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. «АБс-322», «АВТФ»  *Михайлов Е. С.*  «14» октября 2024 г.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Доцент кафедры ЗИ*  *Архипова А. Б.*  «\_\_\_» \_\_\_\_\_\_ 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2024

**Задание**

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

**Решение**

**GitHub:** https://github.com/Discorre/untitled

**Точка входа в программу**

**main.cpp:**  
#include <iostream>

#include <filesystem>

#include "CustomStructures/MyVector.hpp"

#include "CustomStructures/MyHashMap.hpp"

#include "Other/JsonParser.hpp"

#include "Other/Utilities.hpp"

#include "CRUDOperations/SelectValue.hpp"

#include "CRUDOperations/InsertValue.hpp"

#include "CRUDOperations/DeleteValue.hpp"

using namespace std;

// Парсит и выполняет SQL-запросы

void parsingQuery(const string& query, const string& filePath, const string& namesOfSchema, const int limitOfTuples, const MyHashMap<string, MyVector<string>\*>& jsonStructure) {

MyVector<string>\* words = splitRow(query, ' '); // Разбиваем запрос на слова

if (words->data[0] == "SELECT") {

try {

parseSelect(\*words, filePath, namesOfSchema, jsonStructure); // Выполняем SELECT запрос

} catch (const exception& err) {

cerr << err.what() << endl; // Выводим ошибку, если она возникла

}

} else if (words->data[0] == "INSERT" && words->data[1] == "INTO") {

try {

parseInsert(\*words, filePath, namesOfSchema, limitOfTuples, jsonStructure); // Выполняем INSERT запрос

} catch (const exception& err) {

cerr << err.what() << endl; // Выводим ошибку, если она возникла

}

} else if (words->data[0] == "DELETE" && words->data[1] == "FROM") {

try {

parseDelete(\*words, filePath, namesOfSchema, jsonStructure); // Выполняем DELETE запрос

} catch (const exception& err) {

cerr << err.what() << endl; // Выводим ошибку, если она возникла

}

} else {

cout << "Неизвестная команда" << endl; // Выводим сообщение, если команда не распознана

}

}

// Ввод имени файла и директории

int InputNames(string& jsonFileName, string& filePath) {

while (true) {

cout << "Введите имя json файла: ";

getline(cin, jsonFileName);

cout << "Введите путь к файлу: ";

getline(cin, filePath);

try {

if (!filesystem::exists(filePath + "\\" + jsonFileName)) {

throw std::runtime\_error("Файл JSON не найден"); // Выбрасываем ошибку, если файл не найден

} else {

return 0;

}

} catch (const exception& err) {

cerr << "Ошибка: " << err.what() << endl; // Выводим ошибку, если она возникла

}

}

}

int main() {

string jsonFileName;

string filePath;

//InputNames(jsonFileName, filePath);

jsonFileName = "schema.json";

filePath = ".";

MyHashMap<string, MyVector<string>\*>\* jsonStructure = CreateMap<string, MyVector<string>\*>(10, 50); // Создаем хэш-таблицу для структуры JSON

// Создание директорий

int limitOfTuples = 0;

string namesOfSchema = readJsonFile(jsonFileName, filePath, limitOfTuples, \*jsonStructure); // Читаем JSON файл и создаем директории

while (true) {

cout << endl;

cout << "Введите запрос для выполнения или exit для выхода из программы: " << endl;

cout << ">>> ";

string query;

getline(cin, query);

// Парсинг запроса

if (query == "exit") break; // Выход из цикла, если введенo "exit"

cout << endl;

parsingQuery(query, filePath, namesOfSchema, limitOfTuples, \*jsonStructure); // Парсим и выполняем запрос

}

DestroyMap<string, MyVector<string>\*>(\*jsonStructure); // Освобождаем память

return 0;

}

**JsonParser.hpp:**  
#ifndef JSONPARSER\_HPP

#define JSONPARSER\_HPP

#include "../CustomStructures/MyVector.hpp"

#include "../CustomStructures/MyHashMap.hpp"

#include "../CustomStructures/json.hpp"

#include <fstream>

#include "filesystem"

using json = nlohmann::json;

using namespace std;

// Создание директории

void createDirectory(const string& pathToDir) {

filesystem::\_\_cxx11::path path(pathToDir);

if (!filesystem::exists(path)) {

filesystem::create\_directories(path); // Создаем директорию, если она не существует

}

}

// Создание файла с данными

void createFileData(const string& pathToFile, const string& fileName, const string& data, bool isDirectory) {

filesystem::path path(pathToFile);

if (filesystem::exists(path / fileName)) {

if (isDirectory) {

ifstream file(path / fileName);

string line;

getline(file, line);

if (line == data) {

file.close();

return; // Данные уже есть в файле

}

file.close();

} else {

return;

}

}

// Если данные в файле не совпадают с JSON или отсутствуют

ofstream lockFile(path / fileName);

if (lockFile.is\_open()) {

lockFile << data; // Записываем данные в файл

lockFile.close();

} else {

throw runtime\_error("Не удалось создать файл блокировки в директории"); // Выбрасываем ошибку, если не удалось создать файл

}

}

// Чтение json файла и создание директорий

string readJsonFile(const string& fileName, const string& filePath, int& tuplesLimit, MyHashMap<string, MyVector<string>\*>& jsonStructure) {

ifstream file(filePath + "/" + fileName);

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

throw runtime\_error("Не удалось открыть " + fileName); // Выбрасываем ошибку, если не удалось открыть файл

}

// Чтение json

json schema;

file >> schema;

// Чтение имени таблицы

string schemaName = schema["name"];

createDirectory(schemaName); // Создаем директорию для таблицы

// Чтение максимального количества ключей

tuplesLimit = schema["tuples\_limit"];

// Чтение структуры таблицы

json tableStructure = schema["structure"];

for (auto& [key, value] : tableStructure.items()) {

// Создание директорий

createDirectory(schemaName + "/" + key);

MyVector<string>\* tempValue = CreateVector<string>(10, 50); // Создаем вектор для имен столбцов

string colNames = key + "\_pk";

for (auto columns : value) {

colNames += ",";

string temp = columns;

colNames += temp;

AddVector(\*tempValue, temp); // Добавляем имя столбца в вектор

}

createFileData(schemaName + "/" + key, "1.csv", colNames, true); // Создаем файл с именами столбцов

createFileData(schemaName + "/" + key, key + "\_lock.txt", "0", true); // Создаем файл блокировки

createFileData(schemaName + "/" + key, key + "\_pk\_sequence.txt", "0", false); // Создаем файл последовательности первичных ключей

AddMap<string, MyVector<string>\*>(jsonStructure, key, tempValue); // Добавляем структуру таблицы в хэш-таблицу

}

file.close();

return schemaName;

}

#endif //READJSON\_H

Utilities.hpp:  
#ifndef UTILITIES\_HPP

#define UTILITIES\_HPP

#include <string>

#include "../CustomStructures/MyVector.hpp"

#include <fstream>

using namespace std;

// Возвращает длину строки

int getLen(const string &str) {

int length = 0;

while (str[length] != '\0') {

length++;

}

return length;

}

// Возвращает подстроку от start до end (не включая end)

string getSubstring(const string &str, int start, int end) {

string result;

for (int i = start; i < end; i++) {

result += str[i];

}

return result;

}

// Разбивает строку на слова с разделителем delim.

MyVector<string>\* splitRow(const string &str, char delim) {

int index = 0;

MyVector<string>\* words = CreateVector<string>(10, 50); // Создаем вектор для слов

int length = getLen(str);

while (true) {

int delimIndex = index;

while (str[delimIndex] != delim && delimIndex != length) delimIndex++; // Ищем разделитель

string word = getSubstring(str, index, delimIndex); // Получаем слово

AddVector(\*words, word); // Добавляем слово в вектор

index = delimIndex + 1;

if (delimIndex == length) break; // Если достигли конца строки, выходим из цикла

}

return words;

}

// Проверка на занятость таблицы другим пользователем

void CheckTableLock(const string& path, const string& fileName, const int rank) {

fstream lockFile(path + "/" + fileName);

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

throw runtime\_error("Не удалось открыть " + (path + "/" + fileName)); // Выбрасываем ошибку, если не удалось открыть файл

}

int lock = 0;

lockFile >> lock; // Читаем текущее состояние блокировки

if (lock == 1 && rank == 1) {

lockFile.close();

throw runtime\_error("Таблица " + fileName + " заблокирована другим процессом"); // Выбрасываем ошибку, если таблица заблокирована

} else {

lockFile << rank; // Устанавливаем новое состояние блокировки

}

lockFile.close();

}

#endif

MyHashMap.hpp:

#ifndef MYHASHMAP\_H

#define MYHASHMAP\_H

#include <iostream>

#include <string>

// Структура для хранения значения

template <typename TK, typename TV>

struct NodeMap {

TK key; // Ключ

TV value; // Значение

NodeMap\* next; // Указатель на следующий узел (для обработки коллизий)

};

// Структура для хранения ключа и значения

template <typename TK, typename TV>

struct MyHashMap {

NodeMap<TK, TV>\*\* data; // Массив указателей на узлы

int length; // Количество элементов в хэш-таблице

int capacity; // Вместимость хэш-таблицы

int LoadFactor; // Фактор загрузки (процент заполнения, при котором происходит расширение)

};

// Хэш-функция для ключа типа int

template <typename TK>

int HashCode(const TK& key) {

unsigned long hash = 5381; // Начальное значение хэша

int c = 0;

for (char ch : key) {

hash = ((hash << 5) + hash) + ch; // hash \* 33 + c

}

return hash;

}

// Инициализация хэш-таблицы

template <typename TK, typename TV>

MyHashMap<TK, TV>\* CreateMap(int initCapacity, int initLoadFactor) {

if (initCapacity <= 0 || initLoadFactor <= 0 || initLoadFactor > 100) {

throw std::runtime\_error("Индекс вне диапазона"); // Выбрасываем ошибку, если параметры некорректны

}

MyHashMap<TK, TV>\* map = new MyHashMap<TK, TV>; // Создаем новую хэш-таблицу

map->data = new NodeMap<TK, TV>\*[initCapacity]; // Выделяем память под массив указателей

for (int i = 0; i < initCapacity; i++) {

map->data[i] = nullptr; // Инициализируем все указатели как nullptr

}

map->length = 0; // Инициализируем количество элементов

map->capacity = initCapacity; // Устанавливаем вместимость

map->LoadFactor = initLoadFactor; // Устанавливаем фактор загрузки

return map;

}

// Расширение хэш-таблицы

template <typename TK, typename TV>

void Expansion(MyHashMap<TK, TV>& map) {

int newCap = map.capacity \* 2; // Новая вместимость в два раза больше текущей

NodeMap<TK, TV>\*\* newData = new NodeMap<TK, TV>\*[newCap]; // Выделяем память под новый массив указателей

for (int i = 0; i < newCap; i++) {

newData[i] = nullptr; // Инициализируем все указатели как nullptr

}

// Проход по всем ячейкам

for (int i = 0; i < map.capacity; i++) {

NodeMap<TK, TV>\* curr = map.data[i];

// Проход по парам коллизионных значений и обновление

while (curr != nullptr) {

NodeMap<TK, TV>\* next = curr->next;

int index = HashCode(curr->key) % newCap; // Вычисляем новый индекс

curr->next = newData[index];

newData[index] = curr;

curr = next;

}

}

delete[] map.data; // Освобождаем память старого массива

map.data = newData; // Устанавливаем новый массив

map.capacity = newCap; // Обновляем вместимость

}

// Обработка коллизий

template <typename TK, typename TV>

void CollisionManage(MyHashMap<TK, TV>& map, int index, const TK& key, const TV& value) {

NodeMap<TK, TV>\* newNode = new NodeMap<TK, TV>{key, value, nullptr}; // Создаем новый узел

NodeMap<TK, TV>\* curr = map.data[index];

while (curr->next != nullptr) {

curr = curr->next; // Ищем последний узел в цепочке

}

curr->next = newNode; // Добавляем новый узел в конец цепочки

}

// Добавление элементов

template <typename TK, typename TV>

void AddMap(MyHashMap<TK, TV>& map, const TK& key, const TV& value) {

if ((map.length + 1) \* 100 / map.capacity >= map.LoadFactor) {

Expansion(map); // Если достигнут фактор загрузки, расширяем хэш-таблицу

}

int index = HashCode(key) % map.capacity; // Вычисляем индекс

NodeMap<TK, TV>\* temp = map.data[index];

if (temp != nullptr) {

if (temp->key == key) {

// Обновляем значение ключа

temp->value = value;

map.data[index] = temp;

} else {

CollisionManage(map, index, key, value); // Обрабатываем коллизию

}

} else {

NodeMap<TK, TV>\* newNode = new NodeMap<TK, TV>{key, value, map.data[index]}; // Создаем новый узел

map.data[index] = newNode;

map.length++; // Увеличиваем количество элементов

}

}

// Поиск элементов по ключу

template <typename TK, typename TV>

TV GetMap(const MyHashMap<TK, TV>& map, const TK& key) {

int index = HashCode(key) % map.capacity; // Вычисляем индекс

NodeMap<TK, TV>\* curr = map.data[index];

while (curr != nullptr) {

if (curr->key == key) {

return curr->value; // Возвращаем значение, если ключ найден

}

curr = curr->next;

}

throw std::runtime\_error("Ключ не найден"); // Выбрасываем ошибку, если ключ не найден

}

// Удаление элементов

template <typename TK, typename TV>

void DeleteMap(MyHashMap<TK, TV>& map, const TK& key) {

int index = HashCode(key) % map.cap; // Вычисляем индекс

NodeMap<TK, TV>\* curr = map.data[index];

NodeMap<TK, TV>\* prev = nullptr;

while (curr != nullptr) {

if (curr->key == key) {

if (prev == nullptr) {

map.data[index] = curr->next; // Удаляем первый элемент в цепочке

} else {

prev->next = curr->next; // Удаляем элемент из середины или конца цепочки

}

delete curr; // Освобождаем память

map.len--; // Уменьшаем количество элементов

return;

}

prev = curr;

curr = curr->next;

}

throw std::runtime\_error("Ключ не найден"); // Выбрасываем ошибку, если ключ не найден

}

// Очистка памяти

template <typename TK, typename TV>

void DestroyMap(MyHashMap<TK, TV>& map) {

for (int i = 0; i < map.capacity; i++) {

NodeMap<TK, TV>\* curr = map.data[i];

while (curr != nullptr) {

NodeMap<TK, TV>\* next = curr->next;

delete curr; // Освобождаем память каждого узла

curr = next;

}

}

delete[] map.data; // Освобождаем память массива указателей

map.data = nullptr;

map.length = 0;

map.capacity = 0;

}

#endif

**MyVector.hpp:**  
#ifndef MYVECTOR\_H

#define MYVECTOR\_H

#include <iostream>

#include <iomanip>

template <typename T>

struct MyVector {

T\* data; // Массив

int length; // Длина

int capacity; // Вместимость (capacity)

int LoadFactor; // Процент заполнения, при котором увеличиваем объем (например, 50%)

};

// Перегрузка оператора вывода

template <typename T>

std::ostream& operator << (std::ostream& os, const MyVector<T>& vector) {

for (int i = 0; i < vector.length; i++) {

std::cout << vector.data[i];

if (i < vector.length - 1) std::cout << std::setw(25); // Устанавливаем ширину для вывода

}

return os;

}

// Инициализация вектора

template <typename T>

MyVector<T>\* CreateVector(int initCapacity, int initLoadFactor) {

if (initCapacity <= 0 || initLoadFactor <= 0 || initLoadFactor > 100) {

throw std::runtime\_error("Индекс вне диапазона"); // Выбрасываем ошибку, если параметры некорректны

}

MyVector<T>\* vector = new MyVector<T>; // Создаем новый вектор

vector->data = new T[initCapacity]; // Выделяем память под массив

vector->length = 0; // Инициализируем длину

vector->capacity = initCapacity; // Устанавливаем вместимость

vector->LoadFactor = initLoadFactor; // Устанавливаем фактор загрузки

return vector;

}

// Увеличение массива

template <typename T>

void Expansion(MyVector<T>& vector) {

int newCap = vector.capacity \* 2; // Новая вместимость в два раза больше текущей

T\* newData = new T[newCap]; // Выделяем память под новый массив

for (int i = 0; i < vector.length; i++) { // Копируем данные из старого массива в новый

newData[i] = vector.data[i];

}

delete[] vector.data; // Очищаем память старого массива

vector.data = newData;

vector.capacity = newCap;

}

// Добавление элемента в вектор

template <typename T>

void AddVector(MyVector<T>& vector, T value) {

if ((vector.length + 1) \* 100 / vector.capacity >= vector.LoadFactor) { // Если достигнут фактор загрузки, увеличиваем массив

Expansion(vector);

}

vector.data[vector.length] = value; // Добавляем элемент

vector.length++; // Увеличиваем длину

}

// Удаление элемента из вектора

template <typename T>

void DeleteVector(MyVector<T>& vector, int index) {

if (index < 0 || index >= vector.length) {

throw std::runtime\_error("Индекс вне диапазона"); // Выбрасываем ошибку, если индекс некорректен

}

for (int i = index; i < vector.length - 1; i++) {

vector.data[i] = vector.data[i + 1]; // Сдвигаем элементы влево

}

vector.length--; // Уменьшаем длину

}

// Замена элемента по индексу

template <typename T>

void ReplaceVector(MyVector<T>& vector, int index, T value) {

if (index < 0 || index >= vector.length) {

throw std::runtime\_error("Индекс вне диапазона"); // Выбрасываем ошибку, если индекс некорректен

}

vector.data[index] = value; // Заменяем элемент

}

#endif

**WhereValue.hpp:**  
#ifndef WHEREVALUE\_HPP

#define WHEREVALUE\_HPP

#include <iostream>

#include <string>

#include "../CustomStructures/MyHashMap.hpp"

#include "../CustomStructures/MyVector.hpp"

#include "../Other/Utilities.hpp"

using namespace std;

// Определение типа узла

enum class NodeType {

ConditionNode, // Узел условия

OrNode, // Узел логического ИЛИ

AndNode // Узел логического И

};

// Структура узла

struct Node {

NodeType nodeType; // Тип узла

MyVector<std::string> value; // Значение узла

Node\* left; // Левый потомок

Node\* right; // Правый потомок

// Конструктор узла

Node(NodeType type, const MyVector<std::string> val = {}, Node\* l = nullptr, Node\* r = nullptr)

: nodeType(type), value(val), left(l), right(r) {}

};

// Функция для удаления апострофов в начале и конце строки

std::string SanitizeText(std::string str) {

if (str[0] == '\'' && str[str.size() - 1] == '\'') {

str = getSubstring(str, 1, str.size() - 1); // Удаляем апострофы

return str;

} else {

throw std::runtime\_error("Неверный синтаксис в WHERE " + str); // Выбрасываем ошибку, если синтаксис неверный

}

}

// Вспомогательная функция для разделения строки по оператору

MyVector<MyVector<std::string>\*>\* splitByOperator(const MyVector<std::string>& query, const std::string& op) {

MyVector<std::string>\* left = CreateVector<std::string>(6, 50); // Создаем вектор для левой части

MyVector<std::string>\* right = CreateVector<std::string>(6, 50); // Создаем вектор для правой части

bool afterOp = false; // Флаг, указывающий на то, что оператор уже был встречен

for (int i = 0; i < query.length; i++) {

if (query.data[i] == op) {

afterOp = true; // Устанавливаем флаг, если встретили оператор

} else if (afterOp) {

AddVector(\*right, query.data[i]); // Добавляем элемент в правую часть

} else {

AddVector(\*left, query.data[i]); // Добавляем элемент в левую часть

}

}

MyVector<MyVector<std::string>\*>\* parseVector = CreateVector<MyVector<std::string>\*>(5, 50); // Создаем вектор для результата

if (afterOp) {

AddVector(\*parseVector, left); // Добавляем левую часть в результат

AddVector(\*parseVector, right); // Добавляем правую часть в результат

} else {

AddVector(\*parseVector, left); // Добавляем только левую часть, если оператор не был найден

}

return parseVector;

}

// Функция для построения дерева условий

Node\* getConditionTree(const MyVector<std::string>& query) {

MyVector<MyVector<std::string>\*>\* orParts = splitByOperator(query, "OR"); // Разделяем строку по оператору OR

// Если найден оператор OR

if (orParts->length > 1) {

Node\* root = new Node(NodeType::OrNode); // Создаем узел OR

root->left = getConditionTree(\*orParts->data[0]); // Рекурсивно строим левое поддерево

root->right = getConditionTree(\*orParts->data[1]); // Рекурсивно строим правое поддерево

return root;

}

// Если найден оператор AND

MyVector<MyVector<std::string>\*>\* andParts = splitByOperator(query, "AND"); // Разделяем строку по оператору AND

if (andParts->length > 1) {

Node\* root = new Node(NodeType::AndNode); // Создаем узел AND

root->left = getConditionTree(\*andParts->data[0]); // Рекурсивно строим левое поддерево

root->right = getConditionTree(\*andParts->data[1]); // Рекурсивно строим правое поддерево

return root;

}

// Если это простое условие

return new Node(NodeType::ConditionNode, query); // Создаем узел условия

}

// Функция для проверки, удовлетворяет ли строка условию

bool isValidRow(Node\* node, const MyVector<std::string>& row, const MyHashMap<std::string, MyVector<std::string>\*>& jsonStructure, const std::string& namesOfTable) {

if (!node) {

return false; // Если узел пустой, возвращаем false

}

switch (node->nodeType) {

case NodeType::ConditionNode: {

if (node->value.length != 3) {

return false; // Если условие не состоит из трех частей, возвращаем false

}

MyVector<std::string> \*part1Splitted = splitRow(node->value.data[0], '.'); // Разделяем первую часть условия по точке

if (part1Splitted->length != 2) {

return false; // Если разделение не дало две части, возвращаем false

}

// Проверяем, существует ли запрашиваемая таблица

int columnIndex = -1;

try {

MyVector<std::string>\* colNames = GetMap(jsonStructure, part1Splitted->data[0]); // Получаем имена столбцов таблицы

for (int i = 0; i < colNames->length; i++) {

if (colNames->data[i] == part1Splitted->data[1]) {

columnIndex = i; // Находим индекс столбца

break;

}

}

} catch (const std::exception& err) {

std::cerr << err.what() << ": Tаблица " << part1Splitted->data[0] << " отсутствует" << std::endl; // Выводим ошибку, если таблица отсутствует

return false;

}

if (columnIndex == -1) {

std::cerr << "Столбец " << part1Splitted->data[1] << " отсутствует в таблице " << part1Splitted->data[0] << std::endl; // Выводим ошибку, если столбец отсутствует

return false;

}

std::string delApostr = SanitizeText(node->value.data[2]); // Удаляем апострофы из значения условия

if (namesOfTable == part1Splitted->data[0] && row.data[columnIndex + 1] == delApostr) {

return true; // Если строка удовлетворяет условию, возвращаем true

}

return false; // В противном случае возвращаем false

}

case NodeType::OrNode:

return isValidRow(node->left, row, jsonStructure, namesOfTable) || // Проверяем левое поддерево

isValidRow(node->right, row, jsonStructure, namesOfTable); // Проверяем правое поддерево

case NodeType::AndNode:

return isValidRow(node->left, row, jsonStructure, namesOfTable) && // Проверяем левое поддерево

isValidRow(node->right, row, jsonStructure, namesOfTable); // Проверяем правое поддерево

default:

return false; // По умолчанию возвращаем false

}

}

#endif

**SelectValue.hpp:**  
#ifndef SELECTVALUE\_HPP

#define SELECTVALUE\_HPP

#include <iostream>

#include <fstream>

#include <filesystem>

#include "../CustomStructures/MyHashMap.hpp"

#include "../CustomStructures/MyVector.hpp"

#include "../Other/Utilities.hpp"

#include "WhereValue.hpp"

using namespace std;

// Функция для чтения таблицы из файла

MyVector<MyVector<string>\*>\* ReadTable(const string& namesOfTable, const string& namesOfSchema, const string& pathFile, const MyVector<string>& namesOfColumns, const MyVector<string>& listOfCondition, const MyHashMap<string, MyVector<string>\*>& jsonStructure, bool whereValue) {

// Создание вектора для хранения данных таблицы

MyVector<MyVector<string>\*>\* dataOfTable = CreateVector<MyVector<string>\*>(5, 50);

int indexFile = 1;

try {

// Блокировка таблицы для чтения

CheckTableLock(pathFile + "/" + namesOfSchema + "/" + namesOfTable, namesOfTable + "\_lock.txt", 1);

} catch (const std::exception& e) {

// Вывод ошибки, если не удалось заблокировать таблицу

cerr << e.what() << endl;

return dataOfTable;

}

// Получение дерева условий для whereValue

Node\* nodeWere = getConditionTree(listOfCondition);

// Цикл по всем файлам таблицы

while (filesystem::exists(pathFile + "/" + namesOfSchema + "/" + namesOfTable + "/" + to\_string(indexFile) + ".csv")) {

// Открытие файла для чтения

ifstream file(pathFile + "/" + namesOfSchema + "/" + namesOfTable + "/" + to\_string(indexFile) + ".csv");

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

// Выброс исключения, если файл не открылся

throw runtime\_error("Ошибка открытия файла" + (pathFile + "/" + namesOfSchema + "/" + namesOfTable + "/" + to\_string(indexFile) + ".csv"));

}

string firstLine;

// Чтение первой строки файла (заголовка)

getline(file, firstLine);

// Если выбраны все столбцы

if (namesOfColumns.data[0] == "\*") {

string line;

// Чтение всех строк файла

while (getline(file, line)) {

// Разделение строки на столбцы

MyVector<string>\* row = splitRow(line, ',');

if (whereValue) {

try {

// Проверка строки на соответствие условиям whereValue

if (isValidRow(nodeWere, \*row, jsonStructure, namesOfTable)) {

// Удаление первого столбца (обычно это ID)

DeleteVector<string>(\*row, 0);

// Добавление строки в результат

AddVector(\*dataOfTable, row);

}

} catch (const exception& e) {

// Вывод ошибки и закрытие файла

cerr << e.what() << endl;

file.close();

return dataOfTable;

}

} else {

// Удаление первого столбца (обычно это ID)

DeleteVector<string>(\*row, 0);

// Добавление строки в результат

AddVector(\*dataOfTable, row);

}

}

} else {

// Получение имен столбцов из JSON структуры

MyVector<string>\* filenamesOfColumns = GetMap<string, MyVector<string>\*>(jsonStructure, namesOfTable);

// Создание вектора для хранения индексов выбранных столбцов

MyVector<int>\* colIndex = CreateVector<int>(10, 50);

// Заполнение вектора индексов выбранных столбцов

for (int i = 0; i < filenamesOfColumns->length; i++) {

for (int j = 1; j < namesOfColumns.length; j++) {

if (filenamesOfColumns->data[i] == namesOfColumns.data[j]) {

AddVector(\*colIndex, i + 1);

}

}

}

string line;

// Чтение всех строк файла

while (getline(file, line)) {

// Разделение строки на столбцы

MyVector<string>\* row = splitRow(line, ',');

if (whereValue) {

try {

// Проверка строки на соответствие условиям whereValue

if (isValidRow(nodeWere, \*row, jsonStructure, namesOfTable)) {

// Создание новой строки с выбранными столбцами

MyVector<string>\* newRow = CreateVector<string>(colIndex->length, 50);

for (int i = 0; i < colIndex->length; i++) {

AddVector(\*newRow, row->data[colIndex->data[i]]);

}

// Добавление новой строки в результат

AddVector(\*dataOfTable, newRow);

}

} catch (const exception& e) {

// Вывод ошибки и закрытие файла

cerr << e.what() << endl;

file.close();

return dataOfTable;

}

} else {

// Создание новой строки с выбранными столбцами

MyVector<string>\* newRow = CreateVector<string>(colIndex->length, 50);

for (int i = 0; i < colIndex->length; i++) {

AddVector(\*newRow, row->data[colIndex->data[i]]);

}

// Добавление новой строки в результат

AddVector(\*dataOfTable, newRow);

}

}

}

// Закрытие файла

file.close();

// Переход к следующему файлу

indexFile += 1;

}

// Разблокировка таблицы

CheckTableLock(pathFile + "/" + namesOfSchema + "/" + namesOfTable, namesOfTable + "\_lock.txt", 1);

// Возврат данных таблицы

return dataOfTable;

}

// Вывод содержимого таблиц в виде декартового произведения

void CartesianProduct(const MyVector<MyVector<MyVector<string>\*>\*>& tablesData, MyVector<MyVector<string>\*>& temp, int counterTab, int tab) {

// Цикл по всем строкам текущей таблицы

for (int i = 0; i < tablesData.data[counterTab]->length; i++) {

// Добавление текущей строки в временный вектор

temp.data[counterTab] = tablesData.data[counterTab]->data[i];

// Рекурсивный вызов для следующей таблицы

if (counterTab < tab - 1) {

CartesianProduct(tablesData, temp, counterTab + 1, tab);

} else {

// Вывод декартового произведения

for (int j = 0; j < tab; j++) {

cout << \*temp.data[j] << std::setw(25);

}

cout << endl;

}

}

return;

}

// Подготовка к чтению и выводу данных

void selectDataPreparation(const MyVector<string>& namesOfColumns, const MyVector<string>& namesOfTables, const MyVector<string>& listOfCondition, const string& namesOfSchema, const string& pathFile, const MyHashMap<string, MyVector<string>\*>& jsonStructure, bool whereValue) {

// Создание вектора для хранения данных таблиц

MyVector<MyVector<MyVector<string>\*>\*>\* tablesData = CreateVector<MyVector<MyVector<string>\*>\*>(10, 50);

// Если выбраны все столбцы

if (namesOfColumns.data[0] == "\*") {

// Чтение всех данных из таблиц

for (int j = 0; j < namesOfTables.length; j++) {

MyVector<MyVector<string>\*>\* tableData = ReadTable(namesOfTables.data[j], namesOfSchema, pathFile, namesOfColumns, listOfCondition, jsonStructure, whereValue);

AddVector(\*tablesData, tableData);

}

} else {

// Чтение данных из выбранных столбцов

for (int i = 0; i < namesOfTables.length; i++) {

MyVector<string>\* tabColPair = CreateVector<string>(5, 50);

AddVector(\*tabColPair, namesOfTables.data[i]);

for (int j = 0; j < namesOfColumns.length; j++) {

MyVector<string>\* splitnamesOfColumns = splitRow(namesOfColumns.data[j], '.');

try {

GetMap(jsonStructure, splitnamesOfColumns->data[0]);

} catch (const exception& e) {

cerr << e.what() << ": Tаблица " << splitnamesOfColumns->data[0] << " отсутствует" << endl;

return;

}

if (splitnamesOfColumns->data[0] == namesOfTables.data[i]) {

AddVector(\*tabColPair, splitnamesOfColumns->data[1]);

}

}

MyVector<MyVector<string>\*>\* tableData = ReadTable(tabColPair->data[0], namesOfSchema, pathFile, \*tabColPair, listOfCondition, jsonStructure, whereValue);

AddVector(\*tablesData, tableData);

}

}

// Создание временного вектора для хранения строк

MyVector<MyVector<string>\*>\* temp = CreateVector<MyVector<string>\*>(tablesData->length \* 2, 50);

// Вывод декартового произведения

CartesianProduct(\*tablesData, \*temp, 0, tablesData->length);

}

// Парсинг SELECT запроса

void parseSelect(const MyVector<string>& words, const string& pathFile, const string& namesOfSchema, const MyHashMap<string, MyVector<string>\*>& jsonStructure) {

// Создание векторов для хранения имен столбцов, таблиц и условий whereValue

MyVector<string>\* namesOfColumns = CreateVector<string>(10, 50);

MyVector<string>\* namesOfTables = CreateVector<string>(10, 50);

MyVector<string>\* listOfCondition = CreateVector<string>(10, 50);

bool afterFrom = false;

bool afterwhereValue = false;

int countTabNames = 0;

int countData = 0;

int countWhereValueData = 0;

// Цикл по всем словам запроса

for (int i = 1; i < words.length; i++) {

// Удаление запятой в конце слова, если она есть

if (words.data[i][words.data[i].size() - 1] == ',') {

words.data[i] = getSubstring(words.data[i], 0, words.data[i].size() - 1);

}

// Определение разделителей в запросе

if (words.data[i] == "FROM") {

afterFrom = true;

} else if (words.data[i] == "whereValue") {

afterwhereValue = true;

} else if (afterwhereValue) {

// Добавление условий whereValue в список

countWhereValueData++;

AddVector<string>(\*listOfCondition, words.data[i]);

} else if (afterFrom) {

// Проверка наличия таблицы в JSON структуре

try {

GetMap(jsonStructure, words.data[i]);

} catch (const exception& e) {

cerr << e.what() << ": Tаблица" << words.data[i] << " отсутствует" << endl;

return;

}

// Добавление имени таблицы в список

countTabNames++;

AddVector(\*namesOfTables, words.data[i]);

} else {

// Добавление имен столбцов в список

countData++;

AddVector(\*namesOfColumns, words.data[i]);

}

}

// Проверка наличия имен таблиц и столбцов

if (countTabNames == 0 || countData == 0) {

throw runtime\_error("Отсутствует имя таблицы или данные в FROM");

}

// Вызов функции подготовки к чтению данных

if (countWhereValueData == 0) {

selectDataPreparation(\*namesOfColumns, \*namesOfTables, \*listOfCondition, namesOfSchema, pathFile, jsonStructure, false);

} else {

selectDataPreparation(\*namesOfColumns, \*namesOfTables, \*listOfCondition, namesOfSchema, pathFile, jsonStructure, true);

}

}

#endif

**InsertValue.hpp:**  
#ifndef INSERTVALUE\_HPP

#define INSERTVALUE\_HPP

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <string>

#include "../Other/Utilities.hpp"

#include "../CustomStructures/MyHashMap.hpp"

#include "../CustomStructures/MyVector.hpp"

using namespace std;

// Функция для удаления апострофов из строки

string CleanText(string& str) {

// Удаление запятой и скобки в конце строки

if (str[str.size() - 1] == ',' && str[str.size() - 2] == ')') {

str = getSubstring(str, 0, str.size() - 2);

} else if (str[str.size() - 1] == ',' || str[str.size() - 1] == ')') {

str = getSubstring(str, 0, str.size() - 1);

}

// Удаление апострофов в начале и конце строки

if (str[0] == '\'' && str[str.size() - 1] == '\'') {

str = getSubstring(str, 1, str.size() - 1);

return str;

} else {

throw runtime\_error("Неверный синтаксис в VALUES " + str);

}

}

// Функция для проверки количества аргументов относительно столбцов таблиц

void Validate(int colLen, const MyVector<string>& namesOfTable, const MyHashMap<string, MyVector<string>\*>& jsonStructure) {

for (int i = 0; i < namesOfTable.length; i++) {

MyVector<string>\* temp = GetMap<string, MyVector<string>\*>(jsonStructure, namesOfTable.data[i]);

if (temp->length != colLen) {

throw runtime\_error("Количество аргументов не равно столбцам в " + namesOfTable.data[i]);

}

}

}

// Функция для чтения или записи первичного ключа

int readPrKey(const string& path, const bool rec, const int newID) {

fstream pkFile(path);

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

throw runtime\_error("Не удалось открыть " + path);

}

int lastID = 0;

if (rec) {

pkFile << newID;

} else {

pkFile >> lastID;

}

pkFile.close();

return lastID;

}

// Функция для добавления строк в файл

void insertRows(MyVector<MyVector<string>\*>& addNewData, MyVector<string>& namesOfTable, const string& nameOfSchema, const int limitOfTuples, const string& filePath) {

for (int i = 0; i < namesOfTable.length; i++) {

int lastID = 0;

try {

CheckTableLock(filePath + "/" + nameOfSchema + "/" + namesOfTable.data[i], namesOfTable.data[i] + "\_lock.txt", 1);

lastID = readPrKey(filePath + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + namesOfTable.data[i] + "\_pk\_sequence.txt", false, 0);

} catch (const std::exception& err) {

cerr << err.what() << endl;

return;

}

int newID = lastID;

for (int j = 0; j < addNewData.length; j++) {

newID++;

string tempPath;

if (lastID / limitOfTuples < newID / limitOfTuples) {

tempPath = filePath + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(newID / limitOfTuples + 1) + ".csv";

} else {

tempPath = filePath + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(lastID / limitOfTuples + 1) + ".csv";

}

fstream csvFile(tempPath, ios::app);

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

throw runtime\_error("Не удалось открыть " + tempPath);

}

csvFile << endl << newID;

for (int k = 0; k < addNewData.data[j]->length; k++) {

csvFile << "," << addNewData.data[j]->data[k];

}

csvFile.close();

}

readPrKey(filePath + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + namesOfTable.data[i] + "\_pk\_sequence.txt", true, newID);

CheckTableLock(filePath + "/" + nameOfSchema + "/" + namesOfTable.data[i], namesOfTable.data[i] + "\_lock.txt", 0);

}

}

// Функция для парсинга команды INSERT

void parseInsert(const MyVector<string>& slovs, const string& filePath, const string& nameOfSchema, const int limitOfTuples, const MyHashMap<string, MyVector<string>\*>& jsonStructure) {

MyVector<string>\* targetTables = CreateVector<string>(5, 50);

MyVector<MyVector<string>\*>\* dataToInsert = CreateVector<MyVector<string>\*>(10, 50);

bool afterValues = false;

int countOfTable = 0;

int dataCount = 0;

for (int i = 2; i < slovs.length; i++) {

if (slovs.data[i][slovs.data[i].size() - 1] == ',') {

slovs.data[i] = getSubstring(slovs.data[i], 0, slovs.data[i].size() - 1);

}

if (slovs.data[i] == "VALUES") {

afterValues = true;

} else if (afterValues) {

dataCount++;

if (slovs.data[i][0] == '(') {

MyVector<string>\* tempData = CreateVector<string>(5, 50);

slovs.data[i] = getSubstring(slovs.data[i], 1, slovs.data[i].size());

while (slovs.data[i][slovs.data[i].size() - 1] != ')' && slovs.data[i][slovs.data[i].size() - 2] != ')') {

try {

CleanText(slovs.data[i]);

} catch (const exception& err) {

cerr << " 23" << err.what() << slovs.data[i] << endl;

return;

}

AddVector<string>(\*tempData, slovs.data[i]);

i++;

}

try {

CleanText(slovs.data[i]);

AddVector<string>(\*tempData, slovs.data[i]);

Validate(tempData->length, \*targetTables, jsonStructure);

} catch (const exception& err) {

cerr << err.what() << endl;

return;

}

AddVector<MyVector<string>\*>(\*dataToInsert, tempData);

}

} else {

countOfTable++;

try {

GetMap(jsonStructure, slovs.data[i]);

} catch (const exception& err) {

cerr << err.what() << ": Таблица " << slovs.data[i] << " отсутствует" << endl;

return;

}

AddVector<string>(\*targetTables, slovs.data[i]);

}

}

if (countOfTable == 0 || dataCount == 0) {

throw runtime\_error("Отсутствует имя таблицы или данные в VALUES");

}

try {

insertRows(\*dataToInsert, \*targetTables, nameOfSchema, limitOfTuples, filePath);

} catch (const exception& err) {

cerr << err.what() << endl;

return;

}

}

#endif // INSERT\_HPP

**DeleteValue.hpp:**  
#ifndef DELETEVALUE\_HPP

#define DELETEVALUE\_HPP

#include <iostream>

#include <fstream>

#include <filesystem>

#include <string>

#include "../CustomStructures/MyVector.hpp"

#include "../CustomStructures/MyHashMap.hpp"

#include "../Other/Utilities.hpp"

#include "WhereValue.hpp"

using namespace std;

// Функция для удаления данных из таблиц

inline void removeData(MyVector<string>& namesOfTable, MyVector<string>& listOfCondition, const string& nameOfSchema, const string& path, const MyHashMap<string, MyVector<string>\*>& jsonStructure) {

// Создаем дерево условий для фильтрации строк

Node\* nodeWere = getConditionTree(listOfCondition);

// Проходим по всем таблицам, указанным в запросе

for (int i = 0; i < namesOfTable.length; i++) {

int fileIndex = 1;

// Блокируем таблицу для исключения конфликтов при записи

try {

CheckTableLock(path + "/" + nameOfSchema + "/" + namesOfTable.data[i], namesOfTable.data[i] + "\_lock.txt", 1);

} catch (const std::exception& err) {

cerr << err.what() << endl;

return;

}

// Проходим по всем файлам таблицы

while (filesystem::exists(path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + ".csv")) {

ifstream file(path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + ".csv");

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

throw runtime\_error("Ошибка открытия: " + (path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + ".csv"));

}

// Создаем временный файл для записи отфильтрованных данных

ofstream tempFile(path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + "\_temp.csv");

string line;

// Копируем заголовок таблицы во временный файл

getline(file, line);

tempFile << line;

// Обрабатываем каждую строку таблицы

while (getline(file, line)) {

MyVector<string>\* row = splitRow(line, ',');

try {

// Проверяем, соответствует ли строка условиям

if (!isValidRow(nodeWere, \*row, jsonStructure, namesOfTable.data[i])) {

tempFile << endl << line;

}

} catch (const exception& err) {

cerr << err.what() << endl;

tempFile.close();

file.close();

std::remove((path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + "\_temp.csv").c\_str());

return;

}

}

// Закрываем файлы

tempFile.close();

file.close();

// Удаляем исходный файл и переименовываем временный файл

if (std::remove((path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + ".csv").c\_str()) != 0) {

std::cerr << "Ошибка удаления файла" << std::endl;

return;

}

if (std::rename((path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + "\_temp.csv").c\_str(), (path + "/" + nameOfSchema + "/" + namesOfTable.data[i] + "/" + to\_string(fileIndex) + ".csv").c\_str()) != 0) {

std::cerr << "Ошибка присвоения названия файлу" << std::endl;

return;

}

fileIndex++;

}

// Разблокируем таблицу

CheckTableLock(path + "/" + nameOfSchema + "/" + namesOfTable.data[i], namesOfTable.data[i] + "\_lock.txt", 0);

}

}

// Функция для парсинга DELETE запроса

void parseDelete(const MyVector<string>& words, const string& filePath, const string& nameOfSchema, const MyHashMap<string, MyVector<string>\*>& jsonStructure) {

// Создаем векторы для хранения имен таблиц и условий

MyVector<string>\* namesOfTable = CreateVector<string>(5, 50);

MyVector<string>\* listOfCondition = CreateVector<string>(5, 50);

int countTabNames = 0;

int countWereData = 0;

bool afterWhere = false;

// Проходим по всем словам в запросе

for (int i = 2; i < words.length; i++ ) {

// Убираем запятые в конце имен таблиц

if (words.data[i][words.data[i].size() - 1] == ',') {

words.data[i] = getSubstring(words.data[i], 0, words.data[i].size() - 1);

}

// Определяем, где начинаются условия

if (words.data[i] == "WHERE") {

afterWhere = true;

} else if (afterWhere) {

// Добавляем условия в список

AddVector<string>(\*listOfCondition, words.data[i]);

countWereData++;

} else {

// Добавляем имена таблиц в список

countTabNames++;

try {

GetMap(jsonStructure, words.data[i]);

} catch (const exception& err) {

cerr << err.what() << ": таблица " << words.data[i] << " отсутствует." << endl;

return;

}

AddVector<string>(\*namesOfTable, words.data[i]);

}

}

// Проверяем, что указаны и таблицы, и условия

if (countTabNames == 0 || countWereData == 0) {

throw runtime\_error("Отсутствует имя таблицы или данные в WHERE");

}

// Вызываем функцию для удаления данных

try {

removeData(\*namesOfTable, \*listOfCondition, nameOfSchema, filePath, jsonStructure);

} catch (const exception& err) {

cerr << err.what()<< endl;

return;

}

}

#endif

**schema.json:**  
  
{

"name": "BigStore",

"tuples\_limit": 30,

"structure": {

"beer": ["Name", "Type", "Brewery", "Country"],

"cars": ["Make", "Model", "Year", "Price"],

"test": ["col1", "col2"]

}

}

**Результаты работы:**









