SQL and Database Design

Portfolio Samples

Will Kittredge

TABLE OF CONTENTS

ASSIGNMENT 1

Note: A fresh and unaltered setup of the Bike database is assumed for this assignment.

1. Display the count of the number of rows in the `Owner` table.

```
-- count() is used to calculate the amount of rows in the OwnerCount table here
-- Selecting * from Owner would instead show all of the rows in the table
SELECT COUNT(*) AS OwnerCount
FROM Owner
```

2. Delete a specified record in the `Owner` table and prove that it is deleted.

```
-- Delete is used to state that we want to delete something from Owner
-- Where OwnerID = 1 specifies that we want to delete the row with that OwnerID
SELECT * FROM Owner
GO
DELETE FROM BikeOwner
WHERE OwnerID = 1
GO
DELETE FROM Owner
WHERE OwnerID = 1
GO
SELECT * FROM Owner
```

3. Write a two-table implied join qualified with a key value.

```
-- Where is used to establish the the BrandID in the Bike table and the
-- BrandID in the Brand table are the same thing
SELECT BikeID, Brand.BName
FROM Bike, Brand
WHERE Bike.BrandID = Brand.BrandID
```

4. Write a three-table implied join qualified with a key value.

```
-- Same thing as above, except we use the and operator to establish two relationships
SELECT FName, BName, PDate
FROM Owner, Bike, BikeOwner
WHERE Bike.BikeID = BikeOwner.BikeID AND Owner.OwnerID = BikeOwner.OwnerID
```

5. Use twelve SQL functions (twelve queries with at least one function per query).

```
-- Return the average number of gears of all bikes in the database
SELECT AVG(Gear) AS AvgGears
FROM Bike

-- Return the gear count of the bike with the most gears
SELECT MAX(Gear) AS TopGear
FROM Bike

-- Return the gear count of the bike with the least gears
SELECT MIN(Gear) AS BottomGear
FROM Bike

-- Return the number of unique bikes in the database
SELECT COUNT(BikeID) AS BikeCount
FROM Bike

-- Return the length of condition value in the database
SELECT LEN(Condition) AS ConditionLength
FROM Bike

-- Return the sum of all gear count values
SELECT SUM(Gear) AS GearSum
FROM Bike

-- Return value of the condition in uppercase
SELECT UPPER(Condition) AS CONDITION
FROM Bike

-- Return FNames converted to all lowercase
SELECT LOWER(FName) AS fname
FROM Owner

-- Return the first 3 characters from FNames
SELECT TOP 5 LEFT(FName, 3) AS FName3
FROM Owner

-- Return the last 4 characters from LNames
SELECT TOP 5 RIGHT(LName, 5) AS LName5
FROM Owner

-- Return the reverse of FName
SELECT TOP 5 REVERSE(FName) AS emaNF
FROM Owner

-- Select characters 3-5 of LName
SELECT TOP 5 SUBSTRING(LName, 3, 5) AS LName35
FROM Owner
```

6. Write SQL that utilizes `LIKE` in the `WHERE` clause.

```
-- Like is used to specify that we're looking for a value that starts with a 'w'
SELECT FName
FROM Owner
WHERE FName LIKE 'w%'
```

7. Demonstrate the use of the `ROWCOUNT` feature to return 4 rows from a table that has more than 4 rows.

```
-- Setting rowcount to 4 to return only four rows, and then setting to 0 afterwards to
reset
SET ROWCOUNT 4
SELECT * FROM Owner
SET ROWCOUNT 0
```

8. Create SQL routine to insert values into the `Owner` table.

```
-- Insert is saying that some data will be inserted into the Owner table
-- Then the columns that we want to insert to are specified
-- Last, the values to be inserted are given
INSERT INTO Owner (FName, MName, LName)
VALUES ('FNameExample', 'MNameExample', 'LNameExample')
SELECT * FROM Owner
```

9. Create SQL routine to update one middle name in the `Owner` table.

```
-- Update is saying that we want to change some alreadu existing data
-- Set MNane is used to specify the column and the value of the data to be changed
-- Where OwnerID is used to specify the row of the data to be changed
UPDATE Owner
SET MName = 'NewMNameExample'
WHERE OwnerID = 23
SELECT * FROM Owner
```

10. Create SQL to truncate and then populate the `Color` table with at least 10 rows.

```
-- Truncate deletes everything in the color table, but keeps the table
-- 10 new color names are inserted for CName
TRUNCATE TABLE Color
GO
INSERT INTO Color (CName) VALUES
       ('Red'),
       ('Green'),
       ('Blue'),
       ('Magenta'),
       ('Orange'),
       ('Cyan'),
       ('White'),
       ('Yellow'),
       ('Black'),
       ('Purple')
GO
SELECT * FROM Color
```

11. Create SQL that inserts 2 bikes for an owner of your choice and 3 bikes for another owner of your choice.

```
-- Inserting necessary data into Bike and BikeOwner
INSERT INTO Bike (Condition, Gear, MaterialID, TypeID, BrandID, BName) VALUES
       ('New', 24, 1, 1, 1, 'Bike1-1'),
       ('New', 21, 2, 4, 3, 'Bike1-2'),
       ('New', 8, 3, 2, 2, 'Bike2-1'),
       ('New', 12, 2, 4, 3, 'Bike2-2'),
       ('New', 15, 3, 3, 5, 'Bike2-3')

INSERT INTO BikeOwner (BikeID, OwnerID, PDate) VALUES
       (15, 2, '12/25/2022'),
       (16, 2, '02/14/2023'),
       (17, 3, '02/16/2023'),
       (18, 3, '02/16/2023'),
       (19, 3, '02/16/2023')
```

12. Create a query that uses `GROUP BY` with the `Bike` and `Owner` tables.

```
-- Group By is used to return the result in alphabetical order by FName
SELECT FName, Bikes=COUNT(Bike.BikeID)
FROM Owner, BikeOwner, Bike
WHERE Bike.BikeID = BikeOwner.BikeID AND Owner.OwnerID = BikeOwner.OwnerID
GROUP BY FName
```

13. Create SQL statement that utilizes the `SUBSTRING` command.

```
-- Substring is used to get only part of a string
-- In this case, from the first to the fourth character (inclusive)
SELECT LName, ShortLName=SUBSTRING(LName, 1, 4)
FROM Owner
WHERE LName LIKE 'ki%'
```

14. Demonstrate the use of `DISCTINCT`, `ASC`, and `DESC` in a query.

```
-- Distinct is used to specify that only unique FName values should be selected
-- In this case, all owners are shown because all of the FName values are unique
-- Desc is used to order in reverse alphabetical order, asc would order alphabetically
SELECT DISTINCT FName
FROM Owner
ORDER BY FName DESC
```

15. Demonstrate `@@VERSION`.

```
-- Shows SQL server version information
SELECT @@VERSION
```

16. How many owners are in `Owner`?

```
-- There are 22 owners in Owner
SELECT Owners=COUNT(OwnerID)
FROM Owner
```

17. Which owner has the most bikes?

```
-- There's a 3 way tie for most bikes owned
SELECT TOP 5 FName, Bikes=COUNT(Bike.BikeID)
FROM Owner, Bike, BikeOwner
WHERE Owner.OwnerID = BikeOwner.OwnerID AND Bike.BikeID = BikeOwner.BikeID
GROUP BY FName
ORDER BY Bikes DESC
```

18. Which owner has the longest last name?

```
-- Looking at the top five longest last names
-- len(LName) gets the length is characters of a last name
-- max(len(LName)) is used to sort the result in descening order with Order By
-- Excluding the example last name that was inserted earlier, there's a 3 way tie
SELECT TOP 5 FName, LName, LongestLName=MAX(LEN(LName))
FROM Owner
GROUP BY FName, LName
ORDER BY LongestLName DESC
```

19. Write a script that creates the data to have a bike with more than one color as well as a script to display that bike and its colors.

```
INSERT INTO BikeColor (BikeID, ColorID) VALUES
       (15, 100),
       (15, 110),
       (16, 200)
GO
SELECT BikeColor.BikeID, CName
FROM BikeColor, Color
WHERE Color.ColorID = BikeColor.ColorID
```

ASSIGNMENT 2

Note: A fresh and unaltered setup of the Bike database is assumed for this assignment.

1. Display the count of the number of rows in a table.

```
SELECT COUNT(*) AS BikeCount
FROM Bike
```

2. Delete a specified record in a table and prove that it is deleted.

```
SELECT * FROM Bike
GO
DELETE FROM Bike
WHERE BikeID = 11
SELECT * FROM Bike
GO
```

3. Write a two-table join that creates a view.

```
-- Create view is used to create a view (virtual table)
-- This view only includes BikeID, Condition, BrandID, and BName
-- A user would therefore be unable to select Gear (for example) from the view
CREATE VIEW vw_ExampleView AS
SELECT BikeID, Condition, Bike.BrandID, Brand.BName
FROM Bike, Brand
WHERE Bike.BrandID = Brand.BrandID

SELECT * FROM vw_ExampleView
```

4. Write a three-table join qualified with a key value from one view and one other table.

```
SELECT vw_ExampleView.BikeID, BName, PDate
FROM vw_ExampleView, BikeOwner
WHERE vw_ExampleView.BikeID = BikeOwner.BikeID
```

5. Use five SQL functions (five queries with at least one function per query).

```
-- Absolute value
SELECT ABS(-1)

-- Power
SELECT POWER(1,1)

-- Sum
SELECT SUM(1)

-- Round
SELECT ROUND(1, 1)

-- Pi
SELECT PI()
```

6. Create SQL stored procedure to insert values into a row using variables passed in on execution.

```
-- Create proc is used to create a procedure that can be re-used
-- This procedure inserts the passed FName and LName values to the owner table
CREATE PROC spu_InsertOwner
@FName NVARCHAR(50), @LName NVARCHAR(50) AS
INSERT INTO Owner (FName, LName)
VALUES (@FName, @LName)

EXEC spu_InsertOwner 'Will', 'Kittredge'
SELECT * FROM Owner
```

7. Write SQL to create a table, key, and index.

```
-- Create table is used to create and name a table
-- The column 'PrimaryKey' with an int datatype is added, nulls are not allowed
-- A clustered index is created
CREATE TABLE ExampleTable
(PrimaryKey INT NOT NULL,
CONSTRAINT PK_ExampleTable PRIMARY KEY CLUSTERED
(PrimaryKey ASC)
ON [Primary])
```

8. Create a query that uses `GROUP BY`.

```
SELECT FName, Bikes=COUNT(Bike.BikeID)
FROM Owner, BikeOwner, Bike
WHERE Bike.BikeID = BikeOwner.BikeID AND Owner.OwnerID = BikeOwner.OwnerID
GROUP BY FName
```

9. Demonstrate the use of `DISTINCT`, `ASC`, and `DESC` in a query.

```
-- Distinct only gets unique values
-- My first name is only listed once, despite me inserting a duplicate earlier
-- Asc displays the names in alphabetical order, desc would display in reverse order
SELECT DISTINCT FName
FROM Owner
ORDER BY FName ASC
```

10. Create a query that demonstrates an inner join.

```
-- Inner join
SELECT BikeID, Gear, Agency, Registration.RegistrationID
FROM Bike
INNER JOIN Registration
ON Bike.RegistrationID = Registration.RegistrationID
```

11. Create a query that demonstrates a left outer join.

```
-- Left join
SELECT BikeID, Gear, Agency, Registration.RegistrationID
FROM Bike
LEFT JOIN Registration
ON Bike.RegistrationID = Registration.RegistrationID
```

12. Create a query that demonstrates a right outer join.

```
-- Right join
SELECT BikeID, Gear, Agency, Registration.RegistrationID
FROM Bike
RIGHT JOIN Registration
ON Bike.RegistrationID = Registration.RegistrationID
```

ASSIGNMENT 3

1. Write SQL to create a `Donor` table (using appropriate data types) with the following columns: `DonorID`, `FirstName`, `LastName`, `DateOfBirth`, `GraduationDate`, `Gender`, and `NetWorth`.

```
CREATE TABLE Donor(
	DonorID INT IDENTITY(100,1) NOT NULL,
	FirstName NVARCHAR(50),
	LastName NVARCHAR(50),
	DateOfBirth DATE,
	GraduationDate DATE,
	Gender NVARCHAR(50),
	NetWorth MONEY,
CONSTRAINT [PK_Donor] PRIMARY KEY CLUSTERED
([DonorID] ASC))
```

2. Write SQL to define the primary key in the `DonorID` column.

```
-- SQL code and output covered in routine 1
```

3. Write SQL to define an index using the primary key and a second index using `LastName` and `FirstName`.

```
-- SQL for creating an index with the primary key covered in routine 1
-- Use 'Create index' followed by an index name to create an index
-- The next line specifies that the index should be on the LastName and FirstName columns
in the Donor table
CREATE INDEX idx_DonorName
ON Donor (LastName, FirstName)
```

4. Write SQL to insert ten rows of data into the `Donor` table.

```
INSERT INTO Donor (FirstName, LastName, DateOfBirth, GraduationDate, Gender, NetWorth)
VALUES
	('Robert', 'Jones', '4/18/2000', '5/6/2022', 'Male', 75295),
	('David', 'Smith', '3/22/2001', '5/6/2023', 'Male', 33665),
	('Lilly', 'Williams', '10/16/2000', '5/6/2023', 'Female', 53908),
	('Andrew', 'Cooper', '9/2/2001', '5/6/2024', 'Male', 24749),
	('Daniel', 'Brown', '10/11/2000', '5/6/2023', 'Male', 37938),
	('Julie', 'Campbell', '1/6/2000', '5/6/2022', 'Female', 21575),
	('Marilyn', 'Allen', '3/20/2001', '5/6/2023', 'Female', 94452),
	('Harrison', 'Jones', '4/28/2001', '5/6/2023', 'Male', 35461),
	('Steven', 'Miller', '12/7/2000', '5/6/2023', 'Male', 23165),
	('Elizabeth', 'Lee', '11/13/2001', '5/6/2024', 'Female', 92413)
```

5. Write SQL to truncate and drop the `Donor` table.

```
TRUNCATE TABLE Donor
DROP TABLE Donor
```

6. Write SQL to create a view of the `Donor` table using all the columns except `NetWorth`.

```
CREATE VIEW vw_DonorView AS
SELECT FirstName, LastName, DateOfBirth, GraduationDate, Gender
FROM Donor
```

7. Write SQL that uses the view to display all customers in the `Donor` table sorted by `FirstName` in descending order.

```
SELECT * FROM vw_DonorView
ORDER BY FirstName DESC
```

8. Write a routine with at least 3 If statements.

```
-- If statements are used to conditionally execute code
-- If the expression after 'If' evaluates to true, code between the 'Begin' and 'End'
statements is executed
DECLARE @Value INT = 4
PRINT 'Value: ' +  CAST(@Value AS VARCHAR)

IF @Value != 0
	BEGIN
		PRINT 'Value is not 0'
	END
If (@Value % 2) != 1
	BEGIN
		PRINT 'Value is not odd'
	END
IF @Value = 4
	BEGIN
		PRINT 'Value is 4'
	END
```

9. Write a routine with `IF` and `ELSE IF` statements.

```
-- 'Else' is used to execute different code in the event that the expression following
the original 'If' evaluates to false
DECLARE @Value INT = 4
PRINT 'Value: ' +  CAST(@Value AS VARCHAR)

IF @Value = 0
	BEGIN
		PRINT 'Value is 0'
	END
ELSE IF @Value != 0
	BEGIN
		PRINT 'Value is not 0'
	END
IF (@Value % 2) = 1
	BEGIN
		PRINT 'Value is odd'
	END
ELSE IF (@Value % 2) = 0
	BEGIN
		PRINT 'Value is even'
	END
```

10. Write a `CASE` statement with at least 3 `WHEN` options.

```
-- Case statements are similar to if-then-else statements
-- Once one of the expressions after 'When' evaluates to true, the value after 'then' is
returned
SELECT NetWorth,
CASE
	WHEN NetWorth > 0 AND NetWorth <= 30000 THEN 'Net worth between 0 and 30k'
	WHEN NetWorth > 30000 AND NetWorth <= 90000 THEN 'Net worth between 30k and 90k'
	WHEN NetWorth > 90000 THEN 'Net worth above 90k'
END AS [Case]
FROM Donor
```

11. Alter a table to add an integer column that allows nulls to an existing table.

```
-- Alter is used to modify an existing table
-- Here, the column IntegerValue (int datatype) is added to the Donor table
ALTER TABLE Donor
ADD IntegerValue INT
```

12. Write a routine with error checking and error logging.

```
-- SQL code and output covered in routine 18
```

13. Write SQL that defines and uses variables.

```
-- SQL code and output covered in routine 9
```

14. Create a stored procedure that accepts a variable and has error checking and logging as part of the execute stored procedure statement.

```
-- suser_snmame() is used to get the name of the current login
CREATE PROC spu_ErrorTrackingInsert @ErrorHold INT AS
INSERT INTO ErrorTracking (ErrorNbr, UserName)
VALUES (@ErrorHold, SUSER_SNAME())
```

15. Write a statement that inserts ten rows of data that includes at least 5 columns. The data must vary for each record by using variables that are changed within the statement for each record.

```
DECLARE @i INT = 0
DECLARE @FirstName NVARCHAR(50) = 'Jessie'
DECLARE @LastName NVARCHAR(50) = 'Smith'
DECLARE @Gender NVARCHAR(50)
DECLARE @NetWorth MONEY

WHILE @i <= 9
	BEGIN
		IF (@i % 2) = 1
			SET @Gender = 'Male'
		ELSE
			SET @Gender = 'Female'

		SET @NetWorth = 20000 + (@i * 10000)
		SET @i = @i + 1
		INSERT INTO Donor (FirstName, LastName, Gender, NetWorth, IntegerValue)
		VALUES (@FirstName, @LastName, @Gender, @NetWorth, @i)
	END
```

16. Create a table with appropriate structure to remove duplicates when loading the table.

```
-- IGNORE_DUP_KEY = ON tells the table to ignore duplicate key values
-- Setting IDENTITY_INSERT to ON allows one to insert primary key values even when
identity specification is enabled
CREATE TABLE Person(
	PersonID INT IDENTITY(1,1) NOT NULL,
	FirstName NVARCHAR(50),
	LastName NVARCHAR(50),
CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED
([PersonID] ASC))

ALTER TABLE Person REBUILD WITH (IGNORE_DUP_KEY = ON)
SET IDENTITY_INSERT Person ON

INSERT INTO Person (PersonID, FirstName, LastName) VALUES
	(1, 'Will', 'Kittredge'),
	(1, 'Will', 'Kittredge'),
	(1, 'Will', 'Kittredge'),
	(2, 'Different', 'Person'),
	(2, 'Different', 'Person')

SELECT * FROM Person
```

17. Create a table that has 10 records with 3 duplicate records. Write SQL to identify the duplicate records by key value, and SQL to display only the duplicate records.

```
-- 'Having' is used to show only the records that satisfy the following condition
-- In this example, only records with duplicate IDs are selected
CREATE TABLE Person2(
	Person2ID INT NOT NULL,
	FirstName NVARCHAR(50)
)

INSERT INTO Person2 (Person2ID, FirstName) VALUES
	(1, 'Will'),
	(1, 'Will'),
	(1, 'Will'),
	(2, 'Tom'),
	(3, 'Harry'),
	(4, 'Matthew'),
	(5, 'Johanna'),
	(6, 'Isabella'),
	(7, 'Daniel'),
	(8, 'Ella')

SELECT Person2ID, FirstName, Person2IDCount=COUNT(Person2ID) FROM Person2
GROUP BY Person2ID, FirstName
HAVING COUNT(Person2ID) > 1
```

18. Write SQL that uses variables and performs error checking and error logging.

```
-- The stored procedure used here was created to accept a variable
-- @ErrorHold has been set to the value of @@Error as to save the error number
-- When the procedure is executed, we supply it with @ErrorHold
DECLARE @ErrorHold INT = 0
INSERT INTO Donor (DonorID, FirstName, LastName)
VALUES (21, 'Will', 'Kittredge')

SET @ErrorHold = @@ERROR
IF @ErrorHold != 0
	EXEC spu_ErrorTrackingInsert @ErrorHold
SET @ErrorHold = 0

SELECT * FROM ErrorTracking
```

ASSIGNMENT 4

1. Define `Customer` and `Rate` tables.

```
-- Create Customer table
CREATE TABLE Customer(
	[CustomerID] INT IDENTITY(1,1) NOT NULL,
	[FName] NVARCHAR(50) NOT NULL,
	[LName] NVARCHAR(50) NOT NULL,
	[DOB] DATE NOT NULL,
CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
(CustomerID ASC)
ON [PRIMARY])

-- Create Rate table
CREATE TABLE Rate(
	[RateID] INT IDENTITY(1,1) NOT NULL,
	[Rate] FLOAT NOT NULL,
CONSTRAINT [PK_Rate] PRIMARY KEY CLUSTERED
(RateID ASC)
ON [PRIMARY])
```

2. Populate `Customer` and `Rate` tables.

```
-- Populate Customer table
INSERT INTO Customer (FName, LName, DOB) VALUES
       ('John','Smith','1980-01-01'), ('Emily','Brown','1992-03-15'),
       ('David','Lee','1975-07-28'), ('Sarah','Williams','1988-05-12'),
       ('William','Johnson','1995-09-03'), ('Samantha','Davis','1983-11-19'),
       ('Michael','Thompson','1972-02-08'), ('Ashley','Garcia','1990-06-30'),
       ('Christopher','Martinez','1986-08-22'), ('Jennifer','Nguyen','1997-04-17'),
       ('Robert','Rodriguez','1981-12-25'), ('Elizabeth','Herman','1994-10-07'),
       ('Daniel','Gonzalez','1978-09-01'), ('Taylor','Allen','1991-02-14'),
       ('Anthony','Lee','1987-06-09'), ('Nicole','King','1984-04-23'),
       ('Matthew','Taylor','1976-12-11'), ('Stephanie','Adams','1993-08-05'),
       ('James','Mitchell','1989-03-29'), ('Amanda','Wright','1996-11-02'),
       ('Timothy','Phillips','1979-07-20'), ('Brittany','Parker','1998-05-24'),
       ('Kevin','Davis','1982-01-13'), ('Melissa','Jackson','1990-09-15'),
       ('Joseph','Anderson','1985-03-08'), ('Hannah','Nelson','1988-11-12'),
       ('Jacob','Rivera','1999-02-28'), ('Caroline','Perez','1977-10-01'),
       ('Brandon','Collins','1991-05-23'), ('Megan','Smith','1982-08-17'),
       ('David','White','1979-06-05'), ('Abigail','Ramirez','1994-04-16'),
       ('Ryan','Harris','1986-12-03'), ('Natalie','Gonzalez','1997-07-09'),
       ('Scott','Garcia','1984-03-20'), ('Isabella','Martin','1992-09-29'),
       ('Samuel','Anderson','1978-02-13'), ('Katherine','Jones','1993-01-18'),
       ('Gabriel','Williams','1995-08-07'), ('Lauren','Clark','1981-04-14'),
       ('Thomas','Hall','1990-10-25'), ('Madison','Lee','1987-06-15'),
       ('Caleb','Lopez','1976-11-30'), ('Victoria','Taylor','1998-05-03'),
       ('Joshua','Brown','1983-09-11'), ('Olivia','Davis','1996-09-23'),
       ('Ethan','Rodriguez','1980-07-08'), ('Grace','Jackson','1989-02-05'),
       ('Jonathan','Smith','1991-04-21'), ('Avery','Allen','1975-03-27')

-- Populate Rate table
INSERT INTO Rate (Rate) VALUES
       (50.00),  (75.00),
       (100.00), (125.00),
       (150.00), (175.00),
       (200.00), (225.00),
       (250.00), (275.00),
       (300.00), (325.00),
       (350.00), (375.00),
       (400.00), (425.00),
       (450.00), (475.00)
```

3. Include an alter table command to insert `BeginAge` and `EndAge` columns for the `Rate` table (age ranges in 5 year increments up to age 89).

```
-- Add BeginAge and EndAge columns
ALTER TABLE Rate
ADD
	[BeginAge] INT,
	[EndAge] INT

-- Declare variables
DECLARE @BeginAge INT = 0
DECLARE @EndAge INT = 4
DECLARE @i INT = 1

-- Update values for BeginAge and EndAge
WHILE @i <= 18
	BEGIN
		UPDATE Rate
		SET BeginAge = @BeginAge, EndAge = @EndAge
		WHERE RateID = @i

		SET @i = @i + 1
		SET @BeginAge = @BeginAge + 5
		SET @EndAge = @EndAge + 5
	END

-- Demonstrate that BeginAge and EndAge values are in the Rate table
SELECT * FROM Rate
```

4. Write SQL routine to assign an insurance rate based on age.

```
-- Add RateID column (foreign key)
ALTER TABLE Customer
ADD [RateID] INT

-- Update RateID values in the Customer table based on customer age
UPDATE Customer
SET RateID = (
	SELECT TOP 1 RateID
	FROM Rate
	WHERE DATEDIFF(year, Customer.DOB, GETDATE()) BETWEEN Rate.BeginAge
AND Rate.EndAge
	ORDER BY RateID ASC
)

-- Demonstrate that a Rate has been assigned to each customer
SELECT TOP 10 FName, LName, Rate
FROM Customer, Rate
WHERE Customer.RateID = Rate.RateID
```

5. Write SQL routine to alter the `Customer` table to include rider flag. Compute and display rider cost.

```
-- Add Rider column
ALTER TABLE Customer
ADD [Rider] BIT NOT NULL
DEFAULT 0

-- Set ten Rider flags to 1
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 3
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 7
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 12
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 19
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 23
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 31
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 38
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 41
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 46
UPDATE Customer
SET Rider = 1 WHERE CustomerID = 49

-- Demonstrate that ten Rider flags have been set to 1
SELECT * FROM Customer
WHERE Rider = 1
```

6. Write SQL routine to generate a report of all customers sorted by age range, `FName`, and `LName`. Also display the rate (or rate*rider, whichever applies).

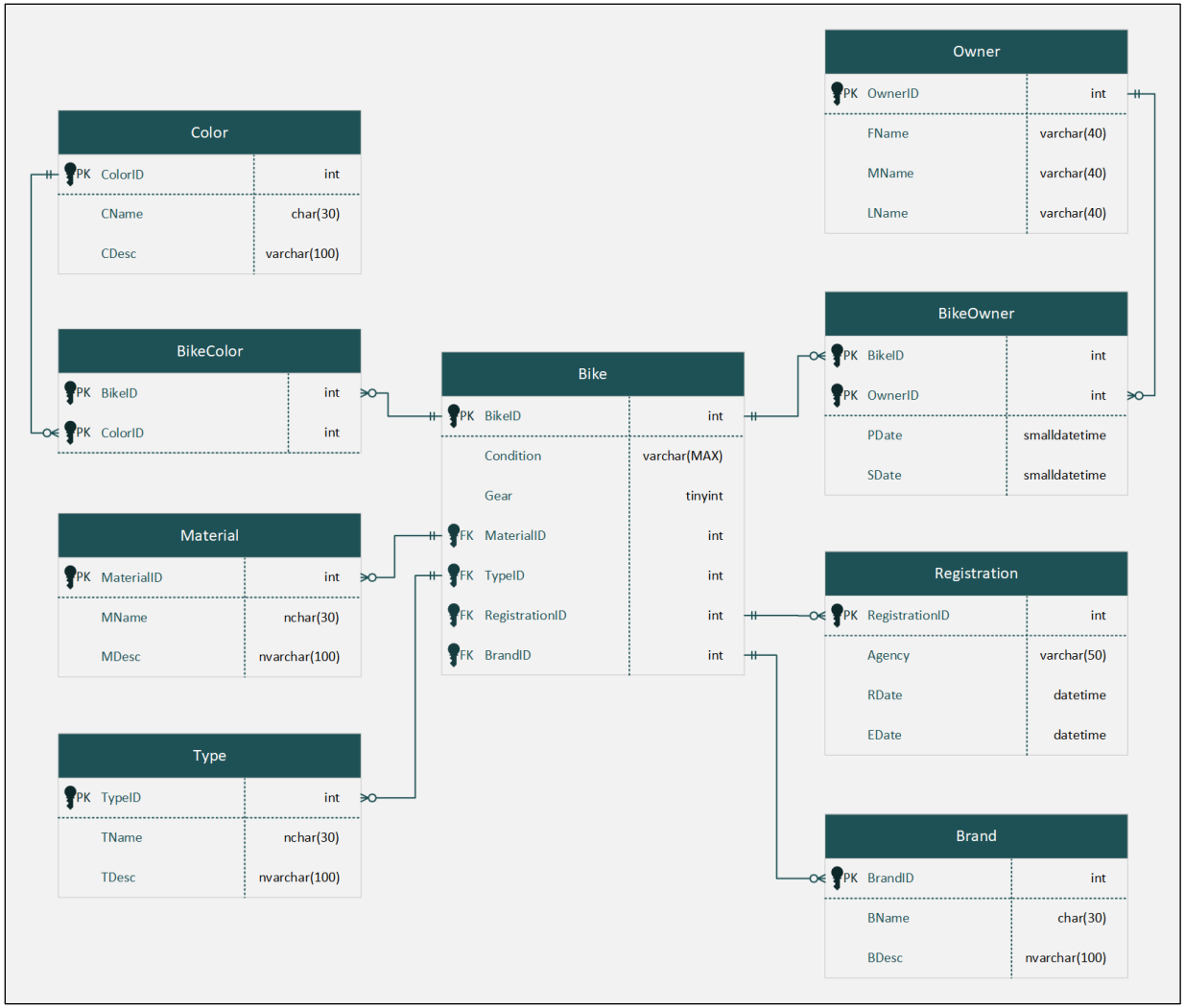
```
-- Generate a report of all customers ordered by age range. Also includes
LName, FName, and Rate
-- If a customer's Rider flag is set to 1, their rate is doubled
SELECT AgeRange=CONCAT(CAST(BeginAge AS VARCHAR),'-',CAST(EndAge AS VAR-
CHAR)), LName, FName, Rate=Rate*(CAST(Rider AS INT)+1)
FROM Customer, Rate
WHERE Customer.RateID = Rate.RateID
ORDER BY AgeRange ASC
```

BIKE DATABASE

The bike database is an organized collection of data used to store, retrieve, or otherwise alter various details associated with individual bikes. It was designed collaboratively in class. Each individual bike is assigned a uniquely identifiable `BikeID` that distinguishes it from other bikes in the database. Data is stored across 9 tables organized such that any fields (i.e., columns) that describe similar concepts are all grouped into the same table. Names and descriptions of every database table are listed below.

Table	Description
Bike	Stores bike data, FKs for brand, material, registration, and type.
Brand	Stores bike brand data.
Color	Stores bike color data.
Material	Stores bike material data.
Owner	Stores bike owner data.
Registration	Stores bike registration data.
Type	Stores bike type data.
BikeColor	Resolves many-to-many bike and color relationships.
BikeOwner	Resolves many-to-many bike and owner relationships.

DESIGN DIAGRAM



STANDARDS

Data organization strategies in the database were designed to satisfy normalization principles:

- No redundant data.
- No repeating groups (i.e., atomic data – records are of individual things).
- Everything in a table depends on the key.

Additionally, the database design adheres to the following standards:

1. Tables and columns follow the CamelCase naming convention.
2. Primary keys are named the name as their table, but with "ID" affixed to the end.
3. Bridge tables are named by concatenating the names of each table being "bridged."
4. Where applicable, the first word in a two-word column name is abbreviated with the first letter of the word in uppercase and the entire second word in title case.
5. Unicode character data types are preferred over non-Unicode character data types.
6. Nulls are allowed on description fields.

DATA DICTIONARY

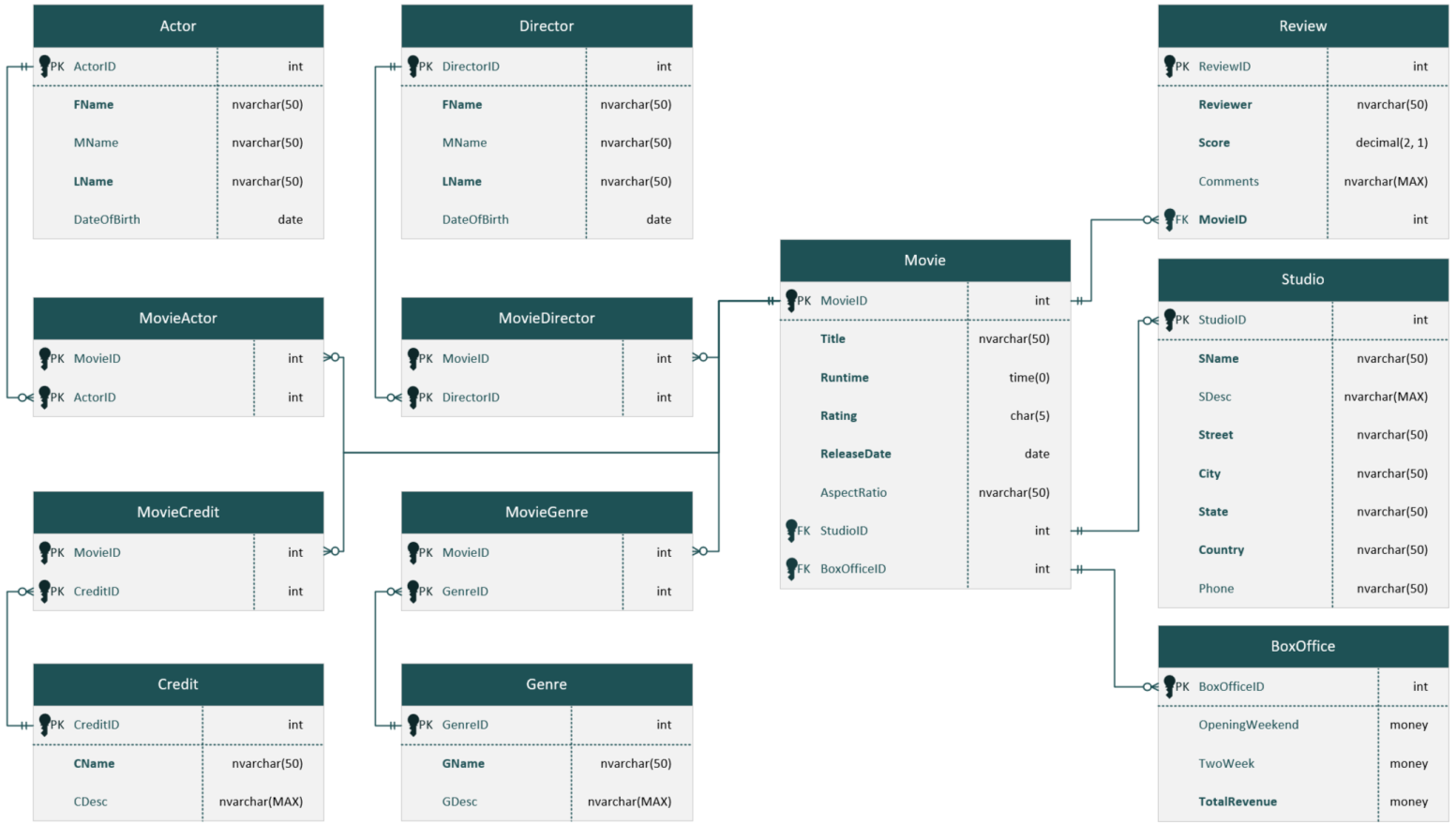
Table	Column Name	Data Type	Key?	Index	Nullable?	Sample Data
Bike	BikeID	INT	PK			4
	Condition	NVARCHAR(MAX)			✓	'Good'
	Gear	TINYINT				21
	MaterialID	INT	FK		✓	6
	TypeID	INT	FK			8
	RegistrationID	INT	FK		✓	4
	BrandID	INT	FK			7
	BName	NVARCHAR(50)			✓	'Wildfire'
Brand	BrandID	INT	PK			2
	BName	NVARCHAR(50)				'Specialized'
	BDesc	NVARCHAR(100)			✓	'American'
Color	ColorID	INT	PK			210
	CName	NVARCHAR(50)				'Maroon'
	CDesc	NVARCHAR(100)			✓	'Shade of red'
Material	MaterialID	INT	PK			6
	MName	NVARCHAR(50)				'Unknown'
	MDesc	NVARCHAR(100)			✓	'Unknown'
Owner	OwnerID	INT	PK			13
	FName	NVARCHAR(50)				'William'
	MName	NVARCHAR(50)			✓	'James'
	LName	NVARCHAR(50)				'Kittredge'
Registration	RegistrationID	INT	PK			4
	Agency	NVARCHAR(50)				'State of MI'
	RDate	SMALLDATETIME				2/23/2023
	EDate	SMALLDATETIME			✓	2/23/2024
	RDesc	NVARCHAR(MAX)			✓	'Valid 1 yr.'
Type	TypeID	INT	PK			4
	TName	NVARCHAR(50)				'e-Bike'
	TDesc	NVARCHAR(100)			✓	'Motorized'
BikeColor	BikeID	INT	PK			4
	ColorID	INT	PK			210
BikeOwner	BikeID	INT	PK			4
	OwnerID	INT	PK			13
	PDate	SMALLDATETIME			✓	2/23/2023
	SDate	SMALLDATETIME			✓	2/23/2025

MOVIE DATABASE

The movie database was designed collaboratively in class. In includes 12 tables in total, with `Movie` being the main one. Bridge tables are used to resolve many-to-many relationships between the main table and the `Actor`, `Director`, `Credit`, and `Genre` tables. The names and descriptions of every database table are listed below.

Table	Description
Movie	Stores information about movies themselves.
Review	Stores movie reviews.
Studio	Stores studio information.
BoxOffice	Stores box office numbers.
Actor	Stores information about actors.
Director	Stores information about directors.
Credit	Stores movie credit information.
Genre	Stores movie genre information.
MovieActor	Resolves many-to-many movie and actor relationships.
MovieDirector	Resolves many-to-many movie and director relationships.
MovieCredit	Resolves many-to-many movie and credit relationships.
MovieGenre	Resolves many-to-many movie and genre relationships.

DESIGN DIAGRAM



STANDARDS

Data organization strategies in the database were designed to satisfy normalization principles:

- No redundant data.
- No repeating groups (i.e., atomic data – records are of individual things).
- Everything in a table depends on the key.

Additionally, the database design adheres to the following standards:

1. Tables and columns follow the CamelCase naming convention.
2. Primary keys are named the name as their table, but with "ID" affixed to the end.
3. Bridge tables are named by concatenating the names of each table being "bridged."
4. Where applicable, the first word in a two-word column name is abbreviated with the first letter of the word in uppercase and the entire second word in title case.
5. Unicode character data types are preferred over non-Unicode character data types.
6. Nulls are allowed on description fields.

CREATE & POPULATE TABLES

► Create tables:

```
USE Movie

-- Create Movie table
CREATE TABLE Movie(
	MovieID INT IDENTITY(1,1) NOT NULL,
	Title NVARCHAR(50) NOT NULL,
	Runtime TIME(0) NOT NULL,
	Rating CHAR(5) NOT NULL,
	ReleaseDate DATE NOT NULL,
	AspectRatio NVARCHAR(50),
	StudioID INT,
	BoxOfficeID INT,
CONSTRAINT PK_Movie primary key clustered
(MovieID ASC)
ON [Primary])

-- Create Review table
CREATE TABLE Review(
	ReviewID INT IDENTITY(1,1) NOT NULL,
	Reviewer NVARCHAR(50) NOT NULL,
	Score DECIMAL(2, 1) NOT NULL,
	Comments NVARCHAR(MAX),
	MovieID INT NOT NULL,
CONSTRAINT PK_Review PRIMARY KEY CLUSTERED
(ReviewID ASC)
ON [Primary])

-- Create Studio table
CREATE TABLE Studio(
	StudioID INT IDENTITY(1,1) NOT NULL,
	SName NVARCHAR(50) NOT NULL,
	SDesc NVARCHAR(MAX),
	Street NVARCHAR(50) NOT NULL,
	City NVARCHAR(50) NOT NULL,
	State NVARCHAR(50) NOT NULL,
	Country NVARCHAR(50) NOT NULL,
	Phone NVARCHAR(50),
CONSTRAINT PK_Studio PRIMARY KEY CLUSTERED
(StudioID ASC)
ON [Primary])

-- Create BoxOffice table
CREATE TABLE BoxOffice(
	BoxOfficeID INT IDENTITY(1,1) NOT NULL,
	OpeningWeekend MONEY,
	TwoWeek MONEY,
	TotalRevenue MONEY NOT NULL,
CONSTRAINT PK_BoxOffice PRIMARY KEY CLUSTERED
(BoxOfficeID ASC)
ON [Primary])
```

► Create tables (continued):

```
-- Create Actor table
CREATE TABLE Actor(
	ActorID INT IDENTITY(1,1) NOT NULL,
	FName NVARCHAR(50) NOT NULL,
	MName NVARCHAR(50),
	LName NVARCHAR(50) NOT NULL,
	DateOfBirth DATE,
CONSTRAINT PK_Actor PRIMARY KEY CLUSTERED
(ActorID ASC)
ON [Primary])

-- Create Director table
CREATE TABLE Director(
	DirectorID INT IDENTITY(1,1) NOT NULL,
	FName NVARCHAR(50) NOT NULL,
	MName NVARCHAR(50),
	LName NVARCHAR(50) NOT NULL,
	DateOfBirth DATE,
CONSTRAINT PK_Director PRIMARY KEY CLUSTERED
(DirectorID ASC)
ON [Primary])

-- Create Credit table
CREATE TABLE Credit(
	CreditID INT IDENTITY(1,1) NOT NULL,
	CName NVARCHAR(50) NOT NULL,
	CDesc NVARCHAR(MAX),
CONSTRAINT PK_Credit PRIMARY KEY CLUSTERED
(CreditID ASC)
ON [Primary])

-- Create Genre table
CREATE TABLE Genre(
	GenreID INT IDENTITY(1,1) NOT NULL,
	GName NVARCHAR(50) NOT NULL,
	GDesc NVARCHAR(MAX),
CONSTRAINT PK_Genre PRIMARY KEY CLUSTERED
(GenreID ASC)
ON [Primary])

-- Create MovieActor table
CREATE TABLE MovieActor(
	MovieID INT NOT NULL,
	ActorID INT NOT NULL,
CONSTRAINT PK_MovieActor PRIMARY KEY CLUSTERED(
MovieID ASC,
ActorID ASC)
ON [Primary])
```

► Create tables (continued):

```
-- Create MovieDirector table
CREATE TABLE MovieDirector(
       MovieID INT NOT NULL,
       DirectorID INT NOT NULL,
CONSTRAINT PK_MovieDirector PRIMARY KEY CLUSTERED(
MovieID ASC,
DirectorID ASC)
ON [Primary])

-- Create MovieCredit table
CREATE TABLE MovieCredit(
       MovieID INT NOT NULL,
       CreditID INT NOT NULL,
CONSTRAINT PK_MovieCredit PRIMARY KEY CLUSTERED(
MovieID ASC,
CreditID ASC)
ON [Primary])

-- Create MovieGenre table
CREATE TABLE MovieGenre(
       MovieID INT NOT NULL,
       GenreID INT NOT NULL,
CONSTRAINT PK_MovieGenre PRIMARY KEY CLUSTERED(
MovieID ASC,
GenreID ASC)
ON [Primary])
```

► Populate tables:

```
USE Movie

-- Insert movies into Movie table
INSERT INTO Movie (Title, Runtime, Rating, ReleaseDate, AspectRatio, StudioID, BoxOfficeID)
VALUES
	('The Sound of Music', '02:52:00', 'G', '03/02/1965', 'Cinemascope', 1, 1),
	('Dune', '02:35:00', 'PG13', '10/22/2021', 'Cinemascope', 2, 2),
	('The Truman Show', '01:43:00', 'PG', '06/05/1998', 'Super-16', 3, 3),
	('Casino Royale', '02:24:00', 'PG13', '11/17/2006', 'Cinemascope', 4, 4),
	('Se7en', '02:07:00', 'R', '09/22/1995', 'Cinemascope', 5, 5),
	('The Thing', '01:49:00', 'R', '06/25/1982', 'Cinemascope', 6, 6),
	('Ocean''s Eleven', '01:56:00', 'PG13', '12/07/2001', 'Cinemascope', 7, 7),
	('Ratatouille', '01:51:00', 'G', '06/27/2007', 'Cinemascope', 8, 8),
	('The Matrix', '02:16:00', 'R', '03/31/1999', 'Cinemascope', 9 ,9),
	('Unbreakable', '01:46:00', 'PG13', '11/22/2000', 'Cinemascope', 10, 10)

-- Insert earnings into BoxOffice table
INSERT INTO BoxOffice (TotalRevenue) VALUES
	(159437744), -- The Sound of Music
	(402027830), -- Dune
	(264118201), -- The Truman Show
	(616505162), -- Casino Royale
	(327333559), -- Se7en
	(19632715),  -- The Thing
	(450717150), -- Ocean's Eleven
	(623726085), -- Ratatouille
	(467222728), -- The Matrix
	(248118121)  -- Unbreakable

-- Insert studios into Studio table
INSERT INTO Studio (SName, Street, City, State, Country) VALUES
	('20th Century Fox', '10201 West Pico Boulevard', 'Los Angeles', 'California',
'United States'),
	('Legendary Pictures', '2900 West Alameda Avenue', 'Burbank', 'California', 'United
States'),
	('Paramount Pictures', '5555 Melrose Avenue', 'Los Angeles', 'California', 'United
States'),
	('Columbia Pictures', '10202 West Washington Boulevard', 'Culver City',
'California', 'United States'),
	('New Line Cinema', '4000 Warner Boulevard', 'Burbank', 'California', 'United
States'),
	('Universal Pictures', '100 Universal City Plaza', 'Universal City', 'California',
'United States'),
	('Warner Bros.', '4000 Warner Boulevard', 'Burbank', 'California', 'United States'),
	('Pixar Animation Studios', '1200 Park Avenue', 'Emeryville', 'California', 'United
States'),
	('Village Roadshow Pictures', '10100 Santa Monica Boulevard', 'Los Angeles',
'California', 'United States'),
	('Touchstone Pictures', '500 South Buena Vista Street', 'Burbank', 'California',
'United States')
```

► Populate tables (continued):

```
-- Insert reviews into the Review table
INSERT INTO Review (Reviewer, MovieID, Score, Comments) VALUES
       ('Will Kittredge', 1, 5.0, 'My favorite movie'),              -- The Sound of Music
       ('Will Kittredge', 2, 5.0, 'Does the book justice'),          -- Dune
       ('Will Kittredge', 3, 5.0, 'Equally entertaining and scary'),-- The Truman Show
       ('Will Kittredge', 4, 5.0, 'Craig''s best Bond movie'),       -- Casino Royale
       ('Will Kittredge', 5, 5.0, 'What''s in the box?!'),           -- Se7en
       ('Will Kittredge', 6, 5.0, '"Nobody trusts anybody now..."'),-- The Thing
       ('Will Kittredge', 7, 5.0, 'Effortlessly entertaining'),      -- Ocean's Eleven
       ('Will Kittredge', 8, 5.0, '"Anyone can cook!"'),             -- Ratatouille
       ('Will Kittredge', 9, 5.0, 'I know kung fu'),                 -- The Matrix
       ('Will Kittredge', 10, 4.5, 'A unique comic hero story')      -- Unbreakable

-- Insert genres into the Genre table
INSERT INTO Genre (GName) VALUES
       ('Musical'),
       ('Drama'),
       ('Romance'),
       ('Action'),
       ('Adventure'),
       ('Sci-Fi'),
       ('Comedy'),
       ('Thriller'),
       ('Crime'),
       ('Mystery'),
       ('Horror'),
       ('Animation'),
       ('Fantasy')
```

BACKUP & RECOVERY PLANNING

The purpose of this section is to provide reference database backup and recovery plans that could potentially be implemented. There are plans for low, standard, and high priority data (including other relevant information such as backup retention periods and security methods).

The following conditions apply to all plans:

- Backups are stored locally and copied to a cloud location in the background.
- Backups are encrypted, and all access is protected using virtual and physical controls.
- Backups on Tuesdays and Thursdays are kept for 1 full month, while backups from the remaining days are overwritten every week. The last full backup of each month is preserved indefinitely.

HIGH PRIORITY

Backups are estimated to take 1 hour on average.

[High Priority]	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
12:00 AM	Log	Log	Log	Log	Log	Log	Log
1:00 AM	Log	Log	Log	Log	Log	Log	Log
2:00 AM	Full	Full	Full	Full	Full	Full	Full
3:00 AM	Log	Log	Log	Log	Log	Log	Log
4:00 AM	Log	Log	Log	Log	Log	Log	Log
5:00 AM	Log	Log	Log	Log	Log	Log	Log
6:00 AM	Log	Log	Log	Log	Log	Log	Log
7:00 AM	Log	Log	Log	Log	Log	Log	Log
8:00 AM	Log	Log	Log	Log	Log	Log	Log
9:00 AM	Log	Log	Log	Log	Log	Log	Log
10:00 AM	Full	Full	Full	Full	Full	Full	Full
11:00 AM	Log	Log	Log	Log	Log	Log	Log
12:00 PM	Log	Log	Log	Log	Log	Log	Log
1:00 PM	Log	Log	Log	Log	Log	Log	Log
2:00 PM	Log	Log	Log	Log	Log	Log	Log
3:00 PM	Log	Log	Log	Log	Log	Log	Log
4:00 PM	Log	Log	Log	Log	Log	Log	Log
5:00 PM	Log	Log	Log	Log	Log	Log	Log
6:00 PM	Full	Full	Full	Full	Full	Full	Full
7:00 PM	Log	Log	Log	Log	Log	Log	Log
8:00 PM	Log	Log	Log	Log	Log	Log	Log
9:00 PM	Log	Log	Log	Log	Log	Log	Log
10:00 PM	Log	Log	Log	Log	Log	Log	Log
11:00 PM	Log	Log	Log	Log	Log	Log	Log

STANDARD PRIORITY

Backups are estimated to take 30 minutes on average.

[Standard Priority]	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
12:00 AM							
1:00 AM							
2:00 AM	Full	Full	Full	Full	Full	Full	Full
3:00 AM							
4:00 AM							
5:00 AM	Log	Log	Log	Log	Log	Log	Log
6:00 AM							
7:00 AM							
8:00 AM							
9:00 AM							
10:00 AM							
11:00 AM	Log	Log	Log	Log	Log	Log	Log
12:00 PM							
1:00 PM							
2:00 PM							
3:00 PM							
4:00 PM							
5:00 PM	Log	Log	Log	Log	Log	Log	Log
6:00 PM							
7:00 PM							
8:00 PM							
9:00 PM							
10:00 PM							
11:00 PM	Log	Log	Log	Log	Log	Log	Log

LOW PRIORITY

Backups are estimated to take 10 minutes on average.

[Low Priority]	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
12:00 AM							
1:00 AM							
2:00 AM	Full						
3:00 AM							
4:00 AM							
5:00 AM							
6:00 AM							
7:00 AM							
8:00 AM							
9:00 AM							
10:00 AM							
11:00 AM							
12:00 PM							
1:00 PM							
2:00 PM							
3:00 PM							
4:00 PM							
5:00 PM	Log	Log	Log	Log	Log	Log	Log
6:00 PM							
7:00 PM							
8:00 PM							
9:00 PM							
10:00 PM							
11:00 PM							

RECOVERY & SUPPORT PROTOCOLS

General database recovery steps are as follows:

1. Backup the log, then start a full backup of the database if time allows.
2. Notify that the database is down.
3. Restrict database access to owner only.
4. Restore the most recent full backup, then begin restoring log backups to present time.
5. Verify that the recovery was successful.
6. Restore previous access settings.
7. Notify that the database is up.
8. Start a full database backup.
9. If a backup of the database was completed in step 1, consider examining it for potentially useful information.

The following protocols should be followed in order to properly support database backup protocols, guidelines, and continencies:

- Always have at least one database administrator (DBA) working at any given time, and at least one other available in an emergency.
- All DBA team members must be able to contact one another outside of the workplace.
- In an emergency, all available DBAs will be summoned as necessary by phone call in order of seniority.
- If a DBA cannot be contacted within 10 minutes, begin attempting to contact the next individual down the list.
- Recovery order is primarily determined by the importance of each affected database, although the estimated recovery time and database user count are also considered.

Consider using some of the following questions and metrics to evaluate plan performance:

- How long was the database down? Was the outage part of a pattern?
- What caused the database to go down? Is it practical to eliminate the cause of the outage?
- How long did it take to restore the database? How long does it typically take?
- What is the size of the database? How big are the tables?
- How long does it typically take to restore the database from a backup?
- How old was the backup that the database was restored from?
- What were the effects of the outage? Was any data lost permanently? How much?

CHANGE MANAGEMENT

Engage in turnover/change control procedures to minimize the chances of an outage occurring due to planned activities. Procedures differ depending on the magnitude (minor/major) of the change. A basic change request form is also provided on the following page.

Minor changes:

These are changes or updates that are basic, take a low amount of time to implement, and are unlikely to disrupt database operations. Should something go wrong, there would not be any significant consequences. These changes are permitted to occur during normal operational hours, with 12:00 PM being the standard.

Major changes:

These are changes or updates that are more complex and take longer to implement compared to minor changes. It is possible that they could temporarily disrupt database operations or cause additional problems should the change go wrong. These changes occur during low usage hours, with 2:00 AM on Thursdays being the standard. A backup should be made before and after the change is implemented.

CHANGE REQUEST FORM			
Request ID			
Database			
Change Type			
Change Priority			
Change Description			
Change Reason			
Comments			
Requester Name		**Date**	
Approver Signature		**Date**	

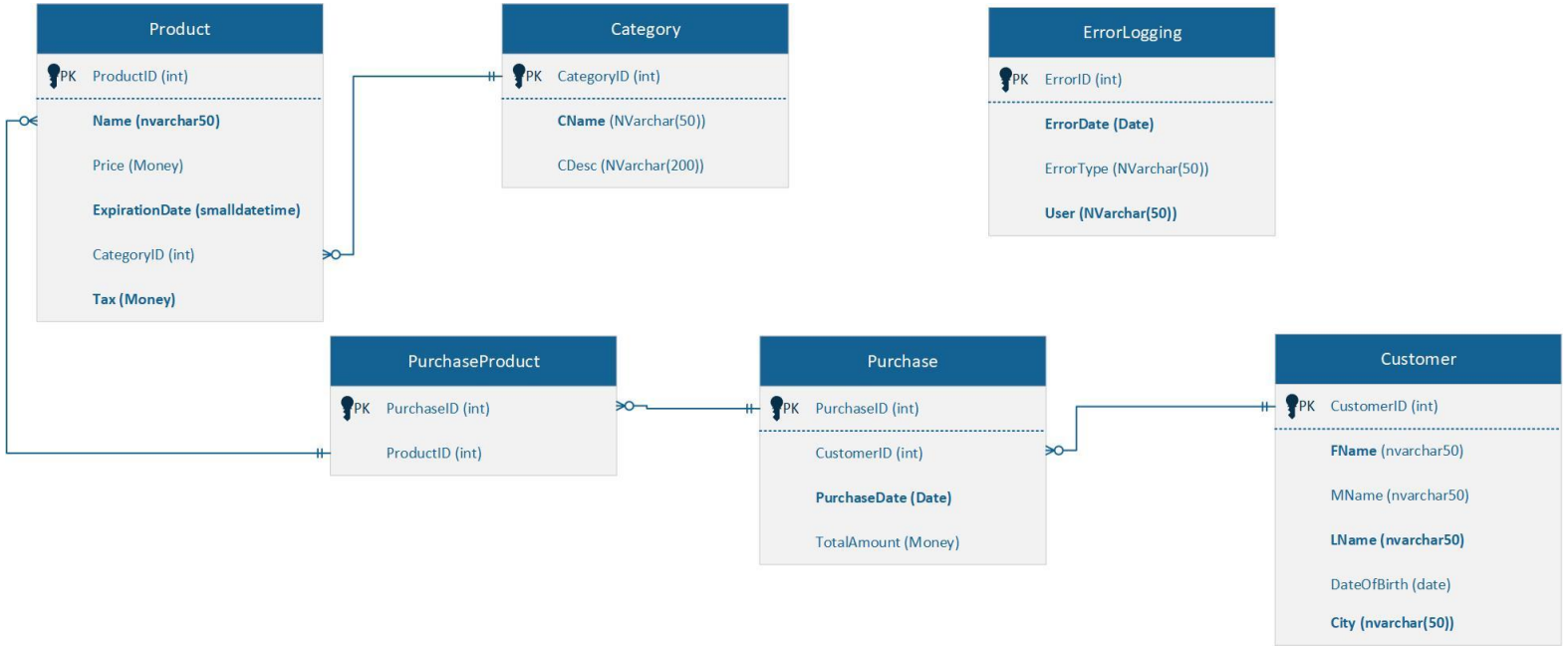
COMPETITION

The database competition is the final assignment of ISIN 325 (Database Security). Students work in groups at their tables (teams of 3-4) to solve as many problems as possible that relate to different database subjects discussed throughout the class.

Team members:

- Arnik de Coo
- Connor Lower
- Will Kittredge (me)

DESIGN DIAGRAM



CREATE & POPULATE TABLES

► Create tables:

```
CREATE TABLE Customer (
	CustomerID INT PRIMARY KEY,
	FName NVARCHAR(50),
	MName NVARCHAR(50),
	LName NVARCHAR(50),
	DateOfBirth DATE,
	City NVARCHAR(50)
);

CREATE TABLE Category (
	CategoryID INT PRIMARY KEY,
	CName VARCHAR(50),
	CDesc NVARCHAR(200)
);

CREATE TABLE Product (
	ProductID INT PRIMARY KEY,
	Name VARCHAR(255),
	Price MONEY,
	ExpirationDate SMALLDATETIME,
	CategoryID INT,
	Tax MONEY,
	FOREIGN KEY (CategoryID) REFERENCES Category(CategoryID)
);

CREATE TABLE Purchase (
	PurchaseID INT PRIMARY KEY,
	CustomerID INT,
	PurchaseDate DATE,
	TotalAmount MONEY
);

CREATE TABLE PurchaseProduct (
	PurchaseID INT,
	ProductID INT
);
```

► Populate tables:

```
INSERT INTO Customer (CustomerID, FName, MName, LName, DateOfBirth, City) VALUES
       (1, 'John', 'D', 'Doe', '1990-01-01', 'New York'),
       (2, 'Jane', 'M', 'Smith', '1985-03-15', 'Los Angeles'),
       (3, 'Bob', 'J', 'Johnson', '1995-07-20', 'Chicago'),
       (4, 'Mary', 'K', 'Jones', '1992-05-10', 'Houston'),
       (5, 'Tom', 'A', 'Williams', '1988-11-30', 'San Francisco'),
       (6, 'Alice', 'L', 'Brown', '1997-02-28', 'Seattle'),
       (7, 'David', 'P', 'Lee', '1983-09-05', 'Boston'),
       (8, 'Sarah', 'R', 'Taylor', '1993-12-25', 'Atlanta'),
       (9, 'Mike', 'S', 'Davis', '1989-04-12', 'Denver'),
       (10, 'Lisa', 'N', 'Clark', '1991-08-22', 'Miami');

INSERT INTO Category (CategoryID, CName, CDesc) VALUES
       (1, 'Electronics', 'Electronic devices and accessories'),
       (2, 'Clothing', 'Clothing and apparel for men and women'),
       (3, 'Home', 'Home goods and furniture'),
       (4, 'Beauty', 'Beauty and personal care products'),
       (5, 'Sports', 'Sports and fitness equipment'),
       (6, 'Toys', 'Toys and games for kids'),
       (7, 'Books', 'Books and literature'),
       (8, 'Music', 'Music CDs and vinyl records'),
       (9, 'Movies', 'DVDs and Blu-ray movies'),
       (10, 'Pet Supplies', 'Pet food and accessories');

INSERT INTO Product (ProductID, Name, Price, ExpirationDate, CategoryID, Tax) VALUES
       (1, 'iPhone 13', 999.00, '2024-10-01', 1, 0.05),
       (2, 'Samsung Galaxy S21', 899.00, '2024-09-01', 1, 0.05),
       (3, 'Levi 501 Jeans', 69.50, '2026-01-01', 2, 0.07),
       (4, 'Nike Air Max 270', 150.00, '2025-05-01', 5, 0.07),
       (5, 'HP Pavilion Laptop', 799.99, '2024-12-01', 1, 0.05),
       (6, 'Olay Regenerist Cream', 24.99, '2024-06-01', 4, 0.08),
       (7, 'Sony PS5 Console', 499.99, '2024-11-01', 5, 0.07),
       (8, 'Lego Classic Box', 29.99, '2025-06-01', 6, 0.05),
       (9, 'Harry Potter and the Philosophers Stone', 12.99, '2025-08-01', 7, 0.07),
       (10, 'Pink Floyd - The Wall Vinyl', 39.99, '2025-12-01', 8, 0.07);

INSERT INTO Purchase (PurchaseID, CustomerID, PurchaseDate, TotalAmount) VALUES
       (1, 101, '2022-01-01', 250.00), (2, 102, '2022-02-05', 150.50),
       (3, 103, '2022-03-10', 50.00),  (4, 104, '2022-04-15', 400.00),
       (5, 105, '2022-05-20', 75.99),  (6, 106, '2022-06-25', 300.00),
       (7, 107, '2022-07-30', 125.00), (8, 108, '2022-08-01', 800.00),
       (9, 109, '2022-09-01', 90.00),  (10, 110, '2022-10-01', 175.50);

INSERT INTO PurchaseProduct (PurchaseID, ProductID) VALUES
       (1,2), (1,3),
       (3,1), (4,2),
       (5,3), (6,5),
       (7,7), (8,6),
       (9,3), (10,2);
```

PART 1: DB DESIGN ROUTINES

1. Display how many purchases and the total amount purchased by each customer using one SQL statement.

```
SELECT FName, COUNT(Purchase.CustomerID) AS PurchaseAmount, SUM(TotalAmount) AS
TotalAmount
FROM Customer, Purchase
GROUP BY FName
```

2. Display how many customers are in each city using one SQL statement.

```
SELECT City, COUNT(FName) AS Customers FROM Customer
GROUP BY City
```

3. Display all the sales each product is tied to, with the output sorted by product and category.

```
SELECT Product.Name, SUM(totalAmount) AS TotalSales, CName
FROM Product, Purchase, PurchaseProduct, Category
WHERE Product.ProductID = PurchaseProduct.ProductID
AND Purchase.PurchaseID = PurchaseProduct.PurchaseID
GROUP BY Product.Name, CName
```

4. Display all the products in each category.

```
SELECT Name, CName FROM Product
JOIN Category ON Category.CategoryID = Product.CategoryID
```

5. Display the product name and category sorted by city, then by category, then by product.

```
SELECT Name, CName, City
FROM Customer, Product
JOIN Category ca ON ca.CategoryID = Product.CategoryID
ORDER BY City, Cname, Name
```

6. Display a count of the number of customers in each city sorted by city.

```
SELECT COUNT(CustomerID) AS AmountCustomerInCity, City
FROM Product, Customer
GROUP BY City
ORDER BY City
```

7. Insert the new category `Health and Beauty Aids` with an appropriate description into the category table.

```
INSERT INTO Category(CategoryID, CName, CDesc)
VALUES (20, 'Health and Beauty Aids', 'Products to achieve better health.')
```

8. Insert records that indicate the purchases Judy Denth and Idris Elba made from `Health and Beauty Aids` and produce a report. Include the SQL to perform the inserts as well.

unfinished during competition

9. Display the category, purchase date and amount for each recipient include `LName`, `FName`) for each category sorted by `LName` that have purchased at least 4 products.

```
SELECT CName, PurchaseDate, TotalAmount, LName, FName
FROM Category, Purchase, PurchaseProduct, Customer
GROUP BY CName, PurchaseDate, TotalAmount, LName, FName
HAVING COUNT(Purchase.CustomerID) > 3
ORDER BY LName
```

10. Display a count of the number of products for each category using one SQL statement.

```
SELECT Category.CName, COUNT(Product.ProductID) AS 'Product Amount'
FROM Category
INNER JOIN Product ON Category.CategoryID = Product.CategoryID
GROUP BY Category.CName
```

PART 2: ADMINISTRATIVE TASKS

11. Display the active logins in SQL Server.

```
SELECT
--The name of the databases with active logins
	DB_NAME(dbid) AS DBName,
--The amount of connections to each DB
	COUNT(dbid) AS NumberOfConnections,
--The Login Name correlated to each connection
	loginame AS LoginName
FROM
--System table with dbid and loginame
	sys.sysprocesses
WHERE
--Connection amount is greater than 0
	dbid >0
GROUP BY
	dbid, loginame
```

12. Demonstrate how to terminate a connection to SQL Server.

```
--See list of all active connections
sp_who 'active'
--Close a connection with kill command and spid
KILL 71
```

13. Create a trigger for update, insert, delete on a table. Trigger writes message to an audit table that provides 'useful' information.

unfinished during competition

14. Create a trigger that performs a cascading delete in a second table when a record is deleted from a table.

```
--If a purchase row is deleted the corresponding rows with the same
--PurchaseID are deleted in the PurchaseProduct bridge table
CREATE TRIGGER P2Delete
ON Purchase
FOR DELETE
AS
BEGIN
	DELETE FROM PurchaseProduct
	WHERE PurchaseID IN (SELECT deleted.PurchaseID FROM deleted)
END;
```

15. SQL to backup a database.

```
--Create a full backup of the database
--Unspecified Disk path automatically saves the backup to the default SMSS backup
directory.
USE CompDB
GO
BACKUP DATABASE CompDB
TO DISK = 'CompDB.bak'
       WITH FORMAT,
              MEDIANAME = 'SQLServerBackups',
              Name = 'Full Backup of CompDB';
GO
```

16. SQL to restore a database.

```
--Restore CompDB using the backup made in previous step
RESTORE DATABASE CompDB
FROM DISK = 'CompDB.bak'
WITH REPLACE;
```

17. SQL to create a user and assign to a role in a database.

```
--Create a new Login for Connor
CREATE LOGIN ConnorLowe WITH PASSWORD = 'O8*()NC!kl+';
--Make a user for that login
CREATE USER Auditor1 FOR LOGIN ConnorLowe;
--Categorize that user as an Auditor
CREATE ROLE PurchaseAuditor;
ALTER ROLE PurchaseAuditor ADD MEMBER Auditor1;
```

PART 3: SQL ROUTINES

18. Create SQL routine that uses the `CASE` statement.

```
-- Create temporary table
CREATE TABLE #Example(
	[Age] INT,
)

-- Insert some values
INSERT INTO #Example (Age)
VALUES (27), (18), (35), (21), (30), (19)

-- Case statement
SELECT Age,
CASE
	WHEN Age < 40 AND Age > 29 THEN '30-39'
	WHEN Age < 30 AND Age > 19 THEN '20-29'
	WHEN Age < 20 AND Age > 9 THEN '10-19'
	ELSE 'Age category unknown' END AS AgeCategory
FROM #Example
ORDER BY Age DESC
```

19. Create SQL routine that uses nested `IF`/`ELSE`/`IF`.

```
-- Declare variable
DECLARE @value INT = 2

-- Nested IF/ELSE/IF
IF @value = 0
		PRINT 'Value is zero.'
ELSE
	IF @value % 2 = 0
		PRINT 'Value is even.'
	ELSE
		PRINT 'Value is odd.'
```

20. Create SQL routine that uses a temp table.

```
-- Create temporary table
CREATE TABLE #TempTable(
	[CustomerID] INT NOT NULL,
	[OrderQuantity] INT
)

-- Insert values
INSERT INTO #TempTable (CustomerID, OrderQuantity)
VALUES (1, 4), (2, 2)
```

21. SQL that joins tables that are located in two databases.

```
--Combines personal data report from both the Bike and Store databases
--Where the OwnerID is equal to the CustomerID
SELECT *
FROM Store.Dbo.Customer AS DB1
FULL JOIN Bike.dbo.Owner AS DB2
ON DB1.CustomerID = DB2.OwnerID
```

22. SQL that joins tables that are located on two database servers.

```
--Cross-Server SQL requires the usage of Linked Servers and fully qualified table names.
--In this example I linked the local SQLEXPRESS server in the SQLEXPRESS01 Server.
--In this context the Customer data is stored on SQLEXPRESS01 and The Purchase Data is
stored on on SQLEXPRESS
SELECT DB1.CustomerID, DB1.FName, DB2.PurchaseID
FROM [DESKTOP-QEK9D7F\SQLEXPRESS].[Store].[dbo].[Customer] AS DB1
JOIN [DESKTOP-QEK9D7F\SQLEXPRESS01].[Store].[dbo].[Purchase] AS DB2
ON DB1.CustomerID = DB2.CustomerID
```

23. Create a table that includes a person's name, address, city, state/province, country. Must have at least 10,000 rows, no more than 5 duplicate names, at least 5 last names must have 100 but less than 250 occurrences.

unfinished during competition

24. Create a phone number table that supports table created in #23 (at least 90% of records in table from 23 must have a unique phone number).

```
-- SQL to create table
CREATE TABLE PhoneNbr(
	[PhoneNbrID] INT IDENTITY(1,1) NOT NULL,
	[PhoneNbr] CHAR(10) NOT NULL,
	[PersonID] INT NOT NULL
CONSTRAINT [PK_PhoneNbr] PRIMARY KEY CLUSTERED
(PhoneNbrID ASC)
ON [PRIMARY])

-- SQL to populate table:
DECLARE @i INT = 1
DECLARE @Endpoint INT
SELECT @Endpoint = COUNT(PersonID) From Person
WHILE (@i <= @Endpoint)
BEGIN
	DECLARE @PhoneNbr CHAR(10) = CAST(CONVERT(NUMERIC(10,0), RAND() * 899999999) +
2000000000 AS CHAR)

	INSERT INTO PhoneNbr (PhoneNbr, PersonID)
	SELECT @PhoneNbr, @i
	WHERE NOT EXISTS(
		SELECT 1 FROM PhoneNbr
		WHERE PhoneNbr = @PhoneNbr
	)
	SET @i = @i + 1
END
```

25. Create an updated SQL routine that includes a trigger for updating data in a second table.

unfinished during competition

26. Create a table with at least 10 columns of data and 1,000,000 rows. Run a series of select statements without an index and record the execution time. Create an index and rerun the select statements previously used. Record the execution time and comment on the run time comparison. Realize execution plans can be cached, which may impact your results. Your series of select statements should include selecting one particular record, selecting a small number of records (10 or so), selecting multiple records that would be randomly distributed (such as a phone book - find Jones, Miller, Roberts, Williams), selecting about 10% of the records.

unfinished during competition

27. Create a routine that captures errors (`@@ERROR`) and useful information that can be utilized in a variety of circumstances. An error logging table should be included.

```
CREATE TABLE ErrorTracking (
	ErrorID INT IDENTITY(1,1),
	ErrorDate DATETIME,
	ErrorType INT,
	ErrorUser NVARCHAR(50)
)

-- SQL to create table
DECLARE @errorHold INT = 0
INSERT INTO Category (CategoryID,Cname,CDesc)
VALUES (0/0,'Education','learning stuff')

BEGIN
SET @ErrorHold = @@Error
END
IF @errorHold != 0
BEGIN
	INSERT INTO ErrorTracking (ErrorDate,ErrorType,ErrorUser)
	VALUES (CURRENT_TIMESTAMP, @errorHold, current_user)
END
SELECT * FROM ErrorTracking
```

28. Create a routine that demonstrates characteristics of `UNION`.

```
--Union Statement to show all Dates related to Expiration and Purchase of products.
SELECT PurchaseDate AS Dates FROM Purchase
UNION
SELECT ExpirationDate FROM Product
```

29. Create a routine that uses `SELECT INTO` to populate a table from another table. Extra point for randomizing.

```
-- Randomly choose a Customer.
DECLARE @CustomerAmount INT
DECLARE @RandomCustomer INT
SET @CustomerAmount = (SELECT COUNT(CustomerID) FROM Customer)
SET @RandomCustomer = (ROUND(RAND()*@CustomerAmount+1,0))

--Based on their CustomerID, Insert their info into a new table called CustomerAudit.
SELECT *
INTO CustomerAudit
FROM Customer
WHERE CustomerID = @RandomCustomer
```

30. SQL routine that produces a graphical output such as a chart.

unfinished during competition

31. Create SQL routine that uses a subquery.

```
-- Create table
CREATE TABLE #SubqueryExample(
	[AGE] INT
)

-- Insert values
INSERT INTO #SubqueryExample (Age)
VALUES (1),(2),(22), (30)

-- Subquery
SELECT * FROM #SubqueryExample
WHERE Age > (
SELECT COUNT(*) FROM #SubqueryExample)
```

32. Capture the IP address of the connection and include it in your `@@ERROR` error logging routine.

```
ALTER TABLE ErrorTracking
ADD IPAdress VARCHAR(50);

-- First add an ip address column to the ErrorTracking table.
DECLARE @errorHold INT = 0
DECLARE @IP_Address VARCHAR(255);

SELECT @IP_Address = client_net_address
FROM sys.dm_exec_connections
WHERE Session_id = @@SPID;

-- Error caused on purpose
INSERT INTO Category (CategoryID,Cname,CDesc)
VALUES('Education',3)

BEGIN
SET @ErrorHold = @@ERROR
END
IF @errorHold != 0
BEGIN
	INSERT INTO ErrorTracking (ErrorDate,ErrorType,ErrorUser,IPAdress)
	VALUES
	(CURRENT_TIMESTAMP,@errorHold,current_user,@IP_Address)
END
```