

Contents

Iteration Introduction	2
Transaction Driven, Reusable Stored Procedures.....	4
Add Ferrari Stored Procedure (Oracle)	5
Execute Ferrari Stored Procedure (Oracle)	5
Add Ferrari Stored Procedure (SQL Server)	6
Execute Ferrari Stored Procedure (SQL Server)	6
Add Ferrari Stored Procedure (Postgres)	6
Execute Ferrari Stored Procedure (Postgres)	7
Implementing Transactions in your Database	7
TrackMyBuys Transaction	7
Maintaining History Tables with Triggers	8
Conceptual Car ERD with Price Change	10
DBMS Physical ERD With Price Change.....	11
PriceChange Table Creation	12
Price Changes for SQL Server	12
Price Change Trigger for SQL Server	12
Price Changes for Oracle	14
Price Change Trigger for Oracle	14
Price Changes for Postgres.....	15
Price Change Trigger for Postgres.....	16
Capturing History for your Project.....	18
TrackMyBuys History.....	18
Organization-Driven Queries	22
Number Available By Make Query	22
Creating Questions and Queries for your Database	22
TrackMyBuys Question and Query	23
Summary and Reflection.....	26
TrackMyBuys Reflection.....	26
Items to Submit.....	27
Evaluation.....	29

Iteration Introduction

This iteration is about putting data into your database and answering questions from the data. Your design structure was implemented in Iteration 4 through creation of your tables, attributes, and constraints. In this iteration, you insert data transactionally with stored procedures, implement a history table, maintain it with a trigger, define questions useful to the organization or application, and answer them with well-written queries. After your hard work, you get to see your database in action!

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>Specialization-Generalization Relationships</i> – You add one or more specialization-generalization relationships, which allows one entity to specialize an abstract entity, to your structural database rules and ERD.</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>Tables and Constraints</i> – You create your tables and constraints in SQL.</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 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>History Table</i> – You create a history table to track changes to values, and develop a trigger to maintain it.</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>

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

Transaction Driven, Reusable Stored Procedures

Recall that a stored procedure is a named code segment that can be executed by name as needed. Stored procedures can contain SQL code and business logic as needed. Stored procedures can take parameters whereby the caller specifies the values when executing the procedure. Stored procedures provide many useful features.

Also recall that a transaction is a logical unit of work consisting of a series of steps for the purpose of accomplishing a task. Organizations group work together in a transaction to transition the database from one consistent state to the next, in a way that is atomic, consistent, and durable (since transactions obey the ACIDS requirements). Transactions are the backbone of modern relational database processing.

Some organizations use a method whereby transactions are implemented in reusable stored procedures. Such a stored procedure takes one or more parameters, and completes all of the steps in the transaction using the values the caller specified in the parameters. Use of this method allows database developers to implement the transactions in the database, eliminating the need for the application to implement all of the steps. Instead, the application executes the stored procedure for the appropriate transaction with the correct parameters.

There are other advantages to using this method. For one, it's not necessary for the application to connect over the network repeatedly to execute SQL, resulting in better performance. For another, transaction steps can be updated without the need to redeploy the application. For another, I.T. staff can execute a transaction manually when necessary to correct an issue, or for other purposes, such as migrating data from another source. This method carries many advantages.

Let's look at a simple example with the Car database we've been developing in this iteration. Imagine for the Car database there is the following simple use case, for when a Ferrari is ready to be entered into this system.

Add Ferrari Use Case

1. The car reseller purchases a used Ferrari to sell.
2. The car salesperson enters the Ferrari's information into the system including its make, model, and other important information.

What kind of transaction would support this use case? Simple! We know that we have two tables – Car and Ferrari – that are relevant to Ferraris, so we need two insert statements for each of those tables in our transaction. What about tying this into a stored procedure? Again, simple! We give the stored procedure parameters for every field in both of those tables.

For Oracle, the transaction would be implemented in a stored procedure as follows.

Add Ferrari Stored Procedure (Oracle)

```
CREATE OR REPLACE PROCEDURE AddFerrari(CarID IN DECIMAL, VIN IN VARCHAR, Price IN DECIMAL,
    Color IN VARCHAR, Make IN VARCHAR, Model IN VARCHAR, Mileage IN DECIMAL)
AS
BEGIN
    INSERT INTO Car(CarID, VIN, Price, Color, Make, Model, Mileage)
    VALUES(CarID, VIN, Price, Color, Make, Model, Mileage);

    INSERT INTO Ferrari(CarID)
    VALUES(CarID);
END;
```

Notice that the stored procedure takes a parameter for every column, then inserts those values into the Car and Ferrari tables.

We would execute the stored procedure in Oracle as follows.

Execute Ferrari Stored Procedure (Oracle)

```
BEGIN
    AddFerrari(1, '1XPADB9X4TN402579', 295000, 'Giallo Modena',
    'Ferrari', 'F12berlinetta', 20000);
COMMIT;
END;
```

We use a BEGIN/END block, then pass in the parameter values we are interested in to execute the stored procedure. After calling the stored procedure (which will implicitly begin a transaction), we issue the COMMIT keyword to commit the results of the transaction. In Oracle, the first SQL statement encountered implicitly starts the transaction, and then the COMMIT statement commits the active transaction.

I chose a fictional, but realistic, example of a Ferrari that the reseller may sale. The values are indicated below.

Parameter	Value
CarID	1
VIN	1XPADB9X4TN402579
Price	\$295,000
Color	Giallo Modena
Make	Ferrari
Model	F12berlinetta
Mileage	20,000

For SQL Server, the transaction would be implemented in a stored procedure as follows.

Add Ferrari Stored Procedure (SQL Server)

```
CREATE PROCEDURE AddFerrari @CarID DECIMAL(12), @VIN VARCHAR(17), @Price DECIMAL(8,2),
@Color VARCHAR(64), @Make VARCHAR(64), @Model VARCHAR(64), @Mileage DECIMAL(7)
AS
BEGIN
    INSERT INTO Car(CarID, VIN, Price, Color, Make, Model, Mileage)
    VALUES(@CarID, @VIN, @Price, @Color, @Make, @Model, @Mileage);

    INSERT INTO Ferrari(CarID)
    VALUES(@CarID);
END;
```

Just as with Oracle, the stored procedure takes a parameter for every column, then inserts those values into the Car and Ferrari tables.

We would execute the stored procedure in SQL Server as follows.

Execute Ferrari Stored Procedure (SQL Server)

```
BEGIN TRANSACTION AddFerrari;
EXECUTE AddFerrari 1, '1XPADB9X4TN402579', 295000,
    'Giallo Modena', 'Ferrari', 'F12berlinetta', 20000;
COMMIT TRANSACTION AddFerrari;
```

The BEGIN TRANSACTION block starts a transaction, named “AddFerrari”. The EXECUTE block executes the AddFerrari stored procedure and passes in the same fictional parameters previously defined. The COMMIT TRANSACTION block commits the transaction.

For Postgres, the transaction would be implemented in a function as follows.

Add Ferrari Stored Procedure (Postgres)

```
CREATE OR REPLACE FUNCTION AddFerrari(CarID IN DECIMAL, VIN IN VARCHAR, Price IN DECIMAL,
    Color IN VARCHAR, Make IN VARCHAR, Model IN VARCHAR, Mileage IN DECIMAL)
RETURNS VOID
AS
$proc$
BEGIN
    INSERT INTO Car(CarID, VIN, Price, Color, Make, Model, Mileage)
    VALUES(CarID, VIN, Price, Color, Make, Model, Mileage);

    INSERT INTO Ferrari(CarID)
    VALUES(CarID);
END;
$proc$ LANGUAGE plpgsql
```

Just as with the other databases, the stored procedure takes a parameter for every column, then inserts those values into the Car and Ferrari tables.

We would execute the stored procedure in Postgres as follows.

Execute Ferrari Stored Procedure (Postgres)

```
START TRANSACTION;  
DO  
$$BEGIN  
    EXECUTE AddFerrari(1, '1XPADB9X4TN402579', 295000, 'Giallo Modena',  
        'Ferrari', 'F12berlinetta', 20000);  
END$$;  
COMMIT TRANSACTION;
```

The START TRANSACTION block starts a transaction. The DO command along with the EXECUTE command executes the AddFerrari stored procedure and passes in the same fictional parameters previously defined. The COMMIT TRANSACTION block commits the transaction.

Note that the stored procedures provided in this section have the basic transaction steps, but do not implement error checking and other features that a more robust implementation would implement.

In summary, one methodology for implementing a transaction is to implement it in a parameterized stored procedure, execute the stored procedure with the necessary values, and surround the store procedure calls with transaction control statements to start and commit the transaction.

Implementing Transactions in your Database

Now that you know a common methodology for implementing transactions, you can take advantage of this to start populating your database with data. Select three of your use cases that involve adding data to the database, create parameterized stored procedures that implement the transactions steps, and execute the stored procedures in the context of a transaction. Provide screenshots of their creation and execution, and also attach a SQL script with them attached.

Below is a sample of one such implementation with TrackMyBuys. Note that only one example is provided in TrackMyBuys for illustrative purposes, but you are asked to create three.

TrackMyBuys Transaction

The first use case for TrackMyBuys 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 either a free or paid account when its first run.
3. The user selects the type of account and 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.

For this use case, I will implement a transaction that creates a free account, using SQL Server.

Here is a screenshot of my stored procedure definition.

```

CREATE PROCEDURE AddFreeAccount @AccountID DECIMAL(12), @AccountUsername VARCHAR(64), @EncryptedPassword VARCHAR(255),
@FirstName VARCHAR(255), @LastName VARCHAR(255), @Email VARCHAR(255)
AS
BEGIN
    INSERT INTO Account(AccountID, AccountUsername, EncryptedPassword,
    FirstName, LastName, Email, CreatedOn, AccountType)
    VALUES(@AccountID, @AccountUsername, @EncryptedPassword,
    @FirstName, @LastName, @Email, GETDATE(), 'F');

    INSERT INTO FreeAccount(AccountID)
    VALUES(@AccountID);
END;
go

```

I name the stored procedure “AddFreeAccount”, and give it parameters that correspond to the Account and FreeAccount tables. Since CreatedDate is always the current date, I do not need a parameter for that, but instead use the GETDATE() function in SQL Server. Since this procedure is always for a free account, I do not use a parameter for AccountType, but hardcode the character “F”.

Inside the stored procedure, there are two insert statements to insert into the two respective tables.

Here is a screenshot of my stored procedure execution.

```

BEGIN TRANSACTION AddFreeAccount;
EXECUTE AddFreeAccount 1, 'dgordon', 'xyz', 'Decker', 'Gordon', 'decker.gordon@gmail.com';
COMMIT TRANSACTION AddFreeAccount;

```

Messages

(1 row affected)

(1 row affected)

I add a fictional person named Decker Gordon with other fictional but realistic information. I nested the stored procedure call between transaction control statements to ensure the transaction is committed.

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.

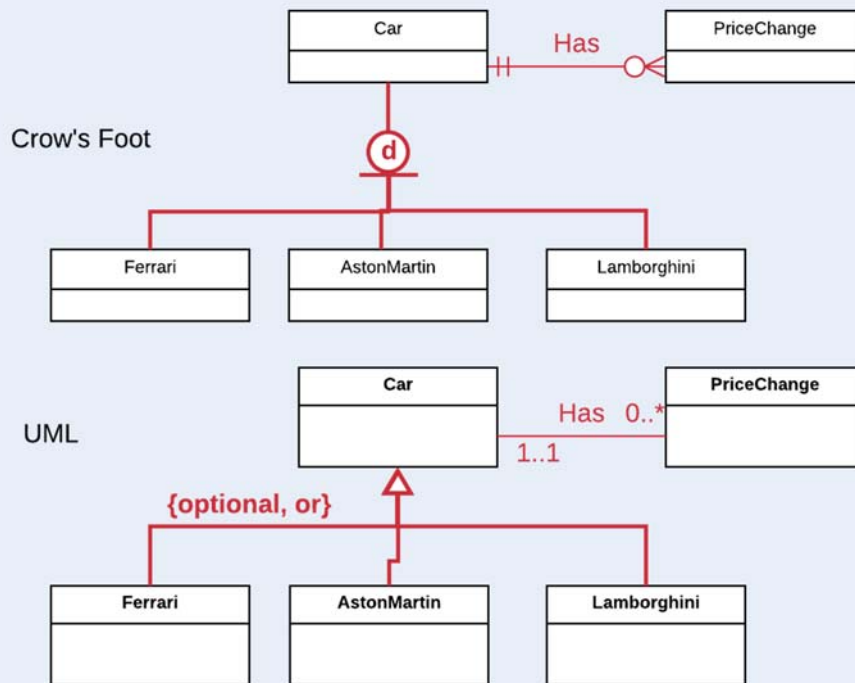
A standard history table contains the old and new value(s) for the column(s) being tracked, a foreign key to the table being tracked, and the date of the change. With such a history table, it is possible to see when the value(s) changed, and when, which is quite useful for analyzing the data over time.

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.

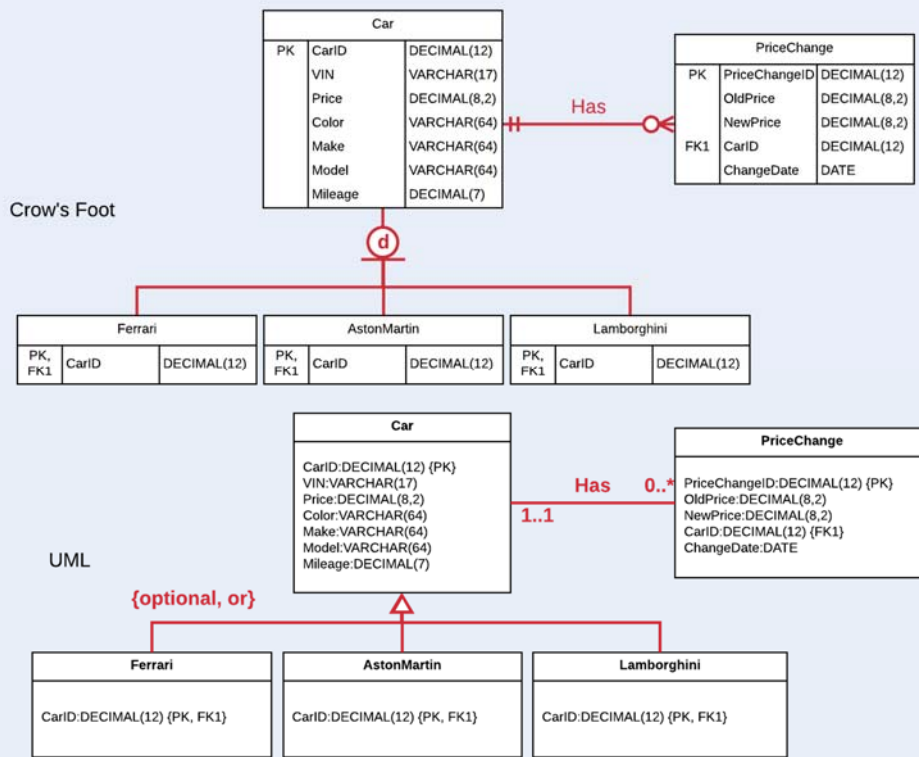
Conceptual Car ERD with Price Change



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 Car entity, we would use the following CREATE TABLE statement.

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));
```

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

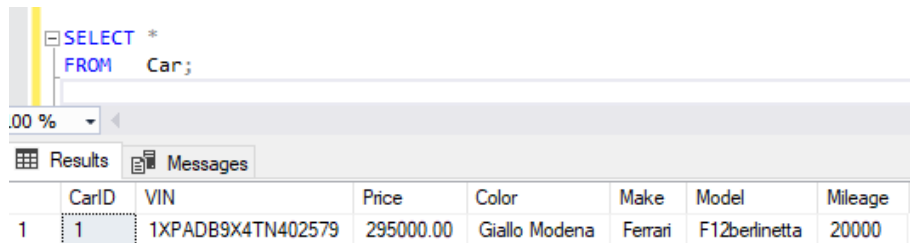
```
CREATE TRIGGER PriceChangeTrigger  
ON Car  
AFTER UPDATE  
AS  
BEGIN  
    DECLARE @OldPrice DECIMAL(8,2) = (SELECT Price FROM DELETED);  
    DECLARE @NewPrice DECIMAL(8,2) = (SELECT Price FROM INSERTED);  
  
    IF (@OldPrice <> @NewPrice)  
        INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate)  
        VALUES(ISNULL((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1),  
            @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,	This is the insert statement that records the price change by adding a

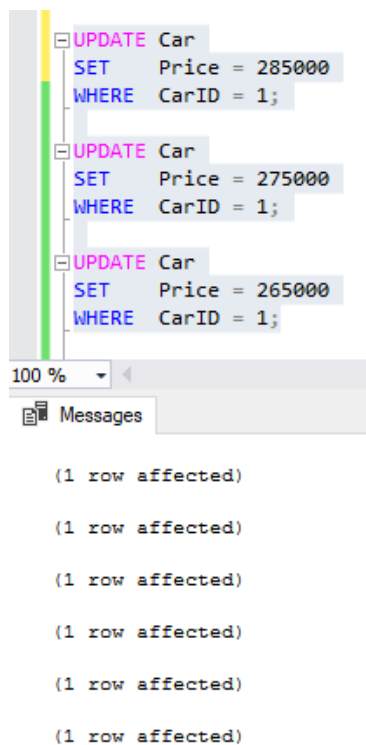
<pre> OldPrice, NewPrice, CarID, ChangeDate) VALUES(ISNULL((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1), @OldPrice, @NewPrice, (SELECT CarID FROM INSERTED), GETDATE()); </pre>	row into the PriceChange table. The PriceChangeID column is generated by getting the maximum PriceChangeID + 1 if one exists, or 1 otherwise (production systems would likely use an IDENTITY column or a sequence). 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.
<pre> END; </pre>	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	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;

```

(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.

	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(NVL((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1),
           :OLD.Price,
           :NEW.Price,
           :New.CarID,
           trunc(sysdate));
END;
```

Let's work through it line by line.

CODE	DESCRIPTION
CREATE OR REPLACE TRIGGER PriceChangeTrigger BEFORE UPDATE OF Price ON Car	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.
FOR EACH ROW BEGIN	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.
INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate) VALUES(NVL((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1), :OLD.Price, :NEW.Price, :New.CarID, trunc(sysdate));	This is the insert statement that records the price change by adding a row into the PriceChange table. The PriceChangeID column is generated by getting the maximum PriceChangeID + 1 if one exists, or 1 otherwise (production systems would likely use a 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.
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.

```
SELECT *  
FROM Car;
```

Script Output x Query Result x

SQL | All Rows Fetched: 1 in 0.012 seconds

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.

```
UPDATE Car
SET Price = 285000
WHERE CarID = 1;

UPDATE Car
SET Price = 275000
WHERE CarID = 1;

UPDATE Car
SET Price = 265000
WHERE CarID = 1;
```

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.

```
SELECT *
FROM PriceChange;
```

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

Price Change Trigger for Postgres

```
CREATE OR REPLACE FUNCTION PriceChangeFunction()
RETURNS TRIGGER LANGUAGE plpgsql
AS $strigfunc$
BEGIN
    INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate)
    VALUES(COALESCE((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1),
            OLD.Price,
            NEW.Price,
            New.CarID,
            current_date);

    RETURN NEW;
END;
$strigfunc$;

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
CREATE OR REPLACE FUNCTION PriceChangeFunction() RETURNS TRIGGER LANGUAGE plpgsql	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.
AS \$strigfunc\$ BEGIN	This is part of the syntax starting the function block.
INSERT INTO PriceChange(PriceChangeID, OldPrice, NewPrice, CarID, ChangeDate) VALUES(COALESCE((SELECT MAX(PriceChangeID)+1 FROM PriceChange), 1), OLD.Price, NEW.Price, New.CarID, current_date);	This is the insert statement that records the price change by adding a row into the PriceChange table. The PriceChangeID column is generated by getting the maximum PriceChangeID + 1 if one exists, or 1 otherwise (production systems would likely use a 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.
RETURN NEW; END; \$strigfunc\$	This ends the function definition.
CREATE TRIGGER PriceChangeTrigger BEFORE UPDATE OF Price ON Car FOR EACH ROW	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.
EXECUTE PROCEDURE PriceChangeFunction();	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

```

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

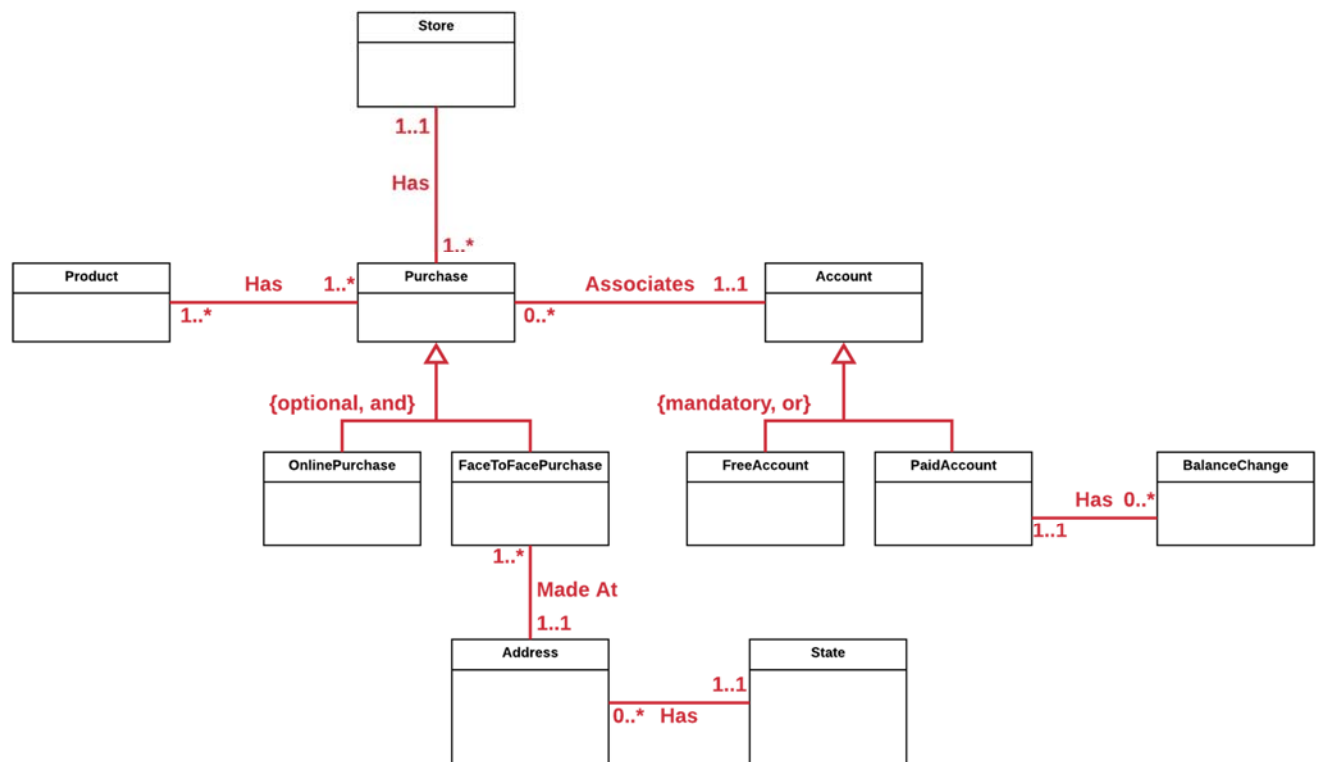
Select a column or columns that would benefit from a record of historical values in your database. Go through the process of designing a new history table by creating a new structural database rule and updating your ERDs. Then create the history table in SQL and define a trigger that maintains the history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

Here is the work for TrackMyBuys.

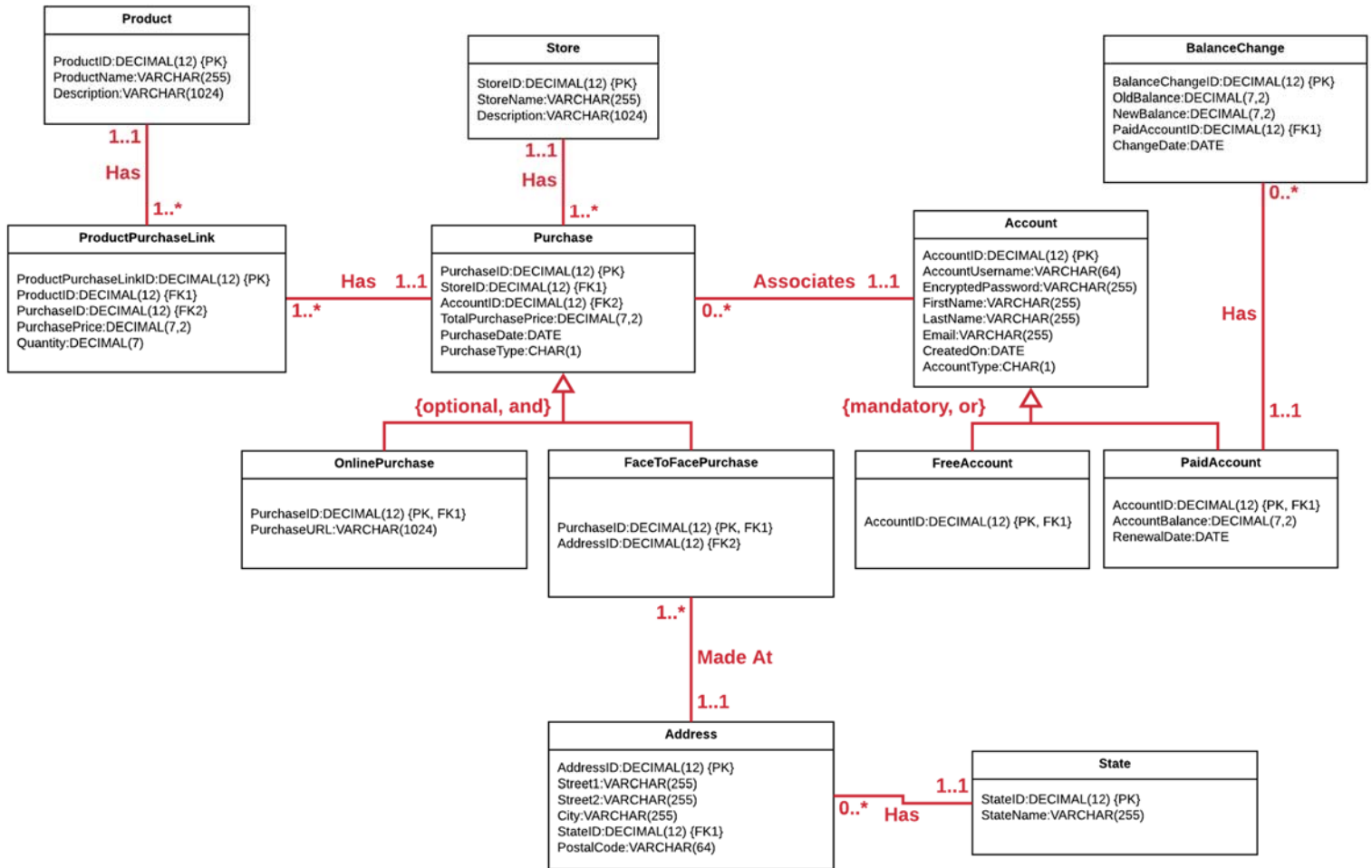
TrackMyBuys History

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
BalanceChangeID	This is the primary key of the history table. It is a DECIMAL(12) to allow for many values.
OldBalance	This is the balance of the account before the change. The datatype mirrors the Balance datatype in the PaidAccount table.
NewBalance	This is the balance of the account after the change. The datatype mirrors the Balance datatype in the PaidAccount table.
PaidAccountID	This is a foreign key to the PaidAccount table, a reference to the account that had the change in balance.
ChangeDate	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.

```
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));
```

100 %

Messages

Commands completed successfully.

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

```
CREATE 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(ISNULL((SELECT MAX(BalanceChangeID)+1 FROM BalanceChange), 1),
            @OldBalance,
            @NewBalance,
            (SELECT AccountID FROM INSERTED),
            GETDATE());
END;
```

100 %

Messages

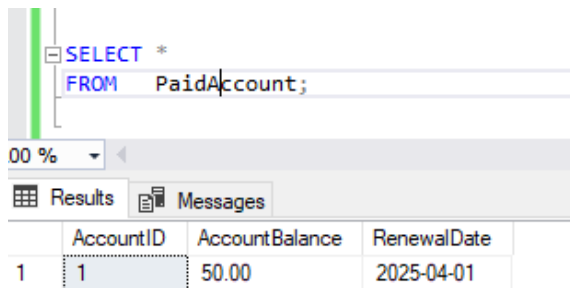
Commands completed successfully.

I explain it here line by line.

CODE	DESCRIPTION
CREATE 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 obtaining one over the next highest. The old and

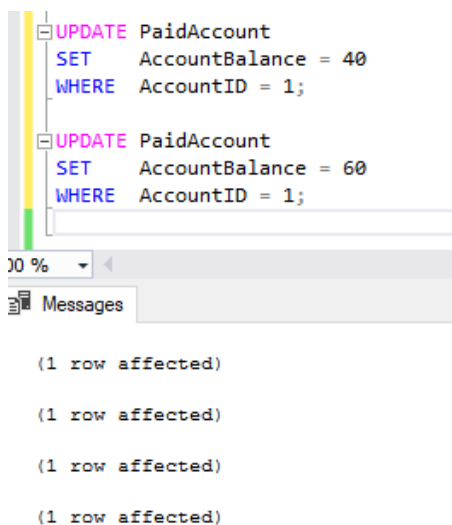
ChangeDate) VALUES(ISNULL((SELECT MAX(BalanceChangeID)+1 FROM BalanceChange), 1), @OldBalance, @NewBalance, (SELECT AccountID FROM INSERTED), GETDATE());	new balances are used from the variables. The account ID is obtained from the INSERTED pseudo table. The date of the change is obtained by using the 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	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;

```

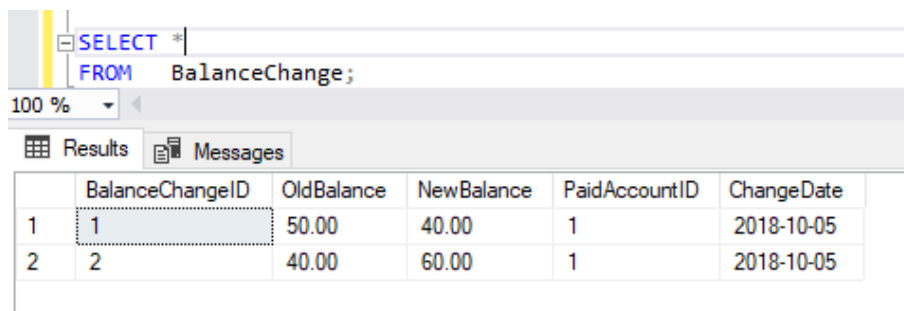
(1 row affected)

(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	1	50.00	40.00	1	2018-10-05
2	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

Organization-Driven Queries

Storing the data in a database has little value if that's where it ends; what's important is using the data. Queries are king in a relational database because they are the tool for pulling out relevant information and using it. While one could write queries that are theoretically beautiful or demonstrate powerful SQL constructs, that's not a useful goal for an organization. An organization needs queries that get it the data it needs. The beauty and the power are merely tools that help accomplish the goal. Therefore, just as with all of our design work, we focus on writing queries that are based on what the organization needs and how the database will be used. "Useful to the organization" is a broad concept, but for purposes of this project, we'll create queries based upon questions we deem useful for the organization and/or application.

Let's look at an example, again from our Car database. A question that is a reasonable use of the system for the car reseller is: how many cars of each make are available? Such a question would be commonly asked because customers and staff alike will want to know the answer to this question. Perhaps a customer only wants to look at Aston Martins so wants to know the number available. Perhaps a customer hasn't decided on a make and wants to have an idea of how many the reseller has of each before visiting the showroom. Perhaps a manager wants to know the answer before purchasing more cars. It's obvious that the answer to this question has many useful applications.

To answer this question, we write a query that takes advantage of aggregation.

Number Available By Make Query

```
SELECT  Make, Count(*) AS NumberAvailable
FROM    Car
GROUP BY Make
ORDER BY Make;
```

This query selects from the Car table, groups and orders by the make column, and counts the number available for each make. The query is simple; however, we could envision many variations on this. For example, by ordering by the NumberAvailable column instead of Make, we could find out a ranking from most available to least available.

Creating Questions and Queries for your Database

Create three questions useful to your organization or application, then write queries to address the questions. *One of those questions should require data from your history table.* To ensure sufficient complexity, each of the queries should contain at least two of the constructs below. There should be at least one from each group.

Group 1 (choose one or more)

- joins of at least two tables.
- one or more restrictions in the WHERE clause.
- an order by statement.

Group 2 (choose one or more)

- at least one aggregate function.
- at least one subquery.
- a having clause.
- a left or right join.
- joins of four or more tables.
- a union of two queries.

If you'd like to use a SQL construct not listed here, that's great, but just run the idea by your facilitator or instructor to ensure it will meet the complexity requirements. The usefulness and complexity of the queries will be a factor in the evaluation of this section. Try to focus on the most useful questions with reasonable complexity requirements.

Add your queries to your SQL script. Each query should have an associated comment which describes the question it's answering. Also provide screenshots of the execution of each query and its results in your iteration.

In order to show useful results, you will need enough rows and variety in your tables. For example, if you are limiting by a certain column, some rows should match the condition, and some rows should not match. If you are using aggregates, there should be enough rows to help prove out the aggregate, and the grouped by column should have more than one distinct value. You want to help prove that your query works by providing useful data.

Make sure to explain the logic of your query and how it shows the expected results.

Below I give an example of two questions useful to TrackMyBuys, one of which is based upon the new history table. Note that these two are provided for illustrative purposes, but you should provide three.

TrackMyBuys Question and Query

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
```

	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
```

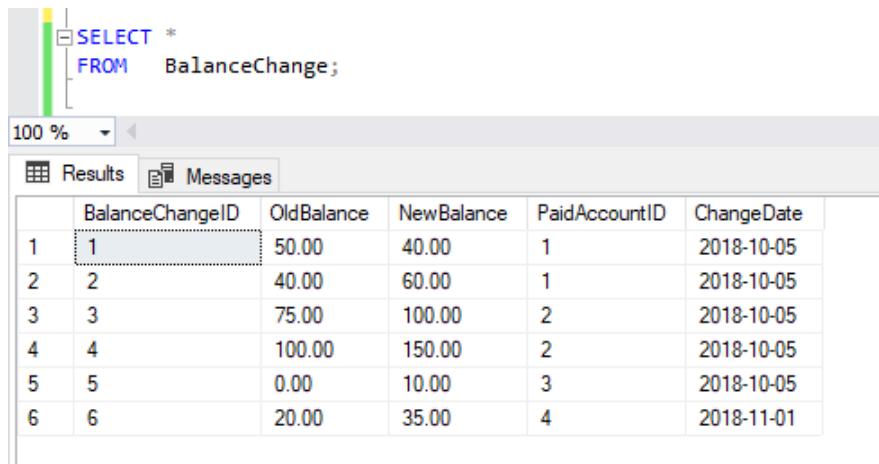
	AccountID	AccountUsername	EncryptedPassword	FirstName	LastName	Email	CreatedOn	AccountType	AccountID	AccountBalance	RenewalDate
1	1	dgordon	xyz	Decker	Gordon	decker.gordon@gmail.com	2018-03-03	F	1	23.99	2019-03-03
2	2	dpopula	xyz	Dolores	Populo	dpopulo@gmail.com	2018-02-02	F	2	110.00	2019-02-02
3	3	xmargie	xyz	Xenith	Margie	mzenith@gmail.com	2018-09-09	F	3	0.00	2019-09-09
4	4	gglass	xyz	Guile	Glass	gguille@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 (the date as of this writing is September 28, 2018). Xenith Margie's account was created on September 9th, 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.

A 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.



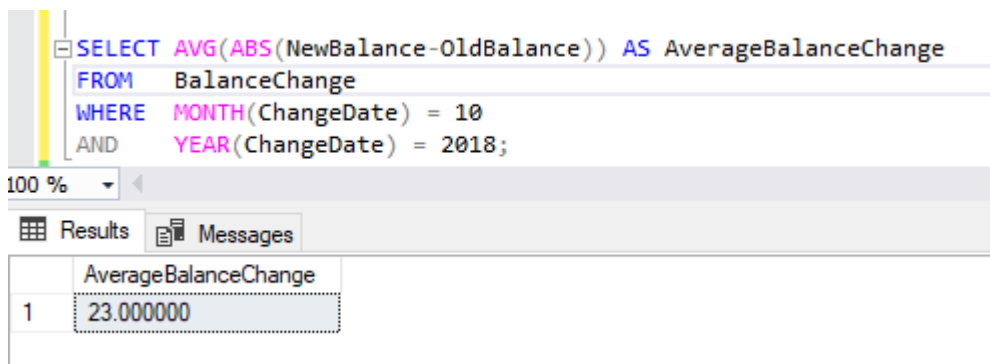
The screenshot shows a SQL query window with the following text:

```
SELECT *  
FROM BalanceChange;
```

Below the query window, the 'Results' tab is active, displaying a table with 6 rows and 6 columns. The columns are: BalanceChangeID, OldBalance, NewBalance, PaidAccountID, and ChangeDate. The data is 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 text:

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

Below the query window, the 'Results' tab is active, displaying a table with 1 row and 1 column. The column is: AverageBalanceChange. The data is as follows:

AverageBalanceChange
23.000000

Here is an explanation of the query.

CODE	DESCRIPTION
SELECT 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.
FROM BalanceChange WHERE MONTH (ChangeDate) = 10 AND 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 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 signup 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. A history of balances has been created as well as a query which tracks changes to balances for a specific month.

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 but 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, create stored procedures that are executed to transactionally add data to your database, track a history for one or more columns with a table and a trigger, identify questions useful to the organization or application, and answer them with queries. Your design document will contain the following items, with items new or partially new to this iteration highlighted.

Component	Description
Project Direction Overview	Revise (if necessary) your 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.
Use Cases and Fields	Revise (if necessary) your use cases that enumerate steps of how the database will be typically used, and the significant database fields needed to support the use case.
Structural Database Rules	Revise (if necessary) your list of structural database rules for all significant entities and relationships, and add support for a history entity.
Conceptual entity-relationship diagram	Revise (if necessary) your conceptual ERD, and add support for a history entity.
DBMS Physical ERD	Revise (if necessary) your DBMS physical ERD, and add support for a history entity.
Index Identification and Creations	Revise (if necessary) the indexes useful to your database.
Stored Procedure Execution and	Provide screenshots and explanations of defining and executing your stored procedures to transactionally add data to your database.

Explanations	
Trigger Creation and Use	Provide a screenshot demonstrating compilation of your history trigger, and demonstrate its execution screenshots that show the history table before and after data changes.
Question Identification and Explanations	Identify questions useful to the organization and explain why they are useful, ensuring one of them involves data from your history table.
Query Executions and Explanations	Answer the questions with queries, provide screenshots, and explain the results.
Summary and Reflection	Revise the concise summary of your project and the work you have completed thus far, and your questions, concerns, and observations.

Your SQL script will contain the following sections.

Script	Description
Tables	Revise (if necessary) the table creations.
Indexes	Revise (if necessary) the script that contains your index creations.
Stored Procedures	Add your stored procedure creations to your script.
Triggers	Add your history trigger creation to your script.
Queries	Add your queries to your script.

Evaluation

Your iteration will be reviewed by your facilitator or instructor with the following criteria and grade breakdown.

Criterion	A	B	C	D	F	Letter Grade
Technical mastery (50%)	Evidence of excellent mastery throughout	Evidence of good mastery throughout	Evidence of basic mastery throughout or good mastery intermittently	Minimal mastery evidenced	Virtually no mastery evidenced	
Depth and thoroughness of coverage (25%)	Excellent depth and coverage of significant topics and issues	Good depth and coverage of significant topics and issues	Basic depth and coverage of significant topics and issues	Minimal depth and coverage of significant topics and issues	Virtually no depth and coverage of significant topics and issues	
Clarity in presentation (25%)	Ideas and designs are exceptionally clear and organized throughout	Ideas and designs are clear and organized throughout	Ideas and designs are somewhat clear and organized throughout	Ideas and designs are mostly obscure and disorganized	Ideas and designs are entirely obscure and disorganized	
					Assignment Grade:	#N/A
The resulting grade is calculated as a weighted average as listed using A+=100, A=96, A-=92, B+=88, B=85, B-=82 etc.						
To obtain an A grade for the course, your weighted average should be >=95, A- >=90, B+ >=87, B >= 82, B- >= 80 etc.						

Use the **Ask the Facilitators Discussion 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.