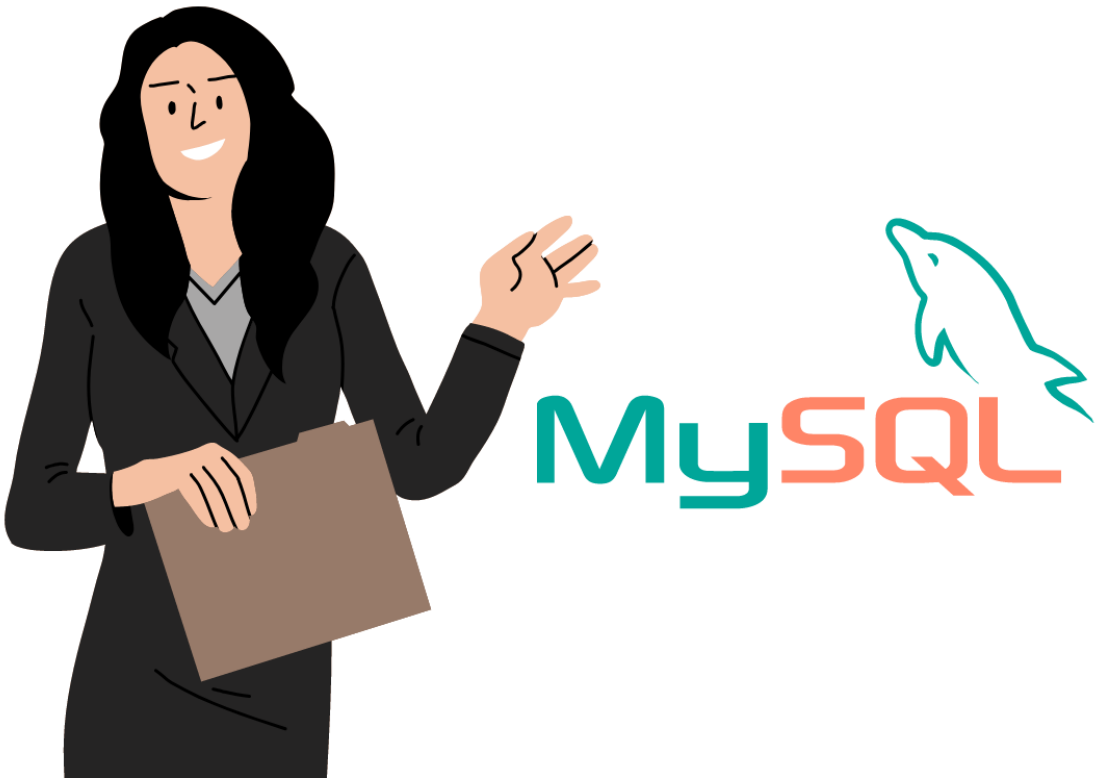
MySQL Interview Questions

Interview questions to learn MySQL from the basics and get ready for your next Data Science Interview involving the most popular open-source SQL database in the world.



MySQL is one of the most popular databases in use and the most popular open-source database and was named the Database of the Year in 2019. Conceptualized in the mid-1990s, MySQL was initially acquired by Sun Microsystems, which in turn was acquired by Oracle Corporation. MySQL is built on C and C++ and works on multiple platforms - Windows, Mac, Linux. It is available both in GNU General Public License as well as Proprietary licenses. MySQL is the M in the LAMP stack used to develop web applications. Here is a brief comparison of the differences between the top SQL flavors.

	Microsoft SQL Server	MySQL	Oracle	PostgreSQL
Supporting Operating Systems	Windows Linux	Windows Linux UNIX MacOS	Windows Linux UNIX MacOS	Windows Linux UNIX MacOS
Features	Union Intersect Except Common Table Expressions Windowing Functions Parallel Query	Union Common Table Expressions Windowing Functions	Union Intersect Except Common Table Expressions Windowing Functions Parallel Query	Union Intersect Except Common Table Expressions Windowing Functions Parallel Query

Unlike Windows SQL server that works only on Windows (obviously!!) and Linux, the other top SQL flavors including MySQL work on all top OS. When it comes to features, MySQL appears to have the least number of features available. However, it is highly unlikely that one will miss these features especially as a Data Analyst or a Data Scientist.

While starting off with learning SQL, MySQL is a good place to start and one should not worry too much about the missing functions. New features and functionalities keep getting added all the time. For instance, suppose you want to find the day of the month from a date value (18 from 18th Sep 2022). Using MySQL, we can simply use the built-in `DAY` function. However, there is no corresponding function in Postgres. But we can accomplish the same using the `EXTRACT` function which is present in both the dialects. You can learn about the differences and similarities between MySQL and Postgres in detail here.

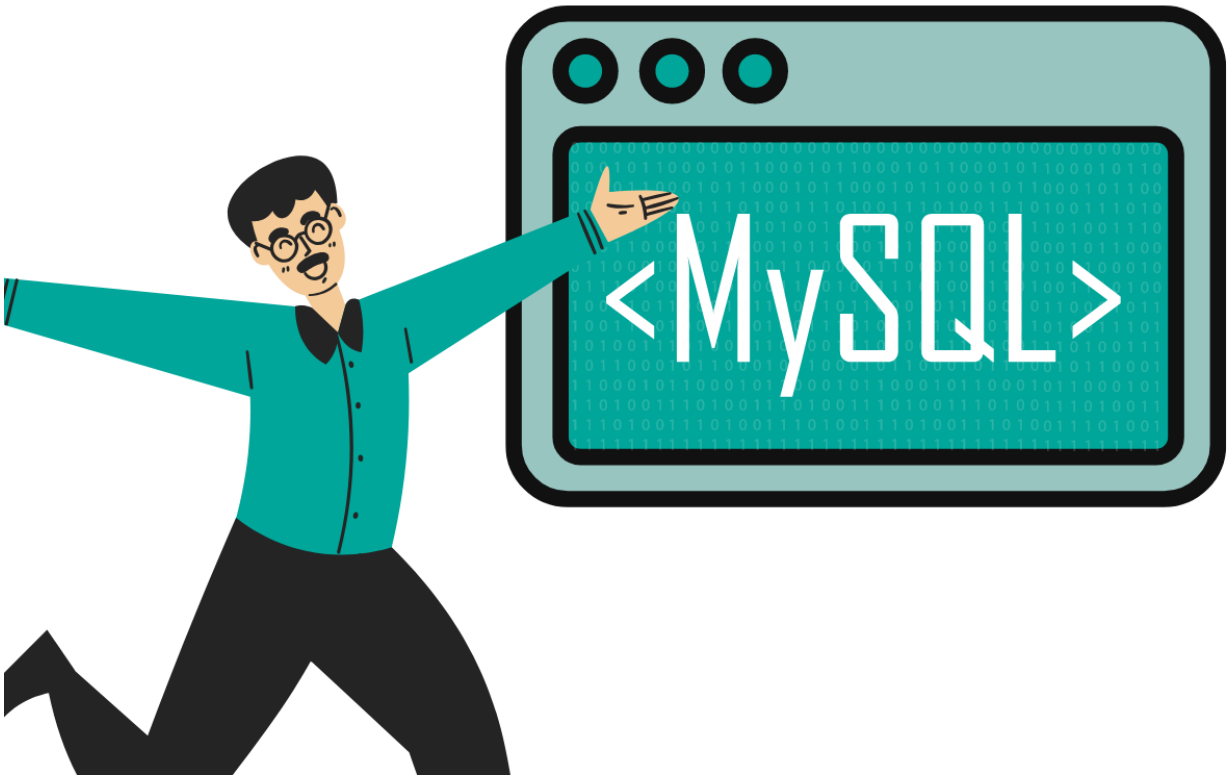
Fun Fact: The “My” in MySQL comes from the name of the daughter of Michael Widenius's daughter My. His other two kids, Maria and Max, too have databases named after them.

MySQL for Data Analysts and Data Scientists

In this article, we will look at the application of MySQL from a Data Science and Data Analyst point of view. I would suggest that you also check out our SQL Interview Quesions preparation guide and The 30 frequently asked SQL Query Interview Questions to complete your interview

preparation for your next data science role. You can solve all these MySQL interview questions and many more on the StrataScratch platform.

Basic MySQL Interview Questions



Let us start from the beginning. Once the data is in the database, we need to start querying it to find the information and insights that we need.

SELECT

The SELECT statement is the heart of SQL. As the name suggests, it simply fetches the information. We can perform mathematical operations.

```
SELECT 2 + 3;
```

```
2 + 3
```

```
5
```

We can find the current date

```
SELECT CURRENT_DATE();
```

```
CURRENT_DATE()
```

```
2022-09-11
```

One can also check the version of MySQL available. This is helpful in identifying which functions one can actually use.

```
SELECT VERSION();
```

```
VERSION()
```

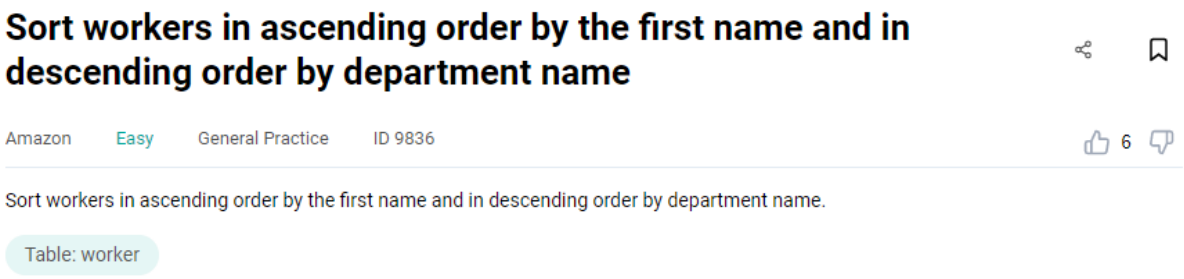
```
8.0.26
```

While these return a single value, usually we will be querying a database table. Let us try querying a table.

SELECT FROM

Let us try to query a full table. Since we do not know the column names, we can simply use the * operator to get all the fields. You can try the code on the below MySQL interview question example.

MySQL Interview Question #1: Sort workers in ascending order by the first name and in descending order by department name



Link to the question: https://platform.stratascratch.com/coding/9836-sort-workers-in-ascending-order-by-the-first-name-and-in-descending-order-by-department-name

```
SELECT * FROM worker;
```

worker_id	first_name	last_name	salary	joining_date	department
1	Monika	Arora	100000	2014-02-20 09:00:00	HR
2	Niharika	Verma	80000	2014-06-11 09:00:00	Admin
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account

While this is convenient, one should use SELECT * only when one knows that she will be using all the fields. Further, right now we are extracting the full table. While this table is tiny, in real life,

tables can run into billions and trillions of rows and hundreds of columns. If you want to explore the data, use only a few rows.

LIMIT

As the name suggests, the LIMIT clause returns only a specific number of rows. Instead of getting the full results, let us just get five rows.

```
SELECT * FROM worker
LIMIT 5
;
```

worker_id	first_name	last_name	salary	joining_date	department
1	Monika	Arora	100000	2014-02-20 09:00:00	HR
2	Niharika	Verma	80000	2014-06-11 09:00:00	Admin
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin

LIMIT clause is very helpful to research large tables and understand what the data looks like. It can also be used to make sure that the query is correct and fetches the data one actually needs.

Selecting a specific column

Instead of selecting all the columns, most of the time we will need only specific columns. We can do that by specifying the columns in the order that we want in the output. Let us select the `department`, `last_name`, `first_name`, `joining_date`, and the `salary` in that particular order.

```
SELECT
    department
    , last_name
    , first_name
    , joining_date
    , salary
FROM worker
;
```

department	last_name	first_name	joining_date	salary
HR	Arora	Monika	2014-02-20 09:00:00	100000
Admin	Verma	Niharika	2014-06-11 09:00:00	80000
HR	Singhal	Vishal	2014-02-20 09:00:00	300000
Admin	Singh	Amitah	2014-02-20 09:00:00	500000
Admin	Bhati	Vivek	2014-06-11 09:00:00	500000
Account	Diwan	Vipul	2014-06-11 09:00:00	200000
Account	Kumar	Satish	2014-01-20 09:00:00	75000
Admin	Chauhan	Geetika	2014-04-11 09:00:00	90000

Filtering Data using the WHERE clause

Just as we specify the columns to keep, we can also filter out the rows based on specific conditions. To do this, we use the WHERE clause. Let us try to find all the rows where the salary is greater than 100,000. This is pretty simple.

```
SELECT *
FROM worker
WHERE salary > 100000
;
```

worker_id	first_name	last_name	salary	joining_date	department
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account

We can use multiple filters in the WHERE clause by using AND, OR, and NOT logical operators.

Here are a few examples -

Find workers in the HR or Admin Department.

```
SELECT
*
FROM worker
WHERE department = 'HR' OR department = 'Admin'
;
```

Find the workers in the Admin department earning a salary lesser than 150,000

```
SELECT
*
FROM worker
WHERE
    department = 'Admin'
    AND salary < 150000
;
```

Find all the workers except for those in the HR department.

```
SELECT
*
FROM worker
WHERE
    NOT (department = 'HR')
;
```

Sorting Data

With the SELECT statement, the output rows are not in a specific order. We can order the output in our desired order. Let us sort the rows in the order of their first names. To do this we use the ORDER BY clause.

```
SELECT
*
FROM worker
ORDER BY first_name
```

```
;
```

worker_id	first_name	last_name	salary	joining_date	department
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
8	Geetika	Chauhan	90000	2014-04-11 09:00:00	Admin
1	Monika	Arora	100000	2014-02-20 09:00:00	HR
2	Niharika	Verma	80000	2014-06-11 09:00:00	Admin
7	Satish	Kumar	75000	2014-01-20 09:00:00	Account
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin

If the order is not specified, it sorts in ascending order. To sort in descending order, we need to specify DESC after the column name. Let us sort the output in the reverse alphabetical order of their first names.

```
SELECT
*
FROM worker
ORDER BY first_name DESC
;
```

worker_id	first_name	last_name	salary	joining_date	department
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account
7	Satish	Kumar	75000	2014-01-20 09:00:00	Account
2	Niharika	Verma	80000	2014-06-11 09:00:00	Admin
1	Monika	Arora	100000	2014-02-20 09:00:00	HR
8	Geetika	Chauhan	90000	2014-04-11 09:00:00	Admin
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin

We also specify multiple sorting orders. Let us sort the output in descending order of the salaries and then by the department name in alphabetical order.

```
SELECT
*
FROM worker
ORDER BY salary DESC, department
;
```

worker_id	first_name	last_name	salary	joining_date	department
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account
1	Monika	Arora	100000	2014-02-20 09:00:00	HR
8	Geetika	Chauhan	90000	2014-04-11 09:00:00	Admin
2	Niharika	Verma	80000	2014-06-11 09:00:00	Admin
7	Satish	Kumar	75000	2014-01-20 09:00:00	Account

Instead of specifying the full column name, we can also specify the position of the columns in the output.

1	2	3	4	5	6
worker_id	first_name	last_name	salary	joining_date	department
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account
1	Monika	Arora	100000	2014-02-20 09:00:00	HR

We can get the same output using the following.

```
SELECT
*
```

```
FROM worker
ORDER BY 4 DESC, 6
;
```

You can read more about sorting in SQL in our *"Ultimate Guide to Sorting in SQL"*.

Finding Unique values

To analyze a dataset, one might want to find the different values that are present in the table. To find the unique values, one can use the DISTINCT clause. The DISTINCT clause can also be used to eliminate duplicate rows. Let us try to find the different departments in the table. To do this, we apply the DISTINCT clause on the department row.

```
SELECT
DISTINCT department
FROM worker
;
```

department
HR
Admin
Account

We can also find the different salary values in the dataset.

```
SELECT
DISTINCT salary
FROM worker
;
```

salary
100000
80000
300000
500000
200000
75000
90000

If we apply the DISTINCT clause on more than one row, it will find all the unique combinations of the rows in the table. Note this will not give all the possible combinations, just those combinations that are present in the table

```
SELECT
DISTINCT department, salary
FROM worker
;
```

department	salary
HR	100000
Admin	80000
HR	300000
Admin	500000
Account	200000
Account	75000
Admin	90000

Aggregations

We can also calculate summary values for a given column. The salient aggregate functions available in MySQL are

Name	Description
AVG()	Return the average value of the argument
COUNT()	Return a count of the number of rows returned
COUNT(DISTINCT)	Return the count of a number of different values
MAX()	Return the maximum value
MIN()	Return the minimum value
SUM()	Return the sum

Let us try a few of these.

If we want to find the number of rows in the table, we can simply use the COUNT(*) option. COUNT(*) is the only aggregation function that will consider even NULL values.

```
SELECT COUNT(*) FROM worker;
```

We can use COUNT in conjunction with the DISTINCT clause. This will return the number of unique values in the rows.

```
SELECT
COUNT(DISTINCT department)
FROM worker
;
```

COUNT(DISTINCT department)
3

There are various other aggregation functions for text, JSON, and bit-wise operations in MySQL. The full list of aggregate functions is available here. We have a detailed article on different aggregation functions and their applications here in this ultimate guide "*SQL Aggregate Functions*".

Grouping Data

As of now, we have aggregated values for the entire column. We can also calculate aggregate metrics by subcategories. Let us calculate the average salary for each department. Let us start off by calculating the overall average. We can use the AVG() function.

```
SELECT
AVG(salary)
FROM worker
;
```

AVG(salary)
230625

We can now split this by each department by using the GROUP BY clause. As the name suggests, this clause will calculate the aggregate metric for each subcategory.

```
SELECT
    department
    , AVG(salary)
FROM worker
GROUP BY 1
;
```

department	AVG(salary)
HR	200000
Admin	292500
Account	137500

We can use the WHERE clause in conjunction with the GROUP BY operation. The WHERE clause will subset the data based, and the GROUP BY operation will then be applied to the subset. Suppose we want to consider only those salaries that are lesser than 200,000, we can do it in the following manner.

```
SELECT
    department
    , AVG(salary)
FROM worker
```

```
WHERE salary < 200000
GROUP BY 1
;
```

department	AVG(salary)
HR	100000
Admin	85000
Account	75000

The WHERE condition will return only those records that have a salary lesser than 200,000. We can verify this.

```
SELECT
*
FROM worker
WHERE salary < 200000
;
```

On occasions, we might want to filter the aggregated value. For example, if we want to find the departments that have an average of more than 150,000. Let us start by finding the average again.

```
SELECT
    department
    , AVG(salary)
FROM worker
GROUP BY 1
;
```

department	AVG(salary)
HR	200000
Admin	292500
Account	137500

We cannot use the WHERE condition here because it is executed before the GROUP BY clause aggregates based on different subcategories; we shall explain this a bit more later in the article. If we want to filter the aggregated value, we use the HAVING clause.

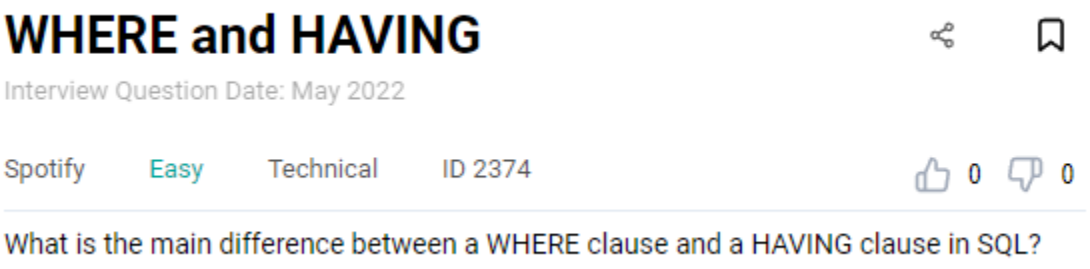
```
SELECT
    department
    , AVG(salary)
FROM worker
GROUP BY 1
HAVING AVG(salary) > 150000
;
```

department	AVG(salary)
HR	200000
Admin	292500

WHERE vs HAVING

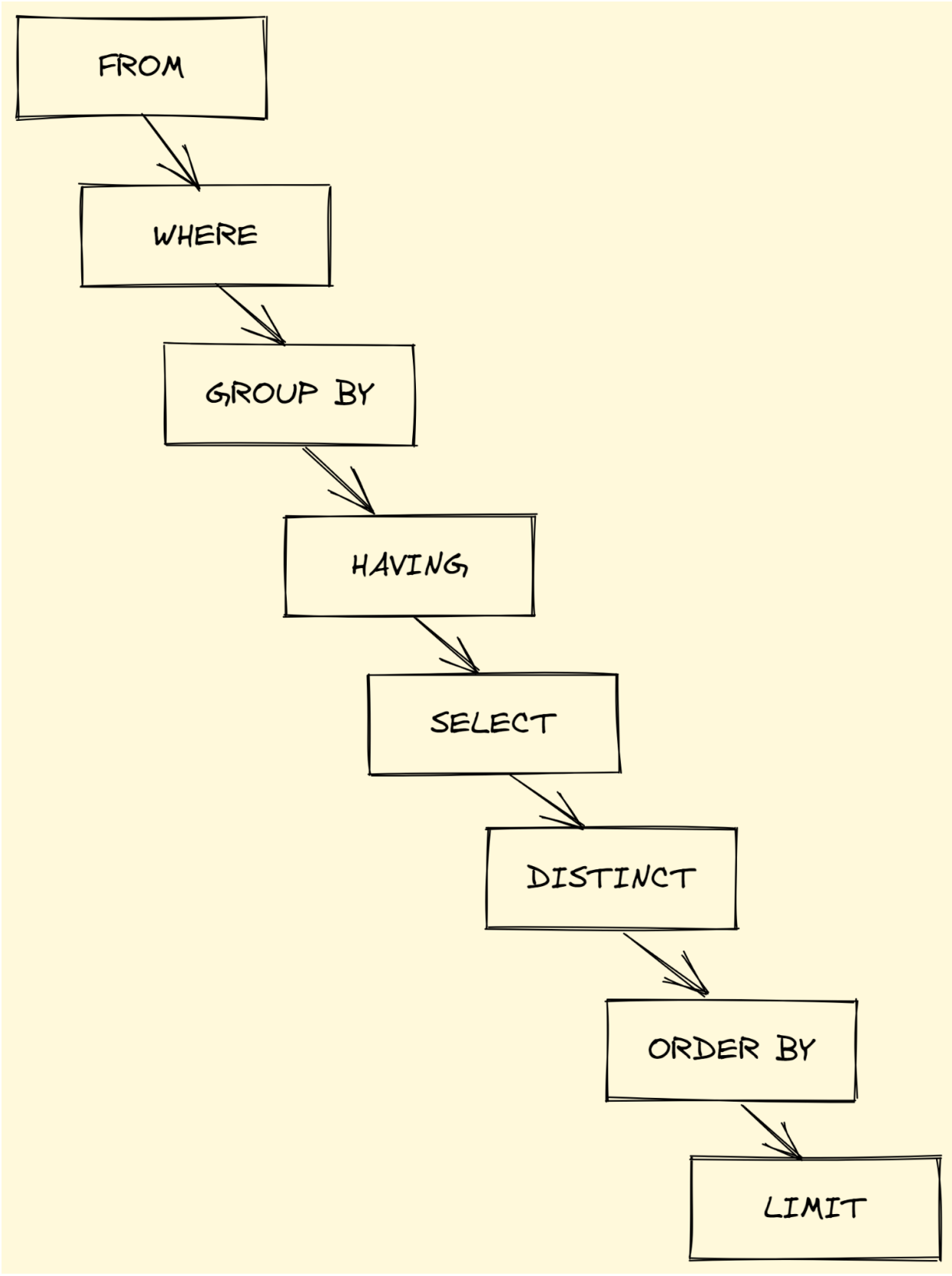
A very common MySQL interview question is the difference between WHERE and HAVING and when to use them.

MySQL Interview Question #2: WHERE and HAVING



Link to the question: https://platform.stratascratch.com/technical/2374-where-and-having

The difference stems from how MySQL (or any other SQL engine) executes a query.


FROM
WHERE
GROUP BY
HAVING
SELECT
DISTINCT
ORDER BY
LIMIT

We cannot use aggregations in the WHERE clause because it is executed before the GROUP BY. On the other hand, since HAVING is executed after the GROUP BY, we must necessarily use an aggregation in the HAVING clause.

ALIASES

It might be useful to be able to rename the output columns to a more readable output. Let us rename the AVG(salary) field in the output.

```
SELECT
    department
    , AVG(salary) AS average_salary
FROM worker
GROUP BY 1
HAVING AVG(salary) > 150000
;
```

department	average_salary
HR	200000
Admin	292500

Note: we cannot use the ALIAS in the WHERE or the HAVING clause.

If we want spaces or special characters in the output, we need to quote them.

```
SELECT
    department
    , AVG(salary) AS "average salary in $"
FROM worker
GROUP BY 1
HAVING AVG(salary) > 150000
;
```

department	average salary in $
HR	200000
Admin	292500

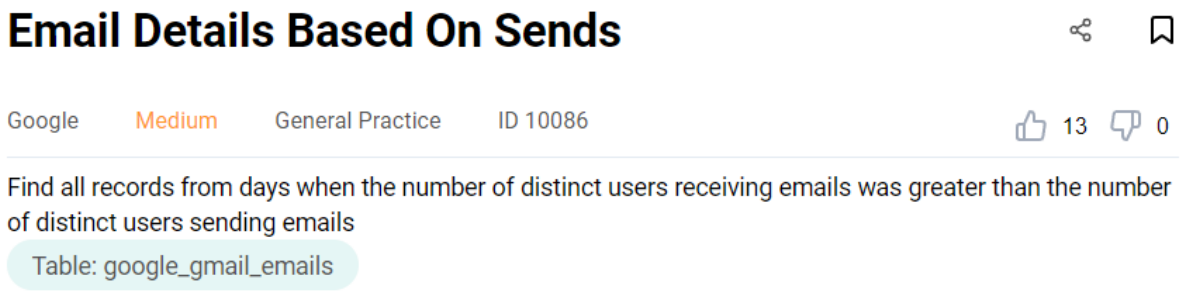
Intermediate MySQL Interview Questions

Now that we have the basics in place, let us go through the intermediate-level MySQL interview questions and amp up our skills.

Subqueries, CTEs, and TEMP tables

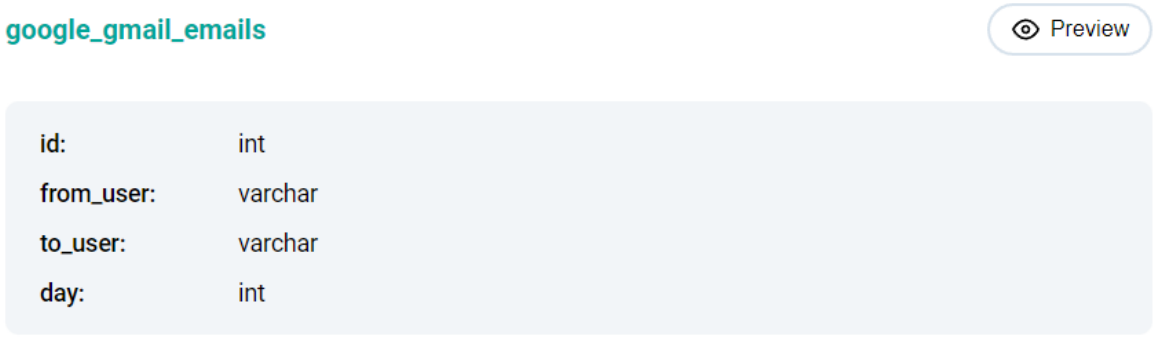
Sometimes it is not possible to get the desired output in a single query. Let us consider the following problem.

MySQL Interview Question #3: Email Details Based On Sends



Link to the question:
https://platform.stratascratch.com/coding/10086-email-details-based-on-sends

This problem uses the `google_gmail_emails` table with the following fields.



The data is presented thus.

id	from_user	to_user	day
0	6edf0be4b2267df1fa	75d295377a46f83236	10
1	6edf0be4b2267df1fa	32ded68d89443e808	6
2	6edf0be4b2267df1fa	55e60cfcc9dc49c17e	10
3	6edf0be4b2267df1fa	e0e0defbb9ec47f6f7	6
4	6edf0be4b2267df1fa	47be2887786891367e	1
5	6edf0be4b2267df1fa	2813e59cf6c1ff698e	6
6	6edf0be4b2267df1fa	a84065b7933ad01019	8
7	6edf0be4b2267df1fa	850badf89ed8f06854	1
8	6edf0be4b2267df1fa	6b503743a13d778200	1
9	6edf0be4b2267df1fa	d63386c884aeb9f71d	3
10	6edf0be4b2267df1fa	5b8754928306a18b68	2

Solution

We start by finding the number of users sending the emails and receiving them each day.

```
SELECT
    day
    , COUNT(DISTINCT from_user) AS num_from
    , COUNT(DISTINCT to_user) AS num_to
FROM google_gmail_emails
GROUP BY 1
;
```

day	num_from	num_to
1	14	19
2	17	16
3	20	19
4	16	17
5	21	17
6	19	21
7	19	19
8	18	19

Since we need only those days where the number of users receiving emails is more than the number sending them, we apply the filter using the HAVING clause.

```
SELECT
    day
    , COUNT(DISTINCT from_user) AS num_from
    , COUNT(DISTINCT to_user) AS num_to
FROM google_gmail_emails
GROUP BY 1
HAVING COUNT(DISTINCT from_user) < COUNT(DISTINCT to_user)
;
```

day	num_from	num_to
1	14	19
4	16	17
6	19	21
8	18	19
9	17	18

Now we have the days that satisfy the criteria, we simply need to subset the main table for these days. We can do that by listing it out using the IN condition.

```
SELECT
*
FROM google_gmail_emails
WHERE day in (1, 4, 6, 8, 9)
;
```

While this solves the problem that we have, it is not a scalable solution. This requires us to run the first query, note down the output values, and then modify the query again. If we want to do this dynamically and for a larger table, we will need a way to store the output. There are multiple ways of going about doing this.

Using a Subquery

As the name suggests, we write a query within a query. Let us do this in parts. We start by extracting all the values from the subquery.

```
SELECT *
FROM (
    SELECT
```

```
        day
        , COUNT(DISTINCT from_user) AS num_from
        , COUNT(DISTINCT to_user) AS num_to
    FROM google_gmail_emails
    GROUP BY 1
    HAVING COUNT(DISTINCT from_user) < COUNT(DISTINCT to_user)
) AS SQ
;
```

We get the same output as earlier.

day	num_from	num_to
1	14	19
4	16	17
6	19	21
8	18	19
9	17	18

Instead of getting all the values, we simply get the days.

```
SELECT DAY FROM (
    SELECT
        day
        , COUNT(DISTINCT from_user) AS num_from
        , COUNT(DISTINCT to_user) AS num_to
    FROM google_gmail_emails
    GROUP BY 1
    HAVING COUNT(DISTINCT from_user) < COUNT(DISTINCT to_user)
) AS SQ
;
```

DAY
1
4
6
8
9

Now we use this sub-query and extract only the relevant days from the main dataset

```
SELECT
*
FROM google_gmail_emails
WHERE day in (
    SELECT
        day
    FROM google_gmail_emails
    GROUP BY 1
    HAVING COUNT(DISTINCT from_user) < COUNT(DISTINCT to_user)
)
;
```

We now have the required output.

Using a CTE

Subqueries are very helpful, but as you might have observed, they become very unwieldy for large subqueries, and the code becomes very difficult to read and debug. We can get the same result with a Common Table Expression (CTE).

```sql
WITH days AS
(
    SELECT
        day
        , COUNT(DISTINCT from_user) AS num_from
        , COUNT(DISTINCT to_user) AS num_to
    FROM google_gmail_emails
    GROUP BY 1
    HAVING COUNT(DISTINCT from_user) < COUNT(DISTINCT to_user)
)
SELECT
G.*
FROM google_gmail_emails AS G
WHERE day in (SELECT day FROM days)
;
```

CTEs work in the same manner a sub-queries. Think of CTEs as SQL tables created on the fly. As we shall see later, we can also JOIN these subqueries and CTEs. Another often used way of using multiple queries is to create a TEMP TABLE.

TEMP Table

As the name suggests a TEMP table is a temporary table that is created in memory. The table is automatically deleted when the SQL session ends.

Note this will return an error on the StrataScratch platform as only Admins have the privilege of creating TEMP Tables.

```sql
CREATE TEMP TABLE days AS
SELECT
    day
    , COUNT(DISTINCT from_user) AS num_from
    , COUNT(DISTINCT to_user) AS num_to
FROM google_gmail_emails
GROUP BY 1
HAVING COUNT(DISTINCT from_user) < COUNT(DISTINCT to_user)
;

SELECT
G.*
```

```
FROM google_gmail_emails AS G
WHERE day in (SELECT day FROM days)
;
```

One of the additional advantages of a TEMP table is that it can be accessed only by the current session. Therefore there is no problem with conflicts with other parallel sessions. This is particularly helpful when a lot of people are querying the same database concurrently. It prevents other sessions from creating a lock on the table. Another use case for a TEMP table is to store a large amount of data without having to worry about having to delete that table later on when the session ends. One can test large Data Pipelines with TEMP tables without having to run CTEs or subquery each time.

Working with multiple tables

In the real world, it is rare to find all the data present in the same table. It is inefficient and expensive to store all the information in a single table as there might be a lot of redundant information. Further, the complexities of most real-world scenarios mean that it is better to store the information in multiple tables. Ensuring that the database is designed efficiently to reduce redundancies is called normalization.

Normalization

Database normalization is the transformation of complex user views and data stores into a set of smaller, stable data structures. In addition to being simpler and more stable, normalized data structures are more easily maintained than other data structures.

Steps

1st Normal Form (1NF): The first stage of the process includes removing all repeating groups and identifying the primary key. To do so, the relationship needs to be broken up into two or more relations. At this point, the relations may already be of the third normal form, but it is likely more steps will be needed to transform the relations into the third normal form.

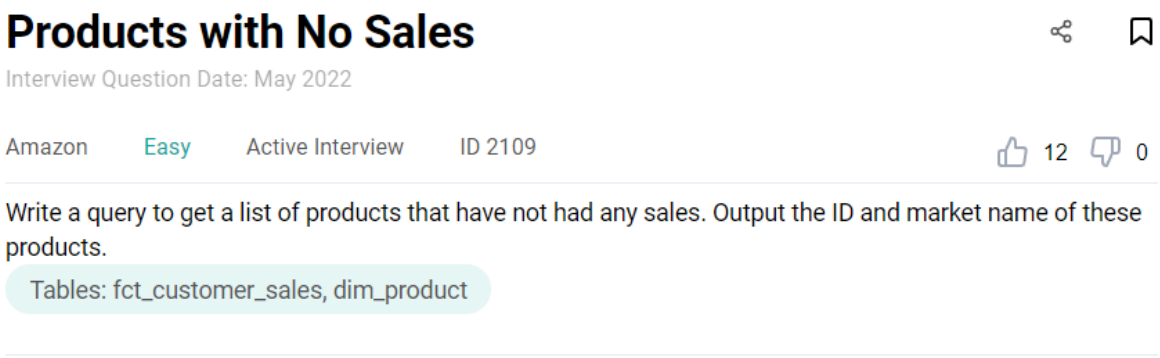
2nd Normal Form (2NF): The second step ensures that all non-key attributes are fully dependent on the primary key. All partial dependencies are removed and placed in another relation.

3rd Normal Form (3NF): The third step removes any transitive dependencies. A transitive dependency is one in which non-key attributes are dependent on other non-key attributes.

JOINS

To get information from multiple tables, we can use JOINs. If you are aware of the LOOKUP function in spreadsheets, JOINs work in a similar manner. Unlike a LOOKUP function, we can have multiple different type of JOINs. To illustrate the JOINs let us take a simple problem and understand how JOINs work.

MySQL Interview Question #4: Products with No Sales



Link to the question:
https://platform.stratascratch.com/coding/2109-products-with-no-sales

This problem uses the `fct_customer_sales` and `dim_product` tables with the following fields.

fct_customer_sales Preview

cust_id:	varchar
prod_sku_id:	varchar
order_date:	datetime
order_value:	int
order_id:	varchar

dim_product Preview

prod_sku_id:	varchar
prod_sku_name:	varchar
prod_brand:	varchar
market_name:	varchar

The data in the `fct_customer_sales` is presented thus.

cust_id	prod_sku_id	order_date	order_value	order_id
C274	P474	2021-06-28	1500	O110
C285	P472	2021-06-28	899	O118
C282	P487	2021-06-30	500	O125
C282	P476	2021-07-02	999	O146
C284	P487	2021-07-07	500	O149
C285	P478	2021-07-12	700	O150
C287	P489	2021-07-13	189	O151
C284	P482	2021-07-15	725	O156
C281	P482	2021-07-19	725	O164
C282	P480	2021-07-22	300	O172

And the data in `dim_product` is presented in the following manner.

prod_sku_id	prod_sku_name	prod_brand	market_name
P472	iphone-13	Apple	Apple IPhone 13
P473	iphone-13-promax	Apple	Apply IPhone 13 Pro Max
P474	macbook-pro-13	Apple	Apple Macbook Pro 13"
P475	macbook-air-13	Apple	Apple Makbook Air 13"
P476	ipad	Apple	Apple IPad
P477	ipad-pro	Apple	Apple IPad Pro
P478	galaxy-s21	Samsung	Samsung Galaxy S21
P479	galaxy-s22plus	Samsung	Samsung Galaxy S22+
P480	galaxy-watch4	Samsung	Samsung Galaxy Watch4
P481	galaxy-tab-a	Samsung	Samsung Galaxy Tab A

INNER JOIN

An inner join takes the data present in both the tables. To join the two tables, we need glue or a key. In simple terms, this is an identifier that is present in both tables. As you would have guessed, the common identifier here is the `prod_sku_id`.

```
SELECT
    fcs.*,
    dp.*
FROM fct_customer_sales AS fcs
INNER JOIN dim_product AS dp
    ON fcs.prod_sku_id = dp.prod_sku_id
;
```

cust_id	prod_sku_id	order_date	order_value	order_id	prod_sku_id	prod_sku_name	prod_brand	market_name
C274	P474	2021-06-28	1500	O110	P474	macbook-pro-13	Apple	Apple Macbook Pro 13"
C285	P472	2021-06-28	899	O118	P472	iphone-13	Apple	Apple IPhone 13
C282	P487	2021-06-30	500	O125	P487	max	GoPro	GoPro Max
C282	P476	2021-07-02	999	O146	P476	ipad	Apple	Apple IPad
C284	P487	2021-07-07	500	O149	P487	max	GoPro	GoPro Max
C285	P478	2021-07-12	700	O150	P478	galaxy-s21	Samsung	Samsung Galaxy S21
C287	P489	2021-07-13	189	O151	P489	tuner-xl	JBL	JBL Tuner XL
C284	P482	2021-07-15	725	O156	P482	galaxy-tab-s8	Samsung	Samsung Galaxy Tab 8
C281	P482	2021-07-19	725	O164	P482	galaxy-tab-s8	Samsung	Samsung Galaxy Tab 8
C282	P480	2021-07-22	300	O172	P480	galaxy-watch4	Samsung	Samsung Galaxy Watch4

As you can see, we get the common rows from both the tables.

LEFT JOIN

The most commonly used join in practice is the LEFT JOIN. The process is similar to the INNER JOIN except, we keep the left table as the base table and keep adding information from other tables to this table. In case there is no information relevant to the key in the other table, NULL rows are returned. In our case, since the product table is the base table, we set to keep the dim_product as the LEFT table. The joining process is essentially the same. However, in this case, the order of specifying the tables matter since we need to set the base (left) table.

cust_id	prod_sku_id	order_date	order_value	order_id	prod_sku_id	prod_sku_name	prod_brand	market_name
C277	P472	2021-11-29	899	O776	P472	iphone-13	Apple	Apple IPhone 13
C284	P472	2021-10-25	899	O567	P472	iphone-13	Apple	Apple IPhone 13
C278	P472	2021-10-11	899	O487	P472	iphone-13	Apple	Apple IPhone 13
C273	P472	2021-10-06	899	O430	P472	iphone-13	Apple	Apple IPhone 13
C285	P472	2021-10-01	899	O398	P472	iphone-13	Apple	Apple IPhone 13
C285	P472	2021-06-28	899	O118	P472	iphone-13	Apple	Apple IPhone 13
					P473	iphone-13-promax	Apple	Apply IPhone 13 Pro Max
C275	P474	2021-12-01	1500	O796	P474	macbook-pro-13	Apple	Apple Macbook Pro 13"
C286	P474	2021-11-25	1500	O747	P474	macbook-pro-13	Apple	Apple Macbook Pro 13"
C280	P474	2020-08-19	1500	O290	P474	macbook-pro-13	Apple	Apple Macbook Pro 13"

As you can observe, we get NULLs for some rows. Now, these are the rows that are relevant to the solution as we need to get the products that did not have any sales (or matches). We can get the solution by simply subsetting the final output.

```
SELECT
    fcs.*,
    dp.*
FROM dim_product AS dp
LEFT JOIN fct_customer_sales AS fcs
    ON fcs.prod_sku_id = dp.prod_sku_id
WHERE fcs.prod_sku_id IS NULL
;
```

RIGHT JOIN

The mirror opposite of the LEFT JOIN is the RIGHT join. As you would have guessed, this reverses the joining process. The second (right) table is considered the base table, and the information is appended from the left table. In case there is no information available from the RIGHT table, we will get a NULL row. We can rewrite the solution using the RIGHT JOIN in the following manner.

```
SELECT
    fcs.*,
    dp.*
FROM fct_customer_sales AS fcs
```

```
RIGHT JOIN dim_product AS dp
    ON fcs.prod_sku_id = dp.prod_sku_id
WHERE fcs.prod_sku_id IS NULL
;
```

The choice of LEFT JOIN and RIGHT JOIN, while arbitrary, can sometimes lead to raging online debates, as evidenced here. Without taking any sides, it is customary to use LEFT JOINs though there are scenarios when RIGHT JOINs might make sense. To assuage the religious, just change the order of the tables!!

UNION

One overlooked aspect of joining tables in MySQL is the UNION. Unlike a JOIN which adds columns (fields), UNIONs stack rows. Let us take a simple example, suppose we want to stack names. We can simply do something like this.

```
SELECT 'VIVEK' AS name
UNION
SELECT 'NATE' AS name
UNION
SELECT 'TIHOMIR' AS name
;
```

name
VIVEK
NATE
TIHOMIR

UNIONs, by defaul,t eliminate duplicate entries.

```
SELECT 'VIVEK' AS name
```

```
UNION
SELECT 'NATE' AS name
UNION
SELECT 'TIHOMIR' AS name
UNION
SELECT 'VIVEK' AS name
;
```

name
VIVEK
NATE
TIHOMIR

If you need duplicates, use UNION ALL

```
SELECT 'VIVEK' AS name
UNION
SELECT 'NATE' AS name
UNION
SELECT 'TIHOMIR' AS name
UNION ALL
SELECT 'VIVEK' AS name
;
```

name
VIVEK
NATE
TIHOMIR
VIVEK

You can read more about JOINs and UNIONs in this comprehensive article about SQL JOINs here.

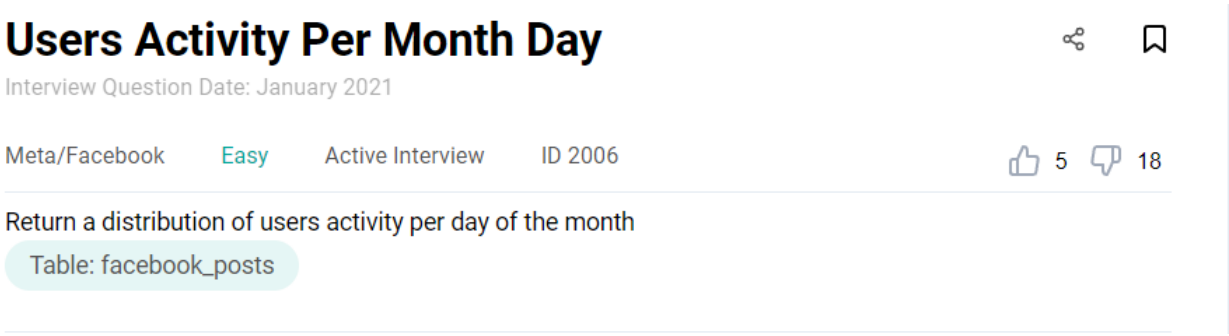
Built-in Functions in MySQL

Let us now use some of the built-in functions present in MySQL. These functions can be used to perform numerous common tasks like Date and Time manipulation, Text manipulation, Mathematics, et al. Let us look at a few common ones.

Working with Date time functions

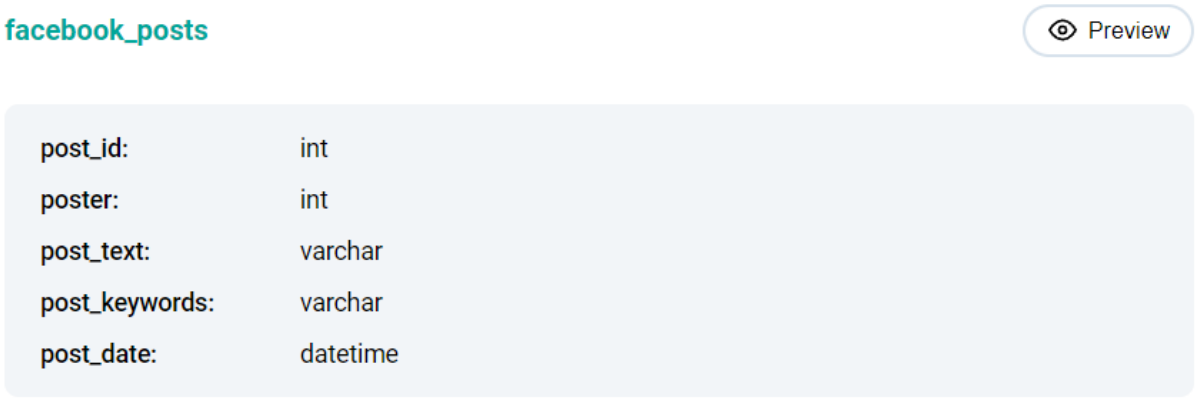
Date and time functions are critical in our day to day life. We encounter them every moment of our lives. Date time manipulation is a very commonly asked problem scenario in SQL interviews. Let us look at a few problems.

MySQL Interview Question #5: Users Activity Per Month Day



Link to the question:
https://platform.stratascratch.com/coding/2006-users-activity-per-month-day

The problem uses the `facebook_posts` table with the following fields.



The data is presented in the following manner.

post_id	poster	post_text	post_keywords	post_date
0	2	The Lakers game from last night was great.	[basketball,lakers,nba]	2019-01-01
1	1	Lebron James is top class.	[basketball,lebron_james,nba]	2019-01-02
2	2	Asparagus tastes OK.	[asparagus,food]	2019-01-01
3	1	Spaghetti is an Italian food.	[spaghetti,food]	2019-01-02
4	3	User 3 is not sharing interests	[#spam#]	2019-01-01
5	3	User 3 posts SPAM content a lot	[#spam#]	2019-01-02

To solve this problem, we need to extract the day of the month from the `post_date`. We can simply use the DAYOFMONTH() or DAY() function in MySQL to do this.

```
SELECT
    *
    , DAY(post_date) AS day_of_month
FROM facebook_posts;
```

post_id	poster	post_text	post_keywords	post_date	day_of_month
0	2	The Lakers game from last night was great.	[basketball,lakers,nba]	2019-01-01	1
1	1	Lebron James is top class.	[basketball,lebron_james,nba]	2019-01-02	2
2	2	Asparagus tastes OK.	[asparagus,food]	2019-01-01	1
3	1	Spaghetti is an Italian food.	[spaghetti,food]	2019-01-02	2
4	3	User 3 is not sharing interests	[#spam#]	2019-01-01	1

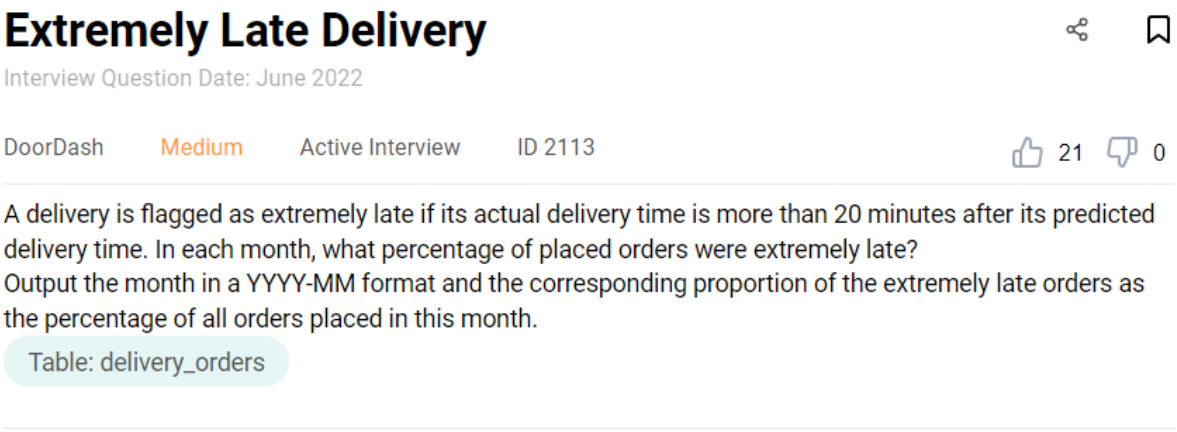
Now we simply aggregate the number of users by each day.

```
SELECT
    DAY(post_date) AS day_of_month
```

```
    , COUNT(*)
FROM facebook_posts
GROUP BY 1
;
```

Let us try a slightly difficult one that uses more datetime functions.

MySQL Interview Question #6: Extremely Late Delivery



Extremely Late Delivery

Interview Question Date: June 2022

DoorDash Medium Active Interview ID 2113 21 0

A delivery is flagged as extremely late if its actual delivery time is more than 20 minutes after its predicted delivery time. In each month, what percentage of placed orders were extremely late?
Output the month in a YYYY-MM format and the corresponding proportion of the extremely late orders as the percentage of all orders placed in this month.

Table: delivery_orders

Link to the question: https://platform.stratascratch.com/coding/2113-extremely-late-delivery

This problem uses the `delivery_orders` table with the following columns.

delivery_orders

Preview

delivery_id:	varchar
order_placed_time:	datetime
predicted_delivery_time:	datetime
actual_delivery_time:	datetime
delivery_rating:	int
dasher_id:	varchar
restaurant_id:	varchar
consumer_id:	varchar

The data is presented in the following manner.

delivery_id	order_placed_time	predicted_delivery_time	actual_delivery_time	delivery_rating	dasher_id	restaurant_id	consumer_id
O2132	2021-11-17 04:45:33	2021-11-17 05:37:33	2021-11-17 05:58:33	4	D239	R633	C1001
O2152	2021-12-09 19:09:43	2021-12-09 19:41:43	2021-12-09 19:41:43	3	D238	R635	C1010
O2158	2022-01-04 02:31:19	2022-01-04 02:56:19	2022-01-04 03:21:19	4	D239	R634	C1010
O2173	2022-02-09 00:45:22	2022-02-09 01:19:22	2022-02-09 01:33:22	0	D239	R633	C1038
O2145	2021-12-04 17:20:27	2021-12-04 18:04:27	2021-12-04 18:31:27	1	D239	R634	C1042
O2144	2021-12-04 12:50:49	2021-12-04 13:34:49	2021-12-04 13:47:49	1	D238	R637	C1059
O2172	2022-02-08 14:53:35	2022-02-08 15:13:35	2022-02-08 15:24:35	5	D237	R636	C1059
O2170	2022-02-04 03:08:23	2022-02-04 03:28:23	2022-02-04 03:26:23	1	D239	R636	C1083
O2150	2021-12-08 10:47:50	2021-12-08 11:27:50	2021-12-08 11:25:50	3	D240	R631	C1130

In the problem, we need to perform two datetime manipulations

- Extract the Year and Month from the `order_placed_time`
- Find the difference in minutes between the `actual_delivery_time` and `predicted_delivery_time`

Let us start by extracting the Year and Month. We can use the DATE_FORMAT() function in MySQL. This function converts a given date time value to a string format. We only keep the fields relevant to the solution.

```
SELECT
    DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
    , predicted_delivery_time
    , actual_delivery_time
FROM delivery_orders
;
```

YYYYMM	predicted_delivery_time	actual_delivery_time
2021-11	2021-11-17 05:37:33	2021-11-17 05:58:33
2021-12	2021-12-09 19:41:43	2021-12-09 19:41:43
2022-01	2022-01-04 02:56:19	2022-01-04 03:21:19
2022-02	2022-02-09 01:19:22	2022-02-09 01:33:22
2021-12	2021-12-04 18:04:27	2021-12-04 18:31:27
2021-12	2021-12-04 13:34:49	2021-12-04 13:47:49
2022-02	2022-02-08 15:13:35	2022-02-08 15:24:35
2022-02	2022-02-04 03:28:23	2022-02-04 03:26:23
2021-12	2021-12-08 11:27:50	2021-12-08 11:25:50

As required, we now have the output column to aggregate by. Now let us find the difference between the actual delivery time and the predicted delivery time. To do this, we have multiple options. We can use the TIMEDIFF() function or the TIMESTAMPDIFF() function. The TIMEDIFF() function returns the difference as a time interval, which we can convert to minutes. Or we can simply combine the two steps using the TIMESTAMPDIFF() function. Let us see how this works in practice.

```
SELECT
    DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
    , predicted_delivery_time
    , actual_delivery_time
    , TIMESTAMPDIFF(MINUTE, predicted_delivery_time, actual_delivery_time)
AS time_difference
FROM delivery_orders
;
```

YYYYMM	predicted_delivery_time	actual_delivery_time	time_difference
2021-11	2021-11-17 05:37:33	2021-11-17 05:58:33	21
2021-12	2021-12-09 19:41:43	2021-12-09 19:41:43	0
2022-01	2022-01-04 02:56:19	2022-01-04 03:21:19	25
2022-02	2022-02-09 01:19:22	2022-02-09 01:33:22	14
2021-12	2021-12-04 18:04:27	2021-12-04 18:31:27	27
2021-12	2021-12-04 13:34:49	2021-12-04 13:47:49	13
2022-02	2022-02-08 15:13:35	2022-02-08 15:24:35	11
2022-02	2022-02-04 03:28:23	2022-02-04 03:26:23	-2
2021-12	2021-12-08 11:27:50	2021-12-08 11:25:50	-2
2022-02	2022-02-26 16:12:49	2022-02-26 16:24:49	12

Now we can simply perform two aggregations and find the percentage of delayed orders.

```
WITH ALL_ORDERS AS (
    SELECT
        DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
        , COUNT(*) AS NUM_ORDERS
    FROM delivery_orders
```

```
    GROUP BY 1
), DELAYED_ORDERS AS (
    SELECT
        DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
        , COUNT(*) AS DELAYED_ORDERS
    FROM delivery_orders
    WHERE TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20
    GROUP BY 1
)
SELECT
    ao.YYYYMM
    , ao.NUM_ORDERS
    , do.DELAYED_ORDERS
FROM ALL_ORDERS AS ao
    LEFT JOIN DELAYED_ORDERS as do
        ON ao.YYYYMM = do.YYYYMM
;
```

YYYYMM	NUM_ORDERS	DELAYED_ORDERS
2021-11	13	4
2021-12	16	2
2022-01	11	4
2022-02	10	1

Finally, we calculate the delayed orders as a percentage of the total orders.

```
WITH ALL_ORDERS AS (
    SELECT
        DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
        , COUNT(*) AS NUM_ORDERS
    FROM delivery_orders
    GROUP BY 1
), DELAYED_ORDERS AS (
    SELECT
```

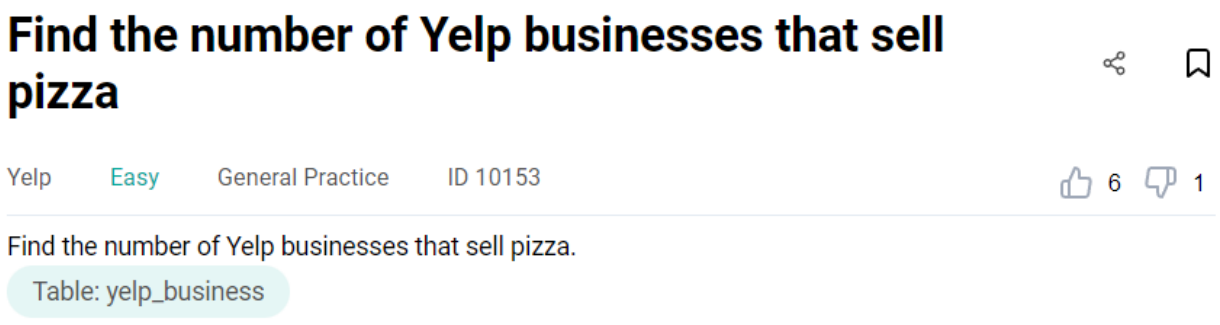
```
        DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
        , COUNT(*) AS DELAYED_ORDERS
    FROM delivery_orders
    WHERE TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20
    GROUP BY 1
)
SELECT
    ao.YYYYMM
    , ao.NUM_ORDERS
    , do.DELAYED_ORDERS
    , do.DELAYED_ORDERS / ao.NUM_ORDERS * 100
FROM ALL_ORDERS AS ao
    LEFT JOIN DELAYED_ORDERS as do
        ON ao.YYYYMM = do.YYYYMM
;
```

We have a detailed article on various date time manipulation with SQL. You can read about it here "*SQL Scenario Based Interview Questions*".

Working with Text Functions

Text manipulation is another important aspect of data analysis. As with other flavors of SQL, MySQL contains a variety of text manipulation functions. Let us try out a few of these.

MySQL Interview Question #7: Find the number of Yelp businesses that sell pizza



Link to the question: https://platform.stratascratch.com/coding/10153-find-the-number-of-yelp-businesses-that-sell-pizza

This problem uses the `yelp_business` table with the following columns.

yelp_business Preview

business_id:	varchar
name:	varchar
neighborhood:	varchar
address:	varchar
city:	varchar
state:	varchar
postal_code:	varchar
latitude:	float
longitude:	float
stars:	float
review_count:	int
is_open:	int
categories:	varchar

The key column in this dataset is `categories`

```
SELECT categories FROM yelp_business;
```

categories
Auto Detailing;Automotive
Personal Chefs;Food;Gluten-Free;Food Delivery Services;Event Planning & Services;Restaurants
Dry Cleaning & Laundry;Laundry Services;Local Services;Dry Cleaning
Wedding Planning;Caterers;Event Planning & Services;Venues & Event Spaces
Notaries;Printing Services;Local Services;Shipping Centers
Home Services;Television Service Providers;Professional Services;Internet Service Providers;Utilities
Bars;Restaurants;Pubs;British;Nightlife
Hotels & Travel;Car Rental
Fast Food;Sandwiches;Restaurants

To solve this problem, we need to identify the businesses that sell pizzas. To do that, we need to search for the word Pizza in the categories field. There are multiple ways of doing this. Let us look at a couple of them.

Using the LIKE operator.

The LIKE operator in MySQL can be used for simple string pattern matching. Since the word can be preceded or succeeded by any number of characters, we prefix and suffix the search string with the % symbol. Further, to ensure that case sensitivity does not affect the results, we convert the entire string to lowercase.

```
SELECT count(*)
FROM yelp_business
WHERE lower(categories) LIKE '%pizza%'
;
```

Note:

- We can alternatively use the UPPER() function and search using `LIKE %PIZZA%`. This will give the same results.
- If you are using Postgres instead of MySQL, the above will work. Alternatively, you can use the ILIKE operator for case-insensitive matching.

Using the LOCATE function

The LOCATE function searches for a particular text within the entire text. Since we need to make a case-insensitive search, we convert the text to lower or upper case. Let us try this with the UPPER() function here. In case of a successful match, the function returns the position of the substring. Otherwise, it returns 0.

```
SELECT count(*)
FROM yelp_business
WHERE LOCATE('PIZZA', UPPER(categories)) > 0
;
```

Let us try another problem.

MySQL Interview Question #8: Find all workers whose first name contains 6 letters and also ends with the letter 'h'

Find all workers whose first name contains 6 letters and also ends with the letter 'h'

Amazon Easy General Practice ID 9842

Find all workers whose first name contains 6 letters and also ends with the letter 'h'.

Table: worker

Link to the question: https://platform.stratascratch.com/coding/9842-find-all-workers-whose-first-name-contains-6-letters-and-also-ends-with-the-letter-h

This problem uses the same `worker` table that we had used earlier.

worker

worker_id:	int
first_name:	varchar
last_name:	varchar
salary:	int
joining_date:	datetime
department:	varchar

The data in the table looks like this.

worker_id	first_name	last_name	salary	joining_date	department
1	Monika	Arora	100000	2014-02-20 09:00:00	HR
2	Niharika	Verma	80000	2014-06-11 09:00:00	Admin
3	Vishal	Singhal	300000	2014-02-20 09:00:00	HR
4	Amitah	Singh	500000	2014-02-20 09:00:00	Admin
5	Vivek	Bhati	500000	2014-06-11 09:00:00	Admin
6	Vipul	Diwan	200000	2014-06-11 09:00:00	Account

There are two key parts to this problem.

- We need to check if the first name has a length of 6
- We also need to ascertain if the last alphabet in the first name is 'h'

To check for the length, we use the LENGTH() function.

```
SELECT
first_name
, LENGTH(first_name) AS name_length
FROM worker;
```

first_name	name_length
Monika	6
Niharika	8
Vishal	6
Amitah	6
Vivek	5
Vipul	5
Satish	6
Geetika	7

We then check if the last character is ‘h’. To do this, we use the LIKE operator, but we need to check for h only at the end of the string. Therefore, we use the % only in the prefix.

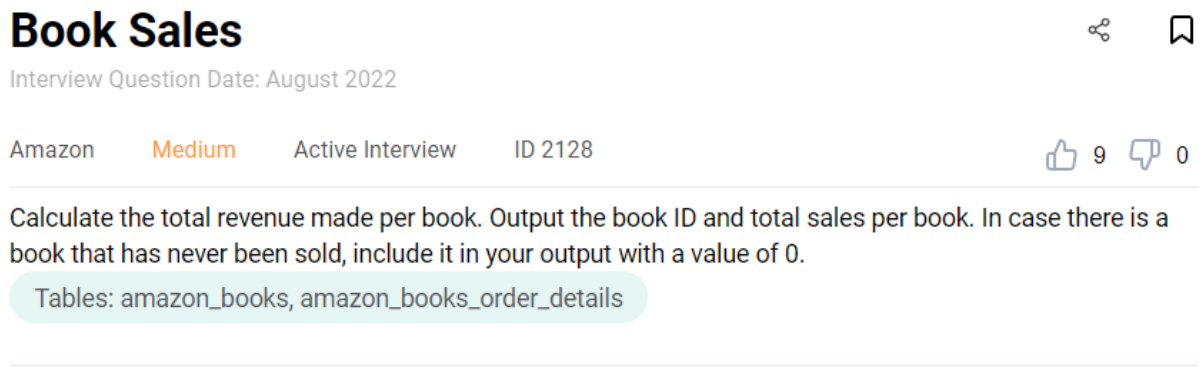
```
SELECT
*
FROM worker
WHERE LENGTH(first_name) = 6 AND UPPER(first_name) LIKE '%H'
;
```

Working with NULL Values

Another common real-life scenario when working with databases is working with NULL or missing values. Not all NULLs are equal. One needs to understand what the null value represents - a missing value or zero. Imputing missing values is a whole discussion in Data

Science. From a technical perspective, one needs to be careful while working with NULLs, as they can result in unpredictable results. Let us see how we can work with NULLs or missing values in MySQL.

MySQL Interview Question #9: Book Sales



Link to the question: https://platform.stratascratch.com/coding/2128-book-sales

This problem uses the `amazon_books` and the `amazon_books_order_details` tables with the following fields.

amazon_books Preview

book_id:	varchar
book_title:	varchar
unit_price:	int

amazon_books_order_details Preview

order_details_id:	varchar
order_id:	varchar
book_id:	varchar
quantity:	int

The data in the `amazon_books` table is presented thus.

book_id	book_title	unit_price
B001	The Hunger Games	25
B002	The Outsiders	50
B003	To Kill a Mockingbird	100
B004	Pride and Prejudice	20
B005	Twilight	30
B006	The Book Thief	50
B007	Animal Farm	40
B008	The Chronicles of Narnia	30

The data in the `amazon_books_order_details` table looks like this.

order_details_id	order_id	book_id	quantity
OD101	O1001	B001	1
OD102	O1001	B009	1
OD103	O1002	B012	2
OD104	O1002	B006	1
OD105	O1002	B019	2
OD106	O1003	B017	1
OD107	O1004	B020	2
OD108	O1005	B020	2

One needs to be a bit careful while solving this problem. It is specified in the problem that even if there is no sale for the book, we need to output the book details. Therefore our base table will be the `amazon_books` table. To this, we append the information.

```
SELECT
ab.*
, abo.*
FROM amazon_books AS ab
    LEFT JOIN amazon_books_order_details AS abo
        ON ab.book_id = abo.book_id
;
```

book_id	book_title	unit_price	order_details_id	order_id	book_id	quantity
B001	The Hunger Games	25	OD120	O1013	B001	2
B001	The Hunger Games	25	OD110	O1006	B001	1
B001	The Hunger Games	25	OD101	O1001	B001	1
B002	The Outsiders	50	OD111	O1007	B002	2
B003	To Kill a Mockingbird	100	OD117	O1011	B003	1
B003	To Kill a Mockingbird	100	OD112	O1007	B003	1
B004	Pride and Prejudice	20				
B005	Twilight	30	OD109	O1006	B005	1
B006	The Book Thief	50	OD123	O1014	B006	1
B006	The Book Thief	50	OD118	O1012	B006	3

We are using LEFT JOIN, as we want to retain the books that did not have any sales as well. Now we proceed to find the total sales of the book by multiplying the unit price by the aggregate number of books sold.

```
SELECT
    ab.book_id
    , SUM(ab.unit_price * abo.quantity) AS total_sales
FROM amazon_books AS ab
    LEFT JOIN amazon_books_order_details AS abo
        ON ab.book_id = abo.book_id
GROUP BY 1
;
```

book_id	total_sales
B001	100
B002	100
B003	200
B004	
B005	30
B006	300
B007	80
B008	
B009	100
B010	50

As you can see, the books with no sales have NULL or empty values. We need to replace these NULLs with 0. There are multiple ways for doing this.

IFNULL()

The IFNULL function checks if a particular value is NULL and replaces it with an alternative value.

```
SELECT
    ab.book_id
    , SUM(ab.unit_price * abo.quantity) AS total_sales
    , IFNULL(SUM(ab.unit_price * abo.quantity), 0) AS adjusted_sales
```

```
FROM amazon_books AS ab
    LEFT JOIN amazon_books_order_details AS abo
        ON ab.book_id = abo.book_id
GROUP BY 1
;
```

book_id	total_sales	adjusted_sales
B001	100	100
B002	100	100
B003	200	200
B004		0
B005	30	30
B006	300	300
B007	80	80
B008		0
B009	100	100
B010	50	50
B011	180	180

While we did not need to create an additional column, we used it to show the effect of the ISNULL function.

COALESCE

An alternate way of solving this is to use the COALESCE function. It is similar to the IFNULL function, but it works across multiple cases. The syntax is

```
COALESCE(value1, value2, ... valuen, default value)
```

COALESCE will check each of the values value1, value2, … valuen and find the first non-NULL value. For example, if you try this

```
SELECT COALESCE(NULL, NULL, 'Alpha');
```

it will return Alpha. As will the following COALESCE statement.

```
SELECT COALESCE(NULL, NULL, 'Alpha', 'Beta', NULL, 'Gamma');
```

We can solve the given problem with COALESCE in the following manner. This will give the same result as with the IFNULL function.

```
SELECT
    ab.book_id
    , SUM(ab.unit_price * abo.quantity) AS total_sales
    , COALESCE(SUM(ab.unit_price * abo.quantity), 0) AS adjusted_sales
FROM amazon_books AS ab
    LEFT JOIN amazon_books_order_details AS abo
        ON ab.book_id = abo.book_id
GROUP BY 1
;
```

Let us try a slightly tricky problem.

MySQL Interview Question #10: Flags per Video

Flags per Video

Interview Question Date: April 2022

Google Medium Active Interview ID 2102 19 2

For each video, find how many unique users flagged it. A unique user can be identified using the combination of their first name and last name. Do not consider rows in which there is no flag ID.

Table: user_flags

Link to the question: https://platform.stratascratch.com/coding/2102-flags-per-video

This problem uses the `user_flags` table with the following fields.

user_flags Preview

user_firstname:	varchar
user_lastname:	varchar
video_id:	varchar
flag_id:	varchar

The data is presented thus.

user_firstname	user_lastname	video_id	flag_id
Richard	Hasson	y6120QOlsfU	0cazx3
Mark	May	Ct6BUPvE2sM	1cn76u
Gina	Korman	dQw4w9WgXcQ	1i43zk
Mark	May	Ct6BUPvE2sM	1n0vef
Mark	May	jNQXAC9IVRw	1sv6ib
Gina	Korman	dQw4w9WgXcQ	20xekb

Prima facie, this looks pretty straightforward. We need to create a unique user by concatenating the `user_firstname` and the `user_lastname` fields. To concatenate two or more values, we use the CONCAT() function. One can also use the || operator, though it is going to be deprecated soon.

```
SELECT
*
, CONCAT(user_firstname,  user_lastname) AS user_id
FROM user_flags;
```

user_firstname	user_lastname	video_id	flag_id	user_id
Richard	Hasson	y6120QOlsfU	0cazx3	RichardHasson
Mark	May	Ct6BUPvE2sM	1cn76u	MarkMay
Gina	Korman	dQw4w9WgXcQ	1i43zk	GinaKorman
Mark	May	Ct6BUPvE2sM	1n0vef	MarkMay
Mark	May	jNQXAC9IVRw	1sv6ib	MarkMay
Gina	Korman	dQw4w9WgXcQ	20xekb	GinaKorman
Mark	May	5qap5aO4i9A	4cvwuv	MarkMay
Daniel	Bell	5qap5aO4i9A	4sd6dv	DanielBell
Richard	Hasson	y6120QOlsfU	6jjkvn	RichardHasson
Pauline	Wilks	jNQXAC9IVRw	7ks264	PaulineWilks
Courtney		dQw4w9WgXcQ		
Helen	Hearn	dQw4w9WgXcQ	8946nx	HelenHearn
Mark	Johnson	y6120QOlsfU	8wwg0l	MarkJohnson
Richard	Hasson	dQw4w9WgXcQ	arydfd	RichardHasson
Gina	Korman			GinaKorman
Mark	Johnson	y6120QOlsfU	bl40qw	MarkJohnson

This method runs into a problem when either the first or the last name is missing. Concatenating a field with a NULL value results in a NULL value. To overcome this, we use the COALESCE function and set a default value in case we encounter a NULL in the first or the last name fields. Further, we also need to eliminate those results where the flag_id is NULL (the blue row above). We can do this by using a WHERE condition.

```
SELECT
*
, CONCAT(COALESCE(user_firstname, '$$'),  COALESCE(user_lastname, '$$')) AS
user_id
FROM user_flags
WHERE flag_id IS NOT NULL
ORDER BY 1
;
```

user_firstname	user_lastname	video_id	flag_id	user_id
	Johnson	y6120QOlsfU	iey5vi	$$Johnson
	Lopez	dQw4w9WgXcQ	hucyzx	$$Lopez
Daniel	Bell	5qap5aO4i9A	xciyse	DanielBell
Daniel	Bell	5qap5aO4i9A	4sd6dv	DanielBell
Evelyn	Johnson	dQw4w9WgXcQ	xvhk6d	EvelynJohnson
Gina	Korman	dQw4w9WgXcQ	1i43zk	GinaKorman
Gina	Korman	dQw4w9WgXcQ	20xekb	GinaKorman
Helen	Hearn	dQw4w9WgXcQ	8946nx	HelenHearn

Note do not use logical operators like >, <, =, etc. with NULL. They will result in NULL.

We finish off the problem by counting the distinct users for each video.

```
SELECT
video_id
,       COUNT(DISTINCT       CONCAT(COALESCE(user_firstname,       '$$'),
COALESCE(user_lastname, '$$'))) AS num_users
FROM user_flags
WHERE flag_id IS NOT NULL
GROUP BY 1
;
```

Advanced MySQL Interview Questions

Now that we have a better understanding of MySQL, let us try to learn a few advanced tricks to help solve the MySQL interview questions faster.

Using Conditional Operations

There are occasions when we want to perform calculations based on conditional values on a particular row. We have the `WHERE` clause that helps us subset the entire table, it is not always the most efficient way of solving a problem. Let us revisit a problem that we encountered earlier.

MySQL Interview Question #11: Extremely Late Delivery

Extremely Late Delivery

Interview Question Date: June 2022

DoorDash Medium Active Interview ID 2113 21 0

A delivery is flagged as extremely late if its actual delivery time is more than 20 minutes after its predicted delivery time. In each month, what percentage of placed orders were extremely late?
Output the month in a YYYY-MM format and the corresponding proportion of the extremely late orders as the percentage of all orders placed in this month.

Table: delivery_orders

Link to the question: https://platform.stratascratch.com/coding/2113-extremely-late-delivery

The problem uses the `delivery_orders` table with the following fields.

delivery_orders Preview

delivery_id:	varchar
order_placed_time:	datetime
predicted_delivery_time:	datetime
actual_delivery_time:	datetime
delivery_rating:	int
dasher_id:	varchar
restaurant_id:	varchar
consumer_id:	varchar

The data is arranged in the following manner.

delivery_id	order_placed_time	predicted_delivery_time	actual_delivery_time	delivery_rating	dasher_id	restaurant_id	consumer_id
O2132	2021-11-17 04:45:33	2021-11-17 05:37:33	2021-11-17 05:58:33	4	D239	R633	C1001
O2152	2021-12-09 19:09:43	2021-12-09 19:41:43	2021-12-09 19:41:43	3	D238	R635	C1010
O2158	2022-01-04 02:31:19	2022-01-04 02:56:19	2022-01-04 03:21:19	4	D239	R634	C1010
O2173	2022-02-09 00:45:22	2022-02-09 01:19:22	2022-02-09 01:33:22	0	D239	R633	C1038
O2145	2021-12-04 17:20:27	2021-12-04 18:04:27	2021-12-04 18:31:27	1	D239	R634	C1042
O2144	2021-12-04 12:50:49	2021-12-04 13:34:49	2021-12-04 13:47:49	1	D238	R637	C1059
O2172	2022-02-08 14:53:35	2022-02-08 15:13:35	2022-02-08 15:24:35	5	D237	R636	C1059
O2170	2022-02-04 03:08:23	2022-02-04 03:28:23	2022-02-04 03:26:23	1	D239	R636	C1083
O2150	2021-12-08 10:47:50	2021-12-08 11:27:50	2021-12-08 11:25:50	3	D240	R631	C1130

We used two CTEs and then merged them to get the final output.

```
WITH ALL_ORDERS AS (
    SELECT
        DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
        , COUNT(*) AS NUM_ORDERS
    FROM delivery_orders
    GROUP BY 1
), DELAYED_ORDERS AS (
    SELECT
        DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
        , COUNT(*) AS DELAYED_ORDERS
    FROM delivery_orders
    WHERE TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20
    GROUP BY 1
)
SELECT
    ao.YYYYMM
    , ao.NUM_ORDERS
    , do.DELAYED_ORDERS
    , do.DELAYED_ORDERS / ao.NUM_ORDERS * 100
FROM ALL_ORDERS AS ao
    LEFT JOIN DELAYED_ORDERS as do
        ON ao.YYYYMM = do.YYYYMM
;
```

We did this because we wanted the overall number of orders as well as the delayed orders. What if we could create a flag only for delayed orders and aggregate it separately? Conditional operators help you do exactly that. Let us look at a few conditional functions present in MySQL.

IF

As the name suggests, the IF function is the MySQL implementation of the if then else statement present in almost all programming languages. The syntax is pretty straightforward

```
IF(condition, value if true, value if false)
```

For this problem, we create an additional field that returns 1 if it is a delayed order.

```
SELECT
    DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
    , predicted_delivery_time
    , actual_delivery_time
    , TIMESTAMPDIFF(MINUTE, predicted_delivery_time, actual_delivery_time)
AS time_difference
    , 1 as NUMBER_OF_ORDERS
    , IF(TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20, 1, 0) as DELAYED_ORDERS
FROM delivery_orders
;
```

YYYYMM	predicted_delivery_time	actual_delivery_time	time_difference	NUMBER_OF_ORDERS	DELAYED_ORDERS
2021-11	2021-11-17 05:37:33	2021-11-17 05:58:33	21	1	1
2021-12	2021-12-09 19:41:43	2021-12-09 19:41:43	0	1	0
2022-01	2022-01-04 02:56:19	2022-01-04 03:21:19	25	1	1
2022-02	2022-02-09 01:19:22	2022-02-09 01:33:22	14	1	0
2021-12	2021-12-04 18:04:27	2021-12-04 18:31:27	27	1	1
2021-12	2021-12-04 13:34:49	2021-12-04 13:47:49	13	1	0
2022-02	2022-02-08 15:13:35	2022-02-08 15:24:35	11	1	0
2022-02	2022-02-04 03:28:23	2022-02-04 03:26:23	-2	1	0

As you can see, the IF condition flags the delayed orders separately. Now we can simply aggregate these two fields to find the totals.

```
SELECT
    DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
    , SUM(1) as NUMBER_OF_ORDERS
    , SUM(IF(TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20, 1, 0)) as DELAYED_ORDERS
FROM delivery_orders
GROUP BY 1
;
```

YYYYMM	NUMBER_OF_ORDERS	DELAYED_ORDERS
2021-11	13	4
2021-12	16	2
2022-01	11	4
2022-02	10	1

And find the portion of the delayed orders as a percentage of all orders.

```
SELECT
    DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
    , SUM(IF(TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20, 1, 0)) / SUM(1) * 100 as PERCENTAGE_DELAYED
FROM delivery_orders
GROUP BY 1
;
```

CASE WHEN

For more complex if then else conditions, we have the option of the CASE expression in MySQL. The CASE WHEN is capable of handling multiple conditions including complex conditions.

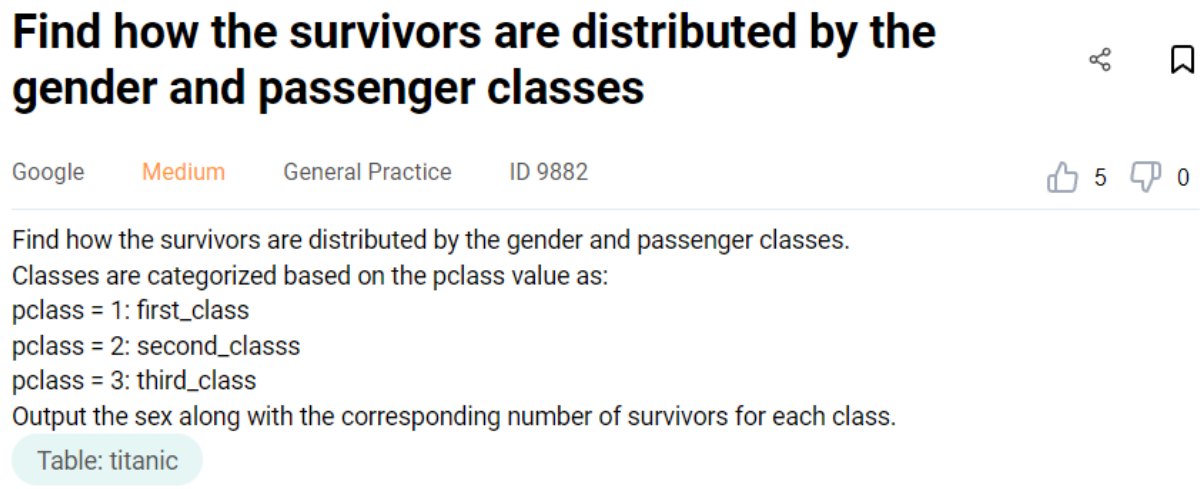
```
SELECT
    DATE_FORMAT(order_placed_time, '%Y-%m')  AS YYYYMM
    , SUM(CASE WHEN TIMESTAMPDIFF(MINUTE, predicted_delivery_time,
actual_delivery_time) > 20 THEN 1 END) / SUM(1) * 100 as PERCENTAGE_DELAYED
FROM delivery_orders
GROUP BY 1
;
```

This gives an identical result as the IF() function.

Pivoting

Another use of conditional operators is to convert a long format table to a wide format table. This is the opposite of aggregation. This is akin to creating a pivot table in spreadsheet programs. Let us see how we can do this.

MySQL Interview Question #12: Find how the survivors are distributed by the gender and passenger classes



Link to the question: https://platform.stratascratch.com/coding/9882-find-how-the-survivors-are-distributed-by-the-gender-and-passenger-classes

This problem uses the `titanic` table with the following fields.

titanic Preview

passengerid:	int
survived:	int
pclass:	int
name:	varchar
sex:	varchar
age:	float
sibsp:	int
parch:	int
ticket:	varchar
fare:	float
cabin:	varchar
embarked:	varchar

The data is presented in the following manner.

passengerid	survived	pclass	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.283	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.05		S
6	0	3	Moran, Mr. James	male		0	0	330877	8.458		Q
7	0	1	McCarthy, Mr. Timothy J	male	54	0	0	17463	51.862	E46	S
8	0	3	Palsson, Master. Gosta Leonard	male	2	3	1	349909	21.075		S

As with the previous problem, we can separately aggregate each class into multiple CTEs and then join them. However, a more efficient way to solve this problem is to tag the class of each passenger using a conditional expression. Here we use the `CASE WHEN` expression.

```
SELECT
    sex
    , pclass
    , survived
    , CASE WHEN pclass = 1 THEN survived END AS first_class
    , CASE WHEN pclass = 2 THEN survived END AS second_class
    , CASE WHEN pclass = 3 THEN survived END AS third_class
FROM titanic
;
```

sex	pclass	survived	first_class	second_class	third_class
male	3	0			0
female	1	1	1		
female	3	1			1
female	1	1	1		
male	3	0			0
male	3	0			0
male	1	0	0		
male	3	0			0
female	3	1			1
female	2	1		1	

As you can observe, we have expanded the width of the table by adding columns that tag the class that the passenger was traveling in. We can now simply aggregate these columns and group them by gender.

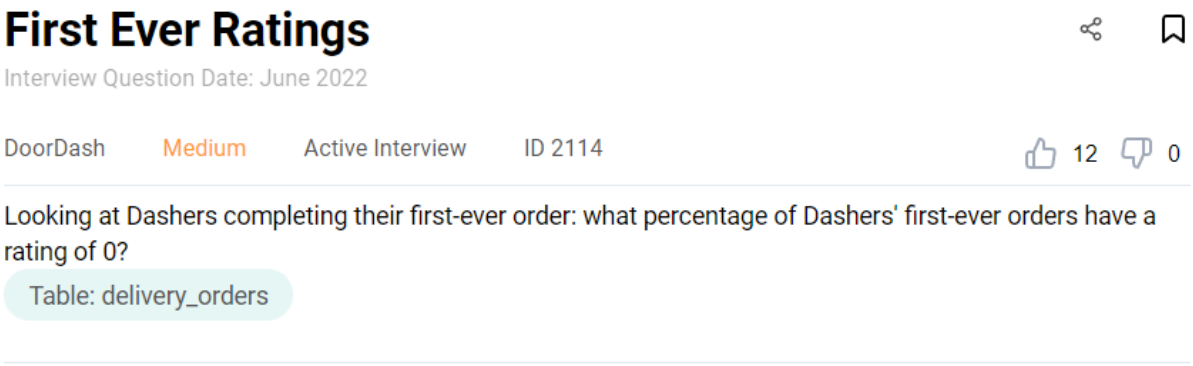
```
SELECT
    sex
    , SUM(CASE WHEN pclass = 1 THEN survived END) AS first_class
    , SUM(CASE WHEN pclass = 2 THEN survived END) AS second_class
    , SUM(CASE WHEN pclass = 3 THEN survived END) AS third_class
FROM titanic
GROUP BY 1
ORDER BY 1
```

```
;
```

Window Functions

Aggregations and CTEs can help solve a lot of problems. However, there are scenarios where one wishes we could do both at the same time. Let us take this problem.

MySQL Interview Question #13: First Ever Ratings



Link to the question: https://platform.stratascratch.com/coding/2114-first-ever-ratings

This problem uses the `delivery_orders` table with the following fields.

delivery_orders Preview

delivery_id:	varchar
order_placed_time:	datetime
predicted_delivery_time:	datetime
actual_delivery_time:	datetime
delivery_rating:	int
dasher_id:	varchar
restaurant_id:	varchar
consumer_id:	varchar

The data is presented in the following manner.

delivery_id	order_placed_time	predicted_delivery_time	actual_delivery_time	delivery_rating	dasher_id	restaurant_id	consumer_id
O2132	2021-11-17 04:45:33	2021-11-17 05:37:33	2021-11-17 05:58:33	4	D239	R633	C1001
O2152	2021-12-09 19:09:43	2021-12-09 19:41:43	2021-12-09 19:41:43	3	D238	R635	C1010
O2158	2022-01-04 02:31:19	2022-01-04 02:56:19	2022-01-04 03:21:19	4	D239	R634	C1010
O2173	2022-02-09 00:45:22	2022-02-09 01:19:22	2022-02-09 01:33:22	0	D239	R633	C1038
O2145	2021-12-04 17:20:27	2021-12-04 18:04:27	2021-12-04 18:31:27	1	D239	R634	C1042
O2144	2021-12-04 12:50:49	2021-12-04 13:34:49	2021-12-04 13:47:49	1	D238	R637	C1059
O2172	2022-02-08 14:53:35	2022-02-08 15:13:35	2022-02-08 15:24:35	5	D237	R636	C1059

The key to solving this problem is to identify the first order for each dasher. Once that is identified, we can use a `CASE WHEN` expression to separate out the dashers with the first delivery rating as zero. Let us try to identify the first order. We can try using a CTE or sub-query to find the first order timestamp and merge it back to the dataset to find the first order. Something like this.

```
WITH first_order_ts AS (
    SELECT
        dasher_id
        , MIN(actual_delivery_time) as first_order_timestamp
    FROM delivery_orders
    GROUP BY 1
```

```
)
SELECT
    do.dasher_id
    , do.delivery_id
    , do.delivery_rating
    , do.actual_delivery_time
    , CASE WHEN fo.dasher_id IS NOT NULL THEN 1 ELSE 0 END AS
is_first_order
FROM
    delivery_orders AS do
    LEFT JOIN first_order_ts AS fo
        ON do.dasher_id = fo.dasher_id
        AND do.actual_delivery_time = fo.first_order_timestamp
;
```

dasher_id	delivery_id	delivery_rating	actual_delivery_time	is_first_order
D239	O2132	4	2021-11-17 05:58:33	1
D238	O2152	3	2021-12-09 19:41:43	0
D239	O2158	4	2022-01-04 03:21:19	0
D239	O2173	0	2022-02-09 01:33:22	0
D239	O2145	1	2021-12-04 18:31:27	0
D238	O2144	1	2021-12-04 13:47:49	0
D237	O2172	5	2022-02-08 15:24:35	0
D239	O2170	1	2022-02-04 03:26:23	0
D240	O2150	3	2021-12-08 11:25:50	0
D240	O2177	0	2022-02-26 16:24:49	0
D240	O2129	3	2021-11-07 07:21:08	1
D238	O2147	3	2021-12-05 15:11:51	0

Now it is pretty easy to find the required output. But what if we had to find the rating for the second order? While it is possible to create CTEs for this, it is going to be complicated. Enter window functions. Window functions work across different table rows like aggregation functions, but they retain the original rows without aggregating them. The rows still retain their separate identities. Let us try to solve this using a window function.

```
WITH ranked_orders AS (
SELECT
    dasher_id
    , delivery_id
    , actual_delivery_time
    , delivery_rating
```

```
    , ROW_NUMBER() OVER (PARTITION BY dasher_id ORDER BY dasher_id,
actual_delivery_time) AS order_rank
FROM delivery_orders
)
SELECT
*
FROM ranked_orders
;
```

dasher_id	delivery_id	actual_delivery_time	delivery_rating	order_rank
D237	O2131	2021-11-15 23:45:56	4	1
D237	O2136	2021-11-28 00:05:44	3	2
D237	O2142	2021-12-04 01:45:59	3	3
D237	O2165	2022-01-27 15:36:57	4	4
D237	O2172	2022-02-08 15:24:35	5	5
D238	O2135	2021-11-23 18:47:38	0	1
D238	O2140	2021-12-01 00:07:09	1	2
D238	O2143	2021-12-04 08:53:37	2	3
D238	O2144	2021-12-04 13:47:49	1	4
D238	O2147	2021-12-05 15:11:51	3	5

This gives an elegant way of ordering the rows as well as ranking them without resorting to aggregations. Now the problem becomes a simple application of the `CASE WHEN` expression.

```
WITH ranked_orders AS (
SELECT
    dasher_id
    , delivery_id
```

```
    , actual_delivery_time
    , delivery_rating
    , ROW_NUMBER() OVER (PARTITION BY dasher_id ORDER BY dasher_id,
actual_delivery_time) AS order_rank
FROM delivery_orders
)
SELECT
    COUNT(DISTINCT CASE WHEN order_rank = 1 AND delivery_rating = 0 THEN
dasher_id END)
    / COUNT(DISTINCT CASE WHEN order_rank = 1 THEN dasher_id END) * 100.0
as pct
FROM ranked_orders
;
```

The solution is customizable. One can find the percentages for the second or fifth on any other rank very easily. This is a simple application of window functions. We can apply them in many more complex cases. You can read our Ultimate Guide to SQL Window functions here to learn more about them.

Conclusion

MySQL is a powerful database tool and is widely used across different industries. If you are experienced in other SQL flavors like Postgres, Oracle, etc., you can easily switch over to MySQL. Remember to use the documentation generously. If you are starting out on your SQL journey, MySQL is a good place to start. As with any other skill in life, all one needs to master is patience, persistence, and practice. On the StrataScratch platform, you will find over 600 real-life interview problems involving coding in SQL (MySQL and Postgres) and Python. Sign up today and join a community of over 20,000 like-minded aspirants and helpful mentors to help you on your mission to join top companies like Microsoft, Apple, Amazon, Google, et al.