МИНИСТЕРСТВО ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ»
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет №8
«Компьютерные науки и прикладная математика»
Кафедра 806
«Вычислительная математика и программирование»

Направление подготовки

02.03.02 Фундаментальная информатика и информационные технологии

Курсовой проект по операционным системам и архитектуре компьютера
«Разработка алгоритмов системы хранения и управления данными на основе динамических структур данных»

Преподаватель: Романенков А.М.

Выполнил: Батоян Р.Л.

Группа: М8О-211Б-22

Оценка:

Дата:

Москва, 2024

СОДЕРЖАНИЕ

Введение 3
Задание 0 5
Задание 1 14
Задание 4 15
Задание 5 18
Задание 7 20
Задание 8. 21
Тестирование кода 25
Заключение 27
Список использованных источников 29
Приложение 30

Введение

Целью данного курсового проекта является разработка приложения на языке программирования C++ (стандарт C++17), которое позволяет выполнять операции над коллекциями данных заданных типов и контекстами их хранения. Приложение должно обеспечивать работу с данными через ассоциативные контейнеры типа B-tree и представлять эти данные на различных уровнях хранения, таких как пулы схем данных, схемы данных и коллекции данных.

В современном мире базы данных играют ключевую роль во всех сферах деятельности человека, обеспечивают эффективное хранение, управление и доступ к огромным объёмам информации. Они позволяют организациям хранить важные данные, обеспечивать их целостность и конфиденциальность, а также проводить анализ данных для принятия обоснованных решений.

Современные информационные системы требуют высокоэффективных методов обработки и управления данными. Применение ассоциативных контейнеров B-tree предоставляет возможность быстрого доступа к данным благодаря своей структурированной организации, что делает их идеальными для реализации задач, связанных со сложными операциями манипуляции данными. Такое приложение будет полезно для анализа, хранения и обработки больших объёмов данных в различных сферах, от финансовой аналитики до научных исследований.

В рамках задачи курсового проекта перед разработкой приложения стоят следующие задачи:

1. Разработка интерфейса ассоциативного контейнера в виде абстрактного класса C++, а также его реализация. Это включает в себя применение приёмов объектно-ориентированного проектирования, описанных в работе Гамма и др. [1]. Реализация основных операций над коллекциями данных, таких как добавление, чтение, обновление и удаление записей по ключу.
2. Поддержка операций над коллекциями данных, включающих добавление или удаление пулов данных, схем данных и коллекций данных.
3. Обеспечение взаимодействия с приложением посредством выполнения команд, поступающих из файла, путь к которому передаётся через аргумент командной строки.

4. Работа приложения в двух режимах: размещение структур и данных в оперативной памяти (in-memory cache) и файловой системе, с обеспечением надёжного хранения данных.

Базы данных необходимы для обеспечения надёжного и быстрого хранения информации, что позволяет мгновенно получать доступ к данным, независимо от их объёма. Использование передовых технологий и методов в области баз данных, таких как B-tree, обеспечивает высокую производительность и надёжность систем, что особенно важно в условиях высокой нагрузки и необходимости быстрого реагирования на изменения.

Разработка собственной базы данных с поддержкой различных уровней хранения данных и механизмов управления ими является важным шагом в изучении принципов организации информационных систем и получении практического опыта в области программирования и алгоритмов управления данными. Это не только позволит освоить практическое применение различных структур данных и алгоритмов, но и углубит понимание архитектурных решений, необходимых для создания надёжных и масштабируемых приложений.

Таким образом, данный курсовой проект представляет собой не только учебное задание, но и значимый вклад в понимание и развитие технологий управления данными, которые являются основополагающими для современной информационной индустрии.

Задание 0

Приложение представляет собой базу данных, разработанную на языке программирования C++ и поддерживающую ассоциативные контейнеры типа B-tree для структур данных. Взаимодействие с этими контейнерами осуществляется через абстрактный интерфейс storage_interface. Основными элементами базы данных являются пулы схем данных, схемы данных и коллекции данных, организованные иерархически. Каждый объект данных имеет уникальный ключ.

Класс data_base является основным интерфейсом взаимодействия с базой данных и наследует интерфейс storage_interface. Он предоставляет методы для выполнения основных операций с базой данных.

Основные методы класса data_base:

- save_data_base_state: сохраняет текущее состояние базы данных. Метод обеспечивает сохранность данных, записывая их в файл или другое постоянное хранилище. Используется при стратегии хранения in_memory.
- load_data_base_state: Загружает состояние базы данных из файла или другого постоянного хранилища. Функция восстанавливает данные при запуске программы в режиме in_memory, обеспечивая их доступность после перезапуска приложения.
- start_console_dialog: запускает консольный интерфейс для взаимодействия с пользователем. В данной функции реализован ввод и обработка команд в реальном времени, что позволяет пользователю взаимодействовать с базой данных через консоль.
- execute_command_from_file: выполняет команды, указанные в файле, путь к которому передаётся как аргумент командной строки. Предназначен для автоматического выполнения команд, что позволяет реализовывать предопределённые сценарии работы с базой данных.
- insert_schemas_pool: вставляет новый пул схем в базу данных. Поддерживает вставку по значению и по rvalue ссылке.
- insert_schema: вставляет новую схему в указанный пул. Поддерживает вставку по значению и по rvalue ссылке.
- insert_table: вставляет новую таблицу в указанную схему. Метод поддерживает как вставку по значению, так и по rvalue ссылке.
- obtain_data: извлекает данные по заданному ключу.
- obtain_between_data: извлекает набор данных, ключи которых лежат в заданном диапазоне.
- update_data: обновляет данные для записи по заданному ключу.

- dispose методы: удаляют пулы данных, схемы данных, таблицы и данные по ключу.

Абстрактный класс storage_interface определяет общие операции для всех реализаций хранилища данных. Он предоставляет интерфейс для вставки, получения, обновления и удаления данных в ассоциативном контейнере типа B-tree.

Перечень защищённых членов класса storage_interface:

- _additional_storage: строка, указывающая на использование дополнительного хранилища (in_memory_storage).
- max_data_length, index_item_max_length: ограничивают максимальную длину данных и элементов индекса.
- _file_format: формат файлов для хранения данных.
- _instance_name: имя экземпляра базы данных.
- _allocator, _logger: указатели на объекты управления памятью и логгирования.
- _storage_strategy: стратегия хранения (либо in_memory, либо filesystem).

Класс также определяет методы для настройки и получения текущей стратегии хранения.Таким образом, код представляет собой многофункциональную базу данных, поддерживающую как оперативное, так и файловое хранилище, объединяя все необходимые операции для работы с коллекциями данных.

Класс user_data стал одним из центральных компонентов системы, отвечая за хранение информации о пользователях. В данной реализации особое внимание было уделено правильному управлению памятью и эффективному обращению с строковыми данными посредством использования строкового пула. Класс предоставляет полный набор методов для манипуляции данными, включая конструкторы, операторы присваивания, а также специализированные методы для создания объектов из строковых данных. Здесь пример интерфейсной части.

Листинг 1. Внутренний класс класса user_data.

```
class ud_obj
{
    friend class user_data;
    size_t _id;
    size_t _name_ind;
    size_t _surname_ind;

    ud_obj() = default;
```

```
ud_obj(size_t id, std::string const &name, std::string const
&surname)
        : _id(id), _name_ind(string_pool::add_string(name, true)),
_surname_ind(string_pool::add_string(surname, true)) {}
};
private:
    ud_obj _current_state;
```

Существенной частью работы стало обеспечение эффективного взаимодействия пользователя с приложением. Для этого был реализован удобный консольный интерфейс, который позволяет вводить команды в интерактивном режиме, а также выполнять предопределённые сценарии посредством чтения команд из файлов. Этот подход значительно повысил удобство использования программы и упростил процесс её настройки и эксплуатации.

Итоговая реализация продемонстрировала высокую степень переносимости и предсказуемое поведение программы на различных платформах, что стало возможным благодаря применению стандартов C++17 и современным методам управления памятью. Соблюдение принципов объектно-ориентированного программирования и использование передовых подходов к разработке программного обеспечения сделали код более структурированным, устойчивым к изменениям в окружении выполнения, а также повысили его безопасность и эффективность.

Файловая система хранения реализована несколько иначе. Основная идея - создавать для каждой отдельной базы данных свою директорию, чтобы разделить контексты различных данных. Далее, в этой директории могут располагаться директории - имитация schemas_pool, внутри schemas_pool могут находиться директории schema, внутри schema - файлы table. Для каждой отдельной таблицы создаётся два файла - файл с данными и индекс файл, который предназначен для эффективного поиска по исходному файлу.

Основная идея - поддерживать отсортированный порядок записей в таблице, чтобы была возможность осуществлять бинарный поиск по всему файлу, что существенно влияет на время работы.

Так как для работы с таблицей приходится изменять существующие данные и работать с дополнительным файлом, то в связи с этим возникает необходимость в логике транзакции с возможностью возвращения к исходному состоянию в случае каких-либо ошибок в любом месте «чувствительных операций».

В моей реализации это достигается за счёт создания backup файлов, которые обеспечивают целостность и долговечность данных, не взирая на любые возникающие ошибки.

Листинг 2. Создание backup файла.

```
void table::create_backup(const std::filesystem::path &source_path)
{
    if (!std::filesystem::exists(source_path))
    {
        throw std::runtime_error("Source file does not exist: " +
source_path.string());
    }

    std::filesystem::path backup_path = source_path;
    backup_path += ".backup";

    std::ifstream src_orig(source_path, std::ios::binary);
    throw_if_not_open(src_orig);

    std::ofstream backup_file(backup_path, std::ios::trunc |
std::ios::binary);
    if (!backup_file.is_open())
    {
        src_orig.close();
        throw std::runtime_error("Failed to create backup file: " +
backup_path.string());
    }

    backup_file << src_orig.rdbuf();

    src_orig.close();
    backup_file.close();
}
```

Данная функция создаёт копию файла. Для каждой такой операции предусмотрена обратная - загрузка из бэкапа.В случае неудачи копия станет «основной» версией файла после вызова метода load_backup. Внутри этих функций так же предусмотрена логика возврата к текущему состоянию, то есть сохранение копии от копии, в связи с тем что операция удаления копии и переименования могут вызывать ошибки и генерировать исключения. Помимо этих двух методов есть третий - удаление backup файла в случае удачного выполнения операции. Данная логика обеспечивает устойчивость таблицы к неожиданным ошибкам.

Листинг 3. Загрузка backup файла.

```
void table::load_backup(const std::filesystem::path &source_path)
{
    std::filesystem::path backup_path = source_path;
    backup_path += ".backup";

    if (!std::filesystem::exists(backup_path))
    {
        throw std::runtime_error("Load backup file failed. Backup
file does not exist: " + backup_path.string());
    }
    try {
        std::filesystem::path temp_backup_path = backup_path;
        temp_backup_path += ".tmp";
        std::filesystem::copy(backup_path, temp_backup_path);

        if (std::filesystem::exists(source_path))
        {
            std::filesystem::remove(source_path);
        }

        std::filesystem::rename(temp_backup_path, source_path);
    }
    catch (...) {
        if (std::filesystem::exists(source_path))
        {
            std::filesystem::remove(source_path);
        }
        std::filesystem::path temp_backup_path = backup_path;
        temp_backup_path += ".tmp";
        if (std::filesystem::exists(temp_backup_path))
        {
            std::filesystem::rename(temp_backup_path, source_path);
        }
        throw;
    }

    std::filesystem::path temp_backup_path = backup_path;
    temp_backup_path += ".tmp";
    if (std::filesystem::exists(temp_backup_path))
    {
        std::filesystem::remove(temp_backup_path);
    }
}
```

Метод insert_ud_to_filesystem класса table отвечает за вставку объекта user_data в файловую систему. Входные параметры - это пути к основному файлу и индексному файлу, ключ и объект user_data. Рассмотрим подробно, что делает каждый блок кода:

Первым делом формируется строка out_str, которая представляет собой сериализованное представление объекта user_data. Строка дополняется до определенной длины. Далее выполняется проверка, существуют ли файлы по указанным путям, и если нет, то они создаются, и в них записывается out_str. Если файлы уже существуют, то в них дописывается out_str. Затем загружается индекс из индексного файла. Если индекс пуст, то в основной файл записывается out_str, а индекс обновляется и сохраняется в индексный файл. Если индекс не пуст, то начинается процесс поиска места для вставки новых данных. Сначала выполняется бинарный поиск по индексу для определения места вставки. Если найден ключ, который уже есть в файле, то выбрасывается исключение. Если место для вставки найдено, то создаются резервные копии основного и индексного файлов. Затем в основной файл дописывается out_str, а индекс обновляется и сохраняется в индексный файл.

Если в процессе работы происходит ошибка, то восстанавливаются резервные копии файлов, и выбрасывается исключение. В конце работы удаляются резервные копии файлов.

Методы dispose, obtain, obtain_between, update имеют следующее назначение:

- dispose: удаляет данные по указанному ключу. Если данные не найдены, то выбрасывается исключение.
- obtain: извлекает данные по указанному ключу. Если данные не найдены, то выбрасывается исключение.
- obtain_between: извлекает набор данных, ключи которых лежат в заданном диапазоне. Если данные не найдены, то выбрасывается исключение.
- update: обновляет данные по указанному ключу. Если данные не найдены, то выбрасывается исключение. Если данные успешно обновлены, то метод возвращает true, иначе false.

Логика остальных методов очень схожа со вставкой. Сначала выполняется бинарный поиск записи, к которой нужно применить операцию и далее применяется сама операция. Логика резервных копий сохраняется для методов, модифицирующих состояние файла - insert, dispose, update.

Листинг 4. Вставка данных в таблицу.

```
void table::insert_ud_to_filesystem(
    std::filesystem::path const &path,
    std::filesystem::path const &index_path,
    std::string const &key,
    user_data &&ud)
{
    std::string out_str =
std::to_string(string_pool::add_string(key)) + "#" +
std::to_string(ud.get_id()) + "#" + ud.get_name() + "#" +
ud.get_surname() + "|";
    length_alignment(out_str);

    if (check_and_create_with_insertion(path, index_path, out_str))
    {
        return;
    }

    auto index_array = load_index(index_path.string());

    if (index_array.empty()) {
        std::ofstream out_f(path);
        throw_if_not_open(out_f);
        std::ofstream index_f(index_path);
        if (!index_f.is_open())
        {
            out_f.close();
            throw_if_not_open(index_f);
        }
        out_f << out_str << std::endl;
        out_f.close();

        storage_interface::update_index(index_array);
        storage_interface::save_index(index_array, index_f);
        index_f.close();

        return;
    }

    std::ifstream src(path);
    throw_if_not_open(src);

    size_t left = 0;
    size_t right = index_array.size() - 1;
    std::string file_key;
```

```
while (left <= right) {
    size_t mid = left + (right - left) / 2;
    src.seekg(index_array[mid]);
    std::string key_index;
    std::getline(src, key_index, '#');
    file_key = string_pool::get_string(std::stol(key_index));
    if (file_key == key) {
        src.close();
     throw std::logic_error("duplicate key"); }
    if (right == left) {
        src.close();
        break; }
    if (file_key < key) {
        left = mid + 1; }
    else { right = mid; }
}
create_backup(path);
create_backup(index_path);
bool is_target_greater = file_key < key;
if (left == index_array.size() - 1 && is_target_greater) {
try {
    std::ofstream data_file(path, std::ios::app);
    throw_if_not_open(data_file);
    data_file << out_str << std::endl;
    data_file.close();
    update_index(index_array);
    save_index(index_array, index_path.string()); }
catch (...) {
    load_backup(path);
    load_backup(index_path);
    throw; }
delete_backup(path);
delete_backup(index_path);
return;
}
try {
std::ifstream data_file(path);
throw_if_not_open(data_file);
auto tmp_filename = path.string() + "_temp" + _file_format;
std::ofstream tmp_file(tmp_filename);
if (!tmp_file.is_open()) {
    data_file.close();
    throw_if_not_open(tmp_file); }
```

```
    std::string src_line;
    size_t pos;
    bool is_found_insertion_position = false;
    while (std::getline(data_file, src_line)) {
        if (!is_found_insertion_position) {
        pos = src_line.find('#');
        if (pos != std::string::npos)
        {
            std::string key_index = src_line.substr(0, pos);
            std::string current_key =
string_pool::get_string(std::stol(key_index));
            if (current_key == file_key)
            {
            is_found_insertion_position = true;
            if (is_target_greater) {
                tmp_file << src_line << std::endl;
                tmp_file << out_str << std::endl;}
            else{
                tmp_file << out_str << std::endl;
                tmp_file << src_line << std::endl;}
            continue;
            }}}
        tmp_file << src_line << std::endl;
    }

    update_index(index_array);
    save_index(index_array, index_path.string());
    data_file.close();
    tmp_file.close();

    std::filesystem::remove(path);
    std::filesystem::rename(tmp_filename, path);}
    catch (...){
    load_backup(index_path);
    load_backup(path);
    throw;}

    delete_backup(index_path);
    delete_backup(path);
}
```

Задание 1

Интерактивный диалог с пользователем реализован в методе класса data_base::start_console_dialog. Он обеспечивает простой способ взаимодействия пользователя с базой данных через консоль.

При вызове метода, программа предлагает пользователю ввести в консоль одну из нескольких команд, которые, в свою очередь, вызывают нужные методы из класса data_base. Доступны следующие команды для ввода пользователем:

1. insert_pool - вставка пула схем данных
2. insert_schema - вставка схемы данных
3. insert_table - вставка таблицы данных
4. insert_data - вставка данных о пользователе в таблицу
5. dispose_table - удаление таблицы схемы данных
6. dispose_schema - удаление схемы данных
7. dispose_pool - удаление пула схем данных
8. dispose_data - удаление пользователя по ключу из таблицы
9. obtain_data - поиск пользователя в таблице по ключу
10. obtain_between - поиск пользователей в диапазоне ключей
11. update_data - обновление данных о пользователе по ключу
12. execute_file_command - выполнение команд из файла
13. save_db_state - сохранение текущего состояния в файловую систему (доступно только в ржиме in_memory)
14. load_db_state - загрузка состояния базы данных из файловой системы в оперативную память (доступно только в режиме in_memory)
15. obtain_all - поиск всех пользователей в таблице (доступно только в режиме filesystem)
16. exit - выход из консольного режима

Метод запускает бесконечный цикл, считывает команду типа std::string и с помощью оператора if/else выполняет соответствующий метод, удовлетворяющий запросу пользователя. Цикл заканчивает работу, когда пользователь введет команду exit.

Задание 4

Хранение объектов строк осуществляется в классе string_pool. Имя, фамилия и ключ пользователя имеют уникальные ключи, что обеспечивает взаимно однозначное соответствие между множеством натуральных целочисленных значений и строками.

Листинг 5. Класс string_pool, основные поля.

```
class string_pool final
{
private:

    std::map<int64_t, std::shared_ptr<std::string>> _pool;

    static int64_t _pool_size;

    static string_pool *_instance;

    std::string _storage_filename = "string_pool.txt";

    std::fstream _file;

    static std::mutex _mutex;
}
```

Так как класс string_pool реализует паттерн проектирования singleton, описанный в работе Гамма и др. [1]., то в связи с этим нужно обеспечить единственность экземпляра объекта. В моем коде это достигается за счёт приватного конструктора. Сама инициализация объекта делегирована методу get_instance, который «лениво» инициализирует экземпляр класса. То есть создание объекта string_pool происходит при первом обращении к нему.

Листинг 6. Реализация паттерна singleton.

```
public:
    static string_pool *get_instance()
    {
        if (!_instance)
        {
            _instance = new string_pool();
        }

        return _instance;
    }
```

Сам string_pool имеет два основных статических метода - int add_string(std::string) и std::string const &get_string(size_t). То есть при запросе на добавление строки сначала проверяется пулл и возвращается индекс, если строка существует. В противном случае - создаётся новая и возвращается уже её индекс. Индекс всех добавленных в пул строк остаётся неизменным на протяжении всего жизненного цикла статического экземпляра класса.

Основное использование этих двух основных методов - в классе user_data. То есть поля name , surname хранятся как индексы в целочисленном типе, и если нужно получить саму строчку или добавить новую, то обращению к пулу - обязательная операция.

Листинг 7. Реализация методов user_data через string_pool.

```
user_data::user_data(size_t id, const std::string &name, const
std::string &surname)
{
    _current_state._id = id;
    _current_state._name_ind = string_pool::add_string(name);
    _current_state._surname_ind = string_pool::add_string(surname);
}

user_data::user_data(const std::string &id, const std::string
&name,const std::string &surname)
{
    _current_state._id = std::stol(id);
    _current_state._name_ind =
string_pool::add_string(string_pool::get_string(std::stol(name)));
    _current_state._surname_ind =
string_pool::add_string(string_pool::get_string(std::stol(surname)));
}
std::string user_data::to_string() const noexcept
{
    return std::to_string(get_id()) + " " + get_name_value() + " " +
get_surname_value();
}
[[nodiscard]] const std::string &get_name_value() const
{
    return string_pool::get_string(_current_state._name_ind);
}
[[nodiscard]] const std::string &get_surname_value() const
{
    return string_pool::get_string(_current_state._surname_ind);
}
```

Листинг 8. Реализация методов add_string, get_string.

```
static const std::string &get_string(size_t index, bool
get_string_by_index = false)
    {
    auto instance = get_instance();
    auto target = instance->obtain_in_file(index);
    if (target.first)
    {
        auto it = instance->_pool.find(index);
        if (it != instance->_pool.end())
        {
            return *(*it).second;
        }
        auto sh_ptr = std::make_shared<std::string>(target.second);
        instance->_pool.emplace(index, sh_ptr);
        return *sh_ptr;
    }

    static const std::string empty_string;
    return empty_string;
    }

static size_t add_string(const std::string &str, bool by_ind = false)
{
    auto inst = get_instance();
    auto find_result = inst->obtain_in_file(str);
    if (find_result.first)
    {
        return find_result.second;
    }

    auto out_str = std::to_string(_pool_size) + "#" + str + "#" +
'\n';

    inst->_file.open(std::filesystem::absolute(inst-
>_storage_filename), std::ios::app);
    throw_if_not_open(inst->_file);

inst->_pool.emplace(_pool_size, std::make_shared<std::string>(str));
inst->_file << out_str;

inst->_file.close();
    return _pool_size++;
    }
```

Задание 5

Для сохранения состояние базы данных в режиме in_memory_cache в файловую систему был реализован метод класса data_base::save_data_base_state(). Программа проходится по всем пулам, схемам и таблицам, размещённым в оперативной памяти, создаёт по их именам директории с проверкой на существование. Для реализации данного функционала была использована библиотека filesystem.

Листинг 9. Реализация save_data_base_to_filesystem.

```
void data_base::save_data_base_state()
{
    if (get_strategy() != storage_strategy::in_memory)//проверка на валидность способа хранения
    {
        throw std::logic_error("Invalid strategy for this operation");
    }
    std::lock_guard<std::mutex> lock(_mtx);

    if (_allocator)
    {
        std::ofstream temp_file(_instance_path / ("meta.txt"));
        throw_if_not_open(temp_file);
        temp_file << _allocator->get_typename() << std::endl;
        temp_file << _allocator->get_fit_mode_str() << std::endl;
        temp_file.close();
    }

    auto it = _data->begin_infix();
    auto it_end = _data->end_infix();
    while (it != it_end)//проход по B-tree на уровне data_base
    {
    auto string_key = std::get<2>(*it);//имя очередного пула
    auto target_schemas_pool = std::get<3>(*it);

    insert_pool_to_filesystem(std::move(string_key), std::move(target_schemas_pool));

++it;
    }
}
```

Алгоритм сохранения:

1. Проверка стратегии хранения. Прежде всего, функция проверяет, используется ли стратегия хранения storage_strategy::in_memory. Если нет, то генерируется исключение, потому что функция предполагает работу только с этим типом хранения.

2. Блокировка мьютекса. Для предотвращения проблем с параллельным доступом к данным, происходит блокировка мьютекса _mtx с помощью std::lock_guard.

3. Сохранение Информации об аллокаторе. Если существует указатель на аллокатор _allocator, то информация об этом аллокаторе сохраняется в файл meta.txt. Файл сохраняется в директории, указанной _instance_path. В файл записываются тип аллокатора и способ выбора подходящего блока памяти (fit_mode). Файловая операция защищена вызовом функции throw_if_not_open, которая, бросает исключение, если файл не открывается корректно.

4. Обход дерева _data. От data_base объекта _data есть обход B-tree с использованием итераторов (begin_infix и end_infix). Для каждого узла дерева, который содержит пары ключ-значение (имена пулов и соответствующие объекты пулов схем), происходит сохранение данных в файловой системе функцией insert_pool_to_filesystem, которая, предположительно, отвечает за физическую запись пула в файлы.

Для загрузки состояния базы данных из файловой системы в оперативную память, был реализован метод класса data_base::load_data_base_state(). В этом методе программа наоборот проходит по всем директориям и файлам базы данных и вставляет их в B-tree соответствующего уровня, собирая объект от самого низкого уровня(класс table), заканчивая классом schemas_pool. Проверка на существование директории data_base_path / in_memory_storage. В этой директории программа итеративно проходит по пулам схем, в каждом пуле она итеративно проходит по схемам, по каждой схеме она итеративно проходит по таблицам, и в каждой таблице она итеративно проходит по пользователям. Каждый уровень она корректно загружает в B-tree соответствующего класса

Задание 7

Кастомизация аллокатора предоставляется передачей объекта аллокатора в конструктор объекта базы данных, если пользователь не подаёт в конструктор собственный аллокатор, то значение поля allocator* _allocator будет по умолчанию nullptr, из-за чего выделение/освобождение памяти будет производиться через операторы new/delete. Также для поля аллокатора были реализованы геттер и сеттер. В рамках курсового проекта был использован аллокатор с дескриптором границ.

Листинг 10. Конструктор класса data_base.

```
explicit data_base(std::string const &instance_name,
storage_strategy _strategy = storage_strategy::in_memory, allocator*
allocator = nullptr);

data_base::data_base(const std::string &instance_name,
storage_strategy strategy, allocator *allocator)
    : _data(std::make_unique<b_tree<std::string, schemas_pool>>(8,
_default_string_comparer, allocator, nullptr))
{
    this->_allocator = allocator;
}
```

Само выделение памяти происходит на уровне b_tree, реализовано с помощью наследования «контракта» allocator_guardant, который обязывает реализовать метод allocator* get_allocator();

Листинг 11. allocator_guardant.

```
void *allocator_guardant::allocate_with_guard(
    size_t value_size,
    size_t values_count) const
{
    allocator *target_allocator = get_allocator();
    return target_allocator == nullptr
        ? ::operator new(value_size * values_count)
        : target_allocator->allocate(value_size, values_count);
}
void allocator_guardant::deallocate_with_guard(
    void *at) const
{
    allocator *target_allocator = get_allocator();
    return target_allocator == nullptr
        ? ::operator delete(at)
        : target_allocator->deallocate(at);}
```

Задание 8.

В рамках выполнения курсового проекта была разработана серверная часть базы данных на языке C++. Основная задача сервера - обрабатывать сетевые запросы, а также управлять работой базы данных.

Сервер был реализован в виде класса db_server, который содержит два основных члена: уникальный указатель на объект базы данных _db и уникальный указатель на HTTP-сервер _server.

При создании объекта класса db_server в конструкторе происходит инициализация указателей. В зависимости от переданного параметра filesystem_strategy, база данных может работать либо с файловой системой, либо с оперативной памятью.

Листинг 12. Основные поля и конструктор сервера.

```
class db_server
{
public:
    static std::atomic<bool> stop_signal;

private:
    int port;
    std::unique_ptr<data_base> _db;
    std::unique_ptr<httplib::Server> _server;

public:
    explicit db_server(std::string const &db_name, bool
filesystem_strategy = false)
        : _server(std::make_unique<httplib::Server>())
    {
        if (filesystem_strategy)
        {
            _db = std::make_unique<data_base>(db_name,
storage_interface<std::string,
schemas_pool>::storage_strategy::filesystem);
        }
        else
        {
            _db = std::make_unique<data_base>(db_name,
storage_interface<std::string,
schemas_pool>::storage_strategy::in_memory);
        }
    }
```

Листинг 13. Endpoints, API.

```
void do_work(int port)
{

_server->Post("/insert_data", [&](const httplib::Request &req,
httplib::Response &res) {
    try
    {
        handle_insert_data(req, res);
        res.status = 201;// Created
    }
    catch (const std::exception &e)
    {
        res.status = 500;// Internal Server Error
        res.set_content(e.what(), "text/plain");
    }
});


_server->Get("/obtain_data", [&](const httplib::Request &req,
httplib::Response &res) {
    try
    {
        handle_obtain_data(req, res);
        res.status = 200;// OK
    }
    catch (const std::exception &e)
    {
        res.status = 404;// Not Found
        res.set_content(e.what(), "text/plain");
    }
});



_server->Put("/update_data", [&](const httplib::Request &req,
httplib::Response &res) {
    try
    {
        handle_update_data(req, res);
        res.status = 200;// OK
    }
    catch (const std::exception &e)
    {
        res.status = 404;// Not Found
        res.set_content(e.what(), "text/plain");
    }
});

//other endpoints
```

```
_server->Delete("/dispose_table", [&](const httplib::Request &req,
httplib::Response &res) {
    try
    {
        handle_dispose_table(req, res);
        res.status = 200;// OK
    }
    catch (const std::exception &e)
    {
        res.status = 404;// Not Found
        res.set_content(e.what(), "text/plain");
    }
});

_server->Get("/alive", [&](const httplib::Request &req,
httplib::Response &res) {
    res.set_content("Server is alive", "text/plain");
    res.status = 200;// OK
});

_server->listen("localhost", port);

}
```

Основная логика работы сервера реализована в методе do_work(int port). В этом методе настраиваются обработчики HTTP-запросов для различных типов операций, таких как вставка данных, обновление данных, удаление данных и т.д. Каждый обработчик принимает HTTP-запрос и возвращает HTTP-ответ. В случае возникновения исключений, сервер возвращает код ошибки 500. В случае успешного выполнения запроса возвращается соответствующий статусный код (201 для операций вставки, 200 для остальных операций).

Сервер поддерживает методы GET, POST, PUT и DELETE, что позволяет полноценно взаимодействовать с базой данных. Для каждого типа запроса реализованы соответствующие обработчики.

Сервер также поддерживает методы для сохранения и загрузки состояния базы данных. Это позволяет сохранять текущее состояние базы данных в файл и загружать его при последующем запуске сервера.

Стоит отметить, что класс db_server реализован с учетом принципа RAII и поддерживает перемещение, но не поддерживает копирование. Это обеспечивает безопасное управление ресурсами и предотвращает возникновение утечек памяти.

В процессе разработки сервера базы данных использовались две внешние библиотеки: httplib и nlohmann::json.

Библиотека httplib используется для создания HTTP сервера и клиента. Эта библиотека предоставляет простой и удобный интерфейс для работы с HTTP, позволяя легко создавать и обрабатывать HTTP-запросы и ответы. В данном проекте httplib используется для создания HTTP-сервера, который слушает определённый порт и обрабатывает входящие HTTP-запросы, направляя их соответствующим обработчикам.

Библиотека nlohmann::json используется для работы с JSON. JSON (JavaScript Object Notation) - это формат данных, который часто используется для передачи данных между сервером и клиентом. Библиотека nlohmann::json предоставляет удобные инструменты для сериализации и десериализации данных в формате JSON, а также для работы с JSON-объектами на уровне языка C++. В данном проекте эта библиотека может быть использована для формирования ответов сервера в формате JSON, а также для обработки данных, полученных от клиента в этом формате.

Обе эти библиотеки являются кросс-платформенными и открытыми, что делает их удобными для использования в различных проектах. Они активно поддерживаются их авторами и сообществом, имеют хорошую документацию и широкий функционал.

В целом, разработанный сервер обеспечивает эффективное и безопасное взаимодействие с базой данных, поддерживает все необходимые операции и обеспечивает удобный интерфейс для работы с базой данных.

Листинг 14. Пример обработчика запроса на вставку данных.

```
void handle_insert_data(
    const httplib::Request &req,
    httplib::Response &res)
   {
auto json = nlohmann::json::parse(req.body);
std::string pool_name = json["pool_name"];
std::string schema_name = json["schema_name"];
std::string table_name = json["table_name"];
std::string user_data_key = json["key"];
user_data data(std::stol(json["id"].get<std::string>()),
json["name"], json["surname"]);
_db->insert_data(pool_name, schema_name, table_name, user_data_key,
std::move(data));
res.set_content("Data inserted successfully", "text/plain");
   }
```

Тестирование кода

Тест 1: PositiveTest1.

Этот тест проверяет базовую функциональность API без использования аллокатора. В тесте создаётся база данных, добавляются и извлекаются данные. Также проверяются случаи с несуществующими ключами и попытками вставки дубликатов ключей.

Тест 2: PositiveTest2FirstFit.

Этот тест проверяет API с использованием аллокатора и стратегией "first fit". В тесте создаётся база данных, добавляются и извлекаются данные. Также проверяются случаи с несуществующими ключами, попытками вставки дубликатов ключей и обновлением данных.

Тест 3: PositiveTest2BestFit.

Этот тест аналогичен тесту 2, но использует стратегию "the best fit" для аллокатора. Также в тесте проверяются операции добавления, извлечения, обновления и удаления данных.

Тест 4: PositiveTest2WorstFit.

Этот тест аналогичен тестам 2 и 3, но использует стратегию "the worst fit" для аллокатора. Также в тесте проверяются операции добавления, извлечения, обновления и удаления данных.

Тест 5: FilesystemStorageTest.

Этот тест проверяет использование файловой системы для хранения данных. В тесте создаётся база данных, добавляются и извлекаются данные. Также проверяются случаи с несуществующими ключами, попытками вставки дубликатов ключей, обновлением и удалением данных.

Тест 6: ObtainBetweenTest.

Этот тест проверяет функцию obtain_between_data, которая извлекает данные между двумя ключами. В тесте создается база данных и вводятся 100 записей. Затем функция obtain_between_data вызывается дважды: сначала для извлечения записей между ключами "key10" и "key50", а затем - между "key98" и "key99". Ожидается, что функция вернёт корректные записи в обоих случаях.

Тест 7: PositiveTestSavingState.

Этот тест проверяет сохранение состояния базы данных. В тесте создаётся база данных, вводятся 50 записей, а затем вызывается функция save_data_base_state, которая проходится по всем таблицам во всех схема и пулах схем конкретной базы данных и сохраняет состояние записей и в целом класса data_base в файловую систему с возможной последующей выгрузкой. Ожидается, что функция выполнится без ошибок.

Тест 8: PositiveTestLoadingState.

Этот тест проверяет загрузку состояния базы данных. В тесте создаётся база данных, а затем вызывается функция load_data_base_state. Ожидается, что функция выполнится без ошибок и данные будут успешно загружены.

Тест 9: ExecutingFileCommand.

Этот тест проверяет выполнение команд из файла. В тесте создается база данных, а затем вызывается функция execute_command_from_file. Ожидается, что функция выполнится без ошибок и корректно обработает команды из файла.

Тест 10: AnyInsertionPoolAndSchemas.

Этот тест проверяет различные сценарии вставки и удаления данных. В тесте создаётся база данных, вводятся и удаляются различные элементы, а также проверяются ситуации с попыткой вставки или удаления несуществующих элементов.

Тест 11: WorkWithUnexistedData.

Этот тест проверяет обработку несуществующих данных. В тесте создаётся база данных, вводятся данные, а затем пытаются удалить данные, которые не существуют.

Тест 12: StringPoolTests.

Этот тест проверяет работу с пулом строк. В тесте создается объект user_data, а затем проверяются его свойства и методы. Ожидается, что все методы вернут корректные результаты.

Для реализации тестов использовалась сторонняя библиотека Gtest.

Листинг 15. Тестирование string_pool.

```
TEST(StringPoolTests, test12)
{
    user_data ud(1,"name", "surname");
    EXPECT_EQ(ud.get_id(), 1);
    EXPECT_EQ(ud.get_name_value(), std::string{"name"});
    EXPECT_EQ(ud.get_surname_value(), std::string{"surname"});

    EXPECT_NE(ud.get_name(), ud.get_name_value());
    EXPECT_NE(ud.get_surname(), ud.get_surname_value());

    EXPECT_FALSE(ud.get_name_value() ==
std::to_string(ud.get_name_ind()));
    EXPECT_FALSE(ud.get_surname_value() ==
std::to_string(ud.get_surname_ind()));
}
```

Заключение

В ходе выполнения курсового проекта по разработке приложения на языке программирования C++ для управления коллекциями данных основное внимание было уделено строгому соблюдению стандартов языка C++17 и построению гибкой и модульной архитектуры программы. Этот аспект оказался критически важным для обеспечения устойчивости, надёжности и переносимости кода на различных платформах.

Ключевым элементом работы стало создание абстрактного интерфейса storage_interface, использующего ассоциативные контейнеры типа B-tree для организации данных. Этот подход был выбран на основе анализа современных методов обработки и управления данными, представленных в документации по языку C++ [2]. В рамках этого интерфейса был реализован класс storage_strategy, предназначенные для работы с данными в оперативной памяти и на файловой системе соответственно. Такой подход обеспечил чёткое разделение логики работы с различными типами хранилищ, что существенно повысило модульность и переиспользуемость кода, а также упростило его тестирование и отладку.

Класс user_data стал одним из центральных компонентов системы, отвечая за хранение информации о пользователях. В данной реализации особое внимание было уделено правильному управлению памятью и эффективному обращению с строковыми данными посредством использования строкового пула. Класс предоставляет полный набор методов для манипуляции данными, включая конструкторы, операторы присваивания, а также специализированные методы для создания объектов из строковых данных. Этот функционал позволил обеспечить высокую производительность и минимизировать вероятность утечек памяти.

Существенной частью работы стало обеспечение эффективного взаимодействия пользователя с приложением. Для этого был реализован удобный консольный интерфейс, который позволяет вводить команды в интерактивном режиме, а также выполнять предопределённые сценарии посредством чтения команд из файлов. Этот подход значительно повысил удобство использования программы и упростил процесс её настройки и эксплуатации.

Итоговая реализация продемонстрировала высокую степень переносимости и предсказуемое поведение программы на различных платформах, что стало возможным благодаря применению современных методов управления памятью.

Соблюдение принципов объектно-ориентированного программирования и использование передовых подходов к разработке программного обеспечения сделали код более структурированным, устойчивым к изменениям в окружении выполнения, а также повысили его безопасность и эффективность.

Курсовой проект позволил углубить знания в области алгоритмов и структур данных, научиться анализировать и сравнивать эффективность различных подходов к управлению памятью и файловыми системами, что было сделано с использованием материалов, представленных в работе Страуструпа [3].

Список использованных источников

[1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. Санкт-Петербург, Питер, 2001. 368 с.

[2] Документация по языку C++. [Электронный ресурс]
[URL: https://en.cppreference.com/] (Дата обращения: 29.05.2024)

[3] Страуструп Б. Программирование. Принципы и практика использования C++. 2-е изд. - Санкт-Петербург, Питер, 2016. - 864 с.

Приложение

Батоян, Р.Л. (2024). Приложение кода к проекту на GitHub. [Электронный ресурс]. GitHub. [URL: https://github.com/rob228rob/cw_os.git]