Wonka Labs

Experiments in Food Discovery

David Mannion

16365576

david.mannion1@ucdconnect.ie

Table of Contents

List of Figures

Section 1: Introduction

As we all know, our cunning competitors Slugworth, Fickelgruber and Prodnose have caught up on Wonka Labs in terms of revenues over the past few years. Despite our many endeavors, their spies have managed to steal our precious candy secrets and use them for their own gain. We can only blame ourselves for that - the outdated system used by Wonka labs – where Oompa Loompas recorded our stocks of delightful treats, candies and snacks on sugar paper written with candy-cane pens has some serious flaws that will need to be addressed.

I propose The Relational Database Paradigm as the solution. It will help Wonka Labs to once again be at the forefront of innovation and design within the food and beverage industry, providing a solid foundation to the high-level application to be built on top of it by Wonka's Python Software Engineers that will enable the Oompla Loompas to access information on the all-new baked goods, cocktail and wine offerings of Wonka Inc quickly and efficiently.

If Wonka labs intends to be the one of the largest global exporters of Baked goods, Cocktails and Beverages in the world, it needs a large offering of products. While other systems like NoSQL could provide a data storage solution, our database will not be extremely big, limited at most to a few hundred or thousand rows of data. Hence the need for a distributed database is not needed. The relational database model is simple, flexible and provides more than enough storage for our needs.

The nature of the data stored here at Wonka labs is hierarchical by nature. A pizza is made up of multiple toppings, sauces and cheese while a cocktail is made up of multiple liquids along with some garnishes, some ice and a glass. The Relational Database model is hierarchical by design. This match makes the RDBMs model the perfect data storage solution for Wonka labs.

While developing a product, a small change to the recipe needs to be sent to many Oompa Loompas across the factory. The legacy data storage system described above reduced the efficiency of this process, due to concurrency issues whereby it was possible that different Oompa Loompas had different versions of the recipe of a product at a given time. Given the speed and frequency of development here at Wonka Labs, this inefficiency could not be tolerated any longer. The RDBMs solves this problem, meaning

Wonka Labs should soon return to the forefront of design innovation in the food and beverage industry.

The RDBMs system, through normalization, prevents data redundancy while also ensuring data integrity. This will make it ever so simple for the hard-working Oompa Loompas to access the data they need, while also being confident that the data they access is accurate, consistent over the entire life cycle of the database. This is essential for the food and beverage industry and especially for Wonka Labs. After the company's reputation was severely damaged due to the rumors of bizarre workplace injuries, the last thing Wonka Labs need is the wrong ingredient being placed into a pizza (for example incorrectly putting in a regular crust instead of a gluten free crust) and our reputation being damaged even further.

The RDBMs should also support Data Definition capabilities, enabling users to create and delete tables or views based on conditional operators and multiple fields/columns, as Wonka labs is an ambitious company who some day may move into other areas such as fine wine development, which would require more tables.

As well as this, Data Manipulation capabilities are also required, letting users use operations such as join, union and order-by to fully utilize the power of RDBMs.

As mentioned above, the spies of Slugworth and Co. also need to be dealt with. RDBMs provide multi-level security through read only access and column-level security. This would help prevent the spies from gaining access to our secret, prized recipes as only trusted Oompa Loompas would be given permission to access our data.

I bequest you to read on as we delve into the finer details of the database management system that will underpin the success of Wonka labs for years to come.

Section 2: Database Plan: A Schematic View

In this section offer a high-level view of the database and its design. State what you think the principal entities are, as well as their main attributes and the key relations that connect them. Provide an E-R diagram (entities and tables) that illustrates your plan. Motivate your design – state why this way and not another.

Principal Entities, Main Attributes and Key Relations

The principal entities in this table are the pizzas, cocktails and beverages.

The main attributes are meats, cheeses, vegetables, sauces and bases for baked goods and liquids, garnishes, ice and glasses for cocktails.

The key relations are how pizzas are linked to their ingredients, how cocktails relate to their ingredients and the bridging tables that connect them all.

Bridge Tables

I have used bridge tables throughout my database. Bridge tables are useful when a data model contains many-to-many relationships, for example many kinds of meats can be used as toppings on many different pizzas. Bridge tables transform many-to-many relationships into one-to-many relationships by adding additional tables.

Bridge tables have many advantages. One being that they properly join the tables that exist on either side of the bridge. We can use foreign keys in the bridge table to ensure that no record in the table is out of sync with the items it acts as a bridge for. To use the meat example again, the meat_allocations bridge table creates a connection between the meat table and the two pizza tables, with one column being the meat id and another being the pizza id.

In bridge tables, the relationship is mandatory. This means that bridge tables restrict the data from one subject area based on the rows that are returned from another subject area. I also considered fact tables for this design, but the key difference between them and bridge tables is that fact tables do not have this restriction because the two data sets operate as non-conformed dimensions. This means that applying a filter to one data set has no impact on the other data set.

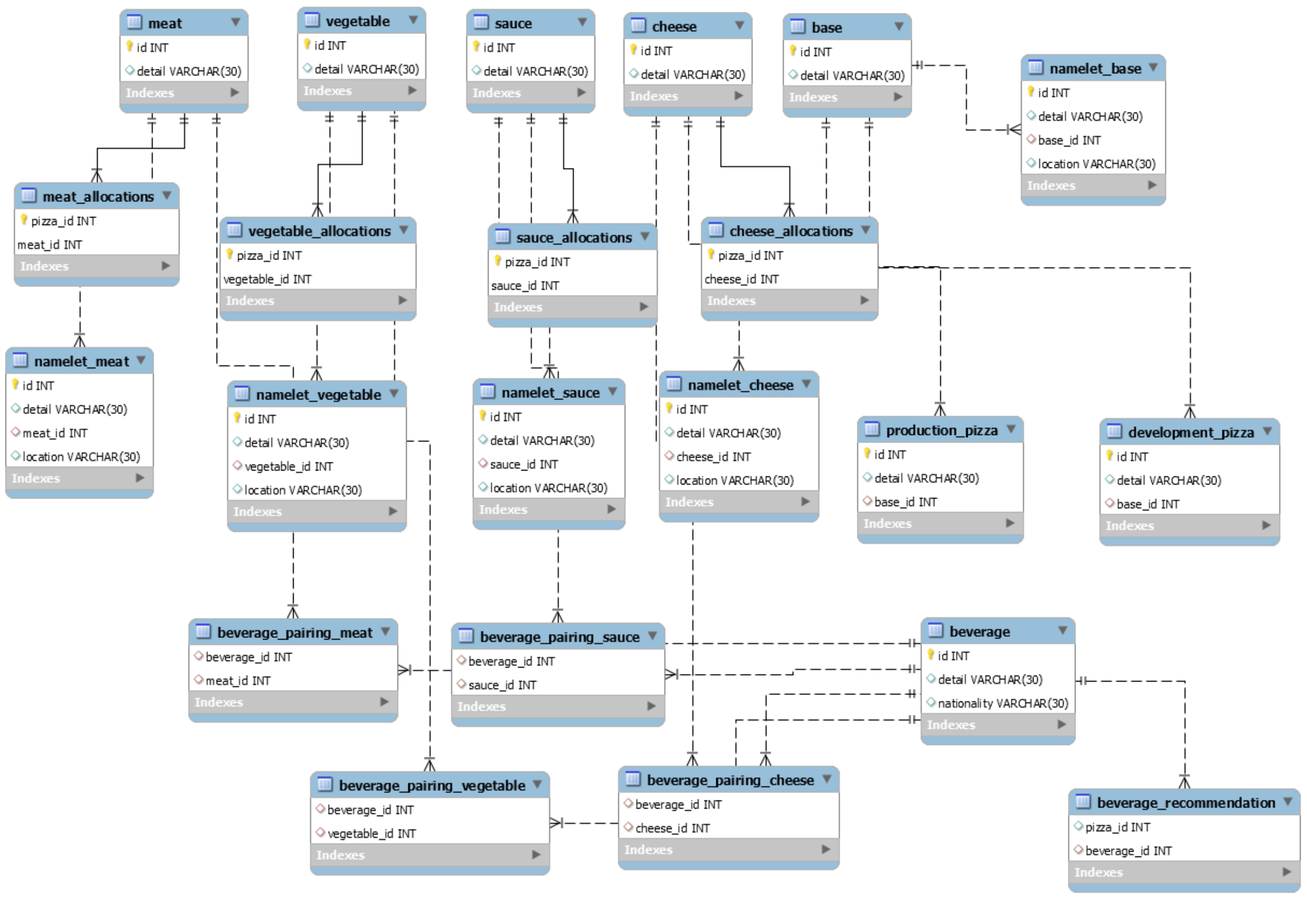


Figure 1: ER Diagram for Baked Goods and Beverages

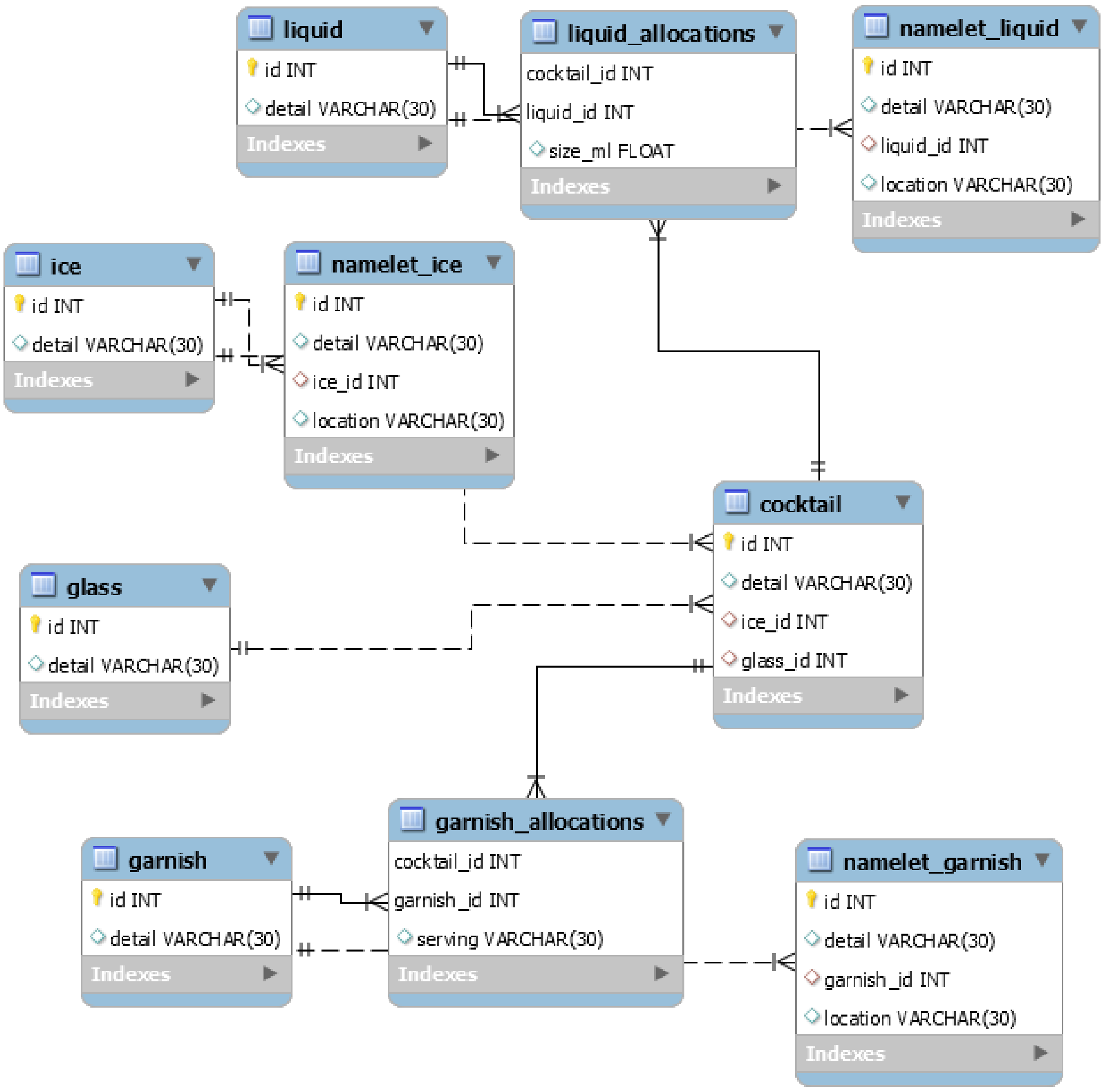


Figure 2: ER Diagram of Cocktails

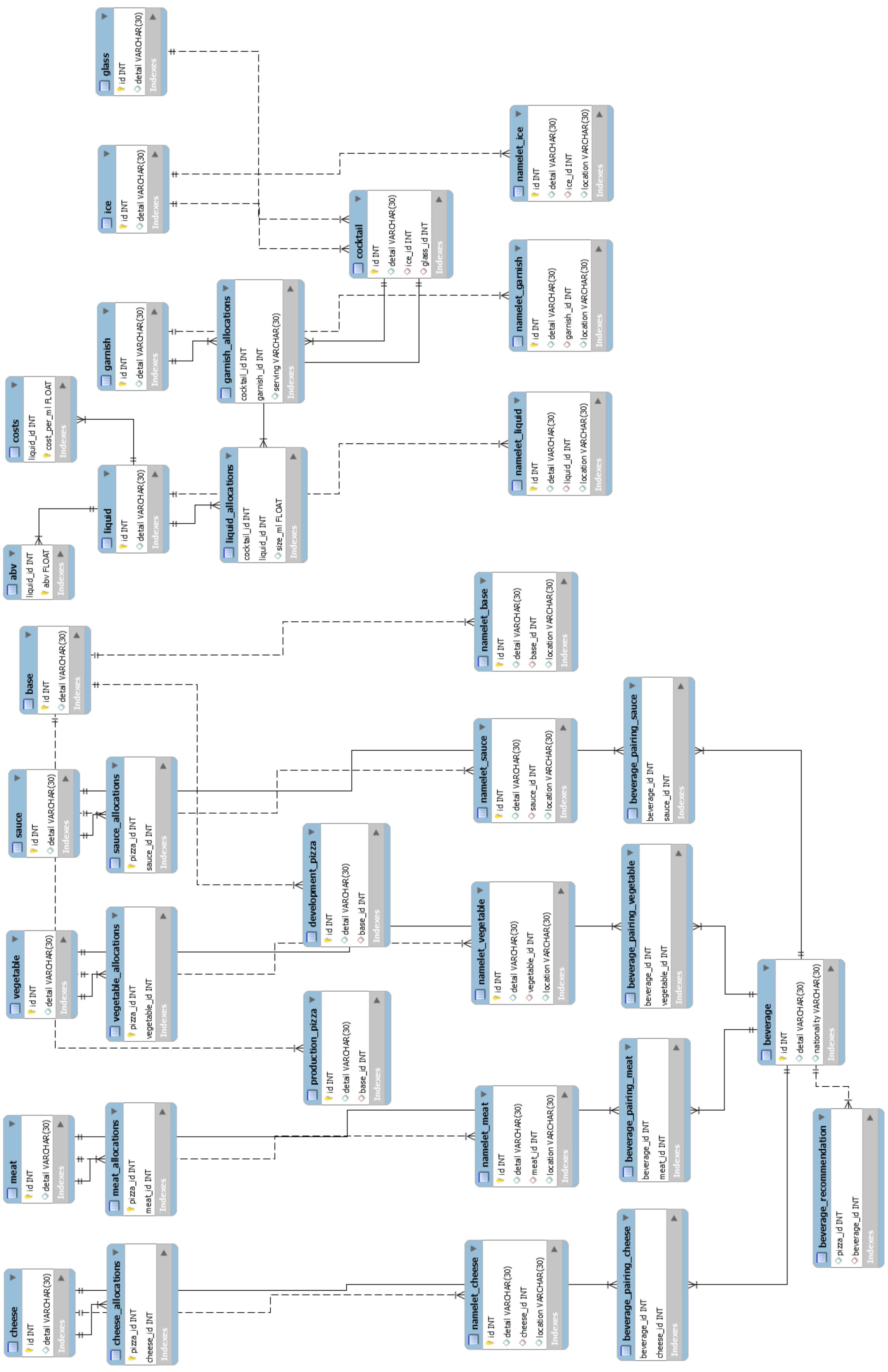
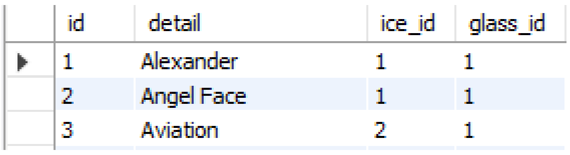


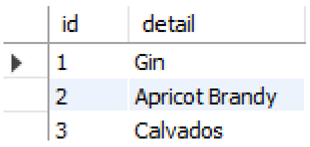
Figure 3: ER Diagram for Full Database

Section 3: Database Structure: A Normalized View

- **Cocktail:** This contains information on each cocktail sold by Wonka Labs.
  - id is the primary key. It is a value unique to this table that identifies each cocktail by an integer.
  - The detail field contains the name of each cocktail as a varchar(30). E.g. 'sidecar'
  - ice_id and glass_id are foreign keys that reference the Ice and Glass tables. Since this is a 1:N relationship, we do not need a bridging table to represent this relationship.
  - Example of table: SELECT * FROM Cocktail LIMIT 3;
  - 

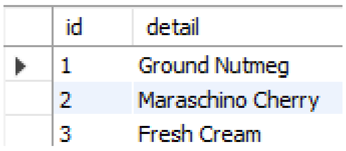
id	detail	ice_id	glass_id
1	Alexander	1	1
2	Angel Face	1	1
3	Aviation	2	1

Figure 4: Cocktail Table

- **Liquid**
  - This table contains data on the liquids that are used as ingredients in the cocktails.
  - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
  - The detail field contains the name of each liquid as a varchar(30), e.g. 'Gin'
  - Example of table: SELECT * FROM Liquid LIMIT 3;
  - 

id	detail
1	Gin
2	Apricot Brandy
3	Calvados

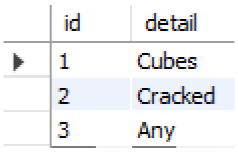
Figure 5: Liquid Table

- **Garnish**
    - This table contains data on the garnishes that are used in the cocktails.
    - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
    - The detail field contains the name of each garnish as a varchar(30), e.g. 'Orange Peel'
    - Example of table: SELECT * FROM Liquid LIMIT 3;
    - 

id	detail
1	Ground Nutmeg
2	Maraschino Cherry
3	Fresh Cream

Figure 6: Garnish Table

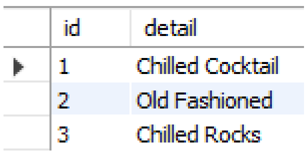
- **Ice**
    - This table contains data on the types of ice that are added to the cocktails.
    - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
    - The detail field contains the name of each Ice type as a varchar(30), e.g. 'Crushed
    - Example of table: SELECT * FROM Ice LIMIT 3;



id	detail
1	Cubes
2	Cracked
3	Any

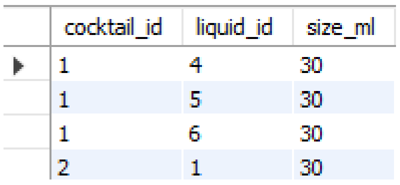
Figure 7: Ice Table

- **Glass**
    - This table contains data on the types of glass that the cocktails are stored in.
    - Id is the primary key. It is a value unique to this table(starts auto increment at 1)
    - The detail field contains the name of each liquid as a varchar(30), e.g. 'Gin'
    - Example of table: SELECT * FROM Glass LIMIT 3;

- 

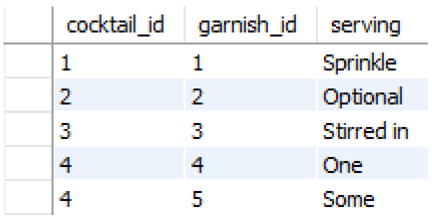
id	detail
1	Chilled Cocktail
2	Old Fashioned
3	Chilled Rocks

Figure 8: Glass Table

- **Liquid_Allocations**
  - This acts as a bridge table between cocktails and liquids.
  - It combines cocktail_id and liquid_id into a composite(multi-column) key.
  - The column size_ml is an integer that contains the amount of a given liquid in a cocktail. For example 30ml of gin.
  - Example of table: SELECT * FROM Liquid_Allocations LIMIT 4;
  - 

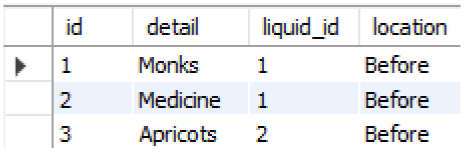
cocktail_id	liquid_id	size_ml
1	4	30
1	5	30
1	6	30
2	1	30

Figure 9: Liquid_Allocations Table

- **Garnish_Allocations**
  - This acts as a bridge table between cocktails and garnishes.
  - It uses cocktail_id and liquid_id as a composite(multi-column) key.
  - The column serving is a varchar(30) that contains the serving recommendation of a garnish for a given cocktail. For example, "on top"
  - Example of table: SELECT * FROM Liquid_Allocations LIMIT 4;
  - 

cocktail_id	garnish_id	serving
1	1	Sprinkle
2	2	Optional
3	3	Stirred in
4	4	One
4	5	Some

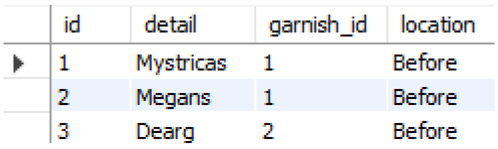
Figure 10: Garnish_Allocations Table

- **Namelet_Liquid**
    - This acts as a store of namelets for cocktail ingredients, which are used to generate name recommendations for new cocktails.
    - The primary key is the id column.
    - The detail column contains a varchar containing one namelet for a specific liquid.
    - Liquids can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the Liquids table)
    - liquid_id is a foreign key that references the liquid table's primary key, id
    - The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: "Before" and "After".
        - If a namelet has the value "before" then it can only be used as the first word in the new name by the procedure that generates names
        - If a namelet has the value "after" then it can only be used as the second word in the new name by the procedure that generates names
    - SELECT * FROM Namelet_Liquid LIMIT 4;
    - 

id	detail	liquid_id	location
1	Monks	1	Before
2	Medicine	1	Before
3	Apricots	2	Before

Figure 11: Namelet_Liquid Table

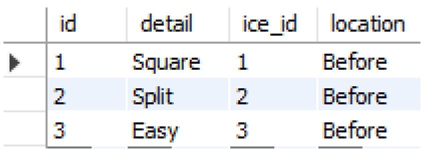
- **Namelet_Garnish**
    - This acts as a store of namelets for garnish ingredients, which are used to generate name recommendations for new cocktails.
    - The primary key is the id column.
    - The detail column contains a varchar containing one namelet for a specific garnish.
    - Garnishes can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the garnishes table)
    - garnish_id is a foreign key that references the garnish table's primary key, id

- The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: “Before” and “After”.
    - If a namelet has the value “before” then it can only be used as the first word in the new name by the procedure that generates names
    - If a namelet has the value “after” then it can only be used as the second word in the new name by the procedure that generates names
- SELECT * FROM Namelet_Garnish LIMIT 4;
- 

id	detail	garnish_id	location
1	Mystricas	1	Before
2	Megans	1	Before
3	Dearg	2	Before

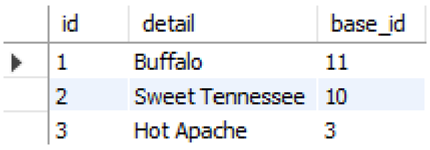
Figure 12: Namelet_Garnish Table

- **Namelet_Ice**
  - This acts as a store of namelets for ice ingredients, which are used to generate name recommendations for new cocktails.
  - The primary key is the id column.
  - The detail column contains a varchar containing one namelet for a specific type of ice.
  - Different types of Ice can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the ice table)
  - ice_id is a foreign key that references the ice table’s primary key, id
  - The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: “Before” and “After”.
    - If a namelet has the value “before” then it can only be used as the first word in the new name by the procedure that generates names
    - If a namelet has the value “after” then it can only be used as the second word in the new name by the procedure that generates names
  - SELECT * FROM Namelet_Ice LIMIT 4;

- 

id	detail	ice_id	location
1	Square	1	Before
2	Split	2	Before
3	Easy	3	Before

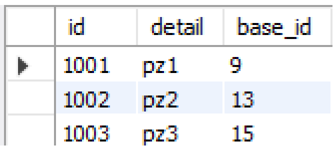
Figure 13:Namelet_Ice

- **Production_Pizza:** This contains information on the pizzas available to purchase from by Wonka Labs.
    - id is the primary key. It is a value unique to this table that identifies each cocktail by an integer.
        - This id cannot go above 1000 because ids over 1000 are reserved for development pizzas
        - This is a design feature to enable us to define both production pizzas and development pizzas
    - The detail field contains the name of each pizza as a varchar(30). E.g. 'Buffalo'
    - base_id is a foreign key that references the base table. Since this is a 1:N relationship, we do not need a bridging table to represent this relationship.
    - Example of table: SELECT * FROM Production_Pizza LIMIT 3;
    - 

id	detail	base_id
1	Buffalo	11
2	Sweet Tennessee	10
3	Hot Apache	3

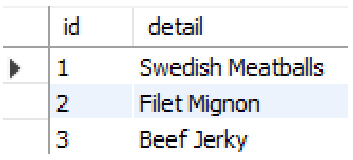
Figure 14: Production_Pizza Table

- **Development_Pizza:** This contains information on the pizzas in the development lab from by Wonka Labs. They are not available to purchase but can be moved to the production_pizza table using the Promote_Pizza procedure detailed in Section 5.
    - id is the primary key. It is a value unique to this table that identifies each cocktail by an integer.
        - Note that this id starts at 1000 instead of the typical 1.
        - This is to prevent a clash of ids with the production_pizza database.

        - For this reason, we cannot define this column as a foreign key in other parts of the database, as a foreign key must reference only one table.
    - The detail field contains the name of each pizza as a varchar(30).
        - Pizzas in this table do not have normal pizza names, rather they have codenames E.g. ‘pz1, pz2’
    - base_id is a foreign key that references the base table. Since this is a 1:N relationship, we do not need a bridging table to represent this relationship.
    - Example of table: SELECT * FROM Development_Pizza LIMIT 3;
    - 

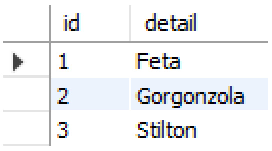
id	detail	base_id
1001	pz1	9
1002	pz2	13
1003	pz3	15

Figure 15: Development_Pizza Table

- **Meat**
    - This table contains data on the meats that are used as toppings on the pizzas.
    - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
    - The detail field contains the name of each meat as a varchar(30), e.g. ‘Swedish Meatballs’
    - Example of table: SELECT * FROM Meat LIMIT 3;
    - 

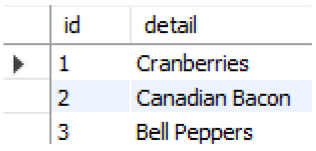
id	detail
1	Swedish Meatballs
2	Filet Mignon
3	Beef Jerky

Figure 16: Meat Table

- **Cheese**
  - This table contains data on the cheeses that are used as toppings on the pizzas.
  - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
  - The detail field contains the name of each cheese as a varchar(30), e.g. 'Gorgonzola'
  - Example of table: SELECT * FROM Cheese LIMIT 3;
  - 

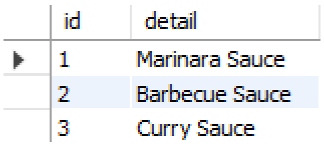
id	detail
1	Feta
2	Gorgonzola
3	Stilton

Figure 17: Cheese Table

- **Vegetable**
  - This table contains data on the vegetables that are used as toppings on the pizzas.
  - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
  - The detail field contains the name of each vegetable as a varchar(30), e.g., 'Bell Peppers'
  - Example of table: SELECT * FROM Vegetable LIMIT 3;
  - 

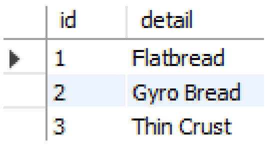
id	detail
1	Cranberries
2	Canadian Bacon
3	Bell Peppers

Figure 18: Vegetable Table

- **Sauce**
    - This table contains data on the sauces that are applied on the pizzas.
    - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
    - The detail field contains the name of each sauce as a varchar(30), e.g. 'Barbeque'
    - Example of table: SELECT * FROM Sauce LIMIT 3;
    - 

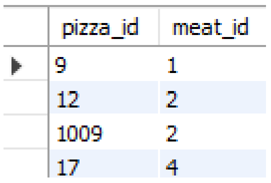
id	detail
1	Marinara Sauce
2	Barbecue Sauce
3	Curry Sauce

Figure 19: Sauce Table

- **Base**
    - This table contains data on the bases that the sauce and toppings are placed on.
    - Id is the primary key. It is a value unique to this table (starts auto increment at 1)
    - The detail field contains the name of each base as a varchar(30), e.g. 'Gluten Free Crust'
    - Example of table: SELECT * FROM Sauce LIMIT 3;
    - 

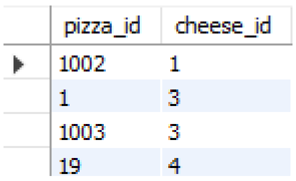
id	detail
1	Flatbread
2	Gyro Bread
3	Thin Crust

Figure 20: Base Table

- **Meat_Allocations**
    - This acts as a bridge table between pizzas and meats.
    - It combines pizza_id and meat_id into a composite(multi-column) key.
    - Example of table: SELECT * FROM Meat_Allocations LIMIT 4;
    - 

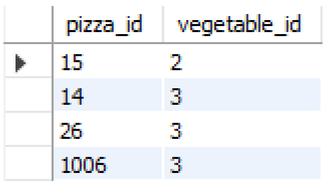
pizza_id	meat_id
9	1
12	2
1009	2
17	4

Figure 21: Meat_Allocations Table

- **Cheese_Allocations**
  - This acts as a bridge table between pizzas and meats.
  - It combines pizza_id and meat_id into a composite(multi-column) key.
  - Example of table: SELECT * FROM Meat_Allocations LIMIT 4;
  - 

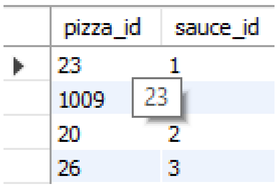
pizza_id	cheese_id
1002	1
1	3
1003	3
19	4

Figure 22: Cheese_Allocations Table

- **Vegetable_Allocations**
  - This acts as a bridge table between pizzas and meats.
  - It combines pizza_id and meat_id into a composite(multi-column) key.
  - Example of table: SELECT * FROM Meat_Allocations LIMIT 4;
  - 

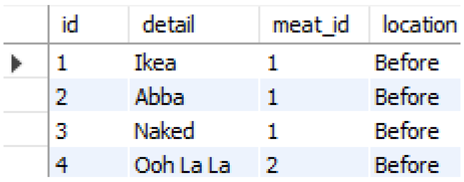
pizza_id	vegetable_id
15	2
14	3
26	3
1006	3

Figure 23: Vegetable_Allocations Table

- **Sauce_Allocations**
  - This acts as a bridge table between pizzas and meats.
  - It combines pizza_id and meat_id into a composite(multi-column) key.
  - Example of table: SELECT * FROM Meat_Allocations LIMIT 4;
  - 

pizza_id	sauce_id
23	1
1009	23
20	2
26	3

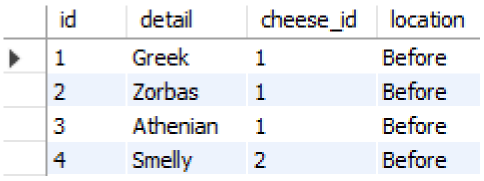
Figure 24: Sauce_Allocations Table

- **Namelet_Meat**
  - This acts as a store of namelets for meat ingredients, which are used to generate name recommendations for development pizzas when they are moved to the production table.
  - The primary key is the id column.
  - The detail column contains a varchar containing one namelet for a specific meat.
  - Meats can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the meats table)
  - meat_id is a foreign key that references the Meat table's primary key, id
  - The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: "Before" and "After".
    - If a namelet has the value "before" then it can only be used as the first word in the new name by the procedure that generates names
    - If a namelet has the value "after" then it can only be used as the second word in the new name by the procedure that generates names
  - SELECT * FROM Namelet_Meat LIMIT 4;
  - 

id	detail	meat_id	location
1	Ikea	1	Before
2	Abba	1	Before
3	Naked	1	Before
4	Ooh La La	2	Before

Figure 25: Namelet_Meat Table

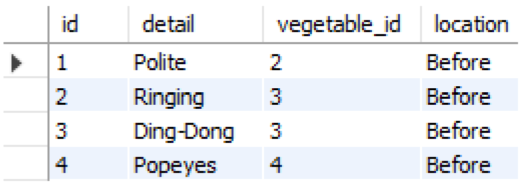
- **Namelet_Cheese**
  - This acts as a store of namelets for cheese ingredients, which are used to generate name recommendations for development pizzas when they are moved to the production table.
  - The primary key is the id column.
  - The detail column contains a varchar containing one namelet for a specific cheese.

- Cheeses can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the Cheese table)
- cheese_id is a foreign key that references the Cheese table's primary key, id
- The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: "Before" and "After".
  - If a namelet has the value "before" then it can only be used as the first word in the new name by the procedure that generates names
  - If a namelet has the value "after" then it can only be used as the second word in the new name by the procedure that generates names
- SELECT * FROM Namelet_Cheese LIMIT 4;
- 

id	detail	cheese_id	location
1	Greek	1	Before
2	Zorbas	1	Before
3	Athenian	1	Before
4	Smelly	2	Before

Figure 26: Namelet_Cheese Table

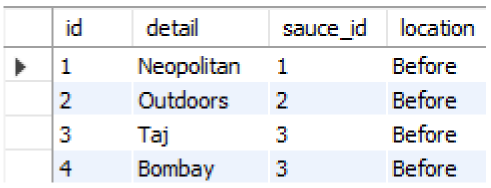
- **Namelet_Vegetable**
  - This acts as a store of namelets for vegetable ingredients, which are used to generate name recommendations for development pizzas when they are moved to the production table.
  - The primary key is the id column.
  - The detail column contains a varchar containing one namelet for a specific meat.
  - Vegetables can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the Vegetable table)
  - vegetable_id is a foreign key that references the Vegetable table's primary key, id
  - The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: "Before" and "After".

        - If a namelet has the value “before” then it can only be used as the first word in the new name by the procedure that generates names
        - If a namelet has the value “after” then it can only be used as the second word in the new name by the procedure that generates names
    - SELECT * FROM Namelet_Vegetable LIMIT 4;
    - 

id	detail	vegetable_id	location
1	Polite	2	Before
2	Ringing	3	Before
3	Ding-Dong	3	Before
4	Popeyes	4	Before

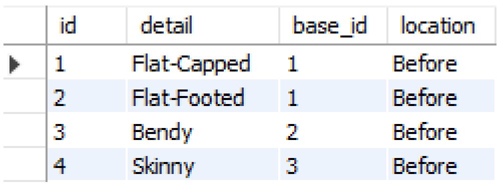
Figure 27: Namelet_Vegetable Table

- **Namelet_Sauce**
    - This acts as a store of namelets for sauce ingredients, which are used to generate name recommendations for development pizzas when they are moved to the production table.
    - The primary key is the id column.
    - The detail column contains a varchar containing one namelet for a specific sauce.
    - Sauces can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the meats table)
    - sauce_id is a foreign key that references the meats table’s primary key, id
    - The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: “Before” and “After”.
        - If a namelet has the value “before” then it can only be used as the first word in the new name by the procedure that generates names
        - If a namelet has the value “after” then it can only be used as the second word in the new name by the procedure that generates names
    - SELECT * FROM Namelet_Sauce LIMIT 4;

- 

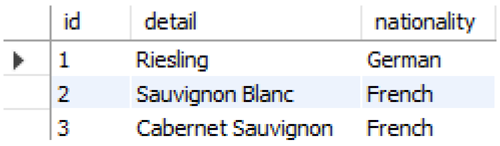
id	detail	sauce_id	location
1	Neopolitan	1	Before
2	Outdoors	2	Before
3	Taj	3	Before
4	Bombay	3	Before

Figure 28: Namelet_Sauce Table

- **Namelet_Base**
  - This acts as a store of namelets for pizza base ingredients, which are used to generate name recommendations for development pizzas when they are moved to the production table.
  - The primary key is the id column.
  - The detail column contains a varchar containing one namelet for a specific base.
  - Bases can have multiple namelets, hence why this ancillary table was needed (as opposed to having a namelet column in the meats table)
  - base_id is a foreign key that references the base table's primary key, id
  - The location field specifies the location of the namelet of the naming process. This field is of type varchar(30) and can contain two values: "Before" and "After".
    - If a namelet has the value "before" then it can only be used as the first word in the new name by the procedure that generates names
    - If a namelet has the value "after" then it can only be used as the second word in the new name by the procedure that generates names
  - SELECT * FROM Namelet_Base LIMIT 4;
  - 

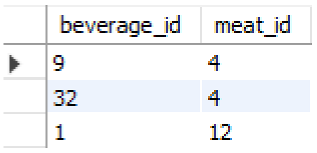
id	detail	base_id	location
1	Flat-Capped	1	Before
2	Flat-Footed	1	Before
3	Bendy	2	Before
4	Skinny	3	Before

Figure 29: Namelet_Base Table

- **Beverage:** This contains information on each cocktail sold by Wonka Labs.
  - id is the primary key. It is a value unique to this table that identifies each beverage by an integer.
  - The detail field contains the name of each beverage as a varchar(30). E.g. ‘Riesling’
  - The nationality field is of type varchar(30) and stores data on the nationality of the wine, for example “German”
  - Example of table: SELECT * FROM Beverage LIMIT 3;
  - 

id	detail	nationality
1	Riesling	German
2	Sauvignon Blanc	French
3	Cabernet Sauvignon	French

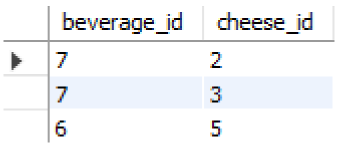
Figure 30: Beverage Table

- **Beverage_Pairing_Meat:**
  - This acts as a bridge table between beverages and meats.
  - The relationship between beverages and meats is many-to-many as many meats can suit a certain beverage and many beverages can suit a certain meat.
    - Hence a bridging table is needed to represent this relationship
  - It combines beverage_id and meat_id into a composite(multi-column) key.
  - Example of table: SELECT * FROM Beverage_Pairing_Meat LIMIT 3;
  - 

beverage_id	meat_id
9	4
32	4
1	12

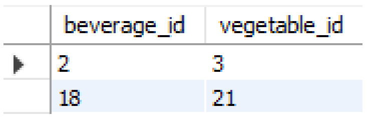
Figure 31: Beverage_Pairing_Meat Table

- **Beverage_Pairing_Cheese:**
  - This acts as a bridge table between beverages and cheeses.
  - The relationship between beverages and cheese is many-to-many as many cheeses can suit a certain beverage and many beverages can suit a certain cheese.
    - Hence a bridging table is needed to represent this relationship
  - It combines beverage_id and cheese_id into a composite(multi-column) key.

- Example of table: SELECT * FROM Beverage_Pairing_Cheese LIMIT 3;
- 

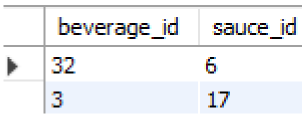
beverage_id	cheese_id
7	2
7	3
6	5

  Figure 32: Beverage_Pairing_Cheese Table

- **Beverage_Pairing_Vegetable:**
  - This acts as a bridge table between beverages and vegetables.
  - The relationship between beverages and vegetable is many-to-many as many vegetables can suit a certain beverage and many beverages can suit a certain vegetable.
    - Hence a bridging table is needed to represent this relationship
  - It combines beverage_id and vegetable_id into a composite(multi-column) key.
  - Example of table: SELECT * FROM Beverage_Pairing_Vegetable LIMIT 2;
  - 

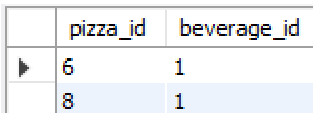
beverage_id	vegetable_id
2	3
18	21

  Figure 33: Beverage_Pairing_Vegetable Table

- **Beverage_Pairing_Sauce:**
  - This acts as a bridge table between beverages and sauce.
  - The relationship between beverages and sauces is many-to-many as many sauce can suit a certain beverage and many beverages can suit a certain sauce.
    - Hence a bridging table is needed to represent this relationship
  - It combines beverage_id and sauce_id into a composite(multi-column) key.
  - Example of table: SELECT * FROM Beverage_Pairing_Sauce LIMIT 2;
  - 

beverage_id	sauce_id
32	6
3	17

  Figure 34: Beverage_Pairing_Sauce Table

- **Beverage_Recommendation:**
    - This acts as a bridge table between beverages and sauce.
    - The relationship between beverages and sauces is many-to-many as many sauce can suit a certain beverage and many beverages can suit a certain sauce.
        - Hence a bridging table is needed to represent this relationship
    - It combines beverage_id and sauce_id into a composite(multi-column) key.
    - Example of table: SELECT * FROM Beverage_Recommendation LIMIT 2;
    - 

	pizza_id	beverage_id
▶	6	1
	8	1

Figure 35: Beverage_Recommendation Table

Normalization

1NF: "A relation is in first normal form if and only if the domain of each attribute contains only atomic (indivisible) values, and the value of each attribute contains only a single value from that domain." (Codd, Further normalization of the database relational model. Data Base Systems, 1972).

If each table has a primary-key then the database is in 1NF

- Since all tables in the database have a primary key, the database can be said to be in 1NF
- Repeating groups were removed from original dataset.
    - A repeating group is any attribute that can have more than one value for a single primary key attribute

2NF: The database should be in first normal form and " does not have any non-prime attribute that is functionally dependent on any proper subset of any candidate key of the relation." (Codd, Further normalization of the database relational model. Data Base Systems, 1972).

If Non-key attributes are dependent on the whole key then the table is in 2NF

- There is no partial dependencies
- This is relevant only to tables with composite keys
- An example of a table with a composite key is garnish_allocations

    - Garnish_allocations is in 2NF because the serving depends on both the cocktail_id and the garnish_id(there are no partial dependencies)
    - i.e. you couldn't deduce from one of these individually what the serving is.
    - For similar reasons, liquid_allocations is also in 2NF
- Meat_Allocations, Cheese_Allocations, Sauce_Allocations and Vegetable_Allocations only contain a composite key and hence cannot have any partial dependencies and are therefore in 2NF.
- Beverage_Pairing_Meat, Beverage_Pairing_Cheese, Beverage_Pairing_Sauce and Beverage_Pairing_Vegetables also only contain a composite key and hence cannot have any partial dependencies and are therefore in 2NF.

3NF: The relation (R) is in 2NF and every non-prime attribute of R is non-transitively dependent on every key of R.
If Non-Key attributes are dependent on nothing but the whole key.

- 3NF means removing any attributes not directly dependent on the key to separate relations.
- We see throughout the database that this holds true.
    - The meat table only have columns that relate to meats and not, for example cheeses and vice versa.
        - The same can be said for the meats, cheese, sauce, vegetables and base tables
    - The development_pizza and similarly the production_pizza tables contain only information on pizzas and not on things like cheese, sauce, vegetable or beverage pairing, meaning the table is in 3NF.
        - These tables do have foreign keys from the base table but this is needed in order to allow for joins and does not result in data redundancy.
    - The cocktails table has only information on the cocktail itself.
        - It does not hold information on liquids or garnishes, meaning the table is in 3NF.
        - These tables do have foreign keys from the ice and glass tables but this is needed in order to allow for joins and does not result in data redundancy, hence 3NF is not violated

BCNF: A relational schema R is in Boyce-Codd normal form if and only if for every one of its dependencies X → Y, at least one of the following conditions hold:

X → Y is a trivial functional dependency (Y ⊆ X),

X is a superkey for schema R.

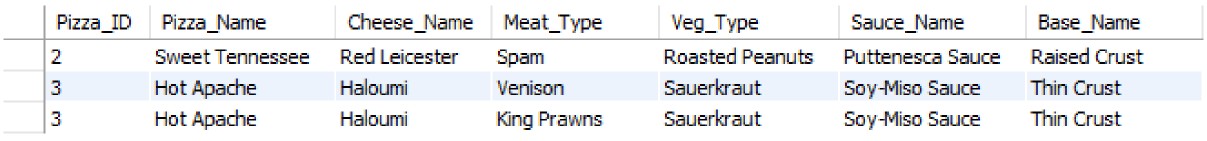
(Codd, Recent Investigations into Relational Data Base, 1974)

A relation is in Boyce Codd Normal form if, and inly if, every determinant is a candidate key.

- Since the database has been established to be in 3NF and does not have any overlapping candidate keys, it is guaranteed to be in BCNF. (Codd, Recent Investigations into Relational Data Base, 1974)

Section 4: Database Views

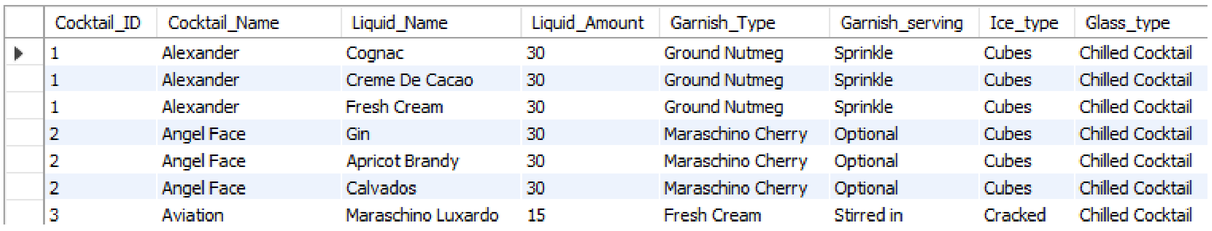
View 1: production_pizzas_specifications

- This is a view of all the pizzas in the production_pizzas database, along with all the ingredients in each pizza
- This is a useful view for the Oompa Loompa Chefs if they want to quickly access the entire specification of any pizza that is on sale.
- This view could not be used as a table as it would violate the normalization of the database.
    - It would not be 1NF as there is no primary key or potential primary key
- Views update automatically when a change is made to the table they are based on
    - The underlying production_pizza table will be likely to be changed often when development pizzas are added
    - Hence the automatic updating feature of views is useful to ensure all production pizzas and their specifications are on display.
- SELECT * FROM production_pizza_specifications;
- 

Pizza_ID	Pizza_Name	Cheese_Name	Meat_Type	Veg_Type	Sauce_Name	Base_Name
2	Sweet Tennessee	Red Leicester	Spam	Roasted Peanuts	Puttenesca Sauce	Raised Crust
3	Hot Apache	Haloumi	Venison	Sauerkraut	Soy-Miso Sauce	Thin Crust
3	Hot Apache	Haloumi	King Prawns	Sauerkraut	Soy-Miso Sauce	Thin Crust

Figure 36: Production_Pizza_Specifications View

View 2: cocktail_specifications

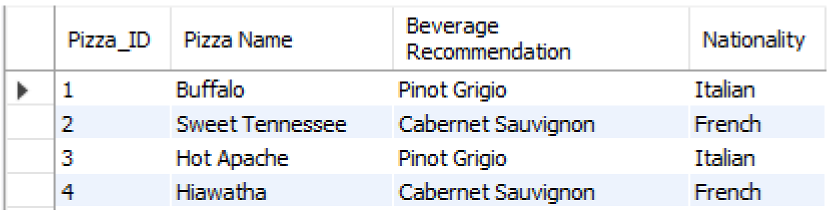
- This is a view of all the cocktails in the cocktail database, along with all the ingredients in each cocktail
- This is a useful view for the Oompa Loompa Chefs if they want to quickly access the entire specification of any cocktail that is on sale.
- This view could not be used as a table as it would violate the normalization of the database.
    - It would not be 1NF as there is no primary key or potential primary key
- Views update automatically when a change is made to the table they are based on
    - The underlying cocktail table will be likely to be changed often when new cocktails are added
    - Hence the automatic updating feature of views is useful to ensure all cocktails and their specifications are on display.
- Example: SELECT * FROM cocktail_specifications;
- 

Cocktail_ID	Cocktail_Name	Liquid_Name	Liquid_Amount	Garnish_Type	Garnish_serving	Ice_type	Glass_type
1	Alexander	Cognac	30	Ground Nutmeg	Sprinkle	Cubes	Chilled Cocktail
1	Alexander	Creme De Cacao	30	Ground Nutmeg	Sprinkle	Cubes	Chilled Cocktail
1	Alexander	Fresh Cream	30	Ground Nutmeg	Sprinkle	Cubes	Chilled Cocktail
2	Angel Face	Gin	30	Maraschino Cherry	Optional	Cubes	Chilled Cocktail
2	Angel Face	Apricot Brandy	30	Maraschino Cherry	Optional	Cubes	Chilled Cocktail
2	Angel Face	Calvados	30	Maraschino Cherry	Optional	Cubes	Chilled Cocktail
3	Aviation	Maraschino Luxardo	15	Fresh Cream	Stirred in	Cracked	Chilled Cocktail

Figure 37: Cocktail_Specifications View

View 3: beverage_recommendations

- This is a view of all the beverage recommendations for each baked good in the production database
- This is a useful view for the Oompa Loompa Waitors if they want to quickly recommend a beverage, such as the 'Riesling' wine for a customer, based on the expert knowledge of food and wine pairings possessed by the Oompa Loompas.
- The high-level application would be able to quickly select this view without having to do any other code or querying.
- This view could not be used as a table as it would violate the normalization of the database.

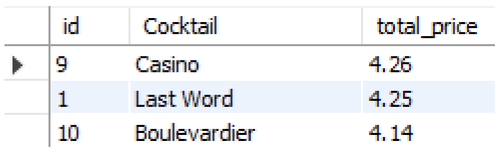
    - It would not be 1NF as there would be redundant data created in the database if this was its own table
- Views update automatically when a change is made to the table they are based on
    - The underlying pizza table will be likely to be changed often when new pizzas are added
    - Hence the automatic updating feature of views is useful to ensure all pizzas and their recommendations are on display.
- Example: SELECT * FROM beverage_recommendations;
- 

Pizza_ID	Pizza Name	Beverage Recommendation	Nationality
1	Buffalo	Pinot Grigio	Italian
2	Sweet Tennessee	Cabernet Sauvignon	French
3	Hot Apache	Pinot Grigio	Italian
4	Hiawatha	Cabernet Sauvignon	French

Figure 38: Beverage_Recommendations View

View 4: Cocktail Cost

- This is a view of the cost of making each cocktail to Wonka Labs
- This is a useful view for the Oompa Loompa Actuaries to find out what the cost price of each cocktail is for their profitability analysis.
- The high-level application would be able to quickly select this view without having to do any other code or querying.
- This view could not be used as a table as it would violate the normalization of the database.
    - It would not be 1NF as there would be redundant data created in the database if this was its own table
- Views update automatically when a change is made to the table they are based on
    - The underlying coctail table will be likely to be changed often when new cocktails are added
    - Hence the automatic updating feature of views is useful to ensure all cocktails and their prices are on display.
- Example: SELECT * FROM CocktailCost;

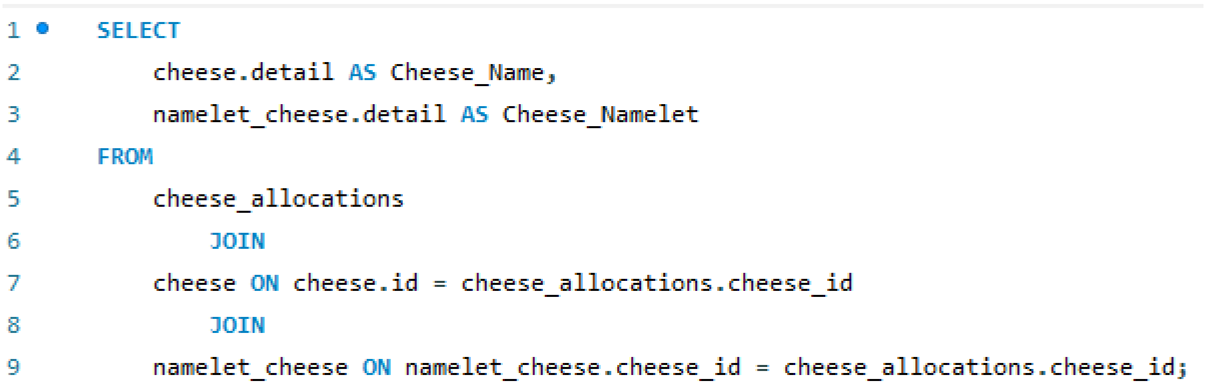
- 

id	Cocktail	total_price
9	Casino	4.26
1	Last Word	4.25
10	Boulevardier	4.14

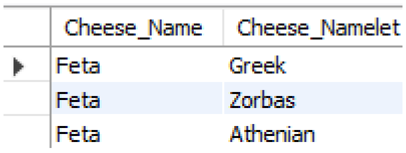
Figure 39: Cocktail Cost View

Section 5. Procedural Elements

Procedure 1: Promote_Pizza(<pizza_id>)

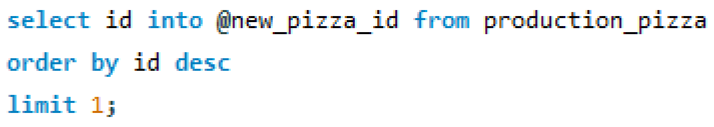
- This procedure takes as an input the pizza_id of a development_pizza that is to be moved to production pizzas.
- First, an empty temporary table is created.
- Where the 'location' column has a value of 'before', the following steps apply:
- For each element that makes up a pizza(cheese, meat, vegetables, sauce and base) we append all the namelets of its ingredients to the column 'namelet' in the temporary table.
    - For example, for a pizza with 'Feta' cheese, its 'before' namelets as seen below are 'Greek', 'Zorbas' and 'Athenian'.
    - 

```sql
SELECT
    cheese.detail AS Cheese_Name,
    namelet_cheese.detail AS Cheese_Namelet
FROM
    cheese_allocations
        JOIN
    cheese ON cheese.id = cheese_allocations.cheese_id
        JOIN
    namelet_cheese ON namelet_cheese.cheese_id = cheese_allocations.cheese_id;
```

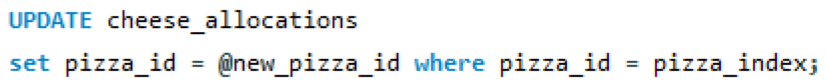
    - 

Cheese_Name	Cheese_Namelet
Feta	Greek
Feta	Zorbas
Feta	Athenian

    - All three of these namelets are added to the temporary table.
    - If the pizza has more than one cheese, then their namelets are also inserted.

- Once all the before namelets are in this temporary table, one is chosen at random using the order by rand() query and inserted into the 'pizza_before' variable defined at the start of the procedure.
- This process is repeated, but with namelets with the value 'after' in the 'location' column of the namelets table
- We then insert into the production_pizza table the following:
    - In the detail column, we insert CONCAT(@pizza_before,' ',@pizza_after)
        - This inserts the two randomly chosen namelets into the detail column with a space separating them.
        - We keep the base_id from the development_pizza table and add this to the 'base_id' column
        - Since the production_pizza table has an auto_increment primary key, we do not need to insert anything into it.
- 

```
select id into @new_pizza_id from production_pizza
order by id desc
limit 1;
```

    - This code is used to assign to the variable 'new_pizza_id' the id of the new pizza in the production_pizza table
- 

```
UPDATE cheese_allocations
set pizza_id = @new_pizza_id where pizza_id = pizza_index;
```

- This updates the cheese_allocations table by changing the old development_pizza id to the pizzas new id.
    - For example:
    - If the pizza being moved from the development pizza table had an id of 1001
    - When it is moved to the production_pizza table, it is assigned a new id – so if there were 26 pizzas in the table beforehand, the new pizza would have a new id of 28.
        - This is required as pizzas in the production_pizza table cannot have a id greater than 1000.
        - This feature is in place to avoid the auto_increment of both tables accidently assigning the same pizza_id to two pizzas.

- This update, along with similar updates to the meat_allocations, sauce_allocations, vegetable_allocations and beverage_recommendation tables enables us to keep the pizza specification of the promoted development pizza
    - Otherwise, there would be no entries in the database for this pizza – as this pizza_id would not have existed when making the database
- This procedure allows for a simple and easy system of moving development pizzas to production_pizzas.
    - All the user has to know is the id of the development pizza they wish to insert into the production_pizza table and the rest will be handled for them
- If the id entered by the user is not in the development_pizza table, then the database remains unchanged.

Procedure 2: New_Cocktail(*parameters)

Input parameters:

liquid1_id,

liquid2_id,

liquid1_size_ml,

liquid2_size_ml,

garnish1_id,

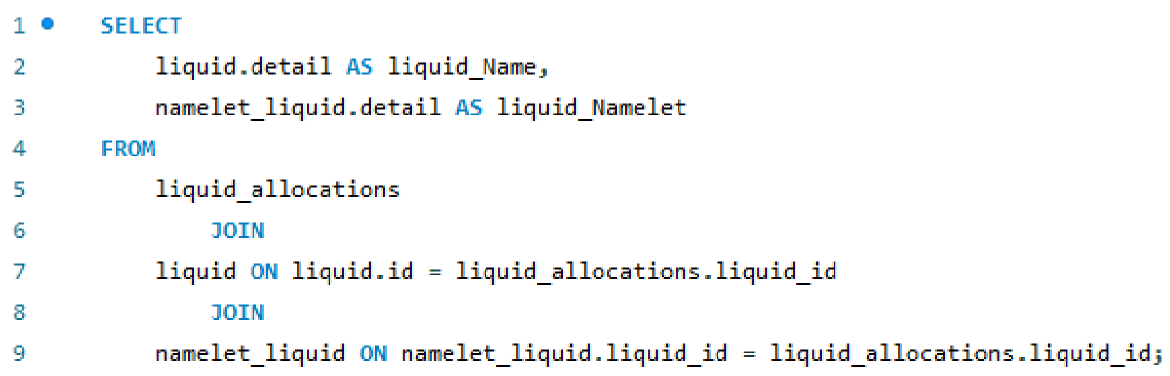
garnish1_serving,

new_ice_id,

new_glass_id

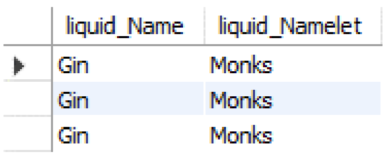
- This procedure has two main elements:
    - Enable the user to insert a new cocktail and all its ingredients into the database
    - Give the new cocktail a name based on the namelets of its ingredients
- The database has a cocktail split up among multiple tables – the cocktail table for its name, the liquid table for its alcoholic and non-alcoholic ingredients, the garnish table for its garnishes

    - This means that if higher level application wants to insert a new cocktail into the database, it would have to know which tables to navigate to. This would be quite cumbersome and possibly prone to human error as there are a lot of different tables in the database.
- All the user has to know is the id of the items they wish to add to the cocktail and how much of each.
- It allows up to two different types of liquid, one type of garnish and one type of both ice and glass to make up a cocktail
- First, a new cocktail is inserted into the cocktail table.
    - It is initially given the default name ‘new_cocktail’ which is changed later on in the procedure once all the ingredients are added and a name can be generated
- The id of the new cocktail is saved into the variable ‘new_cocktail_id’
    - This is used to update the other tables
- The new values for the ingredients of cocktails are inserted into their respective tables
- A temporary table is then created
- Where the ‘location’ column has a value of ‘before’, the following steps apply:
- For the elements that makes up a cocktail - liquid, garnish and ice(glass is omitted as cocktails tend not to be named based on what type of glass they are served in) we append all the namelets of its ingredients to the column ‘namelet_before’ in the temporary table.
    - For example, for a cocktail with ‘Feta’ cheese, its ‘before’ namelets as seen below are ‘Greek’, ‘Zorbas’ and ‘Athenian’.



```
SELECT
    liquid.detail AS liquid_Name,
    namelet_liquid.detail AS liquid_Namelet
FROM
    liquid_allocations
        JOIN
    liquid ON liquid.id = liquid_allocations.liquid_id
        JOIN
    namelet_liquid ON namelet_liquid.liquid_id = liquid_allocations.liquid_id;
```

    -

- 

liquid_Name	liquid_Namelet
Gin	Monks
Gin	Monks
Gin	Monks

  - All of these namelets are added to the temporary table.
  - If the pizza has more than one liquid, then their namelets are also inserted.
- Once all the before namelets are in this temporary table, one is chosen at random using the order by rand() query and inserted into the 'cocktail_before' variable defined at the start of the procedure.
- This process is repeated, but with namelets with the value 'after' in the 'location' column of the namelets table
- We then update the cocktail table the following:
- `update cocktail set detail = CONCAT(@cocktail_before,' ',@cocktail_after) where id = @new_cocktail_id;`
  - We set the detail column where the id = new_cocktail_id to CONCAT(@cocktail_before,' ',@cocktail_after)
    - This inserts the two randomly chosen namelets into the detail column with a space separating them.

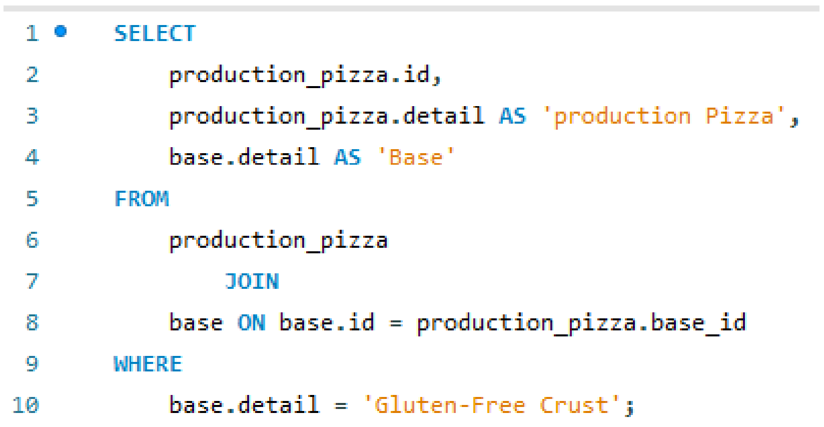
Procedure(Function) 3: Recommender(pizza_id) -> beverage_id

- This function takes a pizza id as an input and returns a recommended beverage id from the beverages table.
- A temporary table is created
  - For each of cheese, meat, vegetable and sauce:
    - The beverage ids that suit each specific item is appended to the temp_table column 'beverage_id'
    - E.g. if cheese_id suits beverage_id 7 and 8, then 7 and 8 are added as rows to the temp table
    - This is repeated for each cheese on the pizza,
      - And then repeated for each topping/sauce
- We then use the count( ) aggregate function along with a group by , limit 1 and order by desc to find the beverage_id that appears most frequently in the temp_table column beverage_id
- This value is inserted into 'rec_beverage_id' and is the output of the function.

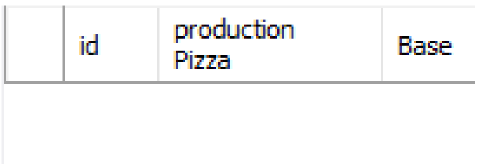
Section 6: Example Queries: The Database in Action

Pizzas

- Proving for the Coeliac Community of this world is something very important to us here at Wonka Labs, with myself (Wonka's grandson in case you forgot) being a member.
- Nothing beats a good pizza after a long day of writing SQL queries, so I decided to check the production pizza database to see what was on offer.

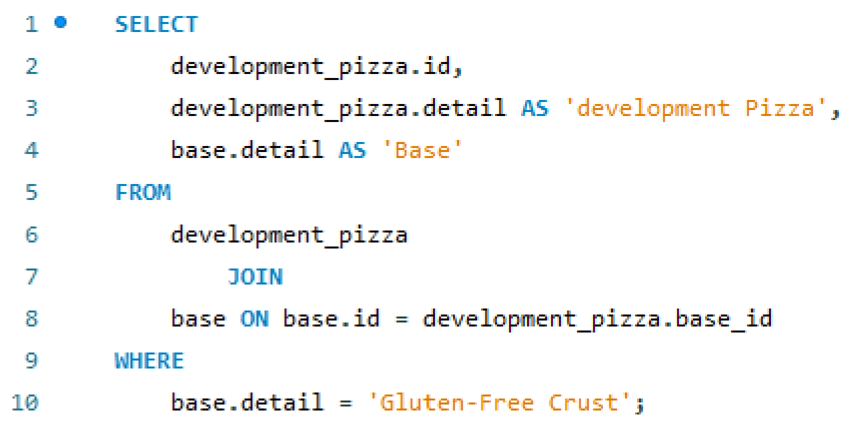


```sql
SELECT
    production_pizza.id,
    production_pizza.detail AS 'production Pizza',
    base.detail AS 'Base'
FROM
    production_pizza
        JOIN
    base ON base.id = production_pizza.base_id
WHERE
    base.detail = 'Gluten-Free Crust';
```

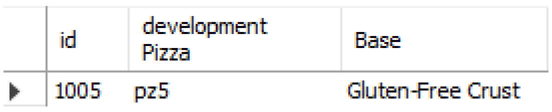


	id	production Pizza	Base

- To my dismay, there was not one gluten free pizza available to purchase from the production pizzas.
- I went in search to the development pizza database. Surely those whacky Oompa Loompas have one that could be put into production.

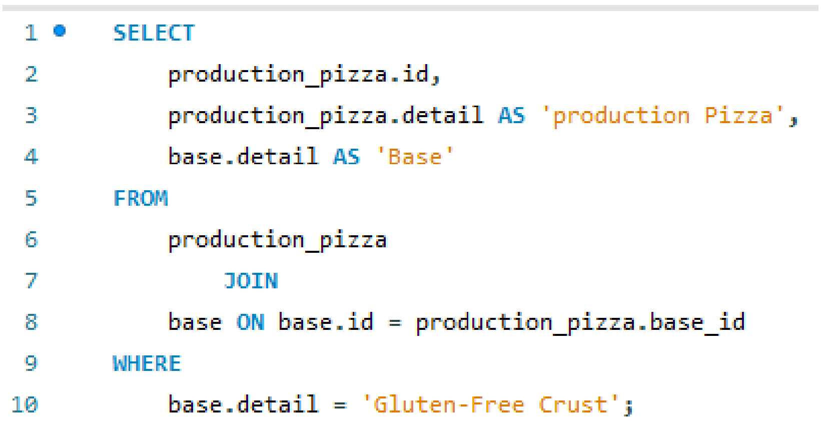
- 

```sql
SELECT
    development_pizza.id,
    development_pizza.detail AS 'development Pizza',
    base.detail AS 'Base'
FROM
    development_pizza
        JOIN
    base ON base.id = development_pizza.base_id
WHERE
    base.detail = 'Gluten-Free Crust';
```

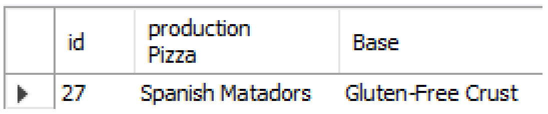
- 

id	development Pizza	Base
1005	pz5	Gluten-Free Crust

- I was delighted to see one pizza with a gluten free crust in the development pizza database.
- I made the executive decision there and then to add this pizza to production.
- Thankfully, the database has the very useful Promote_Pizza( ) procedure which meant all I had to do was call it with the development pizza's id to add the pizza into production. It will also create a new name for this pizza based on its ingredients.
-
```sql
CALL Promote_Pizza(1005);
```
- We run the above commands again, first checking the production pizza table to see if the new pizza has been added.

- 

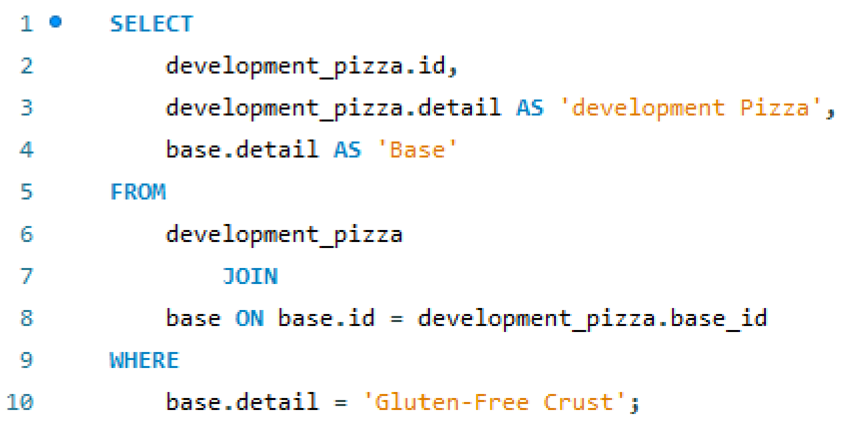
```sql
SELECT
    production_pizza.id,
    production_pizza.detail AS 'production Pizza',
    base.detail AS 'Base'
FROM
    production_pizza
        JOIN
    base ON base.id = production_pizza.base_id
WHERE
    base.detail = 'Gluten-Free Crust';
```

- 

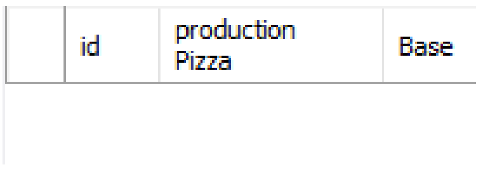
id	production Pizza	Base
27	Spanish Matadors	Gluten-Free Crust

- Delightful! The new pizza has been added with a whimsical new name that should delight those with a distain for gluten.

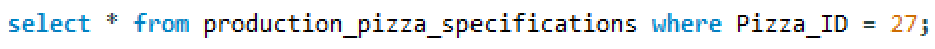
- We check the development pizza database to make sure the pizza no longer exists in that database, as that would lead to a data redundancy issue.



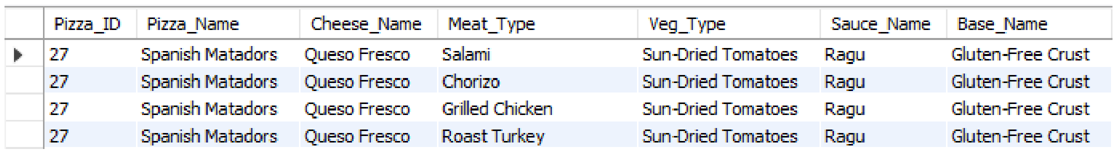
```sql
SELECT
    development_pizza.id,
    development_pizza.detail AS 'development Pizza',
    base.detail AS 'Base'
FROM
    development_pizza
        JOIN
    base ON base.id = development_pizza.base_id
WHERE
    base.detail = 'Gluten-Free Crust';
```

- 

	id	production Pizza	Base

-
- We see the pizza is no longer in development.
- I was curious to see what the toppings on this pizza were, while waiting for it to arrive.
- The production pizza specification view provides a way of checking all the toppings on any pizza. I Checked the make-up of my inbound pizza using the following query:
- 

```sql
select * from production_pizza_specifications where Pizza_ID = 27;
```

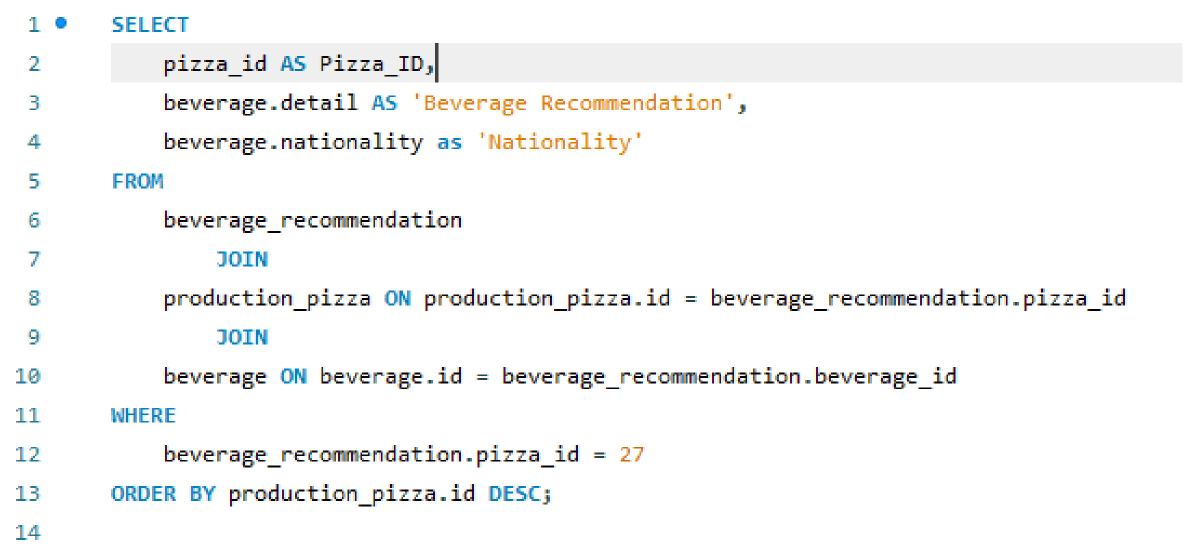
- 

Pizza_ID	Pizza_Name	Cheese_Name	Meat_Type	Veg_Type	Sauce_Name	Base_Name
27	Spanish Matadors	Queso Fresco	Salami	Sun-Dried Tomatoes	Ragu	Gluten-Free Crust
27	Spanish Matadors	Queso Fresco	Chorizo	Sun-Dried Tomatoes	Ragu	Gluten-Free Crust
27	Spanish Matadors	Queso Fresco	Grilled Chicken	Sun-Dried Tomatoes	Ragu	Gluten-Free Crust
27	Spanish Matadors	Queso Fresco	Roast Turkey	Sun-Dried Tomatoes	Ragu	Gluten-Free Crust

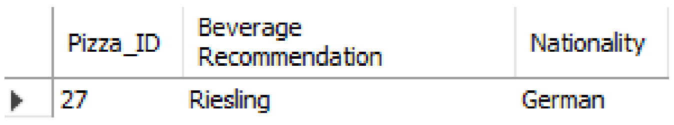
- I was delighted to see a pizza full of meats and sun-dried tomatoes was what was in store for me.

Beverages & Cocktails

- The database contains beverage recommendations for each pizza.
- The query below suggests a wine for my pizza



```sql
SELECT
    pizza_id AS Pizza_ID,
    beverage.detail AS 'Beverage Recommendation',
    beverage.nationality as 'Nationality'
FROM
    beverage_recommendation
        JOIN
    production_pizza ON production_pizza.id = beverage_recommendation.pizza_id
        JOIN
    beverage ON beverage.id = beverage_recommendation.beverage_id
WHERE
    beverage_recommendation.pizza_id = 27
ORDER BY production_pizza.id DESC;
```

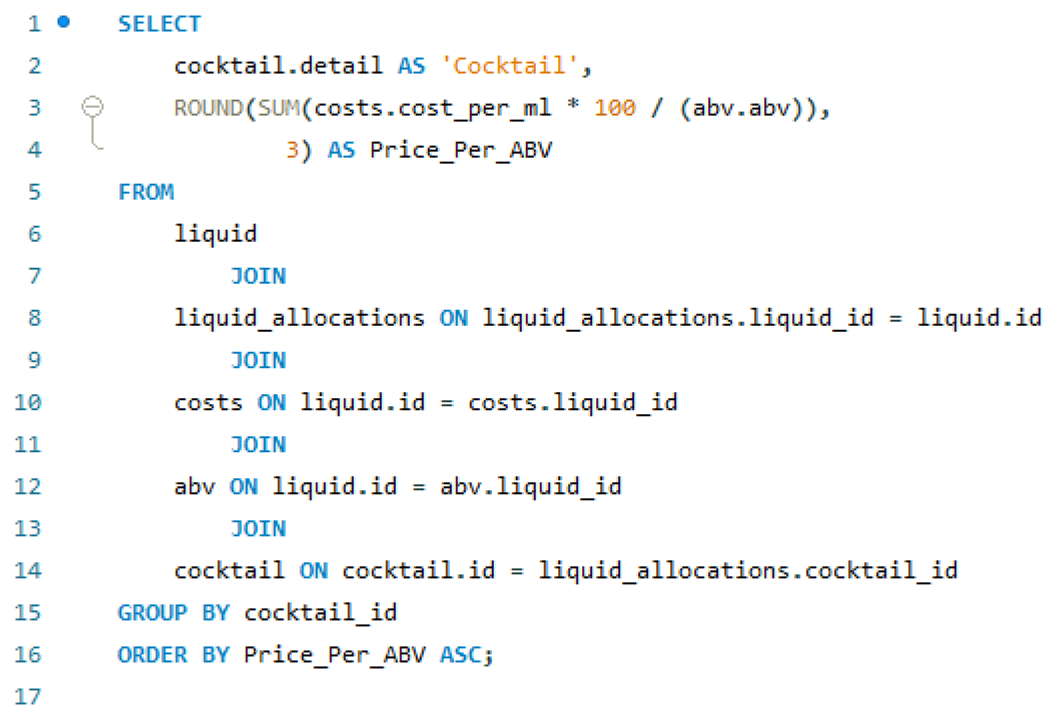


Pizza_ID	Beverage Recommendation	Nationality
27	Riesling	German

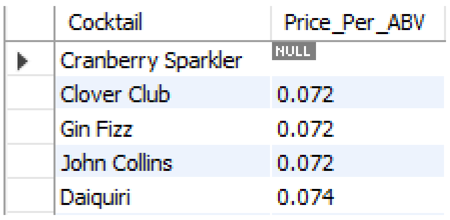
We see that the wine ‘Riesling’ has been recommended to me based on my choice of pizza.

But what if we want a cocktail instead?

Being a poor student, I want to find the best value cocktail on offer. This means finding the cocktail that has the most alcohol per amount paid.



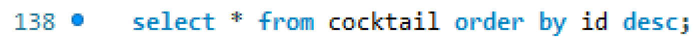
```sql
SELECT
    cocktail.detail AS 'Cocktail',
    ROUND(SUM(costs.cost_per_ml * 100 / (abv.abv)),
            3) AS Price_Per_ABV
FROM
    liquid
        JOIN
    liquid_allocations ON liquid_allocations.liquid_id = liquid.id
        JOIN
    costs ON liquid.id = costs.liquid_id
        JOIN
    abv ON liquid.id = abv.liquid_id
        JOIN
    cocktail ON cocktail.id = liquid_allocations.cocktail_id
GROUP BY cocktail_id
ORDER BY Price_Per_ABV ASC;
```



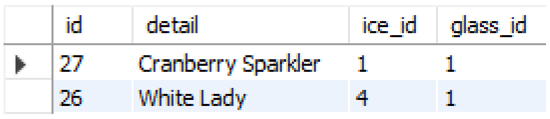
Cocktail	Price_Per_ABV
Cranberry Sparkler	NULL
Clover Club	0.072
Gin Fizz	0.072
John Collins	0.072
Daiquiri	0.074

We see that 'Clover Club', 'Gin Fizz' and 'John Collins' all represent the best bang for your buck.

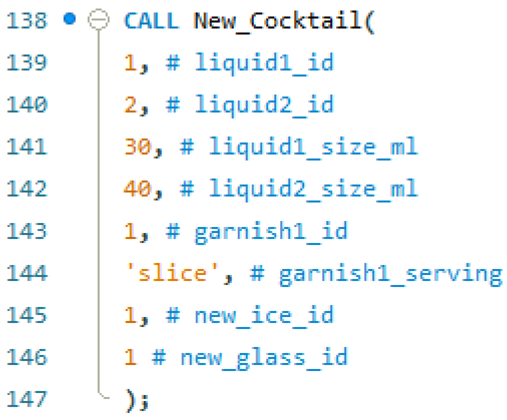
If I wanted to include a new cocktail in the database, I can do so using the New_Cocktail procedure.



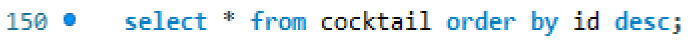
```sql
select * from cocktail order by id desc;
```



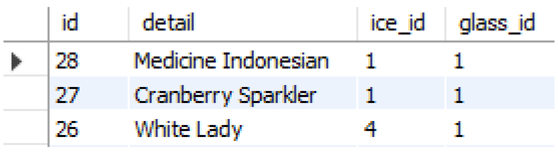
id	detail	ice_id	glass_id
27	Cranberry Sparkler	1	1
26	White Lady	4	1



```sql
CALL New_Cocktail(
1, # liquid1_id
2, # liquid2_id
30, # liquid1_size_ml
40, # liquid2_size_ml
1, # garnish1_id
'slice', # garnish1_serving
1, # new_ice_id
1 # new_glass_id
);
```



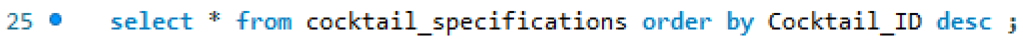
```sql
select * from cocktail order by id desc;
```



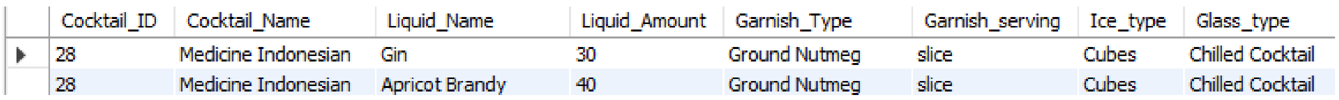
id	detail	ice_id	glass_id
28	Medicine Indonesian	1	1
27	Cranberry Sparkler	1	1
26	White Lady	4	1

We see the new cocktail is inserted into the Cocktail Database.

If we want to see the ingredients, garnishes, ice and glass elements of this new cocktail, we can use the view cocktail_specifications:



```sql
select * from cocktail_specifications order by Cocktail_ID desc ;
```



Cocktail_ID	Cocktail_Name	Liquid_Name	Liquid_Amount	Garnish_Type	Garnish_serving	Ice_type	Glass_type
28	Medicine Indonesian	Gin	30	Ground Nutmeg	slice	Cubes	Chilled Cocktail
28	Medicine Indonesian	Apricot Brandy	40	Ground Nutmeg	slice	Cubes	Chilled Cocktail

Section 7: Conclusions

The database provides an excellent foundation for Wonka Labs food development system. The normalization steps followed in creating this database make it quick, efficient, easy to use and a reliable backbone to the company's ambitious new plans to expand into new areas of the food and beverage system.

The normalization standards set out by Edgar Codd in the seventies may sound outdated, but they are the perfect solution to the problems Wonka labs has faced in the past in regard to data storage issues. No longer will our secrets be stolen by enemy spies, no longer will recipes be dropped into the chocolate river and lost forever and no longer will The Oompa Loompas lose a night's sleep over using a normal base instead of a non-gluten-free base.

This database is not perfect, however. Further development could include recording data on customers and their purchases. This data could then be used to further improve our recommendation systems and help us satisfy our beloved customer even more, with the added bonus of taking even more market share from Mr.Slugworth. More procedures and triggers could be added to make it even more efficient – for example making a trigger that prevents wrong values being entered into the development pizzas id field.

Overall, I am confident it is an excellent solution and I know you will agree that this database will provide the perfect backbone to the company's expansion plans for years to come.

Acknowledgements

I acknowledge that the work is entirely my own, and that every sentence in this report was written by me and me alone.

References

Bibliography

Codd, E. (1972). Further normalization of the database relational model. Data Base Systems.

Codd, E. (1974). Recent Investigations into Relational Data Base. *Proc*.