Project Overview

In this project, we focused on implementing and testing five fundamental Relational Algebra (RA) operators in a Java-based environment using ArrayList for storage. The goal was to build a foundational understanding of how these RA operators work in the context of a relational database system. The operators were implemented within the Table.java class, allowing us to simulate basic database operations without relying on a traditional DBMS. We explored the functionality and performance of these operators, ensuring that they correctly manipulate and retrieve data as expected

Key Features

- **Implementation of Relational Algebra Operators:** We implemented five key RA operators—Select, Project, Union, Set Difference, and Cartesian Product. These operators allow us to perform essential database operations such as filtering, projection, and set-based operations on the data stored in Java ArrayLists.
- **Custom Data Structures:** The project relied on Java's ArrayList as the primary data structure to store and manipulate table data. This choice allowed us to closely manage memory usage and control the complexity of operations like joins and projections.
- **Modular Code Design:** The project structure is highly modular, with each RA operator encapsulated in its method within the Table.java class. This design promotes code reuse, maintainability, and ease of testing, ensuring that each operator functions independently while integrating seamlessly with the others.
- **Predicate-based Selection:** The select operator is designed to filter rows based on a given predicate, supporting simple comparisons and logical operators. This operator mimics the SQL WHERE clause, allowing users to retrieve only the data that meets specific criteria.
- **Project:** The project operator was implemented to extract specific columns from a table, similar to the SQL SELECT clause. This operator helps in reducing the dimensionality of the data and focusing on relevant attributes for further analysis.
- **Union and Set Difference Operations:** The project includes implementations of the union and set difference operators, enabling the combination and comparison of different datasets. These operators are critical for tasks that involve merging or differentiating between two sets of tuples.
- **Cartesian Product:** The cartesianProduct operator is implemented to combine two tables, generating all possible pairs of tuples. This operation is essential for more complex join operations, although it can be resource-intensive.
- **Nested Loop Join Algorithm:** We iterate through each table's tuples using a nested loop, scanning the cartesian product for tuple combinations that have keys that pass the join's predicate.

Usage/Examples

Select Operator
To filter rows from a table using the select operator:
1. Define a predicate (e.g., `column1 > 100`).
2. Pass this predicate to the `select` method.
3. The method returns a new table containing only the rows that meet the predicate.

Project Operator
To extract specific columns using the project operator:
1. Specify the columns of interest (e.g., `column1`, `column3`).
2. Pass these column names to the `project` method.
3. The method returns a new table containing only the specified columns.

Combining Tables with Union
To combine two tables using the union operator:
1. Ensure both tables have the same schema.
2. Pass the tables to the `union` method.
3. The method returns a new table containing all unique rows from both tables.

Join Operator
To join two tables' tuples based on a condition:
1. Determine the type of join: equijoin, theta join, or natural join.
  - For equijoin: Pass the corresponding column names and the other table to join to the `join` method.
  - For theta join: Pass the condition string (e.g., `"studioName != name"`) and the other table to join.
  - For natural join: Pass the other table to join.
2. The method returns a new table containing the joined tuples.
  - For equijoin and theta join, duplicate column names are appended with `2`.
  - For natural join, duplicate columns are removed.

Performance and Testing

We conducted extensive testing of each RA operator to ensure correctness and efficiency. The operations were tested with various datasets to observe their behavior under different conditions. While the project is limited by the in-memory storage using ArrayList, it serves as a valuable learning tool for understanding the mechanics of RA operators and their implementation in a simplified environment.