QUESTIONS THÉORIQUES

A **relational database** is a database that stores data in tables. Tables are similar to folders in a file system, where each table stores data about a particular subject. For example, a customer orders table might store data about customer orders, and a product table might store data about products.
There are three types of **relationships in a database**: one-to-one, one-to-many, and many-to-many.

- A one-to-one relationship is when each row in one table is related to only one row in another table. For example, a customer table might have a one-to-one relationship with an orders table, where each customer is related to only one order.
- A one-to-many relationship is when each row in one table is related to multiple rows in another table. For example, a customers table might have a one-to-many relationship with an orders table, where each customer is related to multiple orders.
- A many-to-many relationship is when each row in one table is related to multiple rows in another table, and each row in the other table is related to multiple rows in the first table. For example, a customers table might have a many-to-many relationship with a products table, where each customer is related to multiple products and each product is related to multiple customers.

What are DDL and DML?
DDL is Data Definition Language which is used to define data structures. For example: create table, alter table are instructions in SQL.
DML is Data Manipulation Language which is used to manipulate data itself. For example: insert, update, delete are instructions in SQL.

What are some of the most important SQL data types?
SQL supports a number of different data types, including numeric, text, date/time, and Boolean values. Numeric values include integers and floating-point numbers, while text values include character strings and date/time values include date, time, and timestamp values. Boolean values can either be TRUE or FALSE.

Types de clés

- Une clé **primaire** permet d'identifier chaque ligne ou chaque enregistrement d'une table de base de données. La valeur de cette clé doit être toujours non nulle et chaque enregistrement doit avoir une clé primaire unique. La clé primaire est un index, chacune des tables ne peut contenir qu'une seule clé primaire, composée d'un ou plusieurs champs ou colonnes.
- La clé **unique** permet d'identifier un enregistrement unique dans une table et une table peut avoir plusieurs clés uniques. Les contraintes de clé uniques ne peuvent accepter qu'une seule valeur NULL pour une colonne.
- La clé **étrangère** représente un champ (ou des champs) qui pointe vers la clé primaire d'une autre table. L'objectif de cette clé est d'assurer l'intégrité référentielle des données.

What's an index and how is it used?
An index is a database structure that is used to improve the performance of SQL queries. Indexes can be created on columns in a table, and they are typically used to speed up searches for specific values in those columns. When a query is executed, the database engine will first check to see if an index exists for the columns that are being searched; if an index exists, the engine will use the index to quickly locate the desired data, which can improve query performance.

How do you optimize a SQL query?
There are a few different ways to optimize a SQL query. One way is to make sure that the columns you're interested in are indexed, so the database can more quickly find the data you're looking for. Another way is to use the EXPLAIN command to see how the database will execute your query, and then make changes to your query based on that information. Finally, you can use query hints to give database-specific instructions on how to execute your query.
Another way you can optimize a SQL query is to use a tool like SQL Profiler to see where the bottlenecks are in your query and then make changes accordingly.
Setting NOCOUNT ON reduces the time required for SQL Server to count rows affected by INSERT, DELETE, and other commands.

Using INNER JOIN with a condition is much faster than using WHERE clauses with conditions SQL compute services can charge based on the amount of data scanned. If myorders has 50 columns, using * will ensure all 50 columns are included in the query which means you have scanned 200GB of data to return orders after July 1, 2021. => specifying the column names of interest

How have you used SQL to solve a problem?

DATABASE-RELATED COMMANDS

See currently available databases
SHOW DATABASES;

Create a new database
CREATE DATABASE <database_name>;

Select a database to use
USE <database_name>;

Import SQL commands from .sql file
SOURCE <path_of_.sql_file>;

Delete a database
DROP DATABASE <database_name>;

Query to Display User Tables
A user-defined table is a representation of defined information in a table, and they can be used as arguments for procedures or user-defined functions. Because they're so useful, it's useful to keep track of them using the following query.
SELECT * FROM Sys.objects WHERE Type='u'

Query to Display Primary Keys
A primary key uniquely identifies all values within a table. The following SQL query lists all the fields in a table's primary key.
SELECT * from Sys.Objects WHERE Type='PK'

Query for Displaying Unique Keys
A Unique Key allows a column to ensure that all of its values are different. It is similar to the primary key but can accept a null value, unlike it.
SELECT * FROM Sys.Objects WHERE Type='uq'

Displaying Foreign Keys
Foreign keys link one table to another — they are attributes in one table which refer to the primary key of another table.
SELECT * FROM Sys.Objects WHERE Type='f'

Displaying Triggers
A Trigger is sort of an 'event listener' — i.e, it's a pre-specified set of instructions that execute when a certain event occurs. The list of defined triggers can be viewed using the following query.
SELECT * FROM Sys.Objects WHERE Type='tr'
Example of trigger :
When a new customer data is entered into the company's database he has to send the welcome message to each new customer. If it is one or two customers John can do it manually, but what if the count is more than a thousand? Well in such scenario triggers come in handy.

Displaying Internal Tables
Internal tables are formed as a by-product of a user-action and are usually not accessible. The data in internal tables cannot be manipulated; however, the metadata of the internal tables can be viewed using the following query.
SELECT * FROM Sys.Objects WHERE Type='it'

Displaying a List of Procedures
A stored procedure is a function. Using the following query you can keep track of them:
SELECT * FROM Sys.Objects WHERE Type='p'

Union
Unir les résultats des 2 tables.

SELECT * FROM table1
UNION
SELECT * FROM table2
Pour garder les doublons : UNION ALL

TABLE-RELATED COMMANDS

Writing comments
/* This query below is commented so it won't execute*/

See currently available tables in a database
SHOW TABLES;

Create a new table
CREATE TABLE <table_name1> (
  <col_name1> <col_type1>,
  <col_name2> <col_type2>,
  <col_name3> <col_type3>
  PRIMARY KEY (<col_name1>),
  FOREIGN KEY (<col_name2>) REFERENCES <table_name2>(<col_name2>)
);
Example :
CREATE TABLE instructor (
  ID CHAR(5),
  name VARCHAR(20) NOT NULL,
  dept_name VARCHAR(20),
  salary NUMERIC(8,2),
  PRIMARY KEY (ID),
  FOREIGN KEY (dept_name) REFERENCES department(dept_name))

Describe columns of a table
View the columns of a table with details such as the type and key.
DESCRIBE <table_name>;

Modify columns of a table
ALTER TABLE Customers ADD Birthday varchar(80)

Insert into a table
INSERT INTO <table_name> (<col_name1>, <col_name2>, <col_name3>, …)
  VALUES (<value1>, <value2>, <value3>, …);
If you are inserting values to all the columns of a table, then there is no need to specify the column names at the beginning.
INSERT INTO <table_name>
  VALUES (<value1>, <value2>, <value3>, …);

Copying Selections from Table to Table
There are a hundred and one uses for this SQL tool. Suppose you want to archive your yearly Orders table into a larger archive table. This next example shows how to do it.
INSERT INTO Yearly_Orders
SELECT * FROM Orders
WHERE Date<=1/1/2018

Update a table
UPDATE <table_name>
  SET <col_name1> = <value1>, <col_name2> = <value2>, ...
  WHERE <condition>;
Example :
UPDATE Customers SET Zip=Phone, Phone=Zip

Delete all contents of a table
DELETE FROM <table_name>;

Delete a table
DROP TABLE <table_name>;

Select
The SELECT statement is used to select data from a particular table.
SELECT <col_name1>, <col_name2>, …
FROM <table_name>;

Select distinct
A column of a table can often contain duplicate values. SELECT DISTINCT allows you to retrieve the distinct values.
SELECT DISTINCT <col_name1>, <col_name2>, …
FROM <table_name>;

Where
You can use WHERE keyword in a SELECT statement in order to include conditions for your data.
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <condition>;

Group By
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
GROUP BY <col_namex>;
Example :
SELECT COUNT(course_id), dept_name
FROM course
GROUP BY dept_name;

Having
The HAVING clause was introduced to SQL because the WHERE keyword could not be used to compare values of aggregate functions.
SELECT <col_name1>, <col_name2>, ...
FROM <table_name>
GROUP BY <column_namex>
HAVING <condition>
Example :
List the number of courses for each department which offers more than one course.
SELECT COUNT(course_id), dept_name
FROM course
GROUP BY dept_name
HAVING COUNT(course_id)>1;

Order By
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
ORDER BY <col_name1>, <col_name2>, … ASC|DESC;

Making a Top 25
Show a sample of 25 of these records to demonstrate the column headings.
SELECT TOP 25 FROM Customers WHERE Customer_ID<>NULL;

Lag and Lead
Lag and lead functions are used to access data from a previous or future row in a table. Lag functions return data from a row that is preceding the current row, while lead functions return data from a row that is following the current row.
Example :
SELECT
seller_name,
sale_value,
LAG(sale_value) OVER(ORDER BY sale_value) as previous_sale_value
FROM sale;

Limit and Offset
SELECT DISTINCT salary
FROM employee
WHERE department = ‘engineering’
ORDER BY salary DESC LIMIT 1 OFFSET 1;
This SQL query will select the second-highest salary from the engineering department.

Between
BETWEEN clause is used to select data within a given range. The values can be numbers, text or even dates.
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <col_namex> BETWEEN <value1> AND <value2>;

Like
The LIKE operator is used in a WHERE clause to search for a specified pattern in text.
There are two wildcard operators used with LIKE.
% (Zero, one, or multiple characters)
_ (A single character)
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <col_namex> LIKE <pattern>;
Example :
SELECT * FROM course WHERE course_id LIKE 'CS-___';

Tie things up with Strings
Let’s have a look at processing the contents of field data using functions. Substring is probably the most valuable of all built-in functions. It gives you some of the power of Regex, but it’s not so complicated as Regex. Suppose you want to find the substring left of the dots in a web address. Here’s how to do it with an SQL Select query:
SELECT SUBSTRING_INDEX("medium.com/gumare64", ".", 2);
This line will return everything to the left of the second occurrence of “. ” and so, in this case, it will return
www.medium.com/ghumare64

In
Using IN clause, you can allow multiple values within a WHERE clause.
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <col_namen> IN (<value1>, <value2>, …);
Example :
List the students in the departments of Comp. Sci., Physics, and Elec. Eng.
SELECT * FROM student
WHERE dept_name IN (‘Comp. Sci.’, ‘Physics’, ‘Elec. Eng.’);

Join
SELECT <col_name1>, <col_name2>, …
FROM <table_name1>
JOIN <table_name2>
ON <table_name1.col_namex> = <table2.col_namex>;
Il y a plusieurs méthodes pour associer 2 tables ensemble. Voici la liste des différentes techniques qui sont utilisées :
- INNER JOIN : jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables. C’est l’une des jointures les plus communes.
- CROSS JOIN : jointure croisée permettant de faire le produit cartésien de 2 tables. En d’autres mots, permet de joindre chaque lignes d’une table avec chaque lignes d’une seconde table. Attention, le nombre de résultats est en général très élevé.
- LEFT JOIN (ou LEFT OUTER JOIN) : jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n’est pas vérifié dans l’autre table.

- RIGHT JOIN (ou RIGHT OUTER JOIN) : jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifié dans l'autre table.
- FULL JOIN (ou FULL OUTER JOIN) : jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables.
- SELF JOIN : permet d'effectuer une jointure d'une table avec elle-même comme si c'était une autre table.
- NATURAL JOIN : jointure naturelle entre 2 tables s'il y a au moins une colonne qui porte le même nom entre les 2 tables SQL
- UNION JOIN : jointure d'union

Views
Views are virtual SQL tables created using a result set of a statement. It contains rows and columns and is very similar to a general SQL table. A view always shows up-to-date data within the database.
CREATE VIEW :
```
CREATE VIEW <view_name> AS
   SELECT <col_name1>, <col_name2>, …
   FROM <table_name>
   WHERE <condition>;
```
Example :
```
CREATE VIEW Failing_Students AS
  SELECT s_name, Student_ID
  FROM STUDENT
  WHERE GPA > 40
```
UPDATE VIEW :
```
CREATE OR REPLACE VIEW <view_name> AS
   SELECT <col_name1>, <col_name2>, …
   FROM <table_name>
   WHERE <condition>;
```
DROP VIEW :
```
DROP VIEW <view_name>;
```

Aggregate Functions
These functions are used to obtain a cumulative result relevant to the data being considered. Following are the commonly used aggregate functions.

COUNT(col_name) — Returns the number of rows
SUM(col_name) — Returns the sum of the values in a given column
AVG (col_name)— Returns the average of the values of a given column
MIN(col_name) — Returns the smallest value of a given column
MAX(col_name) — Returns the largest value of a given column

With Rollup
Sous-totaux.
```
SELECT office_id, job_title, SUM(salary)
FROM employees
GROUP BY office_id, job_title WITH ROLLUP
```

Cumulated Sums
```
SELECT
   invoice_id,
   invoice_total,
   SUM(invoice_total) OVER (ORDER BY invoice_id) AS total_sum
FROM invoices
```

Window Functions
A SQL window function is a function that performs a calculation on a set of values and returns a single value. Unlike aggregate functions, which return one result per group, window functions return one result per row. Common window functions include RANK, DENSE_RANK, and NTILE.

With RANK, if the two 90s are given a ranking of 2, the next lowest value would be assigned a rank of 4, skipping over 3. With DENSE_RANK, the next lowest value would be assigned a rank of 3, not skipping over any values.
Example :

```
SELECT
   student_name,
   grades
   RANK() OVER(ORDER BY grades DESC) AS grade_ranking
FROM students
```

Example with PARTITION BY :
If you partitioned the data by subject, it would give you the ranking of each score, grouped by subject.

```
SELECT
   student_name,
   subject,
   DENSE_RANK() OVER(PARTITION BY subject ORDER BY grades DESC) AS grade_ranking
FROM students
```

Example :

```
SELECT
   Name,
   NTILE(4) OVER(ORDER BY Sales DESC) AS Quartile
FROM Sales
```

Numeric Functions
CAST : Change le type de valeur ; CAST(result AS INT64)
ROUND : Arrondit ; ROUND(318.12, 0) => 318
TRUNC : Enlève les chiffres après la virgule ; TRUNC(1239, -2) => 1200
COALESCE : Donne la première valeur non nulle de la liste ; COALESCE(NULL, 5, ‘GFG’) => 5

Date Functions
Some common date functions in SQL are:
- CURRENT_DATE: Returns the current date.
- CURRENT_TIME: Returns the current time.
- CURRENT_TIMESTAMP: Returns the current date and time.
- DATE_ADD: Adds a specified number of days, months, or years to a date.
- DATE_SUB: Subtracts a specified number of days, months, or years from a date.
- DAY: Returns the day of the month for a given date.
- MONTH: Returns the month for a given date.
- YEAR: Returns the year for a given date.
- DATE_FORMAT: Change format.

Examples :

```
SELECT CURRENT_DATE()
SELECT DATE_ADD("2017-06-15 09:34:21", INTERVAL -3 HOUR)
SELECT DATE_FORMAT("2018-09-24", "%D %b %Y")
```

String Functions
- LENGTH : longueur du string
- LOWER/UPPER : mettre en minuscule/majuscule
- INITCAP : mettre la première lettre en majuscule et les autres en minuscule
- SUBSTRING/SUBSTR : SUBSTRING(chaine, debut) retourne la chaîne de caractère de “chaine” à partir de la position définie par “debut” (position en nombre de caractères) ;
SUBSTRING(chaine, debut, longueur) retourne la chaîne de caractère “chaine” en partant de la position définie par “debut” et sur la longueur définie par “longueur”

Trim
Supprimer un caractère spécifique en début et en fin de chaîne : TRIM(BOTH 'x' FROM 'xxxExemplexxx'); => “Exemple”
Supprimer un caractère spécifique uniquement en début de chaîne : TRIM(LEADING 'x' FROM 'xxxExemplexxx'); => “Exemplexxx”

Supprimer un caractère spécifique uniquement en fin de chaîne : TRIM(TRAILING 'x' FROM 'xxxExemplexxx'); => “xxxExemple”
Sur SQLite : TRIM('x', 'xxxExemplexxx'); => Exemple

Replace
Pour modifier la table :
UPDATE site
SET url = REPLACE(url, 'www.facebook.com', 'fr-fr.facebook.com')
WHERE url LIKE '%www.facebook.com%'
Pour ne pas modifier la table :
SELECT colonne1, colonne2, REPLACE(colonne3, 'exemple insulte, 'CENSURE')
FROM table

Ifnull
In cases where NULL values are allowed in a field, calculations on those values will produce NULL results as well. This can be avoided by use of the IFNULL operator. In this next example, a value of zero is returned rather than a value of NULL when the calculation encounters a field with NULL value:
SELECT Item, Price *
(QtyInStock + IFNULL(QtyOnOrder, 0))
FROM Orders

Nested Subqueries
Nested subqueries are SQL queries which include a SELECT-FROM-WHERE expression that is nested within another query.
Example : Find courses offered in Fall 2009 and in Spring 2010.

```
SELECT DISTINCT course_id
   FROM section
   WHERE semester = ‘Fall’ AND year= 2009 AND course_id IN (
      SELECT course_id
       FROM section
         WHERE semester = ‘Spring’ AND year= 2010);
```

Correlated Subqueries
A correlated subquery is a type of SQL query that contains a reference to a value from outer query. Correlated subqueries are typically used when you want to find rows from a table that match certain conditions, but you can only know those conditions after examining other rows in the same table.
For example, you could use a correlated subquery to find all employees who make more than the average salary in their department. In this case, you would need to calculate the average salary for each department before you could compare each employee’s salary to it.
Example :
The following example finds the products whose list price is equal to the highest list price of the products within the same category:

```
SELECT
   product_name,
   list_price,
   category_id
FROM
   production.products p1
WHERE
   list_price IN (
      SELECT
         MAX (p2.list_price)
      FROM
         production.products p2
      WHERE
         p2.category_id = p1.category_id
      GROUP BY
         p2.category_id
```

```
    )
ORDER BY
    category_id,
    product_name;
```

Conditional Subquery Results
The SQL operator EXISTS tests for the existence of records in a subquery and returns a value TRUE if a subquery returns one or more records. Have a look at this query with a subquery condition:

```
SELECT Name FROM Customers WHERE EXISTS
(SELECT Item FROM Orders
WHERE Customers.ID = Orders.ID AND Price < 50)
```

In this example above, the SELECT returns a value of TRUE when a customer has orders valued at less than $50.

Case when
CASE WHEN condition THEN 'output if true' ELSE 'output if false' END
Example : Select the name of the student and their grade. if their score is more than 70, they passed, or else, they fail.

```
CASE WHEN score > 70 THEN "Pass" ELSE "Fail" END AS Grade
```

IIF
IIF is a shorthand way for writing a CASE expression

```
SELECT
    grade,
    IIF(grade>70, 'You Passed', 'You did not pass') as PassOrNot
From Grades_Table
```

Create function

```
CREATE FUNCTION full_name(first_name VARCHAR(50), last_name VARCHAR(50))
RETURNS VARCHAR(50) DETERMINISTIC
RETURN CONCAT(first_name, ' ', last_name);
```

Call the function :

```
SELECT full_name(first_name, last_name) AS name
FROM customers
```

QUESTIONS 1

https://www.stratascratch.com/

Write a query that calculates the difference between the highest salaries found in the marketing and engineering departments. Output just the absolute difference in salaries.

```
SELECT
(SELECT MAX(e.salary)
FROM db_employee AS e
LEFT JOIN db_dept AS d ON e.department_id = d.id
WHERE d.department = 'marketing')
-
(SELECT MAX(e.salary)
FROM db_employee AS e
LEFT JOIN db_dept AS d ON e.department_id = d.id
WHERE d.department = 'engineering')
```

We have a table with employees and their salaries, however, some of the records are old and contain outdated salary information. Find the current salary of each employee assuming that salaries increase each year. Output their id, first name, last name, department ID, and current salary. Order your list by employee ID in ascending order.

```
SELECT id, first_name, last_name, department_id, MAX(salary)
FROM ms_employee_salary
GROUP BY id
ORDER BY id
```

Ça suffit pour ne prendre que les lignes avec les valeurs max des salaires.

Find the details of each customer regardless of whether the customer made an order. Output the customer's first name, last name, and the city along with the order details.
You may have duplicate rows in your results due to a customer ordering several of the same items. Sort records based on the customer's first name and the order details in ascending order.

```
SELECT c.first_name, c.last_name, c.city, o.order_details
FROM customers AS c LEFT JOIN orders AS o ON c.id = o.cust_id
ORDER BY c.first_name, o.order_details
```

Les clients apparaîtront plusieurs fois s'ils ont fait plusieurs commandes, et les clients qui n'ont rien acheté apparaîtront aussi.

Find the average number of bathrooms and bedrooms for each city’s property types. Output the result along with the city name and the property type.

```
SELECT city, property_type, AVG(bathrooms), AVG(bedrooms)
FROM airbnb_search_details
GROUP BY city, property_type
```

Meta/Facebook has developed a new programing language called Hack.To measure the popularity of Hack they ran a survey with their employees. The survey included data on previous programing familiarity as well as the number of years of experience, age, gender and most importantly satisfaction with Hack. Due to an error location data was not collected, but your supervisor demands a report showing average popularity of Hack by office location. Luckily the user IDs of employees completing the surveys were stored.
Based on the above, find the average popularity of the Hack per office location.
Output the location along with the average popularity.

```
SELECT facebook_employees.location, AVG(facebook_hack_survey.popularity)
FROM facebook_hack_survey JOIN facebook_employees ON
facebook_hack_survey.employee_id = facebook_employees.id
GROUP BY facebook_employees.location
```

Compare each employee's salary with the average salary of the corresponding department.
Output the department, first name, and salary of employees along with the average salary of that department.

```
SELECT department, first_name, salary, AVG(salary) OVER(PARTITION BY department)
FROM employee
```

Find order details made by Jill and Eva.
Consider the Jill and Eva as first names of customers.
Output the order date, details and cost along with the first name.
Order records based on the customer id in ascending order.

```
SELECT c.first_name, o.order_date, o.order_details, o.total_order_cost
FROM customers AS c LEFT JOIN orders AS o ON c.id = o.cust_id
WHERE c.first_name =  'Jill' OR c.first_name = 'Eva'
ORDER BY c.id
```

Find all posts which were reacted to with a heart. For such posts output all columns from facebook_posts table.

```
SELECT DISTINCT p.*
FROM facebook_posts AS p LEFT JOIN facebook_reactions AS r ON p.post_id = r.post_id
WHERE r.reaction = 'heart'
```

Find the last time each bike was in use. Output both the bike number and the date-timestamp of the bike's last use (i.e., the date-time the bike was returned). Order the results by bikes that were most recently used.

```
SELECT bike_number, MAX(end_time)
FROM dc_bikeshare_q1_2012
GROUP BY bike_number
ORDER BY end_time DESC
```

Count the number of user events performed by MacBookPro users.
Output the result along with the event name.
Sort the result based on the event count in the descending order.

```
SELECT event_name, COUNT(*) AS nb_events
FROM playbook_events
WHERE device = 'macbook pro'
GROUP BY event_name
ORDER BY nb_events DESC
```

Find the most profitable company from the financial sector. Output the result along with the continent.

```
SELECT company, continent
FROM forbes_global_2010_2014
WHERE sector = 'financials'
ORDER BY profits DESC LIMIT 1
```

On peut ORDER BY une colonne non sélectionnée.

Find libraries who haven't provided the email address in circulation year 2016 but their notice preference definition is set to email.
Output the library code.

```
SELECT DISTINCT home_library_code
FROM library_usage
WHERE notice_preference_definition = 'email' AND circulation_active_year = 2016 AND
provided_email_address = 0
```

Find the base pay for Police Captains.
Output the employee name along with the corresponding base pay.

```
SELECT employeename, basepay
FROM sf_public_salaries
WHERE jobtitle LIKE 'CAPTAIN % (POLICE DEPARTMENT)'
```

Find how many times each artist appeared on the Spotify ranking list
Output the artist name along with the corresponding number of occurrences.
Order records by the number of occurrences in descending order.

```
SELECT artist, COUNT(*) AS nb_occurrences
FROM spotify_worldwide_daily_song_ranking
GROUP BY artist
ORDER BY nb_occurrences DESC
```

QUESTIONS 2

What is the overall friend acceptance rate by date? Your output should have the rate of acceptances by the date the request was sent. Order by the earliest date to latest.
Assume that each friend request starts by a user sending (i.e., user_id_sender) a friend request to another user (i.e., user_id_receiver) that's logged in the table with action = 'sent'. If the request is accepted, the table logs action = 'accepted'. If the request is not accepted, no record of action = 'accepted' is logged.

```
with a as(select *  from fb_friend_requests where action='accepted'),
 b as(select *  from fb_friend_requests  where action='sent')

select b.date,count(a.user_id_receiver)/count(b.user_id_sender) from a right join b on
a.user_id_sender=b.user_id_sender
and a.user_id_receiver=b.user_id_receiver group by date
```

Write a query that'll identify returning active users. A returning active user is a user that has made a second purchase within 7 days of any other of their purchases. Output a list of user_ids of these returning active users.

```
select distinct a1.user_id
from amazon_transactions as a1 join amazon_transactions as a2
on
    a1.user_id = a2.user_id
    and
    a1.id <> a2.id
    and
    a1.created_at >= a2.created_at
where datediff(a1.created_at, a2.created_at) <= 7
```

Find the customer with the highest daily total order cost between 2019-02-01 to 2019-05-01. If customer had more than one order on a certain day, sum the order costs on daily basis. Output customer's first name, total cost of their items, and the date.
For simplicity, you can assume that every first name in the dataset is unique.

```
select c.first_name, o.order_date, sum(o.total_order_cost) as total_cost
from customers c join orders o
on c.id = o.cust_id
where order_date between '2019-02-01' and '2019-05-01'
group by o.order_date, c.first_name
order by total_cost desc
limit 1;
```

https://quip.com/2gwZArKuWk7W

```
SELECT Customers.name AS CustomersNoOrder
FROM Customers
LEFT JOIN Orders
ON Customers.id = Orders.customerId
WHERE Orders.customerId is NULL
```

```
SELECT
    IFNULL(
        (SELECT DISTINCT salary
        From Employees
        ORDER BY salary DESC LIMIT 1 OFFSET 1),
    NULL)
    AS SecondHighestSalary
```

```
SELECT DISTINCT t1.*
FROM Stadium t1, Stadium t2, Stadium t3
WHERE t1.people >= 100 AND t2.people >= 100 AND t3.people >= 100
AND
(
    (t1.id - t2.id = 1 AND t1.id - t3.id = 2 AND t2.id - t3.id = 1)
    OR
    (t2.id - t1.id = 1 AND t2.id - t3.id = 2 AND t1.id - t3.id = 1)
    OR
    (t3.id - t2.id = 1 AND t2.id - t1.id = 1 AND t3.id - t1.id = 2)
)
ORDER BY t1.id
```

```
SELECT
    i.item_category AS Category,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Monday' THEN o.quantity ELSE 0 END) AS
Monday,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Tuesday' THEN o.quantity ELSE 0 END) AS
Tuesday,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Wednesday' THEN o.quantity ELSE 0 END) AS
Wednesday,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Thursday' THEN o.quantity ELSE 0 END) AS
Thursday,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Friday' THEN o.quantity ELSE 0 END) AS Friday,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Saturday' THEN o.quantity ELSE 0 END) AS
Saturday,
    SUM(CASE WHEN DAYNAME(o.order_date) = 'Sunday' THEN o.quantity ELSE 0 END) AS
Sunday
FROM Items i LEFT JOIN Orders o ON i.item_id = o.item_id
GROUP BY i.item_category
ORDER BY i.item_category
```

```
WITH result AS (
```

```
SELECT
    customer_id,
    product_id,
    product_name,
    DENSE_RANK() OVER(PARTITION BY customer_id ORDER BY total DESC) AS rnk
FROM (
    SELECT
        o.customer_id,
        o.product_id,
        p.product_name,
        COUNT(*) AS total
    FROM Orders AS o INNER JOIN Products AS p ON o.product_id = p.product_id
    GROUP BY o.customer_id, o.product_id, p.product_name
    ORDER BY o.customer_id, rnk DESC
    )
)

SELECT
    customer_id
    product_id
    product_name
FROM result
WHERE rnk = 1
```

```
WITH Company AS(
    SELECT
        DATE_FORMAT(pay_date, '%Y-%m') AS month,
        AVG(amount) AS company_avg_salary
    FROM salary
    GROUP BY month
),
Department AS (
    SELECT
        DATE_FORMAT(pay_date, '%Y-%m') AS month,
        e.department_id,
        AVG(s.amount) AS dept_avg_salary
    FROM salary AS s JOIN employee AS e
    ON s.employee_id = e.employee_id
    GROUP BY month, e.department_id
)

SELECT
    d.month AS pay_month,
    d.department_id
    CASE WHEN d.dept_avg_salary > c.company_avg_salary THEN 'higher'
    WHEN d.dept_avg_salary > c.company_avg_salary THEN 'lower'
    ELSE 'same'
    END AS comparison
FROM Department as d LEFT JOIN Company AS c
ON d.month = c.month
```

If we wanted to get the total two-day rolling average for sales by day, we could use the following SQL query:

```
SELECT date, sales, AVG(sales) OVER (ORDER BY date ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS “Two-Day Rolling Average”
FROM sales
```

```
GROUP BY date
ORDER BY date
```

If we wanted to find all the duplicate rows, we could use the following SQL query:

```
SELECT name, city
FROM customer_orders
GROUP BY name, city
HAVING COUNT(*) > 1
```