Just IT

B2W group

Apprenticeships | Training | Recruitment

Data Technician

Name: LAIBA SHAUKAT

Course Date: 12 MAY 2025

Table of contents

Day 1: Task 1

Please research and complete the below questions relating to key concepts of databases.

What is a primary key?	A primary key is a column (or set of columns) that uniquely identifies each record in a table. It cannot be NULL and must be unique.
How does this differ from a secondary key?	Secondary Key (Alternate Key): • A secondary key is any candidate key that is not chosen to be the primary key. • It can also uniquely identify records, like the primary key, but it's not the main identifier. • It is often used for querying or searching, not necessarily for enforcing record uniqueness (though it can be). • It can be declared with a UNIQUE constraint. • Example: If email is unique for each employee, it could be a secondary key. CREATE TABLE Employees (employee_id INT PRIMARY KEY, name VARCHAR(100), email VARCHAR(100) UNIQUE);
How are primary and foreign keys related?	Primary and foreign keys are closely related because they work together to create relationships between tables in a database. A primary key uniquely identifies each record in a table. A foreign key, on the other hand, is a field (or fields) in one table that refers to the primary key in another table. This creates a link between the two tables and ensures referential integrity—meaning, the foreign key must match an existing value in the related primary key column or be NULL (if allowed). For example, if you have a Customers table with customer_id as the primary key, and an Orders table where each order needs to know which customer it belongs to, the Orders table would

	include customer_id as a foreign key. This setup ensures that every order is connected to a valid customer. So in summary: a foreign key references a primary key in another table to establish a relationship between records in different tables.
Provide a real-world example of a one-to-one relationship	Here's a SQL example showing a one-to-one relationship between Persons and Passports: -- Create the Persons table CREATE TABLE Persons (person_id INT PRIMARY KEY, name VARCHAR(100), date_of_birth DATE); -- Create the Passports table with a one-to-one relationship to Persons CREATE TABLE Passports (passport_id INT PRIMARY KEY, person_id INT UNIQUE, -- Ensures one passport per person issue_date DATE, expiry_date DATE, FOREIGN KEY (person_id) REFERENCES Persons(person_id)); **Key points:** • person_id is the primary key in the Persons table. • person_id in the Passports table is a foreign key that also has a UNIQUE constraint, enforcing that each person can be linked to only one passport. • This structure ensures a strict one-to-one relationship.
Provide a real-world example of a one-to-many relationship	A real-world example of a one-to-many relationship is the relationship between a customer and their orders. **Example:** • One customer can place many orders. • But each order is placed by only one customer.

	-- Create the Customers table CREATE TABLE Customers (customer_id INT PRIMARY KEY, name VARCHAR(100), email VARCHAR(100)); -- Create the Orders table with a one-to-many relationship to Customers CREATE TABLE Orders (order_id INT PRIMARY KEY, customer_id INT, -- Foreign key referencing Customers order_date DATE, total_amount DECIMAL(10, 2), FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)); **Explanation:** • customer_id in Orders is a foreign key pointing to the Customers table. • This setup allows each customer to have multiple orders, but each order is tied to only one customer—establishing a one-to-many relationship.
Provide a real-world example of a many-to-many relationship	A real-world example of a many-to-many relationship is the relationship between students and courses. Example: • One student can enroll in many courses. • One course can have many students enrolled in it. To represent this in a database, we use a junction (bridge) table to manage the relationship. Tables: Students Table: student_id (Primary Key) • name Courses Table: course_id (Primary Key) • course_name Enrolments Table (Junction table): student_id (Foreign Key) • course_id (Foreign Key) • enrollment_date Together, the student_id and course_id in Enrolments form a composite primary key. -- Students table CREATE TABLE Students (student_id INT PRIMARY KEY, name VARCHAR(100));

-- Courses table CREATE TABLE Courses (course_id INT PRIMARY KEY, course_name VARCHAR(100));

-- Enrolments junction table to handle many-to-many relationship CREATE TABLE Enrolments (student_id INT, course_id INT, enrollment_date DATE, PRIMARY KEY (student_id, course_id), FOREIGN KEY (student_id) REFERENCES Students(student_id), FOREIGN KEY (course_id) REFERENCES Courses(course_id)); Explanation:

• Each student can appear in the Enrolments table multiple times for different courses.

• Each course can appear multiple times for different students.

• This structure effectively captures the many-to-many relationship.

Day 1: Task 2

Please research and complete the below questions relating to key concepts of databases.

What is the difference between a relational and non-relational database?	A relational database organizes data into structured tables (rows and columns) where relationships between data are defined using keys—typically primary and foreign keys. These databases follow a strict schema and use SQL (Structured Query Language) for querying and managing data. They are ideal for applications where data integrity, complex queries, and transactions are important—such as banking systems, HR platforms, and traditional business applications. Examples include MySQL, PostgreSQL, Oracle, and SQL Server. On the other hand, a non-relational database (often called NoSQL) stores data in a more flexible format—such as documents, key-value pairs, graphs, or wide-column stores—without requiring a fixed schema. This allows for fast scalability and flexibility in handling unstructured or semi-structured data. Non-relational databases are commonly used in big data, real-time analytics, IoT, and applications with rapidly changing data. Examples include MongoDB (document-based), Redis (key-value), Cassandra (wide-column), and Neo4j (graph-based). In short, relational databases are structured, schema-based, and best for consistent, transactional data, while non-relational databases are more flexible and suited for dynamic or large-scale data with varying structures.
What type of data would benefit off the non-relational model? **Why?**	Non-relational databases are ideal for handling data that is unstructured, semi-structured, or rapidly changing, because they offer flexibility, scalability, and performance that traditional relational models may struggle with. Here are some types of data that benefit most from a non-relational model and why: 1. **Unstructured or Semi-Structured Data (e.g., JSON, XML, multimedia)** Why: Non-relational databases like MongoDB allow storage of data in flexible formats such as JSON-like documents, which can have varying fields and nested structures. This is perfect for storing user profiles, product catalogs, or social media posts where not every record needs to have the same fields.

2. Big Data and Real-Time Analytics

Why: Wide-column stores like Cassandra or key-value stores like Redis are optimized for speed and horizontal scalability. They are great for handling massive volumes of data across distributed systems—like sensor data from IoT devices or real-time user interactions on websites.

3. Content Management Systems

Why: Content like blog posts, articles, and media files often vary in structure. Document databases let you store diverse content types without redesigning your schema each time the structure changes.

4. E-commerce Product Data

Why: Products often have unique attributes (e.g., shoes have sizes, books have authors). Non-relational models allow storing products with different fields in the same collection, making it easy to manage diverse inventories without complex joins or schema changes.

5. Social Networks or Graph Data

Why: Graph databases like Neo4j excel at managing interconnected data—such as friendships, follows, or recommendations—where relationships between entities are as important as the data itself.

Summary:

Non-relational databases are best when flexibility, scalability, and speed are more important than strict consistency or complex joins. They thrive in environments where data doesn't fit neatly into rows and columns, and where adapting quickly to changing data structures is essential.

Day 3: Task 1

Please research the below 'JOIN' types, explain what they are and provide an example of the types of data it would be used on.

Self-join	A SELF JOIN is a type of SQL join where a table is joined to itself. This is useful when you need to compare rows within the same table — in other words, you're matching rows in a table with other rows in the same table based on a related column.

Since a table cannot refer to itself directly in a join, we use table aliases to differentiate the instances of the table.

When to Use a SELF JOIN

You would use a SELF JOIN in situations like:

- Hierarchical relationships: such as employees and managers stored in the same table.
- Finding duplicates or comparing rows.
- Relating products to other products in the same category.

Example Scenario: Employee and Manager

Each employee has a ManagerID that refers to another employee's EmployeeID.

To find out which employees report to which managers, you could use a SELF JOIN like this:

```
SELECT
    E1.Name AS Employee,
    E2.Name AS Manager
FROM
    Employees E1
JOIN
    Employees E2
ON
    E1.ManagerID = E2.EmployeeID;
```

Summary

- SELF JOIN joins a table with itself using table aliases.
- It's used for comparing rows in the same table, often in hierarchical or relational structures.
- Useful in organizational charts, product relationships, and self-referencing data.

Right join	A RIGHT JOIN (also known as RIGHT OUTER JOIN) returns all the rows from the right table, and the matching rows from the left table. If there is no match, NULL values are returned for the columns from the left table. This is useful when you want to retrieve all records from the second (right) table, regardless of whether they have related records in the first (left) table. SELECT columns FROM left_table RIGHT JOIN right_table ON left_table.column = right_table.column; **When to Use RIGHT JOIN:** • To show all entries from the right table even if they have no matching entry in the left. **For example:** • All products with or without sales • All customers regardless of whether they've made purchases • All classes offered, even if no students are enrolled. **Summary:** • RIGHT JOIN keeps all rows from the right table. • Fills in NULLs where there is no match in the left table. • Best for cases where the right table is your "main" dataset and you want to see its full content.

Full join	A FULL JOIN (also called FULL OUTER JOIN) returns all records from both tables, matching rows where possible. If there is no match, the result will contain NULLs for the missing side. It's essentially a combination of a LEFT JOIN and a RIGHT JOIN: • All records from the left table • All records from the right table • Matching records show combined data • Non-matching records show NULL for the missing side. SELECT columns FROM table1 FULL JOIN table2 ON table1.column = table2.column; **When to Use FULL JOIN:** • When you want to combine two tables and see everything from both, regardless of matches • Great for reconciling data or comparing two datasets • Example: All employees and all projects, even if not assigned • All invoices and all payments, even if unmatched **Summary:** • FULL JOIN = LEFT JOIN + RIGHT JOIN • Includes all rows from both tables • Uses NULLs where there's no match • Useful for comprehensive reports, audits, or data checks.
Inner join	An INNER JOIN is the most used type of join in SQL. It returns only the rows that have matching values in both tables. If there is no match between the rows, those rows are excluded from the result. SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column; **When to Use INNER JOIN:**

	• When you only want rows that exist in both tables based on a condition. • Examples: • List of products sold • Employees assigned to a project • Customers who placed orders • Students enrolled in classes **Visual Summary:** Imagine two overlapping circles (like a Venn diagram). The INNER JOIN returns only the middle part — the overlap — where data from both tables matches. **Summary:** • INNER JOIN returns only matching rows from both tables. • It's the default type of join if you just write JOIN. • Best for filtering data to only include related or relevant records.
Cross join	A CROSS JOIN (also called a Cartesian Join) returns the Cartesian product of two tables — meaning it combines every row from the first table with every row from the second table. *If Table A has n rows and Table B has m rows, the result of a CROSS JOIN will have n × m rows. SELECT * FROM table1 CROSS JOIN table2; **When to Use CROSS JOIN:** • When you need every combination of values between two datasets • Generating: • Price lists for combinations (e.g., size × color) • Schedules (e.g., classes × time slots) • Test cases (e.g., software configs × user roles) • Often used with filters or WHERE clauses to reduce the number of combinations afterward **Important Notes:**

	• No ON clause is used in a CROSS JOIN — unless it's filtered with a WHERE. • CROSS JOINs can produce very large result sets if both tables are big — so use with caution. • Often used for reference data, not large transactional data. **Summary:** • CROSS JOIN = all combinations of rows from both tables • Good for generating test data or possible configurations • Can create large results if not managed properly
Left join	A LEFT JOIN (also known as a LEFT OUTER JOIN) returns all records from the left table, and the matched records from the right table. If there is no match, NULLs are returned for columns from the right table. SELECT columns FROM left_table LEFT JOIN right_table ON left_table.column = right_table.column; **When to Use LEFT JOIN:** • When you want to see all rows from the left table, even if there's no match in the right table. • Customers and orders (including customers with no orders) • Products and sales (including unsold products) • Students and classes (including unenrolled students) **Summary:** • LEFT JOIN = all rows from the left table, and matching rows from the right table

- Returns NULL where no match exists
- Useful for identifying missing data or producing complete lists

Day 4: Task 1: Written

In your groups, discuss and complete the below activity. You can either nominate one writer or split the elements between you. Everyone however must have the completed work below:

Imagine you have been hired by a small retail business that wants to streamline its operations by creating a new database system. This database will be used to manage inventory, sales, and customer information. The business is a small corner shop that sells a range of groceries and domestic products. It might help to picture your local convenience store and think of what they sell. They also have a loyalty program, which you will need to consider when deciding what tables to create.

Write a 500-word essay explaining the steps you would take to set up and create this database. Your essay should cover the following points:

1. ***Understanding the Business Requirements**:*
    a. *What kind of data will the database need to store?*
    b. *Who will be the users of the database, and what will they need to accomplish?*
2. ***Designing the Database Schema**:*
    a. *How would you structure the database tables to efficiently store inventory, sales, and customer information?*
    b. *What relationships between tables are necessary (e.g., how sales relate to inventory and customers)?*
3. ***Implementing the Database**:*
    a. *What SQL commands would you use to create the database and its tables?*
    b. *Provide examples of SQL statements for creating tables and defining relationships between them.*
4. ***Populating the Database**:*
    a. *How would you input initial data into the database? Give examples of SQL INSERT statements.*
5. ***Maintaining the Database**:*
    a. *What measures would you take to ensure the database remains accurate and up to date?*
    b. *How would you handle backups and data security?*

Your essay should include specific examples of SQL commands and explain why each step is necessary for creating a functional and efficient database for the retail business.

Please write your 500-word essay here	**Setting Up a Database for a Small Retail Business** Creating a database for a small retail business involves careful planning, design, and implementation to ensure the system supports daily operations such as managing inventory, processing sales, and handling customer information, including a loyalty program. Here's how I would approach this project. Understanding the Business Requirements First, it's essential to identify the data the business needs to store. This includes: • Inventory data: product name, category, quantity in stock, supplier, price, and restock level. • Sales data: date of sale, items sold, quantity, total amount, and payment method. • Customer data: name, contact details, loyalty card number, and accumulated loyalty points. Users of the database will include store staff (to update inventory and record sales), managers (to analyse sales trends and stock levels), and possibly customers (for loyalty program tracking via a simple interface). Each user group will need specific access tailored to their tasks. Designing the Database Schema To support this functionality, the database will include the following main tables: • Products: stores information about each product. • Customers: holds customer details and loyalty information. • Sales: records each sales transaction. • Sales_Items: records which products are sold in each transaction. • Inventory: manages current stock levels (this could be merged with Products or separate depending on design choice). Relationships: • One customer can have many sales. • Each sale can involve many products. • Each product can be sold in many sales.

- Sales_Items acts as a junction table between Sales and Products.

Implementing the Database

Using SQL, the database and its tables can be created as follows:

```
CREATE DATABASE RetailDB;
USE RetailDB;

CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100),
    Email VARCHAR(100),
    Phone VARCHAR(15),
    LoyaltyPoints INT DEFAULT 0
);

CREATE TABLE Products (
    ProductID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100),
    Category VARCHAR(50),
    Price DECIMAL(10,2),
    Stock INT,
    RestockLevel INT
);

CREATE TABLE Sales (
    SaleID INT PRIMARY KEY AUTO_INCREMENT,
    CustomerID INT,
    SaleDate DATETIME DEFAULT CURRENT_TIMESTAMP,
    TotalAmount DECIMAL(10,2),
    FOREIGN KEY (CustomerID) REFERENCES
Customers(CustomerID)
);

CREATE TABLE Sales_Items (
    SaleItemID INT PRIMARY KEY AUTO_INCREMENT,
    SaleID INT,
    ProductID INT,
```

```
    Quantity INT,
    FOREIGN KEY (SaleID) REFERENCES Sales(SaleID),
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

Populating the Database

To input initial data, we use INSERT statements:

```
INSERT INTO Products (Name, Category, Price, Stock,
RestockLevel)
VALUES ('Milk', 'Dairy', 1.50, 50, 10);

INSERT INTO Customers (Name, Email, Phone)
VALUES ('Jane Doe', 'jane@example.com', '123-456-7890');

INSERT INTO Sales (CustomerID, TotalAmount)
VALUES (1, 3.00);

INSERT INTO Sales_Items (SaleID, ProductID, Quantity)
VALUES (1, 1, 2);
```

Maintaining the Database

To keep the database accurate and up to date, I would:

- Implement constraints (e.g., CHECK constraints on quantity and price).
- Use triggers to update stock levels automatically after a sale.
- Schedule regular backups using tools like mysqldump.
- Protect data using user roles and permissions, ensuring only authorized users can make changes.

For example:

```
CREATE USER 'store_clerk'@'localhost' IDENTIFIED BY
'password';
GRANT SELECT, INSERT, UPDATE ON RetailDB.* TO
'store_clerk'@'localhost';
```

In conclusion, designing and building a database for a small retail business requires a structured approach that includes understanding the business needs, creating a well-thought-out schema, implementing the system using SQL, and ensuring

long-term maintenance and security. With this approach, the business can efficiently manage its operations and improve customer service.

Day 4: Task 2: SQL Practical

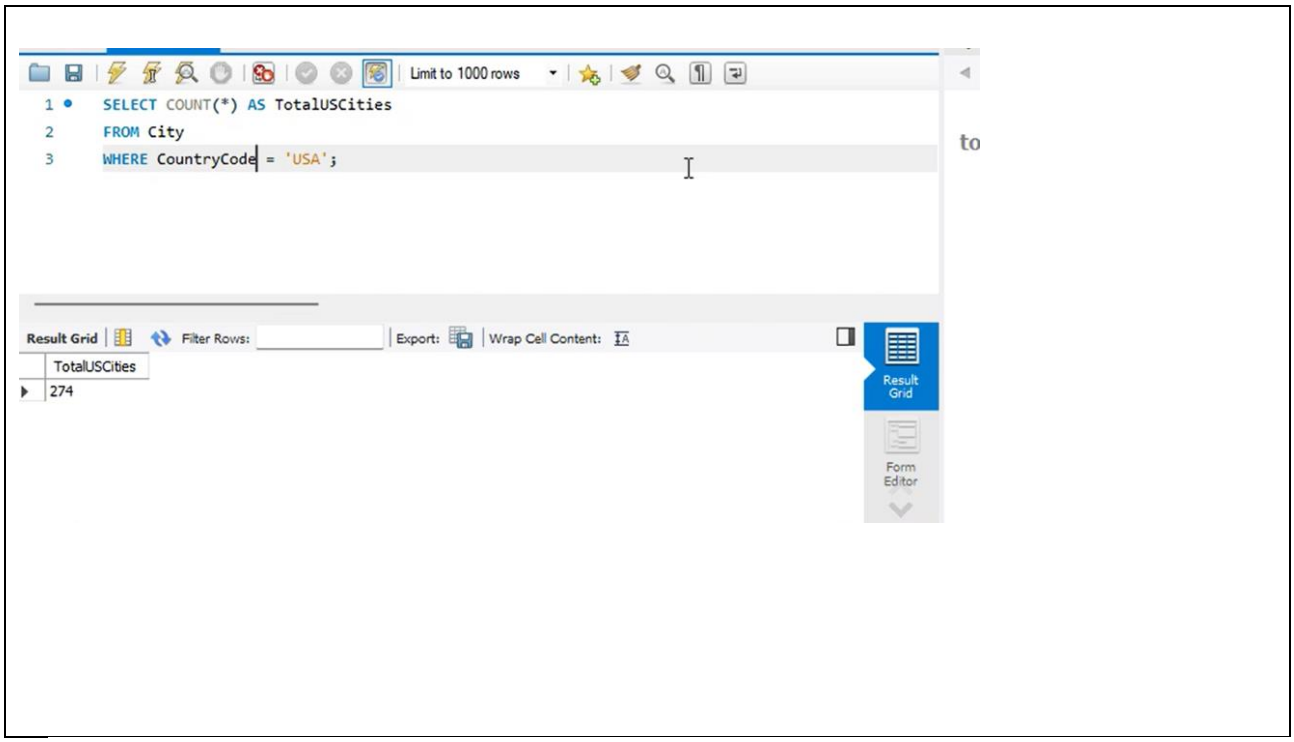
In your groups, work together to answer the below questions. It may be of benefit if one of you shares your screen with the group and as a team answer / take screen shots from there.

Setting up the database:

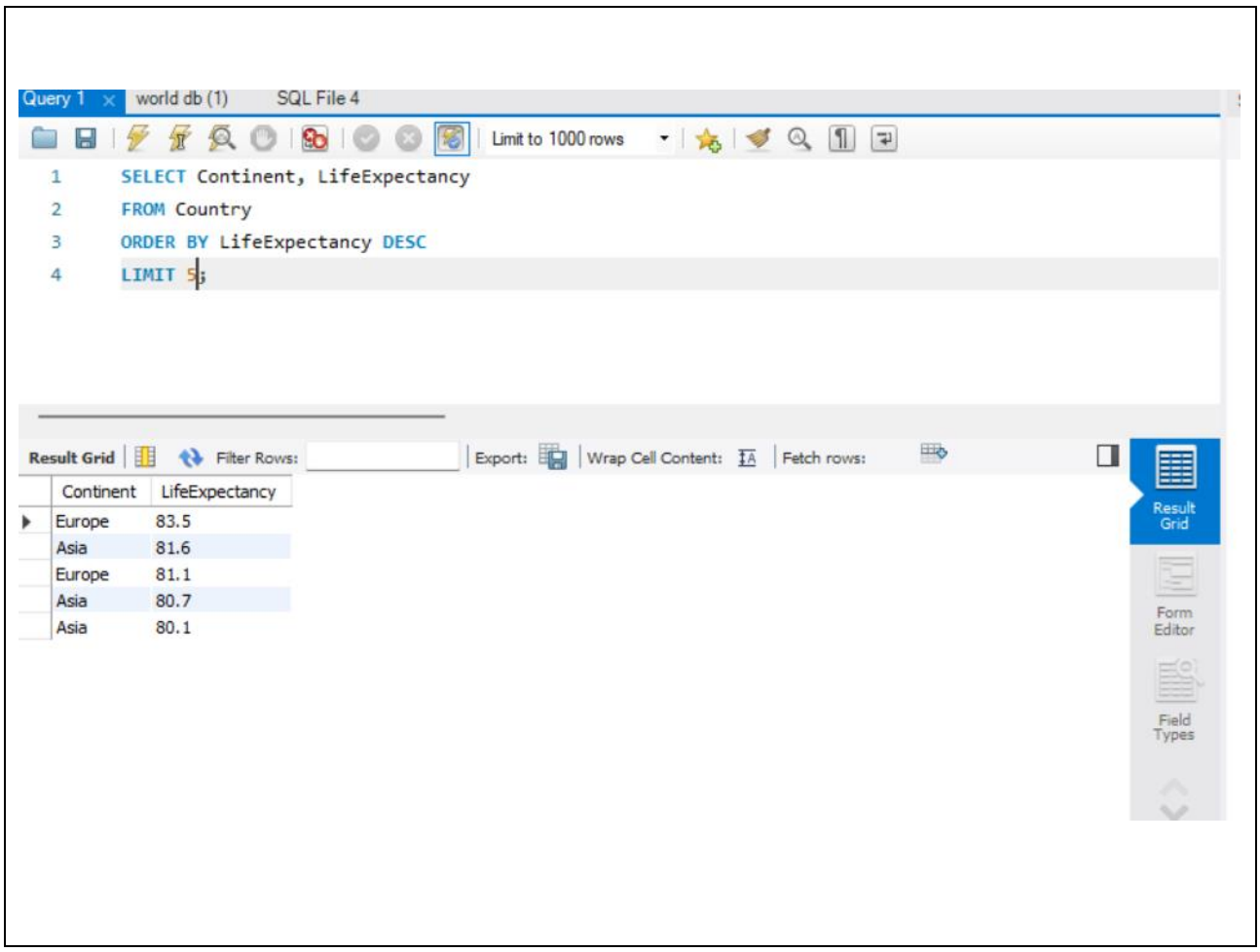
1. **Download world_db(1) here**
2. **Follow each step to create your database here**

For each question I would like to see both the syntax used and the output.

1. **Count Cities in USA:** *Scenario:* You've been tasked with conducting a demographic analysis of cities in the United States. Your first step is to determine the total number of cities within the country to provide a baseline for further analysis.



2. **Country with Highest Life Expectancy:** *Scenario:* As part of a global health initiative, you've been assigned to identify the country with the highest life expectancy. This information will be crucial for prioritising healthcare resources and interventions.



3. **"New Year Promotion: Featuring Cities with 'New :** *Scenario:* In anticipation of the upcoming New Year, your travel agency is gearing up for a special promotion featuring cities with names including the word 'New'. You're tasked with swiftly compiling a list of all cities from around the world. This curated selection will be essential in creating promotional materials and enticing travellers with exciting destinations to kick off the New Year in style.

```sql
SELECT *
FROM city
WHERE name LIKE '%New%'
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

ID	Name	CountryCode	District	Population
137	Newcastle	AUS	New South Wales	270324
482	Newcastle upon Tyne	GBR	England	189150
502	Newport	GBR	Wales	139000
734	Newcastle	ZAF	KwaZulu-Natal	222993
936	Kowloon and New Kowloon	HKG	Kowloon and New Kowl	1987996
1106	New Bombay	IND	Maharashtra	307297
1109	New Delhi	IND	Delhi	301297
2857	Khanewal	PAK	Punjab	133000
3793	New York	USA	New York	8008278
3823	New Orleans	USA	Louisiana	484674
3855	Newark	USA	New Jersey	273546
3905	Newport News	USA	Virginia	180150

Result Grid | Form Editor | Field Types

4. **Display Columns with Limit (First 10 Rows):** *Scenario:* You're tasked with providing a brief overview of the most populous cities in the world. To keep the report concise, you're instructed to list only the first 10 cities by population from the database.

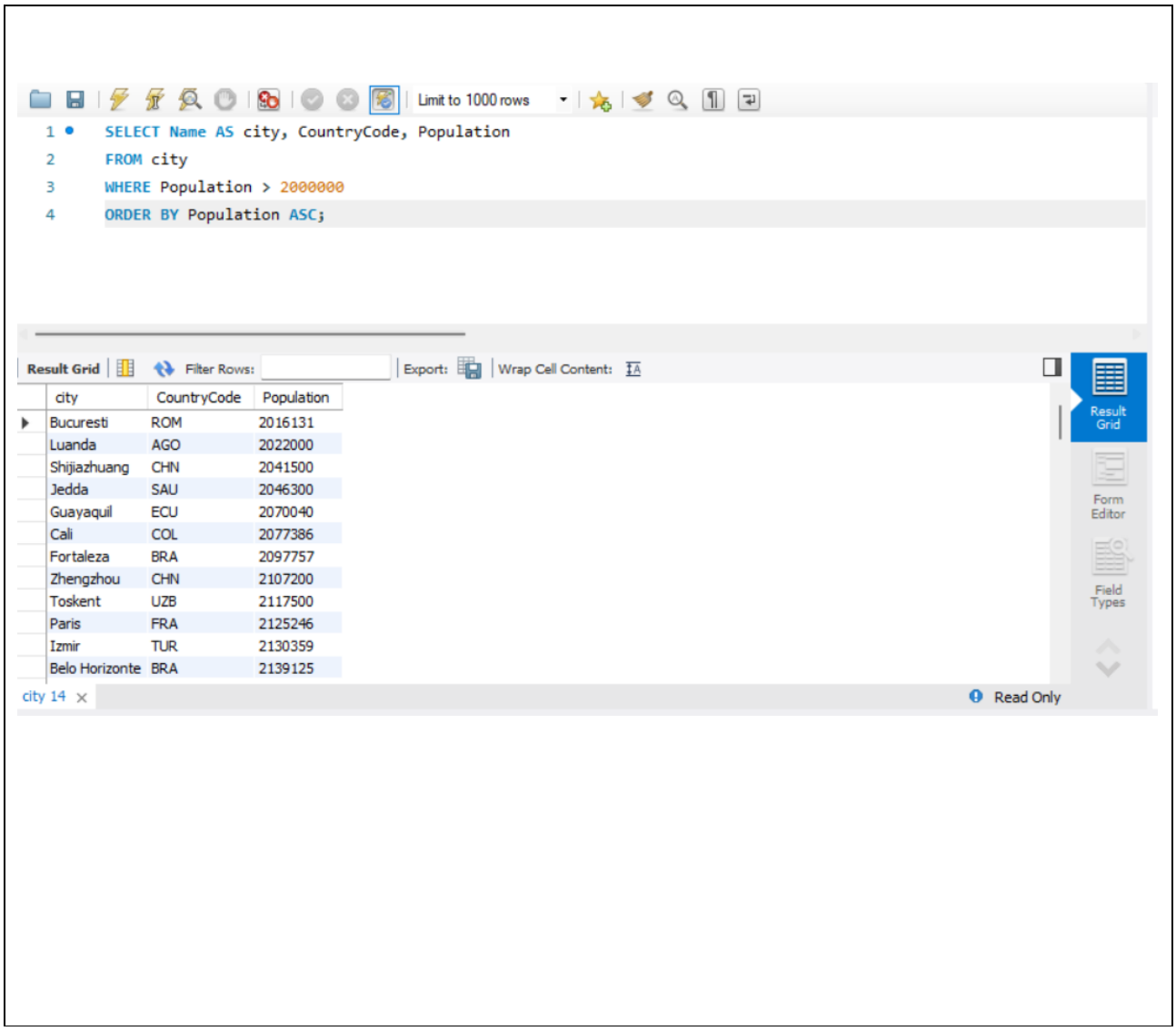
```sql
SELECT *
FROM city
order by Population DESC
LIMIT 10;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: | Fetch rows:

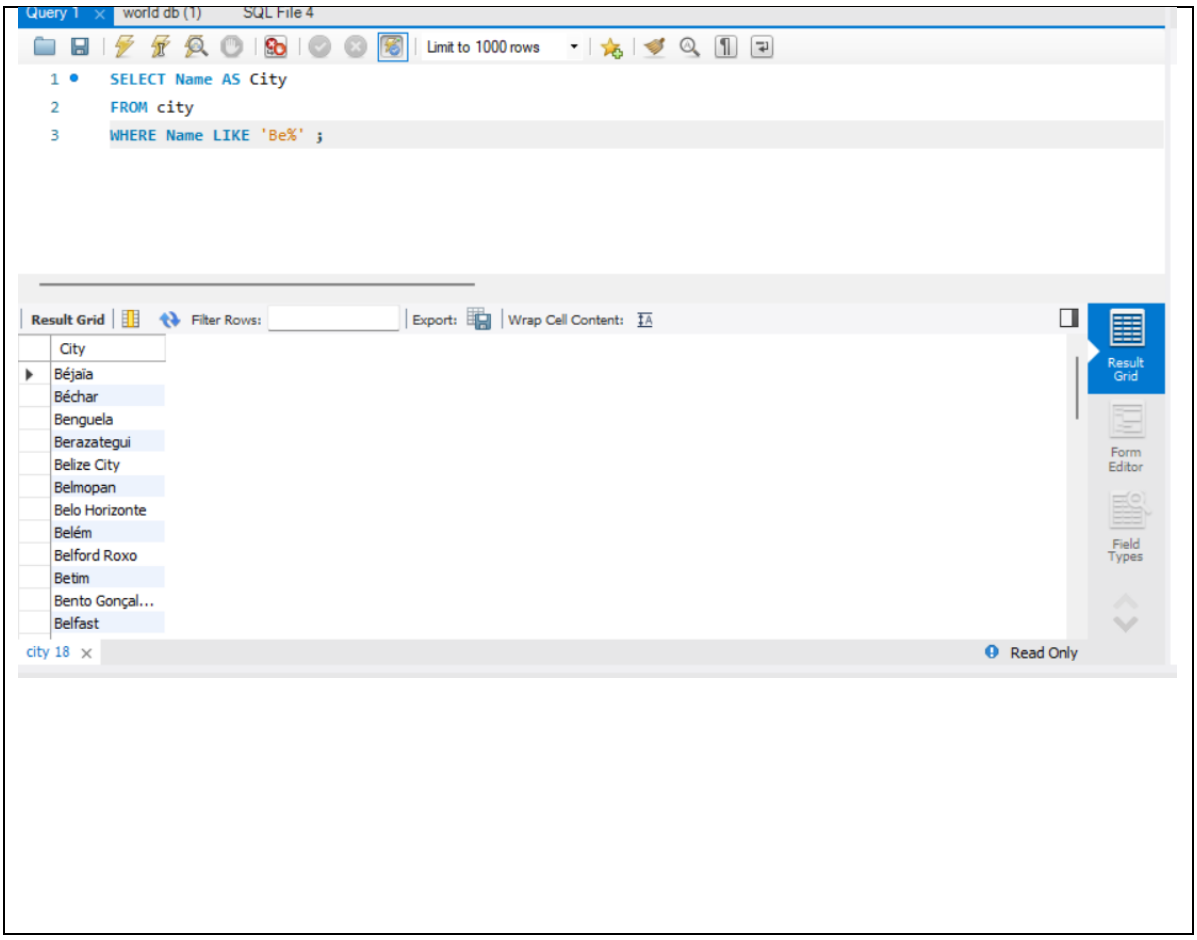
ID	Name	CountryCode	District	Population
1024	Mumbai (Bombay)	IND	Maharashtra	10500000
2331	Seoul	KOR	Seoul	9981619
206	São Paulo	BRA	São Paulo	9968485
1890	Shanghai	CHN	Shanghai	9696300
939	Jakarta	IDN	Jakarta Raya	9604900
2822	Karachi	PAK	Sindh	9269265
3357	Istanbul	TUR	Istanbul	8787958
2515	Ciudad de México	MEX	Distrito Federal	8591309
3580	Moscow	RUS	Moscow (City)	8389200
3793	New York	USA	New York	8008278
NULL	NULL	NULL	NULL	NULL

Result Grid | Form Editor | Field Types

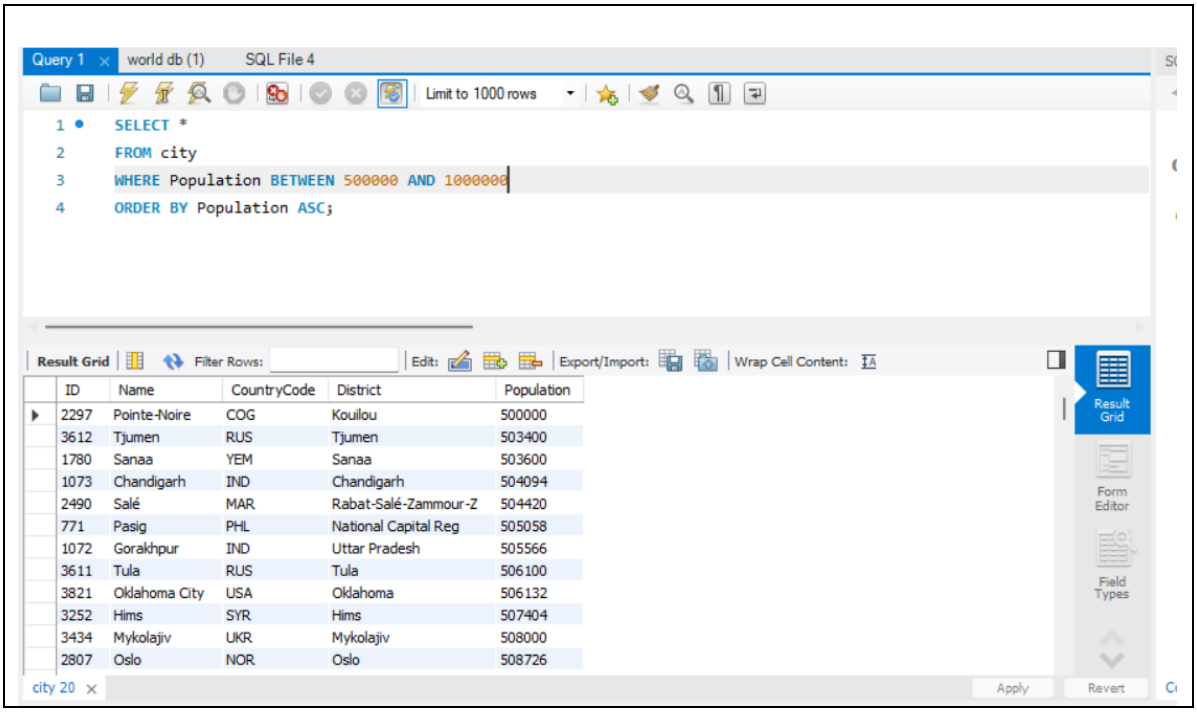
5. **Cities with Population Larger than 2,000,000:** *Scenario:* A real estate developer is interested in cities with substantial population sizes for potential investment opportunities. You're tasked with identifying cities from the database with populations exceeding 2 million to focus their research efforts.



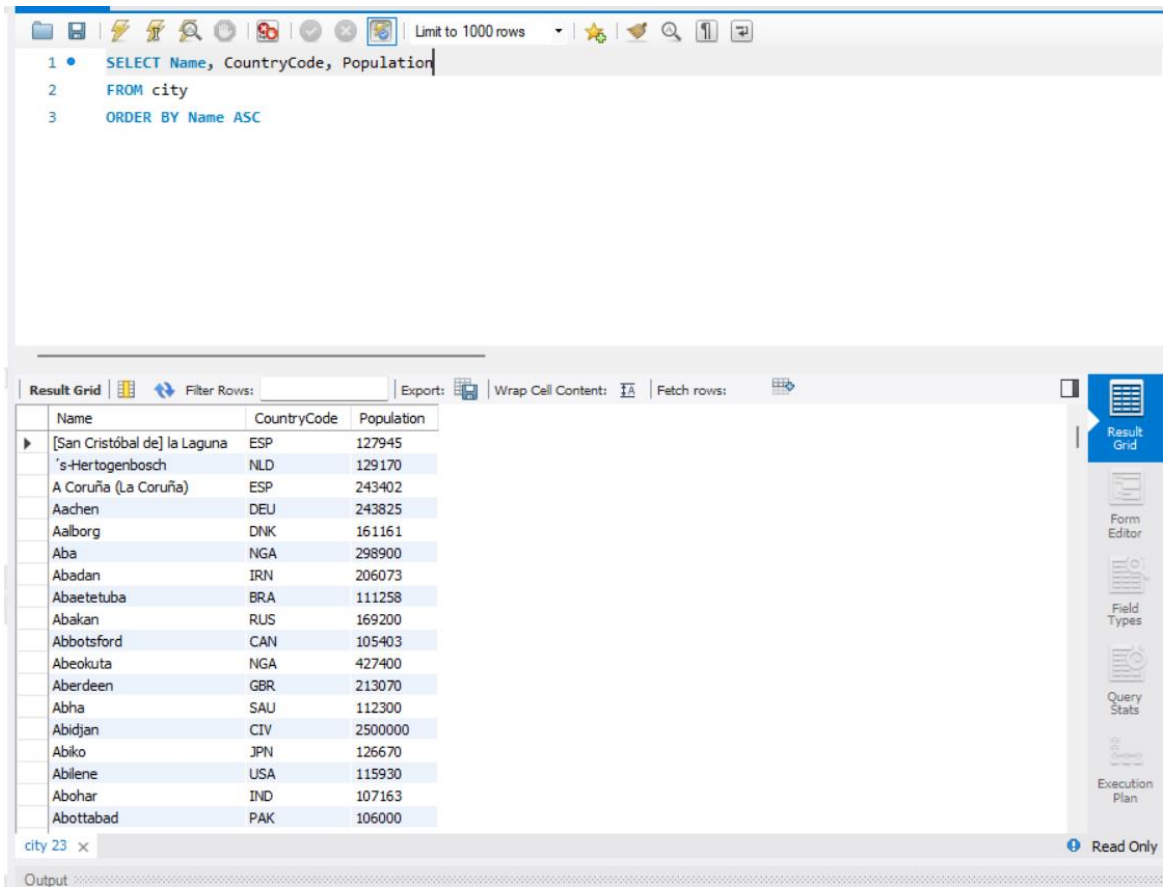
6. **Cities Beginning with 'Be' Prefix:** *Scenario:* A travel blogger is planning a series of articles featuring cities with unique names. You're tasked with compiling a list of cities from the database that start with the prefix 'Be' to assist in the blogger's content creation process.



7. **Cities with Population Between 500,000-1,000,000:** *Scenario:* An urban planning committee needs to identify mid-sized cities suitable for infrastructure development projects. You're tasked with identifying cities with populations ranging between 500,000 and 1 million to inform their decision-making process.

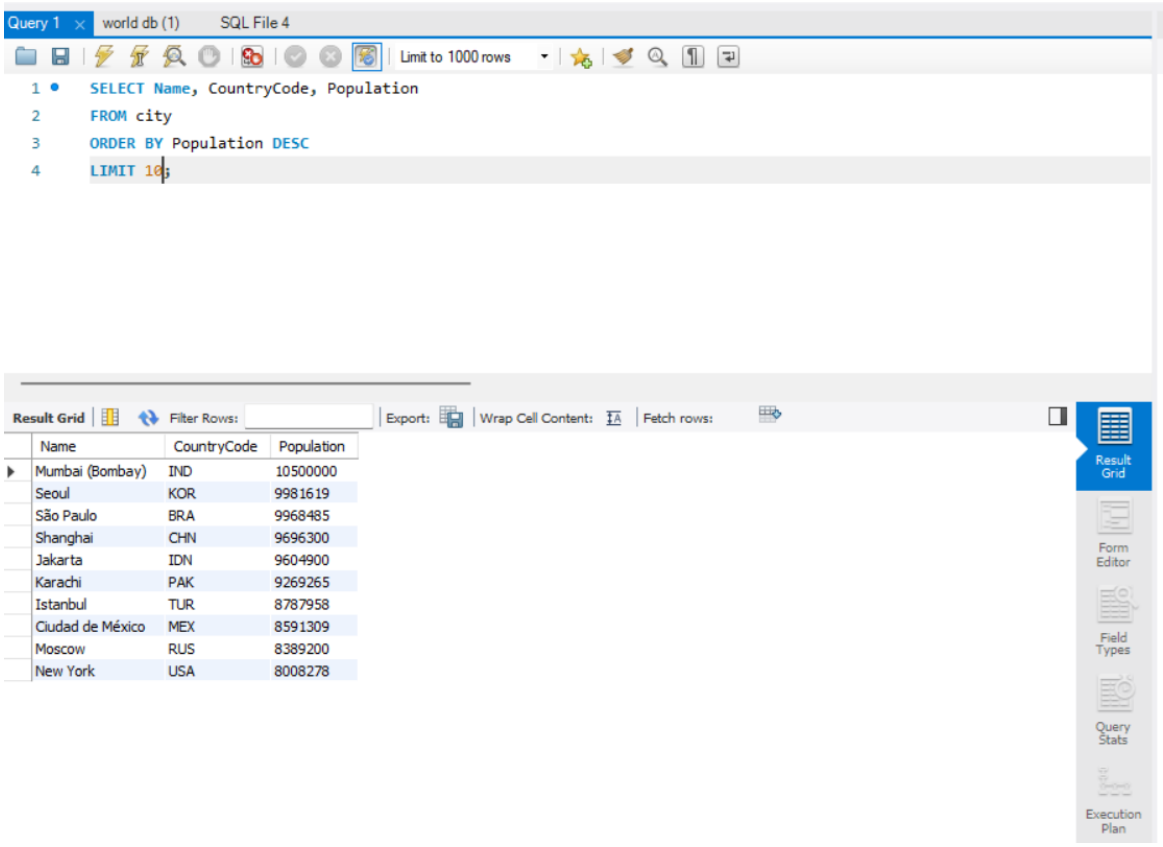


8. **Display Cities Sorted by Name in Ascending Order:** *Scenario:* A geography teacher is preparing a lesson on alphabetical order using city names. You're tasked with providing a sorted list of cities from the database in ascending order by name to support the lesson plan.



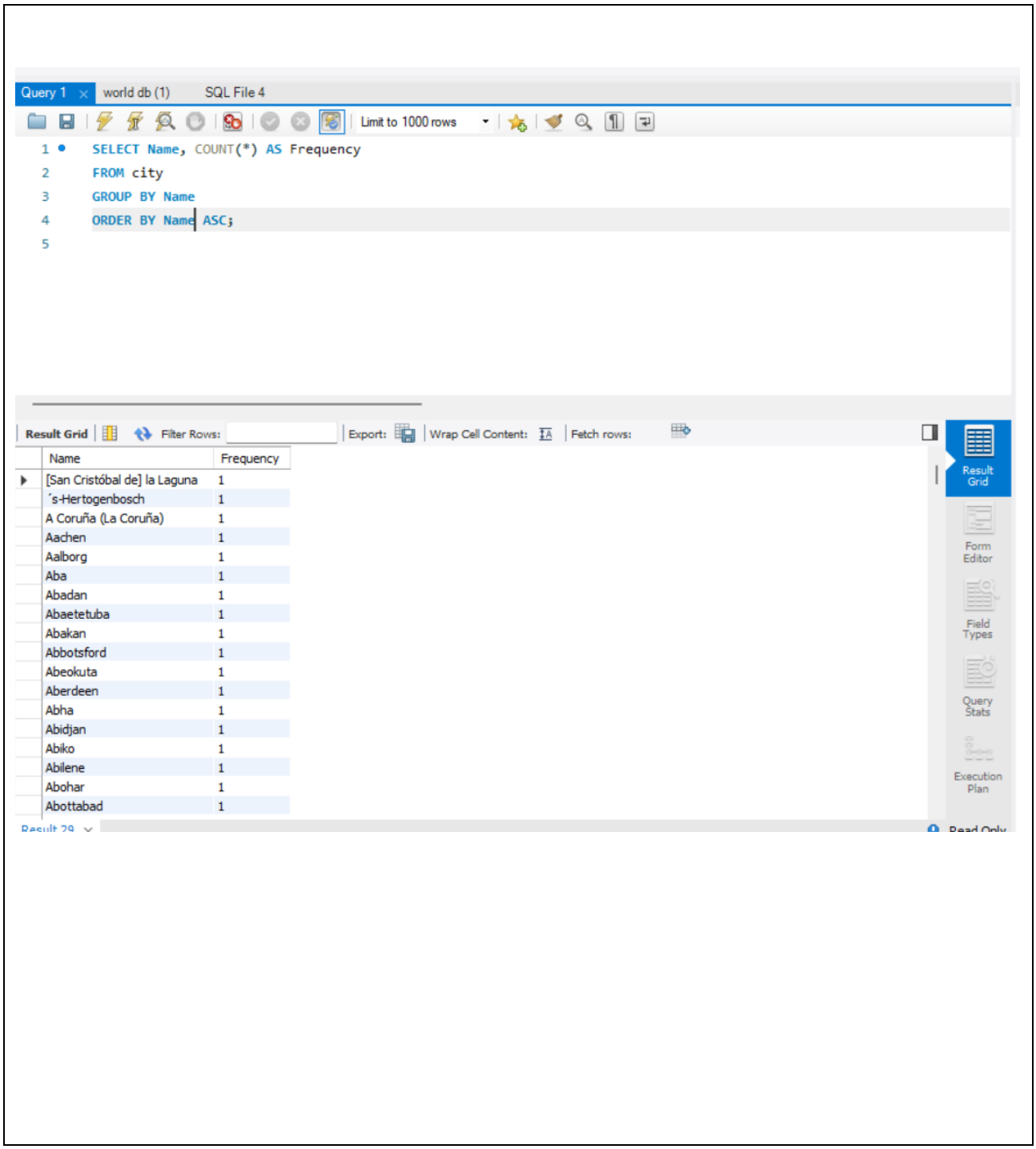
9. **Most Populated City:** *Scenario:* A real estate investment firm is interested in cities with significant population densities for potential development projects. You're tasked with identifying the most populated city from the database to guide their investment decisions and strategic planning.

.

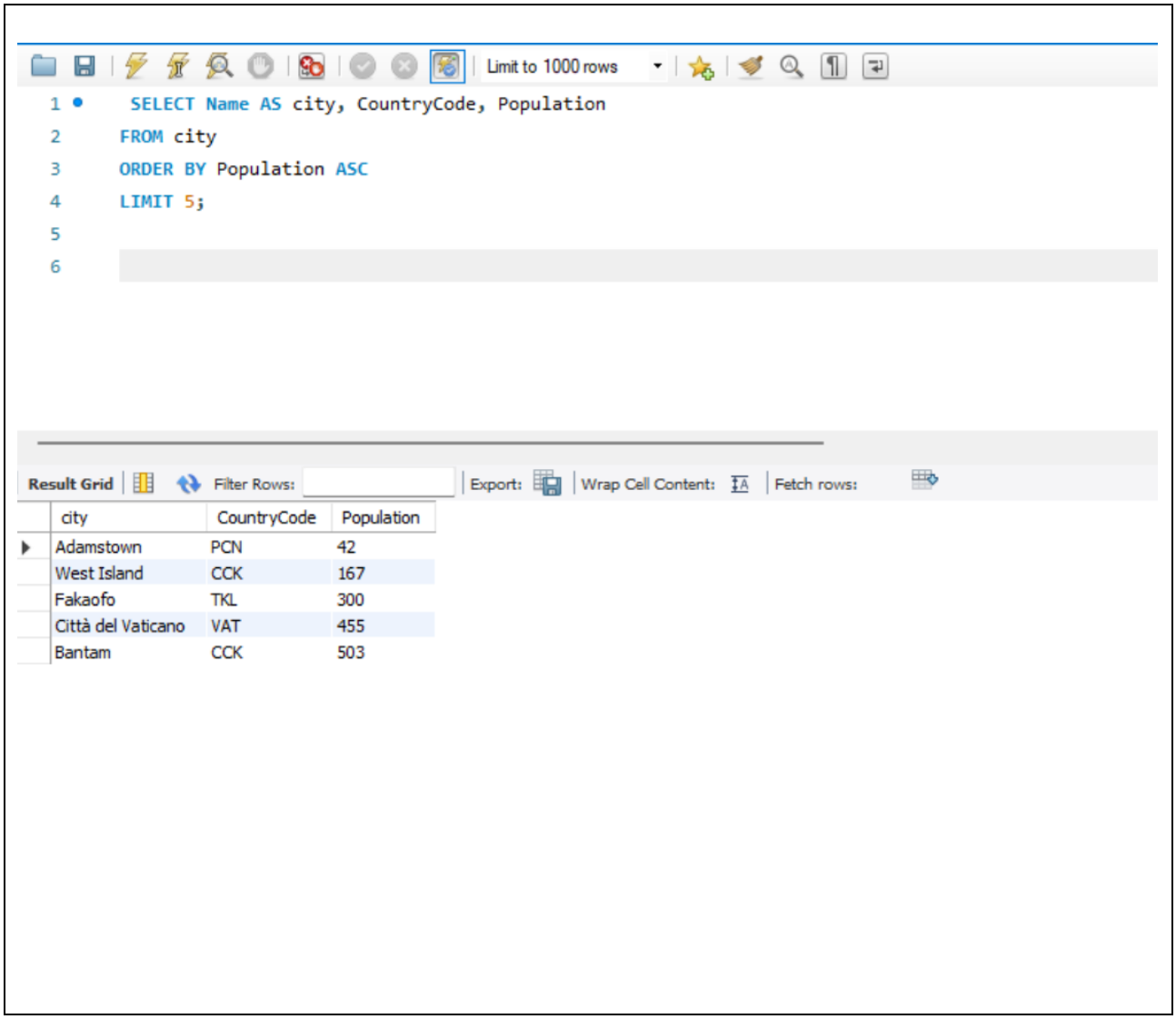


10. **City Name Frequency Analysis: Supporting Geography Education** *Scenario*: In a geography class, students are learning about the distribution of city names around the

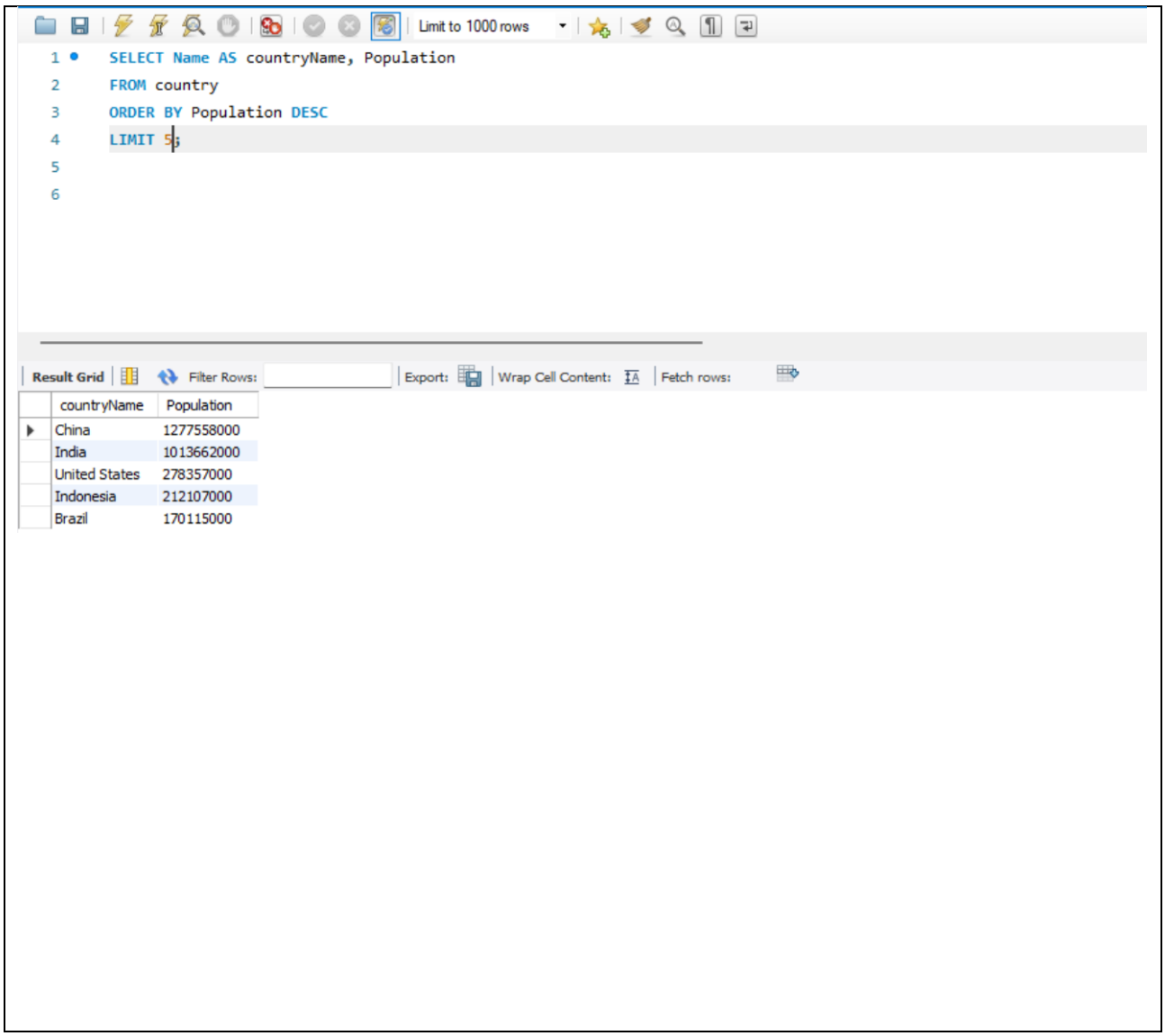
world. The teacher, in preparation for a lesson on city name frequencies, wants to provide students with a list of unique city names sorted alphabetically, along with their respective counts of occurrences in the database. You're tasked with this sorted list to support the geography teacher.



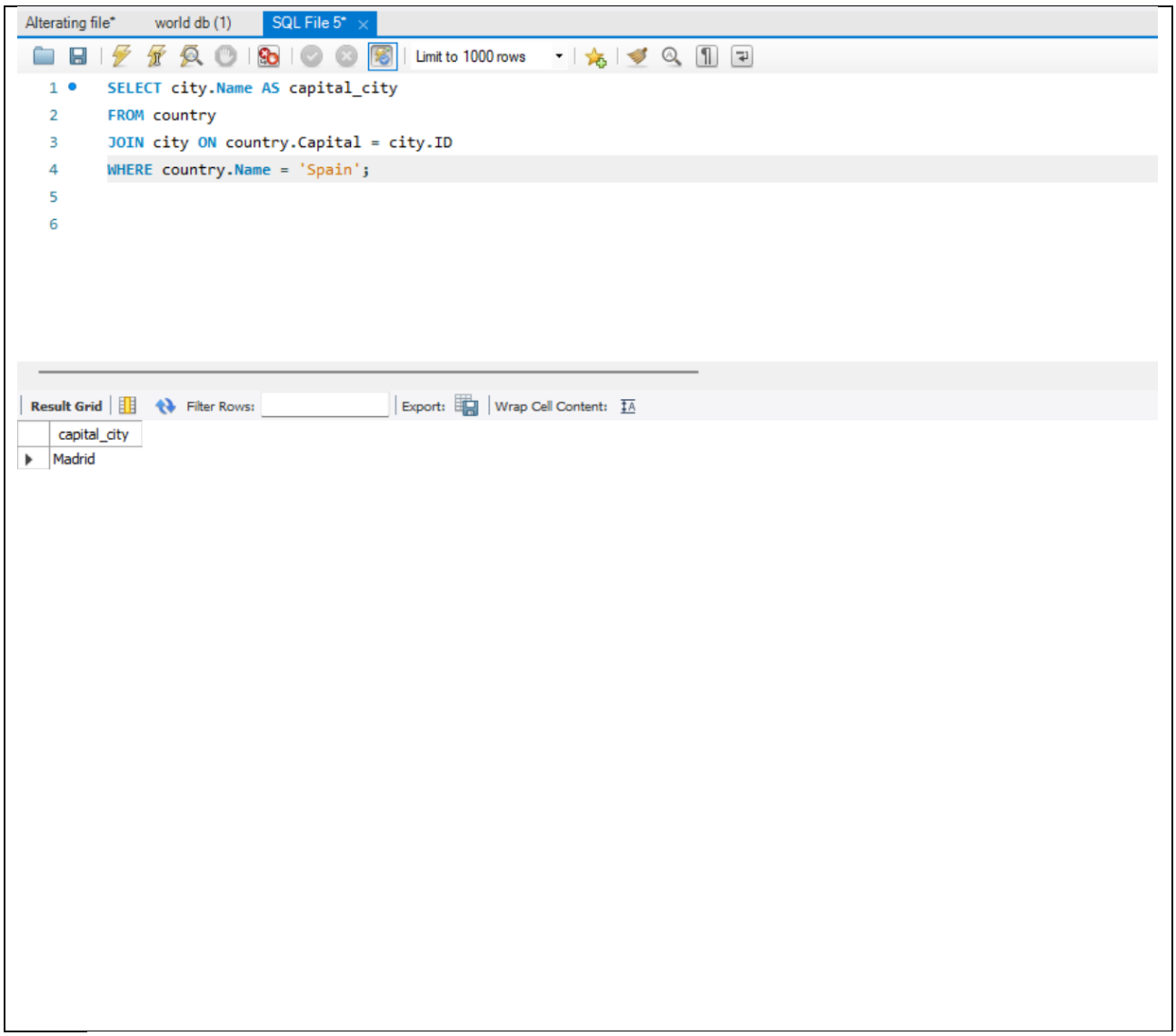
11. **City with the Lowest Population:** *Scenario:* A census bureau is conducting an analysis of urban population distribution. You're tasked with identifying the city with the lowest population from the database to provide a comprehensive overview of demographic trends.



12. **Country with Largest Population:** *Scenario:* A global economic research institute requires data on countries with the largest populations for a comprehensive analysis. You're tasked with identifying the country with the highest population from the database to provide valuable insights into demographic trends.



13. **Capital of Spain:** *Scenario:* A travel agency is organising tours across Europe and needs accurate information on capital cities. You're tasked with identifying the capital of Spain from the database to ensure itinerary accuracy and provide travellers with essential destination information.



14. **Cities in Europe:** *Scenario:* A European cultural exchange program is seeking to connect students with cities across the continent. You're tasked with compiling a list of cities located in Europe from the database to facilitate program planning and student engagement.

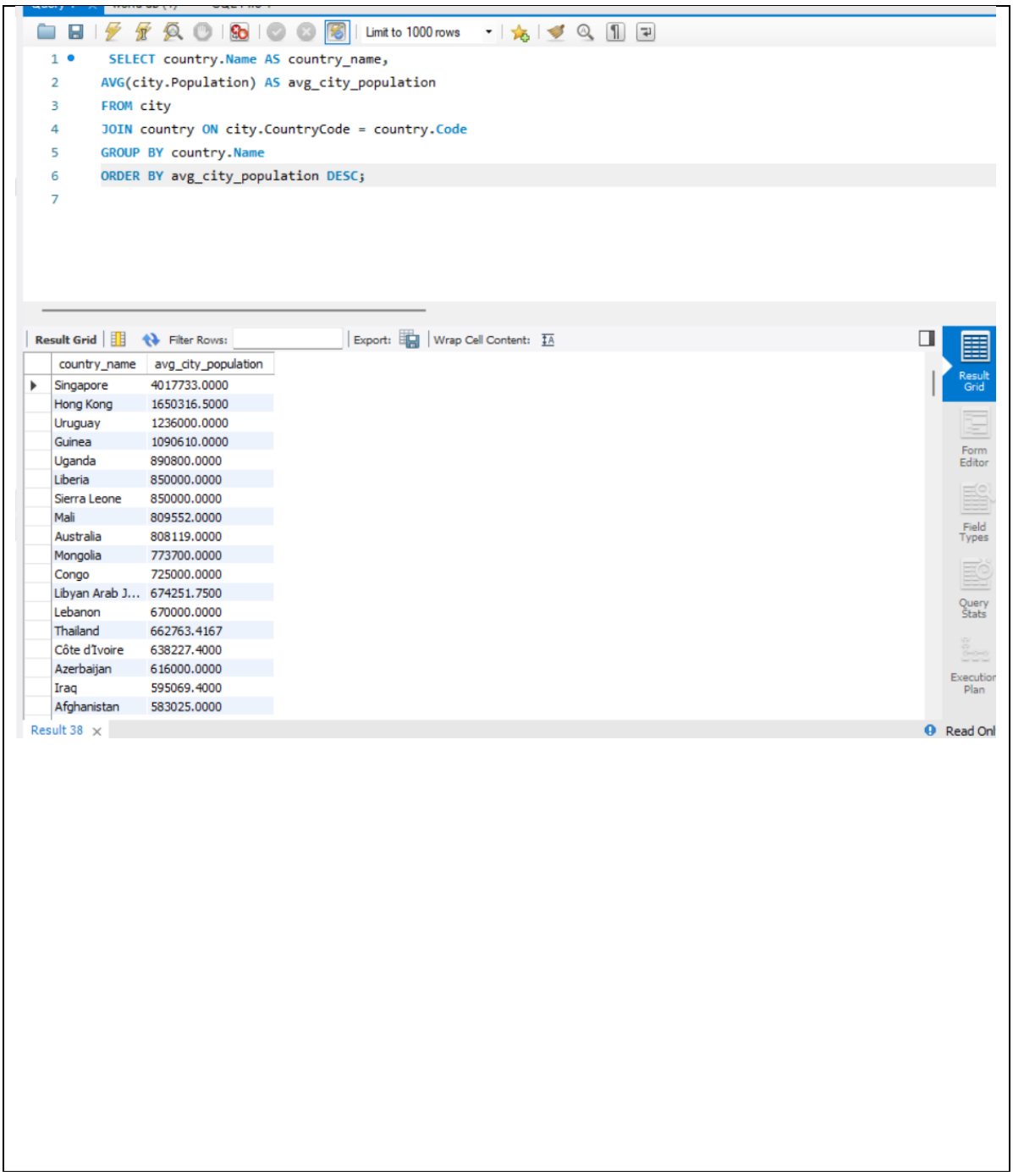
```
SELECT city.Name AS City_Name, country.name AS Country_Name
FROM city
JOIN country ON city.CountryCode = Country.Code
WHERE country.continent = 'Europe'
ORDER BY country.name, city.name;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

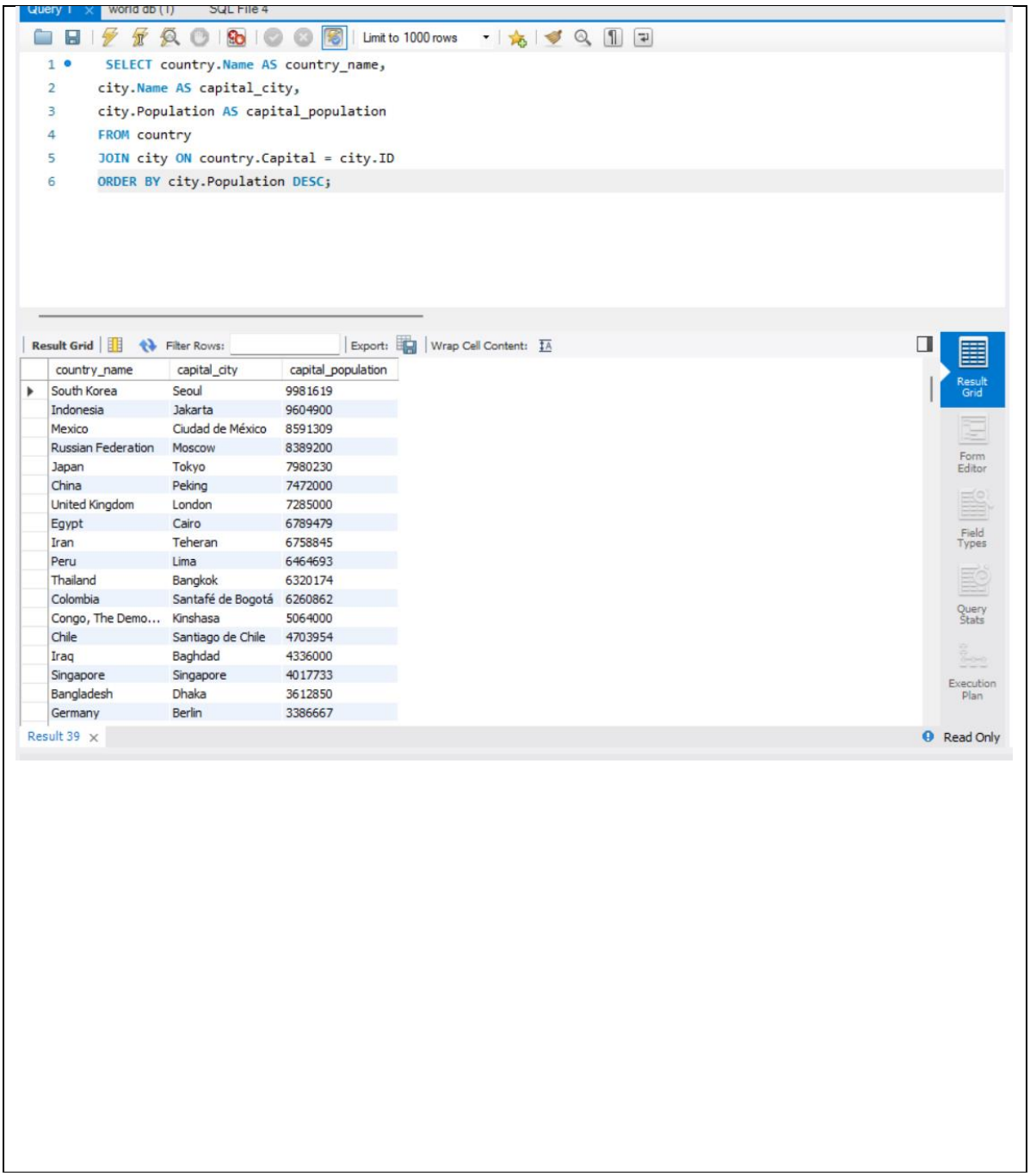
City_Name	Country_Name
Tirana	Albania
Andorra la Vella	Andorra
Graz	Austria
Innsbruck	Austria
Klagenfurt	Austria
Linz	Austria
Salzburg	Austria
Wien	Austria
Baranovitši	Belarus
Bobruisk	Belarus
Borisov	Belarus
Brest	Belarus
Gomel	Belarus
Grodno	Belarus
Lida	Belarus
Minsk	Belarus
Mogiljov	Belarus
Molodetšno	Belarus

Result 36 | Read Only

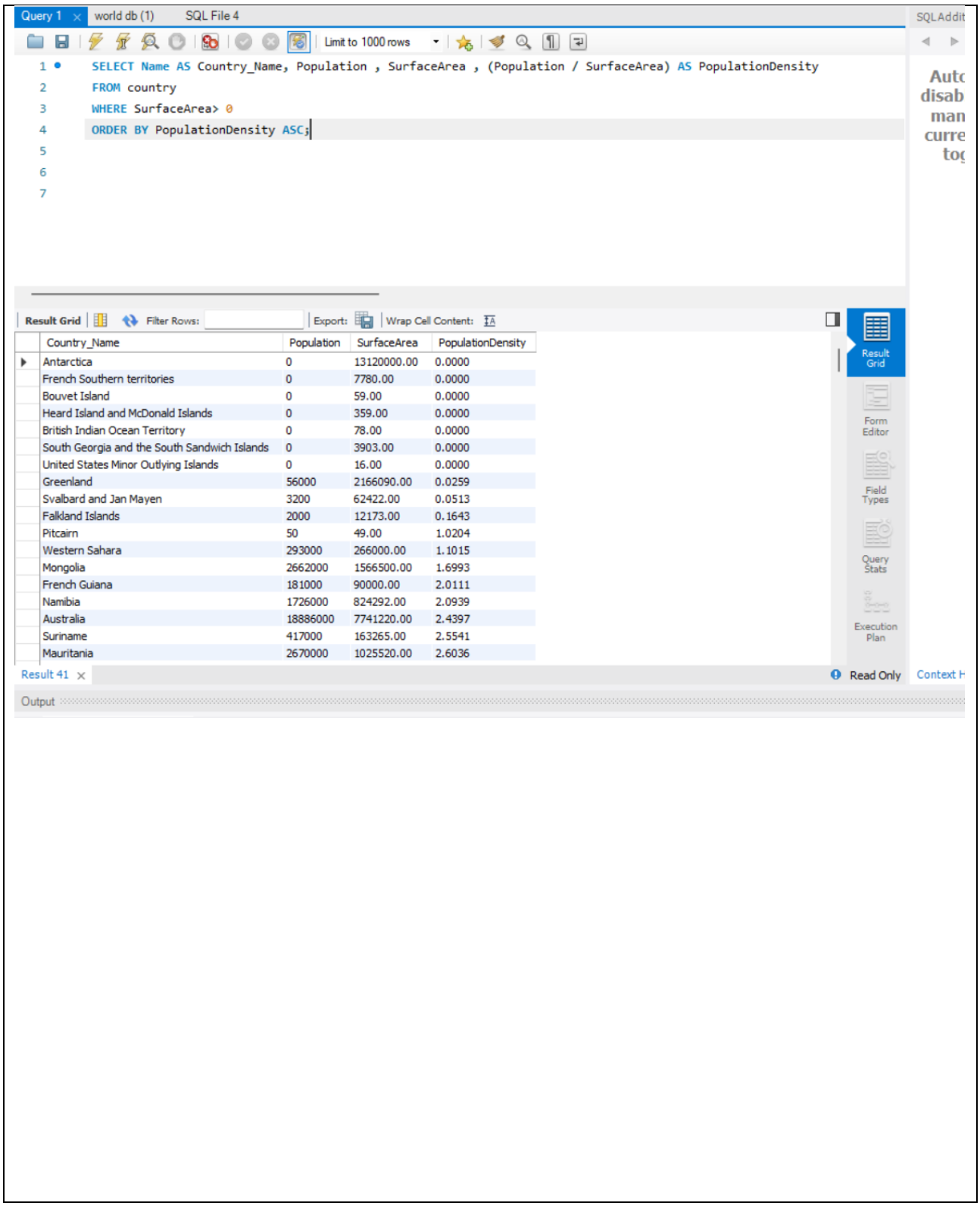
15. **Average Population by Country:** *Scenario:* A demographic research team is conducting a comparative analysis of population distributions across countries. You're tasked with calculating the average population for each country from the database to provide valuable insights into global population trends.



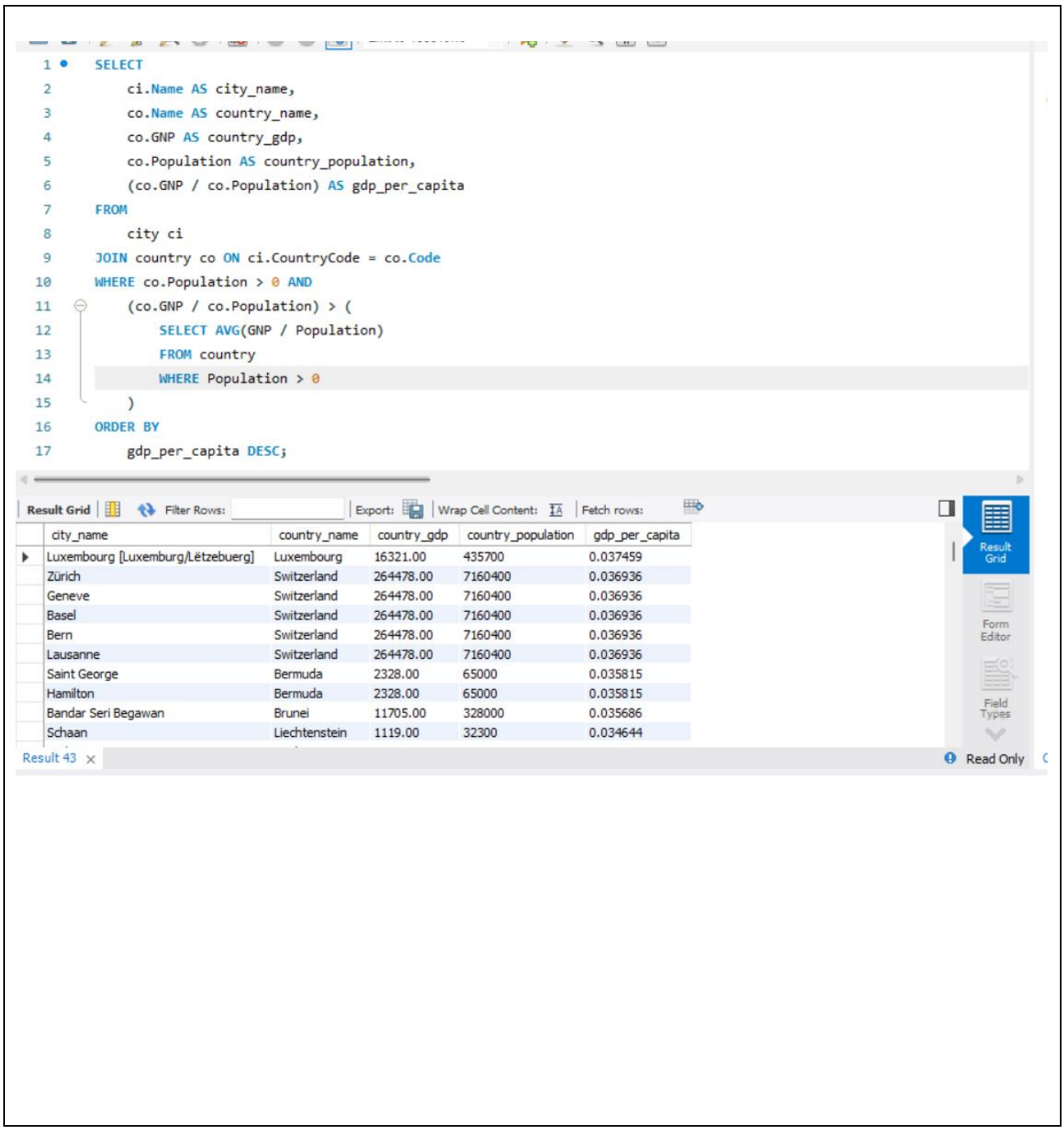
16. **Capital Cities Population Comparison:** *Scenario:* A statistical analysis firm is examining population distributions between capital cities worldwide. You're tasked with comparing the populations of capital cities from different countries to identify trends and patterns in urban demographics.



17. **Countries with Low Population Density:** *Scenario:* An agricultural research institute is studying countries with low population densities for potential agricultural development projects. You're tasked with identifying countries with sparse populations from the database to support the institute's research efforts.



18. **Cities with High GDP per Capita:** *Scenario:* An economic consulting firm is analysing cities with high GDP per capita for investment opportunities. You're tasked with identifying cities with above-average GDP per capita from the database to assist the firm in identifying potential investment destinations.



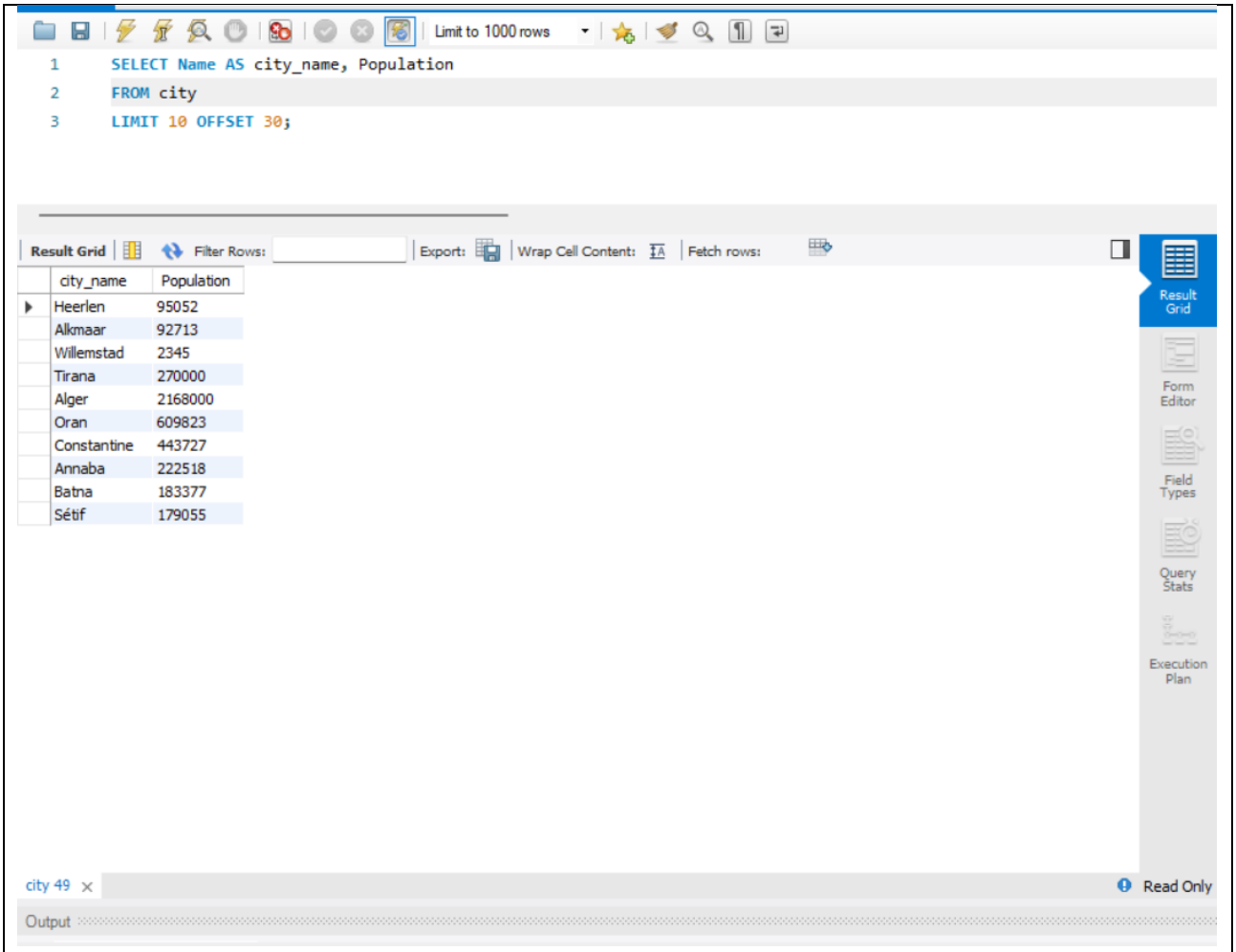
```sql
SELECT
    ci.Name AS city_name,
    co.Name AS country_name,
    co.GNP AS country_gdp,
    co.Population AS country_population,
    (co.GNP / co.Population) AS gdp_per_capita
FROM
    city ci
JOIN country co ON ci.CountryCode = co.Code
WHERE co.Population > 0 AND
    (co.GNP / co.Population) > (
        SELECT AVG(GNP / Population)
        FROM country
        WHERE Population > 0
    )
ORDER BY
    gdp_per_capita DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

city_name	country_name	country_gdp	country_population	gdp_per_capita
Luxembourg [Luxemburg/Lëtzebuerg]	Luxembourg	16321.00	435700	0.037459
Zürich	Switzerland	264478.00	7160400	0.036936
Geneve	Switzerland	264478.00	7160400	0.036936
Basel	Switzerland	264478.00	7160400	0.036936
Bern	Switzerland	264478.00	7160400	0.036936
Lausanne	Switzerland	264478.00	7160400	0.036936
Saint George	Bermuda	2328.00	65000	0.035815
Hamilton	Bermuda	2328.00	65000	0.035815
Bandar Seri Begawan	Brunei	11705.00	328000	0.035686
Schaan	Liechtenstein	1119.00	32300	0.034644

Result 43 | Read Only | Result Grid | Form Editor | Field Types

19. **Display Columns with Limit (Rows 31-40):** *Scenario:* A market research firm requires detailed information on cities beyond the top rankings for a comprehensive analysis. You're tasked with providing data on cities ranked between 31st and 40th by population to ensure a thorough understanding of urban demographics.


Limit to 1000 rows
1 SELECT Name AS city_name, Population
2 FROM city
3 LIMIT 10 OFFSET 30;
Result Grid
Filter Rows:
Export:
Wrap Cell Content:
Fetch rows:
city_name Population
Heerlen 95052
Alkmaar 92713
Willemstad 2345
Tirana 270000
Alger 2168000
Oran 609823
Constantine 443727
Annaba 222518
Batna 183377
Sétif 179055
Result Grid
Form Editor
Field Types
Query Stats
Execution Plan
city 49
Read Only
Output

Course Notes

It is recommended to take notes from the course, use the space below to do so, or use the revision guide shared with the class:

We have included a range of additional links to further resources and information that you may find useful, these can be found within your revision guide.

END OF WORKBOOK

Please check through your work thoroughly before submitting and update the table of contents if required.

Please send your completed work booklet to your trainer.