More SQL Practice Problems (MySQL version)

40 intermediate and advanced challenges for you to solve using a "learn-by-doing" approach

Sylvia Moestl Vasilik

Table of Contents

How to use this book

I was happy to get many emails from purchasers of my first book, SQL Practice Problems, telling my how useful the problems were in helping them learn SQL. And many of them asked for more practice problems! This book was written because of these repeated requests.

Is this an introductory book for SQL learners? No, it isn't. This book assumes that you've worked through the problems in SQL Practice Problems recently, or have equivalent skills. The basic concepts of SQL Select statements (including joins, left joins, grouping, where clauses, etc.) should be familiar to you. There are a few easier questions at the start of this book, but it has no beginner level questions.

The database used for these practice problems is based on the AdventureWorks database, which is one of the sample databases that's a part of Microsoft SQL Server. However, I've made numerous changes to it, both to make it work for MySQL and to create interesting data problems. Do *not* try to use the original AdventureWorks database for the problems, you will not get the right results.

What's the best way to work through this book?

I suggest that you not look at the hints at all, unless you're stuck.

Why is this? Recent brain research shows that if you need to struggle before coming up with the answer, that struggle will make it *much* more likely that you'll remember how to solve similar problems in real life. And in real life, there are no hints!

I've specifically created a no-hint version of this book (available as part of the same download) without hints and answers. I suggest that you use this no hint version, and only refer to the full version if you need to.

But if you're completely stuck and need some direction, please do go ahead and look at the hints. They are designed to help you work through the problem one step at a time, without giving too much away. The hints will guide you through one specific approach to the problem, though there may be many different answers.

You will need to research online, using your favorite search engine. The technical question and answer website https://stackoverflow.com/ is an outstanding resource. If you're stuck, searching online using regular language will usually be helpful. For instance, try searching for this string:

> SQL how to do an if then statement in a where clause

The first results that come up explain the Case statement, which is what you would use in this situation.

My comment from the previous book on research online still applies:

> Should you search online for answers, examples, etc.? Absolutely. I expect you to do research online as you work through the problems. I do not include all the syntax in this book. In my day-to-day work as a data engineer, I would be lost without being able to do online research. Sometimes I search online for a reminder of a certain syntax, sometimes for examples of a particular type of code, and sometimes for approaches to specific problems. Learning to find answers online effectively can cut your problem-solving time dramatically.

The answer to each problem, and frequently a discussion on the answer, are available in the back of the book. The same recommendation applies to the answer as well as the hints – don't use them unless you need to! But once you have an answer that solves the problem, do compare it to the answer that I've provided, just to get some perspective on alternative answers.

Thank you for purchasing this book!

For any questions or issues, please send email to
feedback@SQLPracticeProblems.com.
I will be happy to respond.

Setup

This section will help you with the install of MySQL 8.0, and will walk you through the setup of the practice database.

MySQL 8.0 is a big leap in functionality from previous versions. It includes many very useful features that other DBMS systems (such as Microsoft SQL Server) have had for a long time, such as Window functions and CTEs (Common Table Expressions). The problems in this book are dependent on having MySQL 8.0 instead of previous versions.

Installing MySQL 8.0

To download and install MySQL 8.0, visit this website:

https://dev.mysql.com/downloads/

It's a straightforward setup. Make sure you install version 8.0 (MySQL 8.0 Generally Available (GA) Release) and not an earlier version. You can choose the operating system you wish to use, such as Microsoft Windows, macOS, Linux, etc.

Make sure you also install MySQL Workbench or a similar tool, in order to actually run SQL against the database. MySQL Workbench is included in the default installation, and that's the tool I use to walk through the setup of the practice database.

If you have any issues, the install documentation is online here:

https://dev.mysql.com/doc/refman/8.0/en/installing.html

Setting up the practice database

This video will walk you through the setup of the practice database used for the problems.

https://www.youtube.com/watch?v=7QN600zGUIo

It assumes that you have installed MySQL version 8.0.

Questions or comments on the setup? Email me at feedback@SQLPracticeProblems.com

Intermediate Problems

1. Cost changes for each product

There's a table called ProductCostHistory which contains the history of the cost of the product. Using that table, get the total number of times the product cost has changed.

Sort the results by ProductID

Expected Results

ProductID	TotalPriceChanges
707	3
708	3
709	1
710	1
711	3
712	3
713	3
714	3
715	3
716	4
717	3
718	3
719	3
720	3
721	3

(not all rows shown)

2. Customers with total orders placed

We want to see a list of all the customers that have made orders, and the total number of orders the customer has made.

Sort by the total number of orders, in descending order

Expected Results

CustomerID	TotalOrders
11711	5
12166	4
29586	4
29745	4
29837	4
29812	4
29951	3
30115	3
29675	3
29676	3
11276	3
11247	3
11091	3
11997	3
12216	2

(not all rows shown)

3. Products with first and last order date

For each product that was ordered, show the first and last date that it was ordered.

In the previous problem I gave you the table name to use. For this problem, look at the list of tables, and figure out which ones you need to use. You can use the Navigator Schema in the Object Browser in MySQL Workbench, or you can also run this SQL:

```
show tables
```

Sort the results by ProductID.

Expected Results

ProductID	FirstOrder	LastOrder
707	2011-05-31	2014-06-28
708	2011-05-31	2014-06-25
709	2011-05-31	2012-03-30
710	2011-10-01	2011-10-01
711	2011-05-31	2014-06-30
712	2011-05-31	2014-06-22
713	2013-07-17	2014-05-06
714	2011-05-31	2014-06-29
715	2011-05-31	2014-06-07
716	2011-05-31	2014-06-30
717	2011-10-01	2014-03-31
718	2012-03-30	2014-03-31
719	2012-05-30	2013-06-30
722	2011-05-31	2014-05-01
723	2012-05-30	2013-06-30

(not all rows shown)

4. Products with first and last order date, including name

For each product that was ordered, show the first and last date that it was ordered. This time, include the name of the product in the output, to make it easier to understand.

Sort the results by ProductID.

Expected Results

ProductID	ProductName	FirstOrder	LastOrder
707	Sport-100 Helmet, Red	2011-05-31	2014-06-28
708	Sport-100 Helmet, Black	2011-05-31	2014-06-25
709	Mountain Bike Socks, M	2011-05-31	2012-03-30
710	Mountain Bike Socks, L	2011-10-01	2011-10-01
711	Sport-100 Helmet, Blue	2011-05-31	2014-06-30
712	AWC Logo Cap	2011-05-31	2014-06-22

713	Long-Sleeve Logo Jersey, S	2013-07-17	2014-05-06
714	Long-Sleeve Logo Jersey, M	2011-05-31	2014-06-29
715	Long-Sleeve Logo Jersey, L	2011-05-31	2014-06-07
716	Long-Sleeve Logo Jersey, XL	2011-05-31	2014-06-30
717	HL Road Frame - Red, 62	2011-10-01	2014-03-31
718	HL Road Frame - Red, 44	2012-03-30	2014-03-31
719	HL Road Frame - Red, 48	2012-05-30	2013-06-30
722	LL Road Frame - Black, 58	2011-05-31	2014-05-01
723	LL Road Frame - Black, 60	2012-05-30	2013-06-30

(not all rows shown)

5. Product cost on a specific date

We'd like to get a list of the cost of products, as of a certain date, 2012-04-15. Use the ProductCostHistory to get the results.

Sort the output by ProductID.

Expected Results

ProductID	StandardCost
707	12.0278
708	12.0278
709	3.3963
710	3.3963
711	12.0278
712	5.7052
713	31.7244
714	31.7244
715	31.7244
716	31.7244
717	747.9682
718	747.9682
719	747.9682
720	747.9682
721	747.9682

(not all rows shown)

6. Product cost on a specific date, part 2

It turns out that the answer to the above problem has a problem. Change the date to 2014-04-15. What are your results?

If you use the SQL from the answer above, and just change the date, you won't get the results you want.

Fix the SQL so it gives the correct results with the new date. Note that when the EndDate is null, that means that price is applicable into the future.

Expected Results

ProductID	StandardCost
707	13.0863
708	13.0863
711	13.0863
712	6.9223
713	38.4923
714	38.4923
715	38.4923
716	38.4923
717	868.6342
718	868.6342
719	868.6342
720	868.6342
721	868.6342
722	204.6251
723	204.6251

(not all rows shown)

7. Product List Price: how many price changes?

Show the months from the ProductListPriceHistory table, and the total number of changes made in that month.

Expected Results

ProductListPriceMonth	TotalRows
2011/05	72
2012/05	128
2013/05	203

8. Product List Price: months with no price changes?

After reviewing the results of the previous query, it looks like price changes are made only in one month of the year.

We want a query that makes this pattern very clear. Show all months (within the range of StartDate values in ProductListPriceHistory). This includes the months during which no prices were changed.

Expected Results

CalendarMonth	TotalRows
2011/05 - May	72
2011/06 - Jun	0
2011/07 - Jul	0
2011/08 - Aug	0
2011/09 - Sep	0
2011/10 - Oct	0
2011/11 - Nov	0
2011/12 - Dec	0
2012/01 - Jan	0
2012/02 - Feb	0
2012/03 - Mar	0
2012/04 - Apr	0
2012/05 - May	128
2012/06 - Jun	0
2012/07 - Jul	0
2012/08 - Aug	0
2012/09 - Sep	0
2012/10 - Oct	0
2012/11 - Nov	0
2012/12 - Dec	0
2013/01 - Jan	0
2013/02 - Feb	0

2013/03 - Mar	0
2013/04 - Apr	0
2013/05 - May	203

9. Current list price of every product

What is the current list price of every product, using the ProductListPrice history?

Order the results by ProductID

Expected Results

ProductID	ListPrice
707	34.9900
708	34.9900
711	34.9900
712	8.9900
713	49.9900
713	49.9900
714	49.9900
715	49.9900
716	49.9900
717	1431.5000
718	1431.5000
719	1431.5000
720	1431.5000
721	1431.5000
722	337.2200

(not all rows shown)

10. Products without a list price history

Show a list of all products that do not have any entries in the list price history table.

Sort the results by ProductID

Expected Results

ProductID	ProductName
1	Adjustable Race
2	Bearing Ball
3	BB Ball Bearing
4	Headset Ball Bearings
316	Blade
317	LL Crankarm
318	ML Crankarm
319	HL Crankarm
320	Chainring Bolts
321	Chainring Nut
322	Chainring
323	Crown Race
324	Chain Stays
325	Decal 1
326	Decal 2

(not all rows shown)

11. Product cost on a specific date, part 3

In the earlier problem "Product cost on a specific date, part 2", this answer was given:

```
Select
    ProductID
    ,StandardCost
From ProductCostHistory
Where
    '2014-04-15' Between StartDate and IfNull(EndDate, Now())
Order By ProductID;
```

However, there are many ProductIDs that exist in the ProductCostHistory table that don't show up in this list.

Show every ProductID in the ProductCostHistory table that does not appear when you run the above SQL.

Expected Results

ProductID
709
710
725
726
727
728
729
730
731
732
733
734
735
740
741

(not all rows shown)

12. Products with multiple current list price records

There should only be one current price for each product in the ProductListPriceHistory table, but unfortunately some products have multiple current records.

Find all these, and sort by ProductID

Expected Results

ProductID
713
837
868
899
930

961
992

13. Products with their first and last order date, including name and subcategory

In the problem “Products with their first and last order date, including name", we looked only at product that have been ordered.

It turns out that there are many products that have never been ordered.

This time, show all the products, and the first and last order date. Include the product subcategory as well.

Sort by the ProductName field.

Expected Results

ProductID	ProductName	ProductSubCategoryName	FirstOrder	LastOrder
1	Adjustable Race	NULL	NULL	NULL
879	All-Purpose Bike Stand	Bike Stands	2013-07-23	2014-06-19
712	AWC Logo Cap	Caps	2011-05-31	2014-06-22
3	BB Ball Bearing	NULL	NULL	NULL
2	Bearing Ball	NULL	NULL	NULL
877	Bike Wash - Dissolver	Cleaners	2013-06-13	2014-06-19
316	Blade	NULL	NULL	NULL
843	Cable Lock	Locks	2012-05-30	2013-09-30
952	Chain	Chains	2013-06-30	2014-05-01
324	Chain Stays	NULL	NULL	NULL
322	Chainring	NULL	NULL	NULL
320	Chainring Bolts	NULL	NULL	NULL
321	Chainring Nut	NULL	NULL	NULL
866	Classic Vest, L	Vests	2013-10-26	2014-04-25
865	Classic Vest, M	Vests	2013-06-30	2014-06-26

(not all rows shown)

14. Products with list price discrepancies

It's astonishing how much work with SQL and data is in finding and resolving discrepancies in data. Some of the salespeople have told us that the current price in the price list history doesn't seem to match the actual list price in the Product table.

Find all these discrepancies. Sort the results by ProductID.

Expected Results

ProductID	ProductName	Prod_ListPrice	PriceHist_LatestListPrice	Diff
792	Road-250 Red, 58	2443.3500	2442.3500	1.0000
858	Half-Finger Gloves, S	24.4900	23.4900	1.0000
891	HL Touring Frame - Blue, 50	1003.9100	1004.9100	-1.0000
924	LL Mountain Frame - Black, 42	249.7900	248.7900	1.0000
957	Touring-1000 Yellow, 60	2384.0700	2385.0700	-1.0000
990	Mountain-500 Black, 42	539.9900	538.9900	1.0000

15. Orders for products that were unavailable

It looks like some products were sold before or after they were supposed to be sold, based on the SellStartDate and SellEndDate in the Product table. Show a list of these orders, with details.

Sort the results by ProductID, then OrderDate.

Expected Results

ProductID	OrderDate	ProductName	Qty	SellStartDate	SellEndDate
726	2013-08-30	LL Road Frame - Red, 48	1	2011-05-31	2013-05-29
726	2014-02-28	LL Road Frame - Red, 48	1	2011-05-31	2013-05-29
729	2013-06-30	LL Road Frame - Red, 60	2	2011-05-31	2013-05-29
730	2011-05-31	LL Road Frame - Red, 62	2	2011-06-01	2013-05-29
730	2011-05-31	LL Road Frame - Red, 62	6	2011-06-01	2013-05-29
730	2011-05-31	LL Road Frame - Red, 62	2	2011-06-01	2013-05-29
730	2011-05-31	LL Road Frame - Red, 62	2	2011-06-01	2013-05-29
730	2011-05-31	LL Road Frame - Red, 62	2	2011-06-01	2013-05-29

730	2013-06-30	LL Road Frame - Red, 62	1	2011-06-01	2013-05-29
730	2014-03-30	LL Road Frame - Red, 62	1	2011-06-01	2013-05-29
760	2013-06-30	Road-650 Red, 60	3	2011-05-31	2013-05-29
760	2013-12-31	Road-650 Red, 60	2	2011-05-31	2013-05-29
760	2014-02-28	Road-650 Red, 60	1	2011-05-31	2013-05-29
760	2014-03-30	Road-650 Red, 60	5	2011-05-31	2013-05-29
761	2013-05-30	Road-650 Red, 62	2	2011-05-31	2013-05-29

(not all rows shown)

16. Orders for products that were unavailable: details

We'd like to get more details on when products (that were supposed to be unavailable) were ordered.

Create a new column that shows whether the product was ordered before the sell start date, or after the sell end date.

Sort the results by ProductID and OrderDate.

Expected Results

ProductID	OrderDate	Qty	SellStartDate	SellEndDate	ProblemType
726	2013-08-30	1	2011-05-31	2013-05-29	Sold after end date
726	2014-02-28	1	2011-05-31	2013-05-29	Sold after end date
729	2013-06-30	2	2011-05-31	2013-05-29	Sold after end date
730	2011-05-31	2	2011-06-01	2013-05-29	Sold before start date
730	2011-05-31	6	2011-06-01	2013-05-29	Sold before start date
730	2011-05-31	2	2011-06-01	2013-05-29	Sold before start date
730	2011-05-31	2	2011-06-01	2013-05-29	Sold before start date
730	2011-05-31	2	2011-06-01	2013-05-29	Sold before start date
730	2013-06-30	1	2011-06-01	2013-05-29	Sold after end date
730	2014-03-30	1	2011-06-01	2013-05-29	Sold after end date
760	2013-06-30	3	2011-05-31	2013-05-29	Sold after end date
760	2013-12-31	2	2011-05-31	2013-05-29	Sold after end date
760	2014-02-28	1	2011-05-31	2013-05-29	Sold after end date
760	2014-03-30	5	2011-05-31	2013-05-29	Sold after end date
761	2013-05-30	2	2011-05-31	2013-05-29	Sold after end date

(not all rows shown)

17. OrderDate with time component

How many OrderDate values in SalesOrderHeader have a time component to them?

Show the results as below.

Expected Results

TotalOrderWithTime	TotalOrders	PercentOrdersWithTime
140	1561	0.0897

18. Fix this SQL! Number 1

We want to show details about certain products (name, subcategory, first order date, last order date), similar to what we did in a previous query.

This time, we only want to show the data for products that have Silver in the color field. Because you've looked at the Color field of the Product table directly, you know that there are many products with that color.

A colleague sent you this query, and asked you to look at it. It seems correct, but it returns no rows.

What's wrong with it?

```
Select
    Product.ProductID
    ,ProductName
    ,ProductSubCategoryName
    ,Date(Min(OrderDate)) as FirstOrder
    ,Date(Max(OrderDate)) as LastOrder
From Product
    Left Join SalesOrderDetail  Detail
        on Product.ProductID = Detail.ProductID
    Left Join SalesOrderHeader  Header
        on Header.SalesOrderID = Detail .SalesOrderID
    Left Join ProductSubCategory
```

```
        on ProductSubCategory .ProductSubCategoryID = Product.ProductSubCategoryID
Where
    'Color' = 'Silver'
Group by
    Product.ProductID
    ,ProductName
    ,ProductSubCategoryName
Order by LastOrder desc;
```

19. Raw margin quartile for products

The product manager would like to show information for all products about the raw margin – that is, the price minus the cost. Create a query that will show this information, as well as the raw margin quartile.

For this problem, the quartile should be 1 if the raw margin of the product is in the top 25%, 2 if the product is in the second 25%, etc.

Sort the rows by the product name.

Expected Results

ProductID	ProductName	StandardCost	ListPrice	RawMargin	Quartile
879	All-Purpose Bike Stand	59.466	159.00	99.534	3
712	AWC Logo Cap	6.9223	8.99	2.0677	4
877	Bike Wash - Dissolver	2.9733	7.95	4.9767	4
843	Cable Lock	10.3125	25.00	14.6875	4
952	Chain	8.9866	20.24	11.2534	4
866	Classic Vest, L	23.749	63.50	39.751	3
865	Classic Vest, M	23.749	63.50	39.751	3
864	Classic Vest, S	23.749	63.50	39.751	3
878	Fender Set - Mountain	8.2205	21.98	13.7595	4
948	Front Brakes	47.286	106.50	59.214	3
945	Front Derailleur	40.6216	91.49	50.8684	3
863	Full-Finger Gloves, L	15.6709	37.99	22.3191	4
862	Full-Finger Gloves, M	15.6709	37.99	22.3191	4
861	Full-Finger Gloves, S	15.6709	37.99	22.3191	4
860	Half-Finger Gloves, L	9.1593	24.49	15.3307	4

```
(not all rows shown)
```

20. Customers with purchases from multiple sales people

Show all the customers that have made purchases from multiple sales people.

Sort the results by the customer name (first name plus last name).

Expected Results

CustomerID	CustomerName	TotalDifferentSalesPeople
29675	Aaron Con	3
29637	Donna Carreras	2
29523	John Arthur	2
29486	Kim Abercrombie	2
29617	Lindsey Camacho	2
29508	Marvin Allen	2
29837	Onetha Higgs	2
29614	Ryan Calafato	2

21. Fix this SQL! Number 2

A colleague has sent you the following SQL, which causes an error:

```
Select
    Customer.CustomerID
    ,FirstName + ' ' + LastName as CustomerName
    ,OrderDate
    ,SalesOrderHeader.SalesOrderID
```

```
    ,SalesOrderDetail.ProductID
    ,Product.ProductName
    ,LineTotal
From  SalesOrderHeader
    Join Product
        on Product.ProductID = SalesOrderDetail .ProductID
    Join SalesOrderDetail
        on SalesOrderHeader .SalesOrderID = SalesOrderDetail .SalesOrderID
    Join Customer
        on Customer.CustomerID = SalesOrderHeader.CustomerID
Order by
    CustomerID
    ,OrderDate
Limit 100;
```

The error it gives is this:

```
Error Code: 1054. Unknown column 'SalesOrderDetail.ProductID' in 'on clause'
```

Fix the SQL so it returns the correct results without error.

22. Duplicate product

It looks like the Product table may have duplicate records. Find the names of the products that have duplicate records (based on having the same ProductName).

Expected Results

ProductName
Mountain Pump
Touring Tire

23. Duplicate product: details

We'd like to get some details on the duplicate product issue. For each product that has duplicates, show the product name and the specific ProductID that we believe to be the duplicate (the one that's not the first ProductID for the product name).

Expected Results

PotentialDuplicateProductID	ProductName
1000	Mountain Pump
1001	Touring Tire

Congratulations! You've completed the intermediate problems.

Any questions or feedback on the problems, hints, or answers? I'd like to hear from you. Please email me at feedback@SQLPracticeProblems.com.

Advanced Problems

24. How many cost changes do products generally have?

We've worked on many problems based on the ProductCostHistory table. We know that the cost for some products has changed more than for other products.

Write a query that shows how many cost changes that products have, in general.

For this query, you can ignore the fact that in ProductCostHistory, sometimes there's an additional record for a product where the cost didn't actually change.

Expected Results

TotalPriceChanges	TotalProducts
1	216
2	52
3	22
4	3

25. Size and base ProductNumber for products

The ProductNumber field in the Product table comes from the vendor of the product. The size is sometimes a part of this field.

We need to get the base ProductNumber (without the size), and then the size separately. Some products do not have a size. For those products, the base ProductNumber will be the same as the ProductNumber, and the size field will be null.

Limit the results to those ProductIDs that are greater than 533. Sort by ProductID.

Expected Results

ProductID	ProductNumber	HyphenLocation	BaseProductNumber	Size
534	TO_2301	0	TO_2301	NULL
535	TP_0923	0	TP_0923	NULL
679	RC_0291	0	RC_0291	NULL
680	FR_R92B-58	8	FR_R92B	58
706	FR_R92R-58	8	FR_R92R	58
707	HL_U509R	0	HL_U509R	NULL
708	HL_U509	0	HL_U509	NULL
709	SO_B909-M	8	SO_B909	M
710	SO_B909-L	8	SO_B909	L
711	HL_U509B	0	HL_U509B	NULL
712	CA_1098	0	CA_1098	NULL
713	LJ_0192-S	8	LJ_0192	S
714	LJ_0192-M	8	LJ_0192	M
715	LJ_0192-L	8	LJ_0192	L
716	LJ_0192-X	8	LJ_0192	X

(not all rows shown)

26. Number of sizes for each base product number

Now we'd like to get all the base ProductNumbers, and the number of sizes that they have.

Use the output of the previous problem to get the results. However, do not use the filter from the previous problem (ProductIDs that are greater than 533). Instead of that filter, select only those products that are clothing (ProductCategory = 3).

Order by the base ProductNumber.

Expected Results

BaseProductNumber	TotalSizes
CA_1098	1
GL_F110	3
GL_H102	3

LJ_0192	4
SB_M891	3
SH_M897	4
SH_W890	3
SJ_0194	4
SO_B909	2
SO_R809	2
TG_W091	3
VE_C304	3

27. How many cost changes has each product really had?

A sharp-eyed analyst has pointed out that the total number of product cost changes (from the problem "Cost changes for each product" is not right. Why? Because sometimes, even when there's a new record in the ProductCostHistory table, the cost is not actually different from the previous record!

This eventually will require a fix to the database, to make sure that we do not allow a record like this to be entered. This could be done as a table constraint, or a change to the code used to insert the row.

However, for now, let's just get an accurate count of cost changes per product, where the cost has actually changed. Also include the initial row for a product, even if there's only 1 record.

Sort the output by ProductID.

Expected Results

ProductID	ProductName	TotalCostChanges
707	Sport-100 Helmet, Red	3
708	Sport-100 Helmet, Black	3
709	Mountain Bike Socks, M	1
710	Mountain Bike Socks, L	1
711	Sport-100 Helmet, Blue	3
712	AWC Logo Cap	3
713	Long-Sleeve Logo Jersey, S	3
714	Long-Sleeve Logo Jersey, M	3
715	Long-Sleeve Logo Jersey, L	3
716	Long-Sleeve Logo Jersey, XL	3

717	HL Road Frame - Red, 62	3
718	HL Road Frame - Red, 44	3
719	HL Road Frame - Red, 48	3
720	HL Road Frame - Red, 52	3
721	HL Road Frame - Red, 56	3

(not all rows shown)

28. Which products had the largest increase in cost?

We'd like to show which products have had the largest, one-time increases in cost. Show all of the price increases (and decreases), in decreasing order of difference.

Don't show any records for which there is no price difference. For instance, if a product only has 1 record in the cost history table, you would not show it in the output, because there has been no change in the cost history.

Order by the price difference, and then the ProductID.

Expected Results

ProductID	CostChangeDate	StandardCost	PreviousStandardCost	PriceDifference
717	2012-05-30	722.2568	747.9682	25.7114
718	2012-05-30	722.2568	747.9682	25.7114
719	2012-05-30	722.2568	747.9682	25.7114
720	2012-05-30	722.2568	747.9682	25.7114
721	2012-05-30	722.2568	747.9682	25.7114
722	2012-05-30	170.1428	176.1997	6.0569
723	2012-05-30	170.1428	176.1997	6.0569
724	2012-05-30	170.1428	176.1997	6.0569
736	2012-05-30	170.1428	176.1997	6.0569
737	2012-05-30	170.1428	176.1997	6.0569
738	2012-05-30	170.1428	176.1997	6.0569
713	2012-05-30	29.0807	31.7244	2.6437
714	2012-05-30	29.0807	31.7244	2.6437
715	2012-05-30	29.0807	31.7244	2.6437
716	2012-05-30	29.0807	31.7244	2.6437

(not all rows shown)

29. Fix this SQL! Number 3

There's been some problems with fraudulent transactions. The data science team has requested, for a machine learning job, a unusual set of records. It should include data for 11 CustomerIDs that are specifically identified as fraudulent. It should also include a set of 100 random customers. The set of 100 random customers must exclude the 11 customers suspected of fraud.

The SQL below solves the problem. However, there's a problem with it, which is that the list of bad customers is repeated twice.

Having hard-coded numbers or lists of numbers in SQL is not a good idea in general. But duplicating them is even worse, because of the potential that they will get out of sync.

Rewrite this SQL to not repeat the hard-coded list of CustomerIDs that are fraud suspects.

```
with FraudSuspects as (
    Select *
    From Customer
    Where
        CustomerID in (
            29401
            ,11194
            ,16490
            ,22698
            ,26583
            ,12166
            ,16036
            ,25110
            ,18172
            ,11997
            ,26731
        )
)
, SampleCustomers as (
    Select *
    From Customer
    Where
        CustomerID not in (
            29401
            ,11194
            ,16490
            ,22698
            ,26583
            ,12166
            ,16036
            ,25110
            ,18172
            ,11997
            ,26731
```

```
        )
    Order by
        Rand()
    Limit 100
)
Select * From FraudSuspects
Union all
Select * From SampleCustomers;
```

30. History table with start/end date overlap

There is a product that has an overlapping date ranges in the ProductListPriceHistory table.

Find the products with overlapping records, and show the dates that overlap.

Expected Results

CalendarDate	ProductID	TotalRows
2013-05-15	746	2
2013-05-16	746	2
2013-05-17	746	2
2013-05-18	746	2
2013-05-19	746	2
2013-05-20	746	2
2013-05-21	746	2
2013-05-22	746	2
2013-05-23	746	2
2013-05-24	746	2
2013-05-25	746	2
2013-05-26	746	2
2013-05-27	746	2
2013-05-28	746	2
2013-05-29	746	2

31. History table with start/end date overlap, part 2

It turns out that the SQL that was provided in the Answer section for the previous problem has an error. It's missing a ProductID that also has a date range overlap.

If you wrote SQL that actually showed 2 separate ProductIDs—great job!

If you didn't, then fix the SQL for the previous problem to show all date range overlaps

Sort the results by ProductID and CalendarDate.

Expected Results

CalendarDate	ProductID	TotalRows
2013-05-27	737	2
2013-05-28	737	2
2013-05-29	737	2
2013-05-15	746	2
2013-05-16	746	2
2013-05-17	746	2
2013-05-18	746	2
2013-05-19	746	2
2013-05-20	746	2
2013-05-21	746	2
2013-05-22	746	2
2013-05-23	746	2
2013-05-24	746	2
2013-05-25	746	2
2013-05-26	746	2
2013-05-27	746	2
2013-05-28	746	2
2013-05-29	746	2

32. Running total of orders in last year

For the company dashboard we'd like to calculate the total number of orders, by month, as well as the running total of orders.

Limit the rows to the last year of orders. Sort by calendar month.

Expected Results

CalendarMonth	TotalOrders	RunningTotal
2013/06 - Jun	13	13
2013/07 - Jul	86	99
2013/08 - Aug	88	187
2013/09 - Sep	89	276
2013/10 - Oct	99	375
2013/11 - Nov	105	480
2013/12 - Dec	102	582
2014/01 - Jan	105	687
2014/02 - Feb	88	775
2014/03 - Mar	120	895
2014/04 - Apr	105	1000
2014/05 - May	117	1117
2014/06 - Jun	47	1164

33. Total late orders by territory

Show the number of total orders, and the number of orders that are late.

For this problem, an order is late when the DueDate is before the ShipDate.

Group and sort the rows by Territory.

Expected Results

TerritoryID	TerritoryName	CountryCode	TotalOrders	TotalLateOrders
1	Northwest	US	233	78
2	Northeast	US	14	7
3	Central	US	20	11
4	Southwest	US	296	93
5	Southeast	US	23	6
6	Canada	CA	187	64
7	France	FR	148	57
8	Germany	DE	137	22
9	Australia	AU	324	100
10	United Kingdom	GB	179	62

34. OrderDate with time component—performance aspects

We don't go often get into performance issues in these practice problems. But there's many different ways of getting the answer to the problem “OrderDate with time component”. Looking at the different answers gives us a good opportunity to look at the performance implications of different strategies.

Below are 4 SQL statements that all solve the problem of how many OrderDate values in the SalesOrderHeader table have a time component.

One of the below statements is significantly worse than the others. Figure out which of the below statements is the worst in terms of performance.

Performance optimization has many different components, and entire books can (and have) been written about the topic. But for this problem, use the MySQL session status variable called Innodb_buffer_pool_read_requests. The number returned by this variable is a good metric for the amount of resources used.

The answer to this problem is the number of the solution which consumes the least resources, based on the variable Innodb_buffer_pool_read_requests.

```
-- ---------------------------------------
-- Solution #1
-- ---------------------------------------
Select
    OrdersWithTime.TotalOrderWithTime
    ,TotalOrders.TotalOrders
    ,OrdersWithTime.TotalOrderWithTime  * 1.0
    /
    TotalOrders.TotalOrders
        as PercentOrdersWithTime
From
    (Select Count(*) TotalOrders
    from SalesOrderHeader ) TotalOrders
    left join
    (Select Count(*) as TotalOrderWithTime
    from SalesOrderHeader
    where OrderDate <> Date(OrderDate)) OrdersWithTime
        on 1 = 1
;

-- ---------------------------------------
-- Solution #2
-- ---------------------------------------
with OrdersWithTime as (
    Select Count(*) as TotalOrderWithTime
    from SalesOrderHeader
    where OrderDate <> Date(OrderDate)
```

```
)
, TotalOrders as (
    Select Count(*) as TotalOrders from SalesOrderHeader
)
Select
    OrdersWithTime.TotalOrderWithTime
    ,TotalOrders.TotalOrders
    ,OrdersWithTime.TotalOrderWithTime
    /
    TotalOrders.TotalOrders
        as PercentOrdersWithTime
from TotalOrders
    left join OrdersWithTime
        on 1=1
;

-- ---------------------------------------
-- Solution #3
-- ---------------------------------------
Select
    (Select Count(*)
    from SalesOrderHeader
    where OrderDate <> Date(OrderDate)
    )
        as TotalOrderWithTime
    ,(Select Count(*) from SalesOrderHeader )
        as TotalOrders
    ,(Select Count(*)
    from SalesOrderHeader
    where OrderDate <> Date(OrderDate)
    )
    /
    (Select Count(*) from SalesOrderHeader )
        as PercentOrdersWithTime
;

-- ---------------------------------------
-- Solution #4
-- ---------------------------------------
with Main as (
    Select
        SalesOrderID
        ,Case
            When OrderDate <> Date(OrderDate) then 1
            Else 0
        End
            as HasTimeComponent
    From SalesOrderHeader
)
Select
    Sum(HasTimeComponent ) as TotalOrdersWithTime
```

```
    ,Count(*) as TotalOrders
    ,Sum(HasTimeComponent )
    /
    Count(*)
        as PercentOrdersWithTime
From Main
;
```

35. Customer's last purchase—what was the product subcategory?

For a limited list of customers, we need to show the product subcategory of their last purchase. If they made more than one purchase on a particular day, then show the one that cost the most.

Limit the customers to these customer IDs:

19500
19792
24409
26785

Expected Results

CustomerID	CustomerName	ProductSubCategoryName
19500	Russell Shan	Road Bikes
19792	Cristina Nara	Road Bikes
24409	Jackson Hernandez	Helmets
26785	Raul Raje	Tires and Tubes

36. Order processing: time in each stage

When an order is placed, it goes through different stages, such as processed, readied for pick up, in transit, delivered, etc.

How much time does each order spend in the different stages?

To figure out which tables to use, take a look at the list of tables in the database. You should be able to figure out the tables to use from the table names.

Limit the orders to these SalesOrderIDs:

68857
70531
70421

Sort by the SalesOrderID, and then the date/time.

Expected Results

SalesOrderID	EventName	TrackingEventDate	NextTrackingEventDate	HoursInStage
68857	Order processing	2014-03-23 02:00	2014-03-23 05:00	3
68857	Order processed	2014-03-23 05:00	2014-03-24 00:00	19
68857	In transit	2014-03-24 00:00	2014-03-24 03:00	3
68857	Arrived at facility	2014-03-24 03:00	2014-03-24 05:00	2
68857	Departed from facility	2014-03-24 05:00	2014-03-24 09:00	4
68857	Order delivered	2014-03-24 09:00	NULL	NULL
70421	Order processing	2014-04-11 02:00	2014-04-11 05:00	3
70421	Order processed	2014-04-11 05:00	2014-04-12 00:00	19
70421	In transit	2014-04-12 00:00	2014-04-12 03:00	3
70421	Arrived at facility	2014-04-12 03:00	2014-04-12 05:00	2
70421	Departed from facility	2014-04-12 05:00	2014-04-12 09:00	4
70421	Order delivered	2014-04-12 09:00	NULL	NULL
70531	Order processing	2014-04-13 02:00	2014-04-13 05:00	3
70531	Order processed	2014-04-13 05:00	2014-04-14 00:00	19
70531	In transit	2014-04-14 00:00	2014-04-14 03:00	3
70531	Arrived at facility	2014-04-14 03:00	2014-04-14 05:00	2
70531	Departed from facility	2014-04-14 05:00	2014-04-14 09:00	4
70531	Order delivered	2014-04-14 09:00	NULL	NULL

37. Order processing: time in each stage, part 2

Now we want to show the time spent in each stage of order processing, but instead of showing information for specific orders, we want to show aggregate data, by online vs offline orders.

Sort first by OnlineOfflineStatus, and then TrackingEventID.

Expected Results

OnlineOfflineStatus	EventName	AverageHoursSpentInStage
Offline	Order processing	2.0000
Offline	Order processed	20.0000
Offline	In transit	3.0000
Offline	Arrived at facility	2.0000
Offline	Departed from facility	4.0000
Offline	Order delivered	
Offline	Delivery error	2.5000
Online	Order processing	2.9993
Online	Order processed	19.0000
Online	In transit	3.0000
Online	Arrived at facility	2.0000
Online	Departed from facility	4.0000
Online	Order delivered	
Online	Delivery error	8.0000

38. Order processing: time in each stage, part 3

The previous query was very helpful to the operations manager, to help her get an overview of differences in order processing between online and offline orders.

Now she has another request, which is to have the averages for online/offline status on the same line, to make it easier to compare.

Expected Results

EventName	OfflineAvgHoursInStage	OnlineAvgHoursInStage
Order processing	2.0000	2.9993
Order processed	20.0000	19.0000

In transit	3.0000	3.0000
Arrived at facility	2.0000	2.0000
Departed from facility	4.0000	4.0000
Order delivered		
Delivery error	2.5000	8.0000

39. Top three product subcategories per customer

The marketing department would like to have a listing of customers, with the top 3 product subcategories that they've purchased.

To define “top 3 product subcategories”, we'll order by the total amount purchased for those subcategories (i.e. the line total).

Normally we'd run the query for all customers, but to make it easier to view the results, please limit to just the following customers:

13763
13836
20331
21113
26313

Sort the results by CustomerID

Expected Results

CustomerID	CustomerName	TopProdSubCat1	TopProdSubCat2	TopProdSubCat3
13763	Stephanie Richardson	Fenders	Socks	NULL
13836	Dominic Sara	Road Bikes	Mountain Bikes	Tires and Tubes
20331	Jeremy Cox	Shorts	Vests	NULL
21113	Laura Cai	Tires and Tubes	NULL	NULL
26313	Dalton Simmons	Touring Bikes	Mountain Bikes	Jerseys

40. History table with date gaps

It turns out that, in addition to overlaps, there are also some gaps in the ProductListPriceHistory table. That is, there are some date ranges for which there are no list prices. We need to find the products and the dates for which there are no list prices.

This is one of the most challenging and fun problems in this book, so take your time and enjoy it! Try doing it first without any hints, because even if you don't manage to solve the problem, you will have learned much more.

Expected Results

ProductID	DateWithMissingPrice
742	2013-05-21
742	2013-05-22
742	2013-05-23
742	2013-05-24
742	2013-05-25
742	2013-05-26
742	2013-05-27
742	2013-05-28
742	2013-05-29
747	2013-05-27
747	2013-05-28
747	2013-05-29

Congratulations! You've completed the advanced problems.

Any questions or feedback on the problems, hints, or answers? I'd like to hear from you. Please email me at feedback@SQLPracticeProblems.com.

Answers: Intermediate Problems

1. Cost changes for each product

```
Select
    ProductID
    , Count(*) as TotalPriceChanges
From ProductCostHistory
Group by ProductID
Order by ProductID;
```

2. Customers with total orders placed

```
Select
    CustomerID
    , Count(*) as TotalOrders
From SalesOrderHeader
Group by
    CustomerID
```

```
Order by Count(*) desc;
```

Discussion

The following SQL also works. Note the change to the Order by clause:

```
Select
    CustomerID
    ,Count(*) as TotalOrders
From SalesOrderHeader
Group by
    CustomerID
Order by TotalOrders desc;
```

In a Group By clause, a Where Clause, or a Having clause, you could not reuse a named column like this. However, in the Order By clause, it works.

3. Products with first and last order date

```
Select
    ProductID
    ,Date(Min(OrderDate)) as FirstOrder
    ,Date(Max(OrderDate)) as LastOrder
From SalesOrderDetail
    Join SalesOrderHeader
        on SalesOrderHeader .SalesOrderID = SalesOrderDetail .SalesOrderID
Group by
    ProductID
Order by ProductID;
```

4. Products with first and last order date, including name

```
Select
    Product.ProductID
    ,ProductName
    ,Date(Min(OrderDate)) as FirstOrder
    ,Date(Max(OrderDate)) as LastOrder
From SalesOrderHeader  Header
    Join SalesOrderDetail  Detail
        on Header .SalesOrderID = Detail .SalesOrderID
    Join Product
        on Product.ProductID = Detail.ProductID
Group by
    Product.ProductID
    ,ProductName
Order by Product.ProductID;
```

5. Product cost on a specific date

```
Select
    ProductID
    ,StandardCost
From ProductCostHistory
Where
    '2012-04-15' Between StartDate and EndDate
Order By ProductID;
```

Discussion

Another potential answer is this:

```
Select
    ProductID
    ,StandardCost
From ProductCostHistory
Where
    StartDate <= '2012-04-15'
    and EndDate  >= '2012-04-15'
Order By ProductID;
```

I find the first answer a little easier to read. Also, the first answer does not duplicate the hard-coded date of 2012-04-15, which is preferable.

6. Product cost on a specific date, part 2

```
Select
    ProductID
    ,StandardCost
From ProductCostHistory
Where
    '2014-04-15' Between StartDate and IfNull(EndDate, Now())
Order By ProductID;
```

Dealing with nulls is a big part of writing correct SQL. It's always best to check the structure of the table to see if a field can contain nulls.

A quick way to check the structure of the table (and which fields are nullable) in MySQL is by using the statement Describe. For instance:

```
Describe ProductCostHistory;
```

The Explain statement works identically to Describe, so you could also run the following:

```
Explain ProductCostHistory;
```

7. Product List Price: how many price changes?

```
Select
```

```
    DATE_FORMAT(StartDate, '%Y/%m') as ProductListPriceMonth
    ,Count(*) as TotalRows
From ProductListPriceHistory
Group by DATE_FORMAT(StartDate, '%Y/%m')
Order by ProductListPriceMonth;
```

8. Product List Price: months with no price changes?

```
Select
    CalendarMonth
    ,Count(ProductListPriceHistory.StartDate   ) as TotalRows
From Calendar
    left join ProductListPriceHistory
        on ProductListPriceHistory.StartDate = Calendar.CalendarDate
Where
    Calendar.CalendarDate >=
(Select Min(StartDate) from ProductListPriceHistory)
    and Calendar.CalendarDate <=
(Select Max(StartDate) from ProductListPriceHistory)
Group by
    CalendarMonth
Order by CalendarMonth;
```

Discussion

A calendar table can be very useful for many different types of queries, and we'll be using it again in upcoming problems. The main benefits are:

- It gives you a set of all dates that you can join to, as above.
- It provides some pre-calculated fields (like in our Calendar table, the WeekStartDate and CalendarMonth). It's easier to use these, than to calculate them every time you need them.

Here's a good resource on calendar tables:

https://www.sqlshack.com/designing-a-calendar-table/

9. Current list price of every product

```
Select
    ProductID
    ,ListPrice
From ProductListPriceHistory
Where
    EndDate is null
Order by
    ProductID;
```

Discussion

Normally in history tables of this type, the record where the EndDate is null is the one that contains the current price.

Occasionally, to make it easier to query, there's an IsActive flag field which is set to True (1) for the active record.

10. Products without a list price history

```
Select
    ProductID
    ,ProductName
From Product
Where
    ProductID not in (Select ProductID from ProductListPriceHistory)
Order by ProductID;
```

Discussion

The above is the most straightforward way of answering this question. The NOT IN is very easy to read and understand. The other common alternatives are the LEFT JOIN , and NOT EXISTS. Feel free to research online for the advantages and disadvantages of the different approaches.

I like NOT IN the best, when it's possible to use it. But frequently, it's not possible because it's only suitable for simple problems, like the above.

Here is the LEFT JOIN version of the answer:

```
Select
    Product.ProductID
    ,Product.ProductName
From Product
    Left join ProductListPriceHistory PriceHistory
        on PriceHistory.ProductID = Product.ProductID
Where
    PriceHistory.ProductID is null;
```

And here is the NOT EXISTS version:

```
Select
    ProductID
    ,ProductName
From Product
Where
    not exists
        (
        Select *
        From ProductListPriceHistory History
        Where
            History.ProductID = Product.ProductID
        )
Order by ProductID;
```

11. Product cost on a specific date, part 3

```
Select
    Distinct ProductID
From ProductCostHistory
Where
    ProductID not in (
        Select
            ProductID
            -- ,StandardCost
        From ProductCostHistory
```

```
        Where
            '2014-04-15' Between StartDate and IfNull(EndDate, Now())
        -- Order By ProductID
    )
Order by ProductID;
```

Discussion

Notice that in the NOT IN subquery, I just copied the SQL from the previous answer. What happens if you don't comment out the StandardCost field?

Take a look at some of the ProductIDs that show up in the Expected Results. Why didn't they appear in the original query? For the ProductIDs that are missing – are they all missing for the same reason?

12. Products with multiple current list price records

```
Select
    ProductID
From ProductListPriceHistory
Where EndDate is null
Group By ProductID
Having Count(*) > 1;
```

Discussion

This type of record should not exist in a price history table, of course. But we very frequently have to find and deal with bad data. You need table level constraint or something similar to prevent this type of problem record.

13. Products with their first and last order date, including name and subcategory

```
Select
    Product.ProductID
    ,ProductName
    ,ProductSubCategoryName
    ,Date(Min(OrderDate)) as FirstOrder
    ,Date(Max(OrderDate)) as LastOrder
From Product
    left Join SalesOrderDetail  Detail
        on Product.ProductID = Detail.ProductID
    left join SalesOrderHeader  Header
        on Header.SalesOrderID = Detail .SalesOrderID
    left join ProductSubCategory ProdSubCat
        on ProdSubCat.ProductSubCategoryID = Product.ProductSubCategoryID
Group by
    Product.ProductID
    ,ProductName
    ,ProductSubCategoryName
Order by ProductName;
```

Discussion

It's most common to set up the order of the tables in the from clause so that the main table (the one for which you show all records, in this case Product) is first. This is easier to read.

14. Products with list price discrepancies

```
Select
    Prod.ProductID
    ,ProductName
    ,Prod.ListPrice as Prod_ListPrice
    ,PriceHist.ListPrice as PriceHist_LatestListPrice
    ,Prod.ListPrice - PriceHist.ListPrice as Diff
From Product Prod
    join ProductListPriceHistory PriceHist
        on PriceHist.ProductID = Prod.ProductID
Where
```

```
    Prod.ListPrice <> PriceHist.ListPrice
    and PriceHist.EndDate is null
Order by Prod.ProductID;
```

Discussion

For naming the output columns, it's always better to be crystal clear. For instance, for the latest list price from ProductListPriceHistory, I started out with LatestListPrice as the name, then decided it would be more clear to call it PriceHist_LatestListPrice. The underscore in the middle makes it easier to read.

One of the principles of relational database design is that data should never be duplicated. Sometimes a design decision is made that involves duplication of data, as in this case (the latest list price of a product is stored both in the Product table and in the ProductListPriceHistory table).

What are some ways this problem could have been avoided?

15. Orders for products that were unavailable

```
Select
    Detail.ProductID
    ,Date(Header.OrderDate) as OrderDate
    ,Product.ProductName
    ,Detail.OrderQty as Qty
    ,Date(Product.SellStartDate) as SellStartDate
    ,Date(Product.SellEndDate) as SellEndDate
From SalesOrderHeader Header
    join SalesOrderDetail Detail
        on Detail.SalesOrderID = Header.SalesOrderID
    join Product
        on Detail.ProductID = Product.ProductID
Where
    OrderDate not between SellStartDate and IfNull(SellEndDate , OrderDate)
Order by
    ProductID
    ,OrderDate;
```

16. Orders for products that were unavailable: details

```
Select
    Detail.ProductID
    ,Date(Header.OrderDate) as OrderDate
    ,Detail.OrderQty as Qty
    ,Date(Product.SellStartDate) as SellStartDate
    ,Date(Product.SellEndDate) as SellEndDate
    ,Case
        When Header.OrderDate < Product.SellStartDate
            then 'Sold before start date'
        When Header.OrderDate > Product.SellEndDate
            then 'Sold after end date'
        End as ProblemType
From SalesOrderHeader Header
    join SalesOrderDetail Detail
        on Detail.SalesOrderID = Header.SalesOrderID
    join Product
        on Detail.ProductID = Product.ProductID
Where
    OrderDate not between
        SellStartDate and IfNull(SellEndDate , OrderDate)
Order by
    ProductID
    ,OrderDate;
```

17. OrderDate with time component

There are many potential answers, and in one of the following problems we'll be evaluating the pros and cons of some different answers.

For now, here's an answer that works. Later we'll look at some other potential answers.

```
Select
    (Select Count(*) from SalesOrderHeader where OrderDate <> Date(OrderDate) )
        as TotalOrderWithTime
    ,(Select Count(*) from SalesOrderHeader )
        as TotalOrders
    ,(Select Count(*) from SalesOrderHeader where OrderDate <> Date(OrderDate))
    /
```

```
    (Select Count(*) from SalesOrderHeader )
        as PercentOrdersWithTime
    ;
```

18. Fix this SQL! Number 1

Here is the correct SQL:

```
Select
    Product.ProductID
    ,ProductName
    ,ProductSubCategoryName
    ,Date(Min(OrderDate)) as FirstOrder
    ,Date(Max(OrderDate)) as LastOrder
From Product
    left Join SalesOrderDetail  Detail
        on Product.ProductID = Detail.ProductID
    left join SalesOrderHeader  Header
        on Header.SalesOrderID = Detail .SalesOrderID
    left join ProductSubCategory ProdSubCat
        on ProdSubCat.ProductSubCategoryID = Product.ProductSubCategoryID
Where
    Color = 'Silver'
Group by
    Product.ProductID
    ,ProductName
    ,ProductSubCategoryName
Order by LastOrder desc;
```

Notice that the field name Color doesn't have quotes around it anymore. When you put quotes around it, it is treated it as a simple string, and of course the string 'Color' does not match the string 'Silver'.

19. Raw margin quartile for products

```
Select
    ProductID
    ,ProductName
    ,StandardCost
    ,ListPrice
    ,ListPrice - StandardCost as RawMargin
    ,NTILE(4) OVER(ORDER BY ListPrice - StandardCost desc ) as Quartile
From Product
Where
    ListPrice <> 0
    and StandardCost <> 0
Order by ProductName;
```

Discussion

Before window functions were introduced , this problem would have been a lot more difficult to solve, potentially involving temporary tables and multiple subqueries. The NTILE function makes it simple.

20. Customers with purchases from multiple sales people

```
Select
    Customer.CustomerID
    ,Concat(FirstName, ' ', LastName) as CustomerName
       ,Count(distinct SalesPersonEmployeeID) as TotalDifferentSalesPeople
From SalesOrderHeader
    join Customer
        on Customer.CustomerID = SalesOrderHeader.CustomerID
Group by
    Customer.CustomerID
    ,Concat(FirstName, ' ', LastName)
Having Count(distinct SalesPersonEmployeeID) > 1
Order by CustomerName;
```

Discussion

Here's an alternative way of filtering out for only those customers who have used multiple sales people:

```
Having Min(SalesPersonEmployeeID)  <> Max(SalesPersonEmployeeID)
```

If you do it this way, you don’t need this clause:

```
Having Count(distinct SalesPersonEmployeeID) > 1
```

This may be more intuitive, but the first solution will also give you the count of the different sales people used.

21. Fix this SQL! Number 2

```
Select
    Customer.CustomerID
    ,Concat(FirstName, ' ', LastName) as CustomerName
    ,OrderDate
    ,SalesOrderHeader.SalesOrderID
    ,SalesOrderDetail.ProductID
    ,Product.ProductName
    ,LineTotal
From  SalesOrderHeader
    join SalesOrderDetail
        on SalesOrderHeader .SalesOrderID = SalesOrderDetail.SalesOrderID
    Join Product
        on Product.ProductID = SalesOrderDetail .ProductID
    Join Customer
        on Customer.CustomerID = SalesOrderHeader.CustomerID
Order by
    CustomerID
    ,OrderDate
Limit 100;
```

Discussion

There are many potential causes for this error. In this particular case, the SalesOrderDetail table is being referenced before it shows up in the From clause.

22. Duplicate product

```
Select ProductName
From Product
Group by ProductName
Having Count(*) > 1;
```

23. Duplicate product: details

```
With Main as (
    Select
        ProductID
        ,ProductName
        ,Row_Number() Over (Partition by ProductName Order by ProductID) as RowNumber
    From Product
 )
Select
    Main.ProductID as PotentialDuplicateProductID
    ,ProductName
From Main
Where
    RowNumber <> 1;
```

Answers: Advanced Problems

24. How many cost changes do products generally have?

```
with Main as (
    Select
        ProductID
        ,Count(*)  as TotalPriceChanges
    From ProductCostHistory
    Group by ProductID
)
Select
    TotalPriceChanges
    , Count(*) as TotalProducts
From Main
Group by TotalPriceChanges
Order by TotalPriceChanges;
```

25. Size and base ProductNumber for products

Here's one way of getting the right results.

```
Select
    ProductID
    ,ProductNumber
    ,Locate('-', ProductNumber) as HyphenLocation
    ,Case
        When Locate('-', ProductNumber) = 0 then ProductNumber
        Else Substring(ProductNumber, 1, Locate('-', ProductNumber) - 1)
```

```
    End as BaseProductNumber
    ,Case
        When Locate('-', ProductNumber) = 0 then null
        Else Substring(ProductNumber, Locate('-', ProductNumber) + 1 , 2)
    End as Size
From Product
Where ProductID > 533
Order by ProductID;
```

The above SQL works fine. However, one disadvantage is that it duplicates the function to get the hyphen location 5 times.

It's not so bad in this relatively simple query. But in queries that have multiple, complex functions, that kind of duplication of code can cause bugs when changes are necessary, because the changes are not necessarily made everywhere.

A better solution is the below SQL. Note that the function to find the hyphen is done only once.

```
with Main as (
    Select
        ProductID
        ,ProductName
        ,ProductNumber
        ,Locate('-', ProductNumber) as HyphenLocation
    From Product
    Where ProductID > 533
)
Select
    *
    ,Case
        When HyphenLocation = 0 then ProductNumber
        Else Substring(ProductNumber, 1, HyphenLocation  - 1)
    End as BaseProductNumber
    ,Case
        When HyphenLocation  = 0 then null
        Else Substring(ProductNumber, HyphenLocation  + 1 , 2)
    End as Size
From Main
Order by ProductID;
```

If possible, the best solution would be to have a separate Size field in the table.

26. Number of sizes for each base product number

```
with Main as (
    Select
        ProductID
        ,ProductName
        ,ProductNumber
        ,Locate('-', ProductNumber) as HyphenLocation
    From Product
    Where
        ProductSubCategoryID in
            (
            Select ProductSubCategoryID
            From ProductSubCategory
            where ProductCategoryID = 3
            )
)
, ProductWithSizes as (
    Select
        *
        ,Case
            When HyphenLocation = 0 then ProductNumber
            Else Substring(ProductNumber, 1, HyphenLocation  - 1)
        End as BaseProductNumber
        ,Case
            When HyphenLocation  = 0 then null
            Else Substring(ProductNumber, HyphenLocation  + 1 , 2)
        End as Size
    From Main
)
Select
    BaseProductNumber
    ,Count(*) as TotalSizes
From ProductWithSizes
Group by BaseProductNumber
Order by BaseProductNumber;
```

Discussion

This SQL is based on the second version of the answer to the “Size and base ProductNumber for products” problem.

Note that I did a subselect with an “in” to get the correct products in the correct category. An alternative would be to join to the ProductSubCategory and ProductCategory tables.

27. How many cost changes has each product really had?

```
with Main as (
    Select
        ProductID
        ,StartDate
        ,StandardCost
        ,Lag(StandardCost, 1) over (Partition by ProductID Order by StartDate)
            as PreviousStandardCost
    From ProductCostHistory
)
Select
    Main.ProductID
    ,ProductName
    ,Count(*) as TotalCostChanges
From Main
    join Product
        on Product.ProductID = Main.ProductID
Where
    Main.PreviousStandardCost <> Main.StandardCost
    or PreviousStandardCost is null
Group By
    Main.ProductID
    ,ProductName
Order by
    Main.ProductID;
```

28. Which products had the largest increase in cost?

```
with Main as (
    Select
        ProductID
        ,StartDate
        ,StandardCost
        ,Lag(StandardCost, 1) over (Partition by ProductID Order by StartDate)
            as PreviousStandardCost
    From ProductCostHistory
```

```
)
Select
    Main.ProductID
    ,Main.StartDate as CostChangeDate
    ,Main.StandardCost
    ,Main.PreviousStandardCost
    ,Main.PreviousStandardCost - Main.StandardCost as PriceDifference
From Main
Where
    PreviousStandardCost is not null
Order by PriceDifference  desc;
```

29. Fix this SQL! Number 3

```
with FraudSuspects as (
    Select *
    From Customer
    Where
        CustomerID in (
            29401
            ,11194
            ,16490
            ,22698
            ,26583
            ,12166
            ,16036
            ,25110
            ,18172
            ,11997
            ,26731)
)
, SampleCustomers as (
    Select *
    From Customer
    Where
        CustomerID not in (Select CustomerID From FraudSuspects)
    Order by
        Rand()
    Limit 100
)
```

```
Select * From FraudSuspects
Union all
Select * From SampleCustomers;
```

Discussion

The above query solves the problem of duplicated customer lists, and is fine for some ad-hoc work.

Over the long run, though, a better way to handle this would be to have a separate table for the customers suspected of fraud, instead of hard-coded CustomerIDs in a query.

30. History table with start/end date overlap

```
Select
    Calendar.CalendarDate
    ,ProductID
    ,Count(*)  as TotalRows
From Calendar
    join ProductListPriceHistory
        on ProductListPriceHistory.StartDate <= Calendar.CalendarDate
        and ProductListPriceHistory.EndDate >= Calendar.CalendarDate
Group by
    Calendar.CalendarDate
    ,ProductID
Having
    Count(*) > 1
Order by
    Calendar.CalendarDate;
```

Discussion

You can also search online for something like “SQL overlapping date range”, and you'll get some information and answers. Some of them involve complex logic instead of a calendar table. But using the calendar table is the most straightforward solution.

31. History table with start/end date overlap, part 2

```
Select
    Calendar.CalendarDate
    ,ProductID
    ,Count(*)  as TotalRows
From Calendar
    join ProductListPriceHistory
        on ProductListPriceHistory.StartDate <= Calendar.CalendarDate
        and IfNull(ProductListPriceHistory.EndDate, '2013-05-29' )
            >= Calendar.CalendarDate
Group by
    Calendar.CalendarDate
    ,ProductID
Having
    Count(*) > 1
Order by
    ProductID
    ,Calendar.CalendarDate;
```

Discussion

In the original SQL Practice Problems, there were also a few problems which explicitly dealt with nulls. It's always good to remember that nulls require special processing, because it's one of the most common causes of problems when writing SQL.

If you'd like to read in depth on some issues that can occur with nulls, take a look at this article. It’s written about SQL Server, but is applicable to MySQL as well.

https://www.red-gate.com/simple-talk/sql/t-sql-programming/how-to-get-nulls-horribly-wrong-in-sql-server/

32. Running total of orders in last year

```
with FilteredOrders as (
    Select
        Calendar.CalendarMonth
        ,Count(SalesOrderID) as TotalOrders
```

```
    From SalesOrderHeader
        Join Calendar
            on Calendar.CalendarDate = Date(OrderDate)
    Where
        OrderDate >=
            Date_Add(
                (Select Date(Max(OrderDate))
                From SalesOrderHeader)
                , Interval -1 year
                )
    Group By CalendarMonth
)
Select
    CalendarMonth
    ,TotalOrders
    ,Sum(TotalOrders)  Over (Order By CalendarMonth) as RunningTotal
From FilteredOrders
Order by CalendarMonth;
```

Discussion

Most frequently, this kind of problem that requires a running sum or running total is done in a front-end tool such as Microsoft Excel. However, since Window functions were introduced in MySQL 8, it's easy to do it using just SQL as well.

33. Total late orders by territory

```
With Main as (
    Select
        SalesOrderID
        ,TerritoryID
        ,DueDate
        ,ShipDate
        ,Case
            When DueDate < ShipDate then 1
            Else 0
        End
            as OrderArrivedLate
    From SalesOrderHeader
```

```
    )
Select
    SalesTerritory.TerritoryID
    ,SalesTerritory.TerritoryName
    ,CountryCode
    ,Count(Main.SalesOrderID) as TotalOrders
    -- Sum up all the individual records
    -- where the OrderArrivedLate flag is true (1)
    ,IfNull(Sum(OrderArrivedLate),0 ) as TotalLateOrders
From SalesTerritory
    Left Join Main
        on SalesTerritory.TerritoryID = Main.TerritoryID
Group by
    SalesTerritory.TerritoryID
    ,SalesTerritory.TerritoryName
    ,CountryCode
Order by SalesTerritory.TerritoryID;
```

Discussion

One of the benefits of using the above method is that if you want to have additional descriptive calculated fields, similar to OrderArrivedLate, you can very easily add them.

An alternative method to the above answer is below. Using this method, it's more difficult to add more fields (you would need to add more CTE statements) and the SalesOrderHeader table is scanned multiple times. However, I think it's easier to read and understand.

```
With AllOrders as (
    Select
        TerritoryID
        ,Count(*) as TotalSalesOrders
    From SalesOrderHeader
    Group by TerritoryID
)
,LateOrders as (
    Select
        TerritoryID
        ,Count(*) as TotalSalesOrders
    From SalesOrderHeader
    Where
        DueDate < ShipDate
    Group by TerritoryID
)
Select
    SalesTerritory.TerritoryID
    ,SalesTerritory.TerritoryName
    ,CountryCode
    ,IfNull(AllOrders.TotalSalesOrders, 0) as TotalOrders
```

```
    ,IfNull(LateOrders.TotalSalesOrders, 0) as TotalLateOrders
From SalesTerritory
    left join AllOrders
        on SalesTerritory.TerritoryID = AllOrders.TerritoryID
    left join LateOrders
        on SalesTerritory.TerritoryID = LateOrders.TerritoryID
Order by TerritoryID;
```

34. OrderDate with time component—performance aspects

The worst solution, in terms of resources used, is solution 3.

Why do you think solution 3 is the worst?

One thing to note when reviewing performance is that it can be very different for different database management systems, or DBMSs.

For instance, when doing this comparison on SQL Server, solution 4 is clearly superior to all the others. However, when I tested it in MySQL, solution 1, 2, and 4 are equal, but solution 3 was the worst.

35. Customer's last purchase—what was the product subcategory?

```
With Main as (
    Select
        Customer.CustomerID
             ,Concat(FirstName, ' ', LastName) as CustomerName
             ,OrderDate
        ,Header.SalesOrderID
        ,Detail.ProductID
```

```
        ,Product.ProductName
        ,LineTotal
        ,ProductSubCategoryName
        ,Row_Number () over (
            Partition By Customer.CustomerID
            Order by OrderDate desc, LineTotal desc
            ) as RowNumber
    From SalesOrderDetail Detail
        join SalesOrderHeader Header
            on Header.SalesOrderID = Detail.SalesOrderID
        Join Product
            on Product.ProductID = Detail.ProductID
        Join Customer
            on Customer.CustomerID =Header.CustomerID
        left Join ProductSubCategory SubCat
            on SubCat.ProductSubCategoryID = Product.ProductSubCategoryID
    Where
        Customer.CustomerID in (
            19500
            ,24409
            ,19792
            ,26785
        )
    )
Select
    CustomerID
    ,CustomerName
    ,ProductSubCategoryName
From Main
Where RowNumber = 1;
```

Discussion

Did you at first try to get all the functionality in one step, with the Row_Number function in the where clause like this?

```
    Where Row_Number () over (
            Partition By Customer.CustomerID
            Order by OrderDate desc, LineTotal desc
            )
            = 1
```

That may seem like a logical place for it, but you'll get an error message if you try it. Window functions can only be used in the Select clause and Order By clause. That's why you need to first get the RowNumber, and then filter on it.

36. Order processing: time in each stage

```
With Main as (
    Select
        SalesOrderID
        ,TrackingEvent.TrackingEventID
        ,EventName
        ,EventDateTime as TrackingEventDate
        ,Lead(EventDateTime , 1)
            over (Partition by SalesOrderID Order by EventDateTime)
            as NextTrackingEventDate
    From OrderTracking
        join TrackingEvent
            on OrderTracking.TrackingEventID = TrackingEvent.TrackingEventID
    Where SalesOrderID in (68857, 70531, 70421)
)
Select
    SalesOrderID
    ,EventName
    ,Date_Format(TrackingEventDate, "%Y-%m-%d %H:%i") as TrackingEventDate
    ,Date_Format(NextTrackingEventDate , "%Y-%m-%d %H:%i") as NextTrackingEventDate
    ,TimeStampDiff(Hour, TrackingEventDate, NextTrackingEventDate  ) as HoursInStage
From Main
Order by
    SalesOrderID
    ,TrackingEventDate;
```

Discussion

This problem could be solved with one direct query, without the CTE. Try writing it that way. Compare the 2 methods. Which do you think is better, and why?

It's partly a matter of personal preference, but some of the factors to compare on are:

- readability
- is there code that's duplicated?
- how easy is it to troubleshoot problems?

37. Order processing: time in each stage, part 2

```
With Main as (
    Select
        SalesOrderHeader.SalesOrderID
        ,Case
            When OnlineOrderFlag = 0 Then 'Offline'
            Else 'Online'
        End
            as OnlineOfflineStatus
        ,TrackingEvent.TrackingEventID
        ,EventName
        ,EventDateTime as TrackingEventDate
        ,Lead(EventDateTime , 1)
        over (Partition by OrderTracking.SalesOrderID Order by EventDateTime)
            as NextTrackingEventDate
    From SalesOrderHeader
        join OrderTracking
            on SalesOrderHeader.SalesOrderID = OrderTracking.SalesOrderID
        join TrackingEvent
            on OrderTracking.TrackingEventID = TrackingEvent.TrackingEventID
)
Select
    OnlineOfflineStatus
    ,EventName
    ,Avg(TimeStampDiff(Hour, TrackingEventDate, NextTrackingEventDate ))
        as AverageHoursSpentInStage
From Main
Group by
    OnlineOfflineStatus
    ,EventName
    ,TrackingEventID
Order by
    OnlineOfflineStatus
    ,TrackingEventID;
```

38. Order processing: time in each stage, part 3

```
With Main as (
```

```
    Select
        SalesOrderHeader.SalesOrderID
        ,Case
            When OnlineOrderFlag = 0 Then 'Offline'
            Else 'Online'
        End
            as OnlineOfflineStatus
        ,TrackingEvent.TrackingEventID
        ,EventName
        ,EventDateTime as TrackingEventDate
        ,Lead(EventDateTime , 1)
            over (Partition by OrderTracking.SalesOrderID Order by EventDateTime)
            as NextTrackingEventDate
    From SalesOrderHeader
        join OrderTracking
            on SalesOrderHeader.SalesOrderID = OrderTracking.SalesOrderID
        join TrackingEvent
            on OrderTracking.TrackingEventID = TrackingEvent.TrackingEventID
)
, MainGrouped as (
    Select
        OnlineOfflineStatus
        ,EventName
        ,TrackingEventID
        ,Avg(TimeStampDiff(hour, TrackingEventDate, NextTrackingEventDate ))
            as AverageHoursSpentInStage
    From Main
    Group by
        OnlineOfflineStatus
        ,EventName
        ,TrackingEventID
)
Select
    Offline.EventName
    ,Offline.AverageHoursSpentInStage as OfflineAvgHoursInStage
    ,Online.AverageHoursSpentInStage as OnlineAvgHoursInStage
From (Select * From MainGrouped where OnlineOfflineStatus = 'Offline') Offline
    Join (Select * From MainGrouped where OnlineOfflineStatus = 'Online') Online
        on Offline.EventName = Online.EventName
Order by
    Offline.TrackingEventID;
```

Discussion

It's much easier to see and compare numbers when they're side by side. Also, once they're in this format, it is no problem to get the differences between online and offline for the time spent in each stage.

Note that you needed put the TrackingEventID back into the output of the CTE called MainGrouped, so that you can sort by it in the final output.

Or you could have joined each of the derived tables to the TrackingEvent table directly, and then order by the TrackingEvent.TrackingEventID.

Stacking multiple CTEs like this is fine, until it becomes difficult to read, or you start having performance problems. The next step is to create temporary tables, potentially with indexes.

39. Top three product subcategories per customer

```
With Main as (
    Select
        Customer.CustomerID
             ,Concat(FirstName, ' ', LastName) as CustomerName
        ,LineTotal = Sum(LineTotal)
        ,ProductSubCategoryName
        ,Row_Number ()
        over (
            Partition By Customer.CustomerID
            Order by Sum(LineTotal) desc
            )
            as RowNumber
    From SalesOrderDetail Detail
        join SalesOrderHeader Header
            on Header.SalesOrderID = Detail.SalesOrderID
        Join Product
            on Product.ProductID = Detail.ProductID
        Join Customer
            on Customer.CustomerID =Header.CustomerID
        left Join ProductSubCategory SubCat
            on SubCat.ProductSubCategoryID = Product.ProductSubCategoryID
    Where
        Customer.CustomerID in (
            13836
            ,26313
            ,20331
            ,21113
            ,13763
        )
    Group by
```

```
        Customer.CustomerID
        ,FirstName + ' ' + LastName
        ,ProductSubCategoryName
)
Select
    ProductSubCat1.CustomerID
    ,ProductSubCat1.CustomerName
    ,Min(ProductSubCat1.ProductSubCategoryName) as TopProdSubCat1
    ,Min(ProductSubCat2.ProductSubCategoryName) as TopProdSubCat2
    ,Min(ProductSubCat3.ProductSubCategoryName) as TopProductSubCat3
From (Select * From Main where RowNumber = 1 ) ProductSubCat1
    Left Join (Select * From Main where RowNumber = 2 ) ProductSubCat2
        on ProductSubCat2.CustomerID = ProductSubCat1.CustomerID
    Left Join (Select * From Main where RowNumber = 3 ) ProductSubCat3
        on ProductSubCat3.CustomerID = ProductSubCat1.CustomerID
Group by
    ProductSubCat1.CustomerID
    ,ProductSubCat1.CustomerName
Order by
    ProductSubCat1.CustomerID;
```

Discussion

This is a common request when doing customer segmentation and cross selling. Usually you would put the output of a query like this into a new table that data analysts or marketers could easily query to get more insights about customers.

40. History table with date gaps

```
with HistoryWithMinMax as (
    Select
        ProductID
        ,Min(StartDate) as FirstStartDate
        ,Max(IfNull(EndDate, '2014-05-29')) as LastEndDate
    From ProductListPriceHistory
    Group by ProductID
)
,ProductWithDates as (
```

```
    Select
        HistoryWithMinMax.ProductID
        ,Calendar.CalendarDate
    From HistoryWithMinMax
        left Join Calendar
            on Calendar.CalendarDate
                between
                HistoryWithMinMax.FirstStartDate
                and
                HistoryWithMinMax.LastEndDate
)
Select
    ProductWithDates.ProductID
    ,ProductWithDates.CalendarDate as DateWithMissingPrice
From ProductWithDates
    left join ProductListPriceHistory History
        on ProductWithDates.CalendarDate
            between
                History.StartDate
                and
                IfNull(History.EndDate, '2014-05-29')
        and History.ProductID = ProductWithDates.ProductID
Where
    History.ProductID is null;
```

Discussion

I hope you enjoyed this problem and didn't get too frustrated! I organized the hints to lead to this particular answer, but there are multiple different ways of solving this problem.

If you happen to find a way that's significantly different, feel free to email me (feedback@SQLPracticeProblems.com) and let me know!

Congratulations!

You're finished! Now that you've completed the practice problems, you've improved your SQL skills tremendously, and increased your ability in a skill that's in enormous demand.

Any comments and suggestions are most welcome! Please email me at:
feedback@SQLPracticeProblems.com.

Thank you!

Sylvia Moestl Vasilik