

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп’ютерних систем**

**Розрахунково-графічна робота**

з дисципліни **Бази даних і засоби управління**

*на тему: “* *Створення додатку бази даних, орієнтованого на взаємодію з СУБД PostgreSQL”*

Виконав: студент ІІI курсу

ФПМ групи КВ-23

Атанов Назар

Перевірив:

Київ – 2024

[*https://github.com/NazarBello/BD\_lab.git*](https://github.com/NazarBello/BD_lab.git)

*Метою роботи* є здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

*Завдання* роботи полягає у наступному:

1. Реалізувати функції внесення, редагування та вилучення даних у таблицях бази даних, створених у лабораторній роботі №1, засобами консольного інтерфейсу.

2. Передбачити автоматичне пакетне генерування «рандомізованих» даних у базі.

3. Забезпечити реалізацію пошуку за декількома атрибутами з двох та більше сутностей одночасно: для числових атрибутів – у рамках діапазону, для рядкових – як шаблон функції LIKE оператора SELECT SQL, для логічного типу – значення True/False, для дат – у рамках діапазону дат.

4. Програмний код виконати згідно шаблону MVC (модель-подання-контролер).

**Виконання роботи**

**Логічна модель предметної області «Система бронювань подорожей»**

Логічну модель (схему бази даних) наведено на рисунку 1.

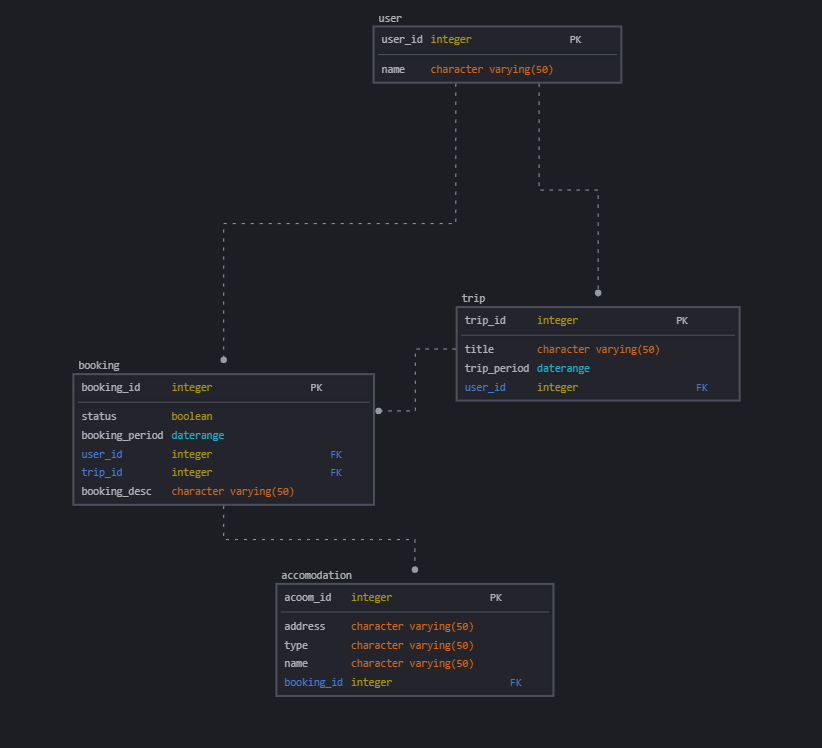


Рисунок 1 -Логічна модель бази даних

Зміни у порівнянні з першою лабораторною роботою відсутні. (інструмент: sqldbm.com)

**Середовище та компоненти розробки**

Для розробки використовувалась мова програмування Java, середовище розробки Intellij Idea Ultimate, сторонній набір бібліотек JDBC також використаний Maven для dependencies

**Шаблон проектування**

MVC - шаблон проектування, який використаний у програмі.

Model – представляє клас, що описує логіку використовуваних даних. Згідно компоненту моделі, у моїй програмі відповідають всі компоненти які знаходяться у папці Models.

View – в нашому випадку консольний інтерфейс з яким буде взаємодіяти наш користувач. Згідно компоненту представлення, то їй відповідають такі компоненти, згідно яким користувач бачить необхідні дані, що є представленням даних у вигляді консольного інтерфейсу.

Controller – представляє клас, що забезпечує зв'язок між користувачем і системою, поданням і сховищем даних. Він отримує вводяться користувачем дані і обробляє їх. І в залежності від результатів обробки відправляє користувачеві певний висновок, наприклад, у вигляді подання.

**Структура програми та її опис**



Рисунок 2 - Структура програми

Програма умовно розділена на 4 модулі: файл DataBaseController.java, файл DataBaseModel.java та файл DataBaseView.java та головний файл Main.java

Класи, як видно з їх назв, повністю відповідають використаному патерну MVC.

У файлі DataBaseModel описаний клас моделі, що займається регулюванням підключення до бази даних, та виконанням запитів до неї.

У файлі DataBaseController описаний інтерфейс взаємодії з користувачем, запит бажаної дії, виконання пошуку, тощо.

У файлі DataBaseView описаний клас, що виводить результати виконання тієї чи іншої дії на екран консолі.

**Структура меню програми**

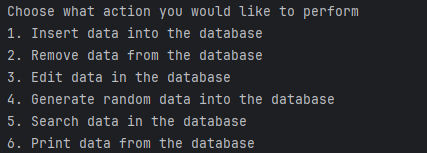


Рисунок 3 - Меню програми

Меню користувача складається з шести пунктів (Рисунок 3).

Перший пункт пропонує введення даних в таблицю

Другий пункт пропонує видалення даних з таблиці

Третій пункт пропонує оновлення даних в таблиці

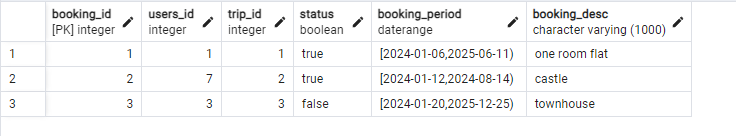
Четвертий пункт пропонує генерування даних в таблиці

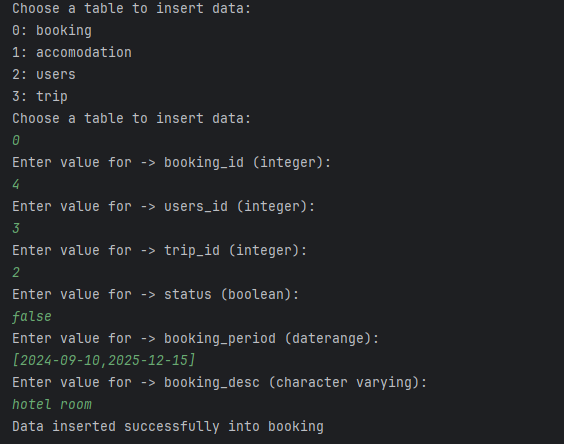
П’ятий пункт пропонує пошук даних у таблиці

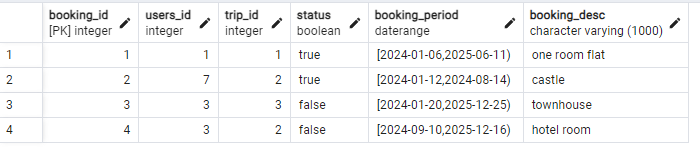
Шостий пункт подає таблиці в текстовому представленні

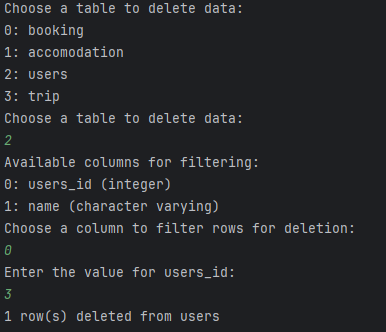
**Insert Delete Update**

Insert operation

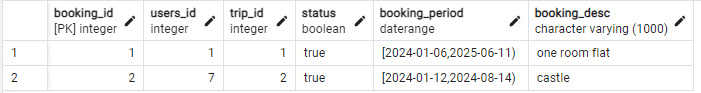




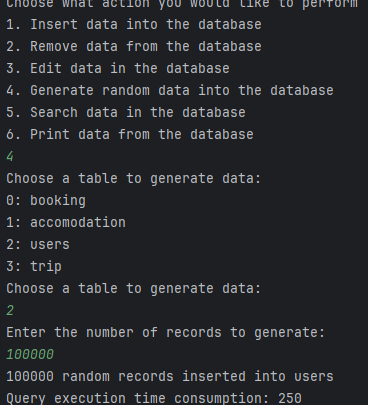




Після видалення юзера з id = 3, через наявність каскаду видаляються і елементи з таблиці booking з тим самим user\_id



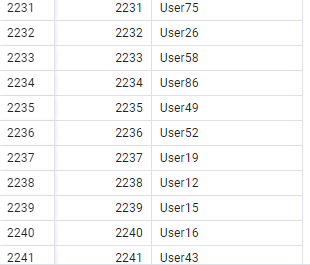
Generate

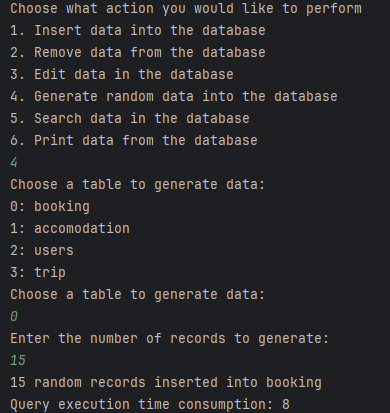


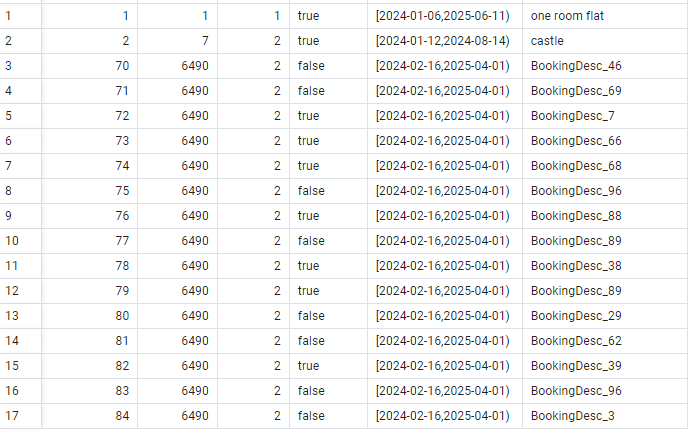
Копія запиту

"INSERT INTO users (name) " +  
 "SELECT 'User' || trunc(random() \* 100)::int FROM generate\_series(1, " + count + ")";

Скріншот з бази данних



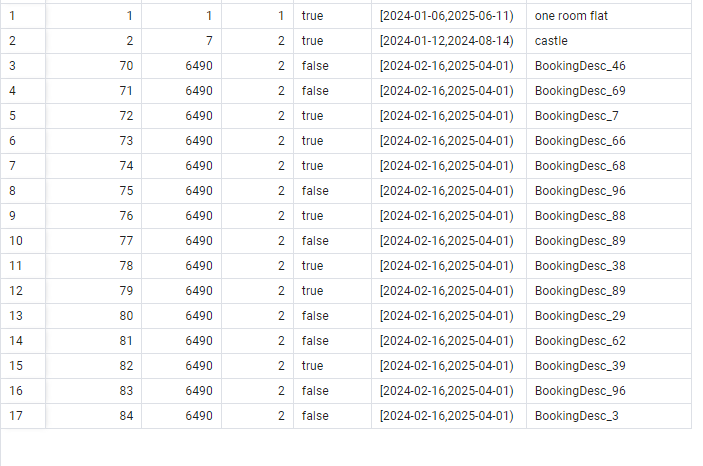


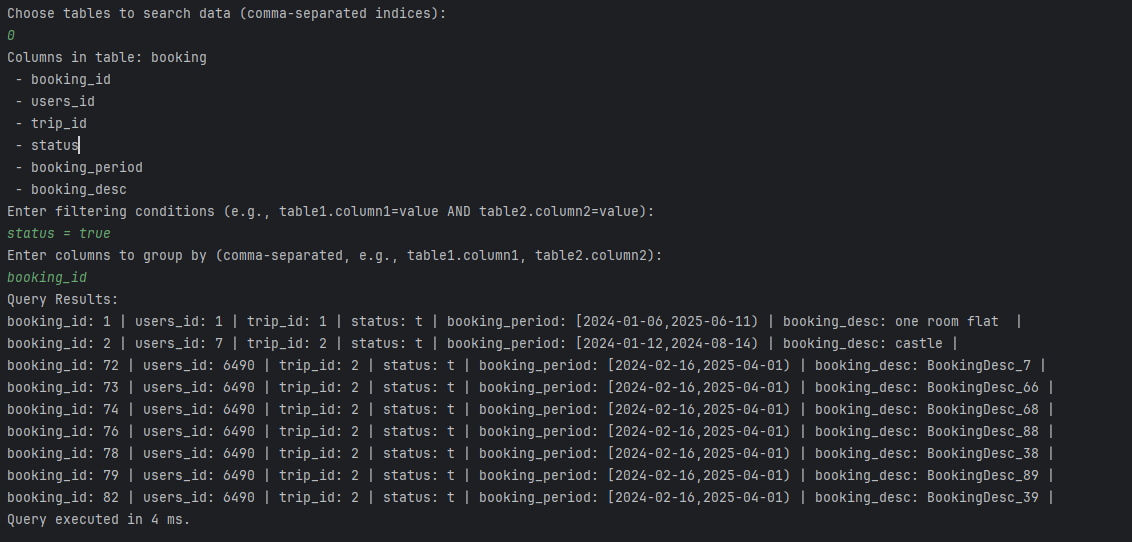


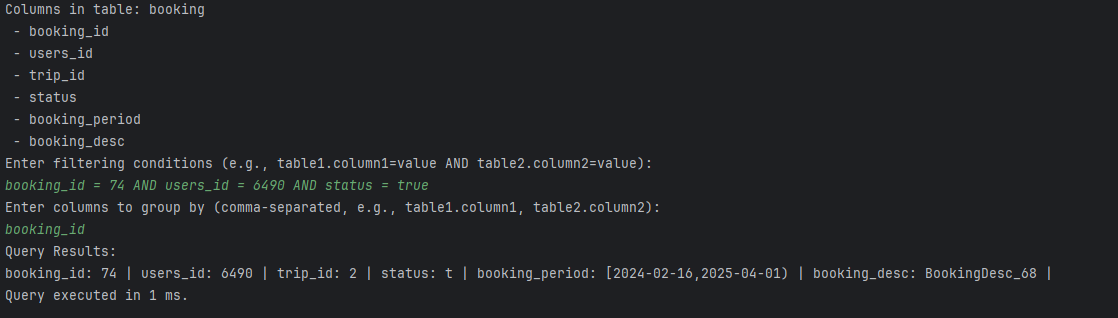
Копія SQL запиту

"INSERT INTO booking (users\_id, trip\_id, status, booking\_period, booking\_desc)\n" +  
 "SELECT\n" +  
 " u.users\_id,\n" +  
 " tr.trip\_id,\n" +  
 " (random() > 0.5) AS status,\n" +  
 " daterange(\n" +  
 " LEAST(date '2024-01-01' + d.day1, date '2024-01-01' + d.day2),\n" +  
 " GREATEST(date '2024-01-01' + d.day1, date '2024-01-01' + d.day2)\n" +  
 " ) AS booking\_period,\n" +  
 " 'BookingDesc\_' || trunc(random() \* 100)::int AS booking\_desc\n" +  
 "FROM generate\_series(1, " + count + ") g\n" +  
 "CROSS JOIN LATERAL (\n" +  
 " -- Approximately 5 years \* 365 ~ 1825 days\n" +  
 " SELECT trunc(random() \* 1825)::int AS day1,\n" +  
 " trunc(random() \* 1825)::int AS day2\n" +  
 ") d\n" +  
 "CROSS JOIN LATERAL (\n" +  
 " SELECT users\_id\n" +  
 " FROM users\n" +  
 " OFFSET floor(random() \* (SELECT count(\*) FROM users))\n" +  
 " LIMIT 1\n" +  
 ") u\n" +  
 "CROSS JOIN LATERAL (\n" +  
 " SELECT trip\_id\n" +  
 " FROM trip\n" +  
 " OFFSET floor(random() \* (SELECT count(\*) FROM trip))\n" +  
 " LIMIT 1\n" +  
 ") tr;";

**Search**

****

****

****

**Опис модуля «Model»**

**Програмний код модуля**

package org.example;  
  
import java.sql.\*;  
import java.util.\*;  
import java.util.regex.Pattern;  
  
public class DataBaseModel {  
 private final Connection connection;  
  
 public DataBaseModel(String jdbcUrl) throws SQLException {  
 connection = DriverManager.*getConnection*(jdbcUrl);  
 System.*out*.println("Connection established");  
 }  
  
 */\*\*  
 \* Insert data into a selected table with validation for data types and foreign keys.  
 \*/* public void insert() throws SQLException {  
 Scanner scanner = new Scanner(System.*in*);  
 List<String> tables = getTables("insert");  
  
 System.*out*.println("Choose a table to insert data:");  
 int tableIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String selectedTable = tables.get(tableIndex);  
 List<String> columns = getColumns(selectedTable);  
 List<String> columnTypes = getColumnTypes(selectedTable);  
  
 List<String> values = new ArrayList<>();  
  
 for (int i = 0; i < columns.size(); i++) {  
 String column = columns.get(i);  
 String columnType = columnTypes.get(i);  
 System.*out*.println("Enter value for -> " + column + " (" + columnType + "):");  
  
 String value = scanner.nextLine();  
  
 if (isForeignKey(column, selectedTable)) {  
 // Check if the foreign key value exists in the parent table  
 if (!validateForeignKey(column, Integer.*parseInt*(value))) {  
 System.*out*.println("Invalid foreign key value for column: " + column);  
 return;  
 }  
 }  
  
 if (columnType.equals("character varying")) {  
 value = "'" + value.replace("'", "''") + "'";  
 } else if (columnType.equals("daterange")) {  
 // Validate and format daterange  
 if (!validateDateRange(value)) {  
 System.*out*.println("Invalid date range format. Use [YYYY-MM-DD,YYYY-MM-DD].");  
 return;  
 }  
 value = "'" + value + "'";  
 }  
  
 values.add(value);  
 }  
  
 String query = "INSERT INTO " + selectedTable + " (" + String.*join*(", ", columns) + ") VALUES (" + String.*join*(", ", values) + ")";  
 executeUpdate(query, "Data inserted successfully into " + selectedTable);  
 }  
  
 */\*\*  
 \* Generate random data using SQL queries for selected table.  
 \*/* public void generate() throws SQLException {  
 Scanner scanner = new Scanner(System.*in*);  
 List<String> tables = getTables("generate");  
  
 System.*out*.println("Choose a table to generate data:");  
 int tableIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 System.*out*.println("Enter the number of records to generate:");  
 int recordCount = scanner.nextInt();  
 long time\_start = System.*nanoTime*();  
 String selectedTable = tables.get(tableIndex);  
 String query = generateSQLForGeneration(selectedTable, recordCount);  
  
 executeUpdate(query, recordCount + " random records inserted into " + selectedTable);  
 long time\_end = System.*nanoTime*();  
 long result = (time\_end - time\_start)/ 1\_000\_000;  
 System.*out*.println("Query execution time consumption: " + result);  
  
 }  
  
  
 public void search() throws SQLException {  
 Scanner scanner = new Scanner(System.*in*);  
 List<String> tables = getTables("search");  
  
 // Step 1: Select tables  
 System.*out*.println("Choose tables to search data (comma-separated indices):");  
 String[] selectedIndices = scanner.nextLine().split(",");  
 List<String> selectedTables = new ArrayList<>();  
 for (String index : selectedIndices) {  
 selectedTables.add(tables.get(Integer.*parseInt*(index.trim())));  
 }  
  
 // Step 2: Display columns for each selected table  
 Map<String, List<String>> tableColumns = new HashMap<>();  
 for (String table : selectedTables) {  
 List<String> columns = getColumns(table);  
 tableColumns.put(table, columns);  
 System.*out*.println("Columns in table: " + table);  
 for (String column : columns) {  
 System.*out*.println(" - " + column);  
 }  
 }  
  
 // Step 3: Input filtering conditions  
 System.*out*.println("Enter filtering conditions (e.g., table1.column1=value AND table2.column2=value):");  
 String filters = scanner.nextLine();  
  
 // Step 4: Input grouping columns  
 System.*out*.println("Enter columns to group by (comma-separated, e.g., table1.column1, table2.column2):");  
 String groupBy = scanner.nextLine();  
  
 // Step 5: Construct the query  
 StringBuilder query = new StringBuilder("SELECT ");  
 for (String table : selectedTables) {  
 for (String column : tableColumns.get(table)) {  
 query.append(table).append(".").append(column).append(", ");  
 }  
 }  
 query.setLength(query.length() - 2); // Remove the last comma and space  
 query.append(" FROM ").append(String.*join*(", ", selectedTables));  
 if (!filters.isEmpty()) {  
 query.append(" WHERE ").append(filters);  
 }  
 if (!groupBy.isEmpty()) {  
 query.append(" GROUP BY ").append(groupBy);  
 }  
  
 // Step 6: Measure query execution time  
 long startTime = System.*currentTimeMillis*();  
 try (PreparedStatement preparedStatement = connection.prepareStatement(query.toString());  
 ResultSet resultSet = preparedStatement.executeQuery()) {  
  
 // Step 7: Display results  
 ResultSetMetaData metaData = resultSet.getMetaData();  
 int columnCount = metaData.getColumnCount();  
  
 System.*out*.println("Query Results:");  
 while (resultSet.next()) {  
 for (int i = 1; i <= columnCount; i++) {  
 System.*out*.print(metaData.getColumnName(i) + ": " + resultSet.getString(i) + " | ");  
 }  
 System.*out*.println();  
 }  
 }  
 long endTime = System.*currentTimeMillis*();  
  
 // Step 8: Output execution time  
 System.*out*.println("Query executed in " + (endTime - startTime) + " ms.");  
 }  
  
  
 public void delete() throws SQLException {  
 Scanner scanner = new Scanner(System.*in*);  
 List<String> tables = getTables("delete");  
  
 System.*out*.println("Choose a table to delete data:");  
 int tableIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String selectedTable = tables.get(tableIndex);  
 List<String> columns = getColumns(selectedTable);  
 List<String> columnTypes = getColumnTypes(selectedTable);  
  
 System.*out*.println("Available columns for filtering:");  
 for (int i = 0; i < columns.size(); i++) {  
 System.*out*.println(i + ": " + columns.get(i) + " (" + columnTypes.get(i) + ")");  
 }  
  
 System.*out*.println("Choose a column to filter rows for deletion:");  
 int columnIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String selectedColumn = columns.get(columnIndex);  
 String columnType = columnTypes.get(columnIndex);  
  
 System.*out*.println("Enter the value for " + selectedColumn + ":");  
 String filterValue = scanner.nextLine();  
  
 String query = "DELETE FROM " + selectedTable + " WHERE " + selectedColumn + " = ?";  
 try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {  
 // Handle type-specific parsing  
 if (columnType.equals("integer")) {  
 preparedStatement.setInt(1, Integer.*parseInt*(filterValue));  
 } else if (columnType.equals("boolean")) {  
 preparedStatement.setBoolean(1, Boolean.*parseBoolean*(filterValue));  
 } else if (columnType.equals("daterange")) {  
 preparedStatement.setObject(1, filterValue, java.sql.Types.*OTHER*); // Daterange in PostgreSQL  
 } else {  
 preparedStatement.setString(1, filterValue); // Default for text or character types  
 }  
  
 int rowsAffected = preparedStatement.executeUpdate();  
  
 if (rowsAffected > 0) {  
 System.*out*.println(rowsAffected + " row(s) deleted from " + selectedTable);  
 } else {  
 System.*out*.println("No rows matched the filter.");  
 }  
 } catch (SQLException e) {  
 System.*err*.println("Error executing delete: " + e.getMessage());  
 throw e;  
 } catch (NumberFormatException e) {  
 System.*err*.println("Invalid input type: " + e.getMessage());  
 }  
 }  
  
  
  
 public void update() throws SQLException {  
 Scanner scanner = new Scanner(System.*in*);  
 List<String> tables = getTables("update");  
  
 System.*out*.println("Choose a table to update data:");  
 int tableIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String selectedTable = tables.get(tableIndex);  
 List<String> columns = getColumns(selectedTable);  
  
 System.*out*.println("Choose a column to filter rows for updating:");  
 for (int i = 0; i < columns.size(); i++) {  
 System.*out*.println(i + ": " + columns.get(i));  
 }  
  
 int columnIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String filterColumn = columns.get(columnIndex);  
 System.*out*.println("Enter the value for " + filterColumn + ":");  
 String filterValue = scanner.nextLine();  
  
 System.*out*.println("Enter the column to update:");  
 int updateColumnIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String updateColumn = columns.get(updateColumnIndex);  
 System.*out*.println("Enter the new value for " + updateColumn + ":");  
 String newValue = scanner.nextLine();  
  
 String query = "UPDATE " + selectedTable + " SET " + updateColumn + " = ? WHERE " + filterColumn + " = ?";  
 try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {  
 preparedStatement.setString(1, newValue);  
 preparedStatement.setString(2, filterValue);  
 int rowsAffected = preparedStatement.executeUpdate();  
  
 if (rowsAffected > 0) {  
 System.*out*.println(rowsAffected + " row(s) updated in " + selectedTable);  
 } else {  
 System.*out*.println("No rows matched the filter.");  
 }  
 } catch (SQLException e) {  
 System.*err*.println("Error executing update: " + e.getMessage());  
 throw e;  
 }  
 }  
  
  
 public void print() throws SQLException {  
 Scanner scanner = new Scanner(System.*in*);  
 List<String> tables = getTables("print");  
  
 System.*out*.println("Choose a table to print data:");  
 int tableIndex = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 String selectedTable = tables.get(tableIndex);  
 String query = "SELECT \* FROM " + selectedTable;  
  
 try (PreparedStatement preparedStatement = connection.prepareStatement(query);  
 ResultSet resultSet = preparedStatement.executeQuery()) {  
  
 ResultSetMetaData metaData = resultSet.getMetaData();  
 int columnCount = metaData.getColumnCount();  
  
 // Print header  
 System.*out*.println("Printing data from: " + selectedTable);  
 for (int i = 1; i <= columnCount; i++) {  
 System.*out*.printf("%-20s", metaData.getColumnName(i)); // Fixed-width formatting  
 }  
 System.*out*.println("\n" + "-".repeat(columnCount \* 20));  
  
 // Print rows  
 while (resultSet.next()) {  
 for (int i = 1; i <= columnCount; i++) {  
 System.*out*.printf("%-20s", resultSet.getString(i)); // Fixed-width formatting  
 }  
 System.*out*.println();  
 }  
 } catch (SQLException e) {  
 System.*err*.println("Error retrieving data from " + selectedTable + ": " + e.getMessage());  
 throw e;  
 }  
 }  
 */\*\*  
 \* Execute an update/insert/delete query.  
 \*/* private void executeUpdate(String query, String successMessage) throws SQLException {  
 try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {  
 preparedStatement.executeUpdate();  
 System.*out*.println(successMessage);  
 } catch (SQLException e) {  
 System.*err*.println("Error executing query: " + e.getMessage());  
 throw e;  
 }  
 }  
  
 */\*\*  
 \* Generate SQL query for generation using PostgreSQL functions.  
 \*/* private String generateSQLForGeneration(String table, int count) {  
  
 switch (table){  
 case "users": return "INSERT INTO users (name) " +  
 "SELECT 'User' || trunc(random() \* 100)::int FROM generate\_series(1, " + count + ")";  
  
 case "booking":  
 return "INSERT INTO booking (users\_id, trip\_id, status, booking\_period, booking\_desc)\n" +  
 "SELECT\n" +  
 " u.users\_id,\n" +  
 " tr.trip\_id,\n" +  
 " (random() > 0.5) AS status,\n" +  
 " daterange(\n" +  
 " LEAST(date '2024-01-01' + d.day1, date '2024-01-01' + d.day2),\n" +  
 " GREATEST(date '2024-01-01' + d.day1, date '2024-01-01' + d.day2)\n" +  
 " ) AS booking\_period,\n" +  
 " 'BookingDesc\_' || trunc(random() \* 100)::int AS booking\_desc\n" +  
 "FROM generate\_series(1, " + count + ") g\n" +  
 "CROSS JOIN LATERAL (\n" +  
 " -- Approximately 5 years \* 365 ~ 1825 days\n" +  
 " SELECT trunc(random() \* 1825)::int AS day1,\n" +  
 " trunc(random() \* 1825)::int AS day2\n" +  
 ") d\n" +  
 "CROSS JOIN LATERAL (\n" +  
 " SELECT users\_id\n" +  
 " FROM users\n" +  
 " OFFSET floor(random() \* (SELECT count(\*) FROM users))\n" +  
 " LIMIT 1\n" +  
 ") u\n" +  
 "CROSS JOIN LATERAL (\n" +  
 " SELECT trip\_id\n" +  
 " FROM trip\n" +  
 " OFFSET floor(random() \* (SELECT count(\*) FROM trip))\n" +  
 " LIMIT 1\n" +  
 ") tr;";  
 }  
 // Add cases for other tables as needed  
 return "";  
 }  
  
 */\*\*  
 \* Validate if a column is a foreign key.  
 \*/* private boolean isForeignKey(String column, String table) {  
 // Identify foreign keys based on naming conventions  
 return column.endsWith("\_id") && !column.equals(table + "\_id");  
 }  
  
  
 */\*\*  
 \* Validate a foreign key exists in the referenced table.  
 \*/* private boolean validateForeignKey(String column, int value) throws SQLException {  
 // Correctly infer the referenced table name  
 String referencedTable;  
 if (column.equals("users\_id")) {  
 referencedTable = "users";  
 } else if (column.equals("trip\_id")) {  
 referencedTable = "trip";  
 } else {  
 // Generic fallback for other foreign keys  
 referencedTable = column.replace("\_id", "");  
 }  
  
 String query = "SELECT 1 FROM " + referencedTable + " WHERE " + column + " = ?";  
 try (PreparedStatement statement = connection.prepareStatement(query)) {  
 statement.setInt(1, value);  
 try (ResultSet resultSet = statement.executeQuery()) {  
 return resultSet.next();  
 }  
 }  
 }  
  
  
 */\*\*  
 \* Validate date range format.  
 \*/* private boolean validateDateRange(String value) {  
 return Pattern.*matches*("\\[[0-9]{4}-[0-9]{2}-[0-9]{2},[0-9]{4}-[0-9]{2}-[0-9]{2}\\]", value);  
 }  
  
 */\*\*  
 \* Get table names from the database schema.  
 \*/* public List<String> getTables(String operation) throws SQLException {  
 List<String> tables = new ArrayList<>();  
 String query = "SELECT table\_name FROM information\_schema.tables WHERE table\_schema = 'public' AND table\_type = 'BASE TABLE'";  
  
 try (PreparedStatement statement = connection.prepareStatement(query);  
 ResultSet resultSet = statement.executeQuery()) {  
  
 System.*out*.println("Choose a table to " + operation + " data:");  
 int index = 0;  
 while (resultSet.next()) {  
 String tableName = resultSet.getString("table\_name");  
 tables.add(tableName);  
 System.*out*.println(index + ": " + tableName);  
 index++;  
 }  
  
 if (tables.isEmpty()) {  
 System.*out*.println("No tables found in the database.");  
 }  
 } catch (SQLException e) {  
 System.*err*.println("Error fetching tables: " + e.getMessage());  
 throw e;  
 }  
 return tables;  
 }  
  
  
 */\*\*  
 \* Get column names for a given table.  
 \*/* public List<String> getColumns(String table) throws SQLException {  
 List<String> columns = new ArrayList<>();  
 String query = "SELECT column\_name FROM information\_schema.columns WHERE table\_name = ?";  
 try (PreparedStatement statement = connection.prepareStatement(query)) {  
 statement.setString(1, table);  
 try (ResultSet resultSet = statement.executeQuery()) {  
 while (resultSet.next()) {  
 columns.add(resultSet.getString("column\_name"));  
 }  
 }  
 }  
 return columns;  
 }  
  
 */\*\*  
 \* Get column types for a given table.  
 \*/* public List<String> getColumnTypes(String table) throws SQLException {  
 List<String> columnTypes = new ArrayList<>();  
 String query = "SELECT data\_type FROM information\_schema.columns WHERE table\_name = ?";  
 try (PreparedStatement statement = connection.prepareStatement(query)) {  
 statement.setString(1, table);  
 try (ResultSet resultSet = statement.executeQuery()) {  
 while (resultSet.next()) {  
 columnTypes.add(resultSet.getString("data\_type"));  
 }  
 }  
 }  
 return columnTypes;  
 }  
}

**Опис:**

1. **insert**:
   * Дозволяє вставляти дані у вибрану таблицю.
   * Перевіряє відповідність введених значень типу даних колонок.
2. **generate**:
   * Автоматично генерує випадкові дані для заданої таблиці( booking or users)
   * Використовує SQL-запити для генерації значень із дотриманням відповідності типів даних (наприклад, daterange).
   * Підраховує час виконання запиту.
3. **search**:
   * Реалізує пошук із фільтрацією (WHERE) і групуванням (GROUP BY) даних з кількох таблиць.
   * Дозволяє задавати умови фільтрації і групування з клавіатури.
   * Виводить результати запиту разом із часом його виконання.
4. **delete**:
   * Видаляє рядки з таблиці відповідно до умов фільтрації.
   * Перевіряє типи даних для коректного виконання запиту.
5. **update**:
   * Оновлює дані в таблиці за вказаними умовами.
   * Підтримує введення нових значень для оновлення рядків.
6. **print**:
   * Виводить усі дані з вибраної таблиці у форматованій таблиці.
   * Використовує фіксовану ширину колонок для читабельності.
7. **getTables**:
   * Отримує список усіх доступних таблиць у базі даних.
8. **getColumns**:
   * Отримує список усіх колонок вибраної таблиці.
9. **getColumnTypes**:
   * Отримує типи даних для кожної колонки вибраної таблиці.
10. **Валідація:**
    * **isForeignKey**: Визначає, чи є колонка зовнішнім ключем.
    * **validateForeignKey**: Перевіряє існування значення зовнішнього ключа у відповідній таблиці.
    * **validateDateRange**: Валідує формат daterange.
11. **executeUpdate**:
    * Виконує запити типу INSERT, UPDATE, DELETE.