Table of Contents

SQL Principles

1. **Master the Order of SQL Clauses and Their Execution Sequence**
   - **Writing Order:** When writing an SQL statement, follow this standard sequence of clauses:
   > **SELECT**
   > **FROM**
   > **JOIN** (if applicable)
   > **WHERE**
   > **GROUP BY**
   > **HAVING**
   > **WINDOW** (for window functions, if applicable)
   > **ORDER BY**
   > **LIMIT** (or equivalent like TOP or FETCH FIRST)

   - **Execution Hierarchy:** Understand the logical order in which these clauses are processed:
   > **FROM** and **JOIN** clauses to determine the total working set of data.
   > **WHERE** clause to filter rows.
   > **GROUP BY** to arrange the data into groups.
   > **HAVING** clause to filter groups.
   > **SELECT** clause to choose/select columns to be displayed.
   > **WINDOW** functions are applied.
   > **ORDER BY** to sort the result set.
   > **LIMIT** to restrict the number of rows returned.
2. **Understand Database Schema Design**
   - Grasp the fundamentals of database normalization for reducing redundancy and ensuring data integrity.
   - Apply denormalization techniques where necessary for performance in analytical queries.
3. **Master Data Retrieval Techniques**
   - Develop proficiency in writing **SELECT** statements, specifying columns and using different types of filters (**WHERE** clauses).
   - Understand and effectively use aggregate functions (**COUNT**, **SUM**, **AVG**, **MIN**, **MAX**) and grouping (**GROUP BY**) to summarize data.
   - Utilize **JOIN** operations (INNER, LEFT, RIGHT, FULL OUTER) to retrieve data from multiple tables and understand their use cases.
   - Employ subqueries and common table expressions (CTEs) for complex data retrieval and organization.
4. **Advanced Querying Techniques**
   - Learn to use window functions (**ROW_NUMBER()**, **RANK()**, **SUM() OVER()**, etc.) for sophisticated data analysis tasks like calculating running totals or moving averages.
   - Apply set operations like **UNION**, **UNION ALL**, **INTERSECT**, and **EXCEPT** (or **MINUS**) to combine or compare datasets.
5. **Proficiency in Date-Time Functions**
   - Gain a solid understanding of the date-time functions available in SQL, such as **GETDATE()**, **DATEADD()**, **DATEDIFF()**, **DATE_FORMAT()**, and others, depending on your SQL dialect.
   - Learn to perform common date-time operations like extracting specific components (day, month, year), manipulating dates (adding or subtracting intervals), and formatting dates for display.
6. **Mastering String Functions**
   - Familiarize yourself with essential string functions like **CONCAT()**, **SUBSTRING()**, **CHAR_LENGTH()**, **UPPER()**, **LOWER()**, and others.
   - Understand how to use these functions for data cleaning, manipulation, and preparation, which is especially important in data analysis and reporting.
7. **Handling of Time Zones**
   - Be aware of time zone considerations when working with date-time data, especially in applications spanning multiple time zones.
   - Understand how your database manages time zones and how to convert date-time values accordingly.
8. **Pattern Matching and Regular Expressions**

- Learn to use **LIKE**, **SIMILAR TO**, or regular expression functions for pattern matching in strings.
- These are invaluable for filtering data based on specific text patterns.

9. Efficient Query Optimization

- Optimize SQL queries for performance by avoiding unnecessary columns in **SELECT** statements and using **JOIN** clauses and indexes efficiently.
- Regularly review and tune SQL queries and database designs, using tools like EXPLAIN plans to understand and optimize query execution.

10. Data Manipulation and Integrity

- Be adept with data manipulation statements (**INSERT**, **UPDATE**, **DELETE**) and understand their impact on database integrity.
- Implement and enforce data integrity through the use of primary keys, foreign keys, and constraints (**UNIQUE**, **NOT NULL**, **CHECK**).

11. Transaction Control and Security

- Use transaction control statements (**BEGIN**, **COMMIT**, **ROLLBACK**) to ensure data consistency and integrity.
- Understand and mitigate SQL injection risks; ensure database security through parameterized queries and proper access controls.

12. Reporting and Data Analysis

- Develop the ability to create effective reports and export data, integrating SQL with business intelligence tools and reporting platforms.
- Translate business requirements into SQL queries, demonstrating strong data modelling and analysis skills.

13. SQL Variants and Compatibility

- Recognize the nuances of different SQL dialects (MySQL, PostgreSQL, SQL Server, etc.) and their unique features and functions.

14. Best Practices in SQL Coding

- Write clear, readable SQL code with proper formatting and comments for better maintainability.
- Use aliases and organize SQL scripts logically, enhancing the readability and understandability of the code.

15. Continuous Learning and Collaboration

- Stay updated with the latest developments in SQL standards and database technologies.
- Engage in continuous learning to adapt to new tools and techniques in data management and analysis.
- Document SQL queries and database structures for effective team collaboration and knowledge sharing.

16. Unit Testing and Validation

- Implement unit testing for SQL queries to ensure the accuracy and reliability of data retrieval and manipulation.
- Validate data and query outputs regularly to maintain high data quality standards.

The above principles will develop a strong foundation in SQL, leading to more efficient data handling, insightful analysis, and informed business decisions. Each principle builds upon the previous one, creating a comprehensive and structured approach to mastering SQL.

Proper use of Aliases

Proper use of aliases in SQL is an important practice for writing clear and maintainable queries, especially in complex database operations. Here are some relevant pointers regarding the use of aliases:

1. **Use Aliases for Clarity and Readability**
   - Aliases are particularly useful in queries with joins or subqueries where tables or subqueries are referenced multiple times.
   - Assign meaningful alias names that clearly indicate the role or content of the table or column.
2. **Column Aliases for Better Output Formatting**
   - Use column aliases to provide more readable and descriptive column names in the output of your query.
   - This is especially useful when the original column names are cryptic, derived from calculations, or when you want to present the data in a specific format to end users.
3. **Table Aliases to Simplify Query Writing**
   - When working with joins, especially involving multiple tables or subqueries, use table aliases to shorten and simplify your SQL syntax.
   - This reduces the need to repeatedly write the full table name, making the query easier to read and write.
4. **Aliases in Complex Queries**
   - In complex queries involving multiple levels of subqueries, aliases help keep track of each level and make the query more navigable.
   - They are essential when the same table is joined to itself (self-join) for clarity in distinguishing different instances of the table.
5. **Consistency in Aliasing**
   - Be consistent in the use of aliases throughout the query. Once you assign an alias to a table or column, use that alias exclusively for all references to it in that query.
   - This consistency is key to avoiding confusion and potential errors in the query.
6. **Using Aliases in Aggregate and Window Functions**
   - When using aggregate functions (like **SUM**, **COUNT**, etc.) or window functions, aliases provide a way to reference the computed columns easily in the query or in the **ORDER BY** clause.
7. **Avoiding Ambiguity**
   - Use aliases to avoid ambiguity, especially when different tables in a join have columns with the same name.
   - This practice is crucial for ensuring that the SQL engine correctly understands which column is being referenced.
8. **Mandatory Aliases in Certain Scenarios**
   - In some SQL operations, like when using subqueries in the **FROM** clause, assigning an alias is mandatory. Ensure compliance with these requirements to avoid syntax errors.
9. **Formatting and Naming Conventions**
   - Follow a consistent naming convention and format for aliases across your queries and within your team or organization to maintain uniformity.
10. **Aliasing and SQL Variants**
    - Be aware that syntax for aliasing can vary slightly between different SQL variants (like MySQL, PostgreSQL, SQL Server). Ensure you're using the correct syntax for your specific database system.

Incorporating these pointers about aliases into SQL practices will enhance the readability, maintainability, and overall quality of the database queries. They are particularly beneficial in complex queries and in scenarios where presentation and clarity of the output are critical.

Filtering data

1. Using the WHERE Clause:

- **Primary Use:** Filters rows before any grouping or aggregation occurs. It's applied directly to the raw data in the tables.
- **Common Operators:** Includes **=, !=, >, <, >=, <=, BETWEEN, IN, IS NULL, IS NOT NULL, LIKE,** and others.
- **Filtering with Subqueries:** Allows complex conditions using subqueries, which can return a list of values, a single value, or even perform existential checks with **EXISTS.**
- **Note:** Cannot be used to filter aggregated data (like sums or averages).

2. Using the FROM Clause (specifically in Joins with the ON Clause):

- **Role in Joins:** The **ON** clause is part of the syntax for joins (e.g., INNER JOIN, LEFT JOIN) and specifies the conditions for how rows from different tables should be matched.
- **Filtering Aspect:** While primarily used for specifying join conditions, it inherently filters data by determining which rows from the joined tables meet the join condition.
- **Note:** The **ON** clause is not a general-purpose filtering tool like **WHERE** but is specific to defining relationships between tables in a join.

3. Using the HAVING Clause:

- **Primary Use:** Filters groups of rows after the **GROUP BY** operation.
- **Applicability:** Used when you have aggregations (**SUM, COUNT, AVG,** etc.) in your **SELECT** statement and want to apply conditions on these aggregated results.
- **Difference from WHERE: WHERE** filters individual rows before grouping, while **HAVING** filters groups after grouping.
- **Note:** Often used in conjunction with **GROUP BY,** but can be used without it if the query involves aggregate functions.

Additional Points:

- **Order of Execution: WHERE -> GROUP BY -> HAVING -> SELECT.** This is the logical processing order, not necessarily the physical execution order used by the SQL engine.
- **Performance Considerations:** Using **WHERE** to filter as much as possible before aggregation can improve query performance. **HAVING** should be used for conditions that cannot be applied before aggregation.

In summary, each of these clauses (**WHERE, ON** in **FROM,** and **HAVING**) plays a distinct role in filtering data in SQL, and understanding their specific uses and limitations is key to writing efficient and effective SQL queries.

WHERE Versus HAVING – CAUTION AND TAKEAWAYS

The purpose of both clauses is to filter data. If you are trying to:

- Filter on particular columns, write your conditions within the WHERE clause
- Filter on aggregations, write your conditions within the HAVING clause

The contents of a WHERE and HAVING clause cannot be swapped:

- Never put a condition with an aggregation in the WHERE clause. You will get an error.
- Never put a condition in the HAVING clause that does not involve an aggregation. Those conditions are evaluated much more efficiently in the WHERE clause.

Filtering data – Concise Summary with Sample codes

- using **WHERE** clause
  - filtering on columns, using =, BETWEEN, IN, IS NULL, LIKE
  - filtering on subqueries

- using **FROM** clause: When joining together tables, the ON clause specifies how they should be linked together. This is where you can include conditions to restrict rows of data returned by the query.

- using **HAVING** clause : If there are aggregations within the SELECT statement, the HAVING clause is where you specify how the aggregations should be filtered.

Refer :

- SQL Pocket Guide by Alice Zhou.
- "SQL for Data Scientists - A Beginner's Guide for Building Datasets for Analysis" by Renee M. P. Teate (**Chapter 3 - The WHERE Clause** for the sample codes in this section.)

-- Filtering Using predicate on column values within Where clause. A predicate also known as conditional statement is a logical comparison that results in one of three values: TRUE/FALSE/UNKNOWN

```
  SELECT
      market_date,
      customer_id,
      vendor_id,
      quantity * cost_to_customer_per_qty AS price
  FROM farmers_market.customer_purchases
  WHERE
      customer_id = 4
      AND vendor_id = 7;
```

-- Filtering Using BETWEEN in WHERE clause

```
SELECT *
  FROM farmers_market.vendor_booth_assignments
  WHERE
      vendor_id = 7
      AND market_date BETWEEN '2019-03-02' and '2019-03-16'
  ORDER BY market_date;
```

-- Filtering Using LIKE in WHERE clause

```
  SELECT
      customer_id,
      customer_first_name,
      customer_last_name
  FROM farmers_market.customer
  WHERE
      customer_first_name LIKE 'Jer%';
```

-- Filtering Using IN condition in WHERE clause

```
SELECT
      customer_id,
      customer_first_name,
      customer_last_name
  FROM farmers_market.customer
  WHERE
      customer_first_name IN ('Renee', 'Rene', 'Renée', 'René', 'Renne');
```

-- Filtering Using Subquery within Where clause

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity * cost_to_customer_per_qty price
FROM farmers_market.customer_purchases
WHERE
    market_date IN
        (
        SELECT market_date
        FROM farmers_market.market_date_info
        WHERE market_rain_flag = 1
        )

LIMIT 5 ;
```

-- Filtering with Having

```
SELECT
      vendor_id,
      COUNT(DISTINCT product_id) AS different_products_offered,
      SUM(quantity * original_price) AS value_of_inventory,
      SUM(quantity) AS inventory_item_count,
      SUM(quantity * original_price) / SUM(quantity) AS average_item_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
HAVING inventory_item_count >= 100
ORDER BY vendor_id;
```

*Note, the SQL engine is designed to recognize and appropriately handle aliases from the **SELECT** clause in the **HAVING** and **ORDER BY** clauses, despite the logical processing order of SQL queries.

Recursive CTEs – Food for thought

```
********************
******************
****************
**************
**********
*********
********
******
****
**
```

```
-- Print patterns using SQL (HackerRank, Alternative Queries)
-- Recursive CTE code without comments
WITH RECURSIVE num(n) AS (
    SELECT 20

    UNION ALL

    SELECT n - 2
    FROM num
    WHERE n - 2 >= 2
)
SELECT
    load('', num.n, '*')
FROM
    num;
```

```
-- Recursive CTE code with comments
-- Define a recursive CTE named 'num'
WITH RECURSIVE num(n) AS (
    -- Anchor Member: Start the sequence with 20
    SELECT 20

    UNION ALL

    -- Recursive Member: In each subsequent iteration,
    -- decrement the previous number by 2
    SELECT n - 2
    FROM num
    -- Termination Condition: Continue recursion as long as the
    -- number is greater than or equal to 2
    WHERE n - 2 >= 2
)

-- Main query to select from the recursive CTE
SELECT
    -- Assuming 'load' is a predefined function in your SQL environment;
    -- replace it with the actual function intended
    -- The function is used here with each number generated by the CTE
    load('', num.n, '*')
FROM
    num; -- Select from the recursively generated numbers in CTE 'num'
```

Recursive Common Table Expressions (CTEs) are a powerful feature in SQL that allow you to execute complex queries, particularly useful for dealing with hierarchical or recursive data structures. Here's a more detailed look at recursive CTEs:

Basic Structure:

A recursive CTE consists of two parts:

1. **Anchor Member**: This is the initial query that forms the base result set of the CTE. It's the starting point of the recursion.
2. **Recursive Member**: This part of the CTE references the CTE itself and is used to extend or transform the result set from the anchor member.

These two parts are combined using the **UNION ALL** operator, ensuring that the results of the anchor member are combined with the results of the recursive member.

How It Works:

- **Initialization**: The anchor member is executed first to create the initial set of rows.
- **Recursion**: The recursive member is then repeatedly executed, taking the results of the previous iteration as its input, and adding to the overall result set.
- **Termination**: The recursion continues until the recursive member returns no rows or a specified condition is met, preventing an infinite loop.

Use Cases:

- **Generating Sequences**: Like creating a series of numbers, dates, etc.
- **Hierarchical Data**: Navigating tree-like structures, such as organizational charts, category trees, or folder structures.
- **Graph Data**: Traversing graph data structures, such as finding the shortest path or all paths between two nodes.

Example:

A simple example of a recursive CTE is generating a series of numbers:

```
WITH RECURSIVE NumberSeries AS (
    SELECT 1 AS Number -- Anchor Member
    UNION ALL
    SELECT Number + 1 FROM NumberSeries -- Recursive Member
    WHERE Number < 10
)
SELECT * FROM NumberSeries;
```

This query generates a series of numbers from 1 to 10.

Key Points to Remember:

- **Avoid Infinite Loops**: Always ensure there is a condition to terminate the recursion, or else the query can go into an infinite loop.
- **Performance**: Recursive CTEs can be resource-intensive, especially with large data sets or complex recursive logic.
- **Database Support**: Most modern relational databases support recursive CTEs, including PostgreSQL, MySQL (from version 8.0), SQL Server, and Oracle.

Recursive CTEs open up a wide array of possibilities for data manipulation and querying, especially in scenarios where traditional iterative programming techniques are not efficient or possible in standard SQL.

Solved Problems

HackerRank Problem : Challenges | Medium Level
https://www.hackerrank.com/challenges/challenges/problem

Julia asked her students to create some coding challenges. 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.

Input Format

The following tables contain challenge data:

- Hackers: The hacker_id is the id of the hacker, and name is the name of the hacker.

Column	Type
hacker_id	Integer
name	String

- Challenges: The challenge_id is the id of the challenge, and hacker_id is the id of the student who created the challenge.

Column	Type
challenge_id	Integer
hacker_id	Integer

Solution :

Before proceeding with the solution, let us rethink and clarify the objective.

Objective: To select hackers based on the following criteria:

1. Maximum Challenge Solvers: Hackers who have solved the highest number of challenges, regardless of whether there are ties for this top spot.
2. Unique Challenge Solvers: Hackers who have solved a number of challenges that is unique compared to other hackers. This means selecting hackers who are the only ones to have solved that specific number of challenges.

Example Scenario for Clarity:

- Suppose hackers A and B have each solved the maximum of 7 challenges.
- Hackers D and E are tied, with each solving 5 challenges.
- Hacker F has solved 3 challenges, and no other hacker has solved exactly 3 challenges.

Expected Outcome:

- The query should return Hackers A and B (for solving the maximum number of challenges) and Hacker F (for solving a unique number of challenges).
- Hackers D and E should be excluded, despite being tied in their challenge count, because their challenge count is not the maximum and is not unique.

In summary, the query aims to identify hackers who stand out either by being at the very top in terms of challenge count or by having a distinct challenge count not shared with any other hacker.

Approach

To achieve the objective of selecting hackers who have either solved the maximum number of challenges or have a unique number of challenges, we can simplify the query by breaking it down into more intuitive steps.

1. **Calculate the number of challenges solved by each hacker.**
2. **Determine the maximum number of challenges solved by any hacker.**
3. **Find the counts of challenges that are unique to individual hackers.**
4. **Select hackers who have either solved the maximum number of challenges or have a unique challenge count.**

In the following SQL query:

- **RankedHackers CTE** calculates the number of challenges solved by each hacker.
- **MaxChallenges CTE** finds the maximum number of challenges solved by any hacker.
- **UniqueChallenges CTE** finds the challenge counts that are unique to individual hackers.
- The final **SELECT** statement uses **LEFT JOIN** to bring in information about whether a hacker's challenge count is the maximum or unique, and filters out those who don't meet either criterion.

SQL Code using CTE (common table expressions)

```
-- Calculate the number of challenges solved by each hacker
WITH ChallengeCounts AS (
    SELECT
        h.hacker_id,
        h.name,
        COUNT(DISTINCT c.challenge_id) AS challengesCount
    FROM
        Challenges c
    JOIN
        Hackers h ON c.hacker_id = h.hacker_id
    GROUP BY
        h.hacker_id, h.name
),
-- Determine the maximum number of challenges solved by any hacker
MaxChallengeCount AS (
    SELECT
        MAX(challengesCount) AS maxChallenges
    FROM
        ChallengeCounts
),
-- Find the counts of challenges that are unique to individual hackers
UniqueChallengeCounts AS (
    SELECT
        challengesCount
    FROM
        ChallengeCounts
    GROUP BY
        challengesCount
    HAVING
        COUNT(*) = 1
)
-- Select hackers who have either solved the maximum number of challenges
-- or have a unique challenge count
SELECT
    cc.hacker_id,
    cc.name,
    cc.challengesCount
FROM
    ChallengeCounts cc
WHERE
    -- Check if the hacker's challenge count is the maximum
    cc.challengesCount IN (SELECT maxChallenges FROM MaxChallengeCount)
    -- OR check if the hacker's challenge count is unique
    OR cc.challengesCount IN (SELECT challengesCount FROM
UniqueChallengeCounts)
ORDER BY
    cc.challengesCount DESC, cc.hacker_id;
```

SQL Code using Subquery approach

As Common Table Expressions (CTEs) is not compatible with older versions of MySQL, we will explore alternative approach of using subqueries directly within the **FROM** and **WHERE** clauses.

Approach

- We first join the **Challenges** and **Hackers** tables and group the results by hacker to calculate each hacker's challenge count.
- The **HAVING** clause filters these results based on two conditions:
    - The hacker's challenge count matches the maximum number of challenges solved, determined by a subquery (**SubMax**) that calculates the maximum challenge count across all hackers.
    - The hacker's challenge count is unique, as determined by another subquery (**SubUnique**). This subquery calculates challenge counts for all hackers and then groups them to find counts that only appear once.
- Finally, the results are ordered by the challenge count in descending order and then by the hacker ID.

```
SELECT
    h.hacker_id,
    h.name,
    COUNT(DISTINCT c.challenge_id) AS challengesCount
FROM
    Challenges c
JOIN
    Hackers h ON c.hacker_id = h.hacker_id
GROUP BY
    h.hacker_id, h.name
HAVING
    -- Check if the hacker's challenge count is the maximum
    COUNT(DISTINCT c.challenge_id) = (
        SELECT MAX(ChallengeCount)
        FROM (
            SELECT COUNT(DISTINCT challenge_id) AS ChallengeCount
            FROM Challenges
            GROUP BY hacker_id
        ) AS SubMax
    )
    -- OR check if the hacker's challenge count is unique
    OR COUNT(DISTINCT c.challenge_id) IN (
        SELECT ChallengeCount
        FROM (
            SELECT COUNT(DISTINCT challenge_id) AS ChallengeCount
            FROM Challenges
            GROUP BY hacker_id
        ) AS SubUnique
        GROUP BY ChallengeCount
        HAVING COUNT(*) = 1
    )
ORDER BY
    COUNT(DISTINCT c.challenge_id) DESC, h.hacker_id;
```

HackerRank Problem : Contest Leaderboard | Medium Level

https://www.hackerrank.com/challenges/contest-leaderboard/problem

You did such a great job helping Julia with her last coding contest challenge that she wants you to work on this one, too!

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 from your result.

Input Format

The following tables contain contest data:

- **Hackers:** The **hacker_id** is the id of the hacker, and **name** is the name of the hacker.

Column	Type
hacker_id	Integer
name	String

- **Submissions:** The **submission_id** is the id of the submission, **hacker_id** is the id of the hacker who made the submission, **challenge_id** is the id of the challenge for which the submission belongs to, and **score** is the score of the submission.

Column	Type
submission_id	Integer
hacker_id	Integer
challenge_id	Integer
score	Integer

SQL Code using CTE approach

```
-- CTE to calculate the maximum score for each challenge per hacker
WITH t AS (
    SELECT
        h.hacker_id,             -- Hacker's ID
        h.name,                  -- Hacker's Name
        c.challenge_id,          -- ID of the challenge
        MAX(c.score) AS score    -- Maximum score per challenge for each hacker
    FROM
        Hackers h                -- Hackers table
        JOIN Submissions c ON h.hacker_id = c.hacker_id
        -- Joining with Submissions table
    GROUP BY
        h.hacker_id, h.name, c.challenge_id
        -- Grouping by hacker_id, name, and challenge_id
)

-- Main query to sum up the maximum scores of each hacker
SELECT
    t.hacker_id,                -- Hacker's ID
    t.name,                     -- Hacker's Name
    SUM(t.score) AS totalscore  -- Total score for each hacker
FROM
    t                           -- Using the CTE defined above
GROUP BY
    t.hacker_id,                -- Grouping by hacker_id
    t.name
    -- and name to sum scores for each unique hacker
HAVING
    SUM(t.score) > 0
    -- Filtering out hackers with zero or negative total scores
ORDER BY
    totalscore DESC,            -- Ordering by total score in descending order
    t.hacker_id ASC;            -- and then by hacker_id in ascending order for ties
```

This query is doing the following:

1. The Common Table Expression (CTE) named **t** is computing the maximum score obtained by each hacker in each challenge. It's important to note that this CTE groups by **hacker_id**, **name**, and **challenge_id**, which means it calculates the maximum score for each hacker for each challenge they participated in.

2. The main query then sums these maximum scores for each hacker across all challenges. This sum represents each hacker's total score.

3. The **HAVING** clause filters out any hackers whose total score is 0 or less, as we're only interested in those who have a positive total score.

4. The result is then ordered by **totalscore** in descending order, so the hacker with the highest total score appears first. In case of ties in the total score, the **hacker_id** is used to order the results in ascending order.

SQL Code using Subquery approach

```
SELECT
    sub.hacker_id,              -- Selecting hacker's ID
    sub.name,                   -- Selecting hacker's name
    SUM(sub.score) AS totalscore
    -- Summing up the scores to get total score
FROM (
    -- Subquery to get maximum score per challenge for each hacker
    SELECT
        h.hacker_id,            -- Hacker's ID
        h.name,                 -- Hacker's name
        c.challenge_id,         -- Challenge ID
        MAX(c.score) AS score
        -- Maximum score for each challenge per hacker
    FROM
        Hackers h               -- From Hackers table
        JOIN Submissions c ON h.hacker_id = c.hacker_id
        -- Joining with Submissions table
    GROUP BY
        h.hacker_id, h.name, c.challenge_id
        -- Grouping by hacker_id, name, and challenge_id
) AS sub                        -- Alias for the subquery
GROUP BY
    sub.hacker_id,               -- Grouping results by hacker_id
    sub.name                     -- and name in the outer query
HAVING
    SUM(sub.score) > 0
    -- Filtering to include only hackers with a positive total score
ORDER BY
    totalscore DESC,             -- Ordering by total score in descending order
    sub.hacker_id ASC;           -- and then by hacker_id in ascending order for ties
```

In the above query:

- The inner subquery (aliased as **sub**) calculates the maximum score for each challenge that each hacker participated in. This is done by grouping the data by **hacker_id**, **name**, and **challenge_id**.
- The outer query then sums these maximum scores to find the total score for each hacker.
- The **HAVING** clause filters out hackers with a total score of 0 or less, focusing on those who have positive scores.
- Finally, the results are ordered by the total score in descending order. If there are ties in the total score, the **hacker_id** is used to order these ties in ascending order.

The query effectively captures the total maximum scores of hackers across different challenges, ensuring that each hacker's contribution in various challenges is considered for their total score.

MySQL from the Command Line interface

Accessing MySQL from terminal in MacOS

Command Objective	Commands
Accessing MySQL from Terminal	
Start MySQL in terminal if no instance running	sudo /usr/local/mysql/support-files/mysql.server start
If MySQL already running, access the MySQL database management system. Password will be prompted as a result	/usr/local/mysql/bin/mysql -u root -p
Enter Password	MySQL@2023
Check if MySQL is Running: First, check if the MySQL server is actually running. You can use the following command to see if MySQL processes are active:	ps aux | grep mysql
Exit mysql from Command Line	exit
Stop the Safe Mode MySQL Server: Go to the terminal window where you started MySQL in safe mode. You can stop it by pressing Control + C to terminate the process. If that doesn't work, you can find the process ID (PID) with ps and then use kill to stop it:	ps aux | grep mysqld_safe
Look for the PID in the output, which is a number in the second column, and then: Replace **[PID]** with the actual process ID of the mysqld_safe command.	sudo kill -SIGTERM [PID]
After running the kill command, you should verify that the process has stopped by running ps aux | grep mysql again. If the process is still running (sometimes it can take a little time to shut down), or if it does not shut down gracefully with SIGTERM, you might need to use SIGKILL, which is a more immediate and forceful stop command:	sudo kill -SIGKILL 3710

Commonly used MySQL commands in terminal

SHOW DATABASES;	This command lists all databases on the MySQL server.
USE database_name;	Selects a particular database to work with.
SHOW TABLES;	Once you've selected a database, this command shows all tables in that database.
DESCRIBE table_name; or DESC table_name;	Provides the structure of a specific table, including column names, data types, and whether they can be NULL.
SELECT * FROM table_name;	Retrieves all data from a table.
SELECT column1, column2 FROM table_name;	Retrieves specific columns from a table.
SELECT * FROM table_name WHERE condition;	Selects data from a table that meets certain conditions.
INSERT INTO table_name (column1, column2) VALUES (value1, value2);	Inserts new data into a table.
UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;	Updates existing data in a table.
DELETE FROM table_name WHERE condition;	Deletes data from a table based on a condition.
CREATE DATABASE database_name;	Creates a new database.
DROP DATABASE database_name;	Deletes a database.
CREATE TABLE table_name (column1 datatype, column2 datatype, ...);	Creates a new table in the database.
DROP TABLE table_name;	Deletes a table from the database.
ALTER TABLE table_name ADD column_name datatype;	Adds a new column to an existing table.
ALTER TABLE table_name DROP COLUMN column_name;	Deletes a column from a table.
GRANT PRIVILEGES ON database.table TO 'username'@'host';	Gives a user specific privileges on a database or table.
REVOKE PRIVILEGES ON database.table FROM 'username'@'host';	Removes specific privileges from a user on a database or table.
EXIT; or QUIT;	Exits the SQL command line interface.

Resources to follow

Highly Recommended

1. SQL Pocket Guide by Alice Zhao **<Simply get a hardcopy and make it your go-to-book>**
2. "SQL for Data Analysis: Advanced Techniques for Transforming Data into Insights" by Cathy Tanimura | **<Load all the databases chapter wise from github repository and read/practice queries along with the text. Chapters 1,2,8 very useful to get SQL overview. Definitely go through Page no. 297 on Understanding Order of SQL Clause Evaluation>**
3. "SQL for Data Scientists - A Beginner's Guide for Building Datasets for Analysis" by Renee M. P. Teate **<Relevant from Data science POV, well written, a breeze to read, treatment is lucid>**
4. Learning SQL: Generate, Manipulate, and Retrieve Data, by Alan Beaulieu | **<Get started with the Sakila Database on MySQL. Great section on MySQL>**
5. Data Analysis Using SQL and Excel by GS Linoff **<Excellent coverage of Excel, SQL, statistics in tandem to tackle business problems>**
6. SQL Window Functions: The Key to Succeeding in Data Science Interviews | https://youtu.be/e-EL-6Vnkbg

Optional

7. Getting Started with SQL: A Hands-On Approach for Beginners by Thomas Nield
8. Getting Started with SQL and Databases: Managing and Manipulating Data with SQL by Mark Simon
9. CS50's Introduction to Databases with SQL|https://cs50.harvard.edu/sql/2023/weeks/0/
10. "SQL Cookbook - Query Solutions and Techniques for All SQL Users" by Anthony Molinaro and Robert de Graaf
11. "Cracking the SQL Interview for DATA SCIENTISTS" by Leon Wei <**90 SQL interview questions and solutions, to** learn or refresh SQL coding skills>
12. "SQL Query Design Patterns and Best Practices - A practical guide to writing readable and maintainable SQL queries using its design patterns" by Steve Hughes Dennis Neer, Dr. Ram Babu Singh, Shabbir H. Mala, Leslie Andrews, Chi Zhang
13. "REAL SQL QUERIES 50 CHALLENGES" by BRIAN COHEN NEIL PEPI NEERJA MISHRA
14. "The SQL Workshop - A New, Interactive Approach to Learning SQL" by Frank Solomon Prashanth Jayaram Awni Al Saqqa
15. Introduction to Database & SQL | One Shot SQL - https://youtu.be/ccgQqpbdN70
16. Practical SQL by Anthony DeBarros
17. SQL: Advanced SQL Query Optimization techniques by Andy Vickler
18. SQL: Build complex SQL Queries by Andy Vickler
19. "Learning MySQL - Get a Handle on Your Data" by Vinicius M. Grippa and Sergey Kuzmichev **<comprehensive overview on how to set up and design an effective database with MySQL>**

Practice

20. https://www.hackerrank.com/domains/sql
21. https://www.hackerrank.com/challenges/challenges/problem
22. https://leetcode.com/problemset/
23. Grokking the SQL Interview by Javin Paul **<Recommended>**
24. Data Carpentry SQL for Ecology | http://lgatto.github.io/sql-ecology/
25. SQL Notes for Professionals by Goalkicker.com
26. MySQL Notes for Professionals by Goalkicker.com

Software Downloads

27. MySQL Workbench community ed.| https://dev.mysql.com/downloads/workbench/