IS-4420 Database Fundamentals

Lab 6, SQL Programming

- Indexes
- Views
- Stored Procedures
- Transactions

20 points

Summary

In Lab 6, we will practice creating indexes, views, stored procedures, and implementing transactions. Please be sure to review the PowerPoints, videos, and Zoom webinar for reference.

The deliverable for the lab is your .sql file with the various SQL objects.

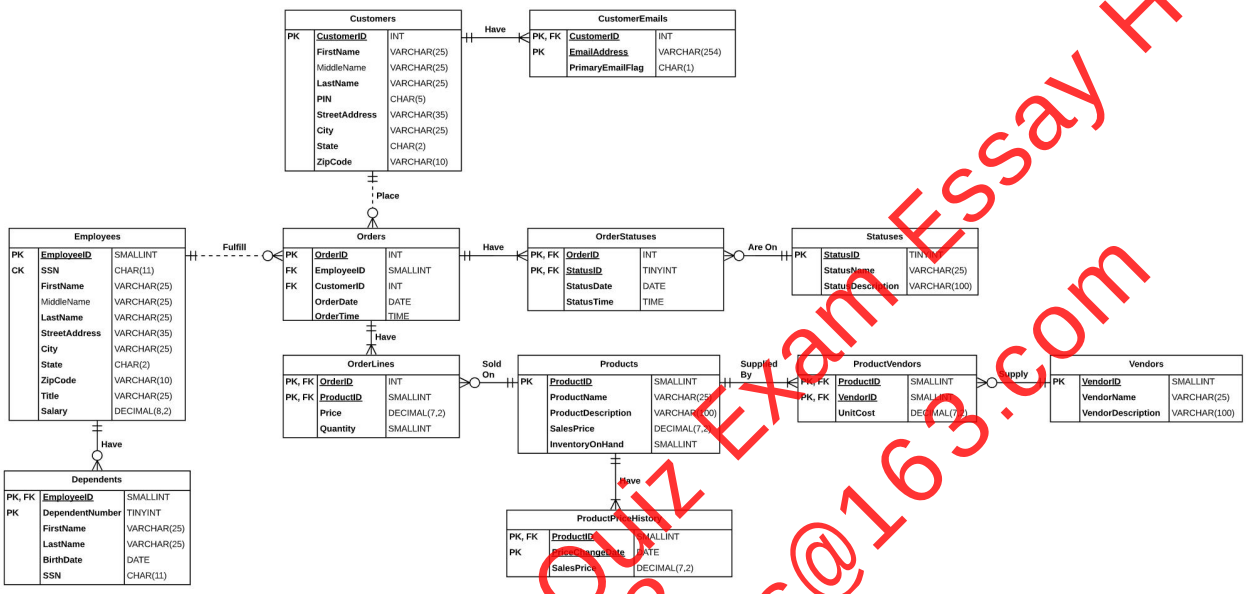
Setup

1. Connect to the SQL lab environment.
    a. If you have not yet connected to the SQL lab environment, follow the steps outlined in the **Connecting_To_IS4420_RDS_WebAccess.docx** file under the SQL Lab Setup module.
    b. Launch SQL Server Management Studio by clicking the Windows charm and typing SSMS. Change the authentication method from Windows to SQL, and then type Swoop for the username and Swoop for the password.
2. Retrieve the **Lab_6_Setup.sql** file that was provided.
3. Change the first 3 lines from Lab_6_<StudentName> to be your name.
    a. Example Lab_6_JacobCase.
4. Run the script to create the database for the lab.
5. Refresh the Databases menu on the left. Your Lab_6_<StudentName> database should be created.

Note

- You are encouraged to collaborate on homework assignments with your peers.
- Please refer to the PowerPoints & videos for SQL syntax.
- If you get stuck, attend tutoring office hours or send questions via email to the TAs.
- Assignments will take time, start them early.
- Class attendance will be rewarded by getting hints in class about the assignments.

Use the below ERD for reference while writing your procedural SQL scripts.



1- Create necessary indexes on tables with Surrogate Primary Keys. (4 points)

Whenever Surrogate Primary Keys are used, it's important to evaluate if it's necessary to enforce uniqueness on the table by creating secondary unique index. Let's evaluate tables with Surrogate Primary Keys and decide if they should have a secondary unique index:

Table	Needs Unique Index?	Unique Index Column(s)	Reason
Customers	Yes	FirstName, MiddleName, LastName, PIN	We can uniquify with PIN
Employees	Yes	SSN	SSN is a Natural Key
Orders	No	-	No candidate key exists and we don't want to limit potential Orders.
Statuses	Yes	StatusName	There should not be multiple Statuses with the exact same name.
Vendors	Yes	VendorName	There should not be multiple Vendors with the exact same name.
Products	Yes	ProductName	There should not be multiple Products with the exact same name.

Here is the syntax to create the unique ***composite*** index on the Customers table.

```
CREATE UNIQUE INDEX UIX_Customers_FullName_PIN ON Customers ( FirstName, MiddleName, LastName, PIN );
```

Here is the syntax to create the unique single-column unique index on the Employees table.

```
CREATE UNIQUE INDEX UIX_Employees_SSN ON Employees ( SSN );
```

Notice they both use the **UNIQUE** keyword, give the index a meaningful name, and specify the index columns in parentheses.

Refer to the screenshots above and the SQL_Programming PowerPoint to create the indexes on the Customers & Employees tables, and then use the same format to create the indexes on the Statuses, Vendors, and Products tables. Be sure to use the UNIQUE keyword and give them meaningful names in case they show up in error messages.

The deliverable for Question 1 is the 5 CREATE INDEX commands.

2- Create a View called SalesPerProductPerDay (4 points).

The SalesPerProductPerDay view should include these columns:

- ProductID
- ProductName
- OrderDate
- UnitsSold (count ProductID)
- SalesVolume (sum Price * Quantity from OrderLines)

Hints:

a. Start by SELECTing ProductID and ProductName FROM Products.
b. JOIN into OrderLines.
c. JOIN into Orders.
d. Add a column called UnitsSold that COUNTs ProductID.
e. Add a column called SalesVolume that SUMs Price * Quantity.
f. Add the appropriate GROUP BY clause.
g. Add: CREATE VIEW SalesPerProductPerDay in front of your SELECT query.
h. Run the command to create & store the View in the database.
i. Test the view by running the query: SELECT * FROM SalesPerProductPerDay.

3- Create a View called CustomerInfo (4 points):

The CustomerInfo view should include these columns:

- CustomerID
- FullName (concatenate FirstName, MiddleName, & LastName with spaces in between)
- Address (concatenate StreetAdddress, City, State, & ZipCode with spaces in between)
- PrimaryEmail (If customer has no email, show NULL for the customer's email address)

Hints:

a. Start by SELECTing CustomerID and concatenating FullName & Address from Customers.
b. JOIN into CustomerEmails.
    - We need the view to include Customers who don't have emails, so the join type is important here.
    - We also only want the Primary Email, so only include those rows from the CustomerEmails table.
    - Please keep in mind, you can have more than 1 predicate in joins. The ON clause might be a good place to specify you only want Primary Email rows from the CustomerEmails table.

4- Create a stored procedure called GetCustomerInfo & test it. (4 points)

The GetCustomerInfo stored procedure should take a parameter called @CustomerEmail and SELECT all columns from the CustomerInfo view from Question 3.

It should:

- Have 1 parameter called @CustomerEmail.
    i. Make sure the @CustomerEmail parameter has the same data type as the EmailAddress column in the CustomerEmails table.
- Declare a variable called @CustomerID.
    i. Make sure the @CustomerID variable has the same data type as the CustomerID column in the Customers table.
- Set the @CustomerID variable to the result of a subquery that reads from the CustomerEmails table WHERE EmailAddress = @CustomerEmail.
- SELECT all columns from the CustomerInfo view WHERE CustomerID = @CustomerID.

Hints:

a- Retrieve the Stored Procedure Template from the Lab 6 files.
b- Change the name to be GetCustomerInfo.
c- Replace the single-line comment in the parameters section with a parameter called @CustomerEmail with a data type of VARCHAR(254).
d- Inside the sproc body, declare a variable called @CustomerID with a data type of INT.
e- Set the @CustomerID variable to the result of a subquery that SELECTs CustomerID from the CustomerEmails table WHERE EmailAddress = @CustomerEmail.
f- SELECT all columns from the CustomerInfo view WHERE CustomerID = @CustomerID.
g- Run the CREAT PROCEDURE command to create the stored procedure.
h- Test the stored procedure by running:
EXECUTE GetCustomerInfo @CustomerEmail = 'tedcodd@dbbguys.com';

5- Create a stored procedure called UpdateProductPrice and test it. (4 points)

The UpdateProductPrice sproc should take 2 input parameters, ProductID and Price

Create a Stored Procedure that can be used to update the SalesPrice of a Product. Make sure the stored procedure also adds a row to the ProductPriceHistory table to maintain price history.

For full credit, the stored procedure must use a transaction.

Hints:

a- Retrieve the Stored Procedure Template from the Lab 6 files.
b- Change the name to be UpdateProductPrice
c- Add 2 parameters. Call the first one @ProductID with a data type of SMALLINT and the second one @SalesPrice with a data type of DECIMAL(7,2).
d- Type BEGIN TRANSACTION;, hit enter a few times, and type COMMIT TRANSACTION;
e- Inside the Transaction, write the command to UPDATE the SalesPrice column in the Products table equal to the value of the @SalesPrice parameter.
f- Next, write the command to INSERT the historical row into the ProductPriceHistory table. Set the PriceChangeDate column equal to CURRENT_DATE().
g- Run the CREAT PROCEDURE command to create the stored procedure.
h- Test the sproc by executing it and passing a ProductID & Price of your choice.

Lab 6 Deliverable:

Place all your commands to create views & stored procedures in a single .sql file called Lab_6_YourName.sql and submit it on Canvas.

For example:

```
-- Question 1

CREATE INDEX …


-- Question 2

CREATE VIEW SalesPerProductPerDay …


-- Question 3

CREATE VIEW CustomerInfo …


-- Question 4

CREATE PROCEDURE GetCustomerInfo …


-- Question 5

CREATE PROCEDURE UpdateProductPrice …
```