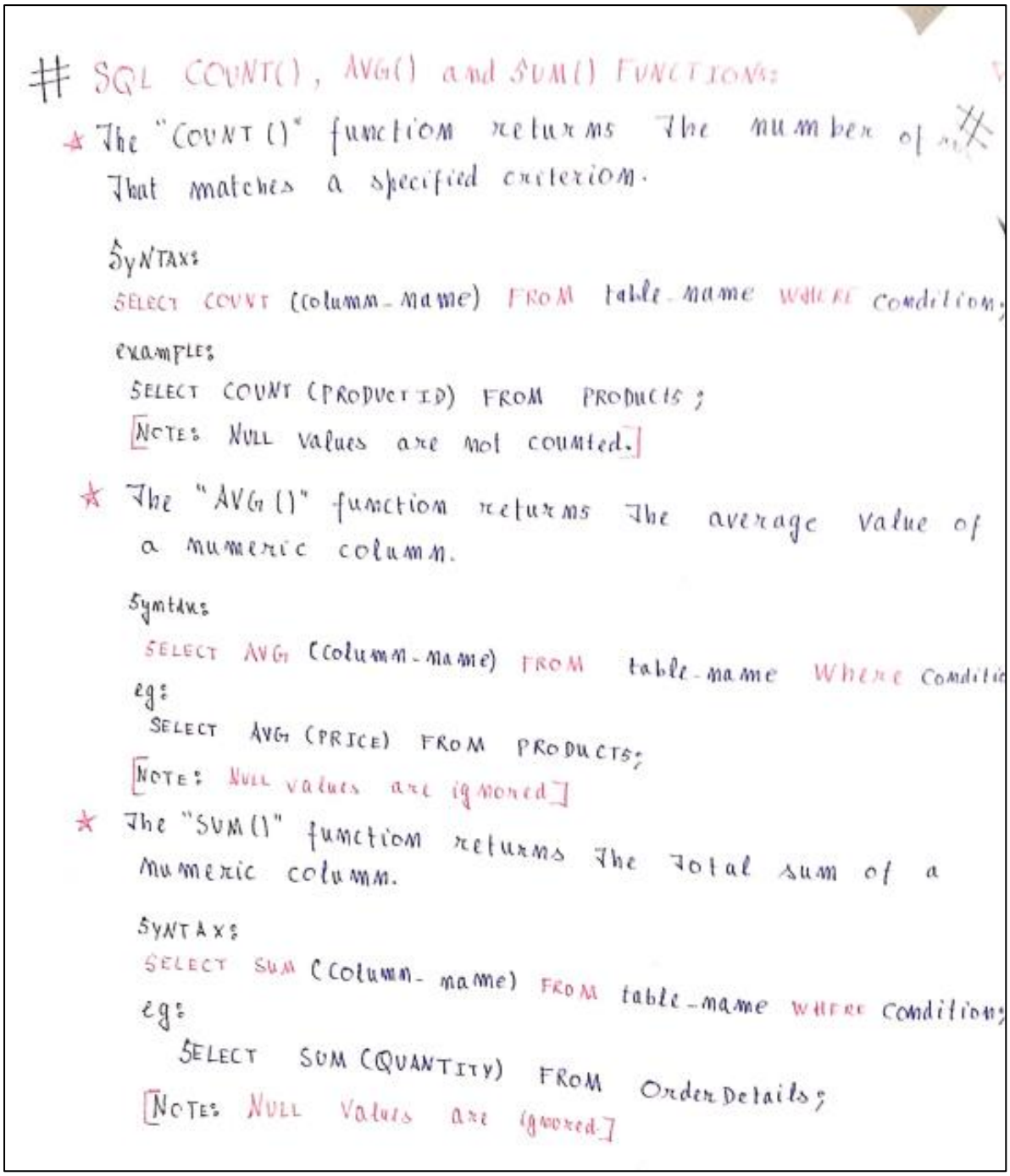
Day 2 of SRE batch (SQL):



SQL COUNT(), AVG() and SUM() FUNCTIONS:

★ The "COUNT()" function returns the number of rows that matches a specified criterion.

SYNTAX:

SELECT COUNT (column_name) FROM table_name WHERE condition;

examples:

SELECT COUNT (PRODUCTID) FROM PRODUCTS;

[NOTE: NULL values are not counted.]

★ The "AVG()" function returns the average value of a numeric column.

Syntax:

SELECT AVG (column_name) FROM table_name Where Conditio

eg:

SELECT AVG (PRICE) FROM PRODUCTS;

[NOTE: NULL values are ignored.]

★ The "SUM()" function returns the total sum of a numeric column.

SYNTAX:

SELECT SUM (column_name) FROM table_name WHERE condition;

eg:

SELECT SUM (QUANTITY) FROM OrderDetails;

[NOTE: NULL values are ignored.]

SQL IN OPERATOR:

The "IN" Operator allows you to specify multiple values in "where" clause. The "IN" OPERATOR is a shorthand for multiple 'OR' conditions

* SYNTAX:

- SELECT Column_name(s) FROM table_name WHERE column_name IN (value 1, value 2, ...);

OR

- SELECT column_name(s) FROM table_name WHERE column_name IN (SELECT statement);

* EXAMPLE:

- The following SQL statement selects all customers that are located in "GERMANY", "FRANCE" OR "UK"

```
SELECT * FROM CUSTOMERS WHERE COUNTRY IN ('GERMANY',
'France', 'UK');
```

- The following SQL statement selects all customers that are NOT LOCATED in "GERMANY", "FRANCE" OR "UK":

```
SELECT * FROM CUSTOMERS WHERE COUNTRY NOT IN (
'GERMANY', 'France', 'UK');
```

- The following SQL statement selects all customers that are from same Country as the suppliers:

```
SELECT * FROM CUSTOMERS WHERE COUNTRY IN (SELECT
COUNTRY FROM SUPPLIERS);
```

P.T.O...

```
select name,country
from Customers
where country in ('USA','Cananda')
```

SQL BETWEEN OPERATOR:

The "BETWEEN" OPERATOR SELECTS VALUES within a given range. The values can be number, text or dates.

The Between operator is inclusive: ie begin & end values are included.

* SYNTAX:

```
SELECT column_name(s) FROM table_name WHERE
column_name BETWEEN Value 1 AND Value 2;
```

* EXAMPLE:

```
SELECT * FROM PRODUCTS WHERE Price Between 10 AND 20;
```

* NOT BETWEEN EXAMPLE:

```
SELECT * FROM PRODUCTS WHERE Price NOT BETWEEN 10 AND 20;
```

* BETWEEN with IN EXAMPLE:

```
SELECT * FROM PRODUCTS WHERE Price BETWEEN 10 AND 20
AND CATEGORY_ID NOT IN (1, 2, 3);
```

* BETWEEN TEXT VALUES:

```
SELECT * FROM Products WHERE ProductName BETWEEN 'BEARDO'
AND 'MAMA EARTH' ORDER By ProductName;
```

* BETWEEN DATES EXAMPLE:

The following SQL statement selects all orders with an OrderDate between '01-July-1996' and '31-July-1996'.

Example:

```
SELECT * FROM Orders WHERE OrderDate BETWEEN #07/01/1996
AND #07/31/1996#;
```

OR

```
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-01'
AND '1996-07-31';
```

```sql
-- display orders in between the price range of 1000 to 2000
select OrderID, TotalAmount from orders where TotalAmount between 1000 and 2000;
```

SQL ALIASES:

SQL aliases are used to give a table, or a column in a table a temporary name.

Aliases are often used to make column name more readable. An Alias only exists for the duration of that query.

An Alias is created with the "AS" Keyword.

* Alias Column Syntax:

```
SELECT column_name AS alias_name FROM table_name;
```

eg: The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column.

```
SELECT CUSTOMERID AS ID, CUSTOMERNAME AS CUSTOMER FROM
CUSTOMERS;
```

* Alias Table Syntax:

```
SELECT column_name(s) FROM table_name AS alias_name;
```

Some Examples:

* The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column.

```
SELECT CUSTOMERNAME AS CUSTOMER, ContactName AS [CONTACT PERSON]
FROM CUSTOMERS;
```

[Note: It requires double quotation marks or square brackets if the alias name contains spaces.]

* The following SQL statement creates an alias named "ADDRESS" that combine four columns (Address, PostalCode, City and Country)

```
SELECT CUSTOMERNAME, ADDRESS + ',' + PostalCode + ' ' + City + ',' +
COUNTRY AS ADDRESSS FROM CUSTOMERS;
```

OR

```
SELECT CUSTOMERNAME, CONCAT(Address, ',', PostalCode, ',', City, ',',
COUNTRY) AS ADDRESS FROM CUSTOMERS;
```

OR

```
SELECT CUSTOMERNAME, (ADDRESS || ',' || PostalCode || ',' || City || ',' || COUNTRY) AS
ADDRESS FROM CUSTOMERS;
```

* Alias for Table Syntax:

SELECT o.orderID, o.OrderDate, c.customerName FROM Customers As c, Orders As o WHERE c.customerName = 'Around the horn' AND c.customerID = o.customerID;

(In the above query, it selects all the orders from the customer with CustomerID = 4 (i.e Around the horn).

We use the "Customers" and "Orders" Tables, and give them the Table aliases of "c" and "o" respectively (Here we use aliases to make SQL shorter)

If we write same query but without alias then

SELECT Orders.OrderID, Orders.OrderDate, CUSTOMERS.CUSTOMERName FROM Customers, Orders WHERE CUSTOMERS.CUSTOMER NAME = 'Around the horn' AND Customers.CustomerID = Orders.CustomerID;

EXAMPLE:

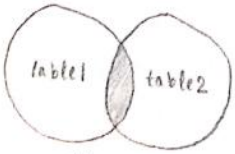
```
170 •  select distinct c.Name,c.Country
171    from Customers as c
172    join orders as o on
173    c.customerID =o.customerID
174    where c.country<>'USA' and o.totalamount >
175    ANY(
176    select totalamount
177    from orders as o2
178    join customers as c2 on c2.customerID=o2.customerId
179    where c2.country='USA'
180    );
```

SQL JOIN:

A "JOIN" clause is used to combine rows from two or more tables, based on a related column between them.

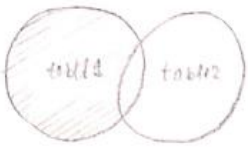
1) INNER JOIN: That Selects records that have matching values in both Tables.

eg: SELECT Orders.OrderID, CUSTOMERS.customerName, Orders.OrderDate FROM ORDERS INNER JOIN CUSTOMERS ON Orders.CustomerID = Customer.customerID;



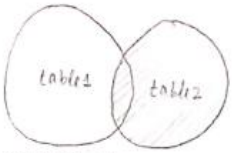
INNER JOIN

2) LEFT (OUTER) JOIN: Returns all records from left table, and the matched records from the right table.



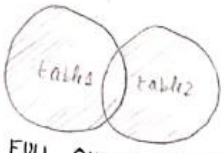
LEFT OUTER JOIN

3) Right (OUTER) JOIN: Returns all records from right table, and matched records from left table.



RIGHT JOIN

4) FULL (OUTER) JOIN: Returns all records when their is match in either left or right Table



FULL OUTER JOIN

FULL OUTER JOIN

SQL INNER JOIN:

The "Inner Join" keyword selects records that have matching values in both Table.

★ SYNTAX:

```
SELECT column_table(s) FROM table1 INNER JOIN table2 ON
table1.column_name = table2.column_name;
```

eg:
```
SELECT Orders.OrderID, Customers.CustomerName FROM Orders
INNER JOIN CUSTOMERS ON Order.customerID = customer.customerID;
```

NOTE: The Inner Join keyword selects all rows from both Tables as long as there is a match between columns.

if There are records in The "orders" table That do not have matches in "Customers", These orders will not be shown.

SQL LEFT JOIN:

The "Left JOIN" Keyword returns all records from the left Table (i.e table1) and the matching records from right table (i.e table2).

The result is 0 records from right side, if there is no match.

★ Syntax:

```
SELECT Column_name FROM table1 LEFT JOIN table2 ON table1.column_name
= table2.column_name ;
```

★ NOTE: The Left JOIN Keyword returns all records from left Table, even if there are no matches in the right Table (orders)

SQL Right JOIN:

The "Right JOIN" Keyword returns all records from the right Table (table2), and matching records from the left table (table1). The result is 0 records from left side, if there is no match.

★ Syntax:

```
SELECT column_name(s) FROM table1 Right JOIN table2 ON
table1.column_name = table2.column_name ;
```

★ NOTE: The right JOIN (even) Keyword returns all records from right table, if there are no matches in left table.

SQL FULL OUTER JOIN:

The "FULL OUTER JOIN" Keyword returns all records when there is a match in a left (table1) or right (table2) table records.

★ Syntax:

```
SELECT column_name(s) FROM table1 FULL OUTER JOIN table2 ON
table1.column_name = table2.column_name where condition;
```

★ NOTE:
it would select all the records from both table that would satisfy where condition. i.e it returns all matching records from both tables when other tables matches or not. so, if there are rows in "customers" that do not matches in "Orders" or vice versa. then all those rows will be listed as well.

SQL self JOIN:

A "SELF JOIN" is a regular join, but the table is JOINED with ITSELF.

★ SYNTAX:

```
SELECT column_name(s) FROM table1 T1, table1 T2 WHERE
condition;
```

EXAMPLE:

```sql
-- returning emoplyee name with their respective manager name using self join
select e.name as Ename,
e.department, m.Name as ManagerName
from employees e
left join employees m on e.managerid = m.empId;
```

SQL UNION Operator:

The "UNION" Operator is used to combine the result set of two or more "SELECT" statements.

- EVERY SELECT statement within UNION must have the same number of columns.
- The columns must also have similar data types
- The columns in every SELECT statement must also be in same order.

★ UNION SYNTAX:

```
SELECT column_name(s) FROM table1 UNION SELECT column_name(s)
FROM table2;
```

[NOTE: The union operator selects only Distinct values by default.
To allow duplicate values, we use UNION ALL.]

★ UNION ALL SYNTAX:

```
SELECT column_name(s) FROM table1 UNION ALL SELECT
column_name(s) FROM table2;
```

```sql
-- list all the countries of customers and their distinct order status
select country as business_feild from customers
union
-- it would return all the countries if we use union all, whereas union only returns distinct countries
select distinct Status from orders;
/* here all the countries and their order status would be returned in a single column, if we want them in two different columns then we
need to join the tables and use where clause*/
```

SQL GROUP BY Statement:

The "GROUP BY" statement groups rows that have the same values into summary rows, (like "find the no. of customers in each country").

The "GROUP BY" statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

★ GROUP BY SYNTAX:

```
SELECT column_name(s) FROM table_name WHERE condition
GROUP BY column_name(s) ORDER BY column_name(s);
```

★ examples:

```
SELECT COUNT(CUSTOMER ID), COUNTRY FROM CUSTOMERS GROUP BY
COUNTRY;
```

The above SQL statement lists the number of customers in each country.

SQL HAVING CLAUSE:

The "HAVING" clause was added to SQL because "where" keyword cannot be used with aggregated functions.

★ HAVING SYNTAX:

```
SELECT column_name(s) FROM table_name where condition
GROUP BY column_name(s) HAVING CONDITION ORDER BY
column_name(s);
```

★ Examples:

① `SELECT COUNT (CUSTOMER ID), country FROM Customers GROUP BY COUNTRY HAVING COUNT (CustomerID) > 5;`

(it lists the number of customers in each country. only include countries with more than 5 customers).

② `SELECT COUNT (CustomerID), country FROM Customers GROUP BY COUNTRY HAVING COUNT (CustomerID) > 5 Order By COUNT (CustomerID) DESC;`

(it lists the no of customers in each country, sorted high to low (only include countries with more than 5 customers)

EXAMPLE:

</> Code

MySQL Auto

```mysql
# Write your MySQL query statement below
select MAX(num) as num from(
    select num from MyNumbers group by num having count( num) =1
) AS unique_numbers;
```

SQL EXISTS Operator:

The "Exists" operator is used to test for the existence of any record in a subquery.

The "Exists" operator returns True if the subquery returns one or more records.

★ Syntax:

SELECT column_name(s) FROM table_name WHERE EXISTS (SELECT column_name FROM table_name WHERE condition);

★ Examples:

SELECT SupplierName FROM Suppliers WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Supplier.SupplierID AND Price < 20);

(The following SQL statement returns TRUE and LISTS the suppliers with a product price less than 20)

SQL ANY and ALL Operators:

The "ANY" and "ALL" operator allows us to perform a comparison b/w a single column value and a range of other values.

SELECT INTO SYNTAX:

The "SELECT INTO" statement copies data from one table into a new table.

★ SYNTAX:

SELECT * INTO newtable [IN externaldb] FROM oldtable WHERE condition;

(it would copy some columns into table)

eg:

SELECT * INTO CUSTOMER GERMANY FROM CUSTOMERS where country = 'Germany';

The SQL INSERT INTO SELECT STATEMENT:

The "INSERT INTO SELECT" statement copies data from one table and inserts it into another table.
(It requires that data types in source & target tables match)

[NOTE: The existing records in the target table are unaffected]

★ Syntax:

[To copy all columns from one table to another table:]

INSERT INTO table2 SELECT * FROM table1 WHERE condition;

[To copy some columns from one table into another table]

INSERT INTO table2 (column1, column2, ...) SELECT column1,
column2, ... FROM table1 where condition;

The SQL CASE Expression:

The "CASE" expression goes through conditions and returns a value ~~within~~ when the first condition is met (like an if-then-else statement). So once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in else clause.

If there is no ELSE part & no condition are true, it returns Null

★ SYNTAX:

```
CASE
    when condition 1   then result 1
    when condition 2   then result 2
    when condition 3   then result 3
    else result
End;
```

Example:

```
SELECT OrderID, QUANTITY, CASE WHEN QUANTITY > 30 THEN 'The
quantity is greater than 30' WHEN QUANTITY = 30 THEN 'The Quantity
is 30' ELSE 'The Quantity is under 30' END AS Quantity Text
FROM Order Details;
```

SQL CONSTRAINTS:

* Syntax:

```
Create table Table_name ( column1 datatype constraints,
                          column2 datatype constraints,
                          column3 datatype constraints,
                          ... );
```

The "SQL CONSTRAINTS" are used to specify rules for the data in a table.

CONSTRAINTS are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to whole table.

* The following are some commonly used SQL constraints

- NOT NULL → Ensures that a column cannot have a Null value
- UNIQUE → Ensures that all values in a column are different.
- PRIMARY KEY → it is a combination of a NOT NULL AND UNIQUE. Uniquely identifies each row in a table.
- FOREIGN KEY → Prevents actions that would destroy links between tables.
- CHECK → Ensures that a value in a column satisfies a specific condition.
- DEFAULT → Sets a default value for a column if no value is specified.
- CREATE INDEX → Used to create and retrieve data from the database very quickly.

SQL NOT NULL CONSTRAINTS

By default a column can hold Null values.

The 'NOT NULL' constraint enforces a column to not accept Null values which means we can't insert a new record (or row), or update a record without adding a value to this field.

* USING NOT NULL while creating a table:

```
Create table Persons ( Id int NOT NULL , LASTNAME VARCHAR(255
NOT NULL , FirstName varchar(255) NOT NULL , age int );
```

* SQL NOT NULL ON ALTER TABLE:

```
ALTER TABLE Persons ALTER COLUMN Age int NOT NULL;
```

Or

```
ALTER TABLE Persons Modify column Age int NOT NULL;
```

Or

```
ALTER TABLE Persons Modify Age int NOT NULL;
```

SQL UNIQUE CONSTRAINT:

The "UNIQUE" Constraint ensures that all values in column are different

Both UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of Columns.

A primary key constraints automatically has a unique constraint.

However, we can have many unique constraints per Table, but only one Primary Key constraints per Table.

P.T.O...

* USING UNIQUE CONSTRAINT ON CREATE TABLE:

The following SQL creates a UNIQUE constraint on the "ID" column, when the Person Table is created.

Syntax:

```
Create table Persons ( Id int NOT NULL UNIQUE , LASTNAME
varchar (255) NOT NULL , FirstNAME varchar (255) );
```

Or

Create Table Persons (Id int NOT NULL, LASTNAME VARCHAR (255) NOT NULL, FirstName varchar (255), UNIQUE (ID));

To APPLY UNIQUE CONSTRAINT ON MULTIPLE COLUMNS:

CREATE TABLE Persons (ID int Not NULL, LASTNAME Varchar (255) Not NULL, FirstName Varchar (255),
CONSTRAINT UC_Person UNIQUE (ID, LASTName));

To APPLY UNIQUE CONSTRAINT ON ALTER TABLE:

ALTER TABLE PERSON ADD UNIQUE (ID);

To APPLY ON MULTIPLE COLUMNS:

ALTER TABLE PERSON ADD CONSTRAINTS UC_Person UNIQUE (ID, LastName);

DROP a unique Constraint:

ALTER TABLE Person DROP INDEX UC_Person;

Or

ALTER TABLE Persons DROP CONSTRAINTS UC_Person;

T.T...

SQL PRIMARY KEY CONSTRAINTS:

The "Primary key" constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain Null Values.

A table can have only ONE Primary key.

[NOTE: This primary key can consist of single or Multiple columns]

☆ Syntax:

CREATE TABLE Persons (Id int NOT NULL, Last Name varchar(255) NOT NULL, Firstname Varchar(255), Age int, Primary Key (ID));

Or

CREATE TABLE Persons (Id int NOT NULL PRIMARY KEY, LAST Name varchar(255) NOT NULL, FirstName Varchar (255), Age int);

★ Syntax: To Set Multiple Column as Primary Key...

```
CREATE TABLE PERSONS ( Id int NOT NULL, LAST Name varchar (255) NOT NU
First Name varchar (255), Age int, CONSTRAINT PK_Person Primary KEY
(Id, LastName));
```

NOTE: In the example above there is only one Primary Key (PK_Person). However, the value of the Primary Key is made up of two Columns (ID + LastName)]

SQL PRIMARY KEY ON ALTER TABLE:

SYNTAX:

```
ALTER TABLE Persons ADD PRIMARY KEY (ID);
```

DROP a PRIMARY KEY CONSTRAINT:

SYNTAX:

```
ALTER TABLE Persons DROP PRIMARY KEY;
```

Or

```
ALTER TABLE Persons DROP CONSTRAINT PK_ Person;
```

SQL FOREIGN KEY CONSTRAINTS

The "FOREIGN KEY" Constraints is used To Prevent actions That would destroy link between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, That refers To Primary Key in Another Table.

NOTE: The table with foreign key is called a child table, and the Table That refers To Primary Key (or with Primary Key) is called Parent Table.

The "foreign Key" constraint Prevents invalid data from being inserted into The foreign Key column, because it has to be One of The values contained in The Parent Table.

★ Syntax:

```
CREATE TABLE Orders ( orderid int NOT NULL, Order Number int
NOT NULL, PersonId int, Primary Key (Order Id), foreign Key (
PersonId) Refrences Persons (PersonId));
```

Or

```
CREATE TABLE Orders ( Orderid int NOT NULL PRIMARY KEY, Order
NUMBER int NOT NULL, PersonId int FOREIGN KEY Refrences
Persons (Person Id));
```

SQL FOREIGN KEY ON ALTER TABLE:

To create a "FOREIGN KEY" constraint on the "Person Id" column when the "Orders" table is already created.

Syntax:

```
ALTER TABLE Orders Add FOREIGN KEY (Person Id) References Persons
(Person Id);
```

DROP a FOREIGN KEY CONSTRAINT:

Syntax:

```
ALTER TABLE Orders DROP FOREIGN KEY FK_PersonOrder;
                    or
ALTER TABLE Orders DROP CONSTRAINT FK_Person Order;
```

SQL CHECK CONSTRAINT:

The "CHECK" constraint is used to limit the value range that can be placed in a column.

if you define a Check constraint on a column it will allow only certain values for this column.

if you define a check constraint on a table it can limit the values in certain columns based on values in other columns in the row.

★ Syntax:

```
CREATE TABLE Persons ( Id int NOT NULL, LastName varchar (255)
NOT NULL, FirstName varchar (255), Age int, CHECK (Age >= 18));
```

★ CHECK CONSTRAINT ON MULTIPLE column:

```
CREATE TABLE Persons ( Id int NOT NULL, LastName varchar (255)
NOT NULL, FirstName varchar (255), Age int, city varchar (255),
CONSTRAINT CHK_Person CHECK (Age >= 18 AND City = "sanders");
```

SQL CHECK ON ALTER TABLE:

```
ALTER TABLE Persons ADD CHECK (Age >= 18);
```

SQL DEFAULT CONSTRAINT:

The "DEFAULT" CONSTRAINT is used to set a default value for a column. The default value will be added to all new records, if no other value is specified.

★ Syntax:

```
CREATE TABLE Persons ( Id int NOT NULL, LastName Varchar (255)
NOT NULL, Firstname Varchar (255), Age int, City varchar (255)
DEFAULT 'sandnes');
                    Or
CREATE TABLE Orders ( Id int NOT NULL, OrderNumber int NOT NULL,
Order_date date Default Getdate());
```

SQL AUTO INCREMENT FIELD:

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a Table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

★ Syntax:

```
CREATE TABLE Persons ( Personid int NOT NULL AUTO_INCREMENT,
LastName Varchar (255) NOT NULL , FirstName Varchar (255), Age int,
PRIMARY KEY (Personid));
```

[NOTE: By default, the starting value for Auto_INCREMENT is 1, and it will increment by 1 for each new record. To let Auto_increment sequence start with another value.]

★ Syntax:

```
ALTER TABLE Persons AUTO_INCREMENT = 100;
```

SQL Date Data Types:

- DATE : format yyyy-MM-DD
- DATETIME: format yyyy-MM-DD HH:MI:SS
- TIMESTAMP: format yyyy-MM-DD HH:MI:SS
- YEAR : format yyyy or yy

eg: SELECT * FROM Orders WHERE OrderDate = '2008-11-11'

P.T.O...

SQL LIKE OPERATOR:

The "LIKE" Operator is used in where clause to search for specified pattern in a column. There are Two wild cards often used in conjuction with the LIKE OPERATOR.

- The Percent sign (%) represents Zero, one, or MULTIPLE Characters
- The underscore sign (_) represents one, single character

[NOTE: MS Access uses an asterisk (*) instead of the Percent sign (%) and a question mark (?) instead of the underscore (_).]

SYNTAX:

```
SELECT column1, column2, .... FROM table_name WHERE column
LIKE PATTERN;
```

LIKE PATTERN;

LIKE OPERATOR WITH '%' and '_' wild cards:

LIKE OPERATOR	DESCRIPTION
• WHERE CUSTOMERNAME LIKE 'a%'	find any values that start with 'a'
• WHERE CUSTOMERNAME LIKE '%a'	find any values that end with 'a'
• WHERE CUSTOMERNAME LIKE '%or%'	find any values that have "or" in any position.
• WHERE CUSTOMERNAME LIKE '_r%'	find any values that have "r" in second position.
• WHERE CUSTOMERNAME LIKE 'a_%'	finds any values that starts with "a" and are atleast 2 characters in length.
• WHERE CUSTOMERNAME LIKE 'a__%'	finds any values that starts with "a" and are atleast 3 characters in length.
• WHERE CONTACTNAME LIKE 'a%o'	finds any values that starts with "a" and ends with "o".

★ Examples:

- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME LIKE 'a%';
  (The following SQL statement selects all customers with CustomerName starting with "a").
- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME LIKE '%a';
  (The following SQL statement selects all customers with CustomerName ending with "a").
- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME LIKE '%or%';
  (The following SQL statement selects all customers with a CustomerName that have "or" in any Position)
- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME NOT LIKE 'a%';
  (The following SQL statement selects all customers with a CustomerName that doesNot start with "a").

SQL WILDCARDS Characters:

A "WILD CARD" Character is used to substitute one or more characters in a string.

Wildcard characters are used with the "LIKE operator". The LIKE OPERATOR is used in a "WHERE" CLAUSE to search for a specified Pattern in a column.

★ WILD CARD CHARACTERS IN SQL SERVER:

SYMBOL	DESCRYPTION	Example
%	Represents zero or more characters.	bl% would find black, blue and blob.
_	Represents a single character	h_t would find hot, hat, hit.
[]	Represents any single character within brackets	h[oa]t would find hot, hat but not hit as i not present in bracket.
^	Represents any character not in brackets.	h[^oa]t would find hit (as not hot or hat)
–	Represents any single character within specified range (-)	c[a-b]t finds cat

★ Examples: ★ USING The "%" Wildcard:

- The following SQL statement selects all Customers with a city starting with "ber":

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE 'ber%';
```

- The following SQL statement selects all Customers with a city containing the pattern "es":

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '%es%';
```

★ USING The "_" WILDCARD:

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '_ondon';
```

(The above statement selects all Customers with a city starting with any character, followed by "ondon".)

- The following SQL statement selects all customers with a city starting with "L", following by any character, followed by "m", followed by any character, followed by "on":

```
SELECT * FROM CUSTOMERS WHERE City LIKE "L-m-on";
```

★ USING The [charlist] WILDCARD:

- The following SQL statement selects all customers with a City starting with "b", "s" or "p":

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '[bsp]%';
```

- The following SQL statement selects all customers with a city starting with "a", "b" or "c"

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '[a-c]%';
```

★ USING The [!charlist] Wildcard:

- The Two following SQL statements select all customers with a city NOT starting with "b", "s" or "p"

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '[!bsp]%';
```

Or

```
SELECT * FROM CUSTOMER WHERE CITY NOT LIKE '[bsp]%';
```

BIT MANIPULATION IN TABLES:

```sql
-- Create a table to demonstrate bit operations
CREATE TABLE permissions (
    user_id INT PRIMARY KEY,
    username VARCHAR(50),
    permission_flags INT  -- Will store permission bits
);
```

```sql
-- Insert sample data
INSERT INTO permissions (user_id, username, permission_flags) VALUES
(1, 'admin', 7),      -- Binary: 111 (Read: 1, Write: 1, Execute: 1)
(2, 'developer', 6), -- Binary: 110 (Read: 1, Write: 1, Execute: 0)
(3, 'viewer', 4),     -- Binary: 100 (Read: 1, Write: 0, Execute: 0)
(4, 'guest', 1);      -- Binary: 001 (Read: 0, Write: 0, Execute: 1)
```

```sql
-- Constants for permission bits
-- READ    = 4 (Binary: 100)
-- WRITE   = 2 (Binary: 010)
-- EXECUTE = 1 (Binary: 001)
```

```sql
-- 1. Bitwise AND (&) to check if user has specific permission
-- Check which users have read permission (4)
select
username,
permission_flags & 4 as has_read_permission,
case
   when permission_flags & 4 > 0 then 'Yes'
   else 'No'
end as can_read
from permissions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

username	has_read_permission	can_read
viewer	4	Yes
guest	0	No

```sql
#Add write permission to all user wh dont have it
update permissions
set permission_flags = permission_flags | 2
where (permission_flags & 2) =0
```

```sql
select * from permissions
```

```sql
#toggle the execute permission for user
update permissions
set permission_flags =permission_flags ^ 1
where (permission_flags & 1)=0;
```

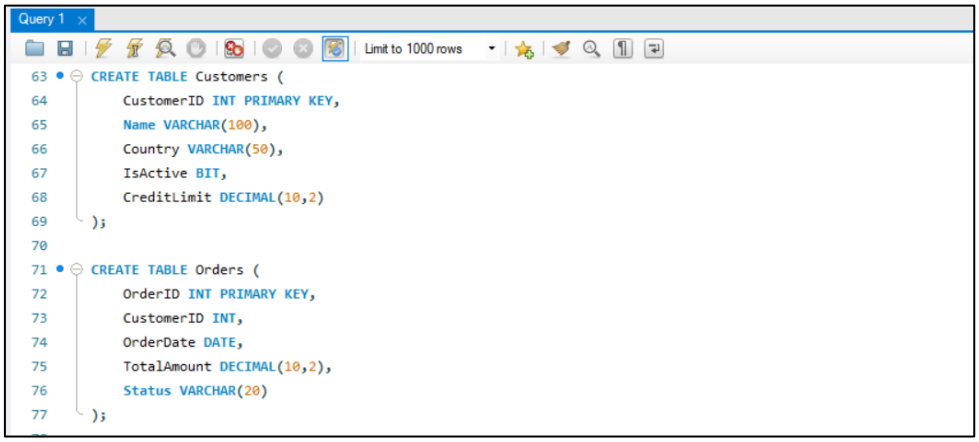
By toggling we mean that , if a user already has execution permission then, we would switch it off and vice versa.
(for this we would use XOR operator) .

```sql
-- 7. Bit shifting operations
CREATE TABLE bit_shift_demo (
    id INT PRIMARY KEY,
    value INT
);

INSERT INTO bit_shift_demo (id, value) VALUES
(1, 8),   -- Binary: 1000
(2, 12),  -- Binary: 1100
(3, 16);  -- Binary: 10000

select
id,
value,
value << 1 as left_shft_1,
value << 2 as left_shift_2
from bit_shift_demo;
```

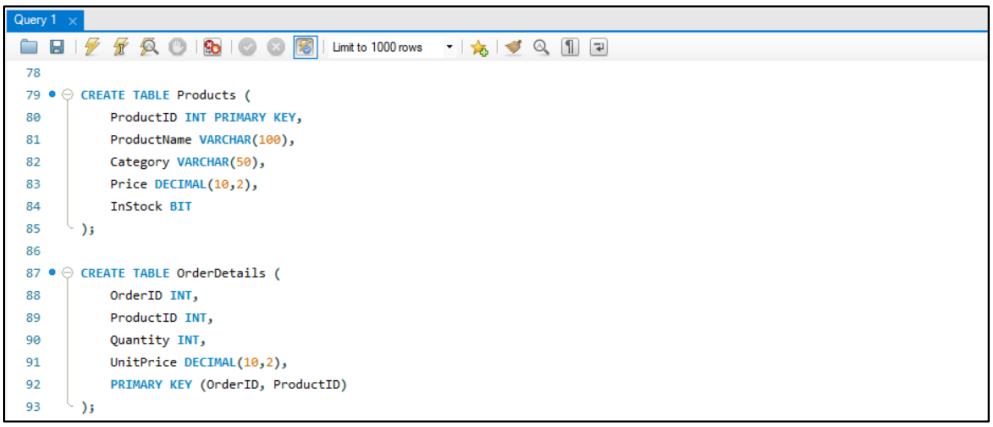
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	value	left_shft_1	left_shift_2
1	8	16	32
2	12	24	48
3	16	32	64

CREATING A FULL FLEDGE CUSTOMER, ORDERS, PRODUCTS,ORDER DETAILS:



```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Country VARCHAR(50),
    IsActive BIT,
    CreditLimit DECIMAL(10,2)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10,2),
    Status VARCHAR(20)
);
```



```sql
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    Category VARCHAR(50),
    Price DECIMAL(10,2),
    InStock BIT
);

CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    UnitPrice DECIMAL(10,2),
    PRIMARY KEY (OrderID, ProductID)
);
```

```sql
-- Insert sample data
INSERT INTO Customers VALUES
(1, 'John Doe', 'USA', 1, 5000.00),
(2, 'Jane Smith', 'Canada', 1, 3000.00),
(3, 'Bob Johnson', 'USA', 0, 2000.00),
(4, 'Alice Brown', 'UK', 1, 4000.00),
(5, 'Charlie Wilson', 'Canada', 1, 6000.00);

INSERT INTO Orders VALUES
(1, 1, '2024-01-01', 1500.00, 'Delivered'),
(2, 1, '2024-01-15', 2000.00, 'Pending'),
(3, 2, '2024-01-20', 1000.00, 'Delivered'),
(4, 3, '2024-02-01', 500.00, 'Cancelled'),
(5, 4, '2024-02-15', 3000.00, 'Processing');
```

```sql
INSERT INTO Products VALUES
(1, 'Laptop', 'Electronics', 1200.00, 1),
(2, 'Smartphone', 'Electronics', 800.00, 1),
(3, 'Desk Chair', 'Furniture', 200.00, 0),
(4, 'Coffee Maker', 'Appliances', 100.00, 1),
(5, 'Headphones', 'Electronics', 150.00, 1);

INSERT INTO OrderDetails VALUES
(1, 1, 1, 1200.00),
(1, 2, 1, 800.00),
(2, 3, 2, 200.00),
(3, 4, 1, 100.00),
(4, 5, 2, 150.00);
```

```sql
/* say we want to return all the users whoose purchase is greater then any purchase of usa
   for this we would firstly use join to merge the orders and customer table on basis of i
   and then we'll use "any clause to slect from usa purchases sub query"
*/

select distinct c.Name,c.Country
from Customers as c
join orders as o on
c.customerID =o.customerID
where c.country<>'USA' and o.totalamount >
ANY(
select totalamount
from orders as o2
join customers as c2 on c2.customerID=o2.customerId
where c2.country='USA'
);
```

```sql
select name from customers c
where exists(
select 1
from orders as o
where o.customerID=c.customerID);
```

```sql
-- to divide customers based on their credit
select name,
case when CreditLimit >=5000 then "premium"
     when CreditLimit >=3000 then "gold"
     else "standard"
end as customerTier
from customers;
```

```sql
-- to display products not in stock
select ProductName from products where not Instock;

-- display orders in between the price range of 1000 to 2000
select OrderID, TotalAmount from orders where TotalAmount between 1000 and 2000;
```

```sql
-- list all the countries of customers and their distinct order status
select country as business_feild from customers
union /* it would return all the countries if we use union all, whereas union only returns distinct countries */
select distinct Status from orders;
/* here all the countries and their order status would be returned in a single column, if we want them in two different columns then we
need to join the tables and use where clause*/
```

```sql
/* find all the products which are in stock and have been ordered and to selct atleast 1 from orderdetails where product id of
orders matches with product id of products*/
SELECT DISTINCT p.ProductID, p.ProductName, p.Category, p.Price
FROM Products p
JOIN OrderDetails od ON p.ProductID = od.ProductID
```

```sql
-- if we have small table and we need specific memory efficient way then we can go with this way
WHERE p.InStock = 1;
SELECT ProductName
FROM Products as P
WHERE P.InStock = 1
AND EXISTS (SELECT 1 FROM OrderDetails as OD WHERE OD.ProductID = P.ProductID);
```

RANKING THE EMPLOYEES BASED ON SALARY:

```sql
-- rank the employees based on salary
-- Create Employee table
CREATE TABLE Employees (
    EmpID INT PRIMARY KEY,
    Name VARCHAR(50),
    Department VARCHAR(50),
    Salary DECIMAL(10,2),
    HireDate DATE,
    ManagerID INT
);
```

```sql
-- Insert sample data
INSERT INTO Employees VALUES
(1, 'John Doe', 'IT', 75000, '2022-01-15', NULL),      -- CEO
(2, 'Jane Smith', 'IT', 65000, '2022-02-01', 1),       -- IT Manager
(3, 'Bob Wilson', 'IT', 55000, '2022-03-15', 2),       -- IT Staff
(4, 'Alice Brown', 'HR', 70000, '2022-02-15', 1),      -- HR Manager
(5, 'Charlie Davis', 'HR', 50000, '2022-04-01', 4),    -- HR Staff
(6, 'Eve Wilson', 'Sales', 60000, '2022-03-01', 1),    -- Sales Manager
(7, 'Frank Miller', 'Sales', 45000, '2022-05-01', 6),  -- Sales Staff
(8, 'Grace Lee', 'IT', 55000, '2022-03-15', 2),        -- IT Staff
(9, 'Henry Ford', 'Sales', 45000, '2022-05-01', 6);    -- Sales Staff
```

```sql
/* for this we use the rank and dense rank function, rank function is used to rank but if two people have same salary then
rank would be skipped for next person, where as in dense rank it gives exact rank for the user*/
select name, department, salary, rank() over (order by salary desc) as SalaryRank,
dense_rank() over (order by salary desc) as SalaryDenserank
from employees;

-- if we want to partion them according based on their department and then ordering based on their salary
select name, department, salary, rank() over (partition by department order by salary desc) as SalaryRank,
dense_rank() over (partition by department order by salary desc) as SalaryDenserank
from employees;
```

```sql
-- returning emoplyee name with their respective manager name using self join
select e.name as Ename,
e.department, m.Name as ManagerName
from employees e
left join employees m on e.managerid = m.empId;
```

Leet-code important SQL questions:

2356. Number of Unique Subjects Taught by Each Teacher

Easy | Topics | Companies

SQL Schema > | Pandas Schema >

Table: `Teacher`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| teacher_id  | int  |
| subject_id  | int  |
| dept_id     | int  |
+-------------+------+
(subject_id, dept_id) is the primary key (combinations of columns with unique values) of this table.
Each row in this table indicates that the teacher with teacher_id teaches the subject subject_id in the department dept_id.
```

Write a solution to calculate the number of unique subjects each teacher teaches in the university.

Return the result table in **any order**.

The result format is shown in the following example.

Example 1:

Input:

Teacher table:

```
+------------+------------+---------+
| teacher_id | subject_id | dept_id |
+------------+------------+---------+
| 1          | 2          | 3       |
| 1          | 2          | 4       |
| 1          | 3          | 3       |
| 2          | 1          | 1       |
| 2          | 2          | 1       |
| 2          | 3          | 1       |
| 2          | 4          | 1       |
+------------+------------+---------+
```

Output:

```
+------------+-----+
| teacher_id | cnt |
+------------+-----+
| 1          | 2   |
| 2          | 4   |
+------------+-----+
```

Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
select teacher_id, count(distinct(subject_id)) as cnt from teacher
group by teacher_id;
```

1141. User Activity for the Past 30 Days I

Easy | Topics | Companies

SQL Schema > | Pandas Schema >

Table: `Activity`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
This table may have duplicate rows.
The activity_type column is an ENUM (category) of type ('open_session',
'end_session', 'scroll_down', 'send_message').
The table shows the user activities for a social media website.
Note that each session belongs to exactly one user.
```

Write a solution to find the daily active user count for a period of `30` days ending `2019-07-27` inclusively. A user was active on someday if they made at least one activity on that day.

Return the result table in **any order**.

Example 1:

Input:

Activity table:

user_id	session_id	activity_date	activity_type
1	1	2019-07-20	open_session
1	1	2019-07-20	scroll_down
1	1	2019-07-20	end_session
2	4	2019-07-20	open_session
2	4	2019-07-21	send_message
2	4	2019-07-21	end_session
3	2	2019-07-21	open_session
3	2	2019-07-21	send_message
3	2	2019-07-21	end_session
4	3	2019-06-25	open_session
4	3	2019-06-25	end_session

Output:

day	active_users
2019-07-20	2
2019-07-21	2

Explanation: Note that we do not care about days with zero active users.

Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
SELECT activity_date AS day, COUNT(DISTINCT user_id) AS active_users
FROM Activity
WHERE (activity_date > "2019-06-27" AND activity_date <= "2019-07-27")
GROUP BY activity_date;
```

1070. Product Sales Analysis III

Medium | Topics | Companies

SQL Schema > Pandas Schema >

Table: `Sales`

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
(sale_id, year) is the primary key (combination of columns with unique values) of this table.
product_id is a foreign key (reference column) to Product table.
Each row of this table shows a sale on the product product_id in a certain year.
Note that the price is per unit.
```

Table: `Product`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
product_id is the primary key (column with unique values) of this table.
Each row of this table indicates the product name of each product.
```

Write a solution to select the **product id**, **year**, **quantity**, and **price** for the **first year** of every product sold.

Return the resulting table in **any order**.

The result format is in the following example.

Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
select product_id, year as first_year, quantity, price from sales
where (product_id, year) IN (
    select product_id, MIN(year) as year
    from sales
    group By product_id);
```

596. Classes More Than 5 Students

Easy | Topics | Companies

SQL Schema > | Pandas Schema >

Table: `Courses`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| student     | varchar |
| class       | varchar |
+-------------+---------+
(student, class) is the primary key (combination of columns with unique values)
for this table.
Each row of this table indicates the name of a student and the class in which
they are enrolled.
```

Write a solution to find all the classes that have **at least five students**.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

Input:

Courses table:

```
+---------+----------+
| student | class    |
+---------+----------+
| A       | Math     |
| B       | English  |
| C       | Math     |
| D       | Biology  |
| E       | Math     |
| F       | Computer |
| G       | Math     |
| H       | Math     |
| I       | Math     |
+---------+----------+
```

Output:

```
+---------+
| class   |
+---------+
| Math    |
+---------+
```

Explanation:

- Math has 6 students, so we include it.
- English has 1 student, so we do not include it.
- Biology has 1 student, so we do not include it.
- Computer has 1 student, so we do not include it.

Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
SELECT class from courses group by class having count(student) >=5;
```

1729. Find Followers Count

Easy | Topics | Companies

SQL Schema > Pandas Schema >

Table: `Followers`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| user_id     | int  |
| follower_id | int  |
+-------------+------+
(user_id, follower_id) is the primary key (combination of columns with unique
values) for this table.
This table contains the IDs of a user and a follower in a social media app where
the follower follows the user.
```

Write a solution that will, for each user, return the number of followers.

Return the result table ordered by `user_id` in ascending order.

The result format is in the following example.

Example 1:

Input:

Followers table:

```
+---------+-------------+
| user_id | follower_id |
+---------+-------------+
| 0       | 1           |
| 1       | 0           |
| 2       | 0           |
| 2       | 1           |
+---------+-------------+
```

Output:

```
+---------+----------------+
| user_id | followers_count|
+---------+----------------+
| 0       | 1              |
| 1       | 1              |
| 2       | 2              |
+---------+----------------+
```

Explanation:

The followers of 0 are {1}
The followers of 1 are {0}
The followers of 2 are {0,1}

Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
SELECT user_id, count(distinct (follower_id)) as followers_count from followers
group by user_id;
```

619. Biggest Single Number

Easy | Topics | Companies

SQL Schema > Pandas Schema >

Table: `MyNumbers`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| num         | int  |
+-------------+------+
This table may contain duplicates (In other words, there is no primary key for
this table in SQL).
Each row of this table contains an integer.
```

A **single number** is a number that appeared only once in the `MyNumbers` table.

Find the largest **single number**. If there is no **single number**, report `null`.

The result format is in the following example.

Example 1:

Input:
MyNumbers table:

num
8
8
3
3
1
4
5
6

Output:

num
6

Explanation: The single numbers are 1, 4, 5, and 6.
Since 6 is the largest single number, we return it.

Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
select MAX(num) as num from(
    select num from MyNumbers group by num having count( num) =1
) AS unique_numbers;
```

1045. Customers Who Bought All Products

Medium | Topics | Companies

SQL Schema > Pandas Schema >

Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| customer_id | int     |
| product_key | int     |
+-------------+---------+
This table may contain duplicates rows.
customer_id is not NULL.
product_key is a foreign key (reference column) to Product table.
```

Table: `Product`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_key | int     |
+-------------+---------+
product_key is the primary key (column with unique values) for this table.
```

Write a solution to report the customer ids from the `Customer` table that bought all the products in the `Product` table.

Return the result table in **any order**.

The result format is in the following example.

Input:

Customer table:

customer_id	product_key
1	5
2	6
3	5
3	6
1	6

Product table:

product_key
5
6

Output:

customer_id
1
3

Explanation:

The customers who bought all the products (5 and 6) are customers with IDs 1 and 3.

</> Code

MySQL | Auto

```mysql
# Write your MySQL query statement below
SELECT customer_id from customer
group by customer_id
having count(distinct product_key)=(select count(distinct product_key) from product);
```