sugar Cane Juice,Tall Tumbler Glass	3,2,4
CKT014	Aromatic Bitters,Cranberry Juice,Creme de Violette,Egg White,Green Olives,...	3,2,3,2,3,4

Figure 16 - Query 4

- **QUERY 5: Returns the wine suggestions of the pizzas which have more than 3 wines suggested.**

```
SELECT pizza_name_id, GROUP_CONCAT(wine_name) as suggested_wines
FROM wine_suggestions
group by pizza_name_id
having count(*) >3
order by pizza_name_id;
```



pizza_name_id	suggested_wines
PZ001	Cabernet Sauvignon,Malbec,Muscat Blanc,Chardonnay,Chenin Blanc,Pinot Noir,Gewurztraminer
PZ003	Cabernet Sauvignon,Malbec,Chenin Blanc,Pinot Grigio,Chardonnay,Muscat Blanc,Zinfandel Rose,Pinot Noir,Gewurztraminer
PZ007	Zinfandel Rose,Guinness Beer,Malbec,Pinot Grigio,Champagne,Prosecco,Riesling
PZ008	Pinot Noir,Pinot Grigio,Malbec,Riesling,Zinfandel Rose,Prosecco,Chardonnay
PZ009	Prosecco,Cava,Malbec,Champagne,Cabernet Sauvignon
PZ010	Cabernet Sauvignon,Chianti,Champagne,Prosecco
PZ011	Cava,Champagne,Muscat Blanc,Gewurztraminer,Pinot Grigio,Chenin Blanc,Prosecco,Malbec,Pinot Noir,Riesling,Sauvignon Blanc,Syrah,Chardonnay,Zinfandel Rose

Figure 17 - Query 5

Apart from the queries given above, we can add more such similar or some different queries to have a look at various other implicit dimensions of the database schema. The complex stored procedures as well as views as discussed in the previous sections, can also be considered as complex sql code queries, which can be generated using just a select query statement. Herein, I have used GROUP_CONCAT to get the list of pizza ingredients/cocktail ingredients/suggested wines in accordance to the condition supplied, i.e., whether the pizza is meat lover's pizza or vegetarian, or whether the cocktail contains only alcoholic drink or not, and so on. I have used group_concat and sub-quering since, as we are writing Data Manipulation Language statements (SELECT), we are just displaying the resultset and we do not intend to store the resultset in the schema, so rather than having a single value for each cell, for displaying purposes, we can surely make the efficient use of group_concat function.

7 CONCLUSION & FUTURE SCOPE

This report paper comprehensively gets you immersed into the world of Wonka Labs baked savory products and adult beverages. The database schema and its design has been implemented taking all the specifications and guidelines prescribed in the white paper. The schema which we have seen in the previous sections is indeed efficient, normalized, secure and flexible to any other bakery or beverage related additions or manipulations. It follows the norms of atomicity, integrity, consistency, isolation, and durability. With the help of such schema, the end-to-end application for Wonka Labs will surely be less cumbersome as a general idea of how the workflow is regularized is thoroughly elaborated in the SQL code as well as in this report. The database design consists of various elementary and intermediate elements of Structured Query Language like the DDL, DML statements, subqueries, joins, database triggers, stored procedures, virtual tables (views), temporary tables, reproducibility (random seed), and many more such phenomena. Even though, the final working of development and production of these products will be handled by the application, but in case of any logical failure or code fault, the database schema will act like a seabed to handle any errors or faults.

Apart from the overarching schema and its elements, discussed in this report, there are much more other things which I would have done if time wasn't a constraint! We have already satisfied the constraints in the white paper, but adding more to it – we can even add the costs, measures, sizes, orders, customers, employees and all the other underrated pillars of the Wonka Labs Food Innovations. The database schema is so flexibly cohesive and loosely coupled that any addition like costs or measures of cocktails/pizzas can intuitively be added post the production tables; once the marketing team has finalized the official names of all the pizzas/cocktails, we can use this relational database as a look up table for the customers buying the pizzas/cocktails, and we can easily link these tables using referencing and unique constraints.

ACKNOWLEDGEMENTS

I hereby acknowledge that this project has been accomplished just by myself and all the references to research papers, sql blogs or websites have been thoroughly mentioned at the end in the References section. In the end, I would like to thank Prof. Tony Veale, TA Du Xiaoyu, along with TA Blanzeisky William & TA Jin Changhong at University College Dublin and any other anonymous referees for giving me the opportunity to perform this comprehensive task & to the references, which helped me form technical ideas & improve the quality of the project implementation.

REFERENCES

[1] Jan Harrington; (2021), Relational Database Design and Implementation 4th Edition, eBook ISBN: 9780128499023, © Morgan Kaufmann 2016

[2] Foreign key in DBMS (2015).

[3] SQL Constraints (2014).

[4] Relational databases: Defining relationships between database tables (2003).

[5] What is Normalization? 1NF, 2NF, 3NF, BCNF Database Example (2018).

[6] Drkusic, E. (2020); Learn SQL: SQL Views, SQL Shack

[7] Atzeni, P., *et. al.,* (1999) Database Systems Conceptual, Language & Architectures. London: McGraw-Hill, pp.155-180.

[8] SQL Stored Procedures (2016).

[9] Prateek T L., (2019); Triggers in SQL Tutorial | SQL Triggers with Examples | Edureka.

[10] MySQL :: MySQL Documentation (2021).