Discount Given Project Direction Overview

I would like to develop a system which can help a staff who works for an online clothing shopping mall give discount to customers, named “Discount Given”. When staff want to send customers a discount, they have to send different customers different discount. If a company send customers same discount, customers may not feel they gain benefit. Specific, on Christmas Day, if the store sends same discounts to a customer who has just spent 50 dollars before and a customer who is a VIP and has already spent 1000 dollars before, the last customer will think they are not be treated as a VIP. So by using “Discount Given” system, staff can distribute benefits more flexibly.

Here are some brief examples for staff of how to use this system. Martin is a staff to send discounts to costumers via e-mail. His work is selecting the specific customers and sending discount e-mails to them. On Black Friday, his company decided to give 50% off to the customers who are level 4 and above, 35% off to level 3, 25% off to level 2 and 10% off to level 1. By using “Discount Given”, Martin can easily group customers by level. Meanwhile, one of his daily work is send 20% discount to people who have birthdays. He can just type today’s date and the system will help he pick up people who are in birthday.

My “Discount Given” systems have two functions (maybe added in the future):

- Given different discount to people of different member level
- Given a 20% discount to people who have birthdays
- Different types of goods will have different discount dates, and people will be given discounts on the discount days. It contains two cases:
  1) Customer has already bought a jacket. On the winter jacket discount day, send this customer a 15% discount when he buys a jacket again.
  2) Customer hasn’t bought a jacket, but he puts the jacket in the shopping cart. On the winter jacket discount day, send this customer a 20% discount to buy jacket.
- When the purchase price of the goods from suppliers’ is reduced, a discount of 50% of the reduced purchase price can be given to the customers.
- Send $5 coupons to users who haven't logged in for a month.

“Discount Given” system will contain customers’ e-mail address, name, birthday, phone number, level. It will also contain a table about the level grading criteria.

Why I am interested in this system? Through daily observation, I find that shopping websites will give me different discounts in different situations. I didn't want to buy anything, but I bought because of the discounts. At the same time, I find that different discounts have different effects on stimulating my purchase desire. So I want to make some rules about sending discounts to help companies improve customers’ desire. For this purpose, a database system is important to help them.

Discount Given Use Cases and Fields

For staff, in order to search different customers, the customers' information is important.The staff can search the keyword "level 1" in "Discount Given" systems, and the system will show the customers who are in level 1

Customers' Information Case

1. The customer visited the online store.
2. The customer fill in a e-form to create a account by using his e-mail address.
3. The customer's level is 1 with 0 point
4. The customer buy some items and 1 dollars is 1 point.

A **Customer** table is needed to record this.

Field	What is stored	Whether is NULL	Why it's needed
Cus_Id	Unique number for each customer	NOT NULL	It is the primary key for this table. Use this Cus_id to identify each customer.When the customer creates an account with the mailbox, a corresponding ID will be generated
Email-Address	The customer's email address	NOT NULL	Customer should use it to create his account. The company's staff can use it to send discounts.
Nickname	The customer's nickname	NOT NULL	This is necessary for displaying the person's name on screens and addressing them when sending them emails or other communications.
Phone Number	The customer's phone number	Can be NULL It is optional.	Some specific situation, the company should communicate with customers and the customers don't reply the e-mail.

Field	What is stored	Whether is NULL	Why it's needed
Last_log	This is the date the account last time logged in	NOT NULL	It is helpful to judge the last login time of customers, and we can give discount to users who have not logged in for a long time.
Level	The customer's level	NOT NULL	The criteria for sending discounts
Points	The customer's point	NOT NULL	The criteria for classifying level

Note: Do not use email address as primary key, because email address is complex and prone to spelling errors.

Item Case

1. The database should also hold information about all products. We need an **Item** table.

Field	What is store	Whether is Null	Why is needed
Item_id	The item's id	NOT NULL	Used to distinguish each item. It is the primary key for this table.
Item_name	The item's name	NOT NULL	
Item_type	The item's type	NOT NULL	Put each item in a different type and distribute the coupon according to the type. For example, in summer, all kinds of dresses are on sale
purchasing_p	Purchase price from supplier	NOT NULL	Record the purchase price from the supplier. When the price decreases, we can give the customer some discounts.
selling_p	Price of goods sold on the website	NOT NULL	Record selling price of products, so as to calculate the points gained by customers

2. At the same time, each product belongs to a different type. Depending on the type, we distribute discounts about that type on a given date, which requires us to design a **Type** table to store different types and the time sent discounts.

Field	What is store	Whether is NULL	Why is needed
Item_type	Different types, like jacket, dress, coat, etc.	NOT NULL	Used to store all types of clothes.
sale_date	Promotion days for each type of product	It can be NULL Some types of products may don't have a promotion day	On these dates, we can find which type has a discount. Through form Item, we can find the corresponding item.

Level defined Case

1. The customer buys items on the website and for every one dollars, he will get one point
2. At the beginning, the customer's level is 1 with 0 point. If he bought 50 dollars in the website, he will get 50 points and his level is up to 2.

A table **Level** to store the level and the criteria is needed.

Field	What is store	Whether is NULL	Why is needed
Level	The level: 1—4	NOT NULL	It is necessary
Point	The criteria for classify the level. Eg. Level 1 — 0-50 points	NOT NULL	It is necessary to classify the level
Privilege	Customer's privilege, like free shipped.	NOT NULL	Provide customers with better shopping experience by providing different privileges

Customer Purchase Case

1. The system needs an **Order** table to record every order generated on the website. At the same time, record the total prince of each order in this table. By using the total price, the system can calculate it's point and decide the level.

field	What is store	Whether is NULL	Why is needed
Order number	The purchase number	NOT NULL	Every purchase will create a order number. Using order number can track each purchase easily
Cus_Id	The customers' id	NOT NULL	To identify customers

field	What is store	Whether is NULL	Why is needed
Total price	The order's total price	NOT NULL	To calculate the point

2. One or more items may be purchased for each order, so we need a table **Order_item** to record the items purchased for each order.

Field	What is store	Whether is NULL	Why is needed
Item_id	ID of the item purchased	NOT NULL	Used to query purchased goods
Item_type	Item type, like jacket, dress, etc.	NOT NULL	Used to identify items' type. By searching the type, we can send this type of coupon to customers.
Cus_Id		NOT NULL	It is used to record who buy this item

3. According to the type in table Order_item, we can find the customers who buy this type of item. On the discount day of the specified type of product, the company can give these customers this type of discount. For example, in Item_type table, we find that June 1 is the discount day of dresses every year. Then, we find all customers who have bought dresses in table Order_item, and then get the email address of these customers according to Table Customer, and send them the discount.

Add Shopping Cart Case

In addition to buying items, customers often put the desired items in the shopping cart. So it's easier to promote customers' consumption by sending corresponding coupons according to the goods in the shopping cart. We need a **Cart** table to store what customers want to buy.

Field	What is store	Whether is NULL	Why is needed
Item_id	ID of the item added in cart	NOT NULL	Used to query the items saved in the shopping cart
Cus_Id	corresponding custom	NOT NULL	Used to see who saved this product. By using this, we can get customers' email in Customer table

If a customer put a dress in her shopping cart. On June 1, the discount day of dresses, I can use Item table to select the item_id which belongs to type 'dress'. Then in Cart table, I can know customers who put these item in their cart. Finally, in Customer table, I can know their email address.

Discount Given Structural Database Rules

Case 1: Customers' Information Case

1. The customer visited the online store.
2. The customer fill in a e-form to create an account by using his e-mail address.
3. The customer's level is 1 with 0 point
4. The customer buy some items and 1 dollars is 1 point.

In this case, I only pay attention to the accounts registered by customers. I store the information of customers in table **Customer.** From step 2, I see one entity – Customer. In looking through the other steps, I do not see any other entity or relationship. I could attempt to break down Customer into multiple entities with relationships, but from this use case alone I don't have enough information. Instead, I will keep in mind the Customer entity is needed for the database, and move on to the next use case.

Case 2: Item Case

1. The database should also hold information about all products. We need an **Item** table.
2. Each product belongs to a different type.

Here, we have two entities, one is item, the other is type.

Structural Rule:
Each item belongs to a type, and each type owns many items.

I created this structural rule because each clothing should belong to a type, such as dress, jacket, etc. But there should be a lot of clothes in each type.

Case 3: Level defined Case

1. The customer buys items on the website and for every one dollars, he will get one point
2. At the beginning, the customer's level is 1 with 0 point. If he bought 50 dollars in the website, he will get 50 points and his level is up to 2.

In this case, we can see the entities: Level and Customer.

Structural Rule:
Every customer has a level, different levels will have many customers.

This rule is because depending on how much the customer spends on the website, we assign a level to the customer. There may be many qualified customers at each level.

Case 4: Customer Purchase Case

1. The system needs an **Order** table to record every order generated on the website. At the same time, record the total prince of each order in this table. By using the total price, the system can calculate the point and decide the level.

2. One or more items may be purchased for each order, so we need a table **Order_item** to record the items purchased for each order.
3. According to the type in table Order_item, we can find the customers who buy this type of item. On the discount day of the specified type of product, the company can give these customers this type of discount. For example, in Item_type table, we find that June 1 is the discount day of dresses every year. Then, we find all customers who have bought dresses in table Order_item, and then get the email address of these customers according to Table Customer, and send them the discount.

Here, there are five entities: Customer, Order, Order_item, Item, Item_type, Level.

First Rule:
Each customer can have many orders, and each order can only belong to one customer.

I create this rule because customers can purchase items on the website many times. Each order generates a unique order number, which corresponds to a row of the order table. Each order can only be ordered by one customer.

Second Rule:
Each order can include one or more items, and each item can belong to one or more orders.

This rule is because customers may buy one or more items at a time. Items can also be purchased by one person or by many people.

Third Rule:
Each item_id in Item table corresponds to no items or more purchased records in Order_item table, and each purchased record in Order_item table corresponds to one item_id in Item table.

This rules is because in the table Order_item, each row represents an item to be purchased, so the item_id may be repeated (because it can be purchased one or more times), but no matter how many times it is repeatedly purchased, each item only corresponds to one item_id in table Item.

Fourth Rule:
Each item_type in Item_type table corresponds to no item or more purchased records in Order_item table, and each purchased record in Order_item table corresponds to one item_type in Item_type table.

This rule is similar to third rule. The differences is that the third rule is relationship between Order_item and Item table, but the fourth rule is relationship between Order_item and Item_type table. In Order_item table, each row represents an item to be purchased and each item has only one type, so the item_type may be repeated (because each type can be purchased one or more times), but no matter how many times it is repeatedly purchased, each item_type only corresponds to one item_type in table Item_type.

Case 5: Add Shopping Cart Case

1. In addition to buying items, customers often put the desired items in the shopping cart. So it's easier to promote customers' consumption by sending corresponding coupons according to the goods in the shopping cart. We need a **Cart** table to store what customers want to buy.
2. If a customer put a dress in her shopping cart. On June 1, the discount day of dresses, I can use Item table to select the item_id which belongs to type ‘dress’. Then in Cart table, I can

know customers who put these item in their cart. Finally, in Customer table, I can know their email address.

Here, we can see the entities: Customer, Cart, Item

First Rule:
Each customer puts no item or one or many items in cart and each item in cart belongs to a customer.

This rule implies that every customer can put a lot of goods in the shopping cart or not, but every product in the shopping cart is put in by the customer.

Second Rule:
Each item_id in Item table corresponds to one or more records in Cart table, and each record in Cart table corresponds to one item_id in Item table.

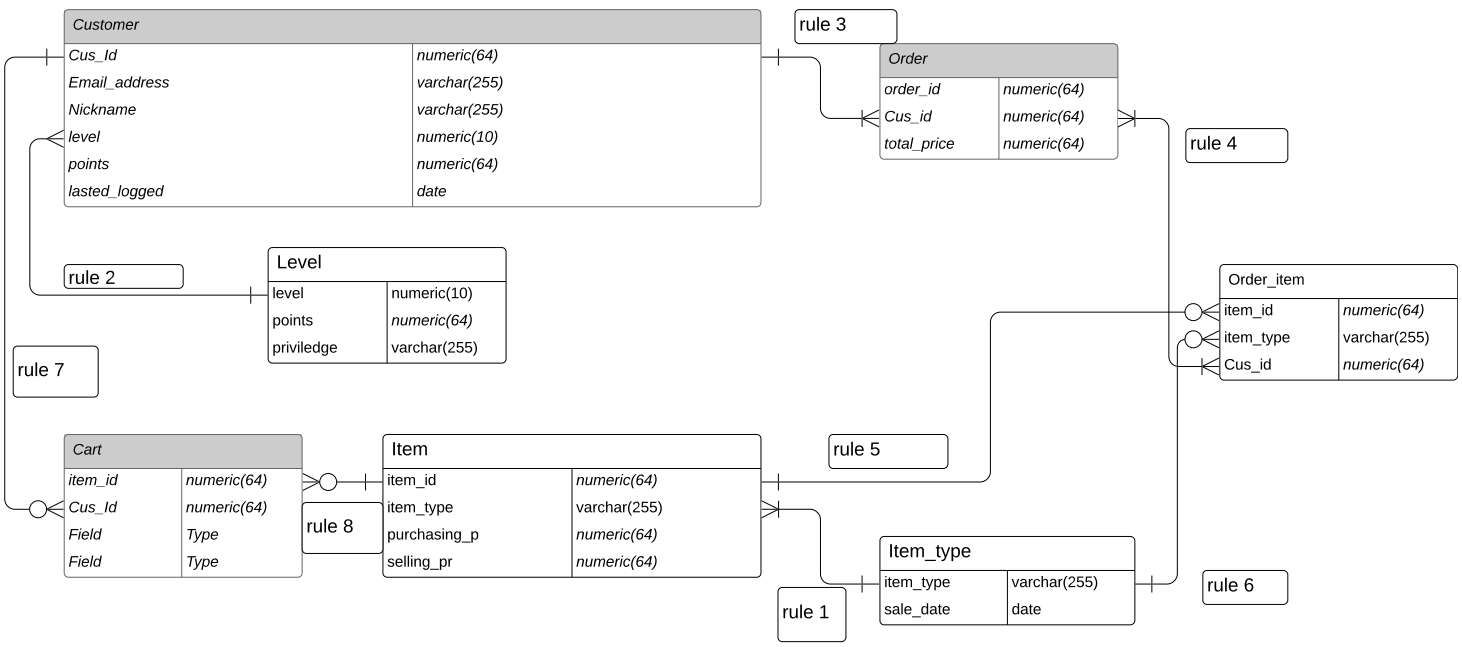
This rule indicates that in Cart table, many same item may be added in cart, but same item corresponds to only one item_id in Item table. In item table, each item_id can correspond to no item or one or many items in Cart table. That is because, same items may be added or not.

Initial Discount Given ERD

Here is my rules in iteration 2:

1. *Each item belongs to a type, and each type owns many items.*
2. *Every customer has a level, different levels will have many customers.*
3. *Each customer can have many orders, and each order can only belong to one customer.*
4. *Each order can include one or more items, and each item can belong to one or more orders.*
5. *Each item_id in Item table corresponds to no items or more purchased records in Order_item table, and each purchased record in Order_item table corresponds to one item_id in Item table.*
6. *Each item_type in Item_type table corresponds to no item or more purchased records in Order_item table, and each purchased record in Order_item table corresponds to one item_type in Item_type table.*
7. *Each customer puts no item or one or many items in cart and each item in cart belongs to a customer.*
8. *Each item_id in Item table corresponds to one or more records in Cart table, and each record in Cart table corresponds to one item_id in Item table.*

Here is my initial ERD



Adding Specialization_Generalization to Discount Given

Case 1: Customers' Information Case

1. The customer visited the online store.
2. The customer fill in a e-form to create an account by using his e-mail address.
3. The customer's level is 1 with 0 point
4. The customer buy some items and 1 dollars is 1 point.

In this case, customer can buy products as a member or as a guest. So in step 2, customer have two choice: fill in a e-form to create an account or login in as a guest. Customer who use guest account just offer an e-mail address to receive the order information and a real address to receive the package.

The difference between member account and guest account is guest account does not have a level. When people buy products via guest account, he will not receive points. So they can not

get level discount. Also, guest account didn't provide their birthday, so they can not get birthday coupons. But, guest account provide their e-mail, so they can get other discount, like getting discount according to their cart. In customer's table, the field: nickname, points and level are NULL.

Case 1: Customers' Information Case (New)

1. The customer visited the online store.
2. The customer fill in a e-form to create an account by using his e-mail address or just login as a guest.
3. The member customer's level is 1 with 0 point. The guest customer does not have a level.
4. The member customer buy some items and 1 dollars is 1 point. The guest customer does not get any point.

Notice that #2 now mentions a member account or a guest account. I derive a ninth structural database rule to support the change to the use case as follows.

An account is a member account or a guest account.

This specialization-generalization rule turned out to be quite short, but does capture the intent. My database only has two kinds of accounts – member and guest – and that is the complete list. The relationship is totally complete. Customer must buy products via member account or guest account. So the relationship is disjoint. I did not put any of the verbiage such as "several of these" or "none of these" since the rule is totally complete and disjoint.

Case 3: Level defined Case

1. The customer buys items on the website and for every one dollars, he will get one point
2. At the beginning, the customer's level is 1 with 0 point. If he bought 50 dollars in the website, he will get 50 points and his level is up to 2.

In this case, Not every customer has a level. Only customer who has a member account has a level. If people buy products via guest account, he will not get point in his account.

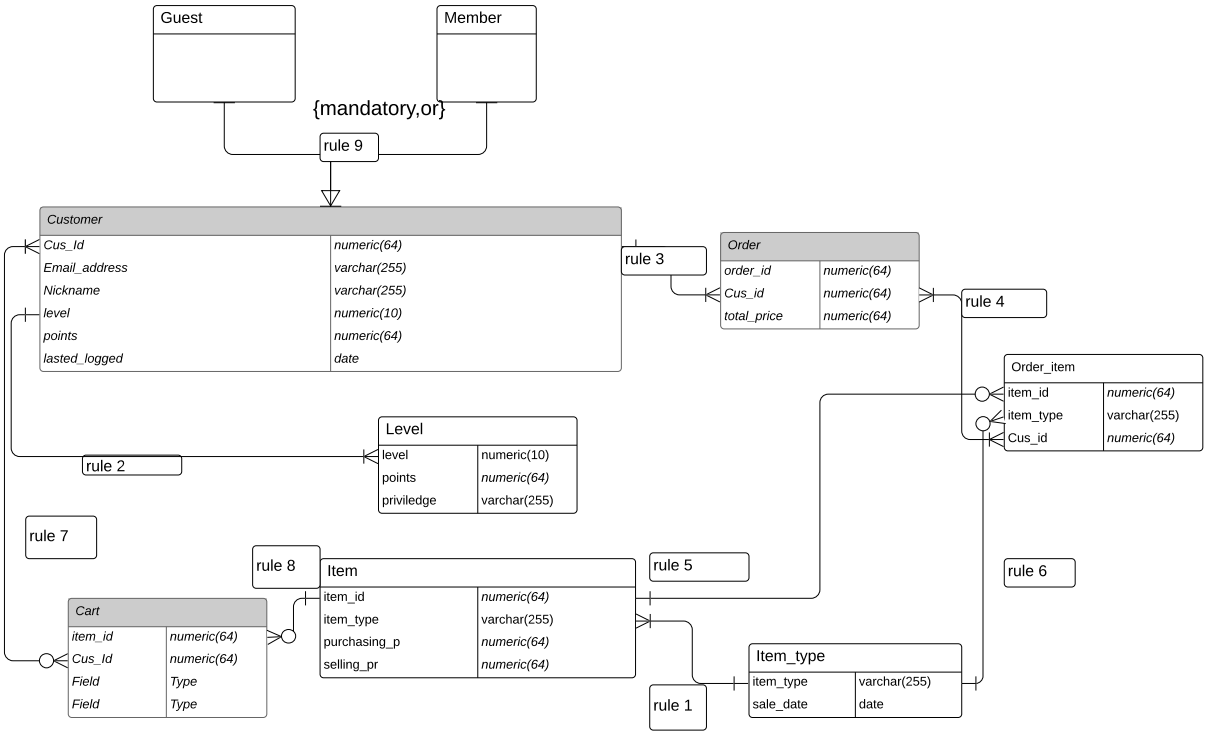
Our previous structural rule 2 is *Every customer has a level, different levels will have many customers.* Now, I change this rule: *Every member customer has a level, different levels will have many member customers. Guest account does not have a level.It is NULL.*

Now, I have 9 structural rules: (I add 9th rule and change the 2nd rule)

1. *Each item belongs to a type, and each type owns many items.*
2. *Every member customer has a level, different levels will have many member customers. Guest account does not have a level.*
3. *Each customer can have many orders, and each order can only belong to one customer.*
4. *Each order can include one or more items, and each item can belong to one or more orders.*
5. *Each item_id in Item table corresponds to no items or more purchased records in Order_item table, and each purchased record in Order_item table corresponds to one item_id in Item table.*

6. *Each item_type in Item_type table corresponds to no item or more purchased records in Order_item table, and each purchased record in Order_item table corresponds to one item_type in Item_type table.*
7. *Each customer puts no item or one or many items in cart and each item in cart belongs to a customer.*
8. *Each item_id in Item table corresponds to one or more records in Cart table, and each record in Cart table corresponds to one item_id in Item table.*
9. *An account is a member account or a guest account.*

I then add on to my ERD to support these new structural database rules.



Discount Given Relationship Classification and Associative Mapping

The Customer/Order relationship is 1:M. Every customer can order many times, but each order can be ordered by only one customer.

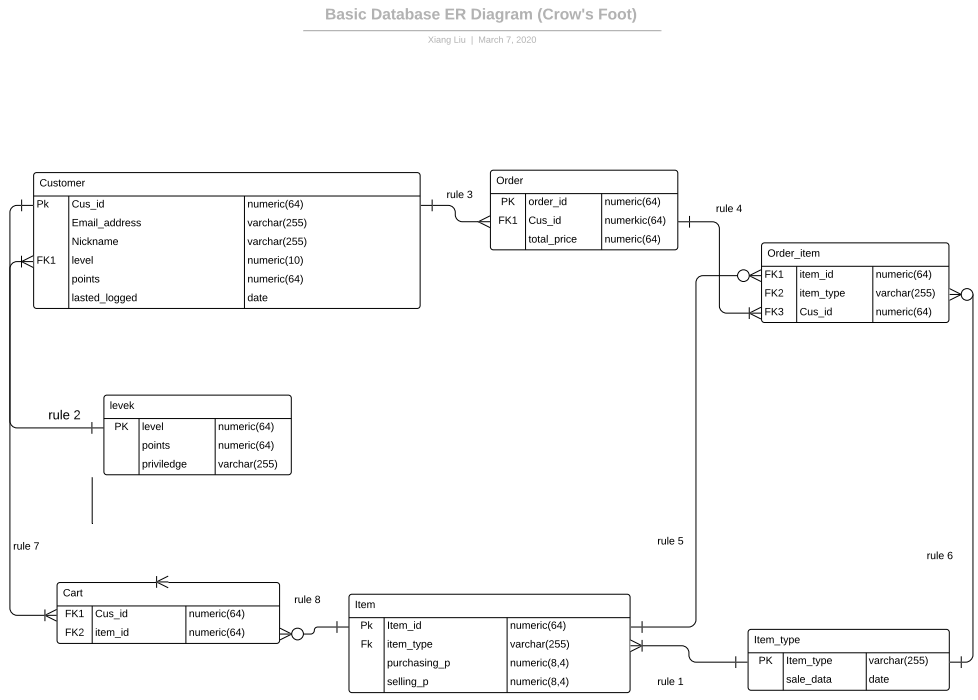
The Customer/Level relationship is 1:M. Every customer has a level, but every level has many customers.

The Order/Item relationship is M:N. Every order contains many items, and every item can be ordered many times. I added a third table Order_item. The Order/Order_item relationship is 1:M, the Item/Order_item relationship is 1:M.
The Item/Item_type relationship is 1:M. Every item belongs to a type, and every type contains many items.

The Customer/Item relationship is M:N. Every customers' cart has many items. And every item can be added in many customers' cart. So I added a third table Cart. The Customer/Cart relationship is 1:M, and the Item/Cart relationship is 1:M.

The Item_type/Order_item relationship is 1:M. Each item type's item can be bought many times, but each ordered item belongs to only one type.

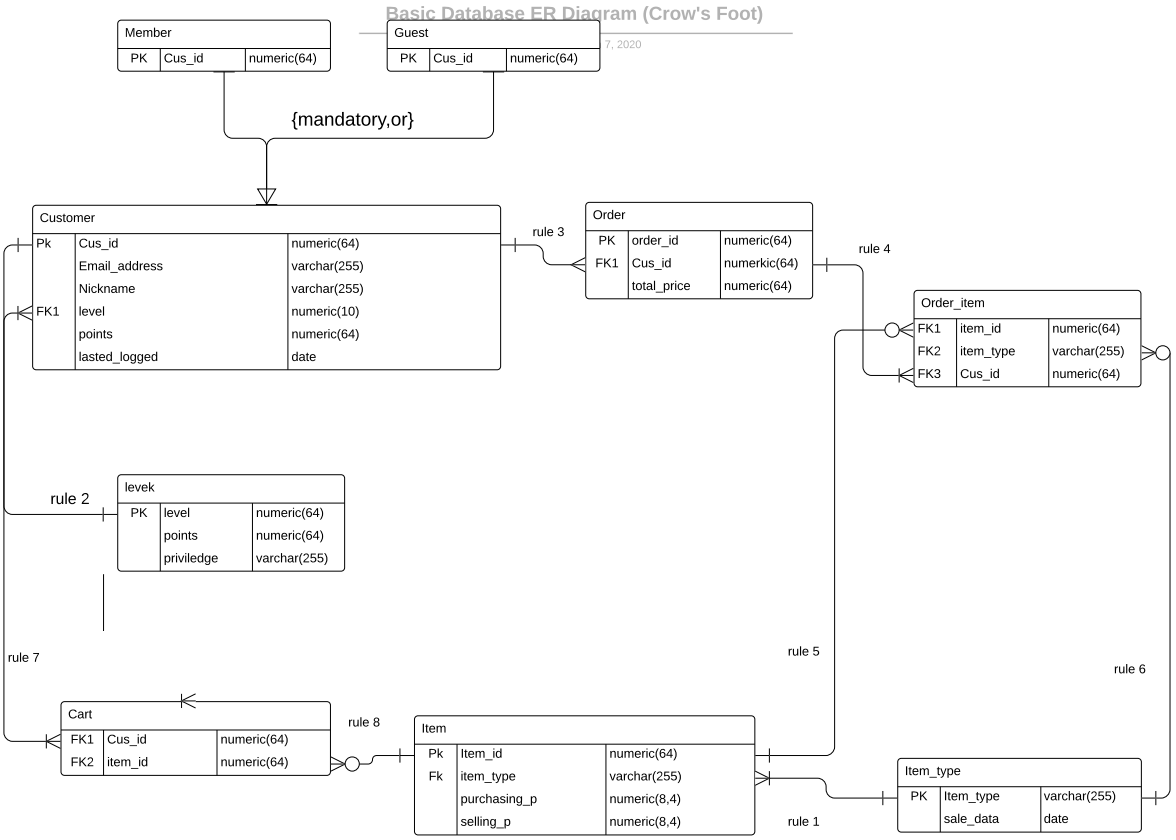
Here is my new physical ERD:



In physical ERD, the M:N relationship should have three table.

Discount Given Specialization-Generalization Mapping

I have one specialization-generalization mapping: the customer entity
Here is my DBMS physical ERD with these relationships mapped into them.

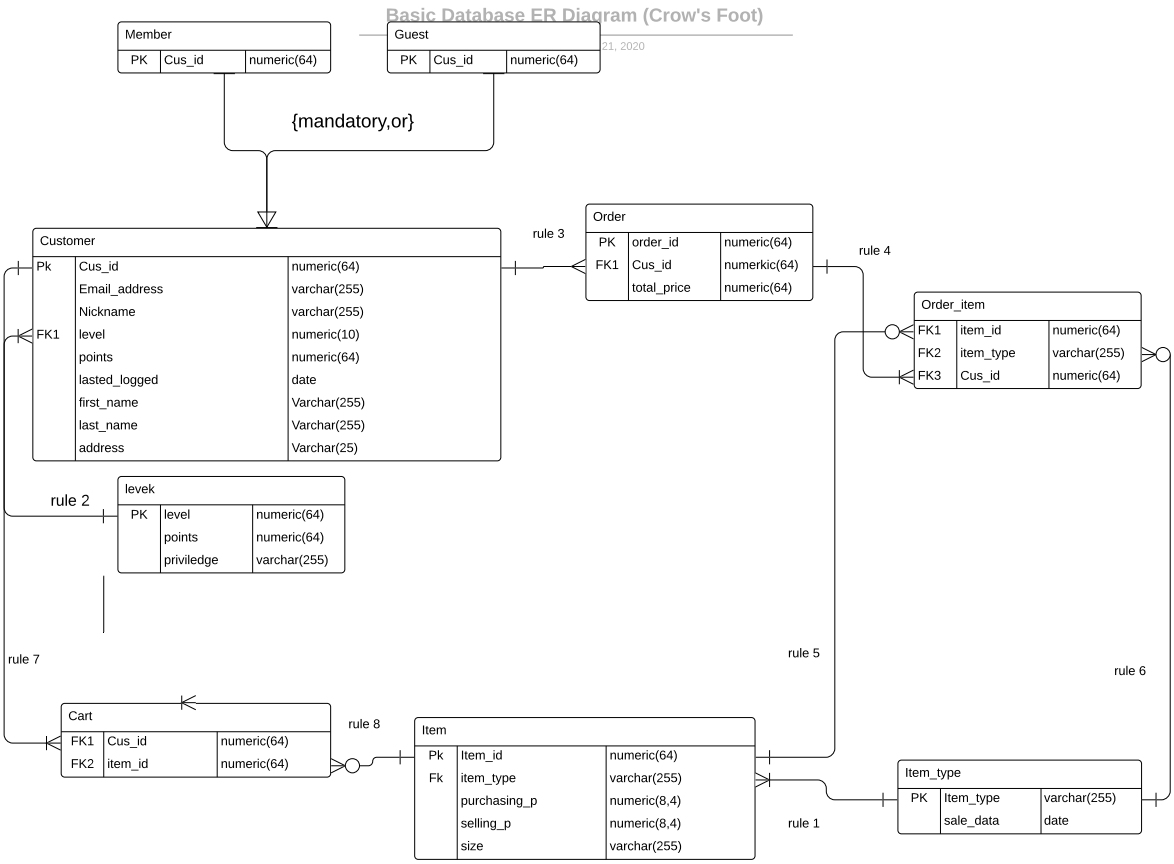


Discount Given Attributes

My database has already had some attributes. I search the similar clothing online store and find some extra attributes that I should add into my database.

Table	Attribute	Datatype	Reason
Customer	first_name	Varchar(255)	This is the first name of the account
Customer	last_name	Varchar(255)	This is the last name of the account
Customer	address	Varchar(255)	This is the customer's deliver address.
Item	size	Varchar(255)	Customers should select the clothing's size

Here is my ERD with the attributes included.

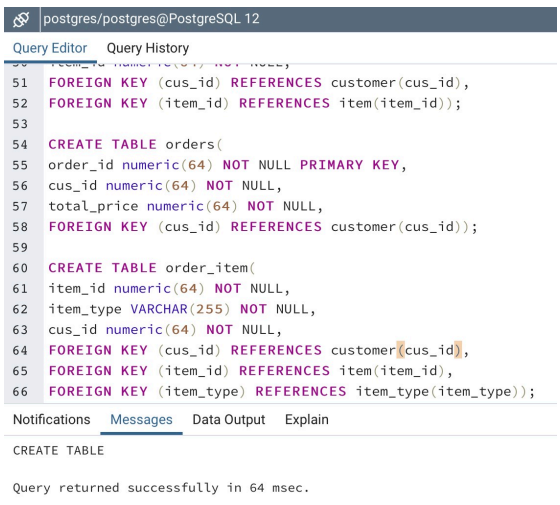


Discount Given Normalization

For Given, I have 9 entities. I think I don't have redundancy in my physical ERD.

Discount Given Create Table Script

In this step, I create the tables along with its subtypes. The screenshot below contains the script along with results of its execution. Note that I am using Postgre SQL for Discount Given.



I put my DROP TABLE commands at the top so that the script is re-runnable, then followed with the CREATE TABLE commands. All columns and constraints are included as illustrated in the ERD.

Discount Given Indexing

As far as primary keys which are already indexed, here is the list.

levels.levels
customer.cus_id
member.cus_id
guest.cus_id
Order.order_id
item.item_id
item_type.item_type

As far as foreign keys, I know all of them need an index. Below is a table identifying each foreign key column, whether or not the index should be unique or not, and why.

Column	Unique?	Description
Customer.levels	Not Unique	when we want to find level 4's customer, we should find all of them. There are many customers are in the same level
order.cus_id	Not Unique	Each customer can order many times
order_item.item_id	Not Unique	Each order can contains numbers of same item
order_item.item_type	Not Unique	Each order's items can be the same type
order_item.cus_id	Unique	The specific order can only be purchased by one user
Item.item_type	Not Unique	Many items belong to one type
cart.cus_id	Not Unique	Customer can have many items in their cart, and each item should be save as a row in cart table. So the customer id can be save many times
Cart.item_id	NOT unique	Customer can have many items in their cart, and each item should be save as a row in cart table. So each item can be put in the cart many times.

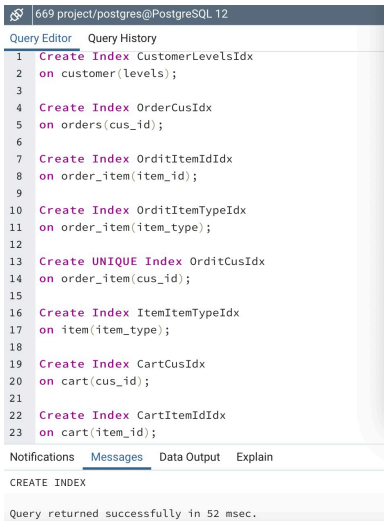
As far as the three query driven indexes, I spotted three fairly easily by predicting what columns will commonly be queried
1) customer.lasted_logged NOT UNIQUE
This attribute is used to find people who haven't logged in the website for one month. I will sent discount to these customer to stimulate their purchase. As index, it can be unique. Because many customers' lasted logged in time are same.

2) Item_type.sale_date NOT UNIQUE
In my project, I can send discount on specific day. For example, on Women's day, it is sale date for dress and skirt. I can sent discount on this day. If I create index for sale date, I can select the discount item type more quick. Some types may sale on the same day, so the index is not unique.

3) Customer.points NOT UNIQUE
When customers' points reach to some value, we can sent them some discounts. It is not unique, because many customers may have the same points.

Discount Given Index Creations

I create all the foreign keys' index as follows:



I named the index "CustomerLevelsIdx" to help identify what it's for, and placed the non-unique index on the customer table in levels.

I named the index "OrderCusIdx" to help identify what it's for, and placed the non-unique index on the orders table in cus_id.

I named the index "OrditItemIdIdx" to help identify what it's for, and placed the non-unique index on the order_item table in item_id.

I named the index "OrditItemTypeIdx" to help identify what it's for, and placed the non-unique index on the order_item table in item_type.

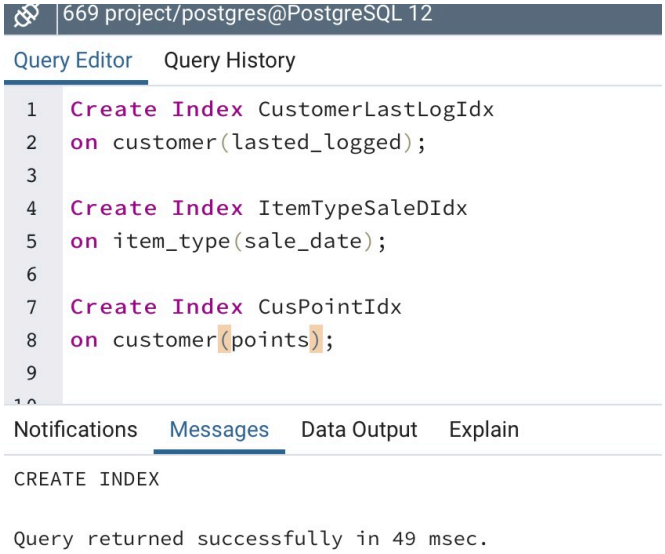
I named the index "OrditCusIdx" to help identify what it's for, and placed the unique index on the order_item table in cus_id.

I named the index “ItemItemTypeIdx” to help identify what it’s for, and placed the non-unique index on the item table in item_type.

I named the index “CartCusIdx” to help identify what it’s for, and placed the non-unique index on the cart table in cus_id.

I named the index “CartItemIdIdx” to help identify what it’s for, and placed the non-unique index on the cart table in item_id.

For the query-driven index:



I named the index “CustomerLastLogIdx” to help identify what it’s for, and placed the non-unique index on the customer table in lasted_logged.

I named the index “ItemTypeSaleDIdx” to help identify what it’s for, and placed the non-unique index on the item_type table in sale_date.

I named the index “CusPointIdx” to help identify what it’s for, and placed the non-unique index on the Customer table in points.

Given Discount Transaction

Case 3: Level defined Case

1. The customer buys items on the website and for every one dollars, he will get one point
2. At the beginning, the customer’s level is 1 with 0 point. If he bought 50 dollars in the website, he will get 50 points and his level is up to 2.

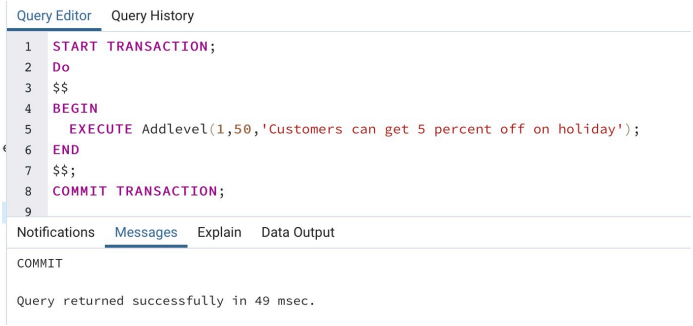
First, we should implement the case 3, because if we want to implement case 1, we need a level table at first.

For this case, I will implement a transaction that create level. In this step. We need one table: level.

Here is the screen shot:

I create Addlevel function to add level in levels table.

Here is the screen shot of procedure execution:



I insert the level 1 in the levels table

Case 1: Customers' Information Case (New)

1. The customer visited the online store.
2. The customer fill in a e-form to create an account by using his e-mail address or just login as a guest.
3. The member customer's level is 1 with 0 point. The guest customer does not have a level.
4. The member customer buy some items and 1 dollars is 1 point. The guest customer does not get any point.

For this case, I will implement a transaction that create a member account. In this step. We need 2 tables: account, Member

Here is a screen shot of postgres:

I create a function Addmembercustomer. I use now() function to get the current date. Since a new account's level is 1 and points in zero, I hardcode the level and points.

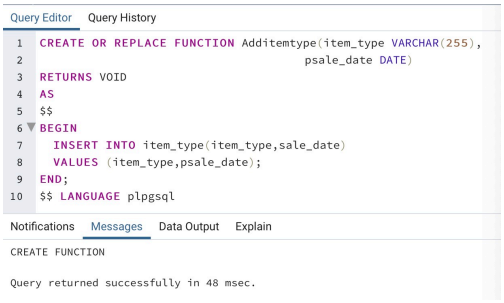
Here is the screen shot of procedure execution:

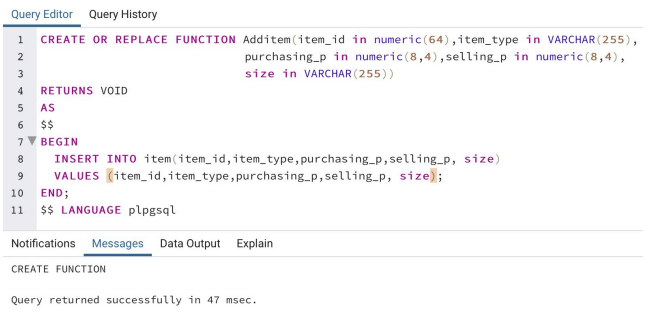
Case 2: Item Case

1. The database should also hold information about all products. We need an **Item** table.
2. Each product belongs to a different type.

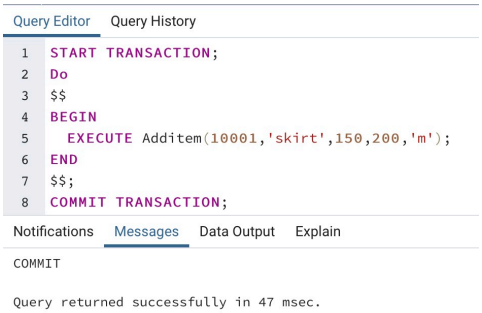
For this case, I will implement a transaction that insert item. In this step. We need 2 tables: itemtype and item.

Here are the screen of function additemtype and function additem:





Here are the screen shot of procedure execution:

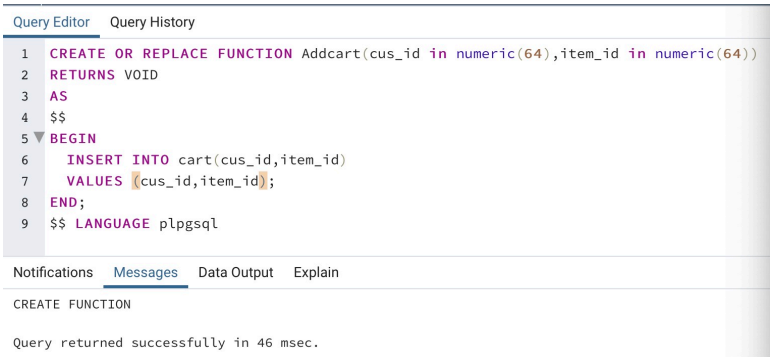


Case 5: Add Shopping Cart Case

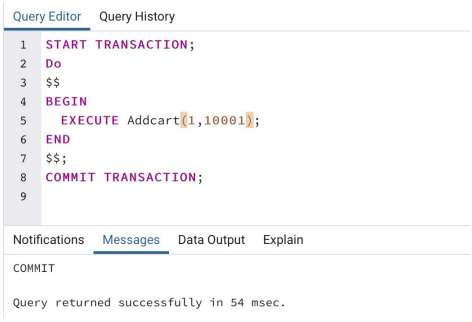
1. In addition to buying items, customers often put the desired items in the shopping cart. So it's easier to promote customers' consumption by sending corresponding coupons according to the goods in the shopping cart. We need a **Cart** table to store what customers want to buy.
2. If a customer put a dress in her shopping cart. On June 1, the discount day of dresses, I can use Item table to select the item_id which belongs to type 'dress'. Then in Cart table, I can know customers who put these item in their cart. Finally, in Customer table, I can know their email address.

For this case, I will implement a transaction that insert cart. In this step. We need one table: cart.

Here is the screen of function addcart;



Here is the screen shot of procedure execution:



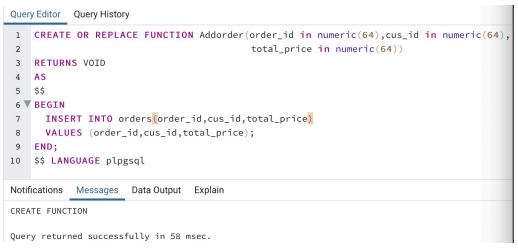
Case 4: Customer Purchase Case

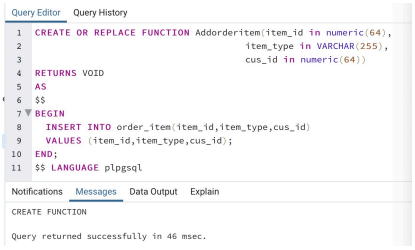
1. The system needs an **Order** table to record every order generated on the website. At the same time, record the total prince of each order in this table. By using the total price, the system can calculate the point and decide the level.

2. One or more items may be purchased for each order, so we need a table **Order_item** to record the items purchased for each order.
3. According to the type in table Order_item, we can find the customers who buy this type of item. On the discount day of the specified type of product, the company can give these customers this type of discount. For example, in Item_type table, we find that June 1 is the discount day of dresses every year. Then, we find all customers who have bought dresses in table Order_item, and then get the email address of these customers according to Table Customer, and send them the discount.

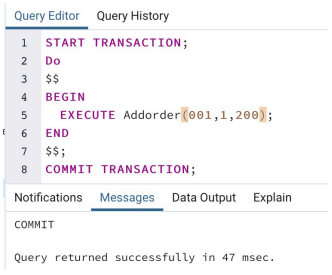
For this case, I will implement a transaction that insert order. In this step. We need two tables: order and order_item.

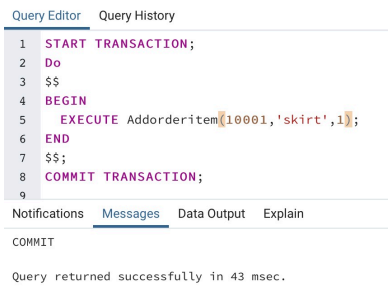
Here are the screen of function addorder an addorderitem;





Here are the screen shot of procedure execution:

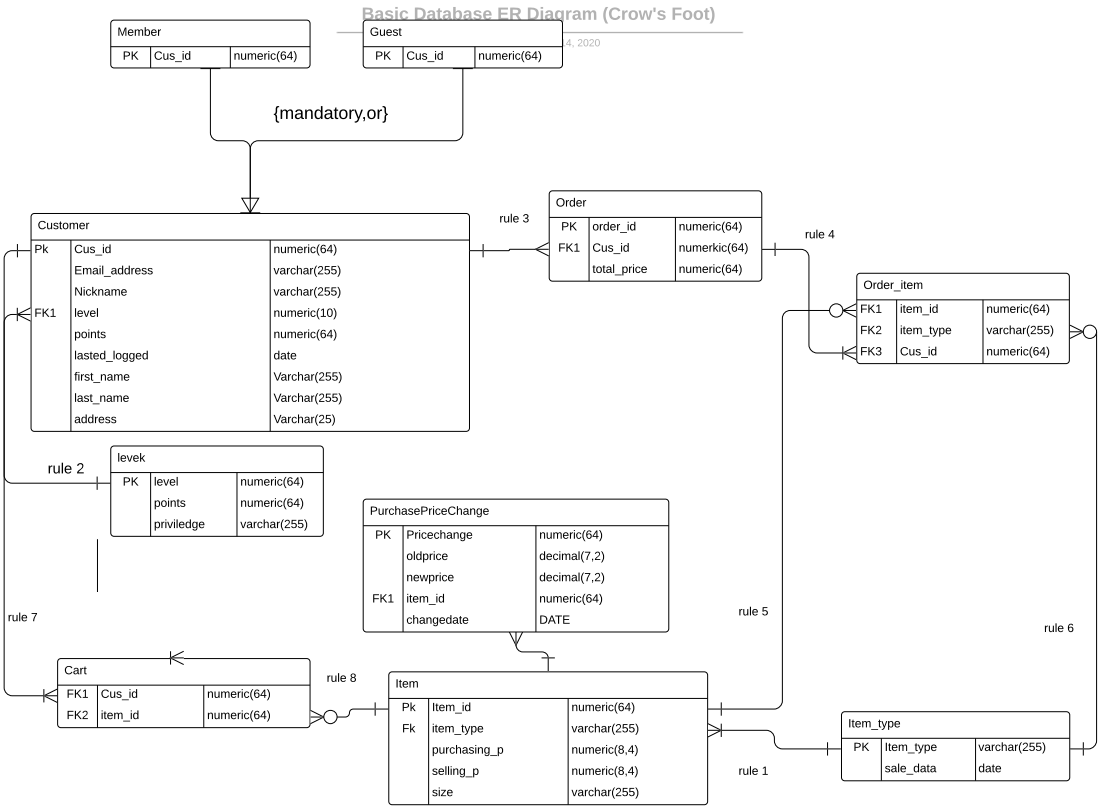




Discount Given History

In reviewing my DBMS physical ERD, one piece of data that would obviously benefit from a historical record an item's purchase price in the Item table. Such a history would help me know the change of the purchase. When the purchase price is decreasing, we can lower the selling price by giving customer a specific discount. First, my new structural database rule is: Each item may have many purchase price; each purchase price is for an item.

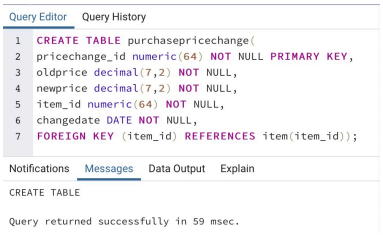
I added the PurchasePriceChange entity and related it to Item. My updated DBMS physical ERD is below.



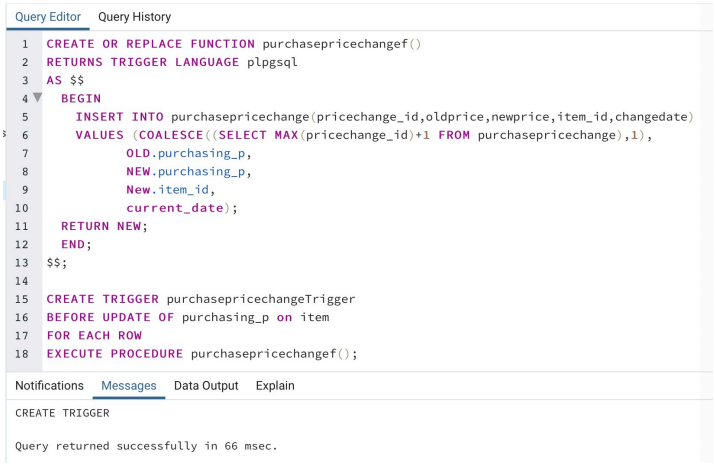
The purchasepricechange entity is present and linked to item. Below are the attributes I added and why.

Attribute	Description
Pricechange_id	This is the primary key of the history table. It is a numeric(64) to allow for many values.
oldprice	This is the old price before the change.
newprice	This is the new price before the change.
item_id	This is a foreign key to the item table, a reference to the item that had the change purchase price.
Change date	This is the date the balance change occurred, with a DATE datatype.

Here is a screenshot of my table creation, which has all of the same attributes and datatypes as indicated in the DBMS physical ERD.



Here is a screenshot of my trigger creation which will maintain the purchasepricechange table.



```
Query Editor  Query History
1  CREATE OR REPLACE FUNCTION purchasepricechangef()
2  RETURNS TRIGGER LANGUAGE plpgsql
3  AS $$
4    BEGIN
5      INSERT INTO purchasepricechange(pricechange_id,oldprice,newprice,item_id,changedate)
6      VALUES (COALESCE((SELECT MAX(pricechange_id)+1 FROM purchasepricechange),1),
7             OLD.purchasing_p,
8             NEW.purchasing_p,
9             New.item_id,
10            current_date);
11   RETURN NEW;
12   END;
13 $$;
14
15 CREATE TRIGGER purchasepricechangeTrigger
16 BEFORE UPDATE OF purchasing_p on item
17 FOR EACH ROW
18 EXECUTE PROCEDURE purchasepricechangef();

Notifications  Messages  Data Output  Explain

CREATE TRIGGER

Query returned successfully in 66 msec.
```

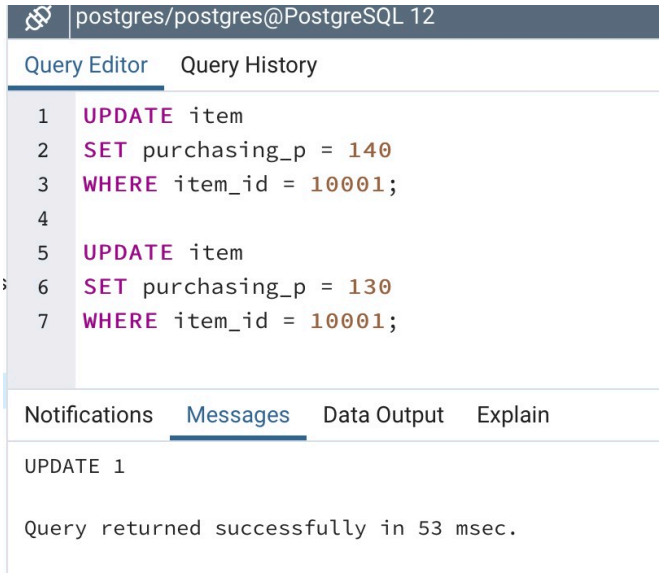
I start by ensuring there is a price created. In this case, it has an item_id of 1 with a purchase price of 150. as illustrated by the screenshot below.

```
Query Editor  Query History
1  SELECT *
2  FROM item

Notifications  Messages  Data Output  Explain
```

	item_id [PK] numeric (64)	item_type character varying (255)	purchasing_p numeric (8,4)	selling_p numeric (8,4)	size character varying (255)
1	10001	skirt	150.0000	200.0000	m

Next, I update the purchase price a couple of times, once to 140, and again to 130.



```
postgres/postgres@PostgreSQL 12
Query Editor  Query History
1  UPDATE item
2  SET purchasing_p = 140
3  WHERE item_id = 10001;
4
5  UPDATE item
6  SET purchasing_p = 130
7  WHERE item_id = 10001;

Notifications  Messages  Data Output  Explain

UPDATE 1

Query returned successfully in 53 msec.
```

Last, I verify that the purchasepricechange table has a record of these purchase price changes in the screenshot below.

Query Editor | Query History

```
SELECT *
FROM purchasepricechange
```

Notifications | Messages | Data Output | Explain

	pricechange_id [PK] numeric (64)	oldprice numeric (7,2)	newprice numeric (7,2)	item_id numeric (64)	changedate date
1	1	150.00	140.00	10001	2020-04-14
2	2	140.00	130.00	10001	2020-04-14

Discount Given Question and Query

Question1

- When the purchase price of item 10001 from suppliers' is reduced, we can give the customer a discount. Now we need to use history table to see whether the item's price reduced.

First, Here is what the purchasepricechange table looks like after some changes.

Query Editor | Query History

```
SELECT *
FROM purchasepricechange
```

Notifications | Messages | Explain | Data Output

	pricechange_id [PK] numeric (64)	oldprice numeric (7,2)	newprice numeric (7,2)	item_id numeric (64)	changedate date
1	1	150.00	140.00	10001	2020-04-14
2	2	140.00	130.00	10001	2020-04-14
3	3	100.00	110.00	10002	2020-04-14
4	4	300.00	310.00	10003	2020-04-14

We can see the item 10001 has changed its purchasing price twice.

Now, I use SQL to query.

Query Editor

```
SELECT sum(newprice - oldprice)as pricechange
FROM purchasepricechange
Where item_id = 10001
```

Notifications | Messages | Explain | Data Output

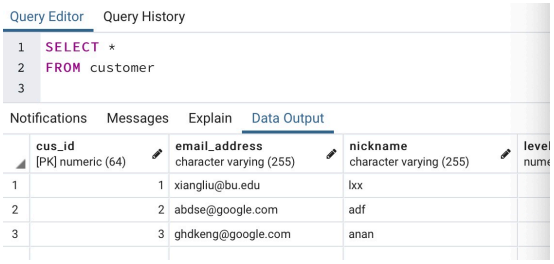
	pricechange numeric
1	-20.00

We can see the item 10001's purchase price totally reduced 20 dollars.

Question 2

Customers put their goods in their shopping cart, and when the purchase price of the goods in their cart decreases, they can receive a discount. Now I want to know which customers have lower purchase prices for goods in their shopping carts and get their email.

First, we can see the customer's list and their cart.

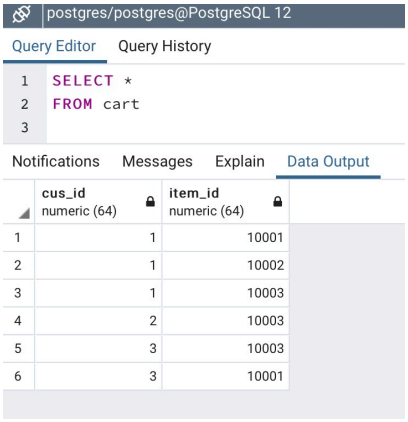


Query Editor | Query History

```
SELECT *
FROM customer
```

Notifications | Messages | Explain | Data Output

	cus_id [PK] numeric (64)	email_address character varying (255)	nickname character varying (255)	level
1	1	xiangliu@bu.edu	lxx	
2	2	abdse@google.com	adf	
3	3	ghdkeng@google.com	anan	



postgres/postgres@PostgreSQL 12

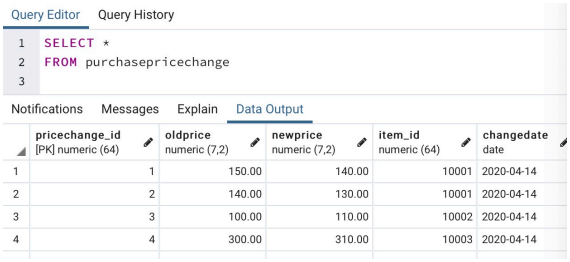
Query Editor | Query History

```
SELECT *
FROM cart
```

Notifications | Messages | Explain | Data Output

	cus_id numeric (64)	item_id numeric (64)
1	1	10001
2	1	10002
3	1	10003
4	2	10003
5	3	10003
6	3	10001

Then, we can see which item's purchase price is decreasing. Item 10001 and 10003's purchase price are decreasing.



Query Editor | Query History

```
SELECT *
FROM purchasepricechange
```

Notifications | Messages | Explain | Data Output

	pricechange_id [PK] numeric (64)	oldprice numeric (7,2)	newprice numeric (7,2)	item_id numeric (64)	changedate date
1	1	150.00	140.00	10001	2020-04-14
2	2	140.00	130.00	10001	2020-04-14
3	3	100.00	110.00	10002	2020-04-14
4	4	300.00	310.00	10003	2020-04-14

Thus, by looking at these three tables, we know in customer 1 and customer 2's cart, there has item's purchasing price is decreasing. But if we have many customers and items in the cart, by observing we can't get this conclusion. So we need SQL query

Here is the screen shot of my query:

(The construct I used are left join(group 2), subquery(group 2), Where clause(group 1))

postgres/postgres@PostgreSQL 12

Query Editor | Query History

```
1  SELECT customer.cus_id,email_address
2  FROM customer
3  LEFT JOIN cart ON customer.cus_id = cart.cus_id
4  WHERE cart.item_id  in (SELECT item_id
5                          FROM purchasepricechange
6                           where (newprice - oldprice) < 0)
7
```

Notifications | Messages | Explain | Data Output

	cus_id [PK] numeric (64)	email_address character varying (255)
1	1	xiangliu@bu.edu
2	3	ghdkeng@google.com

Question 3

As the weather gets hotter, some types of items need to be discounted. Now we want to know which types of goods are on sale in May. And find out the customers and their email address with these items in the shopping cart.

First, we can find out which types are on sale in May:

Query Editor | Query History

```
1  SELECT *
2  FROM item_type
3
```

Notifications | Messages | Explain | Data Output

	item_type [PK] character varying (255)	sale_date date
1	skirt	2020-06-01
2	jacket	2020-05-01
3	coat	2020-05-13

We can see the jacket and coat will be on sale in May.

Then, lets see what items belongs to jacket and coat.

Query Editor | Query History

```
1  SELECT *
2  FROM item
3
```

Notifications | Messages | Explain | Data Output

	item_id [PK] numeric (64)	item_type character varying (255)	purchasing_p numeric (8,4)
1	10001	skirt	130.000
2	10002	skirt	110.000
3	10003	skirt	310.000
4	10004	coat	300.000
5	10005	jacket	310.000
6	10006	coat	120.000

The item 10004, 10005, 10006 belong to jacket and coat.

Next, we see who's cart has these item

Query Editor | Query History

```
1  SELECT *
2  FROM cart
3
```

Notifications | Messages | Explain | Data Out

	cus_id numeric (64)	item_id numeric (64)
1	1	10001
2	1	10002
3	1	10003
4	2	10003
5	3	10003
6	3	10001
7	1	10004
8	2	10005
9	2	10006

We found customer 1 and customer 2's cart have these item.
So the result will found customer 1 and customer 2's email address. But in real world, we have many data in the database, we cannot get the result directly. So we need SQL query.

Here is my query in SQL:

Query Editor | Query History

```
1  SELECT customer.cus_id,email_address
2  FROM customer
3  LEFT JOIN cart ON customer.cus_id = cart.cus_id
4  WHERE cart.item_id  in (SELECT item_id
5                          FROM item
6                          WHERE item_type IN (SELECT item_type
7                                              FROM item_type
8                                              WHERE EXTRACT(MONTH FROM sale_date)=05
9                                              AND EXTRACT(YEAR FROM sale_date)=2020))
10 GROUP BY customer.cus_id
```

Notifications | Messages | Explain | Data Output

	cus_id [PK] numeric (64)	email_address character varying (255)
1	1	xiangliu@bu.edu
2	2	abdse@google.com

Summary and Reflection

My database is for staff to send discounts to customers. The staff can entering the keywords to find the right customers. My database is designed for an online clothing shopping mall.

The structural database rules and ERD for my database design contain the important entities of Customer, Item, Item_type, Cart, Order, Order_item and Level as well as relationships between them. How to ensure the relationships between entities is difficult in this iteration. I have 8 structural rules now. It's a little complicated, but each one is useful.

I now have an initial DBMS physical ERD that includes entities, relationships, and primary and foreign key constraints. I have modeled both associative and specialization-generalization relationships which represent a complete picture of relationships for modern database design.

The SQL script that creates all tables follows the specification from the DBMS physical ERD exactly. Important indexes have been created to help speed up access to my database and are also available in an index script. Stored procedures have been created and executed transactionally to populate some of my database with data. Some questions useful to Discount Given have been identified, and implemented with SQL queries. After the course is over, I should continue to tweak and expand my database to meet the needs of application.