МИНИСТЕРСТВО ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ 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 ДИСЦИПЛИНЕ «АРХИТЕКТУРА КОМПЬЮТЕРОВ И ОПЕРАЦИОННЫХ СИСТЕМ»

# НА ТЕМУ: «РАЗРАБОТКА АЛГОРИТМОВ СИСТЕМЫ ХРАНЕНИЯ И УПРАВЛЕНИЯ ДАННЫМИ НА ОСНОВЕ ДИНАМИЧЕСКИХ СТРУКТУР ДАННЫХ»

Студент: Сафин Тимур Айратович

Группа: М8О-211Б-22

Преподаватель: Романенков Александр Михайлович

Оценка:_____

Дата:_____

Москва, 2024

Оглавление

# Введение

Цель данного курсового проекта — разработка приложения на языке *C++* (стандарт *C++14* и выше), которое реализует работу с коллекциями данных различных типов. Эти коллекции данных должны поддерживать выполнение операций, таких как добавление, чтение, обновление и удаление записей на основе уникальных ключей. Приложение будет оперировать структурами данных, хранимыми в ассоциативных контейнерах вида *B-tree*, что обеспечит эффективный доступ к данным и их структурам.

Основная задача проекта заключается в создании интерфейса для управления коллекциями данных, которые описываются набором параметров, таких как название пула схем данных, название конкретной схемы и название самой коллекции данных. Этот интерфейс должен обеспечивать выполнение базовых операций работы с данными, включая добавление новых записей, чтение записей по ключу, обработку диапазонов ключей и удаление записей.

Помимо базовых операций с записями, приложение должно также предоставлять возможности для добавления и удаления целых пулов данных, схем данных и коллекций данных. Это создаёт дополнительные требования к архитектуре системы, так как необходимо гибко управлять множеством различных наборов данных, поддерживая их целостность и согласованность.

Для удобства использования приложение должно поддерживать два режима работы:

- In-memory-cache — структуры и данные находятся исключительно в оперативной памяти, что обеспечивает высокую скорость доступа, но ограничивает объём данных.
- Файловая система — данные сохраняются в файлах на диске, что позволяет работать с большими объёмами информации, однако скорость операций может быть ниже по сравнению с оперативной памятью.

Выбор режима работы происходит на этапе запуска приложения через аргументы командной строки. Это предоставляет пользователю гибкость в зависимости от конкретных условий эксплуатации: от задач, требующих быстрого доступа к данным, до задач, где важен объём обрабатываемой информации.

Проект предполагает глубокое изучение и практическую реализацию концепций работы с ассоциативными контейнерами, структуры данных типа *B-tree*, организации данных в памяти и на диске, а также управления сложными наборами данных.

Задание 0

Приложение представляет собой базу данных на языке C++, поддерживающие ассоциативные контейнеры типа *B-tree*. Для управления данной базой данных был реализован класс *data_base*, наследованный от абстрактного шаблонного класса *interface<std::string, schema_pool>*, в котором объявлены все необходимые функции для работы с базой данных.

Основной функционал:

- Создание, обновление и удаление схем (представляют структуру данных) и таблиц (контейнеры для данных) в базах данных. Эти функции нужны для организации данных в виде иерархии: база данных → схема → таблица.
- Вставка, обновление и получение данных через методы, работающие с ключами и значениями (например, insert_data, update_data, obtain_data). Поддерживаются как передача значений по ссылке, так и перемещение значений.
- Работа с файлами. Методы по сохранению состояния базы данных в файлы и загрузке из них. Это нужно для обеспечения устойчивости системы (например, при перезапуске программы). Функции типа insert_pool_to_file, update_person_in_file и т. д. работают с сохранением и обновлением данных в файловой системе.
- Управление пулами схем. Пулы схем (schemas_pool) организуют наборы связанных данных. Класс позволяет вставлять пулы схем, получать их и удалять. Это нужно для логической группировки данных.
- Поддерживаются операции поиска данных в заданных диапазонах значений, что полезно для выборки подмножеств данных (например, obtain_between_data).
- Возможность выбора стратегии хранения данных (в памяти или в файловой системе).
- В классе есть функции, которые создают файлы при необходимости, что делает его удобным для работы с файловыми системами без постоянного контроля со стороны разработчика.

Основное применение:

- Этот класс можно использовать для создания систем управления базами данных, которые работают с файлами. Он может быть полезен для легковесных приложений, которые не нуждаются в полномасштабной реляционной базе данных, но все же требуют структурированного хранения информации.
- Ориентирован на работу с таблицами и структурированными данными (например, данными о пользователях).
- Подобная структура может применяться в системах, которые требуют управления объектами или записями на уровне приложений (например, учетные системы, хранилища метаданных).

Класс *interface* служит основой для реализации различных типов хранилищ данных, предоставляя унифицированный интерфейс для работы с данными. Он может быть расширен конкретными классами, которые реализуют методы для работы с данными в соответствии с выбранной стратегией хранения (в памяти или в файлах). Такой подход облегчает разработку и поддержку систем хранения данных, позволяя использовать одни и те же методы для работы с различными типами данных и стратегиями.

Основной функционал класса interface:

- Класс является шаблонным и принимает два параметра: tkey (тип ключа) и tvalue (тип значения), что позволяет создавать различные реализации хранилищ с различными типами данных.
- Перечисление strategy определяет, как будут храниться данные: в памяти или в файлах.
- *allocator *_allocator*: Указатель на объект для управления памятью.
- *logger *_logger*: Указатель на объект для логирования.
- *strategy _strategy*: Текущая стратегия хранения.
- *std::string _instance_name*: Имя экземпляра хранилища.

Описание функций класса *interface*:

1. Методы управления памятью и логированием:
   - *allocator *get_allocator*() *const*: Возвращает указатель на объект аллокатора.
   - *logger *get_logger*() *const*: Возвращает указатель на объект логгера.
2. Методы для работы со стратегией:
   - *strategy get_strategy*() *const*: Возвращает текущую стратегию хранения.
   - *void set_strategy*(*strategy Strategy*): Устанавливает стратегию хранения.
3. Методы для работы с именем экземпляра:
   - *std*::*string get_instance_name*(): Возвращает имя экземпляра.
   - *void set_instance_name*(*std*::*string const &name*): Устанавливает имя экземпляра.
4. Методы для проверки ошибок:
   - *static void throw_wrong_length*(*std*::*string const &data*): Выбрасывает исключение, если длина данных превышает максимально допустимую.
   - *static void throw_if_open_fails*(*std*::*ifstream const &file*) и *static void throw_if_open_fails*(*std*::*ofstream const &file*): Проверяют, успешно ли открыт файл.
   - *static void decrease_index*(*std*::*vector<std*::*streamoff> &vec*): Уменьшает индекс, удаляя последний элемент вектора.

- *static int find_index*(*std*::*ifstream* &*src*, *std*::*vector*<*std*::*streamoff*> *const* &*vec*, *std*::*string const* &*key*): Находит индекс ключа в файле.

5. Методы работы с данными:
   - *virtual void insert*(*const tkey* &*key*, *const tvalue* &*value*) = 0: Виртуальный метод для добавления пары "ключ-значение".
   - *virtual void insert*(*const tkey* &*key*, *tvalue* &&*value*) = 0: Виртуальный метод для добавления пары с перемещением значения.
   - *virtual tvalue* &*obtain*(*const tkey* &*key*) = 0: Виртуальный метод для получения значения по ключу.
   - *virtual std*::*map*<*tkey*, *tvalue*> *obtain_between*(...) = 0: Виртуальный метод для получения значений в заданном диапазоне ключей.
   - *virtual void update*(*const tkey* &*key*, *const tvalue* &*value*) = 0: Виртуальный метод для обновления значения по ключу.
   - *virtual void update*(*const tkey* &*key*, *tvalue* &&*value*) = 0: Виртуальный метод для обновления значения с перемещением.
   - *virtual void dispose*(*const tkey* &*key*) = 0: Виртуальный метод для удаления значения по ключу.
6. Методы для работы с индексами:
   - *static std*::*vector*<*std*::*streamoff*> *load_index*(*const std*::*string* &*index_filename*): Загружает индекс из файла.
   - *static void save_index*(*const std*::*vector*<*std*::*streamoff*> &*vec*, *const std*::*string* &*filename*): Сохраняет индекс в файл.
   - *static void update_index*(*std*::*vector*<*std*::*streamoff*> &*vec*): Обновляет индекс.
   - *static void save_index*(*std*::*vector*<*std*::*streamoff*> *const* &*vec*, *std*::*ofstream* &*file*): Сохраняет индекс в открытый файл.
7. Методы для работы с файлами и резервными копиями:
   - *static person_data obtain_in_file*(...): Получает данные из файла по заданному ключу.
   - *static void delete_backup*(*const std*::*filesystem*::*path* &*source_path*): Удаляет резервную копию.
   - *static void create_backup*(*const std*::*filesystem*::*path* &*source_path*): Создает резервную копию файла.

Класс *person_data* представляет собой структуру, которая используется для хранения информации о человеке, включая его идентификатор, имя и фамилию. Он включает внутренний класс *person_index_data*, который управляет индексами для имени и фамилии, используя пул строк для оптимизации памяти.

Листинг 1. Класс person_index_data

```
class person_index_data {
        friend class person_data;

        size_t id;

        size_t name_id;

        size_t surname_id;

    public:
        person_index_data(size_t id, std::string const &name_id,
std::string const &surname_id): id(id),
            name_id(string_pool::add_string(name_id)),
            surname_id(string_pool::add_string(surname_id)) {
        }

        person_index_data(size_t id, std::string &&name_id, std::string
&&surname_id): id(id),
            name_id(string_pool::add_string(name_id)),
            surname_id(string_pool::add_string(surname_id)) {
        }

        person_index_data() = default;

        person_index_data(person_index_data const &other): id(other.id),
name_id(other.name_id),

surname_id(other.surname_id) {
        }

        person_index_data &operator=(person_index_data const &other) {
            if (this != &other) {
                id = other.id;
                name_id = other.name_id;
                surname_id = other.surname_id;
            }
            return *this;
        }

        person_index_data(person_index_data &&) = delete;

        person_index_data &operator=(person_index_data &&) = delete;

        ~person_index_data() = default;
    };
```

Данный подход оказался весьма удобным, эффективным и безопасным, работая с большими данными. Он позволяет не копировать уже созданные объекты, а использовать уже имеющиеся, что значительно увеличивает

эффективность программы. Внутренний класс оказался так же очень удобным для использования внутри программы, добавления, удаления и получения данных, ведь работает с менее требовательными данными (ключами).

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.

Для таких случаем в классе *interface* реализованы такие функции, как create_backup и load_backup, которые создают копии файлов, которые могу быть повреждены в ходе досрочного выхода из программы.

Листинг 2. Функция create_backup

```
void interface<tkey, tvalue>::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_open_fails(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();
}
```

Функция *create_backup* класса *interface* создает резервную копию файла, указанного в параметре *source_path*. Сначала она проверяет, существует ли исходный файл, и, если нет, выбрасывает исключение. Затем формирует имя резервной копии, добавляя суффикс "*.backup*" к пути исходного файла. Открывает исходный файл для чтения в двоичном режиме и проверяет успешность открытия. Если файл открыт успешно, функция создает новый файл для резервной копии, также в двоичном режиме, и если создание файла не удается, выбрасывает исключение. Затем содержимое исходного файла копируется в файл резервной копии. В конце оба файла закрываются.

Листинг 3. Функция load_backup

```
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);
    }
}
```

Функция *load_backup* класса *table* загружает резервную копию данных из файла, имя которого указано в параметре *source_path*, добавляя к нему суффикс "*.backup*". Сначала она проверяет существование файла резервной копии и, если он не найден, выбрасывает исключение. Затем функция создает временную копию резервного файла и, если исходный файл существует, удаляет его. После этого временная копия переименовывается в имя исходного файла. В случае возникновения ошибок в процессе загрузки, функция удаляет созданный исходный файл (если он существует) и пытается восстановить его из временной копии. В конце работы удаляется временная копия резервной копии, если она существует.

Операция вставки, удаления, обновления и другие основывается на бинарном поиске по ключу, так как все данные в таблицах отсортированы. Рассмотрим работу с файлами на основе функции insert_person_to_file.

Функция *insert_person_to_file* класса *table* добавляет запись о человеке в файл, а также обновляет индексный файл. В начале функции формируется строка, представляющая запись, с помощью данных о человеке и уникального ключа, добавляемого в специальный пул строк. Затем проверяются длина строки и наличие записи с тем же ключом, после чего, если индексный файл пуст, создаются новые файлы. В противном случае функция использует бинарный поиск для проверки наличия дубликата ключа, и в случае его нахождения выбрасывает исключение.

Если дубликатов нет, создаются резервные копии данных и индекса. Если новая запись должна быть добавлена в конец файла, она просто добавляется в конец, в противном случае создается временный файл, в который записываются данные, включая новую запись в нужном порядке. При этом обновляется индекс. В случае ошибок при записи или создании временного файла выполняется восстановление из резервной копии. В конце удаляются временные и резервные файлы. Важные вспомогательные функции включают *throw_wrong_length*, *make_same_length, create_backup, load_backup, delete_backup*, а также функции для работы с индексами, такие как *update_index* и *save_index*.

Листинг 4. Функция insert_person_to_file

```
void table::insert_person_to_file(std::filesystem::path const &path,
std::filesystem::path const &index_path,
                                  std::string const &key, person_data
const &person) {
    std::string out_str = std::to_string(string_pool::add_string(key)) +
"#" + std::to_string(person.get_id()) + "#" +
```

```
                        std::to_string(person.get_name_id()) + "#" +
                        std::to_string(person.get_surname_id()) + "|";
    throw_wrong_length(out_str);
    make_same_length(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_open_fails(out_f);
        std::ofstream index_f(index_path);
        if (!index_f.is_open()) {
            out_f.close();
            throw_if_open_fails(index_f);
        }
        out_f << out_str << std::endl;
        out_f.close();
        update_index(index_array);
        interface::save_index(index_array, index_f);
        index_f.close();
        return;
    }

    std::ifstream src(path);
    throw_if_open_fails(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, '#');
        key_index = std::regex_replace(key_index, std::regex(R"([^\d])"),
"");
        file_key = string_pool::get_string(std::stol(key_index));
        if (file_key == key) {
            src.close();
            throw std::logic_error("void
table::insert_person_to_file(...) -> key 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_open_fails(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_open_fails(data_file);
        auto tmp_filename = path.string() + "_temp" + _format;
        std::ofstream tmp_file(tmp_filename);
        if (!tmp_file.is_open()) {
            data_file.close();
            throw_if_open_fails(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);
                key_index = std::regex_replace(key_index,
std::regex(R"([^\d])"), "");
                std::string current_key =
string_pool::get_string(std::stol(key_index));
```

```
                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 4

Описание задания: обеспечьте хранение объектов строк, размещённых в объектах данных, на основе структурного паттерна "Приспособленец". Дублирования строк для разных объектов при этом запрещены. Доступ к строковому пулу обеспечьте на основе порождающего паттерна "Одиночка".

Класс *string_pool* реализует паттерн "Приспособленец" [1] для хранения строк и паттерн "Одиночка" для обеспечения единого доступа к экземпляру пула строк. Он предотвращает дублирование строк для разных объектов, что позволяет эффективно управлять памятью и избегать лишних копий строковых данных.

Описание функций:

1. Статические переменные и экземпляры:
   - *_pool_size*: хранит текущий размер пула строк.
   - *_instance*: указатель на единственный экземпляр *string_pool*.
2. Методы для работы с данными:
   - *load_data_from_file*(): Загружает данные из файла в пул. Каждая строка разбивается на ключ и значение, и значения добавляются в *_pool* в виде *shared_ptr*. Это позволяет избежать дублирования строк, так как несколько объектов могут ссылаться на одну и ту же строку.
   - *save_to_file*(): Сохраняет текущие строки из пула в файл. Использует формат "ключ#значение" для записи данных, что позволяет легко загружать их позже.
   - *obtain_in_file*(*size_t index*): Извлекает строку по индексу из файла. Возвращает пару, указывающую, была ли строка найдена, и саму строку.
   - *obtain_in_file*(*const std::string &str*): Извлекает индекс строки по её значению. Возвращает пару, указывающую, была ли строка найдена, и соответствующий индекс.
3. Конструктор и деструктор:
   - *string_pool*(): Конструктор, который открывает файл и загружает данные. Если файл не существует, он создаётся.
   - *~string_pool*(): Деструктор, который закрывает файл и удаляет экземпляр string_pool.
4. Методы получения экземпляра:
   - *get_instance*(): Возвращает единственный экземпляр string_pool. Если экземпляр ещё не создан, он создаётся. Это реализует паттерн "Одиночка", который гарантирует, что будет существовать только один экземпляр пула строк на всё приложение.
5. Методы для получения и добавления строк:
   - *get_string*(*size_t index, bool get_string_by_index*): Получает строку по индексу. Если строка найдена, возвращает её. Если нет, создаётся новая строка.
   - *add_string*(*const std::string &str, bool get_string_by_index*) и *add_string*(*std::string &&str, bool get_string_by_index*): Эти методы

добавляют новую строку в пул и возвращают её индекс. Если строка уже существует, возвращается её индекс, что предотвращает дублирование.

6. Методы для поиска строк:
   - *find_in_file*(*size_t index*): Находит строку по индексу, возвращает пустую строку, если не найдена.

Листинг 5. Основные методы класса string_pool

```
size_t string_pool::add_string(const std::string &str, bool
get_string_by_index) {
    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(
            "size_t string_pool::add_string(const std::string&, bool):
file open error " + inst->_storage_filename);
    }
    inst->_pool.emplace(_pool_size, std::make_shared<std::string>(str));
    inst->_file << out_str;
    inst->_file.close();
    return _pool_size++;
}
const std::string &string_pool::get_string(size_t index, bool
get_string_by_index) {
    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;
}
string_pool::string_pool() {
    auto path = std::filesystem::absolute(_storage_filename);
    if (!exists(path)) {
        std::ofstream ofs(path);
        if (!ofs.is_open()) {
            throw std::runtime_error("string_pool::string_pool() -> file
open error: " + _storage_filename);
        }
        _pool_size = 0;
        ofs.close();
    }
    load_data_from_file();
}

string_pool::~string_pool() {
    if (_file.is_open()) {
        _file.close();
    }
    delete _instance;
}

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

    return _instance;
}

void string_pool::load_data_from_file() {
    _file.open(std::filesystem::absolute(_storage_filename),
std::ios::in);
    if (!_file.is_open()) {
        throw std::runtime_error("Cannot open file for reading: " +
_storage_filename);
    }
    std::string line;
    int index = 0;
    while (std::getline(_file, line)) {
        size_t pos1 = line.find('#');
        if (pos1 == std::string::npos) {
            break;
        }
        std::string key = line.substr(0, pos1);
        std::string rest = line.substr(pos1 + 1);
        size_t pos2 = rest.find('#');
        if (pos2 == std::string::npos) {
            continue;
        }
        std::string value = rest.substr(0, pos2);
```

```
        _pool.emplace(std::stol(key),
std::make_shared<std::string>(value));
        ++index;
    }
    _pool_size = index;
    _file.close();
}

void string_pool::save_to_file() {
    _file.seekg(0);
    if (!_file.is_open()) {
        throw std::runtime_error("Cannot open file for writing: " +
_storage_filename);
    }
    for (const auto &pair: _pool) {
        _file << (pair).second.get() << '\n';
    }
}
```

Проявление паттернов:

- Класс *string_pool* использует умные указатели (*std::shared_ptr*) [2], чтобы гарантировать, что все объекты, использующие одну и ту же строку, ссылаются на один экземпляр строки. Таким образом, дублирование строк предотвращается, а память экономится, что соответствует принципам паттерна "Приспособленец".
- Паттерн "Одиночка" реализован через статический метод *get_instance*(), который контролирует создание и доступ к единственному экземпляру *string_pool*. Это обеспечивает глобальный доступ к пулу строк, упрощая управление строками и обеспечивая их уникальность в приложении.

Таким образом, класс *string_pool* эффективно управляет строками, предотвращая дублирование и обеспечивая доступ через единый экземпляр, что делает его мощным инструментом для работы с текстовыми данными.

## Задание 5

Описание задания: для режима *in-memory cache* реализуйте механизмы сохранения состояния системы хранения данных в файловую систему и восстановления состояния системы хранения данных из файловой системы.

В данном проекте механизмы сохранения состояния системы хранения данных в файловую систему и обратно реализованы при помощи методов класса *data_base load_data_base_state* и *save_data_base_state*.

Метод *load_data_base_state* отвечает за восстановление состояния базы данных в режиме *in-memory cache*. Он проверяет, соответствует ли текущая стратегия *memory*, и если это так, загружает данные из файловой системы, проходя через все подкаталоги, представляющие пулы схем и таблицы. Для каждой таблицы вызывается метод *table::load_data_from_filesystem*, который читает данные из файла и заполняет таблицу соответствующими записями.

Листинг 6. Метод load_data_base_state

```
void data_base::load_data_base_state() {
    if (get_strategy() != memory) {
        throw std::logic_error("void data_base::load_data_base_state() ->
invalid strategy for this operation");
    }

    std::filesystem::path db_storage_path = _instance_path;

    if (!std::filesystem::exists(db_storage_path)) {
        throw std::runtime_error("void data_base::load_data_base_state()
-> data base directory doesn't exist");
    }

    for (auto &pool_entry:
std::filesystem::directory_iterator(db_storage_path)) {
        if (!pool_entry.is_directory()) {
            continue;
        }

        const auto &pool_path = pool_entry.path();
        std::cout << "Loading pool: " << pool_path << std::endl;
        schemas_pool pool;

        for (auto &schema_entry:
std::filesystem::directory_iterator(pool_entry)) {
            if (!schema_entry.is_directory()) {
                continue;
            }

            const auto &schema_path = schema_entry.path();
            std::cout << "Loading schema: " << schema_path << std::endl;
            schema schm;
```

```
            for (auto &table_entry:
std::filesystem::directory_iterator(schema_entry)) {
                if (!table_entry.is_regular_file()) {
                    continue;
                }

                if (table_entry.path().filename().string().find("index_")
!= std::string::npos) {
                    continue;
                }

                const auto &table_path = table_entry.path();
                std::cout << "Found table: " << table_path << std::endl;

                std::cout << "Path to table file: " << table_entry.path()
<< std::endl;
                table tbl =
table::load_data_from_filesystem(table_path.string());
                auto tbl_name = table_path.filename().string();
                for (int i = 0; i < _format.length(); ++i) {
                    tbl_name.pop_back();
                }

                schm.insert(tbl_name, std::move(tbl));
            }

            pool.insert(schema_path.filename().string(), schm);
        }
        _data->insert(pool_path.filename().string(), pool);
    }
}
```

Метод *save_data_base_state* также ориентирован на работу в режиме *in-memory cache*. Он проверяет стратегию и, если она корректна, проходит по всем пулам данных, вызывая *insert_pool_to_file* для сохранения состояния каждого пула в файл. Эта функция, в свою очередь, обеспечивает постоянство данных, сохраняя их в файловой системе между сессиями работы программы.

Листинг 7. Методы save_data_base_state и insert_pool_to_file

```
void data_base::save_data_base_state() {
    if (get_strategy() != memory) {
        throw std::logic_error("void data_base::save_data_base_state() ->
invalid strategy for this operation");
    }
    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_file(string_key, std::move(target_schemas_pool));
        ++it;
```

```
    }
}

void data_base::insert_pool_to_file(const std::string &pool_name,
schemas_pool &&value) {
    std::filesystem::path inmem_storage_path = _instance_path;
    create_if_not_exists(inmem_storage_path);
    std::filesystem::path pool_path = inmem_storage_path / pool_name;
    create_if_not_exists(pool_path);
    auto it = value._data->begin_infix();
    auto it_end = value._data->end_infix();
    while (it != it_end) {
        auto schema_name = std::get<2>(*it);
        auto target_schema = std::get<3>(*it);

        std::filesystem::path schema_path = pool_path / schema_name;

        create_if_not_exists(schema_path);
        target_schema.insert_schema_to_file(schema_path);
        ++it;
    }
}
```

Реализация этих функций позволяет динамически загружать и сохранять схемы и таблицы из файловой системы, минимизируя потерю данных и повышая устойчивость системы. Использование файловой системы как постоянного хранилища данных в сочетании с *in-memory* кэшированием обеспечивает быстрый доступ к данным, улучшая общую производительность системы. Таким образом, задача успешно решает проблему управления состоянием базы данных, сохраняя баланс между быстродействием и надежностью.

## Заключение

В процессе выполнения данной курсовой работы была разработана и реализована программа на языке C++ (стандарт C++14 и выше), основная задача которой заключается в выполнении операций над коллекциями данных различных типов и контекстов хранения. Программа нацелена на обеспечение универсального подхода к работе с данными, а также на возможность эффективного управления этими данными как в оперативной памяти, так и на уровне файловой системы.

Ключевым элементом архитектуры программы является абстрактный класс *interface<tkey, tvalue>*. Он предоставляет универсальный интерфейс для работы с различными типами данных, определяя основные операции, такие как добавление, удаление, обновление и поиск записей по ключам. Этот подход обеспечивает гибкость и расширяемость системы, позволяя в будущем легко добавлять новые типы данных и реализации. Важно отметить, что данная абстракция способствует разделению ответственности, так как базовые операции с данными выделены в отдельный интерфейс, что облегчает последующее тестирование и поддержку программы.

Наследником данного интерфейса стал класс *data_base*, который выполняет конкретную реализацию работы с базой данных. Он позволяет эффективно управлять коллекциями данных, обеспечивая взаимодействие с хранилищем, как на уровне оперативной памяти, так и с файлами. Такой подход повышает универсальность системы, делая ее подходящей для применения в различных сценариях: от высокопроизводительных *in-memory* систем до долгосрочного хранения на жестком диске.

Особого внимания заслуживает разработанный класс *person_data*, который был создан для хранения информации о пользователях. В его основе лежат данные, спроектированные с помощью паттерна "Приспособленец" (*Flyweight*), который был внедрен для уменьшения расхода памяти при хранении большого числа объектов с повторяющимися данными. Использование данного паттерна позитивно сказалось на производительности программы, так как сократилась необходимость дублирования данных.

Ассоциативный контейнер, реализованный на основе *B-Tree*, обеспечивает быстрый доступ к данным, поддерживая операции вставки, удаления и поиска с логарифмической сложностью. Это делает его идеальным выбором для использования в сценариях, где требуется высокая производительность при работе с большими коллекциями данных.

Программа также демонстрирует свою универсальность и гибкость за счет поддержки работы как с данными, находящимися в оперативной памяти, так и с файлами на жестком диске. Это достигается благодаря тому, что пользователь

может выбрать нужный режим работы программы, задавая соответствующий параметр при запуске.

На этапе реализации программы были предусмотрены различные команды для выполнения всех необходимых операций над коллекциями данных, включая добавление новых записей, обновление существующих, удаление и поиск данных. Особое внимание было уделено функционалу по управлению коллекциями данных, схемами данных и пулами данных, что позволило добиться высокой степени модульности и удобства в работе с системой.

В итоге разработанная программа продемонстрировала высокую эффективность и надежность. Она успешно справляется с поставленными задачами, обеспечивая быстрое и надежное выполнение операций с данными как в оперативной памяти, так и на жестком диске. Использование абстрактных классов, реализация *B-Tree* и внедрение паттерна "Приспособленец" позволили достичь оптимального соотношения производительности и использования ресурсов.

## Список литературы


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

[2] Липпман Стенли Б., Жози Л., Му Барбара Э. Язык программирования C++. Базовый курс. Вильямс, 2019 г.

[3] Викиконспекты ИТМО [Электронный ресурс]: B-Дерево. *URL*: https://neerc.ifmo.ru/wiki/index.php?title=B-дерево (дата обращения 8.09.2024).

[4] Справочник языка C++ [Электронный ресурс] URL: https://en.cppreference.com/w/ (дата обращения: 8.09.2024)

## Приложение

Сафин Т. А. Приложение кода к проекту на GitHub. [Электронный ресурс]: URL: https://github.com/Ukorp/mp_os/tree/project