SQL Problems on HackerRank: Explanation and Solution (V1)

Delroy Low

Remark

This article aims to provide guidance on solving the SQL problems on HackerRank. This is not a new idea as such cheatsheet has been made available by Rahul Pathak, Akshay Daga, Thomas George, among others. However, this article differs in that it explains clearly the strategy to solve each SQL problem using MySQL queries. In addition, the skills or knowledge required in the problem solving process will also be outlined. On top of that, this article provides the MySQL queries to recreate the sample data set for readers to attempt the problem on their local device and for other SQL practices in the future. This article is merely for educational purpose and it should be served as a guide on solving these problems.

Contents

1 Problems on SELECT Clause

1.1 Basic Select: Select All

<u>*Problem*</u>: Query all columns (attributes) for every row in the `city` table.

We will have to return all columns from the table without any constraint. This problem can be easily solved by returning all entries in the original table using `SELECT * FROM table_name`.

```sql
SELECT * FROM city;

```

Listing 1: Select All

1.2 Basic Select: Select by ID

<u>*Problem*</u>: Query all columns for a city in `city` with the `id = 1661`.

We are required to return all columns in the table with a constraint that the associated *ID* of the entry is equivalent to *1661*. To this end, we first create a sub-table with entries that satisfy the condition, `ID = 1661` using a `WHERE` clause and return all columns from the resulting sub-table.

```sql
SELECT * FROM city
  WHERE id = 1661;

```

Listing 2: Select by ID

1.3 Basic Select: Japanese Cities' Attributes

<u>*Problem*</u>: Query all attributes of every Japanese city in the `city` table. The `countrycode` for Japan is `JPN`.

We have to return all columns for only Japanese cities. This can be achieved by returning all attributes in the sub-table created using the constraint, `COUNTRYCODE = 'JPN'` in `WHERE` clause.

```sql
SELECT * FROM city
  WHERE countrycode = 'JPN';

```

Listing 3: Japanese Cities' Attributes

1.4 Basic Select: Japanese Cities' Names

<u>*Problem*</u>: Query the names of all the Japanese cities in the `city` table. The `countrycode` for Japan is `JPN`.

Similar to *Problem 1.3*, we are only interested with Japanese cities. However, the object to be returned is now restricted to only the names of these cities. Thus, we create a sub-table consists only of entries with `countrycode = 'JPN'` using a `WHERE` clause. In addition, we use `SELECT column_name FROM table_name` instead of `SELECT * FROM table_name` to return the column of interest.

```sql

SELECT name FROM city
  WHERE countrycode = 'JPN';

```

Listing 4: Japanese Cities' Names

1.5 Basic Select: Revising the Select Query I

<u>*Problem*</u>: Query all columns for all American cities in the `city` table with `population` larger than 100000. The `countrycode` for America is `USA`.

There is no restriction on the objects to be returned and this implies that a `SELECT * ...` statement is appropriate. However, we are only interested with entries related to American cities and the population of the city should exceed *100,000*. It can be solved by creating a sub-table that satisfies `countrycode = 'USA'` and `population > 100000`. The logical operator `AND` is required to join the two constraints. All attributes in the resulting sub-table are returned.

```sql
SELECT * FROM city
  WHERE countrycode = 'USA' AND population  > 100000;

```

Listing 5: Revising the Select Query I

1.6 Basic Select: Revising the Select Query II

<u>*Problem*</u>: Query the `name` field for all American cities in the `city` table with populations larger than 120000. The `countrycode` for America is `USA`.

This problem requires us to select the `name` of all American cities from the table with the condition that the `population` of these cities exceeds *120,000*. Similar to *Problem 1.5*, we first create a sub-table that contains only entries that satisfy these constraints and return all entries in the `name` column from the resulting table.

```
SELECT name FROM city
  WHERE countrycode = 'USA' AND population > 120000;

```

Listing 6: Revising the Select Query II

1.7 Basic Select: Weather Observation Station 1

Problem: Query a list of `city` and `state` from the `station` table.

We are required to return all entries in `city` and `state` columns from the table without any constraint. This problem can be solved by `SELECT col_1, col_2 FROM table_name` to return multiple columns of interest from a table (or sub-table).

```
SELECT city, state FROM station;

```

Listing 7: Weather Observation Station 1

1.8 Basic Select: Weather Observation Station 3

Problem: Query a list of `city` names from `station` for cities that have an even ID number. Print the results in any order, but exclude duplicates from the answer.

The object that we want to return is `city`; however, there are duplicate entries in the column. Thus, the object to be returned from the table should be `DISTINCT(city)` to ensure the uniqueness of each entry in the column. In addition, we are only interested with the cities' names with even *ID* number. To this end, we can create a sub-table with only even *id* number using `MOD(id,2) = 0` in a `WHERE` clause and return the distinctive cities' names from the resulting table. Note that, $a \in \mathscr{Z}$ is even if `MOD(a,2) = 0` and is odd if `MOD(a,2) = 1`.

```
SELECT DISTINCT(city) FROM station
  WHERE MOD(id,2) = 0;

```

Listing 8: Weather Observation Station 3

1.9 Basic Select: Weather Observation Station 4

Problem: Find the difference between the total number of `city` entries in the table and the number of distinct `city` entries in the table.

We are required to find the difference between the total number of city entries and the unique city entries in the unconstrained table. Thus, the aggregate function `COUNT(arg1)` will be appropriate. Specifically, `COUNT(col1)` returns the number of rows (or entries) in variable, `col1`. To this end, we apply the aggregate function on column `city` and `DISTINCT` and their difference will be the object of interest.

```
SELECT COUNT(city) - COUNT(DISTINCT(city)) FROM station;

```

Listing 9: Weather Observation Station 4

1.10 Basic Select: Weather Observation Station 5

Problem: Query the two cities in `station` with the shortest and longest `city` names, as well as their respective lengths (i.e.: number of characters in the name). If there is more than one smallest or largest city, choose the one that comes first when ordered alphabetically.

This problem requires us to return `city` names, and their associated character length on a few conditions. Specifically, we are only interested in the city that has either the longest or the shortest name. In addition, if there's more than one city that satisfies these conditions, we return the one that comes first when ordered alphabetically. For instance, suppose `AB`, `BC`, `BCD`, `ABCD` and `BCDE` are the four entries. The expected output would be `AB 2, ABCD 4`. This is because `BC` and `BCDE` are ranked second in the ordering required.

A comparatively easier approach would be to write two queries that return the shortest and longest city names respectively. For both tasks, we employ the string function, `CHAR_LENGTH(string)` to compute the character length of the city's name. In addition, we use `ORDER BY(arg1, arg2)` to rearrange the order of the outputs. Specifically, `ORDER BY` clause first arranges the entries by `arg1` and secondary sort the resulting entries with `arg2`. The first entry in the resulting table is returned using `LIMIT 1`.

```
-- Query to obtain the shortest city's name --

SELECT city, CHAR_LENGTH(city) FROM station

```

```sql
    -- Order the table by character_length ascendingly to identify the shortest length;
    -- then, the table is reorder by city's name for entries with the same character
    -- length

    ORDER BY CHAR_LENGTH(city) ASC, city ASC

    -- This way, the first entry in the resulting table will be the entry with shortest
    -- character length and the associated city's name comes first when ordered
    -- alphabetically.

    LIMIT 1;

  --Query to obtain the longest city's name --

  SELECT CITY, CHAR_LENGTH(city) FROM STATION
    ORDER BY CHAR_LENGTH(city) DESC, CITY ASC
    LIMIT 1;

```

Listing 10: Weather Observation Station 5

Alternatively, one can also attempt to use sub-queries or window functions such as `ROW_NUMBER()` to solve this problem. The queries are outlined as follows. However, these methods are relatively complex and are not necessary for this problem.

```sql
  -- Query to obtain the shortest city's name --

  SELECT city, CHAR_LENGTH(city) FROM station
    WHERE CHAR_LENGTH(city) = (SELECT MIN(CHAR_LENGTH(city)) FROM station)
    ORDER BY city ASC
    LIMIT 1;

  --Query to obtain the longest city's name --

  SELECT city, CHAR_LENGTH(city) FROM station
    WHERE CHAR_LENGTH(city) = (SELECT MAX(CHAR_LENGTH(city)) FROM station)
    ORDER BY city ASC
    LIMIT 1;

```

```sql
  -- Single Query with Window Function --

  SELECT city, CHAR_LENGTH(city) FROM
      (SELECT city, CHAR_LENGTH(city),
      ROW_NUMBER() OVER(PARTITION BY CHAR_LENGTH(city) ORDER BY city) rn,
      MAX(CHAR_LENGTH(city)) OVER() max,
      MIN(CHAR_LENGTH(city)) OVER() min FROM station) sq1
    WHERE
    CHAR_LENGTH(city) = min AND rn = 1 OR
    CHAR_LENGTH(city) = max AND rn = 1;
```

1.11 Basic Select: Weather Observation Station 6

Problem: Query the list of `city` names starting with vowels (i.e., a, e, i, o, or u) from `station`. Your result cannot contain duplicates.

We use `DISTINCT(city)` to ensure that the distinctiveness in the entries returned from the table. However, we are only interested with entries that begin with vowels. To this end, we extract the first letter from the string variable using `LEFT(city, 1)`. Specifically, `LEFT(string_variable,n)` returns the first *n* letters in `string_variable`. In addition, we verify if the extracted letter from each row is a member of the set consists of vowels using `IN (’A’,’E’,’I’,’O’,’U’)`. This query is case-sensitive and the result will be incorrect if we were to use lowercase vowels. Alternatively, one can also attempt the question using a `LIKE` clause with `Wildcard` characters or to use `SUBSTRING` function in place of `LEFT` to extract the letters from the string.

```
    SELECT city FROM station
      WHERE LEFT(city,1) IN ('A','E','I','O','U');

```

Listing 11: Weather Observation Station 6

1.12 Basic Select: Weather Observation Station 7

Problem: Query the list of `city` names ending with vowels (a, e, i, o, u) from `station`. Your result cannot contain duplicates.

This problem setup is similar to *Problem 1.11*; however, we are interested with cities’ names that end with a vowel instead. To this end, we employ `RIGHT(city,1)` to extract the last letter in the string variable in which `RIGHT(string_variable,n)` does the same as `LEFT(···)` starting from the right. On top of that, `DISTINCT(city)` is required to ensure that there will be no duplicate entries in the column returned from the filtered table. Similarly, the query as follows is case-sensitive.

```
    SELECT DISTINCT(city) FROM station
      WHERE RIGHT(city,1) IN ('a', 'e', 'i', 'o', 'u');

```

Listing 12: Weather Observation Station 7

1.13 Basic Select: Weather Observation Station 8

Problem: Query the list of CITY`city` names from `station` which have vowels (i.e., a, e, i, o, and u) as both their first and last characters. Your result cannot contain duplicates.

We are required to return a list of cities' names from the table on the condition that the name begins and ends with vowels. This problem can be solved by combining the conditional statements in *Problem 1.11* and *Problem 1.12* using logical operator, `AND`. Alternatively, we employ `LOWER(string_variable)` to convert the letters extracted by `RIGHT(string_variable, 1)` to lowercase and use the same conditional statement in the `IN` clauses.

```
    SELECT city FROM station
      WHERE LOWER(LEFT(city,1)) IN ('a','e','i','o','u')
      AND RIGHT(city,1) IN ('a','e','i','o','u');

```

Listing 13: Weather Observation Station 8

1.14 Basic Select: Weather Observation Station 9

Problem: Query the list of `city` names from `station` that do not start with vowels. Your result cannot contain duplicates.

Unlike *Problem 1.11*, we are interested in returning the list of distinctive cities' names that do not start with a vowel. We can negate the conditional statement using `NOT IN` or logical operator, `!` to solve this problem. Specifically, a `WHERE` clause with logical operator, `!(Conditional Statement)` returns a sub-table contains of entries in which the conditional statement is not satisfied. In addition, `DISTINCT(city)` is employed to ensure that the entries returned are distinctive.

```
    SELECT DISTINCT(city) FROM station
      WHERE !(LEFT(city,1) IN ('A','E','I','O','U'));

```

Listing 14: Weather Observation Station 9

1.15 Basic Select: Weather Observation Station 10

Problem: Query the list of `city` names from `station` that do not end with vowels. Your result cannot contain duplicates.

In this problem, we are interested with the cities' names that does not end with vowels. Thus, it can be solved following the same strategy as in *Problem 1.14*. However, we will use `NOT IN` clause to negate the conditional statement instead.

```sql
SELECT DISTINCT(city) FROM station
  WHERE RIGHT(city,1) NOT IN ('a','e','i','o','u');

```

Listing 15: Weather Observation Station 10

In addition, *problems Weather Observation Station 11* and *12* can be solved by combining the conditional statement in *Problem 1.14* and this problem with logical operators, `OR` and `AND` respectively. These questions will not be discussed in this article.

1.16 Basic Select: Higher Than 75 Marks

Problem: Query the `name` of any student in `students` who scored higher than 75 Marks. Order your output by the last three characters of each name. If two or more students both have names ending in the same last three characters (i.e.: Bobby, Robby, etc.), secondary sort them by ascending `id`.

Sample Input;

```sql
CREATE TABLE IF NOT EXISTS htm_students(
  id INT AUTO_INCREMENT,
  name VARCHAR(100),
  marks INT,
  CONSTRAINT prim_key PRIMARY KEY(id),
  CONSTRAINT score_con CHECK(marks >= 0 AND marks <=100)
);

INSERT INTO htm_students(name, marks)
  VALUES
    ('Ashley', 81), ('Samantha', 75),
    ('Julia', 76), ('Belvet', 84);

--  Expected Output: Ashley, Julia, Belvet --

```

The object of interest in this question is `name`. However, we want to create a sub-table with entries that satisfy `marks > 75` using a `WHERE` clause. In addition, the returned entries have to be ordered using `ORDER BY(RIGHT(name,3), id)` to first sort the entries with `RIGHT(name,3)` and secondary sort them with `id` for entries that have same last three characters in their names.

```sql
SELECT name FROM htm_students
  WHERE marks > 75
  ORDER BY RIGHT(name,3) ASC, id ASC;
```

Listing 16: Higher Than 75 Marks

1.17 Basic Select: Employee Names

Problem: Write a query that prints a list of employee names (i.e.: the `name` attribute) from the `employee` table in alphabetical order.

Sample Input:

```sql
CREATE TABLE IF NOT EXISTS en_employee(
  employee_id INT,
  name VARCHAR(100),
  months INT,
  salary INT,
  CONSTRAINT prim_key PRIMARY KEY(employee_id),
  CONSTRAINT non_neg_check1 CHECK(months >= 0),
  CONSTRAINT non_neg_check2 CHECK(salary >= 0)
);

INSERT INTO en_employee(employee_id, name, months, salary)
  VALUES
    (12228, 'Rose', 15, 1968), (33645, 'Angela', 1, 3443),
    (45692, 'Frank', 17, 1608), (56118, 'Patrick', 7, 1345),
    (59725, 'Lisa', 11, 2330), (74197, 'Kimberly', 16, 4372),
    (78454, 'Bonnie', 8, 1771), (83565, 'Michael', 6, 2017),
    (98607, 'Todd', 5, 3396), (99989, 'Joe', 9, 3573);

Expected Output:  Angela, Bonnie, Frank, Joe, Kimberly, Lisa, Michael, Patrick, Rose,
                  Todd
```

This is an unconstrained problem with the object of interest to be the `name` column. However, it is required that the output to be ordered by `name` alphabetically. Thus, an `ORDER BY` clause is required.

```sql
SELECT name FROM en_employee
  ORDER BY name ASC;
```

Listing 17: Employee Names

1.18 Basic Select: Employee Salaries

Problem: Write a query that prints a list of employee names (i.e.: the `name` attribute) for employees in `employee` having a salary greater than $2,000 per month who have been employees for less than 10 months. Sort your result by ascending `employee_id`.

This problem shares the similar sample input as *Problem 1.17*. Similarly, `name` is the column of interest and it is required that we sort the output by `employee_id` in an ascending order. However, the employees' names to be returned must have the associated salary to be greater than *$2,000* for this problem. In addition, they have to be working with the company for less than 10 months. Thus, we create a sub-table with entries that satisfy `salary > 2000 AND months < 10` using a `WHERE` clause and return the `name` column from the resulting table.

```sql
SELECT name FROM en_employee
  WHERE salary > 2000 AND months < 10
  ORDER BY name ASC;

```

Listing 18: Employee Salaries

1.19 Advanced Select: Type of Triangle

Problem: Write a query identifying the type of each record in the `triangles` table using its three side lengths. Output one of the following statements for each record in the table:

1. Equilateral: It's a triangle with 3 sides of equal length.
2. Isosceles: It's a triangle with 2 sides of equal length.
3. Scalene: It's a triangle with 3 sides of differing lengths.
4. Not A Triangle: The given values of A, B, and C don't form a triangle.

Sample Input:

```sql
CREATE TABLE IF NOT EXISTS tot_triangles(
  A INT,
  B INT,
  C INT
);

INSERT INTO tot_triangles(A,B,C)
  VALUES
    (20,20,23), (20,20,20),
    (20,21,22), (13,14,30);
```

```

Expected Output: Isosceles, Equilateral, Scalene, Not A Triangle

```

Where a configuration of (a,b,c) cannot form a triangle if $a+b \leq c$, $a+c \leq b$ or $b+c \leq a$. In this problem, we have to write a conditional statement to return different strings based on the configuration of (a,b,c) respectively. A `CASE` clause is thus required.

```
SELECT
    CASE
        WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'
        WHEN A = B AND A = C  THEN 'Equilateral'
        WHEN !(A = B) AND !(A = C) THEN 'Scalene'
        ELSE 'Isosceles'
    END
FROM tot_triangles;

```

Listing 19: Type of Triangle

The order in which the conditional statements are arranged matters. Consider the two queries as follows. Suppose that we have a configuration, `(20, 20, 20)` that indicates an `Equilateral` triangle. However, *Query 1* will identify such configuration as a `Isosceles` triangle due to the arrangement of the conditional statement. On the other hand, *Query 2* identifies a configuration, (20,20,40) as `Isosceles` when these parameters cannot form a triangle. Thus, one has to be cautious about the order of the conditional statements when writing a `CASE` clause.

```
-- Query 1 --

SELECT
    CASE
        WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'
        WHEN A = B OR A = C OR B = C THEN 'Isosceles'
        WHEN A = B AND A = C  THEN 'Equilateral'
        ELSE 'Scalene'
    END
FROM tot_triangles;

-- Query 2 --

  SELECT
      CASE
          WHEN A = B OR A = C OR B = C THEN 'Isosceles'
          WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'
          WHEN A = B AND A = C  THEN 'Equilateral'
          ELSE 'Scalene'
      END
```

```
        FROM tot_triangles;

```

1.20 Advanced Select: Binary Tree Nodes

Problem: Write a query to find the node type of Binary Tree ordered by the value of the node. Output one of the following for each node:

1. Root: If the node is a root node.
2. Leaf: If the node is a leaf node.
3. Inner: If the node neither a root nor leaf node.

Sample Input:

```
CREATE TABLE IF NOT EXISTS btn_bst(
  N INT,
  P INT -- Parent Node of N
);

INSERT INTO btn_bst(N, P)
  VALUES
    (1,2), (3,2), (6,8), (9,8),
    (2,5), (8,5), (5,NULL);

Expected Output: 1 Leaf, 2 Inner, 3 Leaf, 5 Root, 6 Leaf, 8 Inner, 9 Leaf

```

A node is a root (*or leaf*) when it has no parent (*or it is not a parent of any leaf*). Thus, $n \in N$ is a parent if the associated entry in `P` is `NULL`; while, $n \in N$ is a leaf node if $n \notin P$. In addition, a node is an inner node if it is a member of both sets, N and P. Henceforth, we can write a `CASE` clause based on these criteria to identify the node type. However, we need a sub-query when writing the conditional statement for the identification of a leaf node since we have to verify if $n \notin P$ and the sub-query is intended to return the vector of entries in `P`. The `CASE` clause is as follows.

```
CASE
    WHEN P IS NULL THEN 'Parent'
    WHEN N NOT IN (
                SELECT DISTINCT(P) FROM btn_bst
                WHERE P IS NOT NULL
              )  -- The sub-query
              THEN 'Leaf'
    ELSE 'Inner'
END

```

In addition, we use `CONCAT(N, ' ', ···)` function with a `CASE` clause in place of $\cdots$ to return the output in the format required. Specifically, `CONCAT(arg1, arg2)` concatenates the two strings. For each (n_i, p_i), the augmented function validates the conditional statements in `CASE` clause and concatenates the resulting string with n_i. For instance, if p_i is `NULL`, the `CASE` clause returns `'Parent'` and this string is concatenated with n_i to formulate the output as `n_i Parent`. The resulting entries are then ordered by `N`.

```
SELECT
  CONCAT(N, ' ',
    CASE
        WHEN P IS NULL THEN 'Root'
        WHEN !(
                N IN (SELECT DISTINCT(P) FROM btn_bst
                      WHERE P IS NOT NULL)
              ) THEN 'Leaf'
        ELSE 'Inner'
    END
    )
FROM btn_bst
ORDER BY N;

```

Listing 20: Binary Tree Nodes

1.21 Advanced Select: The Pads

Problem: Generate the following two result sets:

1. Query an alphabetically ordered list of all names in `occupations`, immediately followed by the first letter of each profession as a parenthetical (i.e.: enclosed in parentheses). For example: AnActorName(A), ADoctorName(D), AProfessorName(P), and ASingerName(S).

2. Query the number of ocurrences of each occupation in `occupations`. Sort the occurrences in ascending order, and output them in the following format:

   `There are a total of [occupation_count] [occupation]s.`

   where `[occupation_count]` is the number of occurrences of an occupation in `occupations` and `[occupation]` is the lowercase occupation name. If more than one occupation has the same `[occupation_count]`, they should be ordered alphabetically.

Sample Input:

```sql
CREATE TABLE IF NOT EXISTS tp_occupations(
  name VARCHAR(100),
  occupation VARCHAR(100)
);

INSERT INTO tp_occupations(name, occupation)
  VALUES
    ('Samantha', 'Doctor'), ('Julia', 'Actor'), ('Maria', 'Actor'),
    ('Meera', 'Singer'), ('Ashely', 'Professor'), ('Ketty', 'Professor'),
    ('Christine', 'Professor'), ('Jane', 'Actor'), ('Jenny', 'Doctor'),
    ('Priya', 'Singer');

Expected Output: Ashely(P), Christeen(P), Jane(A), Jenny(D), Julia(A), Ketty(P),
                 Maria(A), Meera(S), Priya(S), Samantha(D),
                 There are a total of 2 doctors., There are a total of 2 singers.,
                 There are a total of 3 actors., There are a total of 3 professors.

```

For result set 1, we use `CONCAT(name, '(', LEFT(occupation, 1) ,')'` to modify the entries in the output to meet the format required. Despite it is not mentioned explicitly that we have to order the output by `name` ascendingly; however, the sample result is indicative of this requirement. Thus, an `ORDER BY` clause is required.

On the other, we have to use aggregate function, `COUNT(name)` with a `GROUP BY occupation` clause to solve the second task. Specifically, the `GROUP BY` clause will first create sub-tables based on the distinctive entries in `occupation`. The object to be returned, `COUNT(name)` then provides us with a vector of number of entries (or rows) in each sub-table created using `GROUP BY` clause.

In the context of the sample input, `GROUP BY` clause creates sub-tables for each occupation. For instance, the sub-table for `occupation = 'Doctor'` consists of entries, `'Samantha', 'Jenny'` and this logic applies for the sub-tables of other occupations. Thus, when `COUNT(name)` is applied on these resulting tables, it returns the number of entries in each sub-table (e.g. 2 for `occupation = 'Doctor'`, 3 for `occupation = 'Actor'` and so on).

However, we need to use `CONCAT('There are a total of ', COUNT(name), ' ', LOWER (occupation), 's.')` to return the output in the required format. The inputs in the function are valid because `COUNT(name)` with `GROUP BY` clause produces a sub-table with columns `COUNT(name)` and `occupation`. Verify this by `SELECT COUNT(name), occupation ···` query. The outputs are then ordered by `COUNT(name)` and secondary sorted by `occupation`.

```sql
SELECT
  CONCAT(name, '(', LEFT(occupation,1), ')')
```

```sql
FROM tp_occupations
ORDER BY name;

SELECT
  CONCAT('There are a total of ', COUNT(name), ' ', LOWER(occupation), 's.')
FROM tp_occupations
GROUP BY occupation
ORDER BY COUNT(name) ASC, occupation ASC;

```

Listing 21: The Pads

1.22 Advanced Select: New Companies

<u>*Problem*</u>: Amber's conglomerate corporation just acquired some new companies. Each of the companies follows the hierarchy that *Founder* > *Lead_Manager* > *Senior_Manager* > *Manager* > *Employee*. Write a query to print the `company_code`, founder `name`, total number of lead managers, total number of senior managers, total number of managers, and total number of employees. Order your output by ascending `company_code`. In addition, the tables may contain duplicate records.

<u>*Sample Input*</u>:

```sql
CREATE TABLE IF NOT EXISTS nc_company(
  company_code VARCHAR(100),
  founder VARCHAR(100),
  CONSTRAINT prim_key1 PRIMARY KEY(company_code)
);

CREATE TABLE IF NOT EXISTS nc_lead_manager(
  lead_manager_code VARCHAR(100),
  company_code VARCHAR(100),
  CONSTRAINT prim_key2 PRIMARY KEY(lead_manager_code),
  CONSTRAINT f_key_2_1 FOREIGN KEY(company_code) REFERENCES nc_company(company_code)
);

CREATE TABLE IF NOT EXISTS nc_senior_manager(
  senior_manager_code VARCHAR(100),
  lead_manager_code VARCHAR(100),
  company_code VARCHAR(100),
  CONSTRAINT prim_key3 PRIMARY KEY(senior_manager_code),
  CONSTRAINT f_key_3_1 FOREIGN KEY(company_code) REFERENCES nc_company(company_code),
  CONSTRAINT f_key_3_2 FOREIGN KEY(lead_manager_code)
                  REFERENCES nc_lead_manager(lead_manager_code)
);

CREATE TABLE IF NOT EXISTS nc_manager(
  manager_code VARCHAR(100),
  senior_manager_code VARCHAR(100),
  lead_manager_code VARCHAR(100),
```

```
  company_code VARCHAR(100),
  CONSTRAINT prim_key4 PRIMARY KEY(manager_code),
  CONSTRAINT f_key_4_1 FOREIGN KEY(company_code) REFERENCES nc_company(company_code),
  CONSTRAINT f_key_4_2 FOREIGN KEY(lead_manager_code)
                  REFERENCES nc_lead_manager(lead_manager_code),
  CONSTRAINT f_key_4_3 FOREIGN KEY(senior_manager_code)
                  REFERENCES nc_senior_manager(senior_manager_code)
);

CREATE TABLE IF NOT EXISTS nc_employee(
  employee_code VARCHAR(100),
  manager_code VARCHAR(100),
  senior_manager_code VARCHAR(100),
  lead_manager_code VARCHAR(100),
  company_code VARCHAR(100),
  CONSTRAINT prim_key5 PRIMARY KEY(employee_code),
  CONSTRAINT f_key_5_1 FOREIGN KEY(company_code) REFERENCES nc_company(company_code),
  CONSTRAINT f_key_5_2 FOREIGN KEY(lead_manager_code)
                  REFERENCES nc_lead_manager(lead_manager_code),
  CONSTRAINT f_key_5_3 FOREIGN KEY(senior_manager_code)
                  REFERENCES nc_senior_manager(senior_manager_code),
  CONSTRAINT f_key_5_4 FOREIGN KEY(manager_code) REFERENCES nc_manager(manager_code)
);

INSERT INTO nc_company(company_code, founder)
  VALUES
    ('C1', 'Monika'), ('C2', 'Samantha');

INSERT INTO nc_lead_manager(lead_manager_code, company_code)
  VALUES
    ('LM1', 'C1'), ('LM2', 'C2');

INSERT INTO nc_senior_manager(senior_manager_code, lead_manager_code, company_code)
  VALUES
    ('SM1', 'LM1', 'C1'), ('SM2', 'LM1', 'C1'), ('SM3', 'LM2', 'C2');

INSERT INTO nc_manager(manager_code, senior_manager_code, lead_manager_code,
                      company_code)
  VALUES
    ('M1', 'SM1', 'LM1', 'C1'), ('M2', 'SM3', 'LM2', 'C2'),
    ('M3', 'SM3', 'LM2', 'C2');

INSERT INTO nc_employee(employee_code, manager_code, senior_manager_code,
                        lead_manager_code, company_code)
  VALUES
    ('E1', 'M1', 'SM1', 'LM1', 'C1'), ('E2', 'M1', 'SM1', 'LM1', 'C1'),
    ('E3', 'M2', 'SM3', 'LM2', 'C2'), ('E4', 'M3', 'SM3', 'LM2', 'C2');

Expected Output: C1 Monika 1 2 1 2, C2 Samantha 1 1 2 2
```

We use `INNER JOIN` clause for a cleaner query. However, it should be noted that one can also join two different tables using `SELECT col_name FROM tab_1, tab_2 WHERE condition_1` and the output is similar to that obtained using a `INNER JOIN` with `condition_1`.

For this problem, we need to create a new table that integrates the columns of all tables. Using the *sample input* as an example, we can first run the query as follows to compute the number of lead manager working in the company.

```
    SELECT c.company_code, c.founder, COUNT(DISTINCT(l.lead_manager_code))
      FROM nc_company AS c
        INNER JOIN nc_lead_manager AS l ON c.company_code = l.company_code
      GROUP BY c.company_code, c.founder;

    -- Output: C1 Monika 1, C2 Samantha 1

```

We treat the output as a new table and perform `INNER JOIN` clause with table `nc_senior_manager` on the condition that `c.company_code = senior_manager.company_code` and compute the numbers of distinctive senior managers working in each company using `COUNT(DISTINCT(col_name))` function. The query is as follows.

```
    SELECT c.company_code, c.founder, COUNT(DISTINCT(l.lead_manager_code))
      FROM nc_company AS c
        INNER JOIN nc_lead_manager AS l ON c.company_code = l.company_code
        INNER JOIN nc_senior_manager AS s ON c.company_code = s.company_code
      GROUP BY c.company_code, c.founder;

    -- Output: C1 Monika 1 2, C2 Samantha 1 1

```

Thus, we follow this step to join all the tables in the database together and compute the numbers of distinctive managers and employees in each company. The output is reordered using `ORDER BY c.company_code` to meet the required format.

```
  SELECT c.company_code, c.founder,
      COUNT(DISTINCT(l.lead_manager_code)), COUNT(DISTINCT(s.senior_manager_code)),
      COUNT(DISTINCT(m.manager_code)), COUNT(DISTINCT(e.employee_code))
    FROM nc_company AS c
        INNER JOIN nc_lead_manager AS l ON c.company_code = l.company_code
        INNER JOIN nc_senior_manager AS s ON c.company_code = s.company_code
        INNER JOIN nc_manager AS m on c.company_code = m.company_code
        INNER JOIN nc_employee AS e on c.company_code = e.company_code
    GROUP BY c.company_code, c.founder
    ORDER BY c.company_code;

```

Listing 22: New Companies

2 Questions on JOIN Clause

2.1 Basic Join: Population Census

Problem: Given the `city` and `country` tables, query the sum of the populations of all cities available in `city` table where the `country.continent` is 'Asia'. `city.countrycode` and `country.code` are matching key columns.

The column of interest is `name` in the `city` table; however, the constraint is imposed on variable, `continent` in the `country` table. Where `[table_name].[column_name]` is the syntax to indicate the location of a particular column when dealing with multiple tables. To this end, we can create a new table that augments the entries in the two tables using an `INNER JOIN` clause on the condition that `city.countrycode = country.code`. The augmented table will thus contain both `city.population` and `country.continent` columns. We can conveniently impose the conditional statement, `country.continent = 'Asia'` to filter the resulting table and employ function, `SUM(city.population)` to compute the aggregate of all entries in column, `city.population`.

```
SELECT SUM(city.population) FROM city
    INNER JOIN country
    ON city.countrycode = country.code
  WHERE country.continent = 'Asia';

```

Listing 23: Population Census

2.2 Basic Join: African Cities

Problem: Given the `city` and `country` tables, query the names of all cities where the `continent` is 'Africa'. `city.countrycode` and `country.code` are matching key columns.

Similarly, we are interested with the `name` column in `city` table. However, the original table does not contain a `continent` column for us to impose the constraint on the outputs. Thus, we create a new table that consists of all columns in both tables `city` and `country` using an `INNER JOIN` clause on the condition `city.countrycode = country.code`. Rows in the `city.name` column are returned if the constraint is satisfied.

```
SELECT city.name FROM city
    INNER JOIN country
    ON city.countrycode = country.code
  WHERE country.continent = 'Africa';

```

Listing 24: African Cities

2.3 Basic Join: Average Population of Each Continent

Problem: Given the `city` and `country` tables, query the names of all the continents (`country.continent`) and their respective average city populations (`city.population`) rounded down to the nearest integer. `city.countrycode` and `country.code` are matching key columns.

This problem can be solved using the same method as in *Problem 2.1* and *2.2*. However, the columns of interest are located in two different tables. Specifically, the list of entries on the names of all continents is available in table `country`; while the mean city population (of each continent) has to be computed from table `city`. To this end, we need to integrate the two tables. In addition, we use a `GROUP BY country.continent` clause on the augmented table for every distinctive continent in `country.continent` column and the function `AVG(city.population)` to compute the mean population size in each sub-table. The mean population size is then rounded down to the nearest integer using the function, `FLOOR(arg1)`.

```sql
SELECT country.continent, FLOOR(AVG(city.population)) FROM country
  INNER JOIN city ON country.code = city.countrycode
  GROUP BY country.continent;

```

Listing 25: Average Population of Each Continent

2.4 Basic Join: The Report

Problem: Given the two tables, `students` and `grades` with:

Table 1: Entries in grades

grade	min_mark	max_mark
1	0	9
2	10	19
3	20	29
4	30	39
5	40	49
6	50	59
7	60	69
8	70	79
9	80	89
10	90	100

Ketty gives Eve a task to generate a report containing three columns: `name`, `grade` and `marks`. Ketty doesn't want the names of those students who received a grade lower than 8. The report must be in descending order by `grade` – i.e. higher grades are entered first. If there is more than one student with the same grade (8-10) assigned to them, order those particular students by their `name` alphabetically. Finally, if the grade is lower than 8, use "NULL" as their name and list them by their grades in descending order. If there is more than one student with the same grade (1-7) assigned to them, order those particular students by their `marks` in ascending order.

Sample Input:

```sql
CREATE TABLE IF NOT EXISTS tr_students(
  id INT AUTO_INCREMENT,
  name VARCHAR(100),
  marks INT,
  CONSTRAINT prim_key PRIMARY KEY(id)
);

CREATE TABLE IF NOT EXISTS tr_grades(
  grade INT,
  min_mark INT,
  max_mark INT
);

INSERT INTO tr_students( name, marks)
  VALUES
    ('Julia', 88), ('Samantha', 68), ('Maria', 99),
    ('Scarlet', 78), ('Ashley', 63), ('Jane', 81);

INSERT INTO tr_grades(grade, min_mark, max_mark)
  VALUES
    (1, 0, 9), (2, 10, 19), (3, 20, 29), (4, 30, 39), (5, 40,49),
    (6, 50, 59), (7, 60, 69), (8, 70, 79), (9, 80, 89), (10, 90, 100);

Expected Output: Maria 10 99, Jane 9 81, Julia 9 88,
                    Scarlet 8 78, NULL 7 63, NULL 7 68
```

There are three columns interest in this problem, namely the `name` and `marks` columns from table `students` and the `grade` column from `grades` table. Thus, we will have to join the entries in both tables using a `INNER JOIN` clause on the condition that `students.marks >= grades.min_mark AND students.marks <= grades.max_mark`. Specifically, this function combines the row_s in `students` with row_g in `grades` if it is true that the associated marks in row_s lies within the range of the `min_mark` and `max_mark` in row_g.

However, we need to use a `CASE` clause to return `NULL` for entries with `grades.grade < 8`. In addition, we employ the `ORDER BY(grades.grade DESC, students.name, students.marks)` to sort the outputs to meet the format required. The following query uses table alias to make it more readable.

```sql
SELECT
    CASE
      WHEN g.grade < 8 THEN NULL
      ELSE s.name
    END, g.grade, s.marks FROM tr_students AS s -- Table Alias 's'
INNER JOIN tr_grades AS g -- Table Alias 'g'
    ON s.marks >= g.min_mark AND s.marks <= g.max_mark
ORDER BY g.grade DESC, s.name ASC, s.marks ASC;

```

Listing 26: The Report

2.5 Basic Join: Contest Leaderboard

Problem: The total score of a hacker is the sum of their maximum scores for all of the challenges. Write a query to print the `hacker_id`, `name`, and total score of the hackers ordered by the descending score. If more than one hacker achieved the same total score, then sort the result by ascending `hacker_id`. Exclude all hackers with a total score of 0 from your result.

Sample Input:

```sql
CREATE TABLE IF NOT EXISTS cl_hackers(
  hacker_id INT,
  name VARCHAR(100),
  CONSTRAINT prim_key PRIMARY KEY(hacker_id)
);

CREATE TABLE IF NOT EXISTS cl_submissions(
  submission_id INT,
  hacker_id INT,
  challenge_id INT,
  score INT,
  CONSTRAINT f_key FOREIGN KEY(hacker_id) REFERENCES cl_hackers(hacker_id)
);

INSERT INTO cl_hackers(hacker_id, name)
  VALUES
  (4071, 'Rose'), (4806, 'Angela'), (26071, 'Frank'), (49438, 'Patrick'),
  (74842, 'Lisa'), (80305, 'Kimberly'), (84072, 'Bonnie'), (87868, 'Michael'),
  (92118, 'Todd'), (95895, 'Joe');

INSERT INTO cl_submissions(submission_id, hacker_id, challenge_id, score)
  VALUES
  (67194, 74842, 63132, 76),(64479, 74842, 19797, 98), (40742, 26071, 49593, 20),
  (17513, 4806, 49593, 32), (69846, 80305, 19797, 19),(41002, 26071, 89343, 36),
  (52826, 49438, 49593, 9), (31093, 26071, 19797, 2), (81614, 84072, 49593, 100),
  (44829, 26071, 89343, 7), (75147, 80305, 49593, 48), (14115, 4806, 49593, 76),
  (6943, 4071, 19797, 95), (12855, 4806, 25917, 13), (73343, 80305, 49593, 42),
  (84264, 84072, 63132, 0), (9951, 4071, 49593, 43), (45104, 49438, 25917, 34),
  (53795, 74842, 19797, 5), (26363, 26071, 19797, 29), (10063, 4071, 49593, 96);

```

```
    Expected Output: 4071 Rose 191, 74842 Lisa 174, 84072 Bonnie 100, 4806 Angela 89,
                        26071 Frank 85, 80305 Kimberly 67, 49438 Patrick 43

```

We need to create a new table with columns, `hacker_id`, `name` and the total score of the hackers earned by participating in different competitions. To this end, we can first perform a `INNER JOIN` clause on the condition that `hacker.hacker_id = submissions.hacker_id`.

```
    SELECT h.hacker_id, h.name, s.challenge_id, s.score FROM cl_hackers AS h
        INNER JOIN cl_submissions AS s USING(hacker_id);

```

The output from the query is as follows.

Table 2: Contest Leaderboard I

`hacker_id`	`name`	`challenge_id`	`score`
4071	Rose	19797	95
4071	Rose	49593	43
4071	Rose	49593	96
4806	Angela	49593	32
4806	Angela	49593	76
4806	Angela	25917	13
⋮	⋮	⋮	⋮

Note that, a particular hacker may participate in the same challenge for more than one time. For instance, `Rose` participated in challenge `49593` twice. Thus, we use `GROUP BY hacker_id, name, challenge_id` and apply the function `MAX(score)` to seek for the maximum score for every participant in each challenge.

To this end, we have created a new table that consists of all information that are required for this problem. However, we need to use the function `SUM(MAX(score))` in order to return the total score earned by a particular hacker. Thus, a `GROUP BY hacker_id, name` clause is required. In addition, we impose the condition that `SUM(MAX(score)) > 0` using a `HAVING` clause to filter out entries with 0 total score.

```
    SELECT hacker_id, name, SUM(max_score_per_challenge) sum_score FROM
        (SELECT h.hacker_id, h.name, s.challenge_id, MAX(s.score) AS
  max_score_per_challenge
          FROM cl_hackers AS h
        INNER JOIN cl_submissions AS s ON h.hacker_id = s.hacker_id
```

```
      GROUP BY h.hacker_id, h.name, s.challenge_id) AS sq1
    GROUP BY hacker_id, name
    HAVING sum_score > 0
    ORDER BY sum_score DESC, hacker_id ASC;

```

Listing 27: Contest Leaderboard

2.6 Basic Join: Top Competitors

Problem: Write a query to print the respective `hacker_id` and `name` of hackers who achieved full scores for more than one challenge. Order your output in descending order by the total number of challenges in which the hacker earned a full score. If more than one hacker received full scores in same number of challenges, then sort them by ascending `hacker_id`.

Sample Input:

```
CREATE TABLE IF NOT EXISTS tc_hackers(
  hacker_id INT,
  name VARCHAR(100),
  CONSTRAINT prim_key PRIMARY KEY(hacker_id)
);

CREATE TABLE IF NOT EXISTS tc_challenges(
  challenge_id INT,
  hacker_id INT, -- This indicates the creator of the challenge
  difficulty_level INT, -- This indicates the difficulty level of the challenge
  CONSTRAINT prim_key PRIMARY KEY challenge_id
);

CREATE TABLE IF NOT EXISTS tc_difficulty(
  difficulty_level INT,
  score INT -- This indicates the associated full score for each challenge
);

CREATE TABLE IF NOT EXISTS tc_submissions(
  submission_id INT,
  hacker_id INT,
  challenge_id INT,
  score INT -- This column indicates the actual score of participants
);

INSERT INTO tc_hackers(hacker_id, name)
  VALUES
    (5580, 'Rose'), (8439, 'Angela'), (27205, 'Frank'), (52243, 'Patrick'),
    (52348, 'Lisa'), (57645, 'Kimberly'), (77726, 'Bonnie'), (83082, 'Michael'),
    (86870, 'Todd'), (90411, 'Joe');

INSERT INTO tc_challenges(challenge_id, hacker_id, difficulty_level)
  VALUES
  (4810,77726,4), (21089,27205,1), (36566,5580,7),
```

Table 3: Top Competitors I

hacker_id	challenge_id	var_actual	var_level
77726	77726	30	7
77726	77726	10	1
52243	36566	77	7
⋮	⋮	⋮	⋮

```
  (66730,52243,6), (71055,52243,2);

INSERT INTO tc_difficulty(difficulty_level, score)
  VALUES
  (1,20), (2,30), (3,40), (4,60), (5,80), (6,100), (7,120);

INSERT INTO tc_submissions(submission_id, hacker_id, challenge_id, score)
  VALUES
   (68628,77726,36566,30), (65300,77726,21089,10), (40326,52243,36566,77),
   (8941,27205,4810,4), (83554,77726,66730,30), (43353,52243,66730,0),
   (55385,52348,71055,20), (39784,27205,71055,23), (94613,86870,71055,30),
   (45788,52348,36566,0), (93058,86870,36566,30), (7344,8439,66730,92),
   (2721,8439,4810,36), (523,5580,71055,4), (49105,52348,66730,0),
   (55877,57645,66730,80), (38355,27205,66730,35), (3924,8439,36566,80),
   (97397,90411,66730,100), (84162,83082,4810,40), (97431,90411,71055,30);

Expected Output: 90411 Joe
```

We have to create a new table that consists of the following fields, `h.hacker_id`, `h.name` and total number of times that each hacker earned a full score in the challenge that they participated. To this end, consider the query as follows.

```
SELECT s.hacker_id, s.challenge_id, s.score AS var_actual,
          c.difficulty_level AS var_level
FROM tc_submissions AS s
    INNER JOIN tc_challenges AS c USING(challenge_id);
```

The output contains the columns as in *Table 3*. Where `var_actual` consists of entries on the actual score that each hacker earned in the competition; while, `var_level` indicates the difficulty level of the challenge. However, we need an extra column on the full score for each difficulty level. This column is available in the `difficulty_level` table. To this end, we perform another `INNER JOIN` operation to integrate the resulting sub-table with `score` column in table `difficulty_level AS d` as follows.

In addition, we restrict the output by the condition that `var_actual = d.score` as we are only interested with the number of times that hackers earned a full score. However, we will require `GROUP BY s.hacker_id` for us to apply the aggregate function, `COUNT(*)` to return the number of times that each hacker earned a full score. The query is as follows.

```
    SELECT s.hacker_id, COUNT(*)
        FROM tc_submissions AS s
        INNER JOIN tc_challenges AS c USING(challenge_id)
        INNER JOIN tc_difficulty_level AS d ON d.difficulty_level = c.difficulty level
      WHERE d.score = s.score
      GROUP BY s.hacker_id;

```

The output from the query would be `86870 1, 90411 2`. This implies that only two hackers earned full score in the competition that they participated. Specifically, hacker `86870` earned a full score for a single time; while, hacker `90411` earned full scores for two of the challenges that he participated. To this end, we implement the `HAVING` clause with the condition that `COUNT(*) > 1` to drop rows associated with hackers who have only earned full score once. In addition, the resulting table is joined with `hackers` since we are required to return the `name` of the hacker.

```
    SELECT h.hacker_id, h.name FROM tc_submissions AS s
        INNER JOIN tc_challenges AS c ON s.challenge_id = c.challenge_id
        INNER JOIN tc_difficulty AS d ON c.difficulty_level = d.difficulty_level
        INNER JOIN tc_hackers AS h ON s.hacker_id = h.hacker_id
      WHERE d.score = s.score
      GROUP BY h.hacker_id, h.name
      HAVING COUNT(*) > 1
      ORDER BY COUNT(*) DESC, h.hacker_id ASC;

```

Listing 28: Top Competitors

2.7 Basic Join: Challenges

Problem: Write a query to print the `hacker_id`, `name`, and the total number of challenges created by each student. Sort your results by the total number of challenges in descending order. If more than one student created the same number of challenges, then sort the result by `hacker_id`. If more than one student created the same number of challenges and the count is less than the maximum number of challenges created, then exclude those students from the result.

```
    CREATE TABLE IF NOT EXISTS c_hackers(
      hacker_id INT,
      name VARCHAR(100),
```

```sql
  CONSTRAINT prim_key PRIMARY KEY(hacker_id)
);

CREATE TABLE IF NOT EXISTS c_challenges(
  challenge_id INT,
  hacker_id INT,
  CONSTRAINT prim_key PRIMARY KEY(challenge_id),
  CONSTRAINT f_key1 FOREIGN KEY(hacker_id) REFERENCES c_hackers(hacker_id)
);

INSERT INTO c_hackers(hacker_id, name)
  VALUES
  (12299, 'Rose'), (34856, 'Angela'), (79345, 'Frank'),
  (80491, 'Patrick'), (81041, 'Lisa'), (12234, 'Andrew');

INSERT INTO c_challenges(challenge_id, hacker_id)
  VALUES
  (63963, 81041), (63117, 79345), (28225, 34856), (21989, 12299),
  (4653, 12299), (70070, 79345), (36905, 34856), (61136, 80491),
  (17234, 12299), (80308, 79345), (40510, 34856), (79820, 80491),
  (22720, 12299), (21394, 12299), (36261, 34856), (15334, 12299),
  (71435, 79345), (23157, 34856), (54102, 34856), (69065, 80491),
  (22245, 12234), (33225, 12234), (98788, 12234) ;

Expected Output: 12299 Rose 6, 34856 Angela 6, 79345 Frank 4, 81041 Lisa 1

```

We create a new table by integrateing the rows in the two tables using an `INNER JOIN` clause on the condition that `hackers.hacker_id = challlenges.hacker_id`. In addition, we use `COUNT(*)` with a `GROUP BY hackers.hacker_id, hackers.name` to return the number of challenges that each hacker has created.

```sql
SELECT h.hacker_id, h.name, COUNT(*) AS c_1 FROM c_hackers AS h
    INNER JOIN c_challenges AS c USING(hacker_id)
    GROUP BY h.hacker_id, h.name;

```

Specifically, the column `c_1` contains the numbers of challenges that each hacker has created. The output from the query is as in *Table 4*. Note that, `Andrew` and `Patrick` have created the same number of challenges and this is lesser than the maximum number of challenges created by a single hacker, that is `6`. To this end, the two rows have to be excluded from the result. Thus, we will employ the `HAVING` clause to filter the resulting table and the query to return the vector of interest is as follows.

We write a sub-query to return the non-duplicate entries in `c_1`, that is `4, 1` in *Table 4*. Conse-

Table 4: Challenges I

hacker_id	name	c_1
12299	Rose	6
34856	Angela	6
79345	Frank	4
80491	Patrick	3
12234	Andrew	3
81041	Lisa	1

quently, we use the condition `c_1 IN non-duplicate_entries` to return the associated rows. For a cleaner query, we will first create a new table that consists only column `c_1`.

```
-- Table with variable c_1

CREATE TABLE IF NOT EXISTS temp
  AS (SELECT COUNT(*) c_1 FROM c_challenges
      GROUP BY hacker_id);

-- Query to return non-duplicate entries in c_1

SELECT c_1 FROM temp
  GROUP BY c_1
  HAVING COUNT(c_1) = 1;

```

These queries return a list of entries that are non-duplicating in column `c_1`. Thus, we use the conditional statement `c_1 IN (···)` to return the associated rows. In addition, we also have to return the rows associated with hackers who have created the most challenges, that includes `12299 Rose 6, 34856 Angela 6` in *Table 4*. Thus, `c_1 = (SELECT MAX(c_1) FROM temp)` has to be implemented together in addition to the condition discussed above using a logical function `OR` in the `HAVING` clause.

```
CREATE TABLE IF NOT EXISTS temp
  AS (SELECT COUNT(*) AS c_1 FROM c_challenges
  GROUP BY hacker_id);

SELECT h.hacker_id, h.name, COUNT(*) c_1 FROM c_hackers AS h
  INNER JOIN c_challenges AS c ON h.hacker_id = c.hacker_id
  GROUP BY h.hacker_id, h.name
  HAVING c_1 IN
    (SELECT c_1 FROM temp
    GROUP BY c_1
    HAVING COUNT(*) = 1)
  OR c_1 = (SELECT MAX(c_1) FROM temp)
```

```
    ORDER BY c_1 DESC, h.hacker_id;

```

Listing 29: Challenges

2.8 Basic Join: Ollivander's Inventory

Problem: Hermione decides the best way to choose is by determining the minimum number of gold galleons needed to buy each non-evil wand of high power and age. Write a query to print the `id`, `age`, `coins_needed`, and `power` of the wands that Ron's interested in, sorted in order of descending `power`. If more than one wand has same power, sort the result in order of descending `age`.

Sample Input:

```
CREATE TABLE IF NOT EXISTS o_wands_property(
  code INT,
  age INT,
  is_evil INT,
  CONSTRAINT prim_key PRIMARY KEY(code),
  CONSTRAINT uniq_key UNIQUE(age),
  CONSTRAINT bin_key CHECK(is_evil IN (0, 1))
);

CREATE TABLE IF NOT EXISTS o_wands(
  id INT AUTO_INCREMENT,
  code INT,
  coins_needed INT,
  power INT,
  CONSTRAINT prim_key PRIMARY  KEY(id),
  CONSTRAINT f_key FOREIGN KEY(code)  REFERENCES o_wands_property(code)
);

INSERT INTO  o_wands_property(code, age, is_evil)
  VALUES
  (1, 45, 0), (2, 40, 0), (3, 4, 1), (4, 20, 0), (5, 17, 0);

INSERT INTO o_wands(code, coins_needed, power)
  VALUES
  (4, 3688, 8), (3, 9365, 3), (3, 7187, 10), (3, 734, 8), (1, 6020, 2),
  (2, 6773, 7), (3, 9873,  9), (3, 7721, 7), (1, 1647, 10), (4, 504, 5),
  (2, 7587, 5), (5, 9897, 10), (3, 4651, 8), (2, 5408, 1), (2, 6018, 7),
  (4, 7710, 5), (2, 8798, 7), (2, 3312, 3), (4, 7651, 6), (5, 5689, 3);

Expected Output: 9 45 1647 10, 12 17 9897 10, 1 20 3688 8, 15 40 6018 7,
                  19 20 7651 6, 11 40 7587 5, 10 20 504 5, 18 40 3312 3
                  20 17 5689 3, 5 45 6020 2, 14 40 5408 1

```

The wands that Ron is interested in has `is_evil = 0`. On top of that, for each configuration of (`code`, `age`, `power`), Ron prefers the particular wand that has the least `coins_needed`. To this end,

Table 5: Ollivander's Inventory I

id	code	age	power	coins_needed
5	1	45	2	6020
9	1	45	10	1647
14	2	40	1	5408
18	2	40	3	3312
11	2	40	5	7587
15	2	40	7	6018
6	2	40	7	6773
17	2	40	7	8798
⋮	⋮	⋮	⋮	⋮

we first create a new table that satisfies the condition that `is_evil = 0` using queries as follows.

```
SELECT tab_w.id, tab_w.code, tab_wp.age, tab_w.power, tab_w.coins_needed
FROM o_wands AS tab_w
    INNER JOIN o_wands_property AS tab_wp USING(code)
    WHERE is_evil = 0
    ORDER BY code, age, power, coins_needed;

```

The output from the query is as in *Table 5*. Note that, all entries in the resulting table have `is_evil = 0`. On top of that, wands with `id = 15, 6, 17` have the same configuration of `code, age, power` and in this case, Ron will only be interested with the particular wand that costs the least. To this end, we can employ the window function, `ROW_NUMBER() OVER(PARTITION BY arg1 ORDER BY arg2)` to assign row indexes to entries in the resulting table. The query is as follows.

```
SELECT tab_w.id, tab_w.code, tab_wp.age, tab_w.power, tab_w.coins_needed,
    ROW_NUMBER() OVER(PARTITION BY(tab_wp.age, tab_w.power)
                          ORDER BY tab_w.coins_needed) AS var_rn
FROM o_wands AS tab_w
    INNER JOIN o_wands_property AS tab_wp USING(code)
    WHERE is_evil = 0
    ORDER BY code, age, power, coins_needed;

```

The output is as *Table 6*. Where column `var_rn` ranks the entries by treating each configuration of (`age, power`) as a sub-category using the `PARTITION BY age, power` argument in the `OVER()` clause. In addition, the `ORDER BY coins_needed` argument in the `OVER()` clause is in-

Table 6: Sub-Table II - Ollivander's Inventory

id	code	age	power	coins_needed	var_rn
5	1	45	2	6020	1
9	1	45	10	1647	1
14	2	40	1	5408	1
18	2	40	3	3312	1
11	2	40	5	7587	1
15	2	40	7	6018	1
6	2	40	7	6773	2
17	2	40	7	8798	3
⋮	⋮	⋮	⋮	⋮	

tended to first reorder the entries in each sub-category by the purchasing cost in an ascending order before assigning the row indexes. Consequently, the entries with `var_rn = 1` have the lowest `coins_needed` compared to other wands with the same `age` and `power`. Thus, we return the columns of interest from this new table using the conditional statement `var_rn = 1`.

```
SELECT id, age, coins_needed, power FROM
  (SELECT tab_w.id, tab_wp.age, tab_w.power, tab_w.coins_needed,
    ROW_NUMBER() OVER(PARTITION BY(tab_wp.age, tab_w.power)
                          ORDER BY coins_needed) AS var_rn
    FROM o_wands AS tab_w
    INNER JOIN o_wands_property AS tab_wp USING(code)
    WHERE is_evil = 0
    ORDER BY code, age, power, coins_needed) AS sq1
  WHERE var_rn = 1
  ORDER BY power DESC, age DESC;

```

2.9 Advanced Join: Placements

Problem: Write a query to output the names of those students whose best friends got offered a higher salary than them. Names must be ordered by the salary amount offered to the best friends. It is guaranteed that no two students got same salary offer.

Sample Input:

```
CREATE TABLE IF NOT EXISTS p_students(
  id INT AUTO_INCREMENT,
  name VARCHAR(100),
  CONSTRAINT prim_key PRIMARY KEY(id)
);
```

```sql

CREATE TABLE IF NOT EXISTS p_packages(
  id INT,
  salary DECIMAL(6,2),
  CONSTRAINT p_f_key1 FOREIGN KEY(id) REFERENCES p_students(id),
  CONSTRAINT p_uniq_key1 UNIQUE(salary)
);

CREATE TABLE IF NOT EXISTS p_friends(
  id INT,
  friend_id INT,
  CONSTRAINT p_f_key2 FOREIGN KEY(id) REFERENCES  p_students(id),
  CONSTRAINT p_f_key3 FOREIGN KEY(id) REFERENCES p_students(id)
);

INSERT INTO p_students(name)
  VALUES
  ('Ashley'), ('Samantha'), ('Julia'), ('Scarlet');

INSERT INTO p_packages(id, salary)
  VALUES
  (1, 15.20), (2, 10.06), (3, 11.55), (4, 12.12);

INSERT INTO p_friends(id, friend_id)
  VALUES
  (1, 2), (2, 3), (3, 4), (4,1);

Expected Output: Samantha, Julia, Scarlet

```

We can first create an integrated table with columns `id`, `name`, `salary AS student_income`, `friend_id`. The entries in `salary` indicates the income level of the associated student's id. This table can be created using the `INNER JOIN` clause as follows.

```sql
SELECT s.id, s.name, p1.salary AS student_income, f.friend_id FROM p_students AS s
  INNER JOIN p_packages AS p1 USING(id)
  INNER JOIN p_friends AS f ON s.id = f.id;

```

The output is as in *Table 7*. Note that, we can perform another `INNER JOIN packages AS p2` clause on the condition that `f.id = p2.id` to include the friends' income levels in the table. Consequently, we add another column `p2.salary AS friend_salary` into the table. To this end, the entries with `friend_salary > student_salary` are of our interest. In addition, we are only required to return the `s.name` column and thus, we omit other arguments in the `SELECT` clause.

```sql
SELECT s.name FROM p_students AS s
    INNER JOIN p_friends AS f USING(id)
```

Table 7: Placements I

id	name	student_income	friend_id
1	Ashley	15.20	2
2	Samantha	10.06	3
3	Julia	11.55	4
4	Scarlet	12.12	1

```
      INNER JOIN p_packages AS p1 ON s.id = p1.id
      INNER JOIN p_packages AS p2 ON f.id = p2.id
    WHERE tab_p2.salary >  tab_p1.salary
    ORDER BY tab_p2.salary;
```

2.10 Advanced Join: SQL Project Planning

Problem: You are given a table, Projects, containing three columns: `task_id`, `start_date` and `end_date`. It is guaranteed that the difference between the `end_date` and the `start_date` is equal to 1 day for each row in the table. If the `end_date` of the tasks are consecutive, then they are part of the same project. Samantha is interested in finding the total number of different projects completed. Write a query to output the start and end dates of projects listed by the number of days it took to complete the project in ascending order. If there is more than one project that have the same number of completion days, then order by the start date of the project.

Sample Input:

```
CREATE TABLE IF NOT EXISTS spp_projects(
  id INT AUTO_INCREMENT,
  start_date DATE,
  end_date DATE,
  CONSTRAINT prim_key PRIMARY KEY(id)
);

INSERT INTO spp_projects(start_date, end_date)
  VALUES
  ('2015-10-01', '2015-10-02'), ('2015-10-02', '2015-10-03'),
  ('2015-10-03', '2015-10-04'), ('2015-10-13', '2015-10-14'),
  ('2015-10-14', '2015-10-15'), ('2015-10-28', '2015-10-29'),
  ('2015-10-30', '2015-10-31');

Expected Output: 2015-10-28 2015-10-29, 2015-10-30 2015-10-31,
                  2015-10-13 2015-10-15, 2015-10-01 2015-10-04
```

From the sample input, note that if the `end_date` is also a `start_date`, then one is working on

Table 8: SQL Project Planning I

Start_Date	End_Date
2015-10-01	2015-10-04
2015-10-01	2015-10-15
2015-10-01	2015-10-29
2015-10-01	2015-10-31
2015-10-13	2015-10-15
2015-10-13	2015-10-29
2015-10-13	2015-10-31
⋮	⋮

the same project. On the other hand, the start date of a project is never an end date. Thus, we can first extract these entries from the table using query as follows.

```
-- Extract Start Date --

SELECT start_date FROM spp_projects
  WHERE start_date NOT IN (SELECT end_date FROM spp_projects);

-- Output: 2015-10-01, 2015-10-13, 2015-10-28, 2015-10-30

-- Extract End Date --

SELECT end_date FROM spp_projects
  WHERE end_date NOT IN (SELECT start_date FROM spp_projects);

-- Output: 2015-10-04, 2015-10-15, 2015-10-29, 2015-10-31

```

For any `start_date` of a project, say `2015-10-13`; we know that the associated `end_date` has to be after `2015-10-13`. To this end, the set of candidates of end dates consists of all `end_date`s after the `start_date` of a project. Thus, we perform an `INNER JOIN` for the two columns from the above queries on the condition that `end_date > start_date`.

From *Table 8*, note that for every entry in `start_date`, the true `end_date` will be the one that is closest to the actual start date. For instance, the project that started on `2015-10-01` has the true `end_date = 2015-10-04`. Thus, we implement `GROUP BY start_date` and return `start_date` and `MIN(end_date)` from this new table that we created.

In addition, the output has to be ordered by the time taken to complete a project in ascending order using `ORDER BY` clause with `DATEDIFF(start_date, MIN(end_date))` function. Note that we employ `MIN(end_date)` as the argument for the `DATEDIFF` function as the `MIN(end_date)` refers to the true `end_date` in the resulting table.

```
SELECT sq1.start_date, MIN(sq2.end_date) AS var_end FROM
      (SELECT start_date FROM spp_projects
      WHERE start_date NOT IN (SELECT end_date FROM spp_projects)) sq1
  INNER JOIN
      (SELECT end_date FROM spp_projects
      WHERE end_date NOT IN (SELECT start_date FROM spp_projects)) sq2
  ON sq2.end_date > sq1.start_date
GROUP BY sq1.start_date
ORDER BY DATEDIFF(MIN(sq2.end_date), sq1.start_date), sq1.start_date ;
```

3 Problems on Aggregation

3.1 Aggregation: Revising Aggregations - The Count Function

Problem: Query a count of the number of cities in `city` having a `population` larger than *100,000*.

We create a sub-table that consists only of entries with population size to be greater than *100,000* using a `WHERE` clause. In addition, a `COUNT(*)` function is applied to compute the number of rows in the resulting sub-table.

```
    SELECT COUNT(*) FROM city
      WHERE population > 100000;

```

Listing 30: Revising Aggregations - The Count Function

3.2 Aggregation: Revising Aggregations - The Sum Function

Problem: Query the total population of all cities in `CITY` where `District` is California.

We create a sub-table with entries that satisfy the constraint that `district = 'California'`. The `SUM(population)` function is then employed to return the sum of all entries in column `population` in the resulting table.

```
    SELECT SUM(population) FROM city
      WHERE district = 'California';

```

Listing 31: Revising Aggregations - The Sum Function

Problem Japan Population can also be solved using the same query by replacing the conditional statement from `district = 'California'` to `countrycode = 'JPN'`.

3.3 Aggregation: Revising Aggregations - Averages

Problem: Query the average population of all cities in `city` where `district` is California.

We create a sub-table as in *Problem 3.2* and apply `AVG(population)` to the resulting table. Alternatively, one can also use `SUM(population)/COUNT(population)` in place of `AVG(population)` to obtain the similar result.

```sql
SELECT AVG(population) FROM city
  WHERE district = 'California';
```

Listing 32: Revising Aggregations - Averages

3.4 Aggregation: Population Density Difference

Problem: Query the difference between the maximum and minimum populations in `city`.

We use the functions `MAX(population)` and `MIN(population)` to seek the maximum and minimum values in column `population`. This problem can be solved by computing the difference between them.

```sql
SELECT MAX(population) - MIN(population) FROM city;
```

3.5 Aggregation: The Blunder

Problem: Samantha was tasked with calculating the average monthly salaries for all employees in the `employees` table, but did not realize her keyboard's key, 0 was broken until after completing the calculation. She wants your help finding the difference between her miscalculation (using salaries with any zeros removed), and the actual average salary.

Sample Input:

```sql
CREATE TABLE IF NOT EXISTS tb_employees(
  id INT AUTO_INCREMENT,
  name VARCHAR(100),
  salary INT,
  CONSTRAINT prim_key PRIMARY KEY(id),
  CONSTRAINT range_key CHECK(salary > 0 AND salary < POWER(10,5))
);

INSERT INTO tb_employees(name, salary)
  VALUES
    ('Kristeen', 1420), ('Ashley', 2006), ('Julia', 2210), ('Maria', 3000);
```

We use the function `REPLACE(salary, 0, '')` to drop the '0' from the salary of each employee. Specifically, `SELECT REPLACE(1001,0,'')` produces an output, `11`. Thus, we can seek the difference between the average computed using the original `salary` column and the sample mean computed with the resulting column using the `REPLACE` function to compute the error made in the calculation. In addition, function `CEIL` is employed to round up the output to the nearest integer.

```sql
SELECT CEIL(AVG(salary) - AVG(REPLACE(salary, 0, ''))) FROM tb_employees;
```

3.6 Aggregation: Top Earners

Problem: We define an employee's total earnings to be their monthly $Salary * Months$ worked, and the maximum total earnings to be the maximum total earnings for any employee in the `employee` table. Write a query to find the maximum total earnings for all employees as well as the total number of employees who have maximum total earnings. Then print these values as space-separated integers.

We create a new variable `salary*months`. In addition, we use the function, `COUNT(employee_id)` with `GROUP BY salary*months`. This returns the numbers of employees for each total earnings level. However, we have to order the output by `salary*months` in a descending order and only to return the first row since we are interested with the number of employees who are making the highest total earnings. To this end, we use `ORDER BY salary*months` with `LIMIT 1` to return only the desired output.

```sql
SELECT salary*months, COUNT(employee_id) FROM employee
  GROUP BY salary*months
  ORDER BY salary*months DESC
  LIMIT 1;
```

Listing 33: Top Earners

3.7 Aggregation: Weather Observation Station 2

Problem: Query the following two values from the `station` table:

1. The sum of all values in `lat_n` rounded to a scale of 2 decimal places.
2. The sum of all values in `long_w` rounded to a scale of 2 decimal places.

We use the aggregate function, `COUNT` on both columns `lat_n` and `long_w`. In addition, the numeric function, `ROUND` is required to round the output to 2 decimal places. Specifically, the function `ROUND(arg1, n)` rounds the numerical argument, `arg1` to *n* decimal places.

```sql
SELECT ROUND(SUM(lat_n), 2), ROUND(SUM(long_w), 2) FROM station;

```

Listing 34: Weather Observation Station 2

3.8 Aggregation: Weather Observation Station 13

Problem: Query the sum of Northern Latitudes (`lat_n`) from `station` having values greater than 38.7880 and less than 137.2345. Truncate your answer to 4 decimal places.

Similarly, the `SUM(lat_n)` function is required to compute the aggregate of column `lat_n`; while, function `ROUND(SUM(lat_n), 4)` truncates the output to 4 decimal places as required. However, this statistic has to be computed based on the filtered table. Specifically, all entries in the filtered table should satisfy the constraint that `lat_n > 38.7880 AND lat_n < 137.2345`. Thus, a sub-table using the conditional statement in a `WHERE` clause is required.

```sql
SELECT ROUND(SUM(lat_n), 4) FROM station
  WHERE lat_n > 38.7880 AND lat_n < 137.2345;

```

Listing 35: Weather Observation Station 13

3.9 Aggregation: Weather Observation Station 14

Problem: Query the greatest value of the Northern Latitudes (`lat_n`) from STATION that is less than 137.2345. Truncate your answer to 4 decimal places.

We create a sub-table that consists of rows that satisfy the condition `lat_n < 137.2345`. Thus, the greatest values in column `lat_n` that is smallest than 137.2345 can be computed using the function `MAX(lat_n)` on the sub-table and function `ROUND` is employed to truncate the output to 4 decimal places.

```sql
SELECT ROUND(MAX(lat_n), 4) FROM station
  WHERE lat_n < 137.2345;

```

Listing 36: Weather Observation Station 14

Problem Weather Observation Station 16 can be solved using the same argument. However, we use `MIN(lat_n)` in place of `MAX(lat_n)` and change the conditional statement to `lat_n > 38.7780`.

3.10 Aggregation: Weather Observation Station 15

Problem: Query the Western Longitude (`long_w`) for the largest Northern Latitude (`lat_n`) in `station` that is less than 137.2345. Round your answer to 4 decimal places.

We create a sub-table as in *Problem 3.9*. However, we will re-order the rows in the resulting table by `lat_n` in a descending order such that the first row in the output contains the maximum value of `lat_n` and the associated entry for `long_w`. Thus, only the `long_w` entry in the first row is returned and the result is truncated using the `ROUND` function.

```sql
SELECT ROUND(long_w, 4) FROM station
  WHERE lat_n < 137.2345
  ORDER BY lat_n DESC
  LIMIT 1;

```

Listing 37: Weather Observation Station 15

The same argument can be applied to solve *problem Weather Observation Station 17*, with a few modifications. Specifically, we use `lat_n > 38.7780` as the conditional statement. In addition, the outputs are reordered using `ORDER BY lat_n ASC` to return the desired output.

3.11 Aggregation: Weather Observation Station 18

Problem: Consider $P_1(min_{lat}, min_{long})$ and $P_2(max_{lat}, max_{long})$ to be two points on a 2D plane. Specifically, min_i and max_i refers to the minimum and maximum values in column i, for $i =$ `lat_n`, `long_w`. Query the Manhattan Distance between 2 points and round it to a scale of 4 decimal places.

The Manhattan Distance between two coordinates, (x_1, y_1) and (x_2, y_2) is defined as $|x_1 - x_2| + |y_1 - y_2|$. To this end, we use the function `ABS` to seek for the absolute values of the inputs. In addition, we employ functions `MAX` and `MIN` to compute the maximum and minimum values in columns `lat_n` and `long_w`. The result is also truncated using the `ROUND` function to 4 decimal places.

```sql
SELECT
  ROUND(ABS(MIN(lat_n) - MAX(lat_n)) + ABS(MIN(long_w) - MAX(long_w)), 4)
FROM station;

```

Listing 38: Weather Observation Station 18

3.12 Aggregation: Weather Observation Station 19

Problem: Consider $P_1(min_{lat}, min_{long})$ and $P_2(max_{lat}, max_{long})$ to be two points on a 2D plane. Specifically, min_i and max_i refers to the minimum and maximum values in column i, for $i =$ `lat_n`, `long_w`. Query the Euclidean Distance between 2 points and round it to a scale of 4 decimal places.

Note that the Euclidean Distance between two coordinates, (x_1, y_1) and (x_2, y_2) is defined as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Thus, the same query in *Problem 3.11* is still applicable. However, we use function, `POWER(arg1, 2)` in place of `ABS(arg1)` to raise the difference between the maximum and minimum values in columns `lat_n` and `long_w` to the power of 2. In addition, we use `POWER(arg2, 1/2)` to seek for the square root of the resulting value. Function `ROUND` is also employed to truncate the output to 4 decimal places.

```
SELECT
  ROUND(POWER(POWER(MIN(lat_n) - MAX(lat_n), 2) + POWER(MIN(long_w) - MAX(long_w), 2)
,1/2), 4)
FROM station;

```

Listing 39: Weather Observation Station 19

3.13 Aggregation: Weather Observation Station 20

Problem: A median is defined as a number separating the higher half of a data set from the lower half. Query the median of the Northern Latitudes (`lat_n`) from `station` and round your answer to 4 decimal places.

The sample size in attribute `lat_n` determines the function that is required to compute the sample median. To this end, we first specify a variable, `@T` to be the number of observations in column `lat_n` using `SET @T := (SELECT COUNT(lat_n) FROM station);`. In addition, we create a new column that consists of the associated row number when entries in `lat_n` are arranged in an ascending order using the window function, `ROW_NUMBER()`.

If `MOD(@T,2) = 1`, the median of `lat_n` is equivalent to the `CEIL(@T/2)`th entry in the re-ordered column. Thus, the output is scalar and we have `AVG(arg1) = arg1`. On the other hand, the median is obtained by averaging the entries in row `@T/2` and `@T/2 + 1` in column `lat_n` when we have even number of observations. In this case, the `AVG` function returns the output as desired. The `ROUND` function is intended to truncate the output to 4 decimal places.

```
SET @T := (SELECT COUNT(*) FROM STATION);
```

```sql

SELECT ROUND(AVG(lat_n),4) FROM
  (SELECT lat_n,
  ROW_NUMBER() OVER(ORDER BY lat_n ASC) AS var_rn FROM station) sq1
WHERE
  CASE MOD(@T,2)
    WHEN 1 THEN var_rn = CEIL(@T/2)
    ELSE var_rn IN (@T/2, @T/2 + 1)
  END;

SET @T := NULL;

```
Listing 40: Weather Observation Station 20