

‘BestBuy’ Project Direction Overview

I want to develop an application called 'BestBuy', its purpose is that when someone needs to buy something (especially a lot of items), he may refer to a shopping site (such as 'Walmart', 'Amazon' and so on), may refer to many shopping websites, and the same shopping website also has a large number of similar items to refer to, maybe he will find some items he thinks well in the process, and hope to compare the details to find the best among them after looks many items. At this time, they need to keep a large number of web pages and watch them one by one for comparison, which will cause many web pages to exist above the browser, and the basic information also needs to be memorized by themselves. It's easy to feel complicated.

'BestBuy' allows users to provide a 'storage' option when they see a good choice, and can finally categorize them in a unified manner, and put the same type of items sort by select (such as 'price', 'rating') , so that the user can more easily distinguish the better ones.

Here is are some brief examples of how someone would use the application:

SiCheng has just moved to Boston, and he needs to buy furniture for his newly rented house, which includes bed frames, mattresses, chairs, and tables. He browses many items on Amazon and Walmart, and when he sees a good item, he chooses to store it on 'BestBuy'. After five hours, he was done browsing and was ready to make his final choice. He opened the 'BestBuy' interface, and sorted the 'bed frame' type products he stored by price, and then entered the detailed interface through the URL of each product to refer to the evaluation. In this way, he made a final comparison of the bed frame, mattress, chair, and table and purchased the most satisfactory product among them.

While obviously there will be a significant programming component for 'BestBuy', for this course I am focusing on the database component. The database of 'BestBuy' will be used by many people, so it needs to store each user's personal information, the database will need to associate the purchases with different accounts. 'BestBuy' will record each item's 'type', 'name', 'price', 'rating', 'sale platform', 'stored date' and so on. Some details It will be more clear in the coming weeks.

BestBuy Use Cases and Fields

The first important usage of the database is when a person signs up for an account and installs the application.

Account Signup/Installation Use Case

1. The person visits TrackMyBuys' website or app store and installs the application.
2. The application asks them to create an account when its first run.
3. The user enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them, and store the information when the user needs it.

From a database perspective, this use case requires storing information about accounts (from steps 2 and 3). Steps 1 and 4 apply to the user and application but not the database directly.

Significant fields for an account for this application are listed in the table below.

Field	What it Stores	Why it's Needed
UserName	The name displayed by the user in the program will be used for some sharing, ranking and so on.	Some user maybe have many account, and some people also don't want other people know their real name.
FirstName	This is the first name of the account holder.	This is necessary for displaying the person's name on screens and addressing them when sending them emails or other communications
LastName	This is the last name of the account holder.	This is necessary for displaying the person's name on screens and addressing them when sending them emails or other communications
CreatedDate	Date the account was created.	It is useful to stores the date io distinguish old and new users, it can be used for various activities.
Mail	Stores the e-mail of user.	Can be used to send various emails to users, this function can be turned off during registration.
PhoneNumber	Stores the phone number of user.	Use for contacting users.

Balance	Stores the balance of user.	Some features may require payment to use, and are therefore used to store the user's current balance.
---------	-----------------------------	---

The second important usage of the database is to use browser extensions to store product information that the user wants to store.

Save Store Product Information Use Case

1. The user goes to the product detail page and chooses to save.
2. BestBuy receives the order, and stores the product's selling platform, itemtype, name, price, evaluation, and URL in the database.

Significant fields are detailed below.

Field	What it Stores	Why it's Needed
SellingPlatform	The item's selling platform name	Let the user know which on-line shopping platform sell the item.
Type	The item's type.	Used to categorize products according to their type.
Name	The item's name.	Used to let users understand the name of the product.
Price	The item's price.	Let user know the price, also can use to sort.
evaluation	The item's evaluation	Let user know the evaluation, also can use to sort. Most of the current websites are on a five-point scale, and buyers' detailed reviews can be viewed by using the URL.
URL	The purchase URL of the item.	It is convenient for users to enter the detailed purchase interface to perform more operations.

The third important usage of the database will be used to recommend items to users.

Recommend Items Use Case

1. The user opens the 'BestBuy' program interface.
2. Select 'Product Recommendations'.
3. 'BestBuy' gives the user various conditions, such as brand, selling platform, price range, etc.
4. User choose various conditions what their want.
5. The application pulls all purchases matching the criteria from the database, sorted by the number of times the item has been stored by all users using 'BestBuy', and displayed to the user.
6. The user selects the item they are interested in.
7. The application extracts the product's storage information from the database and displays it to the user.
8. The user can store it in his own account, or make other choices.

This database will use the database which is save store product information. On this basis, some additional content will be added for recommended sorting.

Significant fields are detailed below.

Field	What it Stores	Why it's Needed
Savetime	Save how many user save this item.	To a certain extent, this can indicate how many users are interested in him, and users can be recommended according to this data.
Promote	Store how much promotion fee the merchant offers.	Merchants can promote the application by paying a promotion fee, which can be arranged according to the promotion fee.
SellingPlatform	The item's selling platform name	Let the user know which on-line shopping platform sell the item. Also used to provide the user with a search interval limit.
Type	The item's type.	Used to categorize products according to their type. Also used to provide the user with a search interval limit.

Name	The item's name.	Used to let users understand the name of the product. Also used to provide the user with a search interval limit.
Price	The item's price.	Let user know the price, also can use to sort. also used to provide the user with a search interval limit.
evaluation	The item's evaluation	Let user know the evaluation, also can use to sort. Most of the current websites are on a five-point scale, and buyers' detailed reviews can be viewed by using the URL. Also used to provide the user with a search interval limit.
URL	The purchase URL of the item.	It is convenient for users to enter the detailed purchase interface to perform more operations.

Structural Database Rules

From the step: 'Account Signup/Installation Use Case' we can find an entity-Account.

From the step: 'Save Store Product Information Use Case' we can find six significant data points: selling platform, type, name, price, evaluation, and URL.

We can get some structural rule:

- 1: Each save is associated with an account; Each account may be associated with one to many saves.
- 2: Each save is associated with an selling platform; Each platform associated with one to many saves.
- 3: Each save has one type; Each type has one to many saves.
- 4: Each save has one name; Each name associated with one to many saves.
- 5: Each save has one price; Each price has one to many saves.
- 6: Each save has one to many evaluations; Each evaluation be from one save.
- 7: Each save has one URL; Each URL associated with one save.

From the step: 'Recommend Items Use Case' we can add two significant data points: Savetime and Promote.

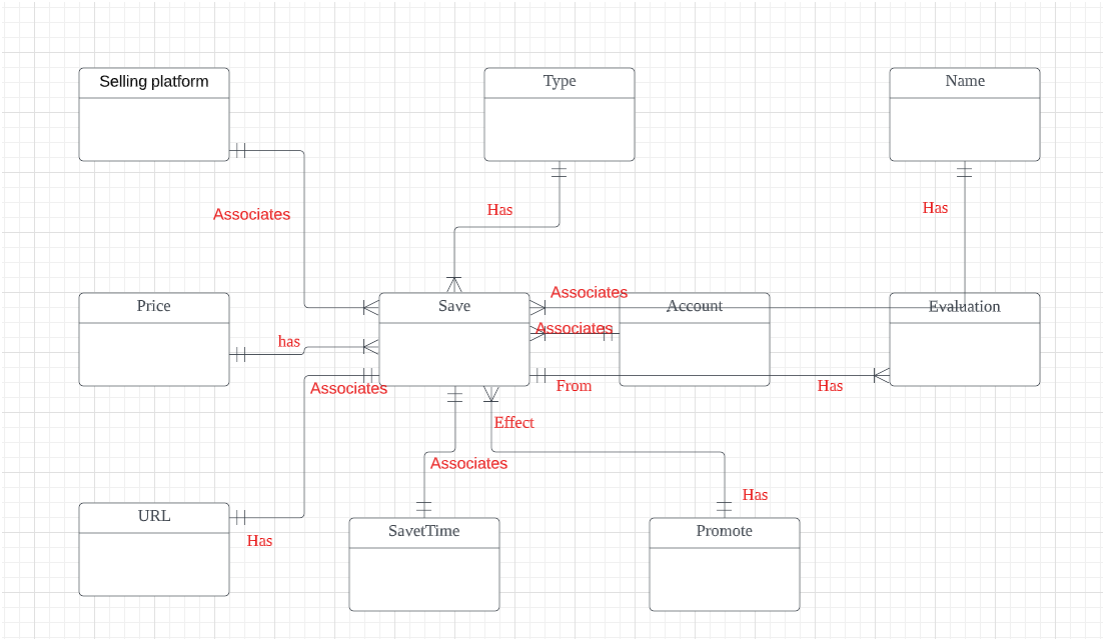
We can get some structural rule:

- 8: Each save associated with one savetime; Each savetime associated with one save.
- 9: Each save has one promote; Each promote effect one to many saves.

So from the three use cases I have thus far, I have nine structural database rules:

- 1: Each save is associated with an account; Each account may be associated with many saves.
- 2: Each save is made from one selling platform; Each platform has one to many saves.
- 3: Each save has one type; Each type has one to many saves.
- 4: Each save has one name; Each name has one to many saves.
- 5: Each save has one price; Each price has one to many saves.
- 6: Each save has one to many evaluations; Each evaluation be from one save.
- 7: Each save has one URL; Each URL appear one save.
- 8: Each save associated with one savetime; Each savetime associated with one save.
- 9: Each save has one promote; Each promote effect one to many saves.

Entity-relationship diagram



Adding Specialization-Generalization

By checking all my cases, after thinking, I judged that "Account Signup/Installation Use Case" can be modified.

Account Signup/Installation Use Case

1. The person visits TrackMyBuys' website or app store and installs the application.
2. The application asks them to create an account when its first run.
3. The user enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them, and store the information when the user needs it.

As expected, I will separate accounts into buyers and sellers, and I will offer a free account that has some limitations on the functionality, and a paid account which offers all the features. I modify the use case as follows:

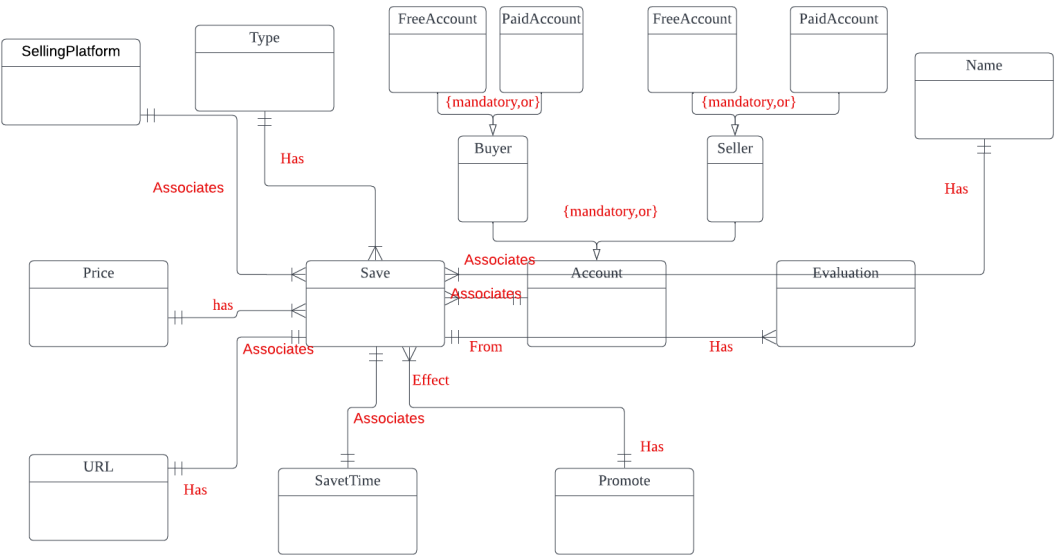
1. The person visits TrackMyBuys' website or app store and installs the application.
2. The application asks them to create a buyer or seller account and choose whether to pay when its first run.
3. The user enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them, and store the information when the user needs it.

According to the #2 change, I deduce two rules:

8. An account is a buyer account or a seller account.
9. An account is a free account or a paid account.

My database only has two kinds of accounts: buyer and seller, and that is the complete list. And each account has two relationships: free and paid. It's also a complete list, the relationship is totally complete. The relationship is disjoint.

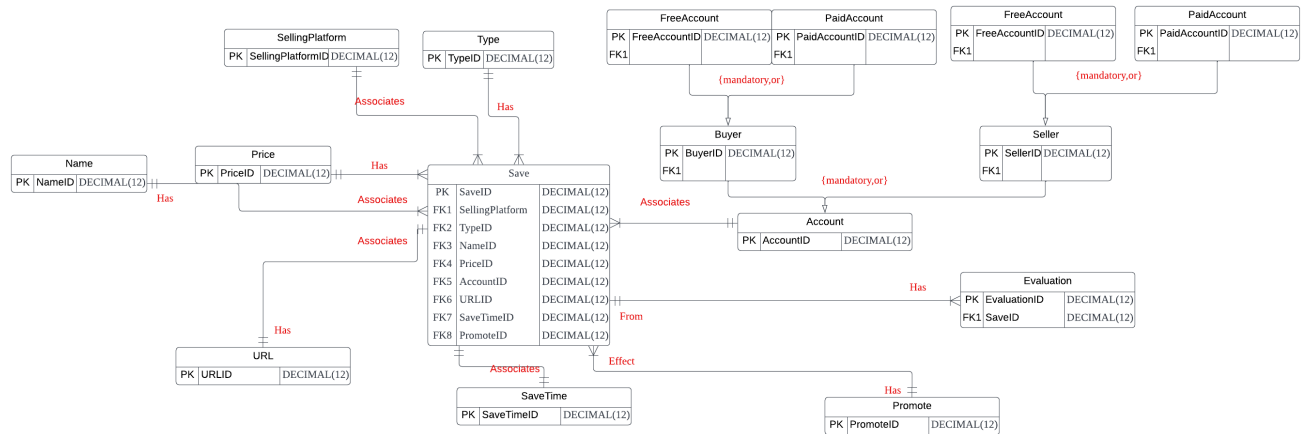
Here is the ERD:



Initial DBMS Physical ERD

I opted to use the conceptual ERD to identify the relationships. After observation, I found that my ERD is basically composed of 1:1 and 1:M, so it is relatively simple to convert it into a physical ERD, just add the corresponding PK and RK.

Here is the Physical ERD:



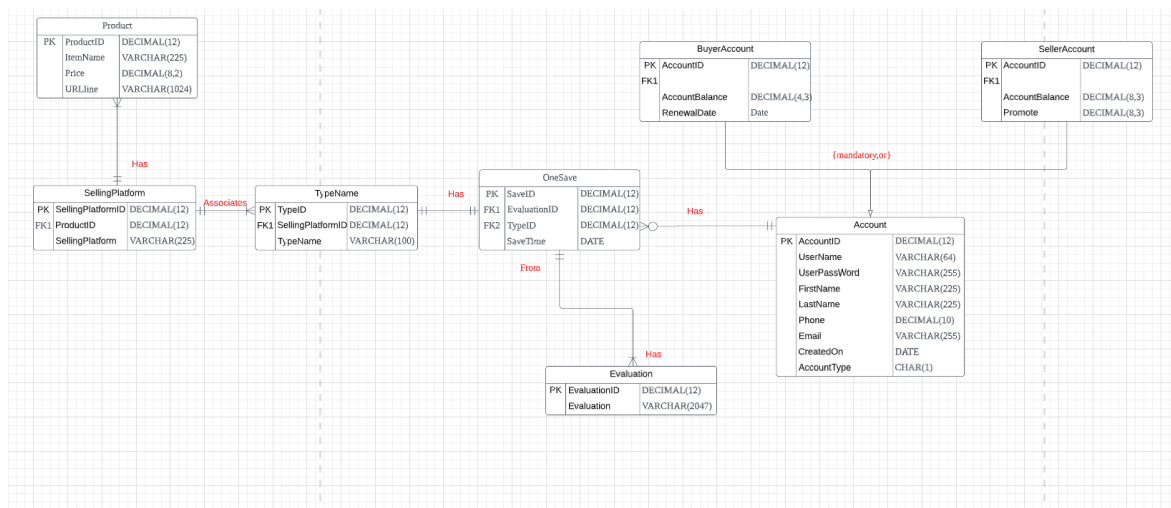
BestBuy Attributes

In order to make the structure clearer and reduce redundant consumption, I made major changes. I unified various product information into a new table: Product, including ItemName, Price, URL, and then Associate Product to each platform, and then to the type of product, Associate the Type with a table containing the data savetime as a storage. I no longer divide the account into paid and free, but simply separate it as Buyer and Seller, where Buyer can open a limited-time paid function by paying a small fee, and Seller can provide recommendations for their stored products by paying an amount. Here are the details:

Table	Attribute	Datatype	Reasoning
Product	ItemName	VARCHAR(225)	This is just the name of the item, so I don't think it need pretty long of the name, 225 is enough.
Product	Price	DECIMAL(8,2)	This is the price of the item, I think tens of millions of dollars is the limit for online shopping transactions, and many merchants like to add .99 to their prices, so I provide 2 decimal places.
SellingPlatform	SellingPlatform	VARCHAR(225)	This is the SellingPlatform name, just like the item name.
Type	Type	VARCHAR(100)	I think the type should not be pretty long, so the varchar(100) is enough.
Product	URL	VARCHAR(1024)	I noticed that some sites have very long URLs, so I designed a larger space.
Evaluation	Evaluation	VARCHAR(2047)	Many people will write a paragraph or even several paragraphs in some

			comments, so I give more space for users to write more content.
OneSave	SaveTime	DATE	This is the time of the date save on the database, so it is a DATE data.
Account	UserName	VARCHAR(64)	This is just the username of the account, not pretty long and short.
Account	UserPassword	VARCHAR(255)	I just set enough space to let the user can make a long password for safety.
Account	FirstName	VARCHAR(225)	I don't think the name of a parson will be pretty long, this is just for guarantee have enough space.
Account	LastName	VARHCAR(225)	I don't think the name of a parson will be pretty long, this is just for guarantee have enough space.
Account	Phone	DECIMAL(10)	I think 10 digits is enough to provide area codes + phone numbers for most countries.
Account	Email	VARCHAR(255)	Provide enough space for various email accounts.
Account	CreatedOn	DATE	A date variable representing the date the account was created.
Account	AccountType	CHAR(1)	A char used to distinguish buyer account from seller account, expected to

			be 0 and 1.
BuyerAccount	AccountBalance	DECIMAL(4,3)	Buyer users only need to pay a small fee to use paid features for a period of time, so I don't think anyone will save more than \$10,000.
BuyerAccount	RenewalDate	DATE	Expiration date for account paid features
SellerAccount	AccountBalance	DECIMAL(8,3)	Seller users are likely to pay more for their product to get a referral, so I offer a larger amount.
SellerAccount	Promote	DECIMAL(8,3)	This variable is meant to represent the amount that has been used



I noticed that after the adjustment, it seems to be in the canonical state, and I don't need to make any additional adjustments.

BestBuy CREATE TABLE

```
Termin4.sql - DES...DV7L8\ASUS (67) * X
DROP TABLE BuyerAccount
DROP TABLE SellerAccount
DROP TABLE Account
DROP TABLE OneSave
DROP TABLE TypeName
DROP TABLE SellingPlatform
DROP TABLE Evaluation
DROP TABLE Product

CREATE TABLE Account(
  AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
  UserName VARCHAR(64) NOT NULL,
  UserPassword VARCHAR(255) NOT NULL,
  FirstName VARCHAR(225) NOT NULL,
  LastName VARCHAR(225) NOT NULL,
  Phone DECIMAL(10) NOT NULL,
  Email VARCHAR(255) NOT NULL,
  CreateOn DATE NOT NULL,
  AccountType CHAR(1) NOT NULL);

CREATE TABLE BuyerAccount(
  AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
```

110 %

消息
命令已成功完成。
完成时间: 2022-11-06T10:52:53.7125845+08:00

BestBuy Indexing

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

Account.AccountID

BuyerAccount.AccountID

SellerAccount.AccountID

OneSave.SaveID

Evaluation.EvaluationID

TypeName.TypeID

SellingPlatform.SellingPlatformID

Product.ProductID

As far as foreign keys, all of it need to have an index:

Column	Unique	Description
OneSave.EvaluationID	Not unique	Not unique because one save can have many evaluation
OneSave.TypeID	Unique	Unique because one save can have only one type
TypeName.SellingPlatformID	Not unique	Not unique because one type can from many platform
SellingPlatform.ProductID	Not unique	Not unique because one platform can have many product

As far as the three query driven indexes, First I set is account balance for many people want to know their balance. Second is savetime because we often need to put together like the save on one day. Third I set the TypeName to make many people need the same type together.

```
/*Make the index*/  
/*Make the index of FK*/  
CREATE INDEX EvaluationIDIdx  
ON OneSave(EvaluationID);  
CREATE UNIQUE INDEX TypeIDIdx  
ON OneSave(TypeID);  
CREATE INDEX SellingPlatformIDIdx  
ON TypeName(SellingPlatformID);  
CREATE INDEX ProductIDIdx  
ON SellingPlatform(ProductID);  
CREATE INDEX BuyerAccountBalanceIdx  
ON BuyerAccount(AccountBalance);  
/*Make the index for account balance to make people know is it enough*/  
CREATE INDEX SellerAccountBalanceIdx  
ON SellerAccount(AccountBalance);  
/*Make the index of savetime to let people can make same time save together*/  
CREATE INDEX SaveTimeIdx  
ON OneSave(SaveTime);  
/*Make the index of type to let people can make same type product together*/  
CREATE INDEX TypeNameIdx  
ON TypeName(TypeName);
```

3 %

消息

命令已成功完成。

完成时间: 2022-11-06T11:45:01.2588098+08:00

BestBuy Transaction

The first use case for BestBuy is the account signup use case listed below.

Account Signup/Installation Use Case

1. The person visits TrackMyBuys' website or app store and installs the application.
2. The application asks them to create an account when its first run.
3. The user enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them, and store the information when the user needs it.

Here is a screenshot of my stored procedure definition.

```
/*Account Signup/Installation Use Case BuyerAccount*/
Create PROCEDURE AddBuyerAccount @AccountID DECIMAL(12), @UserName VARCHAR(64), @UserPassword VARCHAR(255), @FirstName VARCHAR(225),
@LastName VARCHAR(225), @Phone DECIMAL(10), @Email VARCHAR(255)
AS
BEGIN
    INSERT INTO Account(AccountID, UserName, UserPassword, FirstName, LastName, Phone, Email, CreatOn, AccountType)
    Values (@AccountID, @UserName, @UserPassword, @FirstName, @LastName, @Phone, @Email, GETDATE(), 'B');
    INSERT INTO BuyerAccount(AccountID, AccountBalance, RenewalDate)
    VALUES(@AccountID, 0, GETDATE());
END
GO
```

99 %

消息
命令已成功完成。

I make a stored procedure named 'AddBuyerAccount', It inserts data into the table, and the Account balance is 0 because it have no money in account when it create, and the CreateOn and RenewalDate is the time when the account create, the AccountType is 'B' for the buyer account.

Here is a screenshot of stored procedure execution.

```
BEGIN TRANSACTION AddBuyerAccount;
EXECUTE AddBuyerAccount 1, 'dgordn', 'xyz', 'Decker', 'Gordon', 8655368488, 'Decker.gordon@gmail.com';
COMMIT TRANSACTION AddFreeAccount;
```

99 %

消息

(1 行受影响)

(1 行受影响)

The second for BestBuy is the account signup use case listed below.

Save Store Product Information Use Case

3. The user goes to the product detail page and chooses to save.
4. BestBuy receives the order, and stores the product's selling platform, itemtype, name, price, evaluation, and URL in the database.

Here is a screenshot of my stored procedure definition.

```

/*Save Store Product Information Use Case BuyerAccount*/
CREATE PROCEDURE AddProduct @ProductID DECIMAL(12), @ItemName VARCHAR(225), @Price DECIMAL(8,2), @URLline VARCHAR(1024)
AS
BEGIN
    INSERT INTO Product(ProductID, ItemName, Price, URLline)
    Values (@ProductID, @ItemName, @Price, @URLline);
END;
GO

```

I make a stored procedure named 'AddProduct', It insert the product date to the table, just get date from insert.

Here is a screenshot of stored procedure execution.

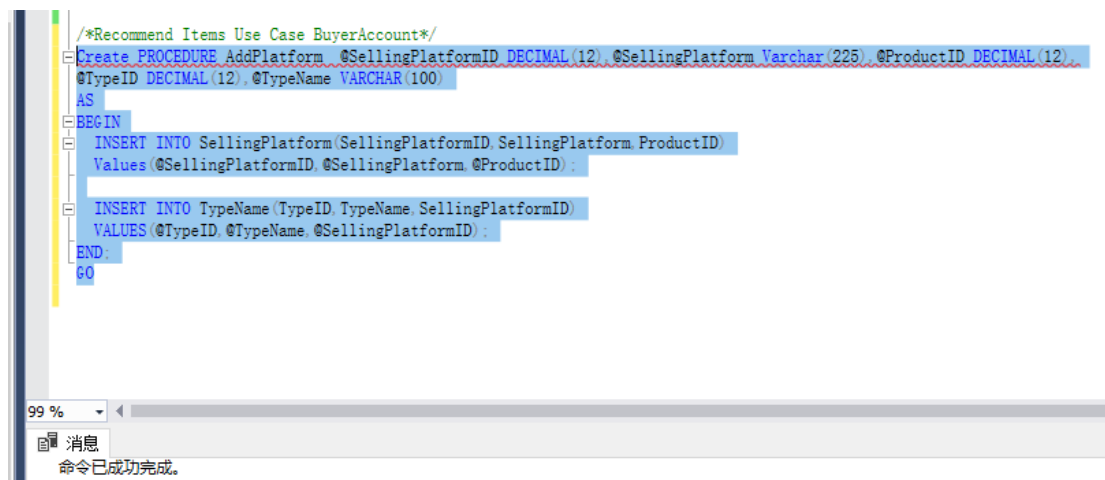
```
BEGIN TRANSACTION AddProduct;
EXECUTE AddProduct 1,'best Value Whole Vitamin D Milk Saloon 100 Fl oz','3.17','https://www.walmart.com/ip/best-value-whole-vitamin-d-milk-saloon-100-fl-oz/10420114?wcat=&P=10420114&sthash=c6e1f8e9&addtoCart=1&addtoCart=2&addtoCart=3&addtoCart=4';
COMMIT TRANSACTION AddProduct;
```

The Third for BestBuy is the account signup use case listed below.

Recommend Items Use Case

9. The user opens the 'BestBuy' program interface.
10. Select 'Product Recommendations'.
11. 'BestBuy' gives the user various conditions, such as brand, selling platform, price range, etc.
12. Users choose various conditions what their want.
13. The application pulls all purchases matching the criteria from the database, sorted by the number of times the item has been stored by all users using 'BestBuy', and displayed to the user.
14. The user selects the item they are interested in.
15. The application extracts the product's storage information from the database and displays it to the user.
16. The user can store it in his own account or make other choices.

Here is a screenshot of my stored procedure definition.



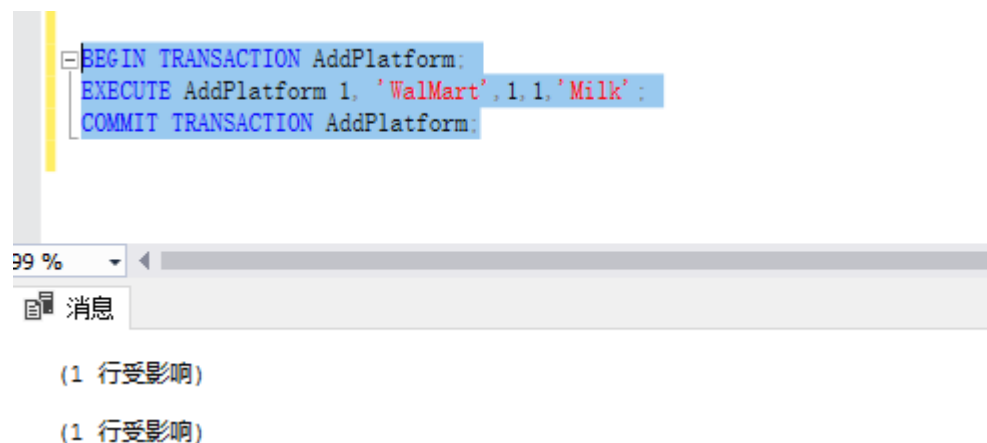
```
/*Recommend Items Use Case BuyerAccount*/
Create PROCEDURE AddPlatform @SellingPlatformID DECIMAL(12), @SellingPlatform Varchar(225), @ProductID DECIMAL(12),
@TypeID DECIMAL(12), @TypeName VARCHAR(100)
AS
BEGIN
    INSERT INTO SellingPlatform(SellingPlatformID, SellingPlatform, ProductID)
    Values(@SellingPlatformID, @SellingPlatform, @ProductID);
    INSERT INTO TypeName (TypeID, TypeName, SellingPlatformID)
    VALUES (@TypeID, @TypeName, @SellingPlatformID);
END;
GO
```

99 %

消息
命令已成功完成。

I make a stored procedure named 'AddPlatform', It is insert the data to table 'SellingPlatform' and 'TypeName'. Just get the data from the insert.

Here is a screenshot of stored procedure execution.



```
BEGIN TRANSACTION AddPlatform;
EXECUTE AddPlatform 1, 'WalMart', 1, 1, 'Milk';
COMMIT TRANSACTION AddPlatform;
```

99 %

消息

(1 行受影响)

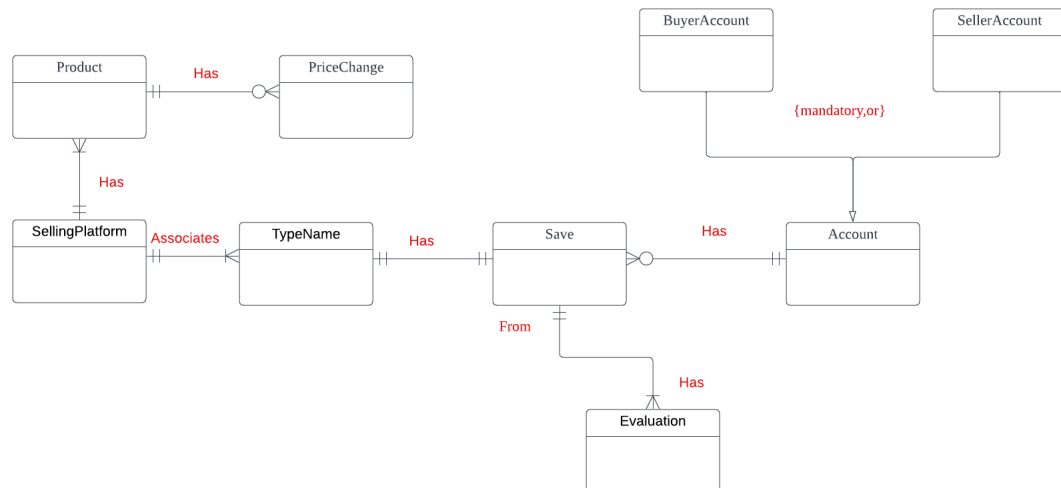
(1 行受影响)

BestBuy History

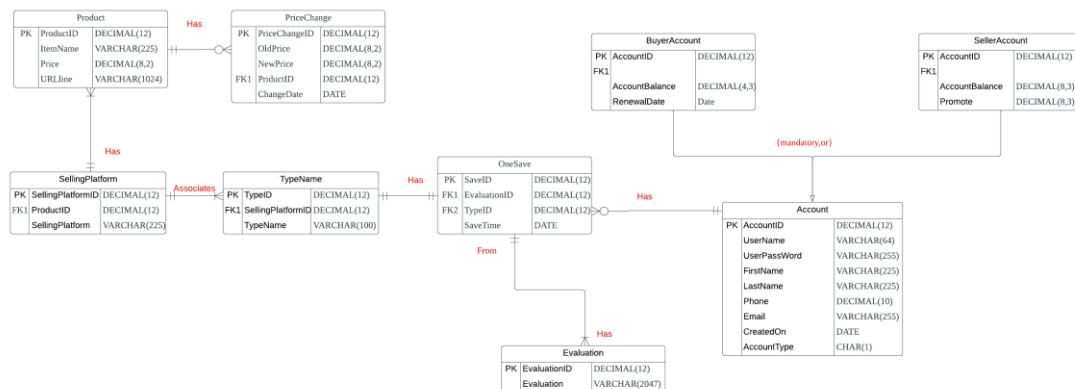
In reviewing my DBMS physical ERD, I find that the price will change in many times for one product, like the activity price and normal price, so it can benefit from historical data updates.

The new structural database rule is: Each product can have many price changes; each price change is for one Product.

Here is the new update ERD:



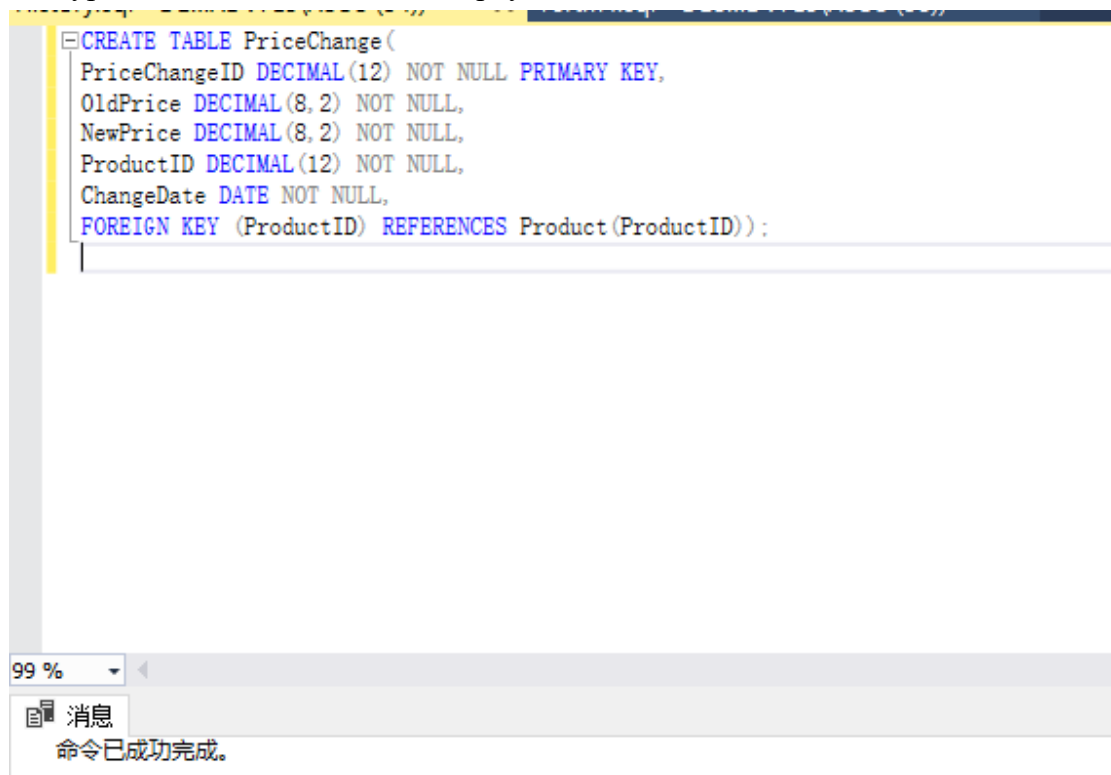
And here is the New physical ERD:



The PriceChange entity is present and linked to Product. Below are the attributes I added and why.

Attribute	Description
PriceChangeID	The primary key for the ID in the table with a DECIMAL(12) type data.
OldPrice	The price of the product before the price change. The datatype mirrors the Price datatype in the Product table.
NewPrice	The price of the product after the price change. The datatype mirrors the Price datatype in the Product table.
ProductID	The foreign key to the Product table, a reference to the product that had the change in price.
ChangeDate	Just save the change time with the DATE type data.

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



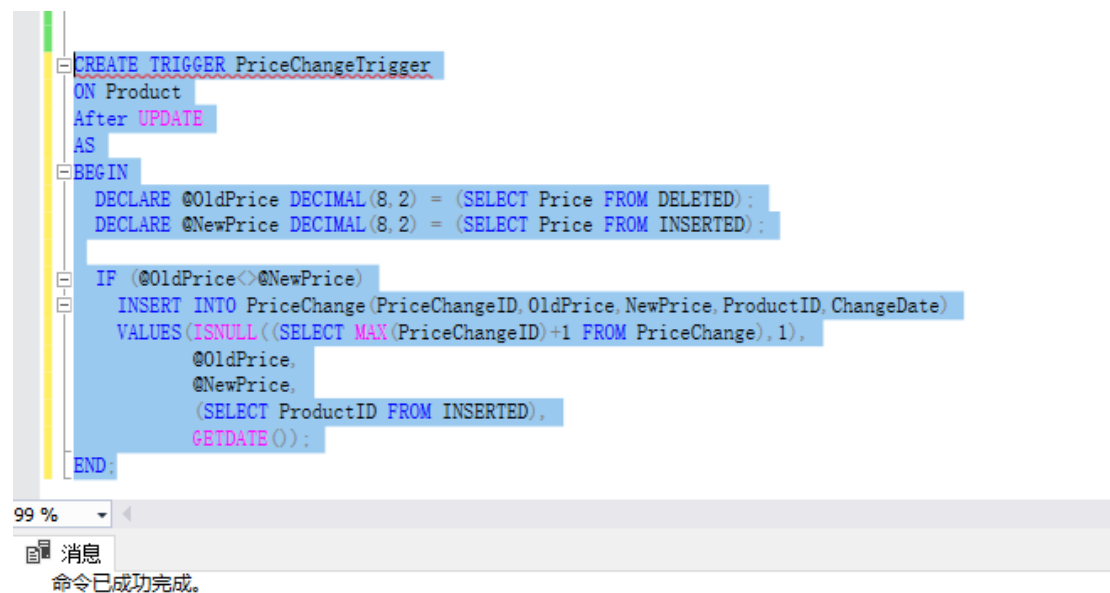
```
CREATE TABLE PriceChange (  
    PriceChangeID DECIMAL(12) NOT NULL PRIMARY KEY,  
    OldPrice DECIMAL(8,2) NOT NULL,  
    NewPrice DECIMAL(8,2) NOT NULL,  
    ProductID DECIMAL(12) NOT NULL,  
    ChangeDate DATE NOT NULL,  
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID));
```

99 %

消息

命令已成功完成。

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



```

CREATE TRIGGER PriceChangeTrigger
ON Product
After UPDATE
AS
BEGIN
    DECLARE @OldPrice DECIMAL(8,2) = (SELECT Price FROM DELETED);
    DECLARE @NewPrice DECIMAL(8,2) = (SELECT Price FROM INSERTED);

    IF (@OldPrice <> @NewPrice)
        INSERT INTO PriceChange (PriceChangeID, OldPrice, NewPrice, ProductID, ChangeDate)
        VALUES (ISNULL((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1),
                @OldPrice,
                @NewPrice,
                (SELECT ProductID FROM INSERTED),
                GETDATE());
END;

```

99 %

消息
命令已成功完成。

I explain it here line by line:

CODE	DESCRIPTION
CREATE TRIGGER PriceChangeTrigger ON Product After UPDATE	This starts the definition of the trigger and names it “PriceChangeTrigger”. The trigger is linked to the Product table, and is executed after any updated to that table.
AS BEGIN	This is part of the syntax starting the trigger block
DECLARE @OldPrice DECIMAL(8,2) = (SELECT Price FROM DELETED); DECLARE @NewPrice DECIMAL(8,2) = (SELECT Price FROM INSERTED);	This saves the old and new price by referencing the DELETED and INSERTED pseudo tables, respectively.
IF (@OldPrice<>@NewPrice)	This check ensures action is only taken if the price has been updated.
INSERT INTO PriceChange(PriceChangeID,OldPrice,NewPrice ,ProductID,ChangeDate) VALUES(ISNULL((SELECT MAX(PriceChangeID)+1 FROM PriceChange),1), @OldPrice, @NewPrice, (SELECT ProductID FROM INSERTED), GETDATE());	This inserts the record into the PriceChange table. The primary key is set by obtaining one over the next highest. The old and new price are used from the variables. The product ID is obtained from the INSERTED pseudo table. The date of the change is obtained by using the built-in GETDATE function.
END;	This ends the trigger definition.

The Product table has a product like this:

```
Select * from Product;
```

99 %

结果 消息

	ProductID	ItemName	Price	URLLine
1	1	Great Value Whole Vitamin D Milk, Gallon, 128 f...	3.17	https://www.walmart.com/ip/Great-Value-Whole-Vi...

The price of the product is 3.17\$.

Next, I update the price two times to 2.50\$ and 2.99\$.

```
UPDATE Product
Set Price = 2.50
Where ProductID = 1;
```

```
UPDATE Product
Set Price = 2.99
Where ProductID = 1;
```

99 %

消息

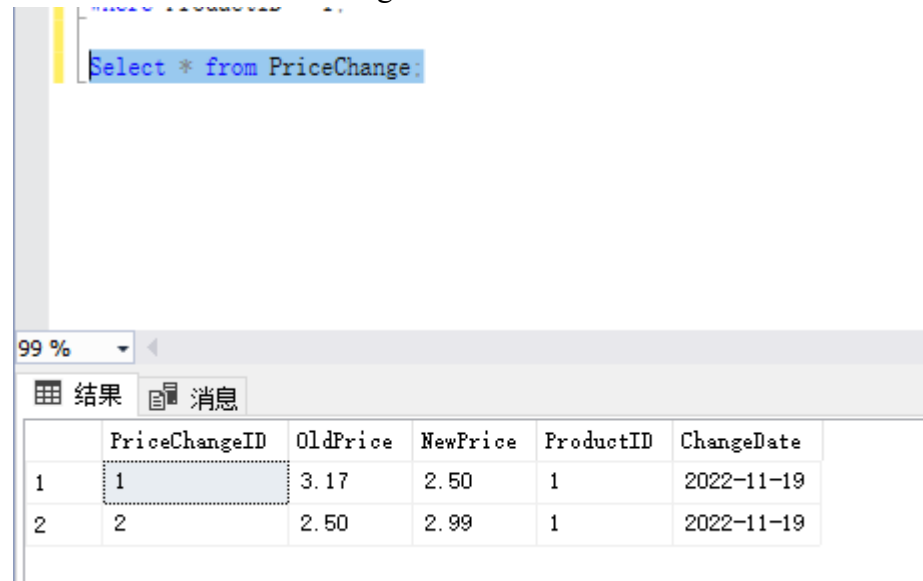
(1 行受影响)

(1 行受影响)

(1 行受影响)

(1 行受影响)

Then let's see the PriceChange Table:



```
Select * from PriceChange;
```

	PriceChangeID	OldPrice	NewPrice	ProductID	ChangeDate
1	1	3.17	2.50	1	2022-11-19
2	2	2.50	2.99	1	2022-11-19

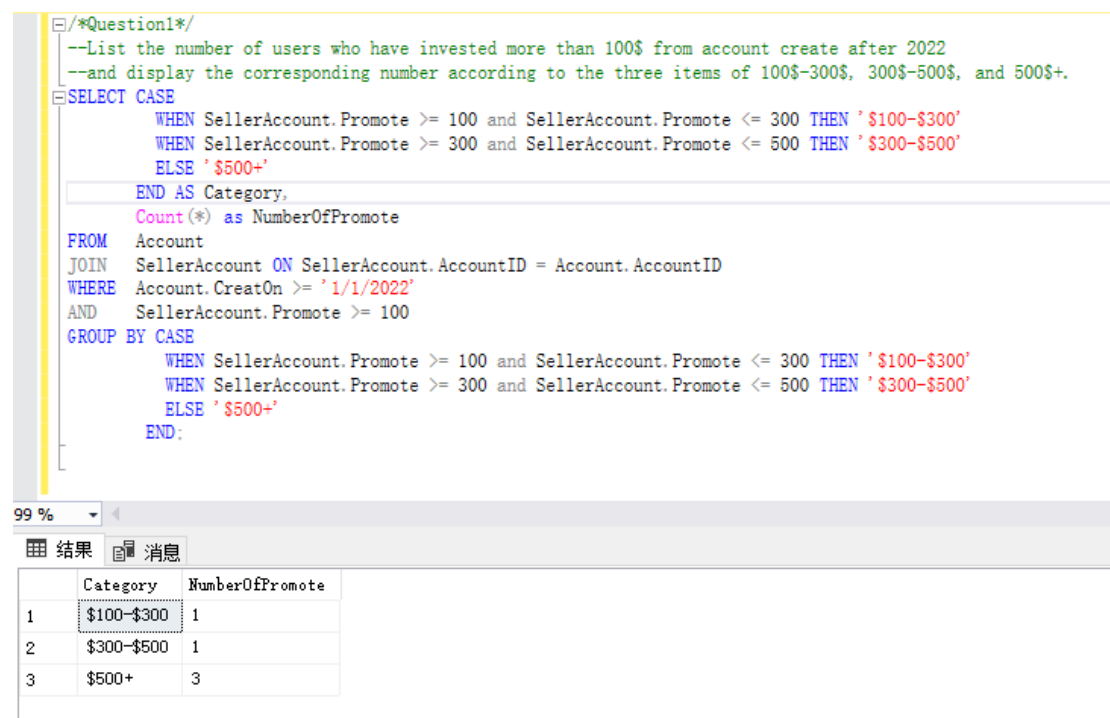
We can see there are two line, one is from 3.17\$ to 2.50\$, second is 2.50\$ to 2.99\$, so we successfully. The old and new price are now tracked with a trigger and a history table

BestBuy Question and Query

Question1: List the number of users who have invested more than 100\$ from account create after 2022 and display the corresponding number according to the three items of 100\$-300\$, 300\$-500\$, and 500\$+.

This question can help us confirm the investment of new users this year, and solve many problems, such as confirming the economic situation, the degree of attention of the app, and so on.

Here is a screenshot of the query I use:



The screenshot shows a SQL query editor with a query for Question 1. The query filters for accounts created on or after 1/1/2022 and where the seller account's promote value is greater than or equal to 100. It then groups the results by investment category (\$100-\$300, \$300-\$500, and \$500+) and counts the number of users in each category.

```
/*Question1*/
--List the number of users who have invested more than 100$ from account create after 2022
--and display the corresponding number according to the three items of 100$-300$, 300$-500$, and 500$+.
SELECT CASE
    WHEN SellerAccount.Promote >= 100 and SellerAccount.Promote <= 300 THEN '$100-$300'
    WHEN SellerAccount.Promote >= 300 and SellerAccount.Promote <= 500 THEN '$300-$500'
    ELSE '$500+'
END AS Category,
Count(*) as NumberOfPromote
FROM Account
JOIN SellerAccount ON SellerAccount.AccountID = Account.AccountID
WHERE Account.CreatOn >= '1/1/2022'
AND SellerAccount.Promote >= 100
GROUP BY CASE
    WHEN SellerAccount.Promote >= 100 and SellerAccount.Promote <= 300 THEN '$100-$300'
    WHEN SellerAccount.Promote >= 300 and SellerAccount.Promote <= 500 THEN '$300-$500'
    ELSE '$500+'
END;
```

The results are displayed in a table with two columns: Category and NumberOfPromote.

	Category	NumberOfPromote
1	\$100-\$300	1
2	\$300-\$500	1
3	\$500+	3

To get the results, I join the Account to the SellerAccount table, use 'SellerAccount.Promote >= 100' to set all the result more than 100\$ and use 'Account.CreatOn >= '1/1/2022'' to set the time in the 2022 years to now. Then just classify then by \$100-\$300, \$300-\$500, \$500+.

Let's Check the result, to prove it, I show the all detail of Account and SellerAccount:

```

SELECT *
FROM Account
JOIN SellerAccount ON SellerAccount.AccountID = Account.AccountID

```

	AccountID	UserName	UserPassword	FirstName	LastName	Phone	Email	CreatOn	AccountType	AccountID	AccountBalance	Promote
1	1	dgordn	xyz	Decker	Gordon	8655368488	Decker.gordon@gmail.com	2022-11-19	B	1	191.030	945.230
2	2	Hally	n2Ns5KvBNqt	Lanette	Massey	3158004388	lmassey1@salon.com	2022-02-09	S	2	982.830	7.710
3	3	Moshe	NmjstOW12	Jana	Ripsher	3972795626	jripsher2@khs.gov	2022-02-21	S	3	828.400	60.430
4	4	Dell	xdETWkaSr4Po	Phillipe	Gange	2727876153	pgange3@surveymonkey.com	2022-05-17	S	4	96.230	316.290
5	5	Chelsey	m2r72eGTm	Jodi	Ruggier	4421678545	jruggier4@spotify.com	2021-12-14	S	5	922.830	9.170
6	6	Aldus	fgv8B3	Clarabelle	Klyn	5751232008	oklyn5@bbc.co	2022-10-04	S	6	256.750	721.930
7	7	Ulberto	NOMMDwb4rg	Shannon	Riddlesden	2382159865	sridlesden6@mapquest.com	2022-06-25	S	7	438.650	2.970
8	8	Ellery	qqDH0RN	Lauritz	Gravenell	6057494861	lgravenell7@everbnation.com	2022-01-21	S	8	4.470	777.090
9	9	Ximenez	83hfyCK	Case	Thewlis	9502279156	othewlis0@wikia.com	2022-04-08	S	9	857.100	137.610

We can see, there are 8 account create after 2022, Ximenez is \$100-\$300, Dell is \$300-\$500, dgordn, Aldus and Ellery are \$500+, others are less than \$100, so it is right.

Question2: List all users created after 2022 and currently have a balance of more than \$300, arranged in the form of \$300-\$500, \$500-\$900, \$900+.

If the promotion is the current income, then the account balance of these sellers can be regarded as the money they may invest in the future, and we can use these data to judge future income.

Here is a screenshot of the query I use:

```

/*Question2*/
--List all users created after 2022 and currently have a balance of more than $300,
--arranged in the form of $300-$500, $500-$900, $900+.
SELECT CASE
    WHEN SellerAccount.AccountBalance >= 300 and SellerAccount.AccountBalance <= 500 THEN '$300-$500'
    WHEN SellerAccount.AccountBalance >= 500 and SellerAccount.AccountBalance <= 900 THEN '$500-$900'
    ELSE '$900+'
END AS Category,
Count(*) as RemainAccountBalance
FROM Account
JOIN SellerAccount ON SellerAccount.AccountID = Account.AccountID
WHERE Account.CreatOn >= '1/1/2022'
AND SellerAccount.AccountBalance >= 300
GROUP BY Category
    WHEN SellerAccount.AccountBalance >= 300 and SellerAccount.AccountBalance <= 500 THEN '$300-$500'
    WHEN SellerAccount.AccountBalance >= 500 and SellerAccount.AccountBalance <= 900 THEN '$500-$900'
    ELSE '$900+'
END.

```

	Category	RemainAccountBalance
1	\$300-\$500	1
2	\$500-\$900	2
3	\$900+	1

This question is similar to question1, join the Account to the SellerAccount table, then use the 'When...When...Else...End AS...Count...' to make the result table, and use 'SellerAccount.AccountBalance >= 300' to exclude the result <300, 'Account.CreatOn >= '1/1/2022' to set the date after 2022.

Select *
FROM Account
JOIN SellerAccount ON SellerAccount.AccountID = Account.AccountID

99 %

结果 消息

	AccountID	UserName	UserPassword	FirstName	LastName	Phone	Email	CreatOn	AccountType	AccountID	AccountBalance	Promote
1	1	dgordn	xyz	Decker	Gordon	8655368488	Decker.gordon@gmail.com	2022-11-19	B	1	191.030	945.230
2	2	Hally	n2Ns5KvBNqt	Lanette	Massey	3158004388	lmassey1@salon.com	2022-02-09	S	2	982.830	7.710
3	3	Moshe	Nmjst0W12	Jana	Ripsheer	3972795626	jripsheer2@hhs.gov	2022-02-21	S	3	828.400	60.430
4	4	Dell	xdETWkaSr4Po	Phillipe	Gange	2727876153	pgange3@surveymonkey.com	2022-05-17	S	4	96.230	316.290
5	5	Chelsey	m2r72eGTm	Jodi	Ruggier	4421678545	jruggier4@spotify.com	2021-12-14	S	5	922.830	9.170
6	6	Aldus	fgvcB3	Clarabelle	Klyn	5751232008	cklyn5@ebc.ca	2022-10-04	S	6	256.750	721.930
7	7	Ulberto	NOlMDwb4rg	Shannon	Riddlesden	2382159865	sriddlesden6@mapquest.com	2022-06-25	S	7	438.650	2.970
8	8	Ellery	qqDHoRW	Lauritz	Gravenell	6057494861	lgravenell7@everbnation.com	2022-01-21	S	8	4.470	777.090
9	9	Ximenez	83hfyCK	Case	Thewlis	9502279156	cthewlis0@wikia.com	2022-04-08	S	9	857.100	137.610

Back to see the join table, we can see Ulberto's balance is \$300-\$500, Ximenez and Moshe's balance is \$500-\$900, Hally's balance is \$900+, Chelsey's account balance is more than \$900 but account create in 2021, other people's balance are all less than \$300, so the result is right.

Question3: A useful question from the PriceChange history table is: Find what the biggest price change in was 2022 (Before Aug).

According to its results, we can infer in which month similar products change the most, and how much change will occur.

Here is PriceChange table:

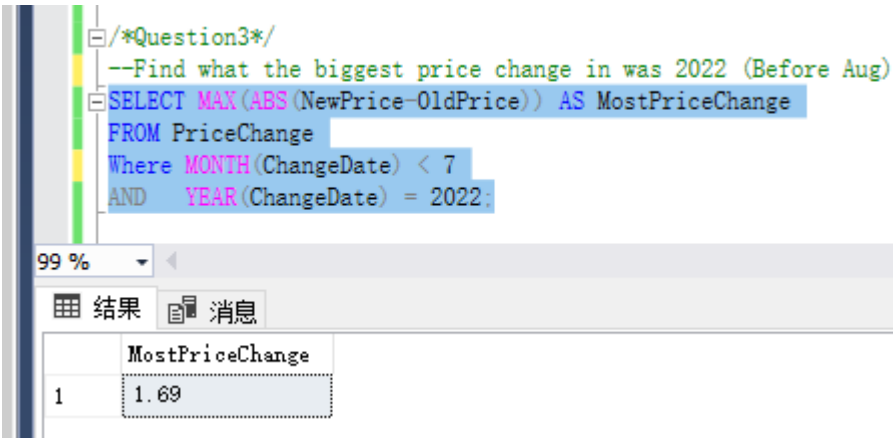
Select *
FROM PriceChange;

99 %

结果 消息

	PriceChangeID	OldPrice	NewPrice	ProductID	ChangeDate
1	1	3.14	3.53	1	2021-08-15
2	2	3.53	3.21	1	2021-09-28
3	3	3.04	3.58	4	2021-12-30
4	4	3.85	2.16	3	2022-03-16
5	5	3.21	3.78	1	2022-05-24
6	6	4.68	3.00	2	2022-05-24
7	7	3.58	2.64	4	2022-05-24
8	8	3.00	3.23	2	2022-08-17

We can see that 2022 Jan-June have 4 Update history, not pick up the 2021 years and 2022-8-17, here a screenshot of the query and its results:



Here is an explanation of the query:

CODE	DISCRIPTION
SELECT MAX(ABS(NewPrice-OldPrice)) AS MostPriceChange	Use new price - old price, get the absolute value for us to compare, and then find the largest one, and put it into the designed 'MostPriceChange'.
FROM PriceChange Where MONTH(ChangeDate) < 7 AND YEAR(ChangeDate) = 2022;	Set the source and limit, set the source in the PriceChange table, and limit it before August 2022, so that we can get the information from January to July 2022.

BestBuy Summary and Reflection

My database is an application called 'BestBuy', usually, each shopping platform can only sort all the products on the platform, and 'BestBuy' will store all the products that the user is interested in, and in the way the user wants. The sorting of categories is convenient for users to perform more refined filtering. The program is dedicated to creating a list of favorite products for users, and can even rate and recommend products based on other people's storage. I also divided the account into buyer account and seller account, and each one is divided into free account and paid account.

When I was designing the database, I realized that 'BestBuy' might be considered a platform, so it might gradually expand and store more and more data, including that it might be necessary to provide separate services for buyers and sellers. specialized program. Keep track of bills, payment records, credit cards, and more. And based on the third case, some basic storage records are needed. My current idea is to wait until there are certain users and enough information to be stored before turning on the recommended function. I hope there are some suggestions on these.

The structural database rules and ERD for my database design contain the important entities of Account, selling platform, type, name, price, evaluation, URL, Savetime and Promote as well as relationships between them.

I saved a lot of entities in the use case, which greatly increased my work volume. Even a small use case may use a lot of entities. It can be seen from this attempt: 'The number of entities is not determined by the size of the use case, but by the system, the people using the system, and the database.'

My database is taking shape with a DBMS Physical ERD. This means I can start building the database now.

Mapping the conceptual ERD to the physical ERD is just a mechanical process for me, hope it and specialization-generalization rules and EERD will give me enough help when I build the database.

In the process, I optimized and adjusted it, I merged some instances, which can save running time and space consumption, and then I established Table and established various corresponding indexes, which is very happy, the database is finally really created.

I performed a data transaction and created a historical update table so that some data changes can be recorded and used. I updated the ERD and rule for this table and set a Trigger to track the data.

Then I set some problems and tried to create organization driven queries to solve them.

As I reflect on the database and my accomplishments, I can say it's been a long but rewarding road. I'm excited to be working on this project and will try to keep improving it.