МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации

**

**ПРАКТИЧЕСКАЯ РАБОТА №1**

**«**Разработка СУБД**»**

**по дисциплине: «*Программирование*»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБ-320», «АВТФ»  *Мартынова Е.Д.*  «20» октября 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Заместитель декана «АВТФ»*  *Архипова А.Б.*  «\_\_\_» \_\_\_\_\_\_ 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2024

**Задание к работе**: Требуется реализовать реляционную СУБД с поддержкой запросов на языке SQL. СУБД использует единственный тип данных - строка. Требуется поддержать следующие SQL выражения

1. SELECT <> FROM <> - выборка

2. WHERE и операторы OR , AND - фильтрация

3. INSERT INTO - вставка данных в таблицы

4. DELETE FROM - удаление данных из таблицы

**Описание функций**

WriteUtf8BOM(ofstream& file) - Записывает BOM для UTF-8 в указанный файл.

FreeTable(RowNode table)\* - Освобождает память, занятую строками и ячейками таблицы.

FreeAllTables(RowNode tables, int count)\*\* - Освобождает память для всех указанных таблиц.

UpdatePrimaryKey(const string& schemaName, const string& tableName, RowNode table)\* - Обновляет последовательность первичного ключа таблицы, записывая количество строк в файл.

countRowsInCSV(const string& csvFilePath) - Считывает и возвращает количество строк в указанном CSV файле.

ReadConfiguration(const string& filename) - Читает конфигурацию из файла и создает директории и CSV файлы для каждой таблицы.

PrintTable(RowNode table) - Выводит содержимое одной таблицы в консоль.

PrintTables(RowNode tables) - Выводит содержимое нескольких таблиц в консоль.

AddTableNames(RowNode table, RowNode\* tables, int countTables) - Добавляет названия таблиц в ячейки строк указанной таблицы.

AddNumColumns(RowNode table, int numColumns[], int countTables)\* - Добавляет номера колонок в ячейки строк указанной таблицы.

InsertInto(RowNode table, const string listString[])\* - Вставляет новую строку в указанную таблицу, используя данные из массива строк.

SelectFromOneTable(RowNode table, int numColumns[]) - Выбирает определенные колонки из одной таблицы и возвращает новую таблицу с выбранными данными.

SelectFromManyTables(RowNode tables, int numColumns[], int countTables) - Выбирает определенные колонки из нескольких таблиц и возвращает новую таблицу с объединенными данными.

SelectFromCartesian(RowNode tables, int countTables) - Выполняет декартово произведение всех указанных таблиц, возвращая новую таблицу.

SplitCondition(const string& filter) - Разделяет условие фильтрации на отдельные логические выражения и возвращает список условий.

CheckingCondition(RowNode row, const string& condition) - Проверяет, соответствует ли строка заданному условию.

CheckingLogicalExpression(RowNode row, Condition condition)\*\* - Оценивает логическое выражение на основе условий, связанных логическими операторами.

FilteringTable(RowNode select, RowNode where, int selectSize, int whereSize, int numColumnsSelect[], int numColumnsWhere[], string condition)\*\* - Фильтрует строки из таблиц на основе заданных условий и возвращает новую таблицу.

DeleteFrom(RowNode table, Condition condition) - Удаляет строки из таблицы, соответствующие заданному условию.

void CreateTable(const string& schemaDir, const string& tableName) - Создает директорию и необходимые файлы для новой таблицы в указанной схеме, включая CSV файл и файлы для первичного ключа и блокировки.

json ConvertTableToJson(RowNode table)\* - Преобразует данные таблицы в формате RowNode в JSON массив, представляющий строки и ячейки таблицы.

RowNode ConvertJsonToTable(const json& jsonData, const string& tableName)\* - Преобразует JSON данные в структуру таблицы RowNode, добавляя строки в таблицу, если они существуют в JSON.

void WriteJsonToCSV(const string& filePath, const json& jsonData) - Записывает данные из JSON массива в CSV файл, добавляя BOM для поддержки UTF-8, если файл пустой.

void splitString(string& str, string tokens)\* - Разделяет строку на токены по запятой, удаляя фигурные скобки и пробелы.

json ReadCSVToJson(const string& filePath) - Читает данные из указанного CSV файла и преобразует их в JSON формат, возвращая массив строк.

void AddColumnsInSchemaJson(const json& newColumns, const string& schemaFilePath, const string& tableName) - Добавляет новые колонки в схему таблицы в JSON формате, обновляя файл схемы.

void RewriteTableSchema(const json& newSchema, const string& schemaFilePath, const string& tableName) - Перезаписывает схему указанной таблицы в файле схемы на новую структуру, переданную в виде JSON.

void RewriteCSVbyJson(const string& filePath, const json& jsonData) - Перезаписывает CSV файл, основываясь на данных из JSON массива, добавляя BOM для поддержки UTF-8.

void incrementSequence(const string& tableName) - Увеличивает значение первичного ключа в соответствующем текстовом файле, хранящемся в директории таблицы.

void decrementLock(const string& tableName) - Уменьшает значение блокировки в соответствующем текстовом файле, хранящемся в директории таблицы.

int getNextCsv(const string& tableName) - Определяет номер следующего CSV файла для таблицы, основываясь на текущем количестве строк и существующих файлах в директории.

int main() – Вызывает все функции в программе, представляя реализованные команды СУБД.

**Подключенные библиотеки**

#include <iostream> - Библиотека для работы с вводом и выводом данных через стандартные потоки (cin, cout).

#include <string> - Библиотека для работы со строками и их манипуляциями. #include <algorithm> - Библиотека с функциями для выполнения операций над контейнерами, например, сортировка и поиск.

#include <sstream> - Библиотека для работы с потоками строк, позволяющая преобразовывать строки в другие типы и обратно.

#include "json.hpp" - Библиотека для работы с JSON данными, обеспечивающая парсинг и генерацию JSON формата.

#include <fstream> - Библиотека для работы с файловыми потоками, позволяющая читать и записывать данные в файлы.

#include <filesystem> - Библиотека для работы с файловой системой, обеспечивающая доступ к файлам и директориям.

#include <locale> - Библиотека для работы с локалями, позволяющая адаптировать программы к различным языковым и культурным условиям.

**Листинг программы**

#include <iostream>

#include <string>

#include <algorithm>

#include <sstream>

#include <string>

#include "json.hpp"

#include <fstream>

#include <filesystem>

#include <locale>

**using** **namespace** std;

**using** json = nlohmann::json;

**namespace** fs = filesystem;

**struct** Node {

**int** numColumn;

string name;

string cell;

Node\* next;

};

**struct** RowNode {

string name;

Node\* cell;

RowNode\* nextRow;

};

**struct** Condition {

**bool** trueOrFalse;

string condition;

string oper;

Condition\* next;

};

**void** WriteUtf8BOM(ofstream& file) {

**const** **char** bom[] = { '\xEF', '\xBB', '\xBF' };

file.write(bom, **sizeof**(bom));

}

**void** FreeTable(RowNode\* table) {

**while** (table != **nullptr**) {

RowNode\* tempRow = table;

table = table->nextRow;

Node\* currNode = tempRow->cell;

**while** (currNode != **nullptr**) {

Node\* tempNode = currNode;

currNode = currNode->next;

**delete** tempNode;

}

**delete** tempRow;

}

}

**void** FreeAllTables(RowNode\*\* tables, **int** count) {

**for** (**int** i = 0; i < count; ++i) {

FreeTable(tables[i]); // Освобождаем память для каждой таблицы

}

}

**void** UpdatePrimaryKey(**const** string& schemaName, **const** string& tableName, RowNode\* table) {

**int** i = 0;

RowNode\* currentRow = table;

**while** (currentRow != **nullptr**) {

i++;

currentRow = currentRow->nextRow;

}

std::string pkFilePath = schemaName + "/" + tableName + "/" + tableName + "\_pk\_sequence.txt";

std::ofstream pkFile(pkFilePath, std::ios::binary);

**if** (pkFile.is\_open()) {

pkFile << i;

pkFile.close();

}

**else** {

std::cerr << "Не удалось открыть файл для обновления: " << pkFilePath << std::endl;

}

}

**int** countRowsInCSV(**const** string& csvFilePath) {

**int** i = 0;

ifstream csvFile(csvFilePath);

**if** (csvFile.is\_open()) {

string line;

**while** (getline(csvFile, line)) {

i++;

}

csvFile.close();

}

**else** {

cerr << "Ошибка открытия файла: " << csvFilePath << endl;

}

**return** i;

}

**void** ReadConfiguration(**const** string& filename) {

ifstream file(filename);

**if** (!file.is\_open()) {

cerr << "Не удалось открыть файл " << filename << endl;

**return**;

}

json config;

file >> config;

string schemaName = config["name"];

**int** tuplesLimit = config["tuples\_limit"];

fs::create\_directory(schemaName);

**for** (**const** **auto**& table : config["structure"].items()) {

string tableName = table.key();

fs::create\_directory(schemaName + "/" + tableName);

string csvFilePath = schemaName + "/" + tableName + "/1.csv";

ofstream csvFile(csvFilePath, ios::binary);

**if** (!csvFile.is\_open()) {

cerr << "Ошибка открытия файла для записи: " << csvFilePath << endl;

**continue**;

}

// Write UTF-8 BOM

WriteUtf8BOM(csvFile);

**const** **auto**& columns = table.value();

cout << "Таблица: " << tableName << ", Количество колонок: " << columns.size() << endl;

**if** (columns.size() > 0) {

**const** **auto**& headers = columns[0];

**for** (**const** **auto**& header : headers) {

csvFile << header.get<string>() << ",";

}

csvFile.seekp(-1, ios\_base::cur);

csvFile << "\n";

**for** (size\_t i = 1; i < columns.size(); ++i) {

**const** **auto**& row = columns[i];

cout << "Запись строки " << i << ": ";

**for** (**const** **auto**& value : row) {

cout << value.get<string>() << " "; // Отладочный вывод

**if** (!(csvFile << value.get<string>() << ",")) {

cerr << "Ошибка записи в файл: " << csvFilePath << endl;

}

}

csvFile.seekp(-1, ios\_base::cur);

csvFile << "\n";

}

}

**else** {

cerr << "Нет данных для записи в таблицу " << tableName << endl;

}

csvFile.close();

**int** rowCount = countRowsInCSV(csvFilePath);

ofstream pkFile(schemaName + "/" + tableName + "/" + tableName + "\_pk\_sequence.txt", ios::binary);

**if** (pkFile.is\_open()) {

pkFile << rowCount;

pkFile.close();

}

ofstream lockFile(schemaName + "/" + tableName + "/" + tableName + "\_lock.txt", ios::binary);

**if** (lockFile.is\_open()) {

lockFile << 0;

lockFile.close();

}

}

cout << "Директории были успешно созданы!" << endl;

}

**void** PrintTable(RowNode\* table) {

**int** i = 0;

**while** (table != **nullptr**) {

Node\* currNode = table->cell;

**while** (currNode != **nullptr**) {

cout << "\t" << currNode->cell;

currNode = currNode->next;

}

table = table->nextRow;

i++;

cout << endl;

}

}

**void** PrintTables(RowNode\*\* tables) {

**int** i = 0;

**while** (tables[i] != **nullptr**) {

cout << i + 1 << ":\n";

PrintTable(tables[i]);

i++;

}

}

RowNode\* AddTableNames(RowNode\* table, RowNode\*\* tables, **int** countTables) {

RowNode\* currRow = table;

**while** (currRow != **nullptr**) {

Node\* currCell = currRow->cell;

**int** i = 0;

**while** (currCell != **nullptr** && i < countTables) {

**if** (countTables == 1) {

currCell->name = tables[0]->name;

currCell = currCell->next;

}

**else** {

currCell->name = tables[i]->name;

currCell = currCell->next;

i++;

}

}

currRow = currRow->nextRow;

}

**return** table;

}

RowNode\* AddNumColumns(RowNode\* table, **int** numColumns[], **int** countTables) {

RowNode\* currRow = table;

**while** (currRow != **nullptr**) {

Node\* currCell = currRow->cell;

**int** i = 0;

**while** (currCell != **nullptr**) {

currCell->numColumn = numColumns[i];

currCell = currCell->next;

i++;

}

currRow = currRow->nextRow;

}

**return** table;

}

RowNode\* InsertInto(RowNode\* table, **const** string listString[]) {

RowNode\* newRow = **new** RowNode;

newRow->cell = **nullptr**;

newRow->nextRow = **nullptr**;

Node\* currNode = **nullptr**;

**for** (**int** i = 0; !listString[i].empty(); i++) {

Node\* newNode = **new** Node;

newNode->cell = listString[i];

newNode->next = **nullptr**;

**if** (newRow->cell == **nullptr**) {

newRow->cell = newNode;

}

**else** {

currNode->next = newNode;

}

currNode = newNode;

}

**if** (table == **nullptr**) {

**return** newRow;

}

**else** {

RowNode\* currRow = table;

**while** (currRow->nextRow != **nullptr**) {

currRow = currRow->nextRow;

}

currRow->nextRow = newRow;

**return** table;

}

}

RowNode\* SelectFromOneTable(RowNode\* table, **int** numColumns[]) {

RowNode\* currRow = table;

RowNode\* lastRow = **nullptr**;

RowNode\* crossTable = **nullptr**;

**int** sizeList = **sizeof**(numColumns) / **sizeof**(numColumns[0]);

**while** (currRow != **nullptr**) {

RowNode\* newRow = **new** RowNode;

newRow->cell = **nullptr**;

newRow->nextRow = **nullptr**;

Node\* currCell = currRow->cell;

**int** i = 1;

Node\* lastCell = **nullptr**;

**while** (currCell != **nullptr**) {

**for** (**int** num = 0; num < sizeList; num++) {

**if** (numColumns[num] == i) {

Node\* newCell = **new** Node;

newCell->cell = currCell->cell;

newCell->next = **nullptr**;

**if** (newRow->cell == **nullptr**) {

newRow->cell = newCell;

}

**else** {

lastCell->next = newCell;

}

lastCell = newCell;

}

}

currCell = currCell->next;

i++;

}

**if** (crossTable == **nullptr**) {

crossTable = newRow;

}

**else** {

lastRow->nextRow = newRow;

}

lastRow = newRow;

currRow = currRow->nextRow;

}

RowNode\* temp[] = { table };

crossTable = AddTableNames(crossTable, temp, 1);

crossTable = AddNumColumns(crossTable, numColumns, 1);

**return** crossTable;

}

RowNode\* SelectFromManyTables(RowNode\*\* tables, **int** numColumns[], **int** countTables) {

**int** maxRows = 0;

**for** (**int** i = 0; i < countTables; i++) {

**int** rows = 0;

RowNode\* current = tables[i];

**while** (current != **nullptr**) {

rows++;

current = current->nextRow;

}

**if** (rows > maxRows) {

maxRows = rows;

}

}

RowNode\* crossTable = **nullptr**;

RowNode\* lastRow = **nullptr**;

**for** (**int** i = 0; i < maxRows; i++) {

RowNode\* newRow = **new** RowNode;

newRow->cell = **nullptr**;

newRow->name = "";

newRow->nextRow = **nullptr**;

Node\* lastCell = **nullptr**;

**for** (**int** j = 0; j < countTables; j++) {

Node\* currCell = **nullptr**;

RowNode\* currRow = tables[j];

**for** (**int** k = 0; k < i && currRow != **nullptr**; k++) {

currRow = currRow->nextRow;

}

**if** (currRow != **nullptr**) {

Node\* columnCell = currRow->cell;

**int** targetColumn = numColumns[j];

**for** (**int** z = 0; z < targetColumn - 1 && columnCell != **nullptr**; z++) {

columnCell = columnCell->next;

}

currCell = **new** Node;

currCell->name = currRow->name;

currCell->cell = columnCell->cell;

currCell->next = **nullptr**;

}

**else** {

currCell = **new** Node;

currCell->name = "";

currCell->cell = "";

currCell->next = **nullptr**;

}

**if** (lastCell == **nullptr**) {

newRow->cell = currCell;

}

**else** {

lastCell->next = currCell;

}

lastCell = currCell;

}

**if** (crossTable == **nullptr**) {

crossTable = newRow;

}

**else** {

lastRow->nextRow = newRow;

}

lastRow = newRow;

}

crossTable = AddTableNames(crossTable, tables, countTables);

crossTable = AddNumColumns(crossTable, numColumns, countTables);

**return** crossTable;

}

RowNode\* SelectFromCartesian(RowNode\*\* tables, **int** countTables) {

RowNode\* crossTable = tables[0];

**for** (**int** i = 1; i < countTables; ++i) {

RowNode\* currentTable = tables[i];

RowNode\* newCrossTable = **nullptr**;

RowNode\* rowA = crossTable;

**while** (rowA != **nullptr**) {

RowNode\* rowB = currentTable;

**while** (rowB != **nullptr**) {

RowNode\* newRow = **new** RowNode;

newRow->cell = **nullptr**;

newRow->nextRow = **nullptr**;

Node\* currCellA = rowA->cell;

Node\* lastCell = **nullptr**;

**while** (currCellA != **nullptr**) {

Node\* newCell = **new** Node;

newCell->cell = currCellA->cell;

newCell->next = **nullptr**;

**if** (newRow->cell == **nullptr**) {

newRow->cell = newCell;

}

**else** {

lastCell->next = newCell;

}

lastCell = newCell;

currCellA = currCellA->next;

}

Node\* currCellB = rowB->cell;

**while** (currCellB != **nullptr**) {

Node\* newCell = **new** Node;

newCell->cell = currCellB->cell;

newCell->next = **nullptr**;

**if** (newRow->cell == **nullptr**) {

newRow->cell = newCell;

}

**else** {

lastCell->next = newCell;

}

lastCell = newCell;

currCellB = currCellB->next;

}

**if** (newCrossTable == **nullptr**) {

newCrossTable = newRow;

}

**else** {

RowNode\* lastRow = newCrossTable;

**while** (lastRow->nextRow != **nullptr**) {

lastRow = lastRow->nextRow;

}

lastRow->nextRow = newRow;

}

rowB = rowB->nextRow;

}

rowA = rowA->nextRow;

}

crossTable = newCrossTable;

}

**return** crossTable;

}

Condition\* SplitCondition(**const** string& filter) {

Condition\* firstElement = **nullptr**;

Condition\* lastElement = **nullptr**;

**int** begin = 0;

**int** end;

**while** (begin < filter.length()) {

end = filter.find("AND", begin);

**int** findOR = filter.find("OR", begin);

**if** (findOR != string::npos && (end == string::npos || findOR < end)) {

end = findOR;

}

**if** (end == string::npos) {

end = filter.length();

}

Condition\* newNode = **new** Condition;

newNode->condition = filter.substr(begin, end - begin);

newNode->next = **nullptr**;

**if** (end < filter.length()) {

**if** (filter.substr(end, 3) == "AND") {

newNode->oper = "AND";

end += 3;

}

**else** **if** (filter.substr(end, 2) == "OR") {

newNode->oper = "OR";

end += 2;

}

**else** {

newNode->oper = "";

}

}

**else** {

newNode->oper = "";

}

**if** (firstElement == **nullptr**) {

firstElement = newNode;

lastElement = firstElement;

}

**else** {

lastElement->next = newNode;

lastElement = newNode;

}

begin = end + 1;

}

**return** firstElement;

}

**bool** CheckingCondition(RowNode\* row, **const** string& condition) {

**int** equalPos = condition.find('=');

string leftSide = condition.substr(0, equalPos);

string rightSide = condition.substr(equalPos + 1);

leftSide.erase(remove(leftSide.begin(), leftSide.end(), ' '), leftSide.end());

rightSide.erase(remove(rightSide.begin(), rightSide.end(), ' '), rightSide.end());

**if** (rightSide[0] == '\'') {

rightSide.erase(remove(rightSide.begin(), rightSide.end(), '\''), rightSide.end());

RowNode\* newRow = row;

Node\* currCell = newRow->cell;

**int** pointPos = condition.find('.');

string nameTable = condition.substr(0, condition.find('.'));

**int** tempPos = condition.find("колонка") + strlen("колонка");

string numColumnStr = condition.substr(tempPos, equalPos - tempPos - 1);

**int** numColumn = stoi(numColumnStr);

**int** i = 1;

**while** (i != numColumn) {

currCell = currCell->next;

i++;

}

**if** (currCell == **nullptr**) {

**return** **false**;

}

**if** (currCell->cell == rightSide) {

**return** **true**;

}

**return** **false**;

}

**else** {

string tableNameA = leftSide.substr(0, leftSide.find('.'));

**int** equalPosA = leftSide.find('=');

**int** tempPosA = leftSide.find("колонка") + strlen("колонка");

string numColumnStrA = leftSide.substr(tempPosA, equalPosA - tempPosA - 1);

**int** numColumnA = stoi(numColumnStrA);

string tableNameB = rightSide.substr(0, rightSide.find('.'));

**int** equalPosB = rightSide.find('=');

**int** tempPosB = rightSide.find("колонка") + strlen("колонка");

string numColumnStrB = rightSide.substr(tempPosB, rightSide.size() - 1 - tempPosB - 1);

**int** numColumnB = stoi(numColumnStrB);

RowNode\* newRow = row;

Node\* currCell = newRow->cell;

Node\* cellA = **new** Node;

cellA->cell = "";

Node\* cellB = **new** Node;

cellB->cell = "";

**while** (currCell != **nullptr**) {

**if** (currCell->name == tableNameA && currCell->numColumn == numColumnA) {

cellA->cell = currCell->cell;

}

**if** (currCell->name == tableNameB && currCell->numColumn == numColumnB) {

cellB->cell = currCell->cell;

}

currCell = currCell->next;

}

**if** (cellA != **nullptr** && cellB != **nullptr**) {

**return** cellA->cell == cellB->cell;

}

**return** **false**;

}

}

**bool** CheckingLogicalExpression(RowNode\* row, Condition\* condition) {

string result;

**while** (condition != **nullptr**) {

**bool** tempResult = CheckingCondition(row, condition->condition);

result += (tempResult ? "1" : "0");

**if** (condition->next != **nullptr**) {

result += " " + condition->oper + " ";

}

condition = condition->next;

}

**while** (result.find("1 AND 0") != string::npos) {

result.replace(result.find("1 AND 0"), 7, "0");

}

**while** (result.find("0 AND 1") != string::npos) {

result.replace(result.find("0 AND 1"), 7, "0");

}

**while** (result.find("0 AND 0") != string::npos) {

result.replace(result.find("0 AND 0"), 7, "0");

}

**while** (result.find("0 AND 0") != string::npos) {

result.replace(result.find("1 AND 1"), 7, "1");

}

string cleanedResult;

**int** pos = 0;

**while** (pos < result.length()) {

**int** nextOp = result.find("AND", pos);

**if** (nextOp == string::npos) {

cleanedResult += result.substr(pos);

**break**;

}

cleanedResult += result.substr(pos, nextOp - pos);

pos = nextOp + 3;

**while** (pos < result.length() && result[pos] == ' ') {

pos++;

}

}

**if** (cleanedResult.find("1") == string::npos) {

**return** **false**;

}

**return** **true**;

}

RowNode\* FilteringTable(RowNode\*\* select, RowNode\*\* where, **int** selectSize, **int** whereSize, **int** numColumnsSelect[], **int** numColumnsWhere[], string condition) {

RowNode\* resultTable = **nullptr**;

RowNode\* editTable;

RowNode\* rateTable;

Condition\* SplitConditioned = SplitCondition(condition);

**if** (selectSize == 1) {

editTable = SelectFromOneTable(select[0], numColumnsSelect);

rateTable = SelectFromOneTable(where[0], numColumnsWhere);

}

**else** {

editTable = SelectFromManyTables(select, numColumnsSelect, selectSize);

rateTable = SelectFromManyTables(where, numColumnsWhere, whereSize);

}

RowNode\* currRow = rateTable;

**while** (currRow != **nullptr**) {

**if** (CheckingLogicalExpression(currRow, SplitConditioned)) {

RowNode\* newRow = editTable;

string listString[1000];

**int** index = 0;

Node\* currCell = newRow->cell;

**while** (currCell != **nullptr**) {

listString[index++] = currCell->cell;

currCell = currCell->next;

}

listString[index] = "";

// Добавляем строку в результирующую таблицу

resultTable = InsertInto(resultTable, listString);

}

currRow = currRow->nextRow;

}

**return** resultTable;

}

RowNode\* DeleteFrom(RowNode\* table, Condition\* condition) {

RowNode\* currRow = table;

RowNode\* prevRow = **nullptr**;

**while** (currRow != **nullptr**) {

**if** (CheckingLogicalExpression(currRow, condition)) {

RowNode\* nextRow = currRow->nextRow;

**delete** currRow;

**if** (prevRow == **nullptr**) {

table = nextRow;

currRow = nextRow;

}

**else** {

prevRow->nextRow = nextRow;

currRow = nextRow;

}

}

**else** {

prevRow = currRow;

currRow = currRow->nextRow;

}

}

**return** table;

}

**void** CreateTable(**const** string& schemaDir, **const** string& tableName) {

string tableDir = schemaDir + "/" + tableName;

**if** (!fs::create\_directory(tableDir)) {

cerr << "Не удалось создать директорию " << tableDir << endl;

**return**;

}

ofstream csvFile(tableDir + "/1.csv", ios::binary);

**if** (!csvFile.is\_open()) {

cerr << "Не удалось создать файл " << tableDir + "/1.csv" << endl;

**return**;

}

csvFile.close();

ofstream pkFile(tableDir + "/" + tableName + "\_pk\_sequence.txt", ios::binary);

**if** (!pkFile.is\_open()) {

cerr << "Не удалось создать файл " << tableDir + "/" + tableName + "\_pk\_sequence.txt" << endl;

**return**;

}

pkFile << "1";

pkFile.close();

ofstream lockFile(tableDir + "/" + tableName + "\_lock.txt", ios::binary);

**if** (!lockFile.is\_open()) {

cerr << "Не удалось создаhь файл " << tableDir + "/" + tableName + "\_lock.txt" << endl;

**return**;

}

lockFile << "0";

lockFile.close();

cout << "Таблица " << tableName << " успешно создана в " << tableDir << endl;

string schemaFilePath = "schema.json";

ifstream schemaFile(schemaFilePath);

**if** (!schemaFile.is\_open()) {

cerr << "Не удалось открыть файл " << schemaFilePath << endl;

**return**;

}

json schema;

schemaFile >> schema;

schemaFile.close();

schema["structure"][tableName] = json::array();

ofstream outputFile(schemaFilePath);

**if** (outputFile.is\_open()) {

outputFile << schema.dump(4);

outputFile.close();

}

**else** {

cerr << "Не удалось записать в файл " << schemaFilePath << endl;

}

}

json ConvertTableToJson(RowNode\* table) {

json jsonData = json::array();

**while** (table != **nullptr**) {

json rowJson;

Node\* currNode = table->cell;

**while** (currNode != **nullptr**) {

rowJson.push\_back(currNode->cell);

currNode = currNode->next;

}

jsonData.push\_back(rowJson);

table = table->nextRow;

}

**return** jsonData;

}

RowNode\* ConvertJsonToTable(**const** json& jsonData, **const** string& tableName) {

RowNode\* table = **nullptr**;

// Проверяем, есть ли данные для конкретной таблицы в структуре

**if** (jsonData.contains("structure") && jsonData["structure"].is\_object()) {

**if** (jsonData["structure"].contains(tableName)) {

**const** **auto**& tableData = jsonData["structure"][tableName];

// Проверяем, что tableData является массивом

**if** (tableData.is\_array()) {

**for** (**const** **auto**& row : tableData) {

**if** (row.is\_array()) {

string listString[100]; // Предполагаем, что строка не больше 100 элементов

**int** index = 0;

// Заполняем массив значениями из текущей строки

**for** (**const** **auto**& cell : row) {

**if** (index < 100) { // Убедитесь, что не выйдете за пределы массива

listString[index++] = cell.get<string>();

}

}

// Вставляем заполненную строку в таблицу

table = InsertInto(table, listString);

}

**else** {

cout << "Ошибка: строка не является массивом." << endl;

}

}

}

**else** {

cout << "Ошибка: данные для таблицы не являются массивом." << endl;

}

}

**else** {

cout << "Ошибка: таблица '" << tableName << "' не найдена в данных." << endl;

}

}

**else** {

cout << "Ошибка: структура не найдена или не является объектом." << endl;

}

**return** table;

}

**void** WriteJsonToCSV(**const** string& filePath, **const** json& jsonData) {

ofstream csvFile(filePath, ios::app);

**if** (csvFile.is\_open()) {

ifstream checkFile(filePath);

**bool** isEmpty = checkFile.peek() == ifstream::traits\_type::eof();

checkFile.close();

**if** (isEmpty) {

WriteUtf8BOM(csvFile);

**for** (size\_t i = 0; i < jsonData[0].size(); ++i) {

csvFile << jsonData[0][i].get<string>();

**if** (i < jsonData[0].size() - 1) {

csvFile << ",";

}

}

csvFile << "\n";

}

**for** (**const** **auto**& row : jsonData) {

**for** (size\_t i = 0; i < row.size(); ++i) {

csvFile << row[i].get<string>();

**if** (i < row.size() - 1) {

csvFile << ",";

}

}

csvFile << "\n";

}

csvFile.close();

}

**else** {

cerr << "Не удалось открыть файл для записи: " << filePath << endl;

}

}

**void** splitString(string& str, string\* tokens) {

**if** (!str.empty() && str.front() == '{') {

str.erase(0, 1);

}

**if** (!str.empty() && str.back() == '}') {

str.erase(str.size() - 1);

}

**int** count = 0;

stringstream ss(str);

string token;

**while** (getline(ss, token, ',') && count < 1000) {

// Убираем пробелы в начале и конце токена

token.erase(0, token.find\_first\_not\_of(" \n\r\t"));

token.erase(token.find\_last\_not\_of(" \n\r\t") + 1);

tokens[count++] = token; // Записываем токен в массив

}

}

json ReadCSVToJson(**const** string& filePath) {

**if** (!ifstream(filePath)) {

cerr << "Файл не найден: " << filePath << endl;

**return** -1;

}

ifstream csvFile(filePath);

json jsonData;

**if** (!csvFile.is\_open()) {

cerr << "Не удалось открыть файл для чтения: " << filePath << endl;

**return** jsonData;

}

string line;

**while** (getline(csvFile, line)) {

**if** (line == "") {

jsonData = json::array({ json::array() });

**return** jsonData;

}

stringstream ss(line);

json rowJson = json::array();

string value;

**while** (getline(ss, value, ',')) {

value.erase(0, value.find\_first\_not\_of(" \n\r\t"));

value.erase(value.find\_last\_not\_of(" \n\r\t") + 1);

**if** (!value.empty()) {

rowJson.push\_back(value);

}

}

**if** (!rowJson.empty()) {

jsonData.push\_back(rowJson);

}

}

csvFile.close();

**return** jsonData;

}

**void** AddColumnsInSchemaJson(**const** json& newColumns, **const** string& schemaFilePath, **const** string& tableName) {

ifstream schemaFile(schemaFilePath);

**if** (!schemaFile.is\_open()) {

cerr << "Не удалось открыть файл схемы: " << schemaFilePath << endl;

**return**;

}

json schemaData;

schemaFile >> schemaData;

schemaFile.close();

**if** (!schemaData["structure"].contains(tableName)) {

cerr << "Таблица не найдена: " << tableName << endl;

**return**;

}

**if** (!newColumns.is\_array()) {

cerr << "Данные колонок должны быть массивом." << endl;

**return**;

}

**for** (**const** **auto**& column : newColumns) {

**if** (column.is\_array()) {

json newRow = json::array();

**for** (**const** **auto**& value : column) {

newRow.push\_back(value);

}

schemaData["structure"][tableName].push\_back(newRow);

}

}

ofstream outFile(schemaFilePath);

**if** (outFile.is\_open()) {

outFile << schemaData.dump(4);

outFile.close();

}

**else** {

cerr << "Не удалось записать в файл: " << schemaFilePath << endl;

}

cout << "Новая строка успешно добавлена! " << endl;

}

**void** RewriteTableSchema(**const** json& newSchema, **const** string& schemaFilePath, **const** string& tableName) {

ifstream schemaFile(schemaFilePath);

**if** (!schemaFile.is\_open()) {

cerr << "Не удалось открыть файл схемы: " << schemaFilePath << endl;

**return**;

}

json schemaData;

schemaFile >> schemaData;

schemaFile.close();

// Проверяем, существует ли таблица

**if** (!schemaData["structure"].contains(tableName)) {

cerr << "Таблица не найдена: " << tableName << endl;

**return**;

}

schemaData["structure"][tableName] = newSchema;

ofstream outFile(schemaFilePath);

**if** (outFile.is\_open()) {

outFile << schemaData.dump(4); // Записываем с отступами

outFile.close();

}

**else** {

cerr << "Не удалось записать в файл: " << schemaFilePath << endl;

}

}

**void** RewriteCSVbyJson(**const** string& filePath, **const** json& jsonData) {

ofstream csvFile(filePath);

**if** (csvFile.is\_open()) {

WriteUtf8BOM(csvFile);

**for** (**const** **auto**& row : jsonData) {

**if** (row.is\_array()) {

**for** (size\_t i = 0; i < row.size(); ++i) {

csvFile << row[i].get<string>();

**if** (i < row.size() - 1) {

csvFile << ",";

}

}

csvFile << "\n";

}

}

csvFile.close();

}

**else** {

cerr << "Не удалось открыть файл для записи: " << filePath << endl;

}

}

**void** incrementSequence(**const** string& tableName) {

string fileName = string("Схема 1") + "/" + tableName + "/" + tableName + "\_pk\_sequence.txt";

ifstream inputFile(fileName);

**if** (!inputFile) {

cerr << "Ошибка открытия файла для чтения." << endl;

**return**;

}

string currValueStr;

inputFile >> currValueStr;

inputFile.close();

**int** currValue = stoi(currValueStr);

currValue++;

ofstream outputFile(fileName);

**if** (!outputFile) {

cerr << "Ошибка открытия файла для записи." << endl;

**return**;

}

outputFile << currValue;

outputFile.close();

}

**void** decrementLock(**const** string& tableName) {

string fileName = string("Схема 1") + "/" + tableName + "/" + tableName + "\_lock.txt";

ifstream inputFile(fileName);

**if** (!inputFile) {

cerr << "Ошибка открытия файла для чтения." << endl;

**return**;

}

string currValueStr;

inputFile >> currValueStr;

inputFile.close();

**int** currValue = stoi(currValueStr);

currValue--;

ofstream outputFile(fileName);

**if** (!outputFile) {

cerr << "Ошибка открытия файла для записи." << endl;

**return**;

}

outputFile << currValue;

outputFile.close();

}

**int** getNextCsv(**const** string& tableName) {

string fileName = "Схема 1/" + tableName + "/" + tableName + "\_lock.txt";

ifstream lockFile(fileName);

**if** (!lockFile) {

cerr << "Ошибка открытия файла: " << fileName << endl;

**return** -1;

}

**int** lineCount = 0;

string line;

**while** (getline(lockFile, line)) {

lineCount++;

}

lockFile.close();

**int** maxIndex = 0;

**for** (**const** **auto**& entry : filesystem::directory\_iterator("Схема 1/" + tableName)) {

**if** (entry.path().extension() == ".csv") {

string filename = entry.path().stem().string();

**int** index = stoi(filename);

maxIndex = max(maxIndex, index);

}

}

**return** (lineCount <= 1000) ? maxIndex : (maxIndex + 1);

}

**int** main() {

setlocale(LC\_ALL, "ru\_RU.UTF-8");

string scheme = "Схема 1";

ReadConfiguration("schema.json");

string command;

**while** (**true**) {

cout << "Введите команду: ";

getline(cin, command);

**if** (command == "exit") {

**break**;

}

**if** (command.find("CREATE TABLE") != string::npos) {

**auto** pos = command.find("CREATE TABLE");

string nameTable = command.substr(pos + strlen("CREATE TABLE "));

CreateTable(scheme, nameTable);

}

**else** **if** (command.find("INSERT INTO") != string::npos) {

RowNode\* tempTable = **nullptr**;

**auto** pos = command.find("INSERT INTO") + strlen("INSERT INTO ");

**auto** endPos = command.find(' ', pos);

string nameTable = (endPos == string::npos)

? command.substr(pos)

: command.substr(pos, endPos - pos);

decrementLock(nameTable);

**auto** valuesStart = endPos + 1;

string valuesString = command.substr(valuesStart);

string value[100];

splitString(valuesString, value);

**int** csv = getNextCsv(nameTable);

ifstream checkFile(scheme + "/" + nameTable + "/" + to\_string(csv) + ".csv");

checkFile.close();

string line;

tempTable = InsertInto(tempTable, value);

PrintTable(tempTable);

json tempJson2 = ConvertTableToJson(tempTable);

WriteJsonToCSV(scheme + "/" + nameTable + "/" + to\_string(csv) + ".csv", tempJson2);

AddColumnsInSchemaJson(tempJson2, "schema.json", nameTable);

incrementSequence(nameTable);

FreeTable(tempTable);

}

**else** **if** (command.find("DELETE FROM") != string::npos) {

**auto** pos = command.find("DELETE FROM") + strlen("DELETE FROM WHERE ");

**auto** posPoint = command.find(".");

string nameTable = (posPoint == string::npos)

? command.substr(pos)

: command.substr(pos, posPoint - pos);

decrementLock(nameTable);

**auto** posGap = command.find(" =");

posPoint = posPoint + strlen("колонка") + 1;

string numColumnStr = (posGap == string::npos)

? command.substr(posPoint)

: command.substr(posPoint, posGap - posPoint);

string value = command.substr(posGap + 3);

string criterion = command.substr(pos);

Condition\* condition = SplitCondition(criterion);

ifstream file("schema.json");

json tempJson;

file >> tempJson;

file.close();

RowNode\* newTable = ConvertJsonToTable(tempJson, nameTable);

RowNode\* tempTables[] = { newTable };

newTable->name = nameTable;

AddTableNames(newTable, tempTables, 1);

**int** i = 0;

Node\* temp = newTable->cell;

**while** (temp != **nullptr**) {

temp = temp->next;

i++;

}

**int** countColumns[i];

AddNumColumns(newTable, countColumns, 1);

newTable = DeleteFrom(newTable, condition);

UpdatePrimaryKey("Схема 1", nameTable, newTable);

json tempJson1 = ConvertTableToJson(newTable);

RewriteTableSchema(tempJson1, "schema.json", nameTable);

RewriteCSVbyJson(scheme + "/" + nameTable + "/" + "1.csv", tempJson1);

cout << endl << "Строки были успешно удалены!" << endl;

FreeTable(newTable);

}

**else** **if** (command.find("SELECT") != string::npos) {

string namesTables;

**if** (command.find("\*") != string::npos) {

**int** pos = command.find("SELECT \* FROM ") + strlen("SELECT \* FROM ");

namesTables = command.substr(pos);

stringstream ss(namesTables);

string tableName;

string tables[2];

**int** count = 0;

**while** (getline(ss, tableName, ',')) {

tableName.erase(remove(tableName.begin(), tableName.end(), ' '), tableName.end());

**if** (count < 2) {

tables[count] = tableName;

decrementLock(tableName);

count++;

}

}

**if** (count < 2) {

cout << "Недостаточное количество таблиц. Необходимо две." << endl;

**continue**;

}

ifstream file("schema.json");

json tempJson;

file >> tempJson;

file.close();

RowNode\* newTableA = ConvertJsonToTable(tempJson, tables[0]);

RowNode\* newTableB = ConvertJsonToTable(tempJson, tables[1]);

RowNode\* newTables[] = { newTableA, newTableB };

RowNode\* crossTable = SelectFromCartesian(newTables, 2);

incrementSequence(tables[0]);

incrementSequence(tables[1]);

cout << endl << "Пересечение двух таблиц: " << endl;

FreeTable(crossTable);

}

**else** **if** (command.find("FROM") == string::npos) {

**int** pos = command.find("SELECT") + strlen("SELECT ");

namesTables = command.substr(pos);

stringstream ss(namesTables);

string tableName;

string tables[5];

**int** numColumns[5];

**int** count = 0;

**while** (getline(ss, tableName, ',')) {

tableName.erase(remove(tableName.begin(), tableName.end(), ' '), tableName.end());

size\_t pos = tableName.find('.');

**if** (pos != string::npos) {

string table = tableName.substr(0, pos);

**int** lenStrColumn = tableName.find("колонка");

string columnStr = tableName.substr(lenStrColumn + strlen("колонка"));

**int** column = stoi(columnStr);

**if** (count < 5) {

tables[count] = table;

decrementLock(table);

numColumns[count] = column;

count++;

}

}

**else** {

cout << "Некорректный формат: " << tableName << endl;

**continue**;

}

}

ifstream file("schema.json");

json tempJson;

file >> tempJson;

**if** (count > 5) {

cout << "Некорректное количество таблиц" << endl;

**continue**;

}

RowNode\* newTables[5];

**for** (**int** i = 0; i < count; i++) {

RowNode\* temp = ConvertJsonToTable(tempJson, tables[i]);

temp->name = tables[i];

newTables[i] = temp;

incrementSequence(tables[i]);

}

RowNode\* crossTable = SelectFromManyTables(newTables, numColumns, count);

cout << endl << "Итоговая таблица: " << endl;

PrintTable(crossTable);

FreeTable(crossTable);

file.close();

}

**else** **if** (command.find("FROM") != string::npos) {

**int** pos1 = command.find("SELECT") + strlen("SELECT ");

**int** posFrom = command.find("FROM");

string namesTablesSelect = command.substr(pos1, posFrom - pos1);

stringstream ss1(namesTablesSelect);

string tableNameSelect;

string tablesSelect[5];

**int** numColumnsSelect[5];

**int** count1 = 0;

**while** (getline(ss1, tableNameSelect, ',')) {

tableNameSelect.erase(remove(tableNameSelect.begin(), tableNameSelect.end(), ' '), tableNameSelect.end());

size\_t pos = tableNameSelect.find('.');

**if** (pos != string::npos) {

string table = tableNameSelect.substr(0, pos);

**int** lenStrColumn = tableNameSelect.find("колонка");

string columnStr = tableNameSelect.substr(lenStrColumn + strlen("колонка"));

**int** column = stoi(columnStr);

**if** (count1 < 5) {

tablesSelect[count1] = table;

decrementLock(table);

numColumnsSelect[count1] = column;

count1++;

}

}

}

**if** (count1 > 5) {

cout << "Некорректное количество таблиц" << endl;

**continue**;

}

**for** (**int** i = count1; i < 5; i++) {

tablesSelect[i].clear();

numColumnsSelect[i] = 0;

}

ifstream file("schema.json");

json tempJson;

file >> tempJson;

RowNode\* newTablesSelect[5];

**for** (**int** i = 0; i < count1; i++) {

RowNode\* temp = ConvertJsonToTable(tempJson, tablesSelect[i]);

temp->name = tablesSelect[i];

incrementSequence(tablesSelect[i]);

newTablesSelect[i] = temp;

}

**int** posWhere = command.find("WHERE") + strlen("WHERE ");

string namesTablesWhere;

Condition\* current = SplitCondition(command.substr(posWhere));

**while** (current != **nullptr**) {

**if** (!namesTablesWhere.empty()) {

namesTablesWhere += ", ";

}

**int** posEqual = current->condition.find('=');

**if** (posEqual != string::npos) {

string leftSide = current->condition.substr(0, posEqual);

namesTablesWhere += leftSide;

}

current = current->next;

}

stringstream ss2(namesTablesWhere);

string tableNameWhere;

string tablesWhere[5];

**int** numColumnsWhere[5];

**int** count2 = 0;

**while** (getline(ss2, tableNameWhere, ',')) {

tableNameWhere.erase(remove(tableNameWhere.begin(), tableNameWhere.end(), ' '), tableNameWhere.end());

size\_t pos = tableNameWhere.find('.');

**if** (pos != string::npos) {

string table = tableNameWhere.substr(0, pos);

**int** lenStrColumn = tableNameWhere.find("колонка");

string columnStr = tableNameWhere.substr(lenStrColumn + strlen("колонка"));

**int** column = stoi(columnStr);

**if** (count2 < 5) {

tablesWhere[count2] = table;

decrementLock(table);

numColumnsWhere[count2] = column;

count2++;

}

}

}

**for** (**int** i = count2; i < 5; i++) {

tablesWhere[i].clear();

numColumnsWhere[i] = 0;

}

json tempJson1;

file >> tempJson1;

RowNode\* newTablesWhere[5];

**for** (**int** i = 0; i < count2; i++) {

RowNode\* temp = ConvertJsonToTable(tempJson, tablesWhere[i]);

temp->name = tablesWhere[i];

newTablesWhere[i] = temp;

}

string filter = command.substr(posWhere);

RowNode\* crossTable = FilteringTable(newTablesSelect, newTablesWhere, count1, count2, numColumnsSelect, numColumnsWhere, filter);

cout << endl << "Итоговая таблица: " << endl;

PrintTable(crossTable);

FreeAllTables(newTablesSelect, count1);

FreeAllTables(newTablesWhere, count2);

file.close();

}

}

**else** {

cout << endl << "Некорректный ввод" << endl;

}

}

**return** 0;

}