SQL-Question Bank

1.What does UNION **do? What is the difference between** UNION **and** UNION ALL?

UNION merges the contents of two structurally-compatible tables into a single combined table. The difference between UNION and UNION ALL is that UNION will omit duplicate records whereas UNION ALL will include duplicate records.
It is important to note that the performance of UNION ALL will typically be better than UNION, since UNION requires the server to do the additional work of removing any duplicates. So, in cases where is certain that there will not be any duplicates, or where having duplicates is not a problem, use of UNION ALL would be recommended for performance reasons.

2.List and explain the different types of JOIN **clauses supported in ANSI-standard SQL.**

ANSI-standard SQL specifies five types of JOIN clauses as follows:

- INNER JOIN (a.k.a. “simple join”): Returns all rows for which there is at least one match in BOTH tables. *This is the default type of join if no specific JOIN type is specified.*
- LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table, and the matched rows from the right table; i.e., the results will contain *all* records from the left table, even if the JOIN condition doesn’t find any matching records in the right table. This means that if the ON clause doesn’t match any records in the right table, the JOIN will still return a row in the result for that record in the left table, but with NULL in each column from the right table.
- RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table, and the matched rows from the left table. This is the exact opposite of a LEFT JOIN; i.e., the results will contain *all* records from the right table, even if the JOIN condition doesn’t find any matching records in the left table. This means that if the ON clause doesn’t match any records in the left table, the JOIN will still return a row in the result for that record in the right table, but with NULL in each column from the left table.

- FULL JOIN (or FULL OUTER JOIN): Returns all rows for which there is a match in EITHER of the tables. Conceptually, a FULL JOIN combines the effect of applying both a LEFT JOIN and a RIGHT JOIN; i.e., its result set is equivalent to performing a UNION of the results of left and right outer queries.
- CROSS JOIN: Returns all records where each row from the first table is combined with each row from the second table (i.e., returns the Cartesian product of the sets of rows from the joined tables). Note that a CROSS JOIN can either be specified using the CROSS JOIN syntax ("explicit join notation") or (b) listing the tables in the FROM clause separated by commas without using a WHERE clause to supply join criteria ("implicit join notation").

3. Given the following tables:

```
sql> SELECT * FROM runners;
+----+--------------+
| id | name         |
+----+--------------+
|  1 | John Doe     |
|  2 | Jane Doe     |
|  3 | Alice Jones  |
|  4 | Bobby Louis  |
|  5 | Lisa Romero  |
+----+--------------+
```

```
sql> SELECT * FROM races;
+----+----------------+-----------+
| id | event          | winner_id |
+----+----------------+-----------+
|  1 | 100 meter dash |  2        |
|  2 | 500 meter dash |  3        |
|  3 | cross-country  |  2        |
|  4 | triathalon     |  NULL     |
+----+----------------+-----------+
```

4. What will be the result of the query below?

```
SELECT * FROM runners WHERE id NOT IN (SELECT winner_id FROM races)
```

Explain your answer and also provide an alternative version of this query that will avoid the issue that it exposes.

Surprisingly, given the sample data provided, the result of this query will be an empty set. The reason for this is as follows: If the set being evaluated by the SQL NOT IN condition contains *any* values that are null, then the outer query here will return an empty set, even if there are many runner ids that match winner_ids in the races table.

Knowing this, a query that avoids this issue would be as follows:

```
SELECT * FROM runners WHERE id NOT IN (SELECT winner_id FROM races WHERE
winner_id IS NOT null)
```

Note, this is assuming the standard SQL behaviour that you get without modifying the default ANSI_NULLS setting.

5. How to select UNIQUE records from a table using a SQL Query?

Consider below EMPLOYEE table as the source data

```
CREATE TABLE EMPLOYEE (

          EMPLOYEE_ID NUMBER(6,0),

          NAME VARCHAR2(20),

          SALARY NUMBER(8,2)

);



INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY)
VALUES(100,'Jennifer',4400);

INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY)
VALUES(100,'Jennifer',4400);
```

```
INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY)
VALUES(101,'Michael',13000);

INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY)
VALUES(101,'Michael',13000);

INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY)
VALUES(101,'Michael',13000);

INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY) VALUES(102,'Pat',6000);

INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY) VALUES(102,'Pat',6000);

INSERT INTO EMPLOYEE(EMPLOYEE_ID,NAME,SALARY)
VALUES(103,'Den',11000);

SELECT * FROM EMPLOYEE;
```

EMPLOYEE_ID	**NAME**	**SALARY**
100	Jennifer	4400
100	Jennifer	4400
101	Michael	13000
101	Michael	13000
101	Michael	13000

102	Pat	6000
102	Pat	6000
103	Den	11000

METHOD-1: Using GROUP BY Function

GROUP BY clause is used with a SELECT statement to collect data from multiple records and group the results by one or more columns. The GROUP BY clause returns one row per group. By applying GROUP BY function on all the source columns, unique records can be queried from the table.

Below is the query to fetch the unique records using GROUP BY function.

Query:

```
SELECT EMPLOYEE_ID,

    NAME,

    SALARY

FROM EMPLOYEE

GROUP BY EMPLOYEE_ID, NAME, SALARY;
```

Result:

EMPLOYEE_ID	**NAME**	**SALARY**

100	Jennifer	4400
101	Michael	13000
102	Pat	6000
103	Den	11000

METHOD-2: Using ROW_NUMBER Analytic Function

The ROW_NUMBER Analytic function is used to provide consecutive numbering of the rows in the result by the ORDER selected for each PARTITION specified in the OVER clause. It will assign the value 1 for the first row and increase the number of the subsequent rows.

Using ROW_NUMBER Analytic function, assign row numbers to each unique set of records.

Query:

```
SELECT EMPLOYEE_ID,

    NAME,

    SALARY,

          ROW_NUMBER() OVER(PARTITION BY EMPLOYEE_ID,NAME,SALARY
ORDER BY EMPLOYEE_ID) AS ROW_NUMBER

FROM EMPLOYEE;
```

Result:

EMPLOYEE_ID	NAME	SALARY	ROW_NUMBER
100	Jennifer	4400	1
100	Jennifer	4400	2
101	Michael	13000	1
101	Michael	13000	2
101	Michael	13000	3
102	Pat	6000	1
102	Pat	6000	2
103	Den	11000	1

Once row numbers are assigned, by querying the rows with row number 1 will give the unique records from the table.

Query:

```
SELECT EMPLOYEE_ID, NAME, SALARY

FROM( SELECT

    EMPLOYEE_ID,

    NAME,
      SALARY,

            ROW_NUMBER()  OVER(PARTITION  BY  EMPLOYEE_ID,NAME,SALARY
ORDER BY EMPLOYEE_ID) AS ROW_NUMBER

    FROM EMPLOYEE)

WHERE ROW_NUMBER = 1;
```

Result:

EMPLOYEE_ID	**NAME**	**SALARY**
101	Michael	13000
100	Jennifer	4400
102	Pat	6000
103	Den	11000

6. How to delete DUPLICATE records from a table using a SQL Query?

Consider the same EMPLOYEE table as source discussed in previous question

STEP-1: Using ROW_NUMBER Analytic function, assign row numbers to each unique set of records. Select ROWID of the rows along with the source columns

Query:

```
SELECT ROWID,
    EMPLOYEE_ID,

    NAME,SALARY,

            ROW_NUMBER() OVER(PARTITION BY EMPLOYEE_ID,NAME,SALARY
ORDER BY EMPLOYEE_ID) AS ROW_NUMBER

FROM EMPLOYEE;
```

Result:

ROWID	**EMPLOYEE_ID**	**NAME**	**SALARY**	**ROW_NUMBER**
AAASnBAAEAAACrWAAA	100	Jennifer	4400	1
AAASnBAAEAAACrWAAB	100	Jennifer	4400	2
AAASnBAAEAAACrWAAC	101	Michael	13000	1
AAASnBAAEAAACrWAAD	101	Michael	13000	2
AAASnBAAEAAACrWAAE	101	Michael	13000	3
AAASnBAAEAAACrWAAF	102	Pat	6000	1
AAASnBAAEAAACrWAAG	102	Pat	6000	2
AAASnBAAEAAACrWAAH	103	Den	11000	1

STEP-2: Select ROWID of records with ROW_NUMBER > 1

Query:

```
SELECT ROWID FROM(
    SELECT ROWID,
         EMPLOYEE_ID,
         NAME,
         SALARY,
              ROW_NUMBER() OVER(PARTITION BY EMPLOYEE_ID,NAME,SALARY ORDER BY EMPLOYEE_ID) AS ROW_NUMBER
    FROM EMPLOYEE)
WHERE ROW_NUMBER > 1;
```

Result:

ROWID
AAASnBAAEAAACrWAAB
AAASnBAAEAAACrWAAD
AAASnBAAEAAACrWAAE
AAASnBAAEAAACrWAAG

STEP-3: Delete the records from the source table using the ROWID values fetched in previous step

Query:

```
DELETE FROM EMP WHERE ROWID IN (

   SELECT ROWID FROM(

      SELECT ROWID,

             ROW_NUMBER()  OVER(PARTITION  BY  EMPLOYEE_ID,NAME,SALARY
ORDER BY EMPLOYEE_ID) AS ROW_NUMBER

   FROM EMPLOYEE)

WHERE ROW_NUMBER > 1);
```

Result:

The table EMPLOYEE will have below records after deleting the duplicates

ROWID	**EMPLOYEE_ID**	**NAME**	**SALARY**
AAASnBAAEAAACrWAAA	100	Jennifer	4400
AAASnBAAEAAACrWAAC	101	Michael	13000
AAASnBAAEAAACrWAAF	102	Pat	6000
AAASnBAAEAAACrWAAH	103	Den	11000

METHOD-2: Using ROWID and Correlated subquery

Correlated subquery is used for row-by-row processing. With a normal nested subquery, the inner SELECT query runs once and executes first. The returning values will be used by the main query. A correlated subquery, however, executes once for every row of the outer query. In other words, the inner query is driven by the outer query.

In the below query, we are comparing the ROWIDs' of the unique set of records and keeping the record with MIN ROWID and deleting all other rows.

Query:

DELETE FROM EMPLOYEE A WHERE ROWID > (SELECT MIN(ROWID) FROM EMPLOYEE B WHERE B.EMPLOYEE_ID = A.EMPLOYEE_ID);

Result:

The table EMPLOYEE will have below records after deleting the duplicates

ROWID	**EMPLOYEE_ID**	**NAME**	**SALARY**
AAASnBAAEAAACrWAAA	100	Jennifer	4400
AAASnBAAEAAACrWAAC	101	Michael	13000
AAASnBAAEAAACrWAAF	102	Pat	6000
AAASnBAAEAAACrWAAH	103	Den	11000

The opposite of above discussed case can be implemented by keeping the record with MAX ROWID from the unique set of records and delete all other duplicates by executing below query.

Query:

DELETE FROM EMPLOYEE A WHERE ROWID < (SELECT MAX(ROWID) FROM EMPLOYEE B WHERE B.EMPLOYEE_ID = A.EMPLOYEE_ID);

Result:

The table EMPLOYEE will have below records after deleting the duplicates

ROWID	EMPLOYEE_ID	NAME	SALARY
AAASnBAAEAAACrWAAA	100	Jennifer	4400
AAASnBAAEAAACrWAAC	101	Michael	13000
AAASnBAAEAAACrWAAF	102	Pat	6000
AAASnBAAEAAACrWAAH	103	Den	11000

---

7. How to read TOP 5 records from a table using a SQL query?

Consider below table DEPARTMENTS as the source data

```
CREATE TABLE Departments(

  Department_ID number,

  Department_Name varchar(50);
```

```sql
INSERT INTO DEPARTMENTS VALUES('10','Administration');

INSERT INTO DEPARTMENTS VALUES('20','Marketing');

INSERT INTO DEPARTMENTS VALUES('30','Purchasing');

INSERT INTO DEPARTMENTS VALUES('40','Human Resources');

INSERT INTO DEPARTMENTS VALUES('50','Shipping');

INSERT INTO DEPARTMENTS VALUES('60','IT');

INSERT INTO DEPARTMENTS VALUES('70','Public Relations');

INSERT INTO DEPARTMENTS VALUES('80','Sales');

SELECT * FROM Departments;
```

DEPARTMENT_ID	**DEPARTMENT_NAME**
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping

60	IT
70	Public Relations
80	Sales

ROWNUM is a “Pseudocolumn” that assigns a number to each row returned by a query indicating the order in which Oracle selects the row from a table. The first row selected has a ROWNUM of 1, the second has 2, and so on.

Query:

```
SELECT * FROM Departments WHERE ROWNUM <= 5;
```

Result:

DEPARTMENT_ID	**DEPARTMENT_NAME**
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping

8. How to read LAST 5 records from a table using a SQL query?

Consider the same DEPARTMENTS table as the source discussed in the previous question.

In order to select the last 5 records we need to find (**count of total number of records – 5***)* which gives the count of records from first to last but 5 records. Using the MINUS function we can compare **all records from DEPARTMENTS table** with **records from first to last but 5 from DEPARTMENTS table** which give the last 5 records of the table as result.

MINUS operator is used to return all rows in the first SELECT statement that are not present in the second SELECT statement.

Query:

```
SELECT * FROM Departments

MINUS

SELECT * FROM Departments WHERE ROWNUM <= (SELECT COUNT(*)-5 FROM Departments);
```

Result:

DEPARTMENT_ID	DEPARTMENT_NAME
40	Human Resources
50	Shipping

60	IT
70	Public Relations
80	Sales

9. What is the result of Normal Join, Left Outer Join, Right Outer Join and Full Outer Join between the tables A & B?

Table_A

COL
1
1
0
null

Table_B

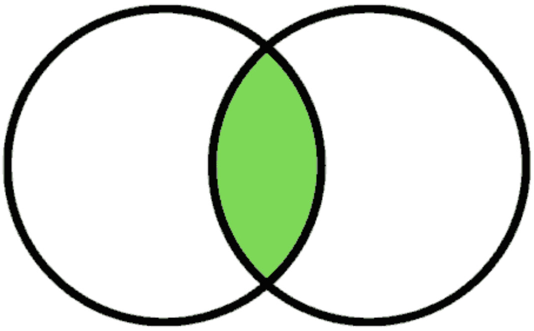
COL
1

0
null
null

Normal Join:

Normal Join or Inner Join is the most common type of join. It returns the rows that are exact match between both the tables.

The following Venn diagram illustrates a Normal join when combining two result sets:



Query:

```
SELECT a.COL as A,

     b.COL as B

FROM TABLE_A a JOIN TABLE_B b
```

ON a.COL = b.COL;

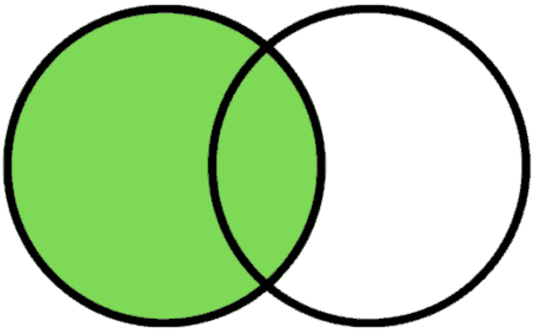
Result:

A	**B**
1	1
1	1
0	0

Left Outer Join:

The Left Outer Join returns all the rows from the left table and only the matching rows from the right table. If there is no matching row found from the right table, the left outer join will have NULL values for the columns from the right table.

The following Venn diagram illustrates a Left join when combining two result sets:



Query:

```
SELECT a.COL as A,

       b.COL as B

FROM TABLE_A a LEFT OUTER JOIN TABLE_B b

ON a.COL = b.COL;
```

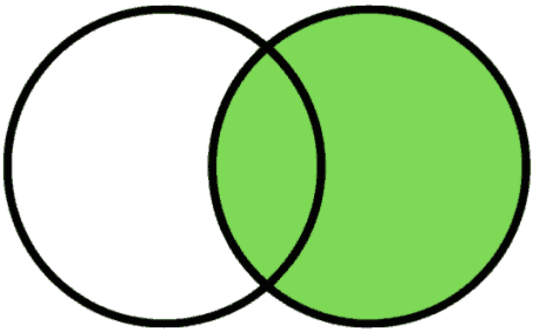
Result:

A	**B**
1	1
1	1
0	0
NULL	NULL

Right Outer Join:

The Right Outer Join returns all the rows from the right table and only the matching rows from the left table. If there is no matching row found from the left table, the right outer join will have NULL values for the columns from the left table.

The following Venn diagram illustrates a Right join when combining two result sets:



Query:

SELECT a.COL as A,

b.COL as B

FROM TABLE_A a RIGHT OUTER JOIN TABLE_B b

ON a.COL = b.COL;

Result:

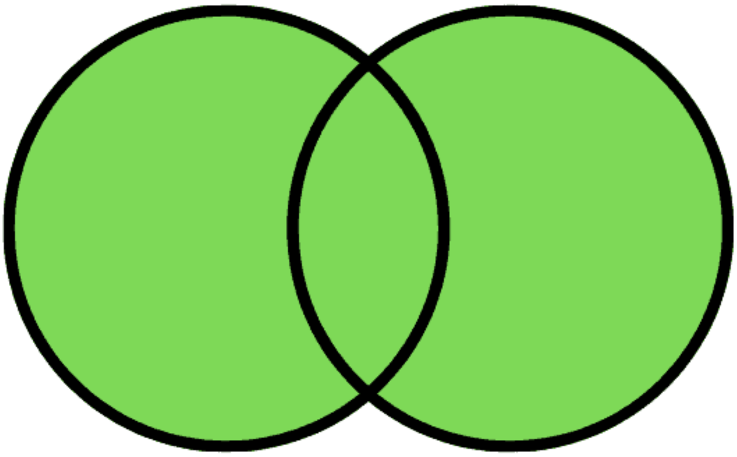
A	**B**
1	1
1	1
0	0

NULL	NULL
NULL	NULL

Full Outer Join:

The Full Outer Join returns all the rows from both the right table and the left table. If there is no matching row found, the missing side columns will have NULL values.

The following Venn diagram illustrates a Full join when combining two result sets:



Query:

```
SELECT a.COL as A,

       b.COL as B

FROM TABLE_A a FULL OUTER JOIN TABLE_B b

ON a.COL = b.COL;
```

Result:

A	**B**
1	1
1	1
0	0
NULL	NULL
NULL	NULL
NULL	NULL

NOTE: *NULL do not match with NULL*

10. How to find the employee with a second MAX Salary using a SQL query?

Consider below EMPLOYEES table as the source data

```
CREATE TABLE Employees(

          EMPLOYEE_ID NUMBER(6,0),

          NAME VARCHAR2(20 BYTE),

          SALARY NUMBER(8,2)

  );
```

```
INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(100,'Jennifer',4400);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(101,'Michael',13000);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(102,'Pat',6000);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(103,'Den', 11000);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(104,'Alexander',3100);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(105,'Shelli',2900);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(106,'Sigal',2800);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(107,'Guy',2600);

INSERT INTO EMPLOYEES(EMPLOYEE_ID,NAME,SALARY) VALUES(108,'Karen',2500);

SELECT * FROM Employees;
```

EMPLOYEE_ID	**NAME**	**SALARY**
100	Jennifer	4400

101	Michael	13000
102	Pat	6000
103	Den	11000
104	Alexander	3100
105	Shelli	2900
106	Sigel	2800
107	Guy	2600
108	Karen	2500

METHOD-1: Without using SQL Analytic Functions

In order to find the second MAX salary, employee records with MAX salary need to be eliminated. It can be achieved by using the below SQL query.

Query:

```
SELECT MAX(salary) AS salary FROM Employees WHERE salary NOT IN (

SELECT MAX(salary) AS salary FROM Employees);
```

Result:

SALARY
11000

The above query only gives the second MAX salary value. In order to fetch the entire employee record with the second MAX salary we need to do a self-join on the Employee table based on Salary value.

Query:

```
WITH

TEMP AS(

  SELECT MAX(salary) AS salary FROM Employees WHERE salary NOT IN (

  SELECT MAX(salary) AS salary FROM Employees)

)

SELECT a.* FROM Employees a JOIN TEMP b on a.salary = b.salary
```

Result:

EMPLOYEE_ID	NAME	SALARY
103	Den	11000

METHOD-2: Using SQL Analytic Functions

Query:

The DENSE_RANK is an analytic function that calculates the rank of a row in an ordered set of rows starting from 1. Unlike the RANK function, the DENSE_RANK function returns rank values as consecutive integers.

```
SELECT Employee_Id,
      Name,
      Salary
FROM(
  SELECT Employees.*,
        DENSE_RANK() OVER(ORDER BY Salary DESC) as SALARY_RANK
  FROM Employees)
WHERE SALARY_RANK =2
```

Result:

EMPLOYEE_ID	**NAME**	**SALARY**
103	Den	11000

By replacing the value of SALARY_RANK, any highest salary rank can be found easily.

11. How to find the employee with the third MAX Salary using a SQL query without using Analytic Functions?

Consider the same EMPLOYEES table as source discussed in previous question

In order to find the third MAX salary, we need to eliminate the top 2 salary records. But we cannot use the same method we used for finding a second MAX salary (not a best practice). Imagine if we have to find the fifth MAX salary. We should not be writing a query with four nested sub queries.

STEP-1:

The approach here is to first list all the records based on Salary in the descending order with MAX salary on top and MIN salary at bottom. Next, using ROWNUM select the top 2 records.

Query:

```
SELECT salary FROM(

SELECT salary FROM Employees ORDER BY salary DESC)

WHERE ROWNUM < 3;
```

Result:

Salary
13000
11000

STEP-2:

Next find the MAX salary from the EMPLOYEE table which is not one of top two salary values fetched in the earlier step.

Query:

```
SELECT MAX(salary) as salary FROM Employees WHERE salary NOT IN (

   SELECT salary FROM(

   SELECT salary FROM Employees ORDER BY salary DESC)

   WHERE ROWNUM < 3

);
```

Result:

SALARY
6000

STEP-3:

In order to fetch the entire employee record with the third MAX salary we need to do a self-join on the Employee table based on Salary value.

Query:

```
WITH

TEMP AS(

 SELECT MAX(salary) as salary FROM Employees WHERE salary NOT IN (

    SELECT salary FROM(

    SELECT salary FROM Employees ORDER BY salary DESC)

    WHERE ROWNUM < 3)
)

SELECT a.* FROM Employees a join TEMP b on a.salary = b.salary
```

Result:

EMPLOYEE_ID	NAME	SALARY
102	Pat	6000

Danny wants to use the data to answer a few simple questions about his customers, especially about their visiting patterns, how much money they've spent and also which menu items are their favourite. Having this deeper connection with his customers will help him deliver a better and more personalized experience for his loyal customers.

He plans on using these insights to help him decide whether he should expand the existing customer loyalty program - additionally he needs help to generate some basic datasets so his team can easily inspect the data without needing to use SQL.

Danny has provided you with a sample of his overall customer data due to privacy issues - but he hopes that these examples are enough for you to write fully functioning SQL queries to help him answer his questions!

Danny has shared with you 3 key datasets for this case study:

- `sales`
- `menu`
- `members`

You can inspect the entity relationship diagram and example data below.

Example Datasets

Table 1: sales

The `sales` table captures all `customer_id` level purchases with an corresponding `order_date` and `product_id` information for when and what menu items were ordered.

customer_id	order_date	product_id
A	2021-01-01	1
A	2021-01-01	2
A	2021-01-07	2
A	2021-01-10	3
A	2021-01-11	3
A	2021-01-11	3
B	2021-01-01	2
B	2021-01-02	2
B	2021-01-04	1
B	2021-01-11	1
B	2021-01-16	3
B	2021-02-01	3
C	2021-01-01	3
C	2021-01-01	3
C	2021-01-07	3

Table 2: menu

The menu table maps the product_id to the actual product_name and price of each menu item.

product_id	product_name	price
1	sushi	10
2	curry	15
3	ramen	12

Table 3: members

The final members table captures the join_date when a customer_id joined the beta version of the Danny’s Diner loyalty program.

customer_id	join_date
A	2021-01-07
B	2021-01-09

Each of the following case study questions can be answered using a single SQL statement:

12.1. What is the total amount each customer spent at the restaurant?

12.2. How many days has each customer visited the restaurant?

12.3. What was the first item from the menu purchased by each customer?

12.4. What is the most purchased item on the menu and how many times was it purchased by all customers?

12.5 Which item was the most popular for each customer?

```
-- SOLUTIONS

--1 Total amount spend by each customer
Select S.customer_id, Sum(M.price)
From Menu m
join Sales s
On m.product_id = s.product_id
group by S.customer_id

--2 How manu dats customer visited the restauraunt
Select customer_id, count(distinct(order_date))
From Sales
Group by customer_id

-- 3. What was the first item from the menu purchased by each customer?

With Rank as
(
Select S.customer_id,
       M.product_name,
       S.order_date,
       DENSE_RANK() OVER (PARTITION BY S.Customer_ID Order by S.order_date) as rank
From Menu m
join Sales s
On m.product_id = s.product_id
group by S.customer_id, M.product_name,S.order_date
)
Select Customer_id, product_name
From Rank
Where rank = 1
```

```sql
-- 4. What is the most purchased item on the menu and how many times was it purchased by all customers?

Select Top 1 M.product_name , Count(S.product_id)
From Menu m
join Sales s
On m.product_id = s.product_id
Group by M.product_name
Order by Count(S.product_id) desc

-- 5. Which item was the most popular for each customer?
With rank as
(
Select S.customer_ID ,
       M.product_name,
       Count(S.product_id) as Count,
       Dense_rank()  Over (Partition by S.Customer_ID order by Count(S.product_id) DESC ) as Rank
From Menu m
join Sales s
On m.product_id = s.product_id
group by S.customer_id,S.product_id,M.product_name
)
Select Customer_id,Product_name,Count
From rank
where rank = 1
```

13. Did you know that over **115 million kilograms** of pizza is consumed daily worldwide???

Danny was scrolling through his Instagram feed when something really caught his eye - “80s Retro Styling and Pizza Is The Future!”

Danny was sold on the idea, but he knew that pizza alone was not going to help him get seed funding to expand his new Pizza Empire - so he had one more genius idea to combine with it - he was going to *Uberize* it - and so Pizza Runner was launched!

Danny started by recruiting “runners” to deliver fresh pizza from Pizza Runner Headquarters (otherwise known as Danny’s house) and also maxed out his credit card to pay freelance developers to build a mobile app to accept orders from customers.

Table 1: runners

The `runners` table shows the `registration_date` for each new runner

runner_id	**registration_date**

1	2021-01-01
2	2021-01-03
3	2021-01-08
4	2021-01-15

Table 2: customer_orders

Customer pizza orders are captured in the `customer_orders` table with 1 row for each individual pizza that is part of the order.

The `pizza_id` relates to the type of pizza which was ordered whilst the `exclusions` are the `ingredient_id` values which should be removed from the pizza and the `extras` are the `ingredient_id` values which need to be added to the pizza.

Note that customers can order multiple pizzas in a single order with varying `exclusions` and `extras` values even if the pizza is the same type!

The `exclusions` and `extras` columns will need to be cleaned up before using them in your queries.

order_id	**customer_id**	**pizza_id**	**exclusions**	**extras**	**order_time**
1	101	1			2021-01-01 18:05:02
2	101	1			2021-01-01 19:00:52

3	102	1			2021-01-02 23:51:23
3	102	2		NaN	2021-01-02 23:51:23
4	103	1	4		2021-01-04 13:23:46
4	103	1	4		2021-01-04 13:23:46
4	103	2	4		2021-01-04 13:23:46
5	104	1	null	1	2021-01-08 21:00:29
6	101	2	null	null	2021-01-08 21:03:13
7	105	2	null	1	2021-01-08 21:20:29
8	102	1	null	null	2021-01-09 23:54:33
9	103	1	4	1, 5	2021-01-10 11:22:59
10	104	1	null	null	2021-01-11 18:34:49
10	104	1	2, 6	1, 4	2021-01-11 18:34:49

Table 3: runner_orders

After each order is received through the system - they are assigned to a runner - however not all orders are fully completed and can be cancelled by the restaurant or the customer.

The `pickup_time` is the timestamp at which the runner arrives at the Pizza Runner headquarters to pick up the freshly cooked pizzas. The `distance` and `duration` fields are related to how far and long the runner had to travel to deliver the order to the respective customer.

There are some known data issues with this table so be careful when using this in your queries - make sure to check the data types for each column in the schema SQL!

order_id	runner_id	pickup_time	distance	duration	cancellation
1	1	2021-01-01 18:15:34	20km	32 minutes	
2	1	2021-01-01 19:10:54	20km	27 minutes	
3	1	2021-01-03 00:12:37	13.4km	20 mins	NaN
4	2	2021-01-04 13:53:03	23.4	40	NaN
5	3	2021-01-08 21:10:57	10	15	NaN
6	3	null	null	null	Restaurant Cancellation
7	2	2020-01-08 21:30:45	25km	25 mins	null

8	2	2020-01-10 00:15:02	23.4 km	15 minute	null
9	2	null	null	null	Customer Cancellation
10	1	2020-01-11 18:50:20	10km	10 minutes	null

Table 4: pizza_names

At the moment - Pizza Runner only has 2 pizzas available: the Meat Lovers or Vegetarian!

pizza_id	**pizza_name**
1	Meat Lovers
2	Vegetarian

Table 5: pizza_recipes

Each `pizza_id` has a standard set of `toppings` which are used as part of the pizza recipe.

pizza_id	**toppings**
1	1, 2, 3, 4, 5, 6, 8, 10
2	4, 6, 7, 9, 11, 12

Table 6: pizza_toppings

This table contains all of the `topping_name` values with their corresponding `topping_id` value

topping_id	topping_name
1	Bacon
2	BBQ Sauce
3	Beef
4	Cheese
5	Chicken
6	Mushrooms
7	Onions
8	Pepperoni
9	Peppers
10	Salami
11	Tomatoes
12	Tomato Sauce

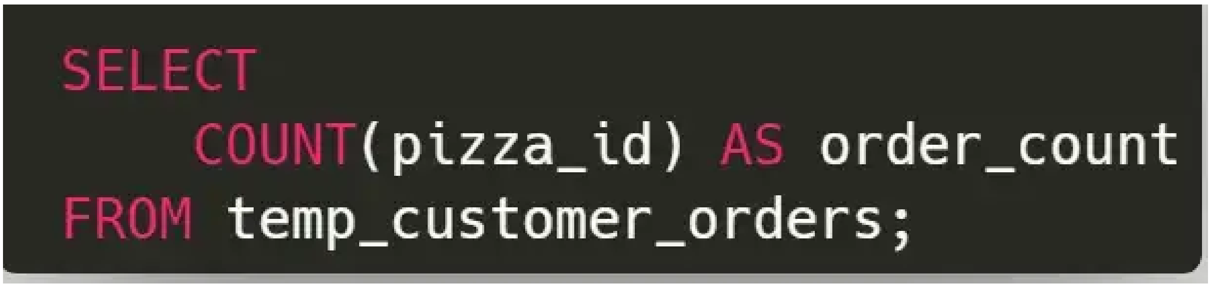
Case Study Questions

This case study has LOTS of questions - they are broken up by area of focus including:

- Pizza Metrics
- Runner and Customer Experience
- Ingredient Optimisation
- Pricing and Ratings
- Bonus DML Challenges (DML = Data Manipulation Language)

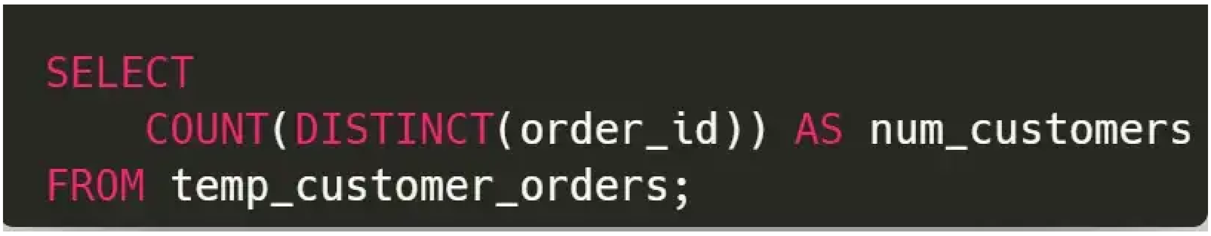
Pizza Metrics

13.1. How many pizzas were ordered?



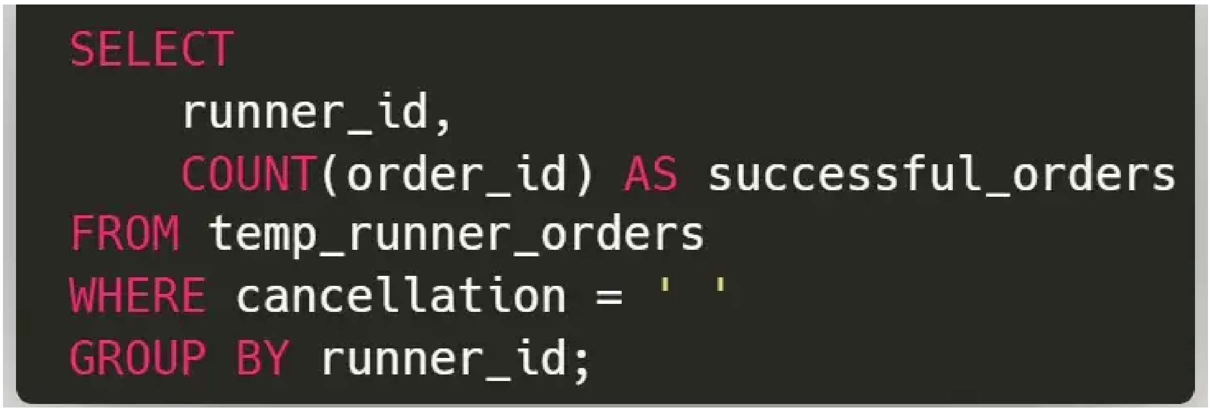
```sql
SELECT
    COUNT(pizza_id) AS order_count
FROM temp_customer_orders;
```

13.2. How many unique customer orders were made?



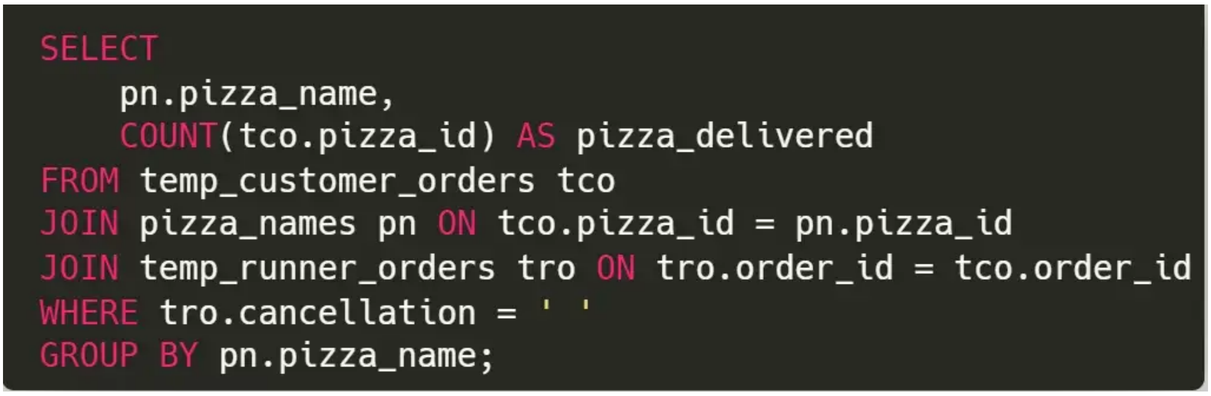
```sql
SELECT
    COUNT(DISTINCT(order_id)) AS num_customers
FROM temp_customer_orders;
```

13.3. How many successful orders were delivered by each runner?



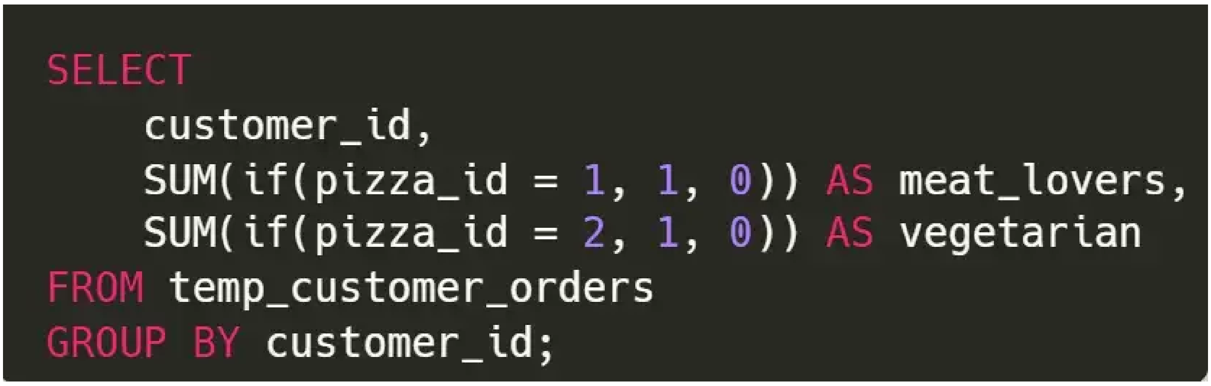
```sql
SELECT
    runner_id,
    COUNT(order_id) AS successful_orders
FROM temp_runner_orders
WHERE cancellation = ' '
GROUP BY runner_id;
```

13.4. How many of each type of pizza was delivered?



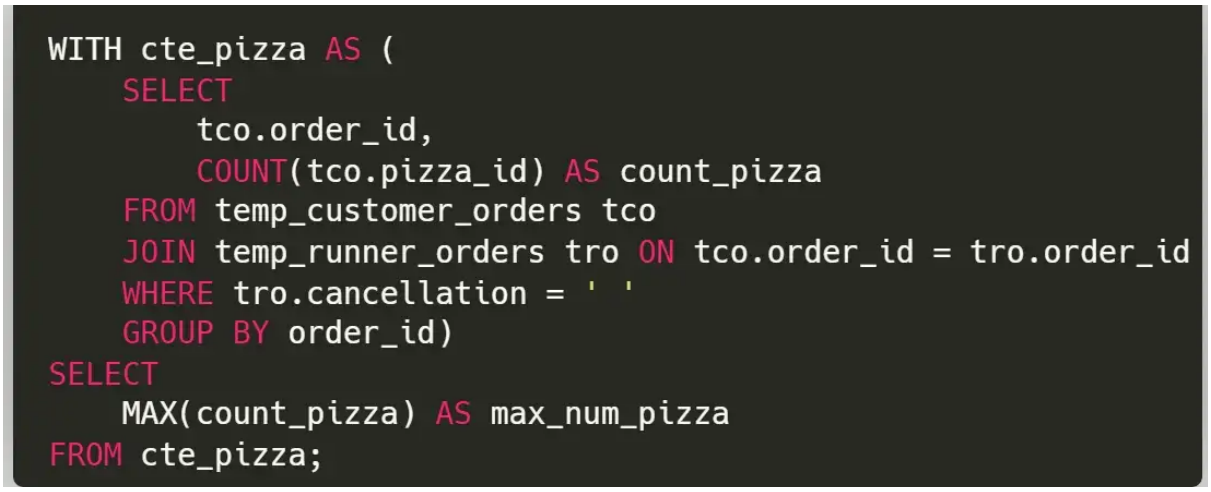
```sql
SELECT
    pn.pizza_name,
    COUNT(tco.pizza_id) AS pizza_delivered
FROM temp_customer_orders tco
JOIN pizza_names pn ON tco.pizza_id = pn.pizza_id
JOIN temp_runner_orders tro ON tro.order_id = tco.order_id
WHERE tro.cancellation = ' '
GROUP BY pn.pizza_name;
```

13.5. How many Vegetarian and Meatlovers were ordered by each customer?



```sql
SELECT
    customer_id,
    SUM(if(pizza_id = 1, 1, 0)) AS meat_lovers,
    SUM(if(pizza_id = 2, 1, 0)) AS vegetarian
FROM temp_customer_orders
GROUP BY customer_id;
```

13.6. What was the maximum number of pizzas delivered in a single order?



```sql
WITH cte_pizza AS (
    SELECT
        tco.order_id,
        COUNT(tco.pizza_id) AS count_pizza
    FROM temp_customer_orders tco
    JOIN temp_runner_orders tro ON tco.order_id = tro.order_id
    WHERE tro.cancellation = ' '
    GROUP BY order_id)
SELECT
    MAX(count_pizza) AS max_num_pizza
FROM cte_pizza;
```

14. Subscription based businesses are super popular and Danny realised that there was a large gap in the market - he wanted to create a new streaming service that only had food related content - something like Netflix but with only cooking shows!

Danny finds a few smart friends to launch his new startup Foodie-Fi in 2020 and starts selling monthly and annual subscriptions, giving their customers unlimited on-demand access to exclusive food videos from around the world!

Danny created Foodie-Fi with a data driven mindset and wanted to ensure all future investment decisions and new features were decided using data. This case study focuses on using subscription style digital data to answer important business questions.

Table 1: plans

Customers can choose which plans to join Foodie-Fi when they first sign up.

Basic plan customers have limited access and can only stream their videos and is only available monthly at $9.90

Pro plan customers have no watch time limits and are able to download videos for offline viewing. Pro plans start at $19.90 a month or $199 for an annual subscription.

Customers who sign up to an initial 7 day free trial will automatically continue with the pro monthly subscription plan unless they cancel, downgrade to basic or upgrade to an annual pro plan at any point during the trial.

When customers cancel their Foodie-Fi service - they will have a `churn` plan record with a `null` price but their plan will continue until the end of the billing period.

plan_id	plan_name	price
0	trial	0
1	basic monthly	9.90
2	pro monthly	19.90

3	pro annual	199
4	churn	null

Table 2: subscriptions

Customer subscriptions show the exact date where their specific `plan_id` starts.

If customers downgrade from a pro plan or cancel their subscription - the higher plan will remain in place until the period is over - the `start_date` in the `subscriptions` table will reflect the date that the actual plan changes.

When customers upgrade their account from a basic plan to a pro or annual pro plan - the higher plan will take effect straight away.

When customers churn - they will keep their access until the end of their current billing period but the `start_date` will be technically the day they decided to cancel their service.

customer_id	plan_id	start_date
1	0	2020-08-01
1	1	2020-08-08
2	0	2020-09-20
2	3	2020-09-27
11	0	2020-11-19
11	4	2020-11-26
13	0	2020-12-15
13	1	2020-12-22
13	2	2021-03-29

15	0	2020-03-17
15	2	2020-03-24
15	4	2020-04-29
16	0	2020-05-31
16	1	2020-06-07
16	3	2020-10-21
18	0	2020-07-06
18	2	2020-07-13
19	0	2020-06-22
19	2	2020-06-29
19	3	2020-08-29

Data Analysis Questions

1. Based on the 8 sample customers provided in the sample from the `subscriptions` table, write a brief description about each customer's onboarding journey.

```
SELECT
 s.customer_id,
 f.plan_id,
 f.plan_name,
 s.start_date
FROM foodie_fi.plans AS f
JOIN foodie_fi.subscriptions AS s
```

```
 ON f.plan_id = s.plan_id

WHERE

 s.customer_id IN (1,2,11,13,15,16,18,19) -- selected 8 customers
```

2. How many customers has Foodie-Fi ever had?

```
SELECT

 COUNT(DISTINCT customer_id) AS unique_customer

FROM foodie_fi.subscriptions;
```

3. What is the monthly distribution of `trial` plan `start_date` values for our dataset - use the start of the month as the group by value.

```
SELECT

  DATE_PART('month',start_date) AS month_date, -- Cast month as integer

  TO_CHAR(start_date, 'Month') AS month_name, -- Cast month as string

 COUNT(*) AS trial_subscriptions

FROM foodie_fi.subscriptions s

JOIN foodie_fi.plans p

 ON s.plan_id = p.plan_id
```

```
WHERE s.plan_id = 0

GROUP BY DATE_PART('month',start_date),

 TO_CHAR(start_date, 'Month')

ORDER BY month_date ASC;
```

4. What plan `start_date` values occur after the year 2020 for our dataset? Show the breakdown by count of events for each `plan_name`

```
SELECT

 p.plan_id,

 p.plan_name,

 COUNT(*) AS events

FROM foodie_fi.subscriptions s

JOIN foodie_fi.plans p

 ON s.plan_id = p.plan_id

WHERE s.start_date >= '2021-01-01'

GROUP BY p.plan_id, p.plan_name

ORDER BY p.plan_id;
```

5. What is the customer count and percentage of customers who have churned rounded to 1 decimal place?

```
SELECT
 COUNT(*) AS churn_count,
 ROUND(100 * COUNT(*)::NUMERIC / (
   SELECT COUNT(DISTINCT customer_id)
   FROM foodie_fi.subscriptions),1) AS churn_percentage
FROM foodie_fi.subscriptions s
JOIN foodie_fi.plans p
 ON s.plan_id = p.plan_id
WHERE s.plan_id = 4;
```

6. How many customers have churned straight after their initial free trial - what percentage is this rounded to the nearest whole number?

```
-- Find ranking of plans by customer and plan type
WITH ranking AS (
SELECT
 s.customer_id,
 s.plan_id,
 p.plan_name,
 -- Run a ROW_NUMBER() to rank plans from 0 to 4
 ROW_NUMBER() OVER (
   PARTITION BY s.customer_id
   ORDER BY s.plan_id) AS plan_rank
FROM foodie_fi.subscriptions s
JOIN foodie_fi.plans p
 ON s.plan_id = p.plan_id)

SELECT
 COUNT(*) AS churn_count,
 ROUND(100 * COUNT(*) / (
   SELECT COUNT(DISTINCT customer_id)
   FROM foodie_fi.subscriptions),0) AS churn_percentage
FROM ranking
WHERE plan_id = 4 -- Filter to churn plan
 AND plan_rank = 2 -- Filter to rank 2 as customers who churned immediately after trial have churn plan ranked as 2
```

7. What is the number and percentage of customer plans after their initial free trial?

```
-- To retrieve next plan's start date located in the next row based on current row

WITH next_plan_cte AS (

SELECT

 customer_id,

 plan_id,

 LEAD(plan_id, 1) OVER( -- Offset by 1 to retrieve the immediate row's value below

   PARTITION BY customer_id

   ORDER BY plan_id) as next_plan

FROM foodie_fi.subscriptions)



SELECT

 next_plan,

 COUNT(*) AS conversions,

 ROUND(100 * COUNT(*)::NUMERIC / (
   SELECT COUNT(DISTINCT customer_id)

FROM foodie_fi.subscriptions),1) AS conversion_percentage

FROM next_plan_cte

WHERE next_plan IS NOT NULL

 AND plan_id = 0

GROUP BY next_plan

ORDER BY next_plan;
```

8. What is the customer count and percentage breakdown of all 5 `plan_name` values at `2020-12-31`?

```
-- Retrieve next plan's start date located in the next row based on current row

WITH next_plan AS(

SELECT

 customer_id,

 plan_id,

 start_date,

  LEAD(start_date, 1) OVER(PARTITION BY customer_id ORDER BY start_date) as next_date
```

```
FROM foodie_fi.subscriptions
WHERE start_date <= '2020-12-31'),
-- Find customer breakdown with existing plans on or after 31 Dec 2020
customer_breakdown AS (
 SELECT
   plan_id,
   COUNT(DISTINCT customer_id) AS customers
 FROM next_plan
 WHERE
   (next_date IS NOT NULL AND (start_date < '2020-12-31'
     AND next_date > '2020-12-31'))
   OR (next_date IS NULL AND start_date < '2020-12-31')
 GROUP BY plan_id)
SELECT plan_id, customers,
 ROUND(100 * customers::NUMERIC / (
   SELECT COUNT(DISTINCT customer_id)
   FROM foodie_fi.subscriptions),1) AS percentage

FROM customer_breakdown

GROUP BY plan_id, customers

ORDER BY plan_id;
```

9. How many customers have upgraded to an annual plan in 2020?

```
SELECT

 COUNT(DISTINCT customer_id) AS unique_customer

FROM foodie_fi.subscriptions

WHERE plan_id = 3

 AND start_date <= '2020-12-31';
```

10. How many days on average does it take for a customer to an annual plan from the day they join Foodie-Fi?

```
-- Filter results to customers at trial plan = 0


WITH trial_plan AS


 (SELECT customer_id, start_date AS trial_date
 FROM foodie_fi.subscriptions

 WHERE plan_id = 0),

-- Filter results to customers at pro annual plan = 3

annual_plan AS

 (SELECT customer_id, start_date AS annual_date

FROM foodie_fi.subscriptions

 WHERE plan_id = 3)

SELECT ROUND(AVG(annual_date - trial_date),0) AS
avg_days_to_upgrade

FROM trial_plan tp

JOIN annual_plan ap

 ON tp.customer_id = ap.customer_id;
```

15. There is a new innovation in the financial industry called Neo-Banks: new aged digital only banks without physical branches.

Danny thought that there should be some sort of intersection between these new age banks, cryptocurrency and the data world…so he decided to launch a new initiative - Data Bank!

Data Bank runs just like any other digital bank - but it isn't only for banking activities, they also have the world's most secure distributed data storage platform!

Customers are allocated cloud data storage limits which are directly linked to how much money they have in their accounts. There are a few interesting caveats that go with this business model, and this is where the Data Bank team needs your help!

The management team at Data Bank want to increase their total customer base - but also need some help tracking just how much data storage their customers will need.

This case study is all about calculating metrics, growth and helping the business analyse their data in a smart way to better forecast and plan for their future developments!

Table 1: Regions

Just like popular cryptocurrency platforms - Data Bank is also run off a network of nodes where both money and data is stored across the globe. In a traditional banking sense - you can think of these nodes as bank branches or stores that exist around the world.

This `regions` table contains the `region_id` and their respective `region_name` values

region_id	region_name
1	Africa
2	America

3	Asia
4	Europe
5	Oceania

Table 2: Customer Nodes

Customers are randomly distributed across the nodes according to their region - this also specifies exactly which node contains both their cash and data.

This random distribution changes frequently to reduce the risk of hackers getting into Data Bank's system and stealing customer's money and data!

Below is a sample of the top 10 rows of the `data_bank.customer_nodes`

customer_id	**region_id**	**node_id**	**start_date**	**end_date**
1	3	4	2020-01-02	2020-01-03
2	3	5	2020-01-03	2020-01-17
3	5	4	2020-01-27	2020-02-18
4	5	4	2020-01-07	2020-01-19
5	3	3	2020-01-15	2020-01-23
6	1	1	2020-01-11	2020-02-06
7	2	5	2020-01-20	2020-02-04
8	1	2	2020-01-15	2020-01-28
9	4	5	2020-01-21	2020-01-25
10	3	4	2020-01-13	2020-01-14

Table 3: Customer Transactions

This table stores all customer deposits, withdrawals and purchases made using their Data Bank debit card.

customer_id	txn_date	txn_type	txn_amount
429	2020-01-21	deposit	82
155	2020-01-10	deposit	712
398	2020-01-01	deposit	196
255	2020-01-14	deposit	563
185	2020-01-29	deposit	626
309	2020-01-13	deposit	995
312	2020-01-20	deposit	485
376	2020-01-03	deposit	706
188	2020-01-13	deposit	601
138	2020-01-11	deposit	520

Customer Nodes Exploration

1. How many unique nodes are there on the Data Bank system?

```
SELECT count(DISTINCT node_id) AS unique_nodes

FROM customer_nodes;
```

2. What is the number of nodes per region?

```
SELECT region_id,
       region_name,
       count(node_id) AS node_count
FROM customer_nodes
INNER JOIN regions USING(region_id)
GROUP BY region_id;
```

3. How many customers are allocated to each region?

```
SELECT region_id,
       region_name,
       count(DISTINCT customer_id) AS customer_count
FROM customer_nodes
INNER JOIN regions USING(region_id)
GROUP BY region_id;
```

4. How many days on average are customers reallocated to a different node?

```
SELECT round(avg(datediff(end_date, start_date)), 2) AS avg_days
FROM customer_nodes
```

```
WHERE end_date!='9999-12-31';
```

5. What is the median, 80th and 95th percentile for this same reallocation days metric for each region?

```
WITH reallocation_days_cte AS

  (SELECT *,

                    (datediff(end_date,  start_date))  AS
reallocation_days

   FROM customer_nodes

   INNER JOIN regions USING (region_id)

   WHERE end_date!='9999-12-31'),

     percentile_cte AS

  (SELECT *,

          percent_rank() over(PARTITION BY region_id

                                                ORDER  BY
reallocation_days)*100 AS p

   FROM reallocation_days_cte)

SELECT region_id,

       region_name,
       reallocation_days
FROM percentile_cte
WHERE p >95
GROUP BY region_id;
```

Customer Transactions

1. What is the unique count and total amount for each transaction type?

```
SELECT txn_type,
       count(*) AS unique_count,
       sum(txn_amount) AS total_amont
FROM customer_transactions
GROUP BY txn_type;
```

2. What is the average total historical deposit counts and amounts for all customers?

```
SELECT round(count(customer_id)/
               (SELECT count(DISTINCT customer_id)
                          FROM customer_transactions)) AS average_deposit_count,
              concat('$', round(avg(txn_amount), 2)) AS
average_deposit_amount

FROM customer_transactions

WHERE txn_type = "deposit";
```

3. For each month - how many Data Bank customers make more than 1 deposit and either 1 purchase or 1 withdrawal in a single month?

```
WITH transaction_count_per_month_cte AS

  (SELECT customer_id,

          month(txn_date) AS txn_month,

                   SUM(IF(txn_type="deposit", 1, 0)) AS
deposit_count,

                 SUM(IF(txn_type="withdrawal", 1, 0)) AS
withdrawal_count,

                  SUM(IF(txn_type="purchase", 1, 0)) AS
purchase_count

   FROM customer_transactions

   GROUP BY customer_id,

            month(txn_date))

SELECT txn_month,
       count(DISTINCT customer_id) as customer_count
FROM transaction_count_per_month_cte
WHERE deposit_count>1
  AND (purchase_count = 1
       OR withdrawal_count = 1)
GROUP BY txn_month;
```

4. What is the closing balance for each customer at the end of the month?

```
WITH txn_monthly_balance_cte AS
  (SELECT customer_id,
          txn_amount,
          month(txn_date) AS txn_month,
          SUM(CASE
                  WHEN txn_type="deposit" THEN txn_amount
                  ELSE -txn_amount
              END) AS net_transaction_amt
   FROM customer_transactions
   GROUP BY customer_id,
            month(txn_date)

   ORDER BY customer_id)

SELECT customer_id,

       txn_month,

       net_transaction_amt,

              sum(net_transaction_amt)  over(PARTITION  BY
customer_id

                                        ORDER BY txn_month
ROWS  BETWEEN  UNBOUNDED  preceding  AND  CURRENT  ROW)  AS
closing_balance

FROM txn_monthly_balance_cte;
```

5. What is the percentage of customers who increase their closing balance by more than 5%?

```
WITH txn_monthly_balance_cte AS

  (SELECT customer_id,

          txn_amount,

          month(txn_date) AS txn_month,

          SUM(CASE

                  WHEN txn_type="deposit" THEN txn_amount
                  ELSE -txn_amount
              END) AS net_transaction_amt
   FROM customer_transactions
   GROUP BY customer_id,
            month(txn_date)
   ORDER BY customer_id)
SELECT customer_id,
       txn_month,
       net_transaction_amt,
              sum(net_transaction_amt)  over(PARTITION  BY
customer_id
ORDER  BY  txn_month  ROWS  BETWEEN  UNBOUNDED  preceding  AND
CURRENT ROW) AS closing_balance
FROM txn_monthly_balance_cte;
```