Managing the World database

Objectives:

With this assignment, you will:

- Learn and practice working with different data types, constraints, and indexes.
- Gain hands-on experience with user management, privileges, and views.
- Understand the use of operators and query optimization techniques.
- Work with transactions to ensure data consistency.

Part 1: Data Types, Validation Restrictions & Indexes

Objective: To introduce and practice using different data types, apply validation restrictions, and understand how indexes improve performance.

Instructions:

1. **Explore the "world" database**:
   - List all tables in the "world" database.
   - Identify the different data types used in the `country`, `city`, and `countrylanguage` tables.
2. **Modify Data Types**:
   - Add a new column `is_population_large` (BOOLEAN) to the `city` table. This column will be `TRUE` if the population of the city is greater than 1 million, otherwise `FALSE`.
   - Create a new column in the `country` table, `region_code (CHAR(3))` with the `DEFAULT 'NA'` value.
3. **Add Validation Constraints**:
   - Add a `CHECK` constraint on the `city` table to ensure that the `population` value is never negative.
   - Ensure that the `country` table's `code` is unique using a `UNIQUE` constraint.
4. **Indexes**:
   - Create an index on the `city` table for the `name` column. Analyze how the index improves search performance by querying the table with and without the index.

Submission:

- **Queries**: Provide the SQL queries used in this part in a GitHub repository under the `/queries/data_types_indexes` directory.
- **Screenshots**: Include a screenshot of the query execution results showing the newly added columns, constraints, and indexes. Name the screenshot `part1_result.png`.

Part 2: Views, Users, and Privileges

Objective: To demonstrate the creation of views, manage users, and assign privileges.

Instructions:

1. **Views**:
   - Create a view named `high_population_cities` that shows cities with a population over 1 million. It should include the city name, country code, and population.
   - Create another view `countries_with_languages` that joins `country` and `countrylanguage` to display country name and language spoken (excluding `English` language).
2. **Users & Privileges**:
   - Create a new user `db_user` with the following privileges:
     - Read access (`SELECT`) on the `city` and `country` tables.
     - Write access (`INSERT`, `UPDATE`) on the `city` table.
   - Revoke the ability of `db_user` to modify the `country` table.
   - Grant the `db_user` full access to the `high_population_cities` view.
3. **Test Privileges**:
   - Log in as `db_user` and attempt to perform operations according to the granted/revoked privileges.

Submission:

- **Queries**: Provide the SQL queries used in this part in the GitHub repository under the `/queries/views_users_privileges` directory.
- **Screenshots**: Include screenshots showing:
  - The creation of the views.
  - The privileges granted to `db_user`.
  - The results of testing `db_user` permissions. Name the screenshots `part2_view_results.png` and `part2_user_privileges.png`.

Part 3: Operators, Comparators, and Logical Operators

Objective: To practice using operators and logical comparators to filter and manipulate data.

Instructions:

1. **Comparison Operators**:
   - Write a query to find all countries where the population is greater than 50 million but less than 200 million.

    - Use `IN` to find all countries with population in a specific range (e.g., between 20 million and 50 million).
2. **Logical Operators**:
    - Use the `AND`, `OR`, and `NOT` operators to write a query that retrieves all cities where the population is between 1 million and 10 million, but excludes cities in the "Asia" region.
3. **Complex Queries**:
    - Write a query that retrieves countries where the population is either over 100 million or the region is “Europe”.
    - Combine `LIKE` and `NOT` to find cities with names starting with 'A' but exclude those that end with 'n'.

Submission:

- **Queries**: Provide the SQL queries used in this part in the GitHub repository under the `/queries/operators_comparators_logical` directory.
- **Screenshots**: Include a screenshot of the results from executing these queries. Name the screenshot `part3_query_results.png`.

Part 4: Internal and External Composition of Data

Objective: To practice subqueries, joins, and derived tables.

Instructions:

1. **Internal Composition**:
    - Write a subquery that lists countries that have more than 5 cities with a population greater than 1 million.
    - Use a subquery in the `WHERE` clause to find all countries with more than 3 official languages.
2. **External Composition**:
    - Write a query that joins `city`, `country`, and `countrylanguage` to find cities with at least one official language spoken that is not English.
    - Use a derived table to calculate the total population of each country (i.e., sum the populations of all cities in each country).

Submission:

- **Queries**: Provide the SQL queries used in this part in the GitHub repository under the `/queries/subqueries_joins_compositions` directory.
- **Screenshots**: Include a screenshot of the results from executing these queries. Name the screenshot `part4_query_results.png`.

Part 5: Query Optimization

Objective: To introduce query optimization techniques for better performance.

Instructions:

1. **Optimizing a Query**:
   - Write a query that retrieves the 10 cities with the highest populations. (This will involve sorting and limiting the results).
   - Investigate the performance of this query by running `EXPLAIN` on the query plan and observing what indexes are being used.
   - Rewrite the query using `LIMIT` and an appropriate index to optimize performance.
2. **Using Indexes for Optimized Search**:
   - Write a query that searches for all cities with a population greater than 1 million and a name starting with "A". Ensure that the query is optimized using indexing.

Submission:

- **Queries**: Provide the SQL queries used in this part in the GitHub repository under the `/queries/query_optimization` directory.
- **Screenshots**: Include a screenshot of the `EXPLAIN` plan results. Name the screenshot `part5_explain_results.png`.

Part 6: Transactions

Objective: To practice working with transactions to ensure atomicity, consistency, isolation, and durability (ACID properties).

Instructions:

1. **Basic Transaction**:
   - Start a transaction and insert a new city into the `city` table. Rollback the transaction after checking the inserted data.
2. **Multiple Operations in a Transaction**:
   - Begin a transaction that inserts a new city into the `city` table and updates the population of an existing country in the `country` table. Commit the transaction only if both operations are successful. If any operation fails, rollback the transaction.
3. **Transaction Management**:
   - Use `SAVEPOINT` and `ROLLBACK TO SAVEPOINT` to manage partial rollbacks during the transaction.
   - Create a situation where you simulate an error in the middle of a transaction to show how rollback works.

Submission:

- **Queries**: Provide the SQL queries used in this part in the GitHub repository under the `/queries/transactions` directory.
- **Screenshots**: Include a screenshot of the results from your transaction operations (e.g., after committing or rolling back). Name the screenshot `part6_transaction_results.png`.

Submission Instructions:

1. **In your personal repository, create a subfolder, named "managing-the-world-database", with the following folder structure:**

```
/queries
  /data_types_indexes
  /views_users_privileges
  /operators_comparators_logical
  /subqueries_joins_compositions
  /query_optimization
  /transactions
```

   - Submit all your SQL queries as `.sql` files in the appropriate folders.

2. **Screenshots**:
   - Include screenshots of your query results in the repository in a folder named `/screenshots`.
   - Ensure that the screenshots are named according to the relevant part (e.g., `part1_result.png`, `part5_explain_results.png`).

Grading Criteria:

- **Correctness**: Ensure all SQL queries execute correctly.
- **Completeness**: Include all queries, explanations, and screenshots as required.
- **Clarity**: Maintain clear and consistent naming conventions for files and folders in the GitHub repository.