Creating a Database Schema using PostgreSQL Database

Purpose

The following tutorial demonstrates how to script SQL statements to create the actual phsyical dynamic working database. The tutorial demostrates how to both physically and methodically:

- Develop the diagram for the relational schema
- Create a logical schema from the relational diagram
- Create Tables
- Craft Constraints
- Use REGEX with Constraints
- Form the foreign keys
- Develop trigger statements used to update/insert data into table after specific events
- Create View tables used by a web application that use complex SQL queries

Background

The idea behind this project has been adopted from my capstone project for my master program in data science where I had to develop a fully functional database using PostgreSQL via PGAdmin4 GUI interface. The database was *wired* into a Flask SQL-Alchemy web application that dynamically responded to the PostgreSQL database via a Heroku web server.

The result was a fully functional website that mimic a site used by medical reference laboratories that created the ability for customers to place orders, view bills, and review laboratory results. Furthermore, the application allowed employees to view orders, print barcodes, review quality control results via visuals/tables, and other job specific functions. The web application was dynamic meaning that when a customer placed an order, the subsequent billing was updated as well as new results adding to cummulative statistics, and new pending list being created.

The highlight of this project is not to review this aforementioned web application, but to specifically highlight the development of the database schema used to create the backbone for all the functional data requirements for the web application. A database schema is a collection of all the view tables, triggers, constraints, and relevant developmental schema such as the logical schema that is diagrammed to create the starting point of the database schema using concepts of cardinality and foreign key constraints.

Below, I will describe the development of each componenet of the database schema alongside excerpted scripts from the capstone project, and hopefully give examples that will provide sufficient insight if one were to create their own database schema. Note, I have only provided small excerpts

from the original SQL scripts used to create the actual database just to provide a more aesthetic and organize feel. If you would like to view all the files please do so at the GitHub repository: https://github.com/RyansStacks/Mock_Laboratory

SQL Scripts with Annotations

Functional Requriements

The functional requirements are the chief information needed to design the database to reflect the necessary functions and views for a company. Listing requirements may range from plethora of pages of necessary interplay of IT requirements, business logic, or simply just cardinality and inter-relationships between various important entities used in the database. Here, the latter approach is taken where listed below is just an example of some requirements needed to form two tables Customers and Orders in the database entity relationship diagram. This information is often obtain using great communicational outreach to site managers or researching the enterprise to elicit what entities relate to each other.

Customers

Only one customer may have one address, email, and phone number listed.

Customers may place many orders but one order only consists of one customer.

A customer contributes many samples but only one sample pertains to one customer.

Customer information that must be recorded includes full name, date of birth, street, city state, zip code, email, and a phone number.

Orders

Orders are the centerpiece of the laboratory and initiated information being stored in many other tables. Specifically, when one and only customer submits an order through a form online, this triggers charges, customer tests, customer results, and samples to be collected. Specifically, each relationship is listed below:

An order is associated with only one charge (no variable pricing).

An order is associated with one 'test' panel and but many panels may be on one order

An order date and time must be initiated when a customer places an order.

Again, one order can only be associated with one customer but one customer may place many different orders. Note, one customer cannot place the same order at the same time though.

One order also prompts that one sample be collected for the order. One sample in this particular laboratory is never associated with many orders.

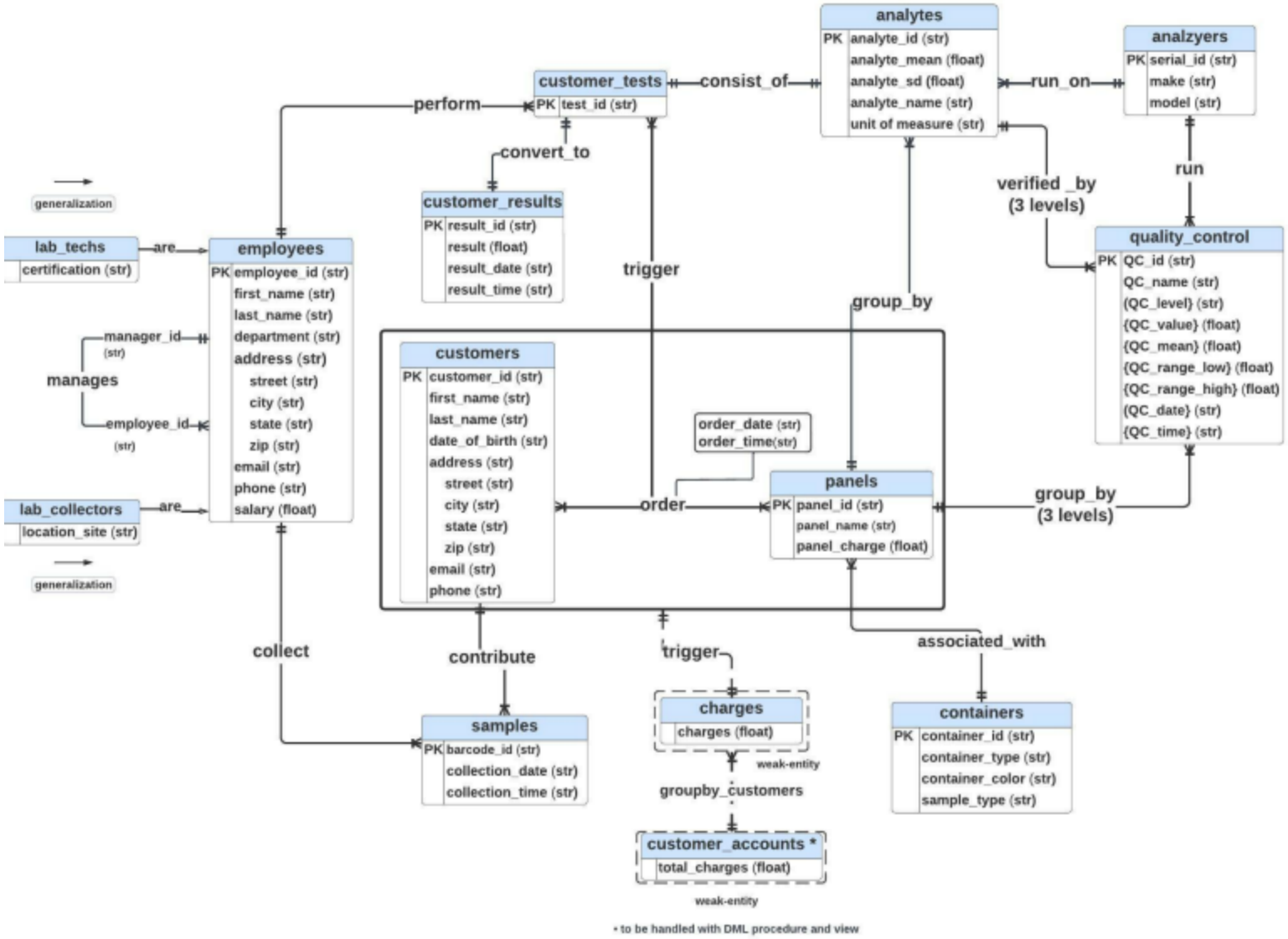
Orders also prompt customer tests to be triggered (through a laboratory information system) where many tests are associated with one order. In this laboratory, no order has just one test associated with it – customers must buy groups of tests called panels (cost effective).

Relational Diagram

Entity Relational Diagram:

The **Entity Relational Diagram** is a construct between abstract business logic that consist of the actual tasks of the enterprice and the hardwired logically wired database. The diagram is created from eliciting relationships and cardinality between entities to form a diagram that represents both the data structures and a level of business logic as seen by the verbose used to describe the relationship between two entity tables. Each table is an entity within the database that represents a major entity with the business where columns within the table are obviously the key features. The crow's feet notation seen below where hash marks form on the ends of each line represent the cardinality of the table opposite in reference to the table the marking connects to. For example, it can be said below that "one employee perform one or more customer tests(as indicated by the crow's feet), but

one customer test may only be associated with one employee(as indicated by the singular marks). Note, that two marks exists at the end of each line where the marking on the inside (furthest from the table) actually indicates if there can exist either zero or at least one instance in referrence to the table oppposite to the marking. For example, employees will always perform at least one test. In fact, no entity exist in this database where there could be no instances associated with another entity - this is simply due to the fact that an account is not created if so.

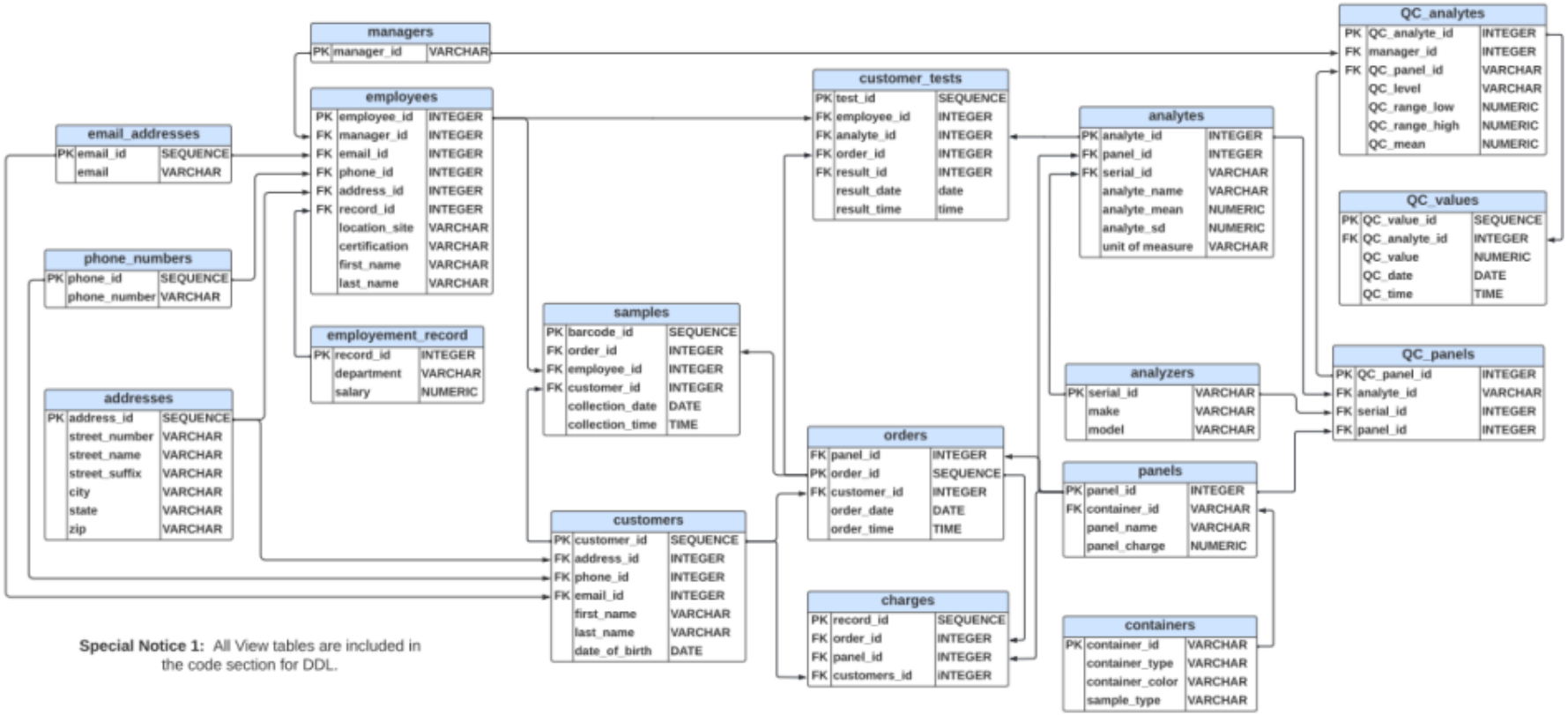


Logical Schema

Logical Schema - How it was Built:

The **Logical Schema Diagram** is a representation of the hardwired logically wired database that is built upon conecpts of *cardinality*, *constraints*, and *normalization* to just name a few key conecpts. The following below briefly describes how each of the three concepts were orchestrated in this project:

- **cardinality**: Atomicity explained below, is created by noting relationships between entities and the occurence of the number of instances in one entity as it relates to the other entity and vice versa. For example, it is noted that only one sample may be referred to one order or many test results may be tied to only one order. The idea that the orders entity (table) holds the capability of deriving to only one row instance adds accountability and efficiency as in the opposite case many instances referring to many other instances would create confusion and computational permutations that would lead to 'Cartesian explosions' of inefficent levels.
- **Foreign keys constraints** then can be created from accountability and the table that holds one instance only for many other instances serves as the 'Parent' table with a specific shared column serving as a reference to the 'Child' table with its many instances referring to only one instance in the 'Parent' table. Same is true to a one to one relationship, but in this case the 'Parent' table may be either on the tables. Finally, for relationships that are many to many, an intermediary table is created that contains the foreign keys of all cooperating tables and serves as the 'Parent' table. It is worth noting that by definition the 'Parent' table foreign key column must be a 'primary key' that simply states that all instances are unique and therefore atomicity is perserved.
- **constraints**: As mentioned above, constraints are used to preseve accountability of inter-relationships between two or more entities (foreign keys) and maintain uniqueness within an attribute (primary keys). Note, many more constraints exist mandating row security, user roles, and other functions that are not discussed in this project. The primary purpose of this project is simply to maintain a database to perform machine learning algorithms to demonstrate insights for enterprises.
- **normalization**: Normalization is the reduction of the possibly of errors due to DDL functions such as inserts, deletions, and updates. Thre process of normalizing a database typically follows a standardized form that follows the colloquial pattern of the 1NF to the 6NF as stated as the first normal form to the sixth normal form. Most enterprises only proceed to the third normal form (3NF) as this provides enough security in safely performing DDL functions while reducing the time and effort needed to maintain a 6NF database. This database tries to mimic a 3NF database where the idea that the most entities possible are created so that no one table could truly be represented as two or more tables. For example, the addresses contained in the 'Employees' table was allocated to an 'Addresses' table so that if an address were to change, it could simply be performed in the 'Addresses' table that would use a foreign key address_id to permutate all necessary updates in the 'Employees' table.



Create Tables

The `Python` function below was used to create the 'CREATE TABLE' statements:

Note: Any numbe of desired `SQL` data types may be interjected in the conditional statements below. The function below literally translates the `Pandas` data frame name into a `SQL` table, `Pandas` columns into `SQL` attributes, and `Pandas` datatypes into corresponding `SQL` datatypes. The statements must be checked manually as the function is just a a 'home-brewed' helper that simply reduces time needed to type out and format all the table statements as seen above.

Below is a SQL command creating an analyzer

The table is listed with its columns and subseqent data types. Note, one may include CHECK constraints for each column to very spefically describe what specific data may be enter to avoid data entry errors. Also, NOT NULL CONSTRAINTS exist to insert data with missing values for a particular column. These steps drastically reduce data cleaning needed in future usage of the database. Finally, the primary key constraint is place with the CREATE TABLE script to note the uniquely sequneced and identifying column.

In [ ]:

```
CREATE TABLE analyzers(
serial_id               varchar
```

```
        NOT NULL
        CHECK (serial_id in ('SN100000','SN100100','SN100300','SN100200')),
make            varchar
        NOT NULL CHECK(make <>'')
        CHECK (make in ('Roche', 'ACL', 'Sysmex')),
model           varchar
        NOT NULL CHECK(model <>'')
        CHECK (model in ('Cobas_C', 'Cobas_I', 'Tops', 'H1000')),
CONSTRAINT analyzers_pkey PRIMARY KEY(serial_id)
);
```

SQL comand that creates customers table

As we can see the pattern continues, with the only caveat we have introduced a serial data type that will auto-increment when we associate created sequences. This serves as pinnacle feature where most people obviously can not track or would be colicky to constantly search previous values prior to adding more data. The serial data type use with sequences also allows for the database to be more dynamic when associated with web applications that insert data without knowledge or prior values in the database.

In [ ]:
```
CREATE TABLE customers(
customer_id             serial,
address_id              serial,
phone_id                        serial,
email_id                        serial,
first_name                      varchar,
last_name                       varchar ,
date_of_birbbth         date,
CONSTRAINT customers_pkey PRIMARY KEY(customer_id)
);
```

Sequences

Below is an example of a sequence used for the orders table

Note, starting digits may be choosen at any number great if migrated databases and incrementation can be manipualted to follow a companies desired column.

In [ ]:
```
CREATE SEQUENCE orders_sequence
START 101000
INCREMENT 1;
ALTER TABLE orders
ALTER COLUMN  order_id
SET DEFAULT nextval('orders_sequence');
```

Foreign Key

Example `SQL` script for Foreign Key creation:

Foreign keys are developed by reviewing the logical schema; hwoever, most GUI-based database administration software such as PGAdmin4 can create schemas that include an option to indicate foreign keys. These are greate because they will create a script for you after using the GUI, but sometimes its quicker to literally type out the statements are create a Python program that may do so.

In [ ]:
```
ALTER TABLE Customers
ADD CONSTRAINT Customers_Addresses_fk
FOREIGN KEY (address_id)
REFERENCES Addresses (address_id);
```

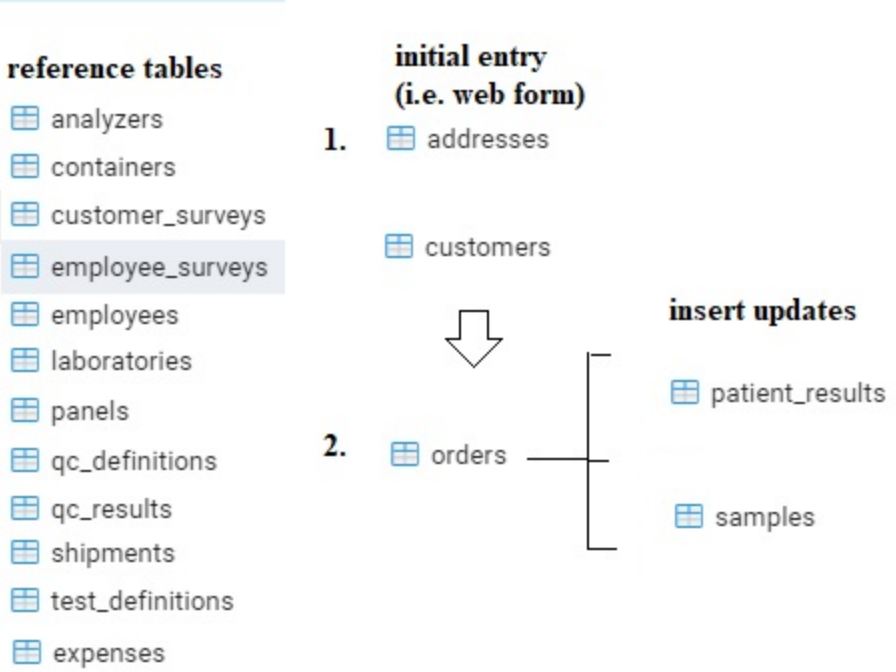
Triggers

Triggers created by procedures, specifically 'trigger procedures' create automated 'Inserts,Updates, Deletions'(IUD) when prompted by a specific action. In this particular project, inserting into customers, addresses, and orders prompts the update/insert of patient results and expenses mimicking if a patient result was actually performed on an analyzer and results were sent across a network.

Triggers used in this project are dependent on the use of sequences that are automated serial integers that represent the primary/foreign key columns. The autosequencing of such sequences are directly prompted per each trigger and therefore remain in parallel to the correct serial numbering of any column.

Furthermore, columns of non-primary/foreign keys, colloquially referred to as *business keys*, are created by `PL/PGSQL` functions such as `Now()`, `Random()`, and indirectly by *Cartesian products* with reference tables that simply contain data needed for a particular column.

Below demonstrates a basic diagramatic overview of such a *trigger system*:



A trigger that updates the charges table

When information is inserted in the orders table (when a new order is place) this triggers an automatic response based on the actions found within the trigger function. In this particular case, attributes panel_id, customer_id, and order_id relative to the most 'newly' added row are also inserted into the charges table. Note, using sequences the charges table primary key charges_id automatically updates. One may ask what about the actually charge? This database is very normalized so charges are simply held in a dictionary like table panels that contains a charge per each panel_id. As we shall see further on, the charges table simply just holds foreign keys that are used to create a view table that inherits such foreign keys by performing logically joins with other tables such as panels to form my complex tables that in this case will also include the charge. Have redundant data all over the database only leads to memory loss and potential issues!

In [ ]:

```
/* updates charges after */
CREATE OR REPLACE FUNCTION orders_charges_trig_func()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
INSERT INTO charges
 SELECT nextval('charges_sequence') as charge_id,
 new.panel_id,
 new.order_id,
 new.customer_id
  FROM orders, charges_sequence
  ORDER BY new.order_id desc
```

```
  LIMIT 1;
RETURN NULL;
END;
$$;

CREATE TRIGGER orders_charges
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE PROCEDURE orders_charges_trig_func();
```

A trigger that inserts into customer tests after an order is placed

After placing an order, the database creates the actual test report used by the lab to report out the patient results using the information from what test were ordered. This is an actually process handled by middleware or the middle person of the laboratory information system in many hospitals. Note, below a logical join of the orders, analytes, and panels tables are created to form the full report.

Note, since this was a mock laboratory database I have used mean values to create the actual results and the RANDOM function in SQL to randomly pick employees to create a more "reall feel".

In [ ]:

```
/*  updates customer tests after insert into orders*/
CREATE OR REPLACE FUNCTION orders_tests_trig_func()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
INSERT INTO customer_tests
SELECT  nextval('customer_tests_sequence') as test_id,
 o.employee_id,
 a.analyte_id,
 o.order_id,
ROUND(a.analyte_mean::numeric + (random()/2)::numeric, 1) AS result,
 o.result_date,
 o.result_time
 FROM(SELECT
floor(random()*(100035-100000+1))+100000 as employee_id,
 orders.order_id,
orders.panel_id,
orders.order_date as result_date,
 orders.order_time as result_time
  FROM orders, customer_tests_sequence
  ORDER BY orders.order_id desc
  LIMIT 1) as o
LEFT OUTER JOIN
(SELECT analytes.panel_id,
analytes.analyte_id,
```

```
analytes.analyte_mean
 FROM panels
LEFT OUTER JOIN analytes
 ON panels.panel_id = analytes.panel_id) as a
ON o.panel_id = a.panel_id;
RETURN NULL;
END;
$$;

CREATE TRIGGER orders_tests
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE PROCEDURE orders_tests_trig_func();
```

A trigger that creates lab functions after an order is placed

Laboratory orders prompt the creation samples in the laboratory that are used to collect the specimens that are tested. These samples are uniquely indentified by the primary key sample_id and this table also serves as a key source of the "phlebotomy view" that tracks samples needed to be collected.

In [ ]:

```
/*   updates samples table after insert into orders */
CREATE OR REPLACE FUNCTION orders_samples_trig_func()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN

INSERT INTO samples
 SELECT nextval('samples_sequence') as barcode_id,
 NEW.order_id,
 floor(random()*(100040-100000+1))+100000 as employee_id,
 NEW.customer_id,
NEW.order_date as collection_date,
 orders.order_time as collection_time
  FROM orders, samples_sequence
  ORDER BY NEW.order_id desc
  LIMIT 1;
RETURN NULL;
END;
$$;
CREATE TRIGGER orders_samples
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE PROCEDURE orders_samples_trig_func();
```

SQL View Tables (Materialized Views)

View tables are the quintessential talbes used for actual business functions. The tables stored in the logical schema may be used for business functions; however, the database is so normalized that the information in such table is limitting. Therefore, more data complete tables that typical involve a join of the database tables.

The materialized views may be formed from queries that use several techniques to join different tables, create aggregations, and transform data.

The primary constinuents of SQL Queries used:

- LEFT OUTER JOIN on a atomic parent table
- GROUP BY or OVER (ORDER BY) statements to create aggregated data sets
- WHERE or HAVING statements to serve as a filter
- Queries from the WHERE statement and FROM statement to adopt information from a table without a logical join. Note, this is a key way to join a table with an aggregated table without aggregate the other table as would be seen if a JOIN is used

Collector Views - Containers Needed View

This view is seen on a GUI based web application that sources data from the view created below. The view contains all pertinent information for a phlebotomist (in the case of this mock laboratory) to relate customer, testing, and order information logically linked in one statement. Note, the final NOT IN statement demonstrates order_id numbers that have not been found in the samples table therefore, samples ordered but without a sample yet created.

In [ ]:

```
/* Collector Views */
/* Containers Needed */
CREATE MATERIALIZED VIEW collector_containers_view
AS
SELECT row_number() OVER (ORDER BY samples.barcode_id,orders.customer_id) AS unique_id,
orders.order_id, orders.customer_id, customers.first_name, customers.last_name,
panels.panel_name, samples.barcode_id, containers.container_type, containers.container_color,
containers.sample_type
FROM orders
LEFT OUTER JOIN panels
ON orders.panel_id = panels.panel_id
LEFT OUTER JOIN containers
ON panels.container_id = containers.container_id
LEFT OUTER JOIN customers
ON orders.customer_id = customers.customer_id
LEFT OUTER JOIN samples
ON samples.order_id = orders.order_id
LEFT OUTER JOIN customer_tests
ON orders.order_id = customer_tests.order_id
WHERE orders.order_id
```

```
NOT IN
(SELECT samples.order_id
FROM samples);

CREATE TABLE collector_containers
AS
SELECT *
FROM collector_containers_view
ORDER BY unique_id;
ALTER TABLE ONLY collector_containers
ADD CONSTRAINT  collector_containers_pk PRIMARY KEY(unique_id );
```

Customer Reports for Lab Results

The following View Table not only links multiple tables (analytes, customer_tests, and orders) together, but also creates new columns using vector operations from different column as well as create a comment for each result using a CASE statement that serves as a conditional.

In fact, two view tables are created - one to calculate a calcualted tests called 'GFR' and antoher to simply obtain the patient results. The two tables are linked by order_id that serves as the primary key in a sense that it limits the result to a specific episode where an order can never be placed in duplicate at the same time as well have the same test twice.

Note, a 'quasi-primary key' is created using the "ELECT row_number() OVER (ORDER BY orders.order_id,orders.customer_id)" allows for a candidate key to be created that is prohibit by direct means in PostgreSQL. A candidate key allows for the combination of multiple columns to serve as a unique group for the primary key for a particular entity. This is essential in creating a unique row identifier for tables that have been aggregated where aggregations by their nature causes the primary key to be aggregate its self (multiple rows now are associated to one primary key - ie. customer_id).

In [ ]:

```
/* Customer Reports */

/* Calculate GFR */
DROP MATERIALIZED VIEW IF EXISTS report_GFR;
CREATE MATERIALIZED VIEW report_GFR
AS
SELECT orders.order_id, analytes.analyte_id,
ROUND(EXTRACT(YEARS FROM AGE(NOW(),customers.date_of_birth))/10 +
        customer_tests.result*10 + 60,1) as GFR
FROM customers
RIGHT OUTER JOIN orders
ON customers.customer_id = orders.customer_id
LEFT OUTER JOIN customer_tests
ON customer_tests.order_id = orders.order_id
LEFT OUTER JOIN analytes
ON analytes.analyte_id = customer_tests.analyte_id
WHERE analytes.analyte_name = 'CREATININE';
```

```
DROP TABLE IF EXISTS report_GFR_table;
CREATE TABLE report_GFR_table
AS
SELECT * FROM report_GFR ;

/* create table with alerts relative to customer results (+/- 2SD)*/
DROP MATERIALIZED VIEW IF EXISTS report_alerts;
CREATE MATERIALIZED VIEW report_alerts
AS
SELECT row_number() OVER (ORDER BY orders.order_id,orders.customer_id) AS unique_id,
        orders.order_id,  orders.customer_id, customers.first_name,
        customers.last_name, analytes.analyte_name, customer_tests.result,
        analytes.units_of_measure,      customer_tests.result_date, customer_tests.result_time,
        CASE
        WHEN (customer_tests.result - analytes.analyte_mean) > (2*analytes.analyte_sd)
        THEN 'High'
        WHEN (customer_tests.result - analytes.analyte_mean) < (-2*analytes.analyte_sd)
        THEN 'Low'
        ELSE 'Normal'
        END alert
FROM orders
LEFT OUTER JOIN customer_tests
ON customer_tests.order_id = orders.order_id
LEFT OUTER JOIN customers
ON customers.customer_id = orders.customer_id
LEFT OUTER JOIN analytes
ON customer_tests.analyte_id = analytes.analyte_id;


DROP TABLE IF EXISTS report_alerts_table;
CREATE TABLE report_alerts_table
AS SELECT * FROM report_alerts;
```

A View for Managers to View Patient Ranges

Patient Ranges are the cumulative stastistics of each test performed in the laboratory in terms of a mean (+/- 2 SD). The view below demonstrates the power of performing a "query from the FROM statement". This is a useful feature when associate a table that has been aggregated with a table that has not. Using JOIN statements between table causes any co-associated aggregations within the logical join to aggregate data in both table.

There are view tables that may need both aggregated and non-aggregated data from the same table. Here it is desireable to place the running aggregated mean with the static target mean within each row.

To obtain the aggregated mean, a sub query that aggreagates the patient results view as discussed above is placed after the from statement in an analytes tables where the aggregated table is on a cartesian join leaving the superqueried analytes table without aggregation.

The cartesian product is then filter by equated the serial id and analyte id of the two tables reducing the overall table to a discrete group of analytes.

In [ ]:

```
/* Manager Patient Ranges */
CREATE MATERIALIZED VIEW ranges AS
SELECT  agg.analyte_name,
                agg.count as N,
                agg.mean_actual,
                agg.sd_actual,
                analytes.analyte_mean as mean_target,
                analytes.analyte_sd as sd_target,
                analyzers.make,
                analyzers.model
FROM analytes,analyzers,
(SELECT analytes.analyte_name,
 COUNT(customer_tests.analyte_id),
 ROUND(AVG(customer_tests.result),2) AS mean_actual,
ROUND(STDDEV_SAMP(customer_tests.result),2) as sd_actual,
analytes.units_of_measure
FROM orders
LEFT OUTER JOIN customer_tests
ON customer_tests.order_id = orders.order_id
LEFT OUTER JOIN analytes
ON customer_tests.analyte_id = analytes.analyte_id
GROUP BY analytes.analyte_name, analytes.units_of_measure) as agg
WHERE agg.analyte_name = analytes.analyte_name
AND analytes.serial_id = analyzers.serial_id;

CREATE TABLE manager_range_report
AS
SELECT *
FROM ranges
ORDER BY analyte_name;

ALTER TABLE ONLY manager_range_report
ADD CONSTRAINT  manager_range_report_pk PRIMARY KEY(analyte_name);
```

Indexing

The use of indexing are used to increase performance and speed of querying the database by using sorting trees and data "arrangements" that reduces the dependency on searching row-by-row versus arrangements that create orderings that reduce such sequentiallity. For example, PostgreSQL defaults to arranging the indexed table (via selected columns) using B-Trees that use a merge sort (other formats exists) where a tree decorated with the indexed values contain parent values with child values that are either less or greater than the parent; thus, reducing the need to sort all values greater or less than the parent dependent on the value. In short, use indexing for tables to increase speed. Only tables with low number of rows will have tepid response since the indexing becomes negible as the need to sort reduces; however, in most cases indexing works!

An index is created on the primary key order_id for orders table but also includes the group by (panel_id, customer_id).

This particular indexing works well to speed up queries from the charges view that contain searches that are made for a particular test for a certain customer.

In [ ]:
```
CREATE UNIQUE INDEX order_id_idx ON orders (orders_id) INCLUDE (panel_id, customer_id);
```

Summary

The report has highlighted essential components needed to create a database schema. To create an entire database, one would simply use these components to create scripts for each of the entity tables or business functions. In fact, if you go to the GitHub repository I created for this project you will see that using these components tailored to specific needes a functional database may be created.

To conclude, a properly formed database schema wil create a structured backbone that will allow the foundations for business insights to be created from connecting to the database from any business insight software such as Power BI or Tableau.

The constraints made while creating the table and the logically connecting relationships formed from the foreign keys allow for confidence that queries are being properly performed.