

Project Overview

I have created a database management system for a car dealership. A car dealership includes multiple functions, aside from just selling cars. This includes providing service/maintenance, as well as selling individual parts for the cars the dealership owns.

The database contains information related to the above mentioned functions, in addition to others. Below are the high-level functions the database is used for:

- Car transactions
 - Buying
 - Selling
 - Trade-ins
- Service transactions
 - Performing different types of maintenance on cars
- Parts transactions
 - Customers purchasing individual parts needed for car
 - Customer can install part(s) themselves, or have parts installed through service transaction

Use Cases and Fields

Use Case #1: New Customer Account

Rationale:

- A new customer inquiries about buying a car
 - As such, the customer is asked to provide their name and contact information to be added to the system, after which a customer service assistant adds them to the system
 - The customer provides their contact information, after which they are contacted about the car they are interested in, as well as any general dealership promotions

Entity: <i>Customer</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Customer_ID	A unique ID which represents the customer's account	If, for example, two customer's shared the same name, a unique CustomerID would differentiate them	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Customer_First_Name	First name of customer	To confirm identity of new customer and address them when following up, whether through phone or email communication	VARCHAR(64)	
Customer_Last_Name	Last name of customer	To confirm identity of new customer and address them when following up, whether through phone or email communication	VARCHAR(64)	
Customer_Phone_Number	Phone number of customer	To confirm phone number of new customer and call back when following up with potential sale	DECIMAL(10)	10 digits, ignoring dashes (ex. if customer enters 860-867-5309, it will be shortened to 8608675309)
Customer_Email	Email address of customer	To email new customer when following up with potential sale, and to include them on mailing list	VARCHAR(64)	

Entity: <i>CustomerServiceAssistant</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Assistant_ID	A unique ID which represents the customer's account	If, for example, two customer's shared the same name, a unique CustomerID would differentiate them	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes

Assistant_First_Name	First name of customer	To confirm identity of new customer and address them when following up, whether through phone or email communication	VARCHAR(64)	
Assistant_Last_Name	Last name of customer	To confirm identity of new customer and address them when following up, whether through phone or email communication	VARCHAR(64)	

Structural Database Rules:

- For this use case, there are two relevant entities, *Customer* and *CustomerServiceAssistant*
 - Below are the structural database rules I see fit:
 - A customer is contacted by a customer service assistant. One customer service assistant contacts many customers.

Use Case #2: New Inventory Added to Dealership

Rationale:

- Relevant car information must be included to determine the value/sale price of the car
 - This includes new car's bought by the manufacturer, as well as car's purchases through a trade-in with a customer
- Also, if a customer currently owns a car (purchased at the dealership), knowing which car the customer owns allows for maintenance and the procurement of spare parts

Entity: <i>Car</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Car_ID	Vehicle Identification Number (VIN)	A unique ID to differentiate each car in the lot	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Model	The model of the car added	The model of the car signifies what type of car it is (within the brand that makes it)	VARCHAR(64)	

Color	The color of the car added	To differentiate the color of the cars (a dealership may have multiple cars of the same make/model, but with different colors)	VARCHAR(64)	
Status	Whether the car is new (ex. from manufacturer), or used (ex. through a trade with a customer)	To determine if a car is <i>New</i> or <i>Used</i>	CHAR(1)	
Number_Miles	If the car is used, the mileage of the car	The number of miles dictates the value of the car	DECIMAL(6)	To account for a car having over 100,00 miles but under 999,999 miles
Value	The value of the car, in dollars	The value dictates how appealing it is to potential customers to purchase	DECIMAL(6,2)	To allow the value of a car to be up to \$999,999, and account for decimal places (ex. \$25,000.25)

Structural Database Rules:

- For this use case, there are two entities at play. First, the *Car* entity, which includes all the relevant car information. Second is the *Customer* entity, to tie certain customers to a car
 - Below are the structural database rules I see fit:
 - A customer may own a car. Many cars may be owned by one customer
 - Adding to the structural database rules, below are the specialization-generalization rules I see fit:
 - The status of a car is either New or Used

Use Case #3: Car Purchased by Customer

Rationale:

- Once a car is purchased by the customer, the relevant sales information is recorded
 - This includes what car was sold, to which customer, as well as which salesperson recorded the sale

Entity: <i>Sale</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Sale_ID	A unique ID which represents a transaction	A unique ID to differentiate each sale of a car at the dealership	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Purchase_Date	The date the car was purchased	To determine when a car was purchased	DATE	
Purchase_Price	The price the car was sold for	Prior to when a car is finally purchased, negotiations and haggling are common between a dealership and a customer	DECIMAL(6,2)	To allow the value of a car to be up to \$999,999; and account for decimal places (ex. \$25,000.25)

Entity: <i>SalesPerson</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Sales_Person_ID	A unique ID which represents the different sales persons	If, for example, two sales persons' shared the same name, a unique sales_id would differentiate them	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Sales_First_Name	First name of sales person	To confirm sales person first name when logging who made the sale	VARCHAR(64)	
Sales_Last_Name	Last name of sales person	To confirm sales person last name when logging who made the sale	VARCHAR(64)	

Structural Database Rule:

- For this use case, there are four entities at play. They are *Customer*, *Car*, *Sale*, and *SalesPerson*. In order for a car to sell, there must be two sides to the transaction. They are the customer and sales person. Along with that, relevant information about the sale, such as which sales person completed the sale, must be recorded
 - Below are the structural database rules I see fit:
 - A customer may be associated with many sales. A sale is associated with one customer
 - A sale is associated with one car. A car may be associated with many sales
 - A salesperson may make many sales. A sale is made by one sales person

Use Case #4: Service/Maintenance Is Performed On A Vehicle

Rationale:

- A customer brings in a car to be serviced
 - All cars require routine maintenance, and the type of maintenance required varies greatly on the type of car, how many miles are on it, etc.

Entity: <i>ServiceTask</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Service_Task_ID	A unique ID which represents a particular type of service	A unique ID to differentiate each service task performed at the dealership	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Service_Type	The type of service being offered	To determine is a service is an <i>OilChange</i> , <i>TireRotation</i> , or <i>BrakeCheck</i>	CHAR(1)	
Service_Price	The amount of money it costs to perform a particular service	To determine how much a service task will cost	DECIMAL(5,2)	To allow the value of a service task to be up to \$99,999; and account for decimal places (ex. \$5,000.25)

Entity: <i>Mechanic</i>				
Field	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Mechanic_ID	A unique ID which represents the different mechanics	If, for example, two mechanics shared the same name, a unique mechanic_id would differentiate them	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Mechanic_First_Name	The first name of the mechanic who performed the service	To keep track of which mechanics work on which cars	VARCHAR(64)	
Mechanic_Last_Name	The last name of the mechanic who performed the service	To keep track of which mechanics work on which cars	VARCHAR(64)	

Structural Database Rule:

- For this use case, there are four entities at play. They are *Customer*, *ServiceTask*, and *Mechanic*. In order for service to be completed on a car, relevant information including the customer, their type of car, and which mechanic performed the service are needed
 - Below are the structural database rules I see fit:
 - A customer may purchase many service tasks. Many service tasks may be purchased by one customer
 - A mechanic performs many service tasks. One service task is performed by one mechanic
 - Adding to the structural database rule, below are the specialization-generalization rules I see fit:
 - A service task is either an oil change, tire rotation, brake check, all of these, or none of these
 - I have included “none of these” to note the possibility that there are other types of service tasks which may not be mentioned here

Use Case #5: Spare Part(s) Are Ordered By A Customer

Rationale:

- A customer wishes to purchase spare part(s) to install into their vehicle
 - Parts for a specific car may only be sold at the dealership

Entity: <i>Part</i>				
Attribute	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Part_ID	A unique ID which represents a particular part	A unique ID to differentiate each part the dealership currently has in inventory	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Part_Description	A short description of what the part is	To differentiate between the different parts in inventory	VARCHAR(255)	
Price	The value of the part	To determine how much a particular part will cost to purchase	DECIMAL(5,2)	To allow the value of a part to be up to \$99,999; and account for decimal places (ex. \$5,000.25)
Quantity	How many of a particular part the dealership has in stock	To ensure a customer can order a part	DECIMAL(3)	To allow for a quantity parts to be up to 999

Structural Database Rule:

- For this use case, there are four entities at play. They are *Customer*, *Car*, and *Part*. In order for a customer to purchase parts, it must be determined which parts are appropriate to purchase depending on their vehicle
 - Below are the structural database rules I see fit:
 - A customer may purchase many parts. Many parts may be purchased by one customer
 - Many parts are installed into many cars. Many cars may have many parts installed

Use Case #6: Apply Incentives To A Customer's Order

Rationale:

- Whenever a customer completes an order at the dealership (purchase car/performance maintenance/etc.), an incentive (or multiple incentives) are applied depending on the customer's history with the dealership and the type of car they drive (*Note: for this database, a customer can only have one incentive applied to their purchase*)
 - This is similar to the real world in terms of manufacturer holdbacks, whereas when certain cars are sold a discount is applied

Entity: <i>Incentive</i>				
Field	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Incentive_ID	A unique ID which represents a particular incentive	A unique ID to differentiate each incentive available to a customer	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Incentive_Type	The type of incentive offered by the dealership	To determine if an incentive is a <i>CustomerLoyaltyDiscount</i> or a <i>NewCarDiscount</i>	CHAR(1)	
Incentive_Description	A description of the incentive being offered	To provide a description of what type of incentive is being offered	VARCHAR(255)	
Incentive_Amount	The value of the incentive being offered	To calculate the final price when a customer makes a purchase	DECIMAL(3,2)	To allow the value of an incentive to be up to \$999; and account for decimal places (ex. \$100.25)

Structural Database Rule:

- For this use case, there are three entities at play. They are *Customer*, *Car*, and *Incentive*. When a customer at the dealership makes a purchase, incentives may be applied to the final cost. Relevant information about the customer, car, and available incentives all play into the calculation of the final price
 - Below are the structural database rules I see fit:

- A customer can be granted one incentive. One incentive is applied for one customer
- Adding to the structural database rule, below are the specialization-generalization rules I see fit:
 - An incentive is either a customer loyalty discount, new car discount, or both
 - I have included both to note the possibility a long-standing customer of dealership may wish to buy a new car

Normalization

In looking at my current database structure, one location I noticed for normalization is the make of the cars. The make of a car (ex. Honda, Toyota) will repeat frequently.

- That being said, I removed the *Make* attribute within the *Car* entity and created a separate *Car_Make* entity:

Entity: <i>CarMake</i>				
Field	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Car_Make_ID	A unique ID which represents a particular car make	A unique ID to differentiate each car make available in the lot	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
Car_Make_Name	The type of incentive offered by the dealership	To provide a description of what type of incentive is being offered	VARCHAR(64)	

Structural Database Rule:

- For this use case, there are two entities at play. They are *Car* and *Car_Make*. There are only a handful of car makes, to many car models.
 - Below are the structural database rules I see fit:
 - A car make is associated with many cars. A car is associated with one car make

Structural Database Rules

Below are the structural database rules based on the use cases mentioned above (highlighted in green), along with specialization-generalization rules (highlighted in blue):

- A customer is contacted by a customer service assistant. One customer service assistant contacts many customers.
- A customer may own a car. A car is be owned by one customer
- A sale is associated with one car. A car may be associated with many sales
- A salesperson may make many sales. A sale is made by one sales person
- A customer may be associated with many sales. A sale is associated with one customer
- A customer may purchase many service tasks. A service task may be purchased by one customer
- A mechanic performs many service tasks. One service task is performed by one mechanic
- A customer may purchase many parts. A part may be purchased by many customers
- Many parts are installed into many cars. A car may have many parts installed
- A customer can be granted one incentive. One incentive is applied for one customer
- A car make is associated with many cars. A car is associated with one car make
- A car has many mileage changes. A mileage change is associated with one car
- A car is either New or Used
- A service task is either an oil change, tire rotation, brake check, all of these, or none of these
- An incentive is either a customer loyalty discount, new car discount, or both

Indexes

Primary Key Indexes

Primary Key Column	Description
Customer.Customer_ID	This is the primary key of the Customer table
CustomerServiceAssistant.Assistant_ID	This is the primary key of the CustomerServiceAssistant table
Car.Car_ID	This is the primary key of the Car table
CarMake.Car_Make_ID	This is the primary key of the CarMake table
Part.Part_ID	This is the primary key of the Part table
Mechanic.Mechanic_ID	This is the primary key of the Mechanic table
ServiceTask.Service_Task_ID	This is the primary key of the ServiceTask table
Sale.Sale_ID	This is the primary key of the Sale table
SalesPerson.Sales_Person_ID	This is the primary key of the SalesPerson table
Incentive.Incentive_ID	This is the primary key of the Incentive table
Car-Parts-Link.Car_Parts_Link_ID	This is the primary key of the Car-Parts-Link table
Customer-Parts-Link.Customer_Parts_Link_ID	This is the primary key of the Customer-Parts-Link table
Customer-ServiceTask-Link.Customer_Service_Task_Link_ID	This is the primary key of the Customer-ServiceTask-Link table

Foreign Key Indexes

Foreign Key Column	Description	Unique?	Reasoning
Customer.Assistant_ID	This foreign key in the Customer table references the CustomerServiceAssistant table	N	Many customers are contacted by one customer service assistant
Car.Customer_ID	This foreign key in the Car table references the Customer table	N	Many cars can be owned by the same customer
Car.Car_Make_ID	This foreign key in the Car table references the CarMake table	N	Many car models are produced by only one manufacturer
ServiceTask.Mechanic_ID	This foreign key in the ServiceTask table references the Mechanic table	N	Many service tasks can be performed by the same mechanic
Sale.Customer_ID	This foreign key in the Sale table references the Customer table	N	Many sales can be associated with the same customer
Sale.Car_ID	This foreign key in the Sale table references the Car table	N	Many sales can be associated with the same the same car
Sales.Sales_Person_ID	This foreign key in the Sales table references the SalesPerson table	N	Many sales are completed by one Sales Person

Incentive.Customer_ID	This foreign key in the Incentive table references the Customer table	N	Many incentives can be associated with one customer
Car-Parts-Link.Car_ID	This foreign key in the Car-Parts-Link table references the Car table	N	Many parts can be installed into the same car
Car-Parts-Link.Part_ID	This foreign key in the Car-Parts-Link table references the Part table	N	Many parts can be installed into the same car
Customer-Parts-Link.Customer_ID	This foreign key in the Customer-Parts-Link table references the Customer table	N	Many parts can be purchased by one customer
Customer-Parts-Link.Part_ID	This foreign key in the Customer-Parts-Link table references the Part table	N	Many parts can be purchased by one customer
Customer-ServiceTask-Link.Customer_ID	This foreign key in the Customer-ServiceTask-Link table references the Customer table	N	Many service tasks can be purchased by one customer
Customer-ServiceTask-Link.Service_Task_ID	This foreign key in the Customer-ServiceTask-Link table references the ServiceTask table	N	Many service tasks can be purchased by one customer

Query-Driven Indexes

1) Car Model

- A car's model name is always unique. Also, it lies within the *Car* table, which will hold a large amount of data

```
SELECT Make  
FROM Car  
WHERE Model = "Civic"
```

2) Customer's Email Addresses

- While customers' first or last names may repeat (ex. John, ex. Smith), their email addresses will always be unique. Also, it lies within the *Customer* table, which will hold a large amount of data

```
SELECT First_Name, Last_Name  
FROM Customer  
WHERE Customer_Email = "mbrick@bu.edu"
```

3) Incentive Description

- Incentive descriptions will usually be unique, or have unique wording which may be used in a WHERE clause

```
SELECT Incentive_Type, Incentive_Amount  
FROM Incentive  
WHERE Incentive_Description LIKE "*high spending*"
```

Stored Procedure Execution & Explanations

Stored Procedure #1

- This stored procedure references ***Use Case #1: New Customer Account***
 - To set up this stored procedure, a customer service assistant was added to the *CustomerServiceAssistant* table, to allow new customers to be contacted by someone:



The screenshot shows a PostgreSQL query editor interface. At the top, the title bar reads "Term_Project/postgres@PostgreSQL 11". Below the title bar, there are two tabs: "Query Editor" and "Query History". The "Query Editor" tab is active, displaying a SQL query with line numbers 1 through 4. The query is an INSERT statement into the "CustomerServiceAssistant" table. Below the query editor, there are four tabs: "Data Output", "Explain", "Messages", and "Notifications". The "Messages" tab is active, showing a status message: "Query returned successfully in 77 msec."

```
1 INSERT INTO CustomerServiceAssistant(Assistant_ID, Assistant_First_Name, Assistant_Last_Name)
2 VALUES
3     (1, 'Lucy', 'Ricardo');
4
```

Data Output Explain Messages Notifications

Query returned successfully in 77 msec.

- Subsequently, the stored procedure was created:



The screenshot shows a PostgreSQL query editor interface. At the top, the title bar reads "Term_Project/postgres@PostgreSQL 11". Below the title bar, there are two tabs: "Query Editor" and "Query History". The "Query Editor" tab is active, displaying a SQL query with line numbers 1 through 21. The query is a CREATE FUNCTION statement for a stored procedure named "ADD_CUSTOMER". Below the query editor, there are four tabs: "Data Output", "Explain", "Messages", and "Notifications". The "Messages" tab is active, showing a status message: "Query returned successfully in 462 msec."

```
1 --create a new customer
2 CREATE OR REPLACE FUNCTION ADD_CUSTOMER (
3     arg_customer_id IN DECIMAL,
4     arg_assistant_id IN DECIMAL,
5     arg_cust_first_name IN VARCHAR,
6     arg_cust_last_name IN VARCHAR,
7     arg_cust_ph_number IN DECIMAL,
8     arg_cust_email IN VARCHAR)
9 RETURNS VOID LANGUAGE plpgsql
10 AS
11
12 --open the $$ quotation for the block
13 $reusableproc$
14 BEGIN
15     INSERT INTO Customer(Customer_ID, Assistant_ID, Customer_First_Name, Customer_Last_Name, Customer_Phone_Number, Customer_Email)
16     VALUES
17         (arg_customer_id, arg_assistant_id, arg_cust_first_name, arg_cust_last_name, arg_cust_ph_number, arg_cust_email);
18 END
19 --close the $$ quotation for the block
20 $reusableproc$;
21
```

Data Output Explain Messages Notifications

Query returned successfully in 462 msec.

- And new customers were added:

```
Term_Project/postgres@PostgreSQL 11
Query Editor Query History
1  START TRANSACTION;
2
3  --execute the stored procedure
4  DO
5  $$
6  BEGIN
7      EXECUTE ADD_CUSTOMER(1,1,'Jackie','Gleason','8605789632','jgleason@cbs.com');
8      EXECUTE ADD_CUSTOMER(2,1,'Rickie','Ricardo','8603499751','rricardo@cbs.com');
9      EXECUTE ADD_CUSTOMER(3,1,'Fred','Sanford','8603148522','fsanford@sanfordandson.com');
10  END;
11  $$;
12
13  --commit
14  COMMIT TRANSACTION;
```

Data Output Explain Messages Notifications

Query returned successfully in 100 msec.

Stored Procedure #2

- This stored procedure references ***Use Case #3: Car Purchased by Customer***
 - For this stored procedure, let's assume the customer ordered a new car from the dealership ahead of time, and will receive it immediately upon hitting the dealership
 - That being said, the *SalesPerson* and *CarMake* tables were initialized with data to properly execute the stored procedure:

```
Term_Project/postgres@PostgreSQL 11
Query Editor Query History
1  INSERT INTO SalesPerson (Sales_Person_ID, Sales_First_Name, Sales_Last_Name)
2  VALUES
3      (101,'Jordan','Belfort');
```

Data Output Explain Messages Notifications

Query returned successfully in 69 msec.

Term_Project/postgres@PostgreSQL 11

Query EditorQuery History

```
1 INSERT INTO CarMake(Car_Make_ID, Car_Make_Name)
2 VALUES
3     (1, 'Honda'),
4     (2, 'DeLorean');
```

Data OutputExplainMessagesNotifications

Query returned successfully in 74 msec.

- Subsequently, the stored procedure was created:

Term_Project/postgres@PostgreSQL 11

Query EditorQuery History

```
1 --new car purchased by customer
2 --add car and sale information
3 CREATE OR REPLACE FUNCTION ADD_SALE (
4     --Car Table
5     arg_car_id IN DECIMAL,
6     arg_customer_id IN DECIMAL,
7     arg_car_make_id IN DECIMAL,
8     arg_model IN VARCHAR,
9     arg_color IN VARCHAR,
10    arg_status IN CHAR,
11    arg_number_miles IN DECIMAL,
12    arg_value IN DECIMAL,
13
14    --Sale Table
15    arg_sale_id IN DECIMAL,
16    --arg_customer_id already created
17    --arg_car_id already created
18    arg_sales_person_id IN DECIMAL,
19    arg_purchase_date IN DATE,
20    arg_purchase_price IN DECIMAL)
21 RETURNS VOID LANGUAGE plpgsql
22 AS
23
24 --open the $$ quotation for the block
25 $reusableproc$
26 BEGIN
27     INSERT INTO Car(Car_ID, Customer_ID, Car_Make_ID, Model, Color, Status, Number_Miles, Value)
28     VALUES
29         (arg_car_id, arg_customer_id, arg_car_make_id, arg_model, arg_color, arg_status, arg_number_miles, arg_value);
30
31     INSERT INTO NewCar(Car_ID)
32     VALUES
33         (arg_car_id);
34
35     INSERT INTO Sale(Sale_ID, Customer_ID, Car_ID, Sales_Person_ID, Purchase_Date, Purchase_Price)
36     VALUES
37         (arg_sale_id, arg_customer_id, arg_car_id, arg_sales_person_id, arg_purchase_date, arg_purchase_price);
38
39 END
40 --close the $$ quotation for the block
41 $reusableproc$;
```

Data OutputExplainMessagesNotifications

Query returned successfully in 74 msec.

- And the sale was recorded:

```
Term_Project/postgres@PostgreSQL 11
Query Editor Query History
1 START TRANSACTION;
2
3 --execute the stored procedure
4 DO
5 $$
6 BEGIN
7     --Fred Sanford (id=3) purchasing a new Honda (id=1), compliments of sales person Jordan Belfort (id=101)
8     EXECUTE ADD_SALE(123456,3,1,'CR-V','Red','N',0,1500.00,111,101,'10/3/2019',1500.00);
9 END;
10 $$;
11
12 --commit
13 COMMIT TRANSACTION;
```

Data Output Explain Messages Notifications

Query returned successfully in 126 msec.

Stored Procedure #3

- This stored procedure references ***Use Case #5: Spare Part(s) Are Ordered By A Customer***
 - To set up this stored procedure, data was added to the following tables:
 - Part
 - Car/NewCar/UsedCar
 - Car_Parts_Link

```
Term_Project/postgres@PostgreSQL 11
Query Editor Query History
1 INSERT INTO Part (Part_ID, Part_Description, Price, Quantity)
2 VALUES
3     (1,'Oil Filter',10.00,3),
4     (2,'Brake Pads',50.00,1),
5     (3,'Wiper Blades',25.00,5),
6     (4,'Battery',50.00,10),
7     (5,'Spoiler',100.00,3);
```

Data Output Explain Messages Notifications

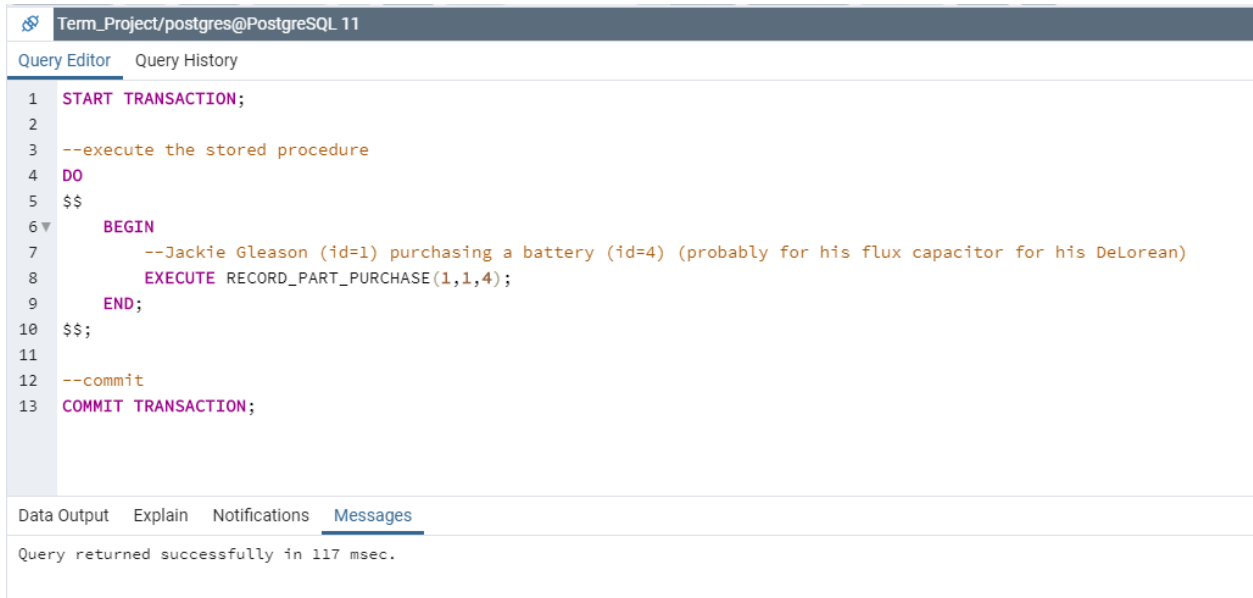
Query returned successfully in 196 msec.

```
Term_Project/postgres@PostgreSQL 11
Query Editor Query History
1 INSERT INTO Car(Car_ID, Customer_ID, Car_Make_ID, Model, Color, Status, Number_Miles, Value)
2 VALUES
3     --new car, unowned
4     (123457,NULL,1,'Odyssey','Grey','N',0,1250.00),
5     --Jackie Gleason owns a Honda Accord
6     (123458,1,1,'Accord','Blue','U',7500,1000.00),
7     --Jackie Gleason owns a DeLorean
8     (123459,1,2,'DMC-12','Blue','U',7500,750.00);
9
10 INSERT INTO NewCar(Car_ID)
11 VALUES
12     (123457);
13
14 INSERT INTO UsedCar(Car_ID)
15 VALUES
16     (123458),
17     (123459);
18
19 INSERT INTO Car_Parts_Link(Car_Parts_Link_ID, Car_ID, Part_ID)
20 VALUES
21     --oil filter can be installed on Honda Accord
22     (1,123458,1),
23     --wiper blades can be installed on Honda Accord
24     (2,123458,3),
25     --battery can be installed on DeLorean
26     (3,123459,4);
27
28
Data Output Explain Messages Notifications
Query returned successfully in 535 msec.
```

- Subsequently, the stored procedure was created:

```
Term_Project/postgres@PostgreSQL 11
Query Editor Query History
1 --parts purchased by customer for car
2 --add data to Customer_Parts_Link tables
3 CREATE OR REPLACE FUNCTION RECORD_PART_PURCHASE(
4     --Customer_Parts_Link_Table
5     arg_customer_parts_link_id IN DECIMAL,
6     arg_customer_id IN DECIMAL,
7     arg_part_id IN DECIMAL)
8 RETURNS VOID LANGUAGE plpgsql
9 AS
10
11 --open the $$ quotation for the block
12 $reusableproc$
13 BEGIN
14     --Customer_Parts_Link Table
15     INSERT INTO Customer_Parts_Link(Customer_Parts_Link_ID, Customer_ID, Part_ID)
16     VALUES
17         (arg_customer_parts_link_id, arg_customer_id, arg_part_id);
18
19 END
20 --close the $$ quotation for the block
21 $reusableproc$;
22
Data Output Explain Notifications Messages
Query returned successfully in 414 msec.
```

- And a record of what part the customer purchased was recorded:



The screenshot shows a PostgreSQL Query Editor window with the title bar "Term_Project/postgres@PostgreSQL 11". The window has two tabs: "Query Editor" and "Query History". The "Query Editor" tab is active and displays a SQL script with line numbers 1 through 13. The script is as follows:

```
1  START TRANSACTION;
2
3  --execute the stored procedure
4  DO
5  $$
6  BEGIN
7      --Jackie Gleason (id=1) purchasing a battery (id=4) (probably for his flux capacitor for his DeLorean)
8      EXECUTE RECORD_PART_PURCHASE(1,1,4);
9  END;
10 $$;
11
12 --commit
13 COMMIT TRANSACTION;
```

Below the query editor, there is a section with four tabs: "Data Output", "Explain", "Notifications", and "Messages". The "Messages" tab is selected and shows the message: "Query returned successfully in 117 msec."

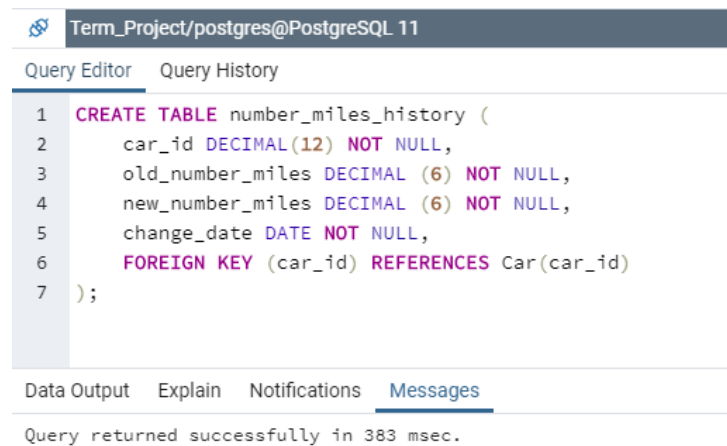
Trigger Creation & Use

- The trigger that I decided to create was create a history table for the *Number_Miles* attribute within the *Car* table
 - This trigger was chosen to resemble a real life scenario whereas a customer brings in their car (ex. for service, etc.) and the updated mileage of the car is recorded. Keeping a record of the mileage allows a dealership to make recommendations as to what the car may need in terms of parts or service
 - Below is the new entity and its attributes:

Entity: <i>number_miles_history</i>				
Field	What It Stores	Why It's Needed	Datatype	Datatype Reasoning (if applicable)
Car_ID	Vehicle Identification Number (VIN)	A unique ID to differentiate each car in the lot	DECIMAL(12)	Primary keys are unique 12 digit-numeric codes
old_number_miles	The old mileage of a car	A keep a record of the old mileage of a particular car	DECIMAL(6)	
new_number_miles	The new mileage of a car	A keep a record of the new mileage of a particular car	DECIMAL(6)	
change_date	The date of the mileage change	To keep a record of the mileage change of a particular car	DATE	

- Structural Database Rules
 - For this use case, there are two entities at play. They are *Car* and *number_miles_history*
 - Below are the structural database rules I see fit:
 - A car has many mileage changes. A mileage change is associated with one car

- Creating the history table:



The screenshot shows a PostgreSQL Query Editor window with the title 'Term_Project/postgres@PostgreSQL 11'. The 'Query Editor' tab is active, displaying a SQL query to create a table named 'number_miles_history'. The query is as follows:

```
1 CREATE TABLE number_miles_history (  
2     car_id DECIMAL(12) NOT NULL,  
3     old_number_miles DECIMAL (6) NOT NULL,  
4     new_number_miles DECIMAL (6) NOT NULL,  
5     change_date DATE NOT NULL,  
6     FOREIGN KEY (car_id) REFERENCES Car(car_id)  
7 );
```

Below the query editor, there are tabs for 'Data Output', 'Explain', 'Notifications', and 'Messages'. The 'Messages' tab is selected, showing the message: 'Query returned successfully in 383 msec.'

- Creating the trigger:



The screenshot shows a PostgreSQL Query Editor window with the title 'Term_Project/postgres@PostgreSQL 11'. The 'Query Editor' tab is active, displaying a SQL query to create a trigger named 'trg_NUMBER_MILES_HISTORY'. The query is as follows:

```
1 --create function for trigger of number_miles change  
2 CREATE OR REPLACE FUNCTION NUMBER_MILES_HISTORY ()  
3 RETURNS TRIGGER LANGUAGE plpgsql  
4 AS  
5  
6 --open $$ for trigger creation  
7 $$  
8 BEGIN  
9     IF OLD.Number_Miles <> NEW.Number_Miles THEN  
10         INSERT INTO number_miles_history (car_id, old_number_miles, new_number_miles, change_date)  
11         VALUES  
12             (NEW.car_id, OLD.Number_Miles, NEW.Number_Miles, CURRENT_DATE);  
13     END IF;  
14 RETURN NEW;  
15 END;  
16 $$;  
17  
18 --create trigger  
19 CREATE TRIGGER trg_NUMBER_MILES_HISTORY  
20 BEFORE UPDATE ON Car  
21 FOR EACH ROW  
22 EXECUTE PROCEDURE NUMBER_MILES_HISTORY();  
23
```

Below the query editor, there are tabs for 'Data Output', 'Explain', 'Notifications', and 'Messages'. The 'Messages' tab is selected, showing the message: 'Query returned successfully in 138 msec.'

- Let's take another look at the *Car* table as a refresher before testing the trigger:

Term_Project/postgres@PostgreSQL 11

Query Editor Query History

```
1 SELECT * FROM Car;
2
3
```

Data Output Explain Notifications Messages

	car_id [PK] numeric (12)	customer_id numeric (12)	car_make_id numeric (12)	model character varying (64)	color character varying (64)	status character (1)	number_miles numeric (6)	value numeric (6,2)
1	123456	3	1	CR-V	Red	N	0	1500.00
2	123457	[null]	1	Odyssey	Grey	N	0	1250.00
3	123458	1	1	Accord	Blue	U	7500	1000.00
4	123459	1	2	DMC-12	Blue	U	7500	750.00

- Let's update the mileage for **Car_ID = 123459**:

Term_Project/postgres@PostgreSQL 11

Query Editor Query History

```
1 UPDATE Car
2 SET Number_Miles = 12500
3 WHERE Car_ID = 123459;
4
```

Data Output Explain Notifications Messages

Query returned successfully in 70 msec.

- Checking to ensure the *number_miles_history* table was successfully updated:

Term_Project/postgres@PostgreSQL 11

Query Editor Query History

```
1 SELECT * FROM number_miles_history;
2
3
```

Data Output Explain Notifications Messages

	car_id numeric (12)	old_number_miles numeric (6)	new_number_miles numeric (6)	change_date date
1	123459	7500	12500	2019-10-04

Question Identification & Explanations (w/ Query Execution)

Question #1

- The first question I would like to answer is ***what are the current inventory levels?***
 - Specifically, it would be nice to know what parts are currently low on inventory, as well as which parts are currently high on inventory. This is useful for a couple of reasons:
 - First, knowing which parts are low allows the dealership to order more, so they do not run out
 - Second, knowing parts with high inventory allows the dealership to try to sell them to customers (ex. through a promotion)
 - Additionally, If a customer has previously bought a specific part, the dealership knows **not** to tailor their promotion to them



The screenshot shows a PostgreSQL query editor interface. The top bar indicates the connection is 'Term_Project/postgres@PostgreSQL 11'. Below the bar are tabs for 'Query Editor' and 'Query History'. The query editor contains the following SQL query:

```
1 SELECT Part_Description, Quantity, Customer_First_Name, Customer_Last_Name
2 FROM Part
3 LEFT JOIN Customer_Parts_Link
4     ON Part.Part_ID = Customer_Parts_Link.Part_ID
5 LEFT JOIN Customer
6     ON Customer_Parts_Link.Customer_ID = Customer.Customer_ID
7 ORDER BY Quantity ASC;
8
```

Below the query editor are tabs for 'Data Output', 'Explain', 'Notifications', and 'Messages'. The 'Data Output' tab is active, displaying a table with the following data:

	part_description character varying (255)	quantity numeric (3)	customer_first_name character varying (64)	customer_last_name character varying (64)
1	Brake Pads	1	[null]	[null]
2	Spoiler	3	[null]	[null]
3	Oil Filter	3	[null]	[null]
4	Wiper Blades	5	[null]	[null]
5	Battery	10	Jackie	Gleason

- Based on the above, brake pads are low on inventory and the dealership must order more. Also, Jackie Gleason is the only customer to purchase a part so far, specifically a car battery. That being said, all customer other than him can be targeted to sell the high inventory parts, such as wiper blades or another car battery

Question #2

- The second question I would like to answer is **who are our best customers, in terms of car sales?**
 - Specifically, who are our “repeat” customers who have bought multiple cars from us? Knowing who our best customers are allows us to provide them with a discount or incentive, to keep them happy and continue buying from us

Term_Project/postgres@PostgreSQL 11			
Query Editor Query History			
<pre>1 SELECT Customer_First_Name, Customer_Last_Name, COUNT(Car_ID) AS cars_purchased 2 FROM Customer 3 INNER JOIN Car 4 ON Car.Customer_ID = Customer.Customer_ID 5 GROUP BY Customer_First_Name, Customer_Last_Name 6 HAVING COUNT(Car_ID) > 1;</pre>			
Data Output Explain Notifications Messages			
	customer_first_name character varying (64)	customer_last_name character varying (64)	cars_purchased bigint
1	Jackie	Gleason	2

- Based on the above, Jackie Gleason is currently the only repeat customer in terms of purchasing cars. He will be sure to get a discount next time he comes to the dealership!

Question #3

- The third question I would like to answer is **what is the mileage difference between when a customer purchased a car, versus when they last brought it into the shop?**
 - Knowing the answer to this question is useful in terms of knowing what services to offer. In the real world, each make/model of a car has different mileage restrictions (ex. every 10,000 miles) to bring it in for its next service appointment
 - This difference is key to making the proper suggestions to the customer to keep their car running smoothly

Term_Project/postgres@PostgreSQL 11				
Query Editor Query History				
<pre> 1 SELECT Customer_First_Name, Customer_Last_Name, Model, SUM(new_number_miles - old_number_miles) AS mileage_difference 2 FROM number_miles_history 3 INNER JOIN Car 4 ON Car.Car_ID = number_miles_history.Car_ID 5 INNER JOIN Customer 6 ON Customer.Customer_ID = Car.Customer_ID 7 GROUP BY Customer_First_Name, Customer_Last_Name, Model; </pre>				
Data Output Explain Notifications Messages				
	customer_first_name character varying (64)	customer_last_name character varying (64)	model character varying (64)	mileage_difference numeric
1	Jackie	Gleason	DMC-12	5000

- Based on the above, Jackie Gleason brought his DMC-12 in for maintenance after 5000 miles. That being said, the dealership can now make the appropriate suggestions in terms of what types of service his car specifically needs