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

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: Некрасов К. Д.

Группа: М8О-311Б-22

Оценка:

Дата:

Москва 2024

Оглавление

Введение

Целью данной курсовой работы является разработка приложения на языке программирования C++ (в случае данной работы использовался стандарт C++23), 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 над данными. Приложение подобного рода будет полезно для анализа, хранения и обработки больших объемов 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.
2. Реализация основных операций над коллекциями данных, таких как добавление, чтение, обновление и удаление записей по ключу.
3. Поддержка операций над коллекциями данных, включающих добавление или удаление пулов данных, схем данных и коллекций данных.
4. Обеспечение взаимодействия с приложением посредством выполнения команд, поступающих из файла, путь к которому передаётся через аргумент командной строки.
5. Работа приложения в двух режимах хранения: размещение структур и данных в оперативной памяти (in-memory cache) и в 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 личном 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 личное понимание развития 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, указанныевфайле,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 для соответствующих типов: 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 (для 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 соответственно.
- _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 хранение в оперативной памяти, 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 со 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++23 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. Функция создания 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. Функция создания backup файла

```
void table::create_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, ив 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. max_data_length, index_item_max_length: ограничивают максимальную длину данных и элементов индекса.
- 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. Границы диапазона могут быть включены или исключены.
- 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 const &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;
```

```
        while (std::getline(data_file, src_line))
        {
            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(src_line));

                if (current_key == file_key)
                {
                    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(path);
        load_backup(index_path);
        throw;
    }
    delete_backup(path);
    delete_backup(index_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 или номер соответствующей команды, 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] dispose_pool – удаление пула схем данных

[5] dispose_schema – удаление схемы данных

[6] dispose_table – удаление таблицы схемы данных

[7] insert_data – вставка данных о пользователе в таблицу

[8] dispose_data – удаление пользователя по ключу из таблицы

[9] obtain_data – поиск пользователя в таблице по ключу

[10] obtain_between_data – 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 режиме 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 или соответствующий ей номер.

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. Основные поля класс string_pool

```
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, 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. Основные поля класс string_pool

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

         return _instance;
     }
```

Сам string_pool имеет два основных статических метода- intadd_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 get_string и add_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 get_string_by_index = false)
{
    auto inst = get_instance();
    std::string current_str;

    if (get_string_by_index)
    {
        current_str = get_string(std::stol(str));
        if (current_str.empty())
        {
            current_str = str;
        }
    }
    else
    {
        current_str = str;
    }
```

```
    auto find_result = inst->obtain_in_file(current_str);
    if (find_result.first)
    {
        return find_result.second;
    }

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

    inst->_file.open(std::filesystem::absolute(inst->_storage_filename),
std::ios::app);
    if (!inst->_file.is_open())
    {
        throw std::runtime_error("Cannot open file for reading: " + inst-
>_storage_filename);
    }

    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_state.

```
void data_base::save_data_base_state()
{
if (get_strategy() != storage_strategy::in_memory)
    {
         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)
    {
        auto string_key = std::get<2>(*it);
        auto target_schemas_pool = std::get<3>(*it);
         insert_pool_to_filesystem(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-treec 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 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.

```
void 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);
}
```

Заключение

В ходе выполнения курсовой работы по разработке приложения на языке программирования C++ для управления коллекциями данных основное внимание было уделено соблюдению стандартов языка C++23 и построению гибкой и модульной архитектуры программы. Этот аспект оказался 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. В рамках этого интерфейса был реализован класс storage_strategy, предназначенный для работы с данными как в оперативной памяти, так и на файловой системе. Такой подход обеспечил четкое разделение логики работы с различными типами хранилищ, что существенно повысило модульность и повторную используемость кода, а также упростило его тестирование и отладку.

Класс user_data стал одним из центральных компонентов системы, отвечающим за хранение информации о пользователях. В реализации программы особое внимание было уделено правильному управлению памятью и эффективному обращению с строковыми данными посредством использования пула строк. 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 как полной текстовой строкой, так и простой цифрой, а также выполнять предопределённые сценарии посредством чтения команд из передаваемых файлов. Такой подход значительно повысил удобство использования программы и упростил процесс её настройки и эксплуатации.

Итоговая реализация продемонстрировала значительную степень переносимости и предсказуемое поведение программы на различных платформах, что было достигнуто за счёт применения стандартов C++23 и современным методам управления памятью. Соблюдение принципов ООП и использование передовых подходов к разработке программного обеспечения сделали код более структурированным, устойчивым к изменениям в окружении выполнения, и вместе с тем повысили его безопасность и эффективность.

Курсовая работа позволила углубить знания в области алгоритмов и структур данных, научиться анализировать и сравнивать эффективность различных подходов к управлению памятью и файловыми системами. Приобретённые навыки и опыт

явно окажутся полезными в дальнейшей профессиональной деятельности, особенно при разработке сложных программных систем и решении задач в области информационных технологий.

Список источников

[1] Документация по языку C++ [Электронный ресурс]
[URL: https://en.cppreference.com/w/]
[2] Э.Таненбаум, Х. Бос – Современные операционные системы

Приложение

Некрасов К. Д. Ссылка на репозиторий с кодом проекта на GitHub. [Электронный ресурс]. GitHub [URL: https://github.com/RedZeroTheCat/cw_OS]