SQL Take-Home Assignment

Assignment Overview

This assignment is an introduction SQL. It starts with a very simple SELECT statement that selects all the columns and rows in a single table. The tutorial then builds on this SELECT statement to show how to select specific columns and then specific rows. The tutorial then continues with creating calculated fields, joining using multiple tables, aggregating rows, etc.

The SQL code covered in this assignment is standard ANSII SQL that (more or less) works on all databases and provides a good foundation for learning additional SQL commands like windowing. In addition to learning SQL, you will learn how to think "data". The data manipulation steps you learn in this assignment is common in data wrangling in general, e.g., many methods and functions in Python (mostly Pandas) allow you to perform similar functions as the SQL code you learn in this assignment. When using SQL, you also learn how to think about manipulating entire tables and columns as objects rather than single cell values. This thinking is also helpful in other applications, e.g., when using vectorized operations in Pandas.

This assignment is a take-home assignment and is not graded (you do not need to submit anything). I have provided tutorials (see blue URL links for each question label) and solution code for each question (at the end of this document). Unless you already have recent SQL experience, I strongly recommend that you review these recorded tutorials at least once. If something is unclear, then review the tutorial again and if needed seek help from me and the course TA (and others in the class) via Piazza or in class. If you work in groups, still create your own solution and please make sure that all group members actually learn SQL.

To assess your learning, you will later complete a skills test during class time. This skills test uses the same data and similar problems as this take-home assignment. The skills test consists of three questions and there are no new SQL commands introduced. The skills test is closed books, but you are allowed to use the SQL Summary Sheet (posted on the course website) – the SQL Summary Sheet will be provided inside the skills test. To do well on the skills test, it is important that you understand what you are doing in the take-home assignment; simply memorizing what you did in the take-home assignment is insufficient for you to do well on the skills test (and to use SQL to enhance your career). Please note that many of the exercises have suggestions for "additional" exercises. These are great for helping you prepare for the skills test. You will also find a separate section with even more additional problems that do not introduce any new function but that can still be used to get additional practice to help you prepare for the skills test. To do well on the skills test you need know how to solve all the exercises in this document (including the additional exercises within the main part of the document).

Take-home Assignment Questions

You are going to create **19** SQL queries using the Orders, Order Details, Products, and Customers tables (see solutions and the end of this document). The first eleven queries are designed to simply get you used to SQL – it gets more interesting after Q11 when we start working with table joins and aggregate queries.

Q1: Use the Customers table and create a query that shows all columns.

- Hint…91 records should be displayed

Q2: Use the Customers table and create a query that shows the CompanyNames and Addresses.

- Hint…91 records and two columns should be displayed

Q3: Use the Customers table and create a query that shows the CompanyNames and Addresses for all customers located in the 'UK'.

- Hint…7 records and two columns should be displayed

Q4: Use the Customers table and create a query that shows the CompanyNames and Addresses for all customers located in the 'UK' or the 'USA'.

- Hint…20 records and two columns should be displayed

Q5: Use the Customers table and create a query that shows CompanyNames, Addresses, Countries, and CustomerIDs. Use wildcards (% for any character and any number of characters and _ for a single character) and select all customers with CompanyNames with the letters 'po' anywhere in their name that are located in the 'USA' or the 'UK'. When including wildcards in a comparison, you always need to use the LIKE operator. Postgres is case sensitive for string comparisons using = and LIKE operators, e.g., CompanyName LIKE '%PO%' is not the same as CompanyName LIKE '%po%' and Country = 'UK' is not the same as Country = 'uk'. SQLite is case sensitive for = and case insensitive for LIKE.

- Hint…2 records and four columns should be displayed

As an additional exercise also create a query that finds all CompanyNames that starts with 'L' and second query that finds all CompanyNames that ends with 'e'.

Q6: Using Order Details table create a query that includes a calculated field named LineItemTotal that is calculated as UnitPrice*Quantity*(1-Discount). Round LineItemTotal to two decimals.

- Display only the following fields: OrderID, ProductName, Discount, and calculated LineItemTotal.
- Check your results…. Order 10248 should have LineItemTotal of 168.00.

Q7: Using Order Details table create a query that includes a calculated field named LineItemTotal that is calculated as UnitPrice*Quantity*(1-Discount). The query should show all order line items with high discounts (greater than or equal to 20%) AND medium to high line item total (greater than $500 but less than or equal to $1000)) OR with medium to high discounts (greater than or equal to 15% but less than 20%) AND high line item total (greater than $1000)).

- Sort in descending order by LineItemTotal.
- Display only the following fields: OrderID, ProductName, Discount, and calculated LineItemTotal.
- Check your results…. there should be 88 records.

Q8: Using the Orders table, find all orders with OrderDate earlier than 1994-11-31. When working with dates you need to enclose the date in quotation marks (single quotation marks are typically used) and use the date format used by the RDMBS (in SQLite and postgres: YYYY-MM-DD).
- Display only the following fields: CustomerID and OrderDate.
- Hint…95 records should be displayed.

Q9: Using the Orders table and the BETWEEN operator, find all orders with OrderDates between 1996-05-01 and 1996-05-31. The between operator selects all records with values within the range specified (including the beginning and end values) and can be used with numbers, text, or dates. As an additional exercise, create a query that returns the same results but instead uses > and < comparison operators. As another additional exercise, create a query using the BETWEEN operator that returns all records in the OrderDetails table where the quantity is between 10 and 30.
- Hint…77 records should be displayed.

Q10: Using the Orders table, find all orders between 1996-05-01 and 1996-05-31 that have not yet been shipped (missing shipping date; note that empty cells are referred to as 'Null' values in database lingo).
- Display only the following fields: CustomerID, OrderDate, ShippedDate
- Hint…12 record should be displayed.

Q11: Using the Orders table, create a query that displays all orders placed between 1996-05-01 and 1996-05-31. For each order displayed, create a new field named 'OrderStatus' that displays the message: 'Not shipped' if the ShippedDate is Null, and otherwise displays the message 'Shipped'. Sort the results in ascending order by OrderStatus.
- Display only the following fields: CustomerID, OrderDate, ShippedDate and OrderStatus.
- Use a CASE expression in the SELECT clause to display the appropriate order status message, e.g., CASE WHEN *comparison 1* THEN *result 1* ELSE *result_else* END. Note that multiple "WHEN *comparison* THEN *result*" expressions can be added and that the "ELSE *result_else*" expression is not required)
- Hint….77 records should be displayed and 12 records should have OrderStatus stating 'Not Shipped'.

More about Null Values

The data you work with will often contain null values. It is additionally common that null values are introduces by your own analyses, e.g., when performing left outer joins (see below). When SQL encounters null values, you can get some surprising results if you are not aware of how SQL handles null values. You will be impacted two ways by this:

Calculations
Expressions that calculate new values will result in null if one or more columns in the expression is null. For example, in the calculation:
QntOnHand + QntOnOrder - QntOutstanding As TotalQntAvailable
If QntOutstanding has one or more rows with null values, then TotalQntAvailable will also be null for these rows. If we truly do not know what the null value means, then that is fine. However, if we know that a null value in QntOutstanding means that nothing (zero) is outstanding, then we do not want to have the expression return null. To overcome this you can either create a CASE statement that checks for null values and then uses different formulas to handle the inclusion/exclusion of null, for example:
QntOnHand + QntOnOrder - CASE WHEN QntOutstanding IS NULL THEN 0 ELSE QntOutstanding END As TotalQntAvailable,

Alternatively, use COALESCE (it is available in all databases that I am aware of expect MS Access – MS Access has a function called IFNULL that is similar). COALESCE() returns the first non-null value in a list of expressions and is commonly used to replace null values with 0s in expressions:
QntOnHand + QntOnOrder – COALESCE(QntOutstanding, 0) As TotalQntAvailable

Comparisons
Expressions that compare values, e.g., WHERE, HAVING, and CASE statements, will return unknown (null) when a null value is being compared. Here is an overview of what is returned by various logical comparisons in postgres (note that null values are unknown):

Comparison	Returns
1=1	true
1=0	false
null=null	unknown (i.e., NULL)
null=1	unknown (i.e., NULL)
1<>1	false
1<>0	true
null<>null	unknown (i.e., NULL)
null<>1	unknown (i.e., NULL)

As an example, say you want to evaluate whether quantity outstanding is equal to 0 using a CASE statement, e.g.:
CASE WHEN QntOutstanding = 0 THEN 'Order fulfilled' ELSE 'Order is outstanding' END.

This CASE statement returns 'Order is outstanding' for all records where quantity outstanding is null (i.e., when it is missing) – which might not be the intended behavior (e.g., if QntOutstanding with NULL means that nothing is outstanding). It is doing this because unknown is not considered True (it is also not considered False, but the comparison is not checking for False). To fix this you again can check whether quantity on hand is null using a CASE statement or use COALESCE to convert null values to zeros.

Aggregate Functions

Aggregate functions “ignore” NULL values, e.g., count, counts the number of non-null values, average returns the average of the non-null values, and max returns the largest number of the non-null values.

Q12: Using the Order Details and Orders tables (note that you need to join these tables - when you decide which fields to use in the join it might help to first figure out how you would manually related records in the two tables – and ERD diagram might also be helpful) create a query that shows GrossLineItemTotal calculated as UnitPrice*Quantity. When working with joins, note that JOIN and INNER JOIN means the same thing.

- Display the following fields: OrderID, CustomerID, ProductName, Discount, and calculated GrossLineItemTotal.
- Check your results…. there should be 2157 records and order 10248 should have GrossLineItemTotal of 168.

Q13: Using the Order Details and Orders tables create a query that shows all order line items with high discounts (greater than or equal to 20%) AND high GrossLineItemTotal (greater than $1000) calculated as UnitPrice*Quantity.

- Display only the following fields: OrderID, CustomerID, ProductName, Discount, and calculated GrossLineItemTotal.
- Check your results…. there should be 65 records, order 10847 should be from SAVEA and for product 'Chai' the discount should be 0.2 with a GrossLineItemTotal of 1,440.

Before moving on make sure you **understand** what table joins do (also review your lecture notes on this topic). Note that SQLite only supports LEFT OUTER, INNER and CROSS joins, postgres additionally supports RIGHT OUTER JOIN and FULL OUTER joins.

Q14: Using the Order Details table, create a query that displays all OrderID, Discount, and a calculated field *LineItemTotal* calculated as UnitPrice*Quantity*(1-Discount).

- Hint…2157 records should be displayed; the line item total for the first item of OrderID 10248 is 168.

Q15: Using the Order Details table, create a query that displays all orders grouped by OrderID that sums the calculated field LineItemTotal and averages the discount for each order. You need an aggregate query for this (you add GROUP BY field after the FROM clause and change the SELECT statement to use aggregate functions, e.g., sum(), avg(), min(), max(), on the selected fields not included in the GROUP BY.

- Display only the following fields: OrderID, SumOfLineItemTotal, and AverageDiscount.
- Hint…831 records should be displayed; the average discount for order 10250 is 0.1, the sum of all the line item totals for OrderID 10248 is 440.

Before moving on make sure you **understand** what table aggregation does. Also review your lecture notes.

Q16 (tutorials a, b, and c): Using the Q15 query result set (see help below) and the orders table, create a query that displays all orders and the order total (name this field OrderTotal, calculated as SumOfLineItemTotal + Freight). Sort the results in descending order by OrderTotal and only show the 8 first rows. Round the OrderTotal to two decimals.

- To use the Q15 result set as input in Q16 you need to create a VIEW using the SELECT statement created in Q15. To create a VIEW named OrderDetailsSum enter: "CREATE VIEW OrderDetailSums AS" immediately before the SELECT statement.
- Display only the following fields: OrderID, CustomerID, ShipName, Freight, SumOfLineItemTotal, and the calculated OrderTotal.
- Hint…8 records should be displayed; the Order total for OrderID 10865 is 16735.64.

  To change the view (if you want to update the SQL code in the view) you first need to drop the view and then create it again, e.g.,:

```
DROP VIEW OrderDetailSums;
CREATE VIEW OrderDetailSums AS
    SELECT...
```

As additional exercises create the result in Q16 that combines Q15 and Q16, i.e., the aggregation and the join is performed in one query:

- Q16 additional 1 - Solve Q16 using a subquery. The subquery will replace the reference to the view OrderDetailSums in the inner join. By creating an alias for the subquery (outside the parentheses) you can then reference the subquery in the other parts of the outer query.
- Q16 additional 2 - Second solve this by joining Orders and OrderDetails and then think about how the joined results set should be aggregated, i.e., which aggregate functions should be used for the different columns (sum(), avg(), max(), min(), etc.). Note for example that after joining the two tables the freight for a single order will be repeated for each order line item, e.g., if an order has Freight of $18 and this order has three line items then $18 will be repeated three times. When calculating order total we want to add $18 (not $54) to the order total.

  Note that in most flavors of SQL, including postgres, when aggregate queries are used, each column listed in the SELECT clause has to either be part of an aggregate function or has to be grouped by. For example, in this query we group by OrderID, but we also want to include CustomerID. Because all line items for each order have the same CustomerID, we can either include CustomerID in the group by clause or we can use an aggregate functions such as min, max, or avg. In SQLite, CustomerID can be included without including customerid in the group by clause or in an aggregate function. SQLite simply uses CustomerID values from arbitrarily selected rows within each group (the first row in each group, but what the first row is in each group is not always guaranteed to be the same).

Q17: Create a view named AveragePriceReceived using the Order and the Order Details tables find the average price received (Average of UnitPrice*(1-Discount)) for each product sold 1/1/1996 or later (based on the OrderDate).

- Name the calculated field AvgProductPrice.
- Use a group by statement to calculate the average price received for each product where the OrderDate is equal to or greater than 1/1/1996.
- Note that in aggregate queries a WHERE clause is used is filter records before the data is grouped. A HAVING clause is used to filter records after the data is grouped. In this instance we only want to include sales in our calculating that are rom 1/1/1996 or later and we as such want to filter to records before the grouping occurs.
- Display only the following fields: ProductID and calculated AvgProductPrice.
- Check your results (to check the view, run: SELECT * FROM AveragePriceReceived;) …. there should be 76 records, and the average price for Product 1 should be $16.5176…

Q18: Using AveragePriceReceived and the Product table (you need to join the two) compare the average price received for each product (from AveragePriceReceived) to the list price (PricePerUnit) in the Product tables and find the top 25 products with the highest PercentPriceDiff.

- Calculate two fields: PriceDiff showing list price minus average price, and PercentPriceDiff showing the percentage difference between the average price and the list price (PriceDiff /AvgProductPrice).
- Multiply PercentPriceDiff by 100.
- Round PriceDiff and PercentPriceDiff to two decimals.
- Display only the following fields: ProductID, and calculated fields PriceDiff and PercentPriceDiff.
- Check your results…. there should be 25 records, and for ProductID 14 the PriceDiff should be 3.72 and the PercentPriceDiff should be 19.05 percent.

As additional exercises, create the results from Q18 but instead of using the view use (1) a subquery and (2) aggregate functions (this requires three tables to be joined).

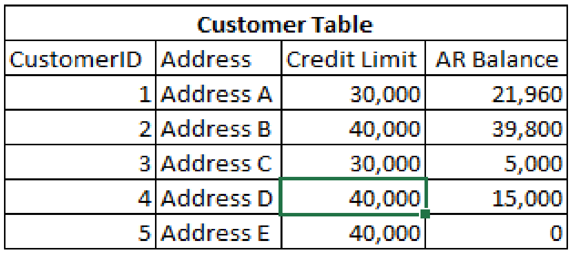
Q19: Using the Orders and the Customer tables find orders that are shipped to an address that does not exist in the customer table. Note that LEFT JOIN and LEFT OUTER JOIN means the same thing.

- Display only the following fields: OrderID, CustomerID, OrderDate, ShipName and ShipAddress.
- Check your results…. there should be 48 records.

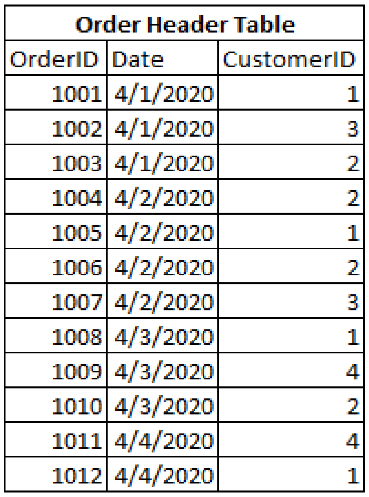
As two additional exercises:
Q19 additional 1: find orders that are shipped to an existing address but to the wrong customer (there should be 2 records).
Q19 additional 2: find orders that are shipped to either an existing address but to the wrong customer or to an address that does not exist in the customers table (there should be 50 records).

More about INNER JOINS and LEFT JOINS
The difference between an inner join and an outer join can best be explained by an example (I also assume you remember this from the lecture and that you watched the tutorial that explains different join types - the one with the Venn diagrams).

Assume you have a table with customers and another table with order headers. The customers table have a CustomerID column and information about customers, e.g., address, phone number, credit limit, and AR balance. The order headers table lists all past orders, including OrderID, date, and CustomerID (to indicate who placed the order). Further, assume that you can have some customers that have never placed an order (perhaps they are new customers). Here is some mock data:



Customer Table			
CustomerID	Address	Credit Limit	AR Balance
1	Address A	30,000	21,960
2	Address B	40,000	39,800
3	Address C	30,000	5,000
4	Address D	40,000	15,000
5	Address E	40,000	0

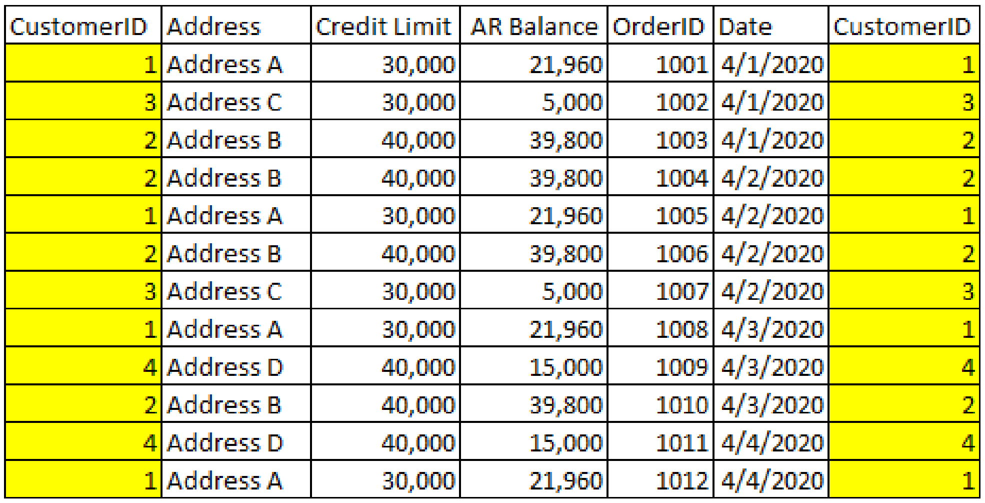


Order Header Table		
OrderID	Date	CustomerID
1001	4/1/2020	1
1002	4/1/2020	3
1003	4/1/2020	2
1004	4/2/2020	2
1005	4/2/2020	1
1006	4/2/2020	2
1007	4/2/2020	3
1008	4/3/2020	1
1009	4/3/2020	4
1010	4/3/2020	2
1011	4/4/2020	4
1012	4/4/2020	1

Inner joins return all data where there are matching records based on the columns defined in the ON statement. Left outer joins return all the records in the left table (the first table in the select statement) and all the matching records in the right table (but only the matching records from the right table). So if we have:

```
SELECT A.*, B.*
FROM Customers A
INNER JOIN OrderHeaders B
ON A.CustomerID = B.CustomerID;
```

Then we will get all customers that have placed orders (and all orders that have customers), but not the customers that have not placed any orders, i.e., the results would be as shown on the right (I highlighted the columns that were used to match the rows). Note that CustomerID 5 is not part of this result.

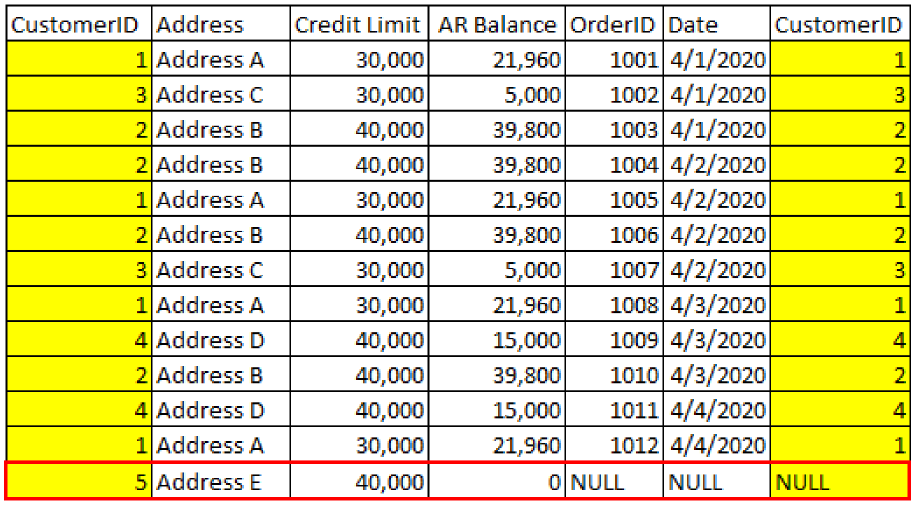


CustomerID	Address	Credit Limit	AR Balance	OrderID	Date	CustomerID
1	Address A	30,000	21,960	1001	4/1/2020	1
3	Address C	30,000	5,000	1002	4/1/2020	3
2	Address B	40,000	39,800	1003	4/1/2020	2
2	Address B	40,000	39,800	1004	4/2/2020	2
1	Address A	30,000	21,960	1005	4/2/2020	1
2	Address B	40,000	39,800	1006	4/2/2020	2
3	Address C	30,000	5,000	1007	4/2/2020	3
1	Address A	30,000	21,960	1008	4/3/2020	1
4	Address D	40,000	15,000	1009	4/3/2020	4
2	Address B	40,000	39,800	1010	4/3/2020	2
4	Address D	40,000	15,000	1011	4/4/2020	4
1	Address A	30,000	21,960	1012	4/4/2020	1

If we instead use a left outer join:

```
SELECT A.*, B.*
FROM Customers A
LEFT OUTER JOIN OrderHeaders B
ON A.CustomerID = B.CustomerID;
```

Then we would get the following result (which now includes CustomerID 5):



CustomerID	Address	Credit Limit	AR Balance	OrderID	Date	CustomerID
1	Address A	30,000	21,960	1001	4/1/2020	1
3	Address C	30,000	5,000	1002	4/1/2020	3
2	Address B	40,000	39,800	1003	4/1/2020	2
2	Address B	40,000	39,800	1004	4/2/2020	2
1	Address A	30,000	21,960	1005	4/2/2020	1
2	Address B	40,000	39,800	1006	4/2/2020	2
3	Address C	30,000	5,000	1007	4/2/2020	3
1	Address A	30,000	21,960	1008	4/3/2020	1
4	Address D	40,000	15,000	1009	4/3/2020	4
2	Address B	40,000	39,800	1010	4/3/2020	2
4	Address D	40,000	15,000	1011	4/4/2020	4
1	Address A	30,000	21,960	1012	4/4/2020	1
5	Address E	40,000	0	NULL	NULL	NULL

Additional extra exercises (without tutorials, see solutions below):
Many of the following practice questions are asking questions about Northwind's vendors (supplier) and the products that Northwind purchases from these vendors. The questions also asks about sales of products associated with different vendors, which is slightly non-intuitive as vendors interact with Northwind when Northwind purchases items (rather than when Northwind sales items). However, we do not have purchases data in this database.

1. Using the Vendors and Products tables, create a list of vendors and the products that Northwind purchases from respective vendor and only show vendors with associated products. Show CompanyName from Vendors and ProductName from products.
2. Using the Vendors and Products tables, show how many products Northwind purchases from each respective vendor? Show CompanyName from Vendors and a calculated field named NumberOfProductSales.
3. Use the Vendors, Products, and OrderDetailst tables and determine how many times have each supplier's products been sold by Northwind (assume that the supplier of a given product has not changed)? Show CompanyName from Vendors, ProductName from products, and a calculated field named NumberOfProductSales.
4. Use the Vendors, Products, and OrderDetailst tables and determine, for each vendor, how many times have Northwind sold items purchased from that vendor? Show CompanyName from Vendors and a calculated field named NumberOfProductSales.

5. Answer questions 3 and 4, but only for sales in 1996 (based on OrderDate) – you also need the Orders table for this analysis.
6. Answer question 5, but only show rows with counts that are higher than 20.
7. Using the Products and OrderDetails tables, find products that have never been sold. Include all columns from the Products table (note that this will return 0 rows, i.e., all products have been sold at least once).
8. Using the Products, OrderDetails, and Orders tables, find products that have not been sold in 1996. Include all columns from the Products table.
9. Using the Orders and OrderDetails tables, calculate total sales (including freight from the Orders table and line item totals from the OrderDetails table) for each customer. Include CustomerID from Orders and a calculated field named TotalSalesPerCustomer.

Solutions

The solutions below work in postgres. Some video tutorials create slightly different solutions that work in SQLite. These differences have been highlighted below.

Q1:

```
SELECT * FROM Customers;
```

Q2:

```
SELECT CompanyName, Address
FROM Customers;
```

Q3:

```
SELECT CompanyName, Address
FROM Customers
WHERE Country = 'UK';
```

Comment: Most RDBMS, including postgres, use double quotation marks to indicate quoted identifiers and single quotation marks to indicate string literals. SQLite (and MySQL) will interpret double-quoted strings as string literal if it does not match any valid identifier, i.e., in SQLite we can use either "UK" or 'UK" in the code above (assuming "UK" is not an identifier).

Q4:

```
SELECT CompanyName, Address
FROM Customers
WHERE Country = 'UK' OR Country = 'USA';
```

Q5:

```
SELECT CompanyName, Address, Country, CustomerID
FROM Customers
WHERE (Country = 'UK' OR Country = 'USA') AND CompanyName Like '%po%';
```

Q6:

```
SELECT OrderID, ProductName, Discount, round(UnitPrice*Quantity*(1-Discount),2) AS
LineItemTotal
FROM OrderDetails;
```

Q7:

```
SELECT OrderID, ProductName, Discount, round(UnitPrice*Quantity*(1-Discount),2) AS
LineItemTotal
FROM OrderDetails
WHERE Discount >=0.2 AND UnitPrice*Quantity*(1-Discount) > 500 AND
UnitPrice*Quantity*(1-Discount) <= 1000 OR Discount >=0.15 AND Discount <0.2 AND
UnitPrice*Quantity*(1-Discount) > 1000
ORDER BY LineItemTotal DESC;
```

Comment: In postgres, output expression alias names can be used to refer to the expressions in ORDER BY and GROUP BY clauses, but not in the WHERE or HAVING clauses (you must then write out the expressions again).

Q8:

```
SELECT CustomerID, OrderDate
```

```
      FROM Orders
      WHERE OrderDate < '1994-11-31';

Q9:   SELECT CustomerID, OrderDate
      FROM Orders
      WHERE OrderDate BETWEEN '1996-05-01' AND '1996-05-31';

Q10:  SELECT CustomerID, OrderDate, shippeddate
      FROM Orders
      WHERE OrderDate BETWEEN '1996-05-01' AND '1996-05-31' AND ShippedDate IS NULL;

Q11:  SELECT CustomerID, OrderID, OrderDate, ShippedDate, CASE WHEN ShippedDate IS
      NULL THEN 'Not shipped' ELSE 'Shipped' END AS OrderStatus
      FROM Orders
      WHERE OrderDate BETWEEN '1996-05-01' AND '1996-05-31'
      ORDER BY OrderStatus asc;

Q12:  SELECT A.OrderID, A.CustomerID, B.ProductName, B.Discount, B.Quantity*B.UnitPrice
      AS GrossLineItemTotal
      FROM Orders A
      INNER JOIN OrderDetails B
      ON A.OrderID = B.OrderID;

Q13:  SELECT A.OrderID, A.CustomerID, B.ProductName, B.Discount, B.Quantity*B.UnitPrice
      AS GrossLineItemTotal
      FROM Orders A
      INNER JOIN OrderDetails B
      ON A.OrderID = B.OrderID
      WHERE B.Discount >= 0.2 AND B.Quantity*B.UnitPrice > 1000;

Q14:  SELECT OrderID, Discount, UnitPrice*Quantity*(1-Discount) AS LineItemTotal
      FROM OrderDetails;

Q15:  SELECT OrderID, round(avg(Discount),2) AS Discount, round(sum(UnitPrice*Quantity*(1-
      Discount)),2) AS SumOfLineItemTotal
      FROM OrderDetails
      GROUP BY OrderID;
```

Alternatively (either approach is fine), calculate a weighted average (weighted by the discount dollar amount on each row):

```
      sum(UnitPrice*Quantity*(Discount))/sum(UnitPrice*Quantity) AS AverageDiscount

Q16:  CREATE VIEW OrderDetailSums AS
```

```
SELECT OrderID, round(avg(Discount),2) AS Discount, round(sum(UnitPrice*Quantity*(1-
Discount)),2) AS SumOfLineItemTotal
FROM OrderDetails
GROUP BY OrderID;
```

```
SELECT A.OrderID, A.CustomerID, A.ShipName, A.Freight, B.SumOfLineItemTotal,
round(A.Freight + B.SumOfLineItemTotal, 2) As OrderTotal
FROM Orders A
INNER JOIN OrderDetailSums B
ON A.OrderID = B.OrderID
ORDER BY OrderTotal DESC LIMIT 8;
```

Alternative 1 – subquery

```
SELECT A.OrderID, A.Freight, B.SumOfLineItemTotal, round(A.Freight +
B.SumOfLineItemTotal, 2) As OrderTotal
FROM Orders A
INNER JOIN (SELECT OrderID, SUM(UnitPrice*Quantity*(1-Discount)) AS
SumOfLineItemTotal FROM OrderDetails GROUP BY OrderID) B
ON A.OrderID = B.OrderID
ORDER BY OrderTotal DESC LIMIT 8;
```

Alternative 2 – Join then Aggregate

```
SELECT A.OrderID, avg(A.Freight) AS Freight, sum(B.UnitPrice*Quantity*(1-Discount)) AS
SumOfLineItemTotal, round(avg(A.Freight) + sum(B.UnitPrice*Quantity*(1-Discount)), 2)
As OrderTotal
FROM Orders A
INNER JOIN OrderDetails B
ON A.OrderID = B.OrderID
GROUP BY A.OrderID
ORDER BY OrderTotal DESC LIMIT 8;
```

Q17:

```
CREATE VIEW AveragePriceReceived AS
SELECT B.ProductID, avg(B.UnitPrice*(1-B.Discount)) AS AvgProductPrice
FROM Orders A
INNER JOIN OrderDetails B
ON A.OrderID = B.OrderID
WHERE A.OrderDate >= '1996-01-01'
GROUP BY B.ProductID;
```

Q18:

```
SELECT A.ProductID, round(A.PricePerUnit - B.AvgProductPrice,2) AS PriceDiff,
round(100*(A.PricePerUnit - B.AvgProductPrice)/B.AvgProductPrice,2) AS
PercentPriceDiff
FROM Products A
INNER JOIN AveragePriceReceived B
```

```
ON A.ProductID = B.ProductID
ORDER BY PercentPriceDiff DESC LIMIT 25;
```

Alternative 1 – subquery

```
SELECT A.ProductID, round(A.PricePerUnit - B.AvgProductPrice,2) AS PriceDiff,
round(100*(A.PricePerUnit - B.AvgProductPrice)/B.AvgProductPrice,2) AS
PercentPriceDiff
FROM Products A
INNER JOIN (SELECT B.ProductID, avg(B.UnitPrice*(1-B.Discount)) AS AvgProductPrice
FROM Orders A
INNER JOIN OrderDetails B
ON A.OrderID = B.OrderID
WHERE A.OrderDate >= '1996-01-01'
GROUP BY B.ProductID) AS B
ON A.ProductID = B.ProductID
ORDER BY PercentPriceDiff desc LIMIT 25;
```

Alternative 2 – Join then Aggregate

```
SELECT A.ProductID, round(min(A.PricePerUnit) - avg(B.UnitPrice*(1-B.Discount)),2) AS
PriceDiff, round(100*(min(A.PricePerUnit) - avg(B.UnitPrice*(1-
B.Discount)))/avg(B.UnitPrice*(1-B.Discount)),2) AS PercentPriceDiff
FROM Products A
INNER JOIN OrderDetails B ON A.ProductID = B.ProductID
INNER JOIN Orders C ON B.OrderID = C.OrderID
WHERE C.OrderDate >= '1996-01-01'
GROUP BY A.ProductID
ORDER BY PercentPriceDiff desc Limit 25;
```

Q19:

```
SELECT A.OrderID, A.CustomerID, A.OrderDate, A.ShipName, A.ShipAddress
FROM Orders A
LEFT OUTER JOIN Customers B ON A.ShipAddress = B.Address
WHERE B.CustomerID IS NULL;
```

Alternative 1

```
SELECT A.OrderID, A.CustomerID, A.OrderDate, A.ShipName, A.ShipAddress
FROM Orders A
JOIN Customers B ON A.ShipAddress = B.Address
WHERE A.CustomerID <> B.CustomerID;
```

Alternative 2

```
SELECT A.OrderID, A.CustomerID, A.OrderDate, A.ShipName, A.ShipAddress
FROM Orders A
LEFT OUTER JOIN Customers B ON A.ShipAddress = B.Address
WHERE B.Address IS NULL OR A.CustomerID <> B.CustomerID;
```

```
SELECT A.OrderID, A.CustomerID, A.OrderDate, A.ShipName, A.ShipAddress
FROM Orders A
LEFT OUTER JOIN Customers B ON A.CustomerID = B.CustomerID
WHERE A.ShipAddress <> B.Address;
```

Extra Practice Problems

1)
```
SELECT CompanyName, ProductName
FROM Vendors A
INNER JOIN Products B
ON A.SupplierID = B.SupplierID;
```

2)
```
SELECT CompanyName, Count(ProductName) As NumberofProducts
FROM Vendors A
INNER JOIN Products B
ON A.SupplierID = B.SupplierID
Group By CompanyName;
```

3)
```
SELECT A.CompanyName, B.ProductName, Count(B.ProductName)
        NumberOfProductSales
FROM Vendors A
INNER JOIN Products B ON A.SupplierID = B.SupplierID
INNER JOIN OrderDetails C ON B.ProductID = C.ProductID
GROUP BY A.CompanyName, B.ProductName;
```

Need to also group by A.CompanyName if a single product can have different vendors (but the question indicated that you can assume that this is not possible).

4)
```
SELECT A.CompanyName, Count(A.CompanyName) NumberOfSales
FROM Vendors A
INNER JOIN Products B ON A.SupplierID = B.SupplierID
INNER JOIN OrderDetails C ON B.ProductID = C.ProductID
GROUP BY A.CompanyName;
```

5)
```
SELECT A.CompanyName, B.ProductName, Count(B.ProductName)
        NumberOfProductSales
FROM Vendors A
INNER JOIN Products B ON A.SupplierID = B.SupplierID
INNER JOIN OrderDetails C ON B.ProductID = C.ProductID
INNER JOIN Orders D ON C.OrderID = D.OrderID
WHERE D.OrderDate >= '1996-01-01' AND D.OrderDate < '1997-01-01'
GROUP BY A.CompanyName, B.ProductName;
```

```
SELECT A.CompanyName, Count(A.CompanyName) NumberOfSales
FROM Vendors A
INNER JOIN Products B ON A.SupplierID = B.SupplierID
INNER JOIN OrderDetails C ON B.ProductID = C.ProductID
INNER JOIN Orders D ON C.OrderID = D.OrderID
WHERE D.OrderDate >= '1996-01-01' AND D.OrderDate < '1997-01-01'
```

```
GROUP BY A.CompanyName;
```

6)
```
SELECT A.CompanyName, B.ProductName, Count(B.ProductName)
        NumberOfProductSales
FROM Vendors A
INNER JOIN Products B ON A.SupplierID = B.SupplierID
INNER JOIN OrderDetails C ON B.ProductID = C.ProductID
INNER JOIN Orders D ON C.OrderID = D.OrderID
WHERE D.OrderDate >= '1996-01-01' AND D.OrderDate < '1997-01-01'
GROUP BY A.CompanyName, B.ProductName
HAVING Count(B.ProductName) > 20;

SELECT A.CompanyName, Count(A.CompanyName) NumberOfSales
FROM Vendors A
INNER JOIN Products B ON A.SupplierID = B.SupplierID
INNER JOIN OrderDetails C ON B.ProductID = C.ProductID
INNER JOIN Orders D ON C.OrderID = D.OrderID
WHERE D.OrderDate >= '1996-01-01' AND D.OrderDate < '1997-01-01'
GROUP BY A.CompanyName
HAVING Count(A.CompanyName)> 20;
```

7)
```
SELECT A.*
FROM Products A
LEFT JOIN OrderDetails B ON A.ProductID = B.ProductID
WHERE B.ProductID IS NULL;
```

8)
```
SELECT A.*
FROM Products A
LEFT JOIN (SELECT * FROM OrderDetails B INNER JOIN Orders C ON B.OrderID =
        C.OrderID WHERE OrderDate >= '1996-01-01' AND OrderDate < '1997-01-01') B
        ON A.ProductID = B.ProductID
WHERE B.ProductID IS NULL;
```

9)
```
SELECT A.CustomerID, SUM(A.Freight + B.LineItemAmount) AS
        TotalSalesPerCustomer
FROM Orders A
INNER JOIN (SELECT OrderID, SUM(UnitPrice*Quantity*(1-Discount)) AS
        LineItemAmount FROM OrderDetails GROUP BY OrderID) B ON A.OrderID =
        B.OrderID
GROUP BY A.CustomerID;
```