MariaDB Tuorial Session 3

Yang Bai

This tutorial will guide you through the process of loading the provided tables into MariaDB and using advanced SQL commands based on the SQL_2 lecture slides.

Step 1: Load the Tables into MariaDB

1. **Connect to MariaDB**:
    - Open a terminal and run:
        i. `mysql -u <user_name> -p`
    - Enter your password when prompted.

2. **Create and Load the Database**
    - Use the provided SQL file content to create the database and tables.
        i. `SOURCE /path/to/festive.txt;`

    - **Explanation:** This command executes all the SQL commands from the provided text file to create the database `festive` and populate the tables.

Step 2: Advanced SQL Commands with Examples

2.1 Renaming Columns Using Aliases

Query:

```
SELECT name AS bar_name, license AS license_type
FROM Bars;
```

Explanation:

- This query renames the `name` column to `bar_name` and the `license` column to `license_type` in the **returned result set**.
- **Note:** The `SELECT ... AS` command only affects the **temporary result table** shown in the query output.

Important: This does **not** change the actual column names in the original table. To change column names permanently, you need to use the `ALTER` command:

```
ALTER TABLE Bars CHANGE COLUMN name bar_name VARCHAR(25);
```

2.2 Complex WHERE Conditions

Query:

```
SELECT bar, beer, price
FROM Sells
WHERE bar = 'Joes Place' AND beer = 'Bud';
```

Explanation: Returns the price of Bud sold at Joes Place.

2.3 Handling NULL Values

Query:

```
SELECT bar, beer, price
FROM Sells
WHERE price IS NULL;
```

Explanation: Lists bars and beers where the price is unknown.

2.4 Multi-Relation Queries (Joins)

Query:

```
SELECT Likes.drinker AS Drinker, beer AS Likes_Beer, Frequents.bar
as Frequents_Bar
FROM Likes, Frequents
WHERE Frequents.bar = 'Joes Place'
AND Likes.drinker = Frequents.drinker;
```

Explanation: Finds beers liked by those who frequent Joe's Place.

2.5 Self-Joins

Query:

```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
```

```
WHERE b1.brewer = b2.brewer
AND b1.name < b2.name;
```

Explanation: Finds pairs of beers from the same brewer.

2.6 Controlling Duplicates

Query:

```
SELECT DISTINCT price
FROM Sells;
```

Explanation: This SQL query retrieves a **list of distinct (unique) prices** from the `Sells` table, ensuring that duplicate prices are excluded from the result.

2.7 Subqueries

Query:

```
SELECT DISTINCT bar, Beer, brewer, price
FROM Sells, Beers
WHERE brewer = 'Miller'
AND price = (SELECT price FROM Sells WHERE bar = 'Joes Place' AND
beer = 'MGD');
```

Explanation: Finds bars that sell beers brewed by Miller at the same price of MGD sold by Joes Place.

2.8 Using IN and EXISTS Operators

IN Example:

```
SELECT name AS beer, brewer, calories
FROM Beers
WHERE name IN (SELECT beer FROM Likes WHERE drinker = 'Corrie');
```

Explanation: This SQL query retrieves detailed information of beers from the `Beers` table, using the **`IN` operator** to filter only those beers that appear in the list of beers liked by 'Corrie' as returned by the subquery.

EXISTS Example:

```
SELECT name
FROM Beers b1
WHERE NOT EXISTS (SELECT * FROM Beers WHERE brewer = b1.brewer AND
name <> b1.name);
```

Explanation: This SQL query retrieves the names of beers from the `Beers` table, using the **`NOT EXISTS` operator** to ensure only those beers are selected that are the **unique beer** produced by their brewer, with no other beers from the same brewer present.

2.9 ANY and ALL Operators

ALL Example:

```
SELECT bar, beer, price
FROM Sells
WHERE price >= ALL (SELECT price FROM Sells WHERE price IS NOT
NULL);
```

Explanation: This SQL query retrieves the beers from the `Sells` table, using the **`ALL` operator** to ensure only those beers are selected whose price is **greater than or equal to every other price** in the `Sells` table.

2.10 Set Operations

Query:

```
(SELECT drinker FROM Likes)
INTERSECT
(SELECT drinker FROM Frequents);
```

Explanation: This SQL query retrieves the names of drinkers who appear in both the `Likes` and `Frequents` tables, using the **`INTERSECT` operator** to return only those drinkers who **like at least one beer** and **frequent at least one bar**.

2.11 Outer Joins

Query:

```
SELECT Sells.bar, Beers.name, Sells.price, Beers.brewer,
Beers.calories
FROM Sells
RIGHT OUTER JOIN Beers ON Sells.beer = Beers.name;
```

Explanation: This SQL query retrieves detailed information about the **bars and beers**, using a **`RIGHT OUTER JOIN`** to ensure that all beers from the `Beers` table are included, even if they are not sold in any bar.

2.12 Aggregations and Group By and HAVING Clause with Restrictions

Query:

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3;
```

Explanation: This SQL query retrieves the **average price** for each beer in the `Sells` table, but it applies additional filtering using the **`HAVING` clause** to ensure only includes beers that are sold in at least 3 bars.

Summary of Advanced SQL Commands Tutorial Using MariaDB

This tutorial covers advanced SQL concepts by loading and querying the festive database in MariaDB. Below is a concise summary of each topic and examples provided:

1. **Loading Data**
    - Use the `SOURCE` command to load the festive database with pre-defined tables (Bars, Beers, Drinkers, Frequents, Likes, and Sells).
2. **Renaming Columns with Aliases**
    - **Query Example:** `SELECT name AS bar_name FROM Bars;`

    - **Note:** Aliases only rename columns in the temporary result set. Use the ALTER command to change original column names permanently.
3. **Complex WHERE Conditions**
    - Combine conditions using AND/OR to filter data based on multiple criteria.
4. **Pattern Matching with LIKE**
    - Use % and _ wildcards to search for patterns within string data.
5. **Handling NULL Values**
    - Use IS NULL to identify missing values and learn how SQL handles three-valued logic (TRUE, FALSE, and UNKNOWN).
6. **Multi-Relation Queries (Joins)**
    - Use JOIN to combine data from multiple tables, ensuring meaningful relationships between them.
7. **Self-Joins**
    - Join a table with itself using tuple variables to compare related data within the same table.
8. **Subqueries**
    - Use nested queries to perform operations like filtering, comparisons, and dynamic value selection within larger queries.
9. **IN and EXISTS Operators**
    - Use IN to check membership in a set and EXISTS to verify the existence of matching data in a subquery.
10. **ANY and ALL Operators**

- Use ANY/ALL to compare a value against multiple values in a subquery.

11. **Set Operations** (UNION, INTERSECT, EXCEPT)

- Perform set operations to find common or distinct data across multiple queries.

12. **Aggregations** (SUM, AVG, COUNT, MIN, MAX)

- Use aggregate functions to compute summaries and statistics on datasets.

13. **GROUP BY Clause and Restrictions**

- Group rows by specific attributes and ensure non-aggregated columns appear in the GROUP BY clause.

14. **HAVING Clause and Restrictions**

- Use HAVING to filter groups after aggregation, ensuring only valid attributes and aggregates are used.

15. **Outer Joins**

- Use LEFT, RIGHT, and FULL OUTER JOIN to include unmatched rows in the result.

16. **Controlling Duplicates**

- Use DISTINCT to remove duplicate rows or use ALL to retain them.