

## Contents

Iteration Introduction .....	2
Maintaining History Tables with Triggers .....	4
Conceptual Car ERD with Price Change .....	5
DBMS Physical ERD With Price Change .....	5
PriceChange Table Creation .....	6
Price Changes for SQL Server .....	6
Price Change Trigger for SQL Server .....	7
Price Changes for Oracle .....	8
Price Change Trigger for Oracle .....	8
Price Changes for Postgres .....	10
Code: Price Change Trigger for Postgres .....	10
Capturing History for your Project .....	12
TrackMyBuys History .....	12
Data Visualizations .....	17
TrackMyBuys Data Visualization .....	17
Summary and Reflection .....	21
TrackMyBuys Reflection .....	21
Items to Submit .....	22
Evaluation .....	23

## Iteration Introduction

This iteration is about putting the finishing touches on your database. After the last iteration, your database has data and you are able to ask it useful questions. In this iteration, you learn keep history for attributes, so it's possible to know what an attribute's value was at any point in the past. You also create useful data visualizations and stories.

To help you keep a bearing on what you have left to complete, let's again look at an outline of what you created in prior iterations, and what you will be creating in this final iteration.

Prior Iterations	Iteration 1	<p><i>Project Direction Overview</i> – You provide an overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it.</p> <p><i>Use Cases and Fields</i> – You provide use cases that enumerate steps of how the database will be typically used, also identify significant database fields needed to support the use case.</p> <p><i>Summary and Reflection</i> – You concisely summarize your project and the work you have completed thus far, and additionally record your questions, concerns, and observations, so that you and your facilitator or instructor are aware of them and can communicate about them.</p>
	Iteration 2	<p><i>Structural Database Rules</i> – You define structural database rules which formally specify the entities, relationships, and constraints for your database design.</p> <p><i>Conceptual Entity Relationship Diagram (ERD)</i> – You create an initial ERD, the universally accepted method of modeling and visualizing database designs, to visualize the entities and relationships defined by the structural database rules.</p>
	Iteration 3	<p><i>Conceptual Extended Entity Relationship Diagram (EERD)</i> – You add specialization-generalization into your conceptual ERD and structural database rules.</p> <p><i>Initial DBMS Physical ERD</i> – You create an initial DBMS physical ERD, which is tied to a specific relational database vendor and version, with SQL-based constraints and datatypes.</p>
	Iteration 4	<p><i>Full DBMS Physical ERD</i> – You define the attributes for your database design and add them to your DBMS Physical ERD.</p> <p><i>Normalization</i> – You normalize your DBMS physical ERD to reduce or eliminate data redundancy.</p> <p><i>Database Structure</i> – You create your tables, sequences, and constraints in SQL.</p>
	Iteration 5	<p><i>Reusable, Transaction-Oriented Store Procedures</i> – You create and execute reusable stored procedures that complete the steps of transactions necessary to add data to your database.</p> <p><i>Questions and Queries</i> – You define questions useful to the organization or application that will use your database, then write queries to address the questions.</p> <p><i>Index Placement and Creation</i> – To speed up performance, you identify columns needing indexes for your database, then create them in SQL.</p>
Current Iteration	Iteration 6	<p><i>History Table</i> – You create a history table to track changes to values, and develop a trigger to maintain it.</p> <p><i>Data Visualizations</i> – You tell effective data stories with data visualizations.</p>

First, make any revisions to your design and scripts that you are necessary before you proceed further.

## **Maintaining History Tables with Triggers**

History tables are an important tool for tracking important changes over time. For some data, we're only concerned with its current value. An example could be a person's name. If a person changes their name, we need to update our record in our database, but may not be concerned with what the name used to be. For some data, we're concerned with its current value and its past values. For example, if we're selling a product and its price changes, we would want to record its old price in addition to its new price. This way, research into past purchases would give us accurate pricing, as we could use the correct price the product had at any point in time.

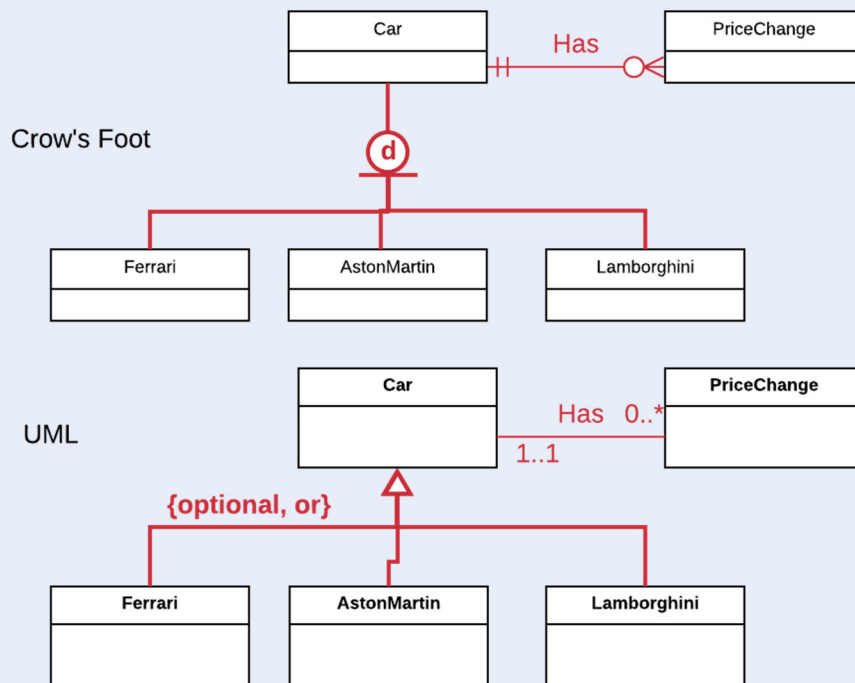
Maintaining a history table with a trigger has its advantages. If a column's value changes, a trigger will record the change, regardless of whether the application, a person, or a script made the change. If the history table is maintained through the application, changes to the database made outside of the application would not be recorded. Whether the application or a trigger is used is for a particular situation depends upon the circumstance and the organization. For this project, you'll use a trigger to maintain a history table for your database.

A standard history table contains a primary key, the old and new value(s) for the column(s) being tracked, a foreign key to the table being tracked, and the date of the change. With such a history table, it is possible to see when the value(s) changed, and when, which is quite useful for analyzing the data over time.

Let's look at an example of maintaining a history table for the Car example we've been following from the prior iteration. We'll track price changes for cars, which would be useful to track, for example, which kinds of cars have the most price drops. First, we need to create the history table with the old price, the new price, a foreign key to the Car table, and a date. To ensure we're designing this properly, we'll follow the process of creating the structural database rule and updating the ERDs, before creating the table itself in SQL.

The new structural database rule would be: each car can have many price changes; each price change is for one car. The new conceptual ERD would look as illustrated below.

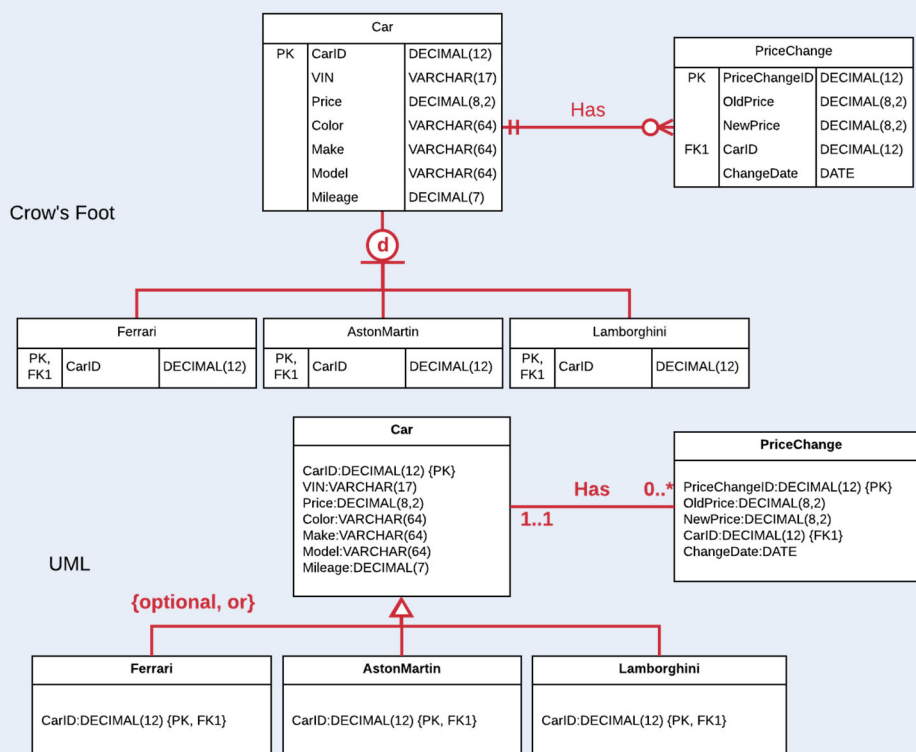
## Conceptual Car ERD with Price Change



Notice that the new PriceChange entity is present in the ERD as defined by the structural database rule.

The new DBMS physical ERD would look as illustrated below.

## DBMS Physical ERD With Price Change



You'll notice the PriceChange entity is present, now with attributes. The attributes are described in the table below.

Attribute	Description
PriceChangeID	This is the primary key of the history table. It is a DECIMAL(12) to allow for many values.
OldPrice	This is the price of the car before the price change. The datatype mirrors the Price datatype in the Car table.
NewPrice	This is the price of the car after the price change. The datatype mirrors the Price datatype in the Car table.
CarID	This is a foreign key to the Car table, a reference to the car that had the change in price.
ChangeDate	This is the date the price change occurred, with a DATE datatype.

Now that we've modeled the new entity, we can now create it in SQL.

To create the PriceChange entity and sequence, we would use the following CREATE TABLE and CREATE SEQUENCE statements.

#### PriceChange Table Creation

```
CREATE TABLE PriceChange (  
  PriceChangeID DECIMAL(12) NOT NULL PRIMARY KEY,  
  OldPrice DECIMAL(8,2) NOT NULL,  
  NewPrice DECIMAL(8,2) NOT NULL,  
  CarID DECIMAL(12) NOT NULL,  
  ChangeDate DATE NOT NULL,  
  FOREIGN KEY (CarID) REFERENCES Car(CarID));  
  
CREATE SEQUENCE PriceChangeSeq START WITH 1;
```

Notice that the table definition corresponds exactly to the entity definition in the DBMS physical ERD. By modeling the new entity first, then creating it in SQL, we've helped ensure that it is designed and implemented properly.

### Price Changes for SQL Server

Next, we need to create the trigger that will maintain the table. You learned how to create a similar trigger in Lab 4, so I will not detail that here. Feel free to review that portion of Lab 4 should you need. Below is the trigger for SQL Server.

## Price Change Trigger for SQL Server

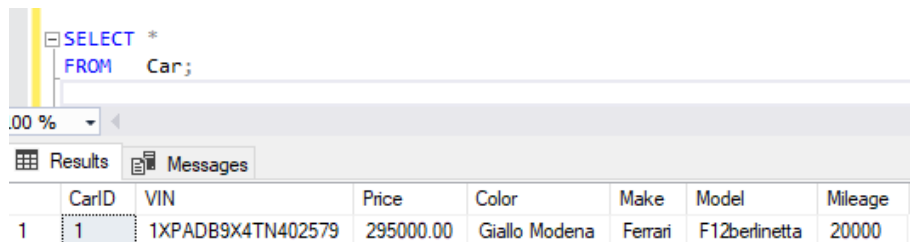
```
CREATE TRIGGER PriceChangeTrigger
ON Car
AFTER UPDATE
AS
BEGIN
    DECLARE @OldPrice DECIMAL(8,2) = (SELECT Price FROM DELETED);
    DECLARE @NewPrice DECIMAL(8,2) = (SELECT Price FROM INSERTED);

    IF (@OldPrice <> @NewPrice)
        INSERT INTO PriceChange (PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate)
        VALUES (NEXT VALUE FOR PriceChangeSeq,
                @OldPrice,
                @NewPrice,
                (SELECT CarID FROM INSERTED),
                GETDATE());
END;
```

Let's work through it line by line.

CODE	DESCRIPTION
CREATE TRIGGER PriceChangeTrigger ON Car	This names the trigger "PriceChangeTrigger" and links it to the CAR table.
AFTER UPDATE AS BEGIN	This indicates that the trigger should run after the table is updated (ignoring INSERTS and DELETES), along with some keywords needed by the T-SQL syntax.
DECLARE @OldPrice DECIMAL(8,2) = (SELECT Price FROM DELETED); DECLARE @NewPrice DECIMAL(8,2) = (SELECT Price FROM INSERTED);	This captures both the old and the new price from before and after the update, by accessing the DELETED and INSERTED pseudo tables provide by T-SQL.
IF (@OldPrice <> @NewPrice)	This is a check that the new price differs from the old price. A price change is only recorded if the prices differ. Perhaps another column in the table was updated, in which case no price change is recorded.
INSERT INTO PriceChange (PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate) VALUES (NEXT VALUE FOR PriceChangeSeq, @OldPrice, @NewPrice, (SELECT CarID FROM INSERTED), GETDATE());	This is the insert statement that records the price change by adding a row into the PriceChange table. The PriceChangeID column is generated with the PriceChangeSeq. The old and new price as already saved in the variables are used. The CarID is extracted from the INSERTED pseudo table provided by T-SQL. The built-in function GETDATE() is used to obtain the date of the change.
END;	This ends the trigger definition.

To test out that the trigger works, I created a row in the Car table with a CarID of 1, and an initial price of \$295,000, resulting in this Car row.



	CarID	VIN	Price	Color	Make	Model	Mileage
1	1	1XPADB9X4TN402579	295000.00	Giallo Modena	Ferrari	F12berlinetta	20000

I then lower the price three times, to \$285,000, \$275,000, and \$265,000, as demonstrated below.

```

UPDATE Car
SET Price = 285000
WHERE CarID = 1;

UPDATE Car
SET Price = 275000
WHERE CarID = 1;

UPDATE Car
SET Price = 265000
WHERE CarID = 1;

```

100 %

Messages

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

Last, I verify that my trigger worked as expected by selecting from the PriceChange table, illustrated below.

```

SELECT *
FROM PriceChange;

```

100 %

Results Messages

	PriceChangeID	OldPrice	NewPrice	CarID	ChangeDate
1	1	295000.00	285000.00	1	2018-10-04
2	2	285000.00	275000.00	1	2018-10-04
3	3	275000.00	265000.00	1	2018-10-04

The results demonstrate that the price went from \$295,000 to \$285,000, then to \$275,000, then to \$265,000, all for the car with CarID 1.

## Price Changes for Oracle

The trigger below tracks price changes for the Car table in Oracle.

### Price Change Trigger for Oracle

```

CREATE OR REPLACE TRIGGER PriceChangeTrigger
BEFORE UPDATE OF Price ON Car
FOR EACH ROW
BEGIN
    INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate)
    VALUES (PriceChangeSeq.nextval,
            :OLD.Price,
            :NEW.Price,
            :New.CarID,
            trunc(sysdate));
END;

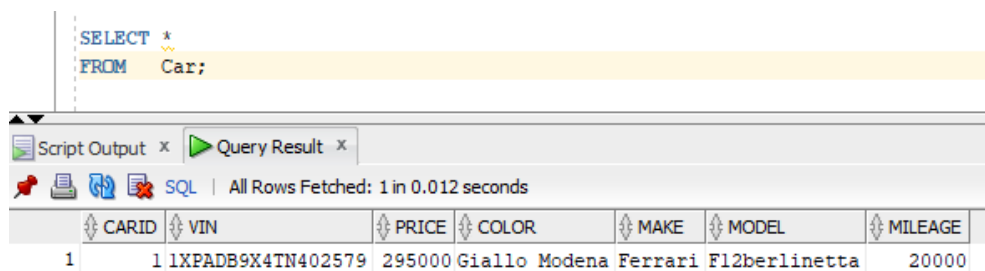
```



Let's work through it line by line.

CODE	DESCRIPTION
<b>CREATE OR REPLACE TRIGGER PriceChangeTrigger BEFORE UPDATE OF Price ON Car</b>	This names the trigger "PriceChangeTrigger" and links it to the Car table. Further, it specifically indicates that the trigger will run before any update on Car table (ignoring deletes and inserts), and it only runs if the Price column is updated.
<b>FOR EACH ROW BEGIN</b>	This indicates that the trigger should run for every row that is updated, which is necessary to get access to the specific prices that changed.
<b>INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate) VALUES (PriceChangeSeq.nextval,       :OLD.Price,       :NEW.Price,       :New.CarID,       trunc(sysdate)) ;</b>	This is the insert statement that records the price change by adding a row into the PriceChange table. The PriceChangeID column is generated using the PriceChangeSeq sequence.. The old and new prices are accessed through the :NEW and :OLD pseudo tables provided in PL/SQL triggers. The CarID is extracted from the :NEW pseudo table. The built-in variable sysdate obtains the current date, and the built-in function trunc() removes the time.
<b>END ;</b>	This ends the trigger definition.

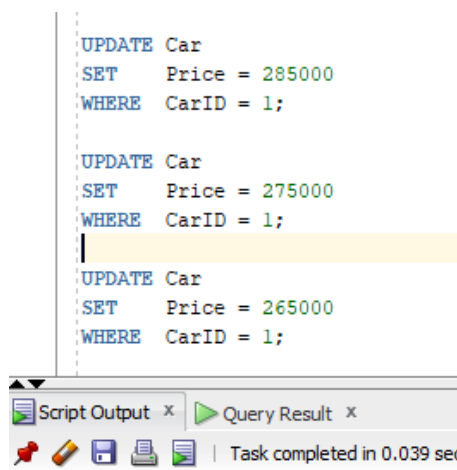
To test out that the trigger works, I created a row in the Car table with a CarID of 1, and an initial price of \$295,000, resulting in this Car row.



The screenshot shows a SQL Developer window with a query result. The query is `SELECT * FROM Car;`. The result set contains one row with the following values:

CARID	VIN	PRICE	COLOR	MAKE	MODEL	MILEAGE
1	1XPADB9X4TN402579	295000	Giallo	Modena	Ferrari Fl2berlinetta	20000

I then lower the price three times, to \$285,000, \$275,000, and \$265,000, as demonstrated below.



The screenshot shows a SQL Developer window with three UPDATE queries executed sequentially. The queries are:

```

UPDATE Car
SET Price = 285000
WHERE CarID = 1;

UPDATE Car
SET Price = 275000
WHERE CarID = 1;

UPDATE Car
SET Price = 265000
WHERE CarID = 1;

```

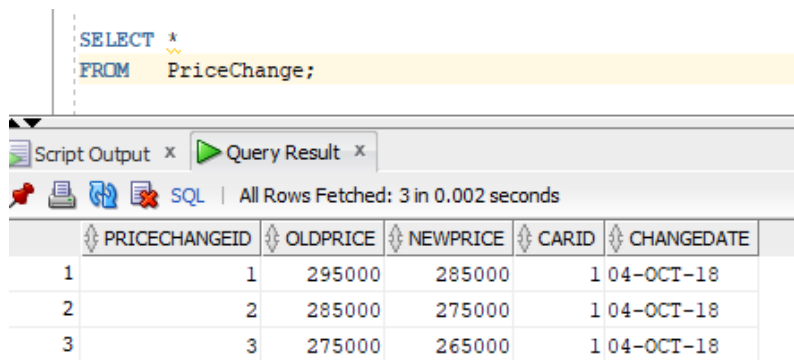
The status bar at the bottom indicates "Task completed in 0.039 seconds".

1 row updated.

1 row updated.

1 row updated.

Last, I verify that my trigger worked as expected by selecting from the PriceChange table, illustrated below.



The screenshot shows a SQL query window with the query: `SELECT * FROM PriceChange;` The results are displayed in a table with the following columns: PRICECHANGEID, OLDPRICE, NEWPRICE, CARID, and CHANGEDATE. The results show three rows of data for CarID 1.

PRICECHANGEID	OLDPRICE	NEWPRICE	CARID	CHANGEDATE
1	295000	285000	1	04-OCT-18
2	285000	275000	1	04-OCT-18
3	275000	265000	1	04-OCT-18

The results demonstrate that the price went from \$295,000 to \$285,000, then to \$275,000, then to \$265,000, all for the car with CarID 1.

## Price Changes for Postgres

The trigger below tracks price changes for the Car table in Postgres.

### Code: Price Change Trigger for Postgres

```
CREATE OR REPLACE FUNCTION PriceChangeFunction()
RETURNS TRIGGER LANGUAGE plpgsql
AS $trigfunc$
BEGIN
    INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate)
    VALUES (nextval('PriceChangeSeq'),
            OLD.Price,
            NEW.Price,
            New.CarID,
            current_date);

    RETURN NEW;
END;
$trigfunc$;

CREATE TRIGGER PriceChangeTrigger
BEFORE UPDATE OF Price ON Car
FOR EACH ROW
EXECUTE PROCEDURE PriceChangeFunction();
```

Let's work through it line by line.

CODE	DESCRIPTION
<code>CREATE OR REPLACE FUNCTION PriceChangeFunction() RETURNS TRIGGER LANGUAGE plpgsql</code>	This starts the definition of a function named "PriceChangeFunction" that will be executed when the trigger fires. The language used is Postgres' version of PL/SQL.
<code>AS \$trigfunc\$ BEGIN</code>	This is part of the syntax starting the function block.
<code>INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate) VALUES (nextval('PriceChangeSeq'),         OLD.Price,         NEW.Price,         New.CarID,         current_date);</code>	This is the insert statement that records the price change by adding a row into the PriceChange table. The PriceChangeID column is generated by using the PriceChangeSeq sequence.. The old and new prices are accessed through the NEW and OLD pseudo tables provided in plpgsql triggers. The CarID is extracted from the NEW pseudo table. The built-in variable current_date

	obtains the current date.
<b>RETURN NEW;</b> <b>END;</b> <b>\$trigfunc\$</b>	This ends the function definition.
<b>CREATE TRIGGER PriceChangeTrigger</b> <b>BEFORE UPDATE OF Price ON Car</b> <b>FOR EACH ROW</b>	This indicates that a trigger named "PriceChangeTrigger" is being defined, to be triggered whenever the Price column is updated in the Car table. The trigger is to run for each row updated.
<b>EXECUTE PROCEDURE PriceChangeFunction() ;</b>	This indicates that the trigger executes the function PriceChangeFunction() whenever it is executed.

To test out that the trigger works, I created a row in the Car table with a CarID of 1, and an initial price of \$295,000, resulting in this Car row.

65

SELECT \*

66

FROM Car

67

Data Output

[Explain](#)
[Messages](#)
[Notifications](#)
[Query History](#)

	carid numeric (12)	vin character varying (17)	price numeric (8,2)	color character varying (64)	make character varying (64)	model character varying (64)	mileage numeric (7)
1	1	1XPADB9X4TN402579	295000.00	Giallo Modena	Ferrari	F12berlinetta	20000

I then lower the price three times, to \$285,000, \$275,000, and \$265,000, as demonstrated below.

```

77 UPDATE Car
78 SET Price = 285000
79 WHERE CarID = 1;
80
81 UPDATE Car
82 SET Price = 275000
83 WHERE CarID = 1;
84
85 UPDATE Car
86 SET Price = 265000
87 WHERE CarID = 1;
88
89

```

Data Output	Explain	Messages	Notifications	Que
UPDATE 1				
Query returned successfully in 51 msec.				

Last, I verify that my trigger worked as expected by selecting from the PriceChange table, illustrated below.

```

70 SELECT *
71 FROM PriceChange;
72
73

```

Data Output

[Explain](#)

[Messages](#)

[Notifications](#)

[Query History](#)

	pricechangeid numeric (12)	oldprice numeric (8,2)	newprice numeric (8,2)	carid numeric (12)	changedate date
1	1	295000.00	285000.00	1	2018-10-05
2	2	285000.00	275000.00	1	2018-10-05
3	3	275000.00	265000.00	1	2018-10-05

The results demonstrate that the price went from \$295,000 to \$285,000, then to \$275,000, then to \$265,000, all for the car with CarID 1.

## Capturing History for your Project

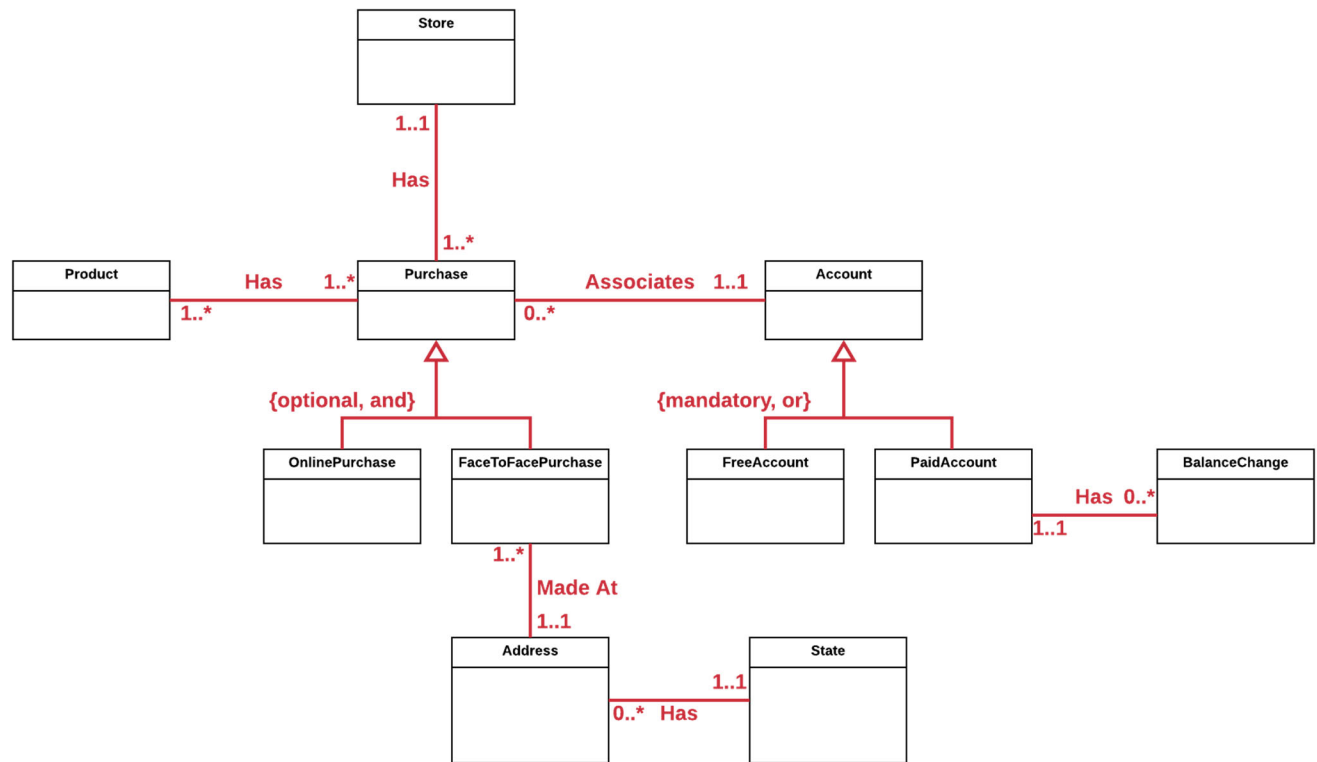
Select a column or columns that would benefit from a record of historical values in your database. Go through the process of designing a new history table by creating a new structural database rule and updating your ERDs. Then create the history table and sequence in SQL, and define a trigger that maintains the history table. Provide proof that your trigger maintains the history correctly with screenshots.

Here is the work for TrackMyBuys.

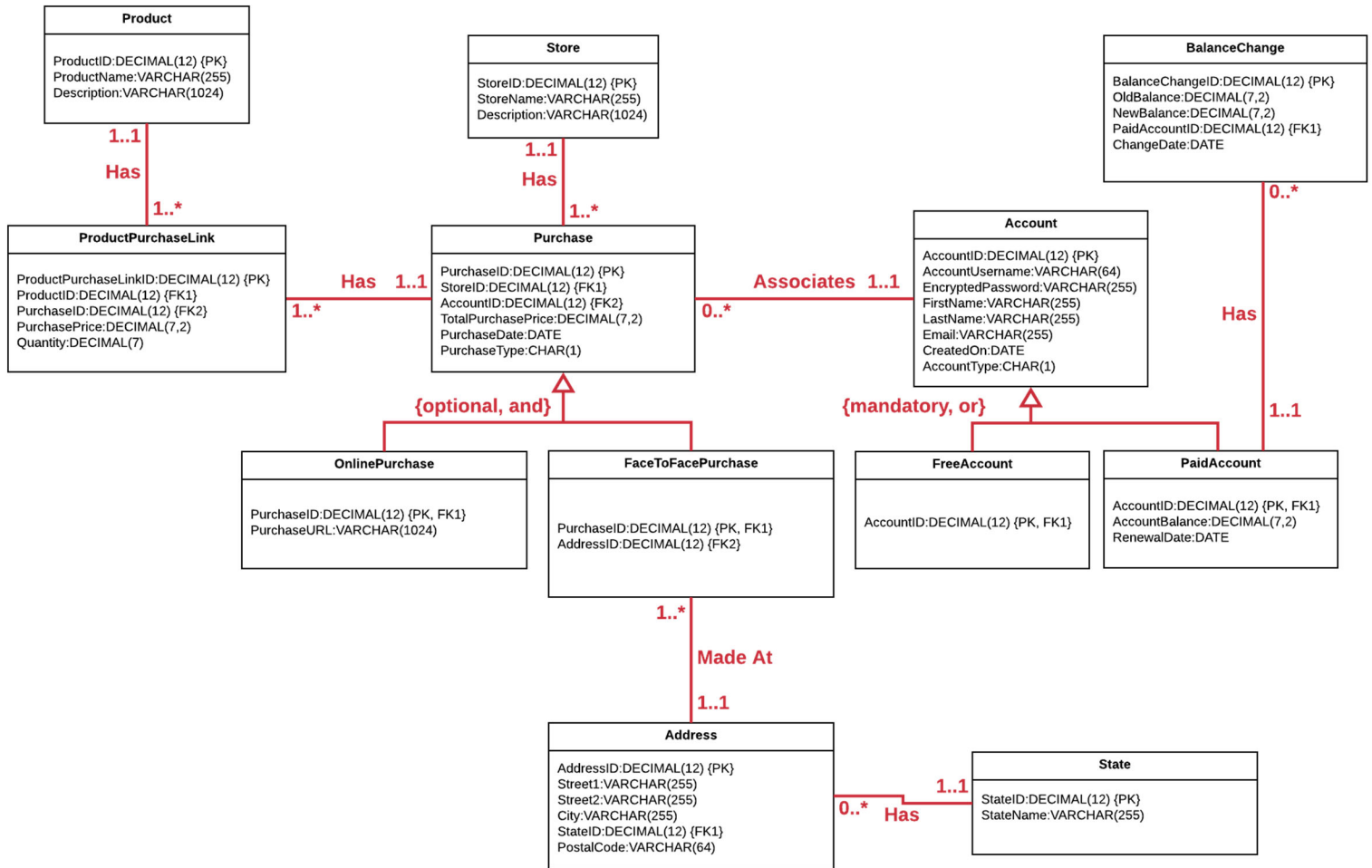
### TrackMyBuys History

In reviewing my DBMS physical ERD, one piece of data that would obviously benefit from a historical record a person's balance in the PaidAccount table. Such a history would help me calculate statistics about account balances that are accurate over time. First, my new structural database rule is: Each paid account may have many balance changes; each balance change is for a paid account.

My updated conceptual ERD is below.



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



The BalanceChange entity is present and linked to PaidAccount. Below are the attributes I added and why.

Attribute	Description	Example
BalanceChangeID	This is the primary key of the history table. It is a DECIMAL(12) to allow for many values.	1
OldBalance	This is the balance of the account before the change. The datatype mirrors the Balance datatype in the PaidAccount table.	25.00
NewBalance	This is the balance of the account after the change. The datatype mirrors the Balance datatype in the PaidAccount table.	35.00
PaidAccountID	This is a foreign key to the PaidAccount table, a reference to the account that had the change in balance.	1
ChangeDate	This is the date the balance change occurred, with a DATE datatype.	11/14/2022

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

```
CREATE TABLE BalanceChange (
    BalanceChangeID DECIMAL(12) NOT NULL PRIMARY KEY,
    OldBalance DECIMAL(7,2) NOT NULL,
    NewBalance DECIMAL(7,2) NOT NULL,
    PaidAccountID DECIMAL(12) NOT NULL,
    ChangeDate DATE NOT NULL,
    FOREIGN KEY (PaidAccountID) REFERENCES PaidAccount(AccountID));

CREATE SEQUENCE BalanceChangeSeq START WITH 1;
```

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

```
CREATE OR ALTER TRIGGER BalanceChangeTrigger
ON PaidAccount
AFTER UPDATE
AS
BEGIN
    DECLARE @OldBalance DECIMAL(7,2) = (SELECT AccountBalance FROM DELETED);
    DECLARE @NewBalance DECIMAL(7,2) = (SELECT AccountBalance FROM INSERTED);

    IF (@OldBalance <> @NewBalance)
        INSERT INTO BalanceChange(BalanceChangeID, OldBalance, NewBalance, PaidAccountID, ChangeDate)
        VALUES(NEXT VALUE FOR BalanceChangeSeq,
            @OldBalance,
            @NewBalance,
            (SELECT AccountID FROM INSERTED),
            GETDATE());
END;
```

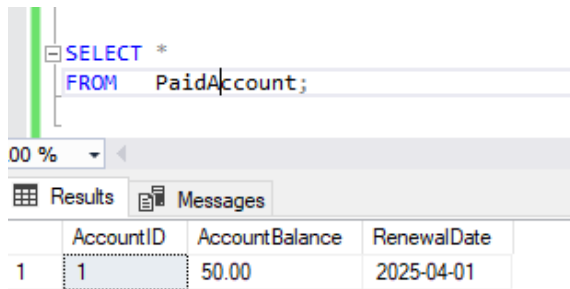
Messages  
Commands completed successfully.

I explain it here line by line.

CODE	DESCRIPTION
CREATE OR ALTER TRIGGER BalanceChangeTrigger ON PaidAccount AFTER UPDATE	This starts the definition of the trigger and names it "BalanceChangeTrigger". The trigger is linked to the PaidAccount table, and is executed after any updated to that table.
AS BEGIN	This is part of the syntax starting the trigger block.
DECLARE @OldBalance DECIMAL(7,2) = (SELECT AccountBalance FROM DELETED); DECLARE @NewBalance DECIMAL(7,2) = (SELECT AccountBalance FROM INSERTED);	This saves the old and new balances by referencing the DELETED and INSERTED pseudo tables, respectively.
IF (@OldBalance <> @NewBalance)	This check ensures action is only taken if the balance has been updated.
INSERT INTO BalanceChange (BalanceChangeID, OldBalance, NewBalance, PaidAccountID,	This inserts the record into the BalanceChange table. The primary key is set by using the BalanceChangeSeq sequence. The old and

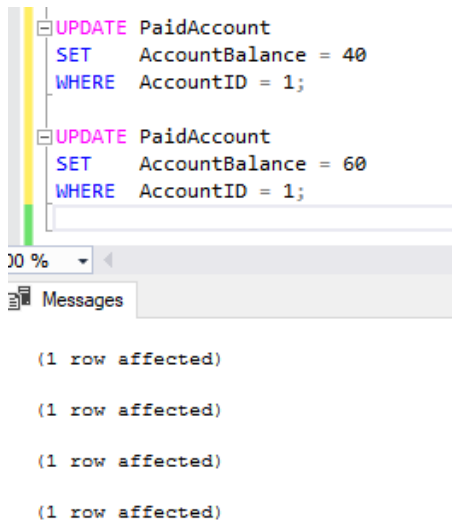
ChangeDate) VALUES(NEXT VALUE FOR BalanceChangeSeq, @OldBalance, @NewBalance, (SELECT AccountID FROM INSERTED), GETDATE() );	new balances are used from the variables. The account ID is obtained from the INSERTED pseudo table. The date of the change is obtained by using the built-in GETDATE function.
END ;	This ends the trigger definition.

I start by ensuring there is an account created. In this case, it has an ID of 1 with a balance of \$50. as illustrated by the screenshot below.



AccountID	AccountBalance	RenewalDate
1	50.00	2025-04-01

Next, I update the balance a couple of times, once to \$40, and again to \$60.



```

UPDATE PaidAccount
SET AccountBalance = 40
WHERE AccountID = 1;

UPDATE PaidAccount
SET AccountBalance = 60
WHERE AccountID = 1;

```

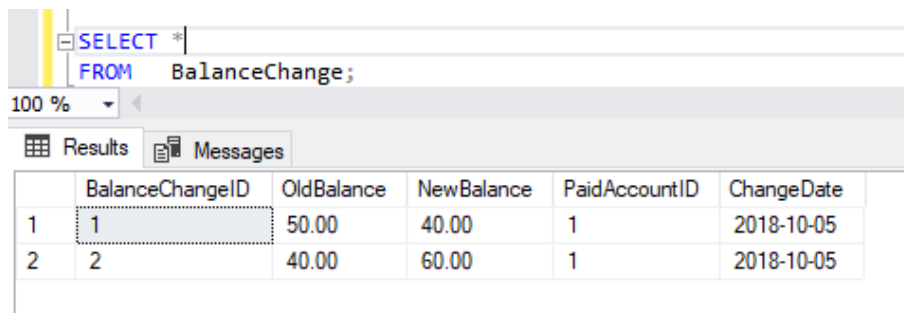
(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

Last, I verify that the BalanceChange table has a record of these balance changes in the screenshot below.



BalanceChangeID	OldBalance	NewBalance	PaidAccountID	ChangeDate
1	50.00	40.00	1	2018-10-05
2	40.00	60.00	1	2018-10-05



You'll notice that there are two rows, one for the change from \$50 to \$40, and the second from \$40 to \$60. The old and new balances are now tracked with a trigger and a history table.

## Data Visualizations

In a prior lab, you learned how to create effective data visualizations and stories. You may want to review that learning before proceeding.

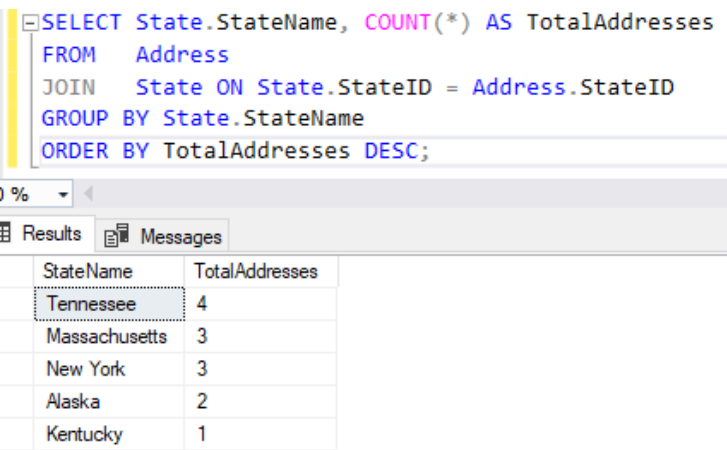
Create two visualizations of the data in your database using charts, graphs, or other visualizations. Clearly explain the data story conveyed by each visualization. Ensure that the visualizations and data stories are useful and appropriate given the intended use of your database.

Below is an example visualization for TrackMyBuys.

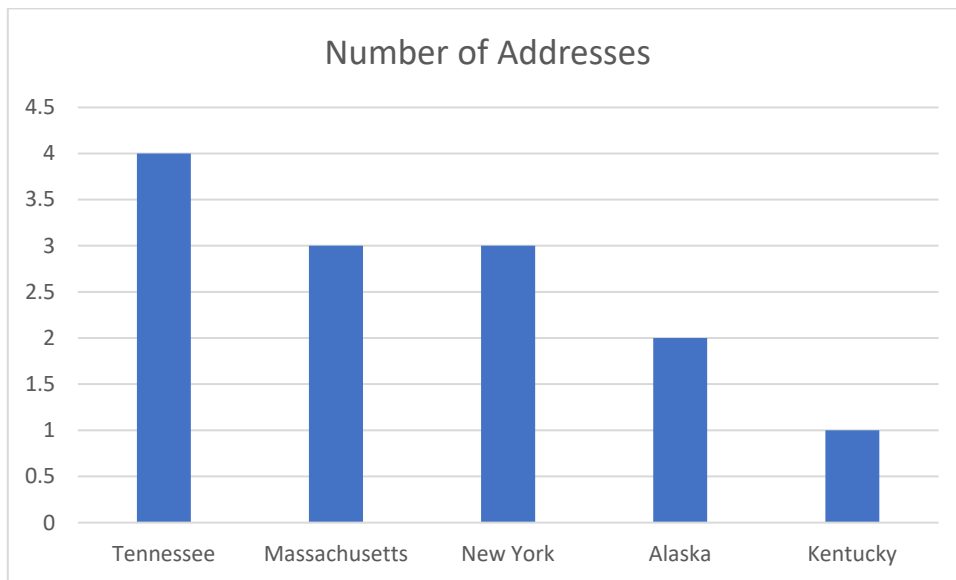
### TrackMyBuys Data Visualization

To be effective, any business should understand the demographics of their customers. The more that is known about their customers, the more effectively they can retain them, and reach new customers. The simple TrackMyBuys schema only has one type of demographic information – address. To that end, I ask the question, “Which states do TrackMyBuys’ customers reside in?”. The answer will give me an idea of where the customers live. Perhaps I could use that to market more effectively in states that do not have customers, or to gain even more customers in the most popular states.

To answer this question, I develop and execute a query that counts the number of addresses residing in each state. The results are ordered from the most to the least. This query and results are shown below.



Since these results only have one measure, I plan to use a simple bar chart to visualize the results. After exporting to a CSV and creating the bar chart in Excel, I see the following result.



What story does this visualization tell us? I observe several things. First, I observe that TrackMyBuys customers are limited to only 5 states, which is surprising given that it's available for download in all 50 states. Second, I observe that most customers reside in either the Southeastern or Northeastern U.S. The Midwestern and Western states are missing entirely. Third, I observe that the remote state of Alaska has some customers, enough to get my attention. I would want to dig deeper to find out why the customer presence is in these areas in particular, and not in other areas.

This information could be used in a variety of ways. First, I would want to market more effectively to the entire U.S. Second, I would want to increase the presence in the Midwestern and Western states. Third, I would want to increase the total number of customers regardless of what state they lived in.

---

You are encouraged but not required to base one of your visualizations on data from your history table. If you opt to do this, the query should be grouped by some unit of time (such as by day, by month, by quarter, by year, or by date). Typically, this means the unit of time is one axis, and the aggregated value is another axis. Querying history and other time-aware tables in this way, with two axes – one as a unit of time, and another as an aggregation – is common in data analytics, and the results are often charted with a line or bar graph.

Below are two example queries for TrackMyBuys that make use of the history table, to give you an idea of what kinds of questions can be asked of a history table.

#### *First History Question: Paid Accounts*

Here is a question useful to the core operation of TrackMyBuys: How many paid accounts created at least 6 months ago have a balance of at least \$20, with the results broken down into three categories for the balance – \$20-\$50, \$50-\$100, \$100+?

First, I explain why this question is useful. The answer can be used to determine how many people who have had paid accounts for a while are choosing not to pay their balance. Perhaps I want to reach out to them and encourage them to pay their balance. Perhaps I want to consider turning their account over to

a credit agency. The categories help me see how much the carried balances are for, whether something small or something more egregious, so I can better make my decision as to how to handle it.

Here is a screenshot of the query I use.

```
--This query answers this question:
--How many paid accounts created at least 6 months ago have a balance of at least $20, with
--the results broken down into three categories for the balance - $20-$50, $50-$100, $100+?

SELECT CASE
    WHEN PaidAccount.AccountBalance >= 20 AND PaidAccount.AccountBalance <= 50 THEN '$20-$50'
    WHEN PaidAccount.AccountBalance >= 50 AND PaidAccount.AccountBalance <= 100 THEN '$50-$100'
    ELSE '$100+'
END As Category,
Count(*) as NumberWithBalance
FROM Account
JOIN PaidAccount ON PaidAccount.AccountID = Account.AccountID
WHERE Account.CreatedOn <= DATEADD(m, -6, GETDATE())
AND PaidAccount.AccountBalance >= 20
GROUP BY CASE
    WHEN PaidAccount.AccountBalance >= 20 AND PaidAccount.AccountBalance <= 50 THEN '$20-$50'
    WHEN PaidAccount.AccountBalance >= 50 AND PaidAccount.AccountBalance <= 100 THEN '$50-$100'
    ELSE '$100+'
END
```

100 %

Results Messages

	Category	NumberWithBalance
1	\$100+	1
2	\$20-\$50	1
3	\$50-\$100	1

To get the results, I join the Account to the PaidAccount table, limit the results to those with balances of at least \$20, and then use the DATEADD function to additionally limit the results to accounts created at least 6 months ago. I use a case statement to categorize the balances into \$20-\$50, \$50-\$100, and \$100+.

To help prove that the query is working properly, I show the full contents of the Account and PaidAccount tables with a simple query.

```
SELECT *
FROM Account
JOIN PaidAccount ON PaidAccount.AccountID = Account.AccountID
```

100 %

Results Messages

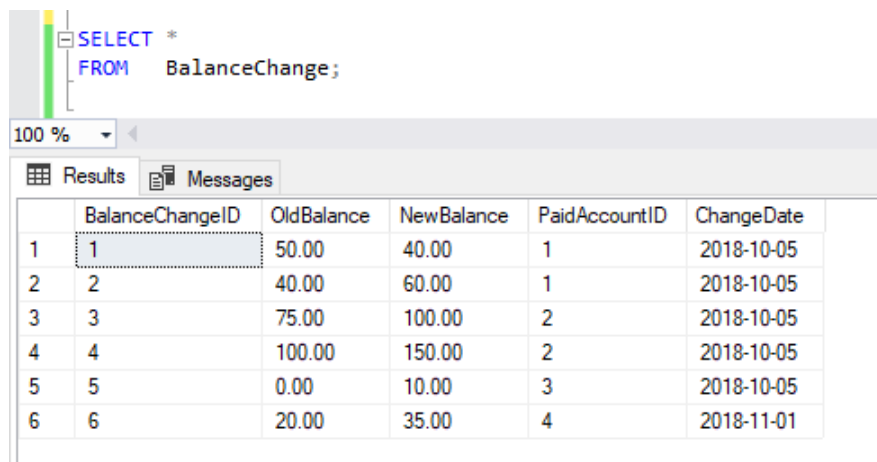
	AccountID	AccountUsername	EncryptedPassword	FirstName	LastName	Email	CreatedOn	AccountType	AccountID	AccountBalance	RenewalDate
1	1	dgordon	xyz	Decker	Gordon	decker.gordon@gmail.com	2018-03-03	F	1	23.99	2019-03-03
2	2	dpopula	xyz	Dolores	Populo	dpopulo@gmail.com	2018-02-02	F	2	110.00	2019-02-02
3	3	xmargie	xyz	Xenith	Margie	mzenith@gmail.com	2018-09-09	F	3	0.00	2019-09-09
4	4	gglass	xyz	Guile	Glass	gguile@gmail.com	2017-09-13	F	4	15.00	2018-09-13
5	5	SSmall	xyz	Sally	Small	ssmall@gmail.com	2017-04-06	F	5	55.00	2018-04-06

Upon inspection, you see that there are 5 paid accounts in my database. Only four of them were created at least 6 months ago (assuming the date in questions is September 28, 2018). Xenith Margie's account was created on September 9<sup>th</sup>, 2018, so it is excluded. Out of those four, only three of them have a balance of at least \$20. Guile Glass's account only has a balance \$15, so that is excluded. Last, you can see an even distribution across categories. Decker Gordon's account has a balance of \$23.99, so falls into the \$20-\$50 category. Dolores Populo's account has a balance of \$110, so falls into the \$100+ category. Sally Small's balance is \$55, so falls into the \$50-\$100 category. So as is demonstrated, the query appears to be returning the correct results based upon the question.

### Second History Question: Average Balance

Another useful question from the BalanceChange history table is: What was the average change in balance for the month of October, 2018? This question is useful because I may want get an idea of the frequency or magnitude of changes in balance for any month (in this case, October). This will help me to know if people's balances are growing or shrinking and by how much.

In order to provide more data for this, I added a couple more accounts and changed their balances a few times. Here is what the BalanceChange table looks like after these changes.



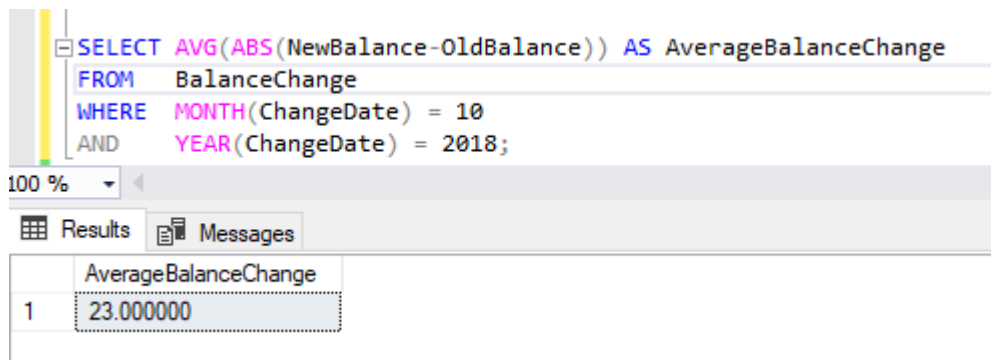
The screenshot shows a SQL query window with the following query:

```
SELECT *
FROM BalanceChange;
```

Below the query, the results are displayed in a table with 6 columns: BalanceChangeID, OldBalance, NewBalance, PaidAccountID, and ChangeDate. The results are as follows:

	BalanceChangeID	OldBalance	NewBalance	PaidAccountID	ChangeDate
1	1	50.00	40.00	1	2018-10-05
2	2	40.00	60.00	1	2018-10-05
3	3	75.00	100.00	2	2018-10-05
4	4	100.00	150.00	2	2018-10-05
5	5	0.00	10.00	3	2018-10-05
6	6	20.00	35.00	4	2018-11-01

There are five balance changes that happened in October 2018, and one in November 2018. Only the October ones should be picked up in the query. Here a screenshot of the query and its results.



The screenshot shows a SQL query window with the following query:

```
SELECT AVG(ABS(NewBalance-OldBalance)) AS AverageBalanceChange
FROM BalanceChange
WHERE MONTH(ChangeDate) = 10
AND YEAR(ChangeDate) = 2018;
```

Below the query, the results are displayed in a table with 1 column: AverageBalanceChange. The result is as follows:

AverageBalanceChange
23.000000

Here is an explanation of the query.

CODE	DESCRIPTION
------	-------------

<b>SELECT</b> AVG (ABS (NewBalance-OldBalance)) AS AverageBalanceChange	This subtracts the new balance from the old balance for each row, takes the absolute value of that to get rid of negative numbers (in case the balance goes up), then takes the average of that. This is how the average change of balance is obtained.
<b>FROM</b> BalanceChange <b>WHERE</b> MONTH (ChangeDate) = 10 <b>AND</b> YEAR (ChangeDate) = 2018;	This obtains rows from the BalanceChange table that are for the month of October in the year 2018. This ensures only the desired rows are included.

## Summary and Reflection

Take a moment to reflect on all you have learned and accomplished. You formally designed your own database from scratch using universally understood language and diagrams. You created your database in SQL, the universal language for relational databases, and wrote transactions against it to accomplish useful tasks for your organization or application. You indexed your database to help it perform well. You added data history, and answered useful questions from your data with queries, and told data stories with data visualizations. And you did all of this by learning and applying many best practices for modern database design and implementation. What an amazing accomplishment!

Perhaps even more importantly, as databases are used directly or indirectly by virtually every person and organization in the world, you have started to develop an incredibly in-demand and lucrative skillset. You can undoubtedly utilize these skills in your current work, or choose to pursue this further and see where it takes your career. As you use these skills to solve real problems people and organizations are facing, you will be noticed, and your unique skills will stand out.

While this iteration represents your final chance to improve upon your design and implementation in the course, this doesn't need to be the end of your database. You can continue to tweak and expand your database to meet the needs of your organization or application after the course is over.

Update your project summary to reflect your new work on data history. Write down your reflections on your database and all that you've learned. If you're proud of something, don't hesitate to point it out! Your instructor or facilitator will be thrilled to know your thoughts.

Here is an updated summary as well as some reflections I have about TrackMyBuys.

### TrackMyBuys Reflection

My database is for a mobile app named TrackMyBuys which records purchases made across all stores, making it the one stop for any purchase history. Typically, when a person purchases something, they can only see their purchase history with that same vendor, and TrackMyBuys seeks to provide a single interface for all purchases. The database must support a person entering, searching, and even analyzing their purchases across all stores.

The structural database rules and conceptual ERD for my database design contain the important entities of Store, Product, Purchase, and Account, as well as relationships between them. The design contains a hierarchy of Purchase/FaceToFacePurchase and Purchase/OnlinePurchase to reflect the two primary ways people purchase products. The design also contains a hierarchy of Account/PaidAccount and Account/FreeAccount to reflect the fact that people can sign up for a free account or a paid account for TrackMyBuys. The DBMS physical ERD contains the same

entities and relationships, uses the best practice of synthetic keys, and contains the important attributes needed by the database to support the application.

The SQL script that contains all table creations that 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 TrackMyBuys have been identified, and implemented with SQL queries. Some useful visualizations have been presented, along with the stories they tell.

As I reflect on the database and my accomplishments, I can say it's been a long but rewarding road. It is amazing to see a real database in action that can be hooked up to the mobile application TrackMyBuys. I can envision many of the screens already, and should I choose to make this a real Android or iPhone application, a good portion of the database is already implemented. I won't be slowed down by the database and can just hook up the application to it. I can see there is still more to develop in my database, but feel it's a solid foundation to move forward with.

---

## Items to Submit

In summary, for this iteration, you revise your design one final time, track a history for one or more columns with a table and a trigger, and use the data to provide effective visualizations and data stories. Make sure to use the template provided with this iteration to ensure you are submitting all necessary items.

**MET CS 669 Database Design and Implementation for Business**  
**Term Project Iteration 6**

## Evaluation

Your iteration will be reviewed by your facilitator or instructor with the criteria outlined in the table below. Note that the grading process:

- involves the grader assigning an appropriate letter grade to each criterion.
- uses the following letter-to-number grade mapping – A+=100,A=96,A-=92,B+=88,B=85,B-=82,C+=88,C=85,C-=82,D=67,F=0.
- provides an overall grade for the submission based upon the grade and weight assigned to each criterion.
- allows the grader to apply additional deductions or adjustments as appropriate for the submission.
- applies equally to every student in the course.

Aspect	What is Measured	A+ Excellent	B Good	C Fair/Satisfactory	D Insufficient	F Failure	Letter Grade
<b>History Table Implementation (20%)</b>	This is a measure of the soundness and usefulness of the history table implementation. Excellent solutions exhibit all of the following properties. The history table is entirely useful given the needs of the database. All attributes necessary to track history correctly for the identified field(s) are present, and no unnecessary attributes are present. The trigger demonstrably maintains correct history table records.	Entirely sound Entirely useful	Mostly sound Mostly useful	Somewhat sound Somewhat useful	Mostly unsound Mostly useless	History table and trigger missing or Entirely unsound Useless	A+
<b>History Table Design (10%)</b>	This is a measure of how accurately the database design depicts the history table implementation, and how much the implementation agrees with the design. With excellent solutions, the history table implementation is depicted entirely accurately in the database design, and entirely agrees with the relevant use cases, structural database rules, conceptual ERD, and DBMS physical ERD.	Entirely accurate Fully agrees	Mostly accurate Mostly agrees	Somewhat accurate Some agreement	Mostly inaccurate Mostly disagrees	History table design missing or Entirely inaccurate Entirely disagrees	A+
<b>Results for Visualizations (10%)</b>	This is a measure of how useful the SQL results are given the needs of the database, and the quantity of queries provided. Excellent solutions provide two entirely useful queries.	Entirely useful Two queries	Mostly useful Two queries	Somewhat useful One query	Mostly useless One query provided	SQL results missing or Entirely useless	A+

<b>Data Visualization Presentation (10%)</b>	This measures the accuracy and clarity of the data visualizations, and the quantity of visualizations provided. Excellent solutions provide two data visualizations which present the SQL results entirely accurately, are labeled well, use appropriate ranges, are legible and organized, and are clearly understood.	Entirely accurate Entirely clear Two visualizations	Mostly accurate Mostly clear Two visualizations	Somewhat accurate Somewhat clear One visualization	Mostly inaccurate Mostly unclear One visualization	The data visualizations are missing or Entirely inaccurate Entirely unclear	A+
<b>Data Stories (10%)</b>	This measures the accuracy and clarity of the data stories, and the quantity of stories provided. Excellent solutions have two data stores which are entirely clear and useful, are organized well, and accurately describe the data and visualizations.	Entirely accurate Entirely clear Two data stories	Mostly accurate Mostly clear Two data stories	Somewhat accurate Somewhat clear One data story	Mostly inaccurate Mostly unclear One data story	The data stories are missing or Entirely inaccurate Entirely unclear	A+
<b>Overall Presentation (20%)</b>	This is a measure of how well your choices are supported with explanations, as well as the quality of your documentation organization and presentation.	Excellent support Well organized and presented	Good support Organized and presentable	Partial support Somewhat organized and presented	Mostly unsupported Mostly disorganized presentation	No explanations Entirely disorganized presentation	A+
<b>Prior Work (20%)</b>	This measures how well any issues from prior iterations have been improved in order to provide a frame of reference for this iteration.	Completely improved or No improvement necessary	Mostly improved	Somewhat improved	Mostly not improved	No improvements	A+
<b>Preliminary Grade:</b>		<b>Entities Deduction:</b> At least 10 required at the conceptual level At least two subtypes required 3 point deduction for each missing		<b>Lateness Deduction:</b> 5 points per day 4 days maximum Contact your facilitator for any exceptions		<b>Iteration Grade:</b>	

Use the **Ask the Teaching Team Forum** if you have any questions regarding how to approach this iteration. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.