

# SQL Questions in HackerRank: Explanation and Solution (V1)

Delroy Low

## ***Remark***

This article aims to provide guidance on solving the SQL problems in 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 required 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

<b>1</b>	<b>Problems on SELECT Clause</b>	<b>4</b>
1.1	Basic Select: Select All . . . . .	4
1.2	Basic Select: Select by ID . . . . .	4
1.3	Basic Select: Japanese Cities' Attributes . . . . .	4
1.4	Basic Select: Japanese Cities' Names . . . . .	5
1.5	Basic Select: Revising the Select Query I . . . . .	5
1.6	Basic Select: Revising the Select Query II . . . . .	5
1.7	Basic Select: Weather Observation Station 1 . . . . .	6
1.8	Basic Select: Weather Observation Station 3 . . . . .	6
1.9	Basic Select: Weather Observation Station 4 . . . . .	7
1.10	Basic Select: Weather Observation Station 5 . . . . .	7
1.11	Basic Select: Weather Observation Station 6 . . . . .	9
1.12	Basic Select: Weather Observation Station 7 . . . . .	9
1.13	Basic Select: Weather Observation Station 8 . . . . .	10
1.14	Basic Select: Weather Observation Station 9 . . . . .	10
1.15	Basic Select: Weather Observation Station 10 . . . . .	10
1.16	Basic Select: Higher Than 75 Marks . . . . .	11
1.17	Basic Select: Employee Names . . . . .	12
1.18	Basic Select: Employee Salaries . . . . .	13
1.19	Advanced Select: Type of Triangle . . . . .	13
1.20	Advanced Select: Binary Tree Nodes . . . . .	15
1.21	Advanced Select: The Pads . . . . .	16
1.22	Advanced Select: New Companies . . . . .	18
<b>2</b>	<b>Questions on JOIN Clause</b>	<b>21</b>
2.1	Basic Join: Population Census . . . . .	21
2.2	Basic Join: African Cities . . . . .	21
2.3	Basic Join: Average Population of Each Continent . . . . .	22
2.4	Basic Join: The Report . . . . .	22
2.5	Basic Join: Contest Leaderboard . . . . .	24
2.6	Basic Join: Top Competitors . . . . .	26
2.7	Basic Join: Challenges . . . . .	28
2.8	Basic Join: Ollivander's Inventory . . . . .	31
2.9	Advanced Join: Placements . . . . .	32
2.10	Advanced Join: SQL Project Planning . . . . .	33

<b>3</b>	<b>Problems on Aggregation</b>	<b>35</b>
3.1	Aggregation: Revising Aggregations - The Count Function . . . . .	35
3.2	Aggregation: Revising Aggregations - The Sum Function . . . . .	35
3.3	Aggregation: Revising Aggregations - Averages . . . . .	35
3.4	Aggregation: Population Density Difference . . . . .	36
3.5	Aggregation: The Blunder . . . . .	36
3.6	Aggregation: Top Earners . . . . .	37
3.7	Aggregation: Weather Observation Station 2 . . . . .	37
3.8	Aggregation: Weather Observation Station 13 . . . . .	38
3.9	Aggregation: Weather Observation Station 14 . . . . .	38
3.10	Aggregation: Weather Observation Station 15 . . . . .	39
3.11	Aggregation: Weather Observation Station 18 . . . . .	39
3.12	Aggregation: Weather Observation Station 19 . . . . .	40
3.13	Aggregation: Weather Observation Station 20 . . . . .	40

# 1 Problems on SELECT Clause

## 1.1 Basic Select: Select All

*Problem:* 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`.

```
1 SELECT * FROM city;  
2
```

Listing 1: Select All

## 1.2 Basic Select: Select by ID

*Problem:* 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.

```
1 SELECT * FROM city  
2 WHERE id = 1661;  
3
```

Listing 2: Select by ID

## 1.3 Basic Select: Japanese Cities' Attributes

*Problem:* 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 single constraint, `COUNTRYCODE = 'JPN'` in the `WHERE` clause.

```
1 SELECT * FROM city  
2 WHERE countrycode = 'JPN';  
3
```

Listing 3: Japanese Cities' Attributes

## 1.4 Basic Select: Japanese Cities' Names

*Problem:* 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.

```
1
2  SELECT name FROM city
3  WHERE countrycode = 'JPN';
4
```

Listing 4: Japanese Cities' Names

## 1.5 Basic Select: Revising the Select Query I

*Problem:* 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.

```
1  SELECT * FROM city
2  WHERE countrycode = 'USA' AND population > 100000;
3
```

Listing 5: Revising the Select Query I

## 1.6 Basic Select: Revising the Select Query II

*Problem:* 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.

```
1 SELECT name FROM city
2 WHERE countrycode = 'USA' AND population > 120000;
3
```

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. Thus, 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).

```
1 SELECT city, state FROM station;
2
```

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,  $a \in \mathcal{Z}$  is even if `MOD(a,2) = 0` and is odd if `MOD(a,2) = 1`.

```
1 SELECT DISTINCT(city) FROM station
2 WHERE MOD(id,2) = 0;
3
```

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.

```
1 SELECT COUNT(city) - COUNT(DISTINCT(city)) FROM station;
2
```

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 re-arrange the resulting entries with duplicates `arg1` with `arg2`. The first entry in the resulting table is returned using `LIMIT 1`.

```
1 -- Query to obtain the shortest city's name --
2
3 SELECT city, CHAR_LENGTH(city) FROM station
4
```

```

5      -- Order the table by character_length ascendingly to identify the shortest length;
6      -- then, the table is reorder by city's name for entries with the same character
7      -- length
8
9      ORDER BY CHAR_LENGTH(city) ASC, city ASC
10
11     -- This way, the first entry in the resulting table will be the entry with shortest
12     -- character length and the associated city's name comes first when ordered
13     -- alphabetically.
14
15     LIMIT 1;
16
17     --Query to obtain the longest city's name --
18
19     SELECT CITY, CHAR_LENGTH(city) FROM STATION
20     ORDER BY CHAR_LENGTH(city) DESC, CITY ASC
21     LIMIT 1;
22

```

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.

```

1      -- Query to obtain the shortest city's name --
2
3      SELECT CITY, CHAR_LENGTH(city) FROM station
4      WHERE CHAR_LENGTH(city) = (SELECT MIN(CHAR_LENGTH(city)) FROM station)
5      ORDER BY city ASC
6      LIMIT 1;
7
8      --Query to obtain the longest city's name --
9
10     SELECT CITY, CHAR_LENGTH(city) FROM station
11     WHERE CHAR_LENGTH(city) = (SELECT MAX(CHAR_LENGTH(city)) FROM station)
12     ORDER BY city ASC
13     LIMIT 1;
14

```

```

1      -- Single Query with Window Function --
2
3      SELECT city, CHAR_LENGTH(city) FROM
4      (SELECT city, CHAR_LENGTH(city),
5      ROW_NUMBER() OVER(PARTITION BY CHAR_LENGTH(city) ORDER BY city) rn,
6      MAX(CHAR_LENGTH(city)) OVER() max,
7      MIN(CHAR_LENGTH(city)) OVER() min FROM station) sq1
8      WHERE
9      CHAR_LENGTH(city) = min AND rn = 1 OR
10     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.

```

1  SELECT CITY FROM STATION
2  WHERE LEFT(CITY,1) IN ('A','E','I','O','U');
3

```

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 where `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.

```

1  SELECT DISTINCT(CITY) FROM STATION
2  WHERE RIGHT(CITY,1) IN ('a', 'e', 'i', 'o', 'u');
3

```

Listing 12: Weather Observation Station 7

### 1.13 Basic Select: Weather Observation Station 8

*Problem:* Query the list of 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 statement in *Problem 1.11* and *Problem 1.12* using logical operator, AND. Alternatively, we employ LOWER(string\_variable) to convert the letter extracted RIGHT(string\_variable),1 to lowercase and use the same conditional statement in the IN clauses.

```
1 SELECT CITY FROM STATION
2 WHERE LOWER(LEFT(CITY,1)) IN ('a','e','i','o','u')
3 AND RIGHT(CITY,1) IN ('a','e','i','o','u');
4
```

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.

```
1 SELECT DISTINCT(CITY) FROM STATION
2 WHERE !(LEFT(CITY,1) IN ('A','E','I','O','U'));
3
```

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.

```
1 SELECT DISTINCT(CITY) FROM STATION
2 WHERE RIGHT(CITY,1) NOT IN ('a','e','i','o','u');
3
```

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;

```
1 CREATE TABLE IF NOT EXISTS higher_than_75_marks(
2     id INT AUTO_INCREMENT,
3     name VARCHAR(100),
4     marks INT,
5     CONSTRAINT prim_key PRIMARY KEY(id),
6     CONSTRAINT score_con CHECK(marks >= 0 AND marks <=100)
7 );
8
9 INSERT INTO higher_than_75_marks(name, marks)
10 VALUES
11     ('Ashley', 81), ('Samantha', 75),
12     ('Julia', 76), ('Belvet', 84);
13
14 -- Expected Output: Ashley, Julia, Belvet --
15
```

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.

```

1  SELECT name FROM STUDENTS
2  WHERE marks > 75
3  ORDER BY RIGHT(name,3) ASC, id ASC;
4

```

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:*

```

1  CREATE TABLE IF NOT EXISTS employee_names(
2      employee_id INT,
3      name VARCHAR(100),
4      months INT,
5      salary INT,
6      CONSTRAINT prim_key PRIMARY KEY(employee_id),
7      CONSTRAINT non_neg_check1 CHECK(months >= 0),
8      CONSTRAINT non_neg_check2 CHECK(salary >= 0)
9  );
10
11 INSERT INTO employee_names(employee_id, name, months, salary)
12 VALUES
13     (12228, 'Rose', 15, 1968), (33645, 'Angela', 1, 3443),
14     (45692, 'Frank', 17, 1608), (56118, 'Patrick', 7, 1345),
15     (59725, 'Lisa', 11, 2330), (74197, 'Kimberly', 16, 4372),
16     (78454, 'Bonnie', 8, 1771), (83565, 'Michael', 6, 2017),
17     (98607, 'Todd', 5, 3396), (99989, 'Joe', 9, 3573);
18
19 Expected Output:  Angela, Bonnie, Frank, Joe, Kimberly, Lisa, Michael, Patrick, Rose,
20                  Todd
21

```

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.

```

1  SELECT name FROM employee
2  ORDER BY name ASC;
3

```

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.

```
1 SELECT name FROM employee
2 WHERE salary > 2000 AND months < 10
3 ORDER BY name ASC;
4
```

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:**

```
1 CREATE TABLE IF NOT EXISTS type_of_triangle(
2     A INT,
3     B INT,
4     C INT
5 );
6
7 INSERT INTO type_of_triangle(A,B,C)
8 VALUES
9     (20,20,23), (20,20,20),
10    (20,21,22), (13,14,30);
```

```

11 Expected Output: Isosceles, Equilateral, Scalene, Not A Triangle
12
13

```

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.

```

1  SELECT
2      CASE
3          WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'
4          WHEN A = B AND A = C THEN 'Equilateral'
5          WHEN !(A = B) AND !(A = C) THEN 'Scalene'
6          ELSE 'Isosceles'
7      END
8  FROM triangles;
9

```

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.

```

1  -- Query 1 --
2
3  SELECT
4      CASE
5          WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'
6          WHEN A = B OR A = C OR B = C THEN 'Isosceles'
7          WHEN A = B AND A = C THEN 'Equilateral'
8          ELSE 'Scalene'
9      END
10 FROM triangles;
11
12 -- Query 2 --
13
14 SELECT
15     CASE
16         WHEN A = B OR A = C OR B = C THEN 'Isosceles'
17         WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'
18         WHEN A = B AND A = C THEN 'Equilateral'
19         ELSE 'Scalene'
20     END

```

```

21 FROM triangles;
22

```

## 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:**

```

1 CREATE TABLE IF NOT EXISTS binary_tree_nodes (
2     N INT,
3     P INT -- Parent Node of N
4 );
5
6 INSERT INTO binary_tree_nodes(N, P)
7 VALUES
8     (1,2), (3,2), (6,8), (9,8),
9     (2,5), (8,5), (5,NULL);
10
11 Expected Output: 1 Leaf, 2 Inner, 3 Leaf, 5 Root, 6 Leaf, 8 Inner, 9 Leaf
12

```

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.

```

1 CASE
2     WHEN P IS NULL THEN 'Parent'
3     WHEN N NOT IN (
4         SELECT DISTINCT(P) FROM BST
5         WHERE P IS NOT NULL
6     ) -- The sub-query
7         THEN 'Leaf'
8     ELSE 'Inner'
9 END
10

```

In addition, we use `CONCAT(N, ' ', ...)` function with a `CASE` clause in place of `...` to return the output in the format required. Specifically, a `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`.

```

1      SELECT
2          CONCAT(N, ' ',
3              CASE
4                  WHEN P IS NULL THEN 'Root'
5                  WHEN !(
6                      N IN (SELECT DISTINCT(P) FROM BST
7                          WHERE P IS NOT NULL)
8                      ) THEN 'Leaf'
9                  ELSE 'Inner'
10             END
11         )
12     FROM BST
13     ORDER BY N;
14

```

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 occurrences 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:*



```

1 CREATE TABLE IF NOT EXISTS the_pads(
2     name VARCHAR(100),
3     occupation VARCHAR(100)
4 );
5
6 INSERT INTO the_pads(name, occupation)
7 VALUES
8     ('Samantha', 'Doctor'), ('Julia', 'Actor'), ('Maria', 'Actor'),
9     ('Meera', 'Singer'), ('Ashely', 'Professor'), ('Ketty', 'Professor'),
10    ('Christine', 'Professor'), ('Jane', 'Actor'), ('Jenny', 'Doctor'),
11    ('Priya', 'Singer');
12
13 Expected Output: Ashely(P), Christeen(P), Jane(A), Jenny(D), Julia(A), Ketty(P),
14                  Maria(A), Meera(S), Priya(S), Samantha(D),
15                  There are a total of 2 doctors., There are a total of 2 singers.,
16                  There are a total of 3 actors., There are a total of 3 professors.
17

```

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`. This can be verified by running a `SELECT COUNT(name), occupation ...` query. The outputs are then ordered by `COUNT(name)` and secondary sorted by `occupation`.

```

1 SELECT
2     CONCAT(name, '(' , LEFT(occupation,1) , ')')

```

```

3      FROM occupations
4      ORDER BY name;
5
6      SELECT
7          CONCAT('There are a total of ', COUNT(name), ' ', LOWER(occupation), 's.')
8      FROM occupations
9      GROUP BY occupation
10     ORDER BY COUNT(name) ASC, occupation ASC;
11

```

Listing 21: The Pads

## 1.22 Advanced Select: New Companies

*Problem:* 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.

*Sample Input:*

```

1      CREATE TABLE IF NOT EXISTS company(
2          company_code VARCHAR(100),
3          founder VARCHAR(100),
4          CONSTRAINT prim_key1 PRIMARY KEY(company_code)
5      );
6
7      CREATE TABLE IF NOT EXISTS lead_manager(
8          lead_manager_code VARCHAR(100),
9          company_code VARCHAR(100),
10         CONSTRAINT prim_key2 PRIMARY KEY(lead_manager_code),
11         CONSTRAINT f_key_2_1 FOREIGN KEY(company_code) REFERENCES company(company_code)
12     );
13
14     CREATE TABLE IF NOT EXISTS senior_manager(
15         senior_manager_code VARCHAR(100),
16         lead_manager_code VARCHAR(100),
17         company_code VARCHAR(100),
18         CONSTRAINT prim_key3 PRIMARY KEY(senior_manager_code),
19         CONSTRAINT f_key_3_1 FOREIGN KEY(company_code) REFERENCES company(company_code),
20         CONSTRAINT f_key_3_2 FOREIGN KEY(lead_manager_code) REFERENCES lead_manager(
21     lead_manager_code)
22     );
23
24     CREATE TABLE IF NOT EXISTS manager(
25         manager_code VARCHAR(100),
26         senior_manager_code VARCHAR(100),
27         lead_manager_code VARCHAR(100),

```

```

27     company_code VARCHAR(100),
28     CONSTRAINT prim_key4 PRIMARY KEY(manager_code),
29     CONSTRAINT f_key_4_1 FOREIGN KEY(company_code) REFERENCES company(company_code),
30     CONSTRAINT f_key_4_2 FOREIGN KEY(lead_manager_code) REFERENCES lead_manager(
lead_manager_code),
31     CONSTRAINT f_key_4_3 FOREIGN KEY(senior_manager_code) REFERENCES senior_manager(
senior_manager_code)
32 );
33
34 CREATE TABLE IF NOT EXISTS employee(
35     employee_code VARCHAR(100),
36     manager_code VARCHAR(100),
37     senior_manager_code VARCHAR(100),
38     lead_manager_code VARCHAR(100),
39     company_code VARCHAR(100),
40     CONSTRAINT prim_key5 PRIMARY KEY(employee_code),
41     CONSTRAINT f_key_5_1 FOREIGN KEY(company_code) REFERENCES company(company_code),
42     CONSTRAINT f_key_5_2 FOREIGN KEY(lead_manager_code) REFERENCES lead_manager(
lead_manager_code),
43     CONSTRAINT f_key_5_3 FOREIGN KEY(senior_manager_code) REFERENCES senior_manager(
senior_manager_code),
44     CONSTRAINT f_key_5_4 FOREIGN KEY(manager_code) REFERENCES manager(manager_code)
45 );
46
47 INSERT INTO company(company_code, founder)
48 VALUES
49     ('C1', 'Monika'), ('C2', 'Samantha');
50
51 INSERT INTO lead_manager(lead_manager_code, company_code)
52 VALUES
53     ('LM1', 'C1'), ('LM2', 'C2');
54
55 INSERT INTO senior_manager(senior_manager_code, lead_manager_code, company_code)
56 VALUES
57     ('SM1', 'LM1', 'C1'), ('SM2', 'LM1', 'C1'), ('SM3', 'LM2', 'C2');
58
59 INSERT INTO manager(manager_code, senior_manager_code, lead_manager_code,
60                     company_code)
61 VALUES
62     ('M1', 'SM1', 'LM1', 'C1'), ('M2', 'SM3', 'LM2', 'C2'),
63     ('M3', 'SM3', 'LM2', 'C2');
64
65 INSERT INTO employee(employee_code, manager_code, senior_manager_code,
66                     lead_manager_code, company_code)
67 VALUES
68     ('E1', 'M1', 'SM1', 'LM1', 'C1'), ('E2', 'M1', 'SM1', 'LM1', 'C1'),
69     ('E3', 'M2', 'SM3', 'LM2', 'C2'), ('E4', 'M3', 'SM3', 'LM2', 'C2');
70
71 Expected Output: C1 Monika 1 2 1 2, C2 Samantha 1 1 2 2
72

```

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.

```
1 SELECT c.company_code, c.founder, COUNT(DISTINCT(l.lead_manager_code))
2 FROM company AS c
3     INNER JOIN lead_manager AS l ON c.company_code = l.company_code
4 GROUP BY c.company_code, c.founder;
5
6 -- Output: C1 Monika 1, C2 Samantha 1
7
```

We then treat the outputs as the new table and perform another `INNER JOIN` clause with table `senior_manager` on the condition that `c.company_code = senior_manager.company_code` and to compute the number of distinctive senior manager working in each company using the `COUNT(DISTINCT(col_name))` function. The query is as follows.

```
1 SELECT c.company_code, c.founder, COUNT(DISTINCT(l.lead_manager_code))
2 FROM company AS c
3     INNER JOIN lead_manager AS l ON c.company_code = l.company_code
4     INNER JOIN senior_manager AS s ON c.company_code = s.company_code
5 GROUP BY c.company_code, c.founder;
6
7 -- Output: C1 Monika 1 2, C2 Samantha 1 1
8
```

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

```
1 SELECT c.company_code, c.founder,
2     COUNT(DISTINCT(l.lead_manager_code)), COUNT(DISTINCT(s.senior_manager_code)),
3     COUNT(DISTINCT(m.manager_code)), COUNT(DISTINCT(e.employee_code))
4 FROM company AS c
5     INNER JOIN lead_manager AS l ON c.company_code = l.company_code
6     INNER JOIN senior_manager AS s ON c.company_code = s.company_code
7     INNER JOIN manager AS m ON c.company_code = m.company_code
8     INNER JOIN employee AS e ON c.company_code = e.company_code
9 GROUP BY c.company_code, c.founder
10 ORDER BY c.company_code;
11
```

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.

```
1 SELECT SUM(CITY.POPULATION) FROM CITY
2     INNER JOIN COUNTRY
3     ON CITY.COUNTRYCODE = COUNTRY.CODE
4     WHERE COUNTRY.CONTINENT = 'Asia';
5
```

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.

```
1 SELECT CITY.NAME FROM CITY
2     INNER JOIN COUNTRY
3     ON CITY.COUNTRYCODE = COUNTRY.CODE
4     WHERE COUNTRY.CONTINENT = 'Africa';
5
```

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, note that the columns of interest are located in two different tables for in this problem. 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).

```
1 SELECT COUNTRY.CONTINENT, FLOOR(AVG(CITY.POPULATION)) FROM COUNTRY
2     INNER JOIN CITY ON COUNTRY.CODE = CITY.COUNTRYCODE
3     GROUP BY COUNTRY.CONTINENT;
4
```

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 Grade

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 Mark. 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:

```

1  CREATE TABLE IF NOT EXISTS the_report_students(
2      id INT AUTO_INCREMENT,
3      name VARCHAR(100),
4      marks INT,
5      CONSTRAINT prim_key PRIMARY KEY(id)
6  );
7
8  CREATE TABLE IF NOT EXISTS the_report_grades(
9      grade INT,
10     min_mark INT,
11     max_mark INT
12 );
13
14 INSERT INTO the_report_students( name, marks)
15 VALUES
16     ('Julia', 88), ('Samantha', 68), ('Maria', 99),
17     ('Scarlet', 78), ('Ashley', 63), ('Jane', 81);
18
19 INSERT INTO the_report_grades(grade, min_mark, max_mark)
20 VALUES
21     (1, 0, 9), (2, 10, 19), (3, 20, 29), (4, 30, 39), (5, 40, 49),
22     (6, 50, 59), (7, 60, 69), (8, 70, 79), (9, 80, 89), (10, 90, 100);
23
24 Expected Output: Maria 10 99, Jane 9 81, Julia 9 88,
25                  Scarlet 8 78, NULL 7 63, NULL 7 68
26

```

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.

```

1      SELECT
2          CASE
3              WHEN g.grade < 8 THEN NULL
4              ELSE s.name
5          END, g.grade, s.marks FROM students AS s -- Table Alias 's'
6      INNER JOIN grades AS g -- Table Alias 'g'
7          ON s.marks >= g.min_mark AND s.marks <= g.max_mark
8      ORDER BY g.grade DESC, s.name ASC, s.marks ASC;
9

```

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:*

```

1      CREATE TABLE IF NOT EXISTS cl_hackers(
2          hacker_id INT,
3          name VARCHAR(100),
4          CONSTRAINT prim_key PRIMARY KEY(hacker_id)
5      );
6
7      CREATE TABLE IF NOT EXISTS cl_submissions(
8          submission_id INT,
9          hacker_id INT,
10         challenge_id INT,
11         score INT,
12         CONSTRAINT f_key FOREIGN KEY(hacker_id) REFERENCES cl_hackers(hacker_id)
13     );
14
15     INSERT INTO cl_hackers(hacker_id, name)
16     VALUES
17     (4071, 'Rose'), (4806, 'Angela'), (26071, 'Frank'), (49438, 'Patrick'),
18     (74842, 'Lisa'), (80305, 'Kimberly'), (84072, 'Bonnie'), (87868, 'Michael'),
19     (92118, 'Todd'), (95895, 'Joe');
20
21     INSERT INTO cl_submissions(submission_id, hacker_id, challenge_id, score)
22     VALUES
23     (67194, 74842, 63132, 76), (64479, 74842, 19797, 98), (40742, 26071, 49593, 20),
24     (17513, 4806, 49593, 32), (69846, 80305, 19797, 19), (41002, 26071, 89343, 36),
25     (52826, 49438, 49593, 9), (31093, 26071, 19797, 2), (81614, 84072, 49593, 100),
26     (44829, 26071, 89343, 7), (75147, 80305, 49593, 48), (14115, 4806, 49593, 76),
27     (6943, 4071, 19797, 95), (12855, 4806, 25917, 13), (73343, 80305, 49593, 42),
28     (84264, 84072, 63132, 0), (9951, 4071, 49593, 43), (45104, 49438, 25917, 34),
29     (53795, 74842, 19797, 5), (26363, 26071, 19797, 29), (10063, 4071, 49593, 96);
30

```



```
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: Sub-Table 1

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 hackers AS h
INNER JOIN submissions AS s ON h.hacker_id = s.hacker_id
```

```

5      GROUP BY h.hacker_id, h.name, s.challenge_id) AS sql
6  GROUP BY hacker_id, name
7  HAVING sum_score > 0
8  ORDER BY sum_score DESC, hacker_id ASC;
9

```

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:*

```

1  CREATE TABLE IF NOT EXISTS tc_hackers(
2      hacker_id INT,
3      name VARCHAR(100),
4      CONSTRAINT prim_key PRIMARY KEY(hacker_id)
5  );
6
7  CREATE TABLE IF NOT EXISTS tc_challenges(
8      challenge_id INT,
9      hacker_id INT, -- This indicates the creator of the challenge
10     difficulty_level INT, -- This indicates the difficulty level of the challenge
11     CONSTRAINT prim_key PRIMARY KEY challenge_id
12 );
13
14 CREATE TABLE IF NOT EXISTS tc_difficulty(
15     difficulty_level INT,
16     score INT -- This indicates the associated full score for each challenge
17 );
18
19 CREATE TABLE IF NOT EXISTS tc_submissions(
20     submission_id INT,
21     hacker_id INT,
22     challenge_id INT,
23     score INT -- This column indicates the actual score of participants
24 );
25
26 INSERT INTO tc_hackers(hacker_id, name)
27 VALUES
28     (5580, 'Rose'), (8439, 'Angela'), (27205, 'Frank'), (52243, 'Patrick'),
29     (52348, 'Lisa'), (57645, 'Kimberly'), (77726, 'Bonnie'), (83082, 'Michael'),
30     (86870, 'Todd'), (90411, 'Joe');
31
32 INSERT INTO tc_challenges(challenge_id, hacker_id, difficulty_level)
33 VALUES
34     (4810, 77726, 4), (21089, 27205, 1), (36566, 5580, 7),

```

Table 3: Sub-Table I

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

```

35      (66730,52243,6), (71055,52243,2);
36
37      INSERT INTO tc_difficulty(difficulty_level, score)
38      VALUES
39      (1,20), (2,30), (3,40), (4,60), (5,80), (6,100), (7,120);
40
41      INSERT INTO tc_submissions(submission_id, hacker_id, challenge_id, score)
42      VALUES
43      (68628,77726,36566,30), (65300,77726,21089,10), (40326,52243,36566,77),
44      (8941,27205,4810,4), (83554,77726,66730,30), (43353,52243,66730,0),
45      (55385,52348,71055,20), (39784,27205,71055,23), (94613,86870,71055,30),
46      (45788,52348,36566,0), (93058,86870,36566,30), (7344,8439,66730,92),
47      (2721,8439,4810,36), (523,5580,71055,4), (49105,52348,66730,0),
48      (55877,57645,66730,80), (38355,27205,66730,35), (3924,8439,36566,80),
49      (97397,90411,66730,100), (84162,83082,4810,40), (97431,90411,71055,30);
50
51      Expected Output: 90411 Joe
52

```

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.

```

1      SELECT s.hacker_id, s.challenge_id, s.score AS var_actual,
2             c.difficulty_level AS var_level
3      FROM submissions AS s
4           INNER JOIN challenges AS c USING(challenge_id);
5

```

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, the `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 outputs 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 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.

```

1  SELECT s.hacker_id, COUNT(*)
2      FROM submissions AS s
3      INNER JOIN challenges AS c USING(challenge_id)
4      INNER JOIN difficulty_level AS d ON d.difficulty_level = c.difficulty_level
5  WHERE d.score = s.score
6  GROUP BY s.hacker_id;
7

```

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 need to apply 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.

```

1  SELECT h.hacker_id, h.name FROM submissions AS s
2      INNER JOIN challenges AS c ON s.challenge_id = c.challenge_id
3      INNER JOIN difficulty AS d ON c.difficulty_level = d.difficulty_level
4      INNER JOIN hackers AS h ON s.hacker_id = h.hacker_id
5  WHERE d.score = s.score
6  GROUP BY h.hacker_id, h.name
7  HAVING COUNT(*) > 1
8  ORDER BY COUNT(*) DESC, h.hacker_id ASC;
9

```

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.

```

1  CREATE TABLE IF NOT EXISTS c_hackers(
2      hacker_id INT,
3      name VARCHAR(100),

```

```

4      CONSTRAINT prim_key PRIMARY KEY(hacker_id)
5  );
6
7  CREATE TABLE IF NOT EXISTS c_challenges(
8      challenge_id INT,
9      hacker_id INT,
10     CONSTRAINT prim_key PRIMARY KEY(challenge_id),
11     CONSTRAINT f_key1 FOREIGN KEY(hacker_id) REFERENCES c_hackers(hacker_id)
12 );
13
14 INSERT INTO c_hackers(hacker_id, name)
15 VALUES
16 (12299, 'Rose'), (34856, 'Angela'), (79345, 'Frank'),
17 (80491, 'Patrick'), (81041, 'Lisa'), (12234, 'Andrew');
18
19 INSERT INTO c_challenges(challenge_id, hacker_id)
20 VALUES
21 (63963, 81041), (63117, 79345), (28225, 34856), (21989, 12299),
22 (4653, 12299), (70070, 79345), (36905, 34856), (61136, 80491),
23 (17234, 12299), (80308, 79345), (40510, 34856), (79820, 80491),
24 (22720, 12299), (21394, 12299), (36261, 34856), (15334, 12299),
25 (71435, 79345), (23157, 34856), (54102, 34856), (69065, 80491),
26 (22245, 12234), (33225, 12234), (98788, 12234) ;
27
28 Expected Output: 12299 Rose 6, 34856 Angela 6, 79345 Frank 4, 81041 Lisa 1
29

```

We create a new table by integrating the rows in the two tables using an `INNER JOIN` clause on the condition that `hackers.hacker_id = challenges.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.

```

1  SELECT h.hacker_id, h.name, COUNT(*) AS c_1 FROM c_hackers AS h
2      INNER JOIN c_challenges AS c USING(hacker_id)
3      GROUP BY h.hacker_id, h.name;
4

```

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.

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

Table 4: Sub-Table I

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

sequently, 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`.

```

1  -- Table with variable c_1
2
3  CREATE TABLE IF NOT EXISTS temp
4  AS (SELECT COUNT(*) c_1 FROM c_challenges
5      GROUP BY hacker_id);
6
7  -- Query to return non-duplicate entries in c_1
8
9  SELECT c_1 FROM temp
10 GROUP BY c_1
11 HAVING COUNT(c_1) = 1;
12

```

These queries return a list of entries that are non-duplicate in column `c_1`. Thus, we use the conditional statement `c_1 IN (···)` to return the associated rows. However, we have to return the rows associated with hackers who have created the most challenges, 12299 Rose 6, 34856 Angela 6 in Table 4. Thus, another conditional statement as `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.

```

1  CREATE TABLE IF NOT EXISTS temp
2  AS (SELECT COUNT(*) AS c_1 FROM challenges
3      GROUP BY hacker_id);
4
5  SELECT h.hacker_id, h.name, COUNT(*) c_1 FROM hackers AS h
6  INNER JOIN challenges AS c ON h.hacker_id = c.hacker_id
7  GROUP BY h.hacker_id, h.name
8  HAVING c_1 IN
9      (SELECT c_1 FROM temp
10       GROUP BY c_1
11       HAVING COUNT(*) = 1)
12  OR c_1 = (SELECT MAX(c_1) FROM temp)

```

```

13 ORDER BY c_1 DESC, h.hacker_id;
14

```

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:*

```

1 CREATE TABLE IF NOT EXISTS o_wands_property(
2     code INT,
3     age INT,
4     is_evil INT,
5     CONSTRAINT prim_key PRIMARY KEY(code),
6     CONSTRAINT uniq_key UNIQUE(age),
7     CONSTRAINT bin_key CHECK(is_evil IN (0, 1))
8 );
9
10 CREATE TABLE IF NOT EXISTS o_wands(
11     id INT AUTO_INCREMENT,
12     code INT,
13     coins_needed INT,
14     power INT,
15     CONSTRAINT prim_key PRIMARY KEY(id),
16     CONSTRAINT f_key FOREIGN KEY(code) REFERENCES o_wands_property(code)
17 );
18
19 INSERT INTO o_wands_property(code, age, is_evil)
20 VALUES
21 (1, 45, 0), (2, 40, 0), (3, 4, 1), (4, 20, 0), (5, 17, 0);
22
23 INSERT INTO o_wands(code, coins_needed, power)
24 VALUES
25 (4, 3688, 8), (3, 9365, 3), (3, 7187, 10), (3, 734, 8), (1, 6020, 2),
26 (2, 6773, 7), (3, 9873, 9), (3, 7721, 7), (1, 1647, 10), (4, 504, 5),
27 (2, 7587, 5), (5, 9897, 10), (3, 4651, 8), (2, 5408, 1), (2, 6018, 7),
28 (4, 7710, 5), (2, 8798, 7), (2, 3312, 3), (4, 7651, 6), (5, 5689, 3);
29
30 Expected Output: 9 45 1647 10, 12 17 9897 10, 1 20 3688 8, 15 40 6018 7,
31                  19 20 7651 6, 11 40 7587 5, 10 20 504 5, 18 40 3312 3
32                  20 17 5689 3, 5 45 6020 2, 14 40 5408 1
33

```

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 one that has the least `coins_needed`. To this end, we can

Table 5: Sub-Table I

content...

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

```

1  SELECT tab_w.id, tab_w.code, tab_wp.age, tab_w.power, tab_w.coins_needed
2  FROM o_wands AS tab_w
3       INNER JOIN o_wands_property AS tab_wp USING(code)
4       WHERE is_evil = 0
5       ORDER BY code, age, power, coins_needed;
6

```

The output from the query is as follows.

```

1  SELECT id, age, coins_needed, power FROM
2      (SELECT tab_w.id, tab_wp.age, tab_w.coins_needed, tab_w.power,
3       ROW_NUMBER() OVER(PARTITION BY tab_w.code, tab_wp.age, tab_w.power ORDER BY
4       tab_w.coins_needed ASC) AS rn
5       FROM wands AS tab_w
6       INNER JOIN wands_property AS tab_wp ON tab_w.code = tab_wp.code
7       WHERE is_evil = 0) AS sql
8  WHERE rn = 1
9  ORDER BY power DESC, age DESC;

```

Listing 30: text

## 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:

```

1  CREATE TABLE IF NOT EXISTS p_students(
2      id INT AUTO_INCREMENT,
3      name VARCHAR(100),
4      CONSTRAINT prim_key PRIMARY KEY(id)
5  );
6
7  CREATE TABLE IF NOT EXISTS p_packages(
8      id INT,
9      salary DECIMAL(6,2),
10     CONSTRAINT p_f_key1 FOREIGN KEY(id) REFERENCES p_students(id),
11     CONSTRAINT p_uniq_key1 UNIQUE(salary)
12 );
13

```



```

14 CREATE TABLE IF NOT EXISTS p_friends(
15     id INT,
16     friend_id INT,
17     CONSTRAINT p_f_key2 FOREIGN KEY(id) REFERENCES p_students(id),
18     CONSTRAINT p_f_key3 FOREIGN KEY(id) REFERENCES p_students(id)
19 );
20
21 INSERT INTO p_students(name)
22 VALUES
23     ('Ashley'), ('Samantha'), ('Julia'), ('Scarlet');
24
25 INSERT INTO p_packages(id, salary)
26 VALUES
27     (1, 15.20), (2, 10.06), (3, 11.55), (4, 12.12)
28
29 INSERT INTO p_friends(id, friend_id)
30 VALUES
31     (1, 2), (2, 3), (3, 4), (4,1);
32
33 Expected Output: Samantha, Julia, Scarlet
34

```

```

1 SELECT tab_s.name AS var_f_sal FROM students AS tab_s
2     INNER JOIN friends AS tab_f USING(id)
3     INNER JOIN packages AS tab_p1 ON tab_f.friend_id = tab_p1.id
4     INNER JOIN packages AS tab_p2 ON tab_s.id = tab_p2.id
5 WHERE tab_p2.salary < tab_p1.salary
6 ORDER BY tab_p1.salary;
7

```

## 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:

```

1 CREATE TABLE IF NOT EXISTS spp_projects(
2     id INT AUTO_INCREMENT,
3     start_date DATE,
4     end_date DATE,
5     CONSTRAINT prim_key PRIMARY KEY(id)
6 );
7

```

```

8      INSERT INTO spp_projects(start_date, end_date)
9      VALUES
10      ('2015-10-01', '2015-10-02'), ('2015-10-02', '2015-10-03'),
11      ('2015-10-03', '2015-10-04'), ('2015-10-13', '2015-10-14'),
12      ('2015-10-14', '2015-10-15'), ('2015-10-28', '2015-10-29'),
13      ('2015-10-30', '2015-10-31');
14
15      Expected Output: 2015-10-28 2015-10-29, 2015-10-30 2015-10-31,
16                      2015-10-13 2015-10-15, 2015-10-01 2015-10-04
17

```

```

1      SELECT sq1.start_date, MIN(sq2.end_date) AS var_end FROM
2      (SELECT start_date FROM projects
3      WHERE start_date NOT IN (SELECT end_date FROM projects)) sq1
4      INNER JOIN
5      (SELECT end_date FROM projects
6      WHERE end_date NOT IN (SELECT start_date FROM projects)) sq2
7      ON sq2.end_date > sq1.start_date
8      GROUP BY sq1.start_date
9      ORDER BY DATEDIFF(MIN(sq2.end_date), sq1.start_date), sq1.start_date ;
10

```

### 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.

```
1 SELECT COUNT(*) FROM city
2 WHERE population > 100000;
3
```

Listing 31: 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.

```
1 SELECT SUM(population) FROM city
2 WHERE district = 'California';
3
```

Listing 32: 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.

```

1 SELECT AVG(population) FROM city
2 WHERE district = 'California';
3

```

Listing 33: 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.

```

1 SELECT MAX(population) - MIN(population) FROM city;
2

```

### 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:*

```

1 CREATE TABLE IF NOT EXISTS tb_employees(
2     id INT AUTO_INCREMENT,
3     name VARCHAR(100),
4     salary INT,
5     CONSTRAINT prim_key PRIMARY KEY(id),
6     CONSTRAINT range_key CHECK(salary > 0 AND salary < POWER(10,5))
7 );
8
9 INSERT INTO tb_employees(name, salary)
10 VALUES
11     ('Kristeen', 1420), ('Ashley', 2006), ('Julia', 2210), ('Maria', 3000);
12

```

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 computing 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

```
1 SELECT CEIL(AVG(salary) - AVG(REPLACE(salary, 0, ''))) FROM employees;
2
```

### 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.

```
1 SELECT salary*months, COUNT(employee_id) FROM employee
2 GROUP BY salary*months
3 ORDER BY salary*months DESC
4 LIMIT 1;
5
```

Listing 34: 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 outputs to 2 decimal places. Specifically, the function `ROUND(arg1, n)` rounds the numerical argument, `arg1` to `n` decimal places.

```

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

```

Listing 35: 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.

```

1 SELECT ROUND(SUM(lat_n), 4) FROM station
2 WHERE lat_n > 38.7880 AND lat_n < 137.2345;
3

```

Listing 36: 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 that 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.

```

1 SELECT ROUND(MAX(lat_n), 4) FROM station
2 WHERE lat_n < 137.2345;
3

```

Listing 37: 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.

```
1 SELECT ROUND(long_w, 4) FROM station
2 WHERE lat_n < 137.2345
3 ORDER BY lat_n DESC
4 LIMIT 1;
5
```

Listing 38: 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.

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

Listing 39: 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.

```
1  SELECT
2      ROUND(POWER(POWER(MIN(lat_n) - MAX(lat_n), 2) + POWER(MIN(long_w) - MAX(long_w), 2)
3          ,1/2), 4)
4  FROM station;
```

Listing 40: 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.

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



```

2
3      SELECT ROUND(AVG(lat_n),4) FROM
4          (SELECT lat_n,
5             ROW_NUMBER() OVER(ORDER BY lat_n ASC) AS var_rn FROM station) sq1
6 WHERE
7     CASE MOD(@T,2)
8         WHEN 1 THEN var_rn = CEIL(@T/2)
9         ELSE var_rn IN (@T/2, @T/2 + 1)
10    END;
11
12 SET @T := NULL;
13

```

Listing 41: Weather Observation Station 20