

**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное  
учреждение высшего образования**

**«Национальный исследовательский университет ИТМО»**

***Факультет Программной инженерии и компьютерной техники***

**Отчёт к практическому заданию №1**

**по «Низкоуровнему программированию»**

**Выполнил: Группа Р33312      Хайкин О.И.**

**Преподаватель:**

**Кореньков Ю.Д.**

**Санкт-Петербург, 2023**

## Цели

Целью данного задания является создание программного модуля, реализующего хранение в файле данных информации, поддерживающим операции выборки, вставки, обновления и удаления. Файл данных может достигать размером 10 гигабайт.

## Вариант:

Реляционные таблицы - чтение/запись

## Задачи

### Настройка репозитория

Первым делом был создан локальный git-репозиторий и 2 remote-репозитория - на Github'е и на Gitlab'е. Затем была произведена настройка репозитория - создание структуры директорий проекта, настройка системы сборки (в случае данного проекта - CMake), настройка файлов .clang-format и .clang-tidy и т.д.

Этот пункт можно считать подготовительным

### Проектирование и реализация модулей “снизу-вверх”

После настройки репозитория начались проектирование и реализация модулей. В данном проекте я реализовывал их снизу-вверх, начиная от самых “низкоуровневых” функций - работа непосредственно с файлом, работа со страницами и т.д., и поднимался до “высокоуровневых” функций - API модуля со структурами запросов, мэнэджер запросов и т.д.

## Тестирование

После реализации необходимого функционала я перешёл к созданию тестов для программы. Тесты включают в себя простые тесты, которые подключены к Github Actions, и стресс-тесты, необходимые для отчёта

## Подготовка отчёта

С информацией, полученной от стресс-тестов, можно было приступить к подготовке этого отчёта

# Описание работы

## Структура программы

Созданную программу можно разделить на 2 части: библиотека и тесты. Библиотека реализует весь функционал и собирается как отдельный модуль. Тесты же используют API предоставленный библиотекой и выполняют различные операции с хранилищем данных.

Для использования программы пользователем ему нужно самому написать свою программу, которая будет вызывать предоставленные функции.

## Структура библиотеки

Библиотека состоит из исходных файлов и набора заголовочных файлов. Заголовочные файлы разделены на 2 директории: public и private. Заголовки, находящиеся в public директории, предоставляют публичный интерфейс соответствующего исходника. Заголовки из private описывают защищённый интерфейс соответствующего исходника. Private-заголовки могут использоваться в других исходниках и обычно содержат информацию о полях структур.

Библиотека построена на объектной парадигме. Непрозрачные структуры данных, для которых заданы “методы” (функции, принимающие первым аргументом такие структуры в качестве “объекта”), будем называть далее классами.

*Примечание: в данной программе в ряде случаев используются принципы наследования и полиморфизма. В то же время, этот функционал не поддерживается для всех “классов”, а только для тех, для которых это было необходимо в контексте реализованной программы.*

## Модули библиотеки

### util

Тут всё просто - это утилитный модуль, содержащий функционал, не относящийся к реализации хранилища данных

### error

Это модуль, содержащий функционал, относящийся к системе ошибок в программе. Программа построена таким образом, чтобы вызываемые функции могли возвращать результат с ошибкой вызывающим функциям, а те могли в свою очередь “пробрасывать” эти результаты дальше.

## file

Это модуль, содержащий функционал, относящийся к работе с файлом, которые содержит страницы. Этот модуль абстрагирован от конкретики задания и “знает” только о работе с файлом и существовании в нём страниц - он даже не “знает” ничего о содержимом этих страниц. Рассмотрим подробнее следующие классы:

### file\_manager

Этот класс ничего не знает о таблицах, страницах или чём-то ещё. По сути, этот класс является обёрткой над вызовами системного API (функции вроде `fread`, `fwrite`...), предоставляя свой API для работы с файлом, соответствующий парадигме остальной программы

### page\_resolver

Этот класс всё ещё ничего не знает о таблицах. Он “оборачивает” `file_manager`, предоставляя API для работы со страницами - операции вроде “прочитать страницу по номеру” и “записать страницу по номеру”. При этом этот класс только реализует вычисления, связанные с доступом к страницам в файле, оставляя работу по выделению памяти на вызывающего

### page\_manager

Этот класс всё ещё ничего не знает о таблицах. Он “оборачивает” `page_resolver`, реализуя свой внутренний кэш для хранения в оперативной памяти загруженных страниц. Этот класс предоставляет API для получения страниц по номеру. В реализации `page_manager` будет искать страницу в своём кэше, и если не найдёт, то загрузит её через `page_resolver`, при этом выгрузив одну из страниц, уже находящихся в кэше.

## storage

Этот модуль находится “над” модулем `file`, реализуя работу с “предметами”, хранящимися на страницах и группами страниц. Модуль `storage` знает содержимое страниц и предоставляет API для работы с “предметами”, хранящимися на них, но всё ещё абстрагирован от таблиц из задания.

### page\_group\_manager

Этот класс реализует работу с группами страниц. Страницы в файле связаны между собой на подобие связного списка: каждая страница начинается с заголовка, который содержит номер следующей и предыдущей страниц. Группа страниц задается номером первой страницы в этом списке.

Этот класс также реализует работу со “свободными” страницами - они тоже являются группой страниц. Операции по удалению группы или освобождению страниц, предоставляемые в API этого класса, добавляют страницы в эту группу. Операции по созданию группы или добавления страницы в группу удаляют страницы из этой группы,

или, если свободные страницы кончились, запрашивает создание новой страницы у `page_manager`.

### `page_data_manager`

Этот класс реализует работу с предметами, расположенными на страницах (которые находятся в группе). Класс предоставляет API для добавления, чтения и удаления предметов из групп страниц. Этот класс ничего не знает о содержимом самих предметов.

### `database`

Этот модуль реализует работу с реляционными таблицами, пользуясь API из модуля `storage`.

### `table_manager`

Этот класс реализует работу с таблицами - операции по вставке, выборке, обновлению, удалению записей, а также операции по созданию и удалению таблиц.

### `database_manager`

Этот класс реализует работу с запросами, которые уже вызывают операции из API `table_manager`.

## Аспекты реализации

### Обработка ошибок

Для обработки ошибок в программе используется следующая парадигма:

- Функции, которые могут возвращать ошибку, использует тип возврата `result_t`. Этот тип содержит значение `RESULT_OK/RESULT_ERR` и ссылку на объект ошибки, при её существовании.
- Функции, вызывающие процедуры, возвращающие тип `result_t`, оборачивают вызов в макросы `TRY` и `CATCH`, что позволяет среагировать на получение ошибки. Наиболее частый кейс использования представляет из себя освободить данные, использованные в функции, и пробросить ошибку выше.

Пример:

```
TRY(initialize_meta_page(self, meta_contents));
CATCH(error, {
    page_data_manager_destroy(page_data_manager);
    free(meta_contents);
    free(self);
    THROW(error);
});
```

## Полиморфизм и наследование

В программе в ряде модулей используются принципы наследования и полиморфизма. Рассмотрим это на примере модуля выражений (expression)

Выражения используются в предикатах и запросах на обновление данных. Выражения позволяют представить литералы, значения из столбца записи, а также вычисленные из них значения - арифметические операции, операции сравнения, логические операции.

Пример:

```
struct predicate *where = predicate_of(expr_of(
    column_expr(TABLE_NAME(), STATUS_COL(), COLUMN_TYPE_BOOL),
    literal_expr((bool)true), comparison_operator(NEQ, COLUMN_TYPE_BOOL)));
```

Данный предикат выполнится, если значение в столбце STATUS\_COL будет НЕ равно значению true.

Для реализации такой системы используется следующий подход:

Существует интерфейс, предоставляющий следующий API:

```
struct i_expression;

column_type_t expression_get_type(struct i_expression *self);
result_t expression_get(struct i_expression *self, struct record *record,
    column_value_t *result);
struct i_expression *expression_clone(struct i_expression *self);
void expression_destroy(struct i_expression *self);
```

И существуют имплементации, предоставляющие следующий API для их создания:

```
struct i_expression *literal_expr_uint64(uint64_t value);
struct i_expression *literal_expr_int32(int32_t value);
struct i_expression *literal_expr_string(str_t value);
struct i_expression *literal_expr_bool(bool value);
struct i_expression *literal_expr_float(float value);

struct i_expression *column_expr(str_t table_name, str_t column_name,
    column_type_t column_type);

struct i_expression *expr_of(struct i_expression *first,
    struct i_expression *second,
    struct i_expression_operator
    *expression_operator);
```

“Под капотом” интерфейс представляет из себя набор “общих” полей и полей для указателей на имплементации методов:

```

struct i_expression {
    column_type_t type;
    result_t (*get_impl)(struct i_expression *self, struct record *record,
                        column_value_t *result);
    struct i_expression *(*clone_impl)(struct i_expression *self);
    void (*destroy_impl)(struct i_expression *self);
};

```

Имплементации, в свою очередь, содержат родителя как поле, что позволяет использовать указатель на имплементацию как указатель на интерфейс - а это уже позволяет достичь полиморфизма:

```

struct literal_expression {
    struct i_expression parent;
    column_value_t value;
};

struct column_expression {
    struct i_expression parent;
    str_t table_name;
    str_t column_name;
};

struct operator_expression {
    struct i_expression parent;
    struct i_expression *first;
    struct i_expression *second;
    struct i_expression_operator *expression_operator;
};

```

## Результаты

### Артефакты

В результате сборки программы создаются следующие артефакты:

- Файл библиотеки, предназначенный для линковки с пользовательской программой.
- Исполняемые файлы тестов, линкующиеся с файлом библиотеки

В результат запуска benchmark-тестов будет также создан набор графиков, отражающих время работы теста и размер файла с данными по его окончанию. Эти графики будут предоставлены ниже.

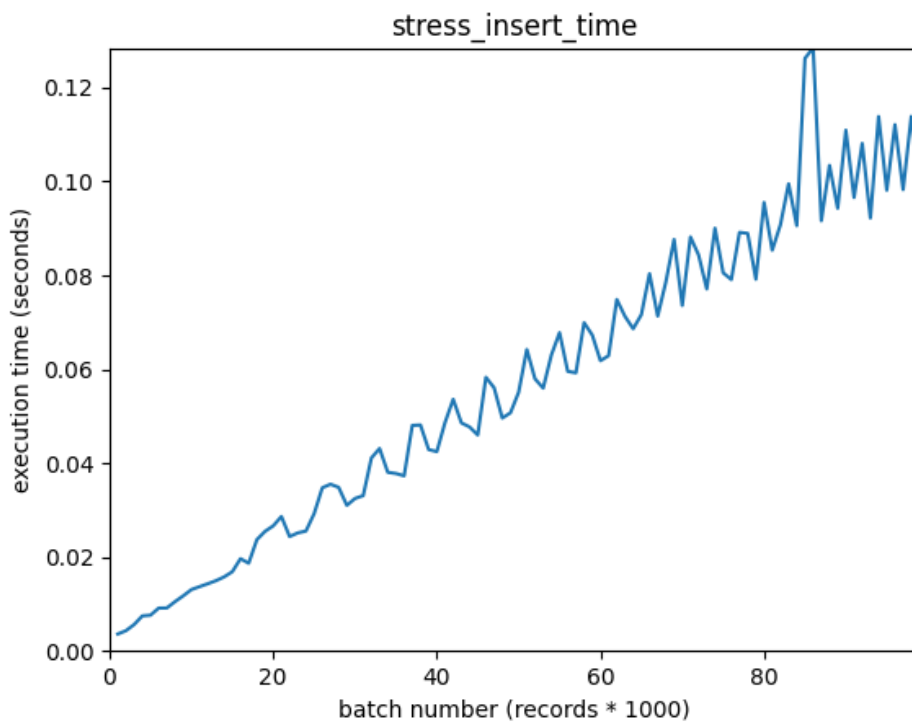
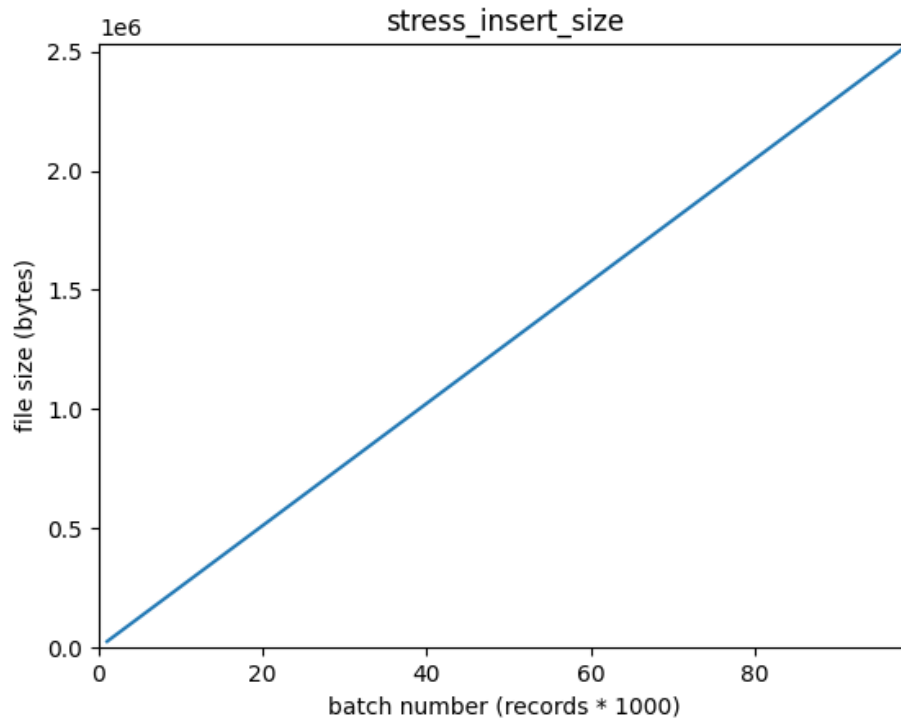
## Результаты тестов

Для каждого теста будут представлены 2 графика. Первый график отражает размер файла относительно количества записей N. Второй график отражает время действия программы относительно количества записей N.



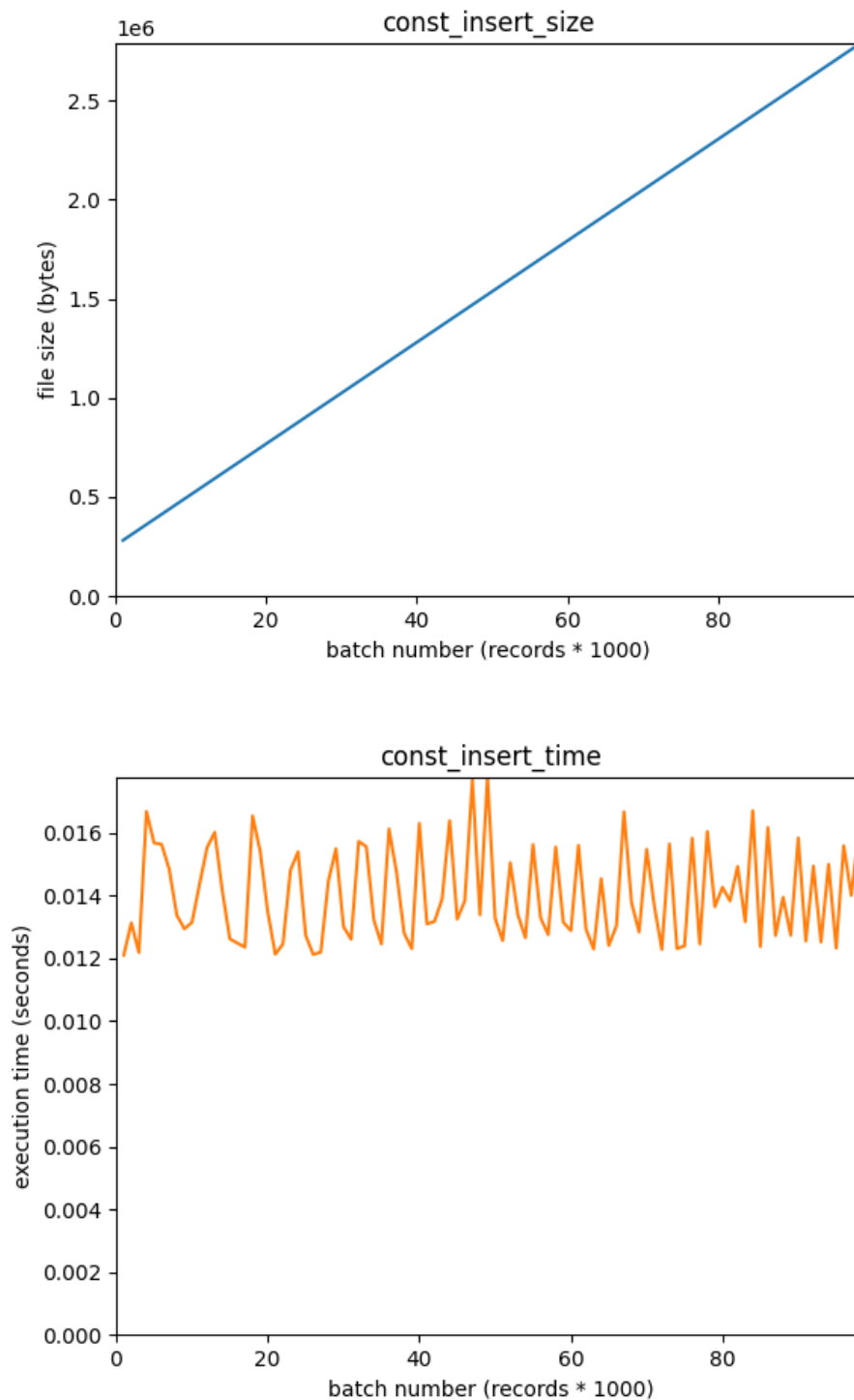
## Вставка N записей

Данный тест выполняет создание файла данных и вставку в него N записей. Он показывает, что время вставки набора записей пропорционально числу этих записей.



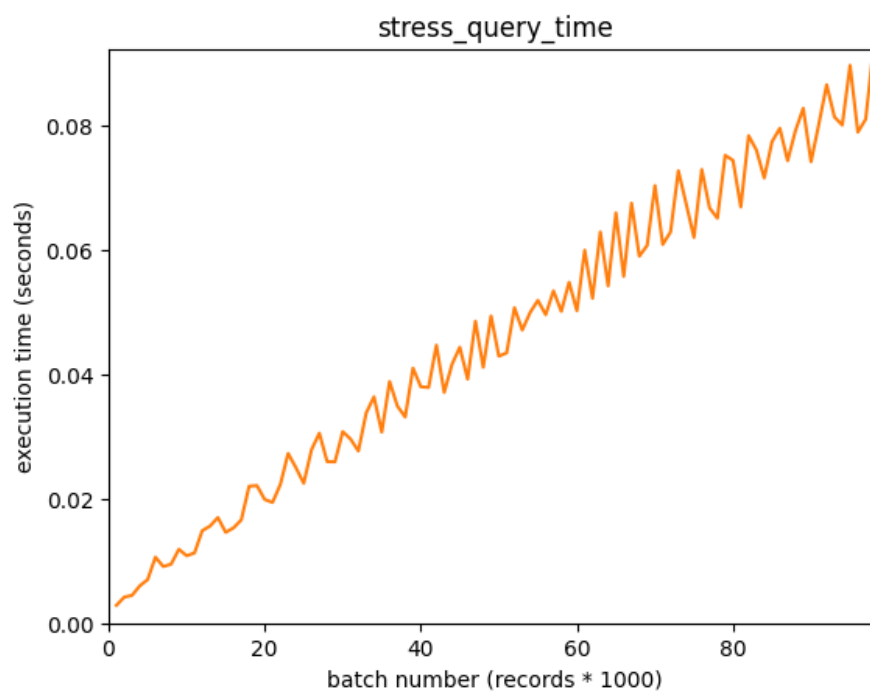
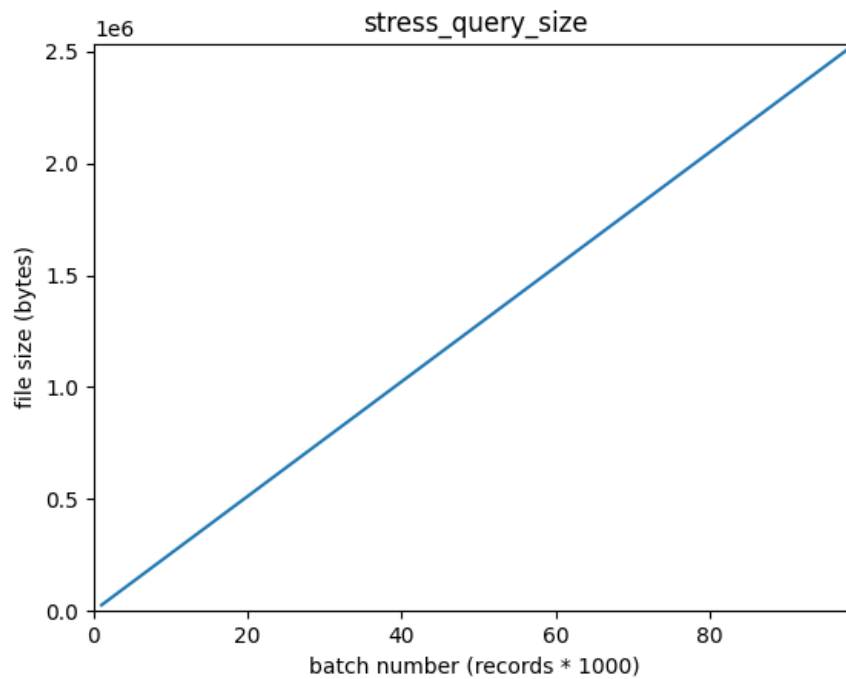
## Вставка M записей в файл с N записями

Данный тест выполняет вставку константного числа M записей в файл, уже содержащий N записей. Он показывает, что время вставки некоторого числа записей в файл не зависит от числа записей, уже содержащихся в нём



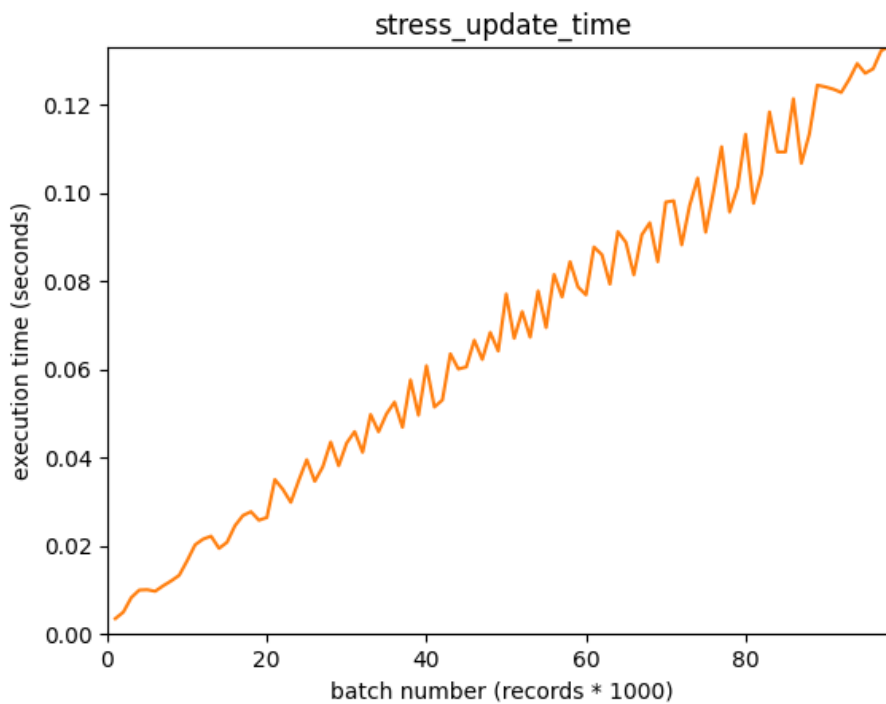
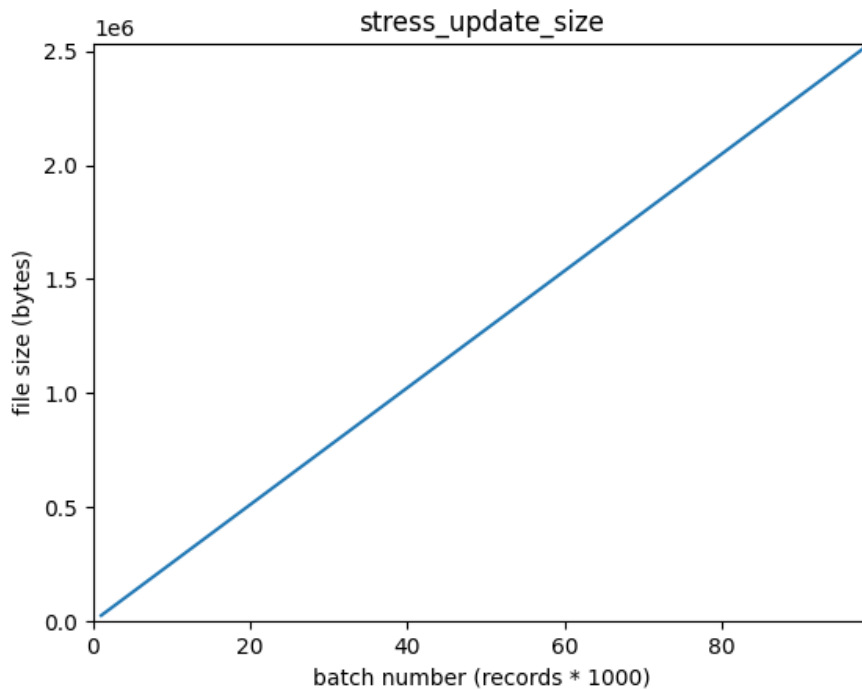
## Выборка N записей

Данный тест выполняет выборку из файла, содержащего N записей в рассматриваемой таблице, по условию. Он показывает, что время выборки записей по условию пропорционально общему числу записей в таблице.



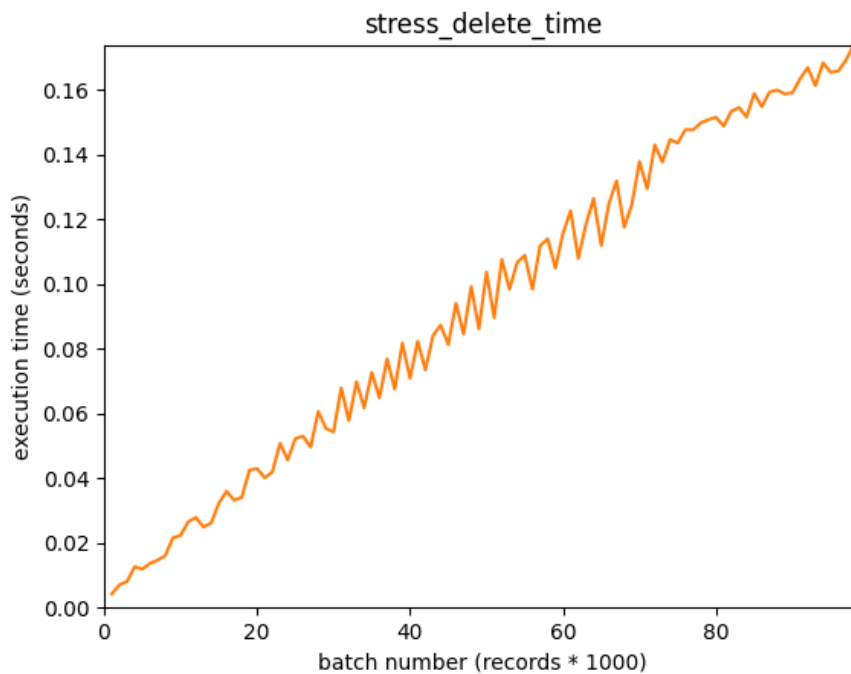
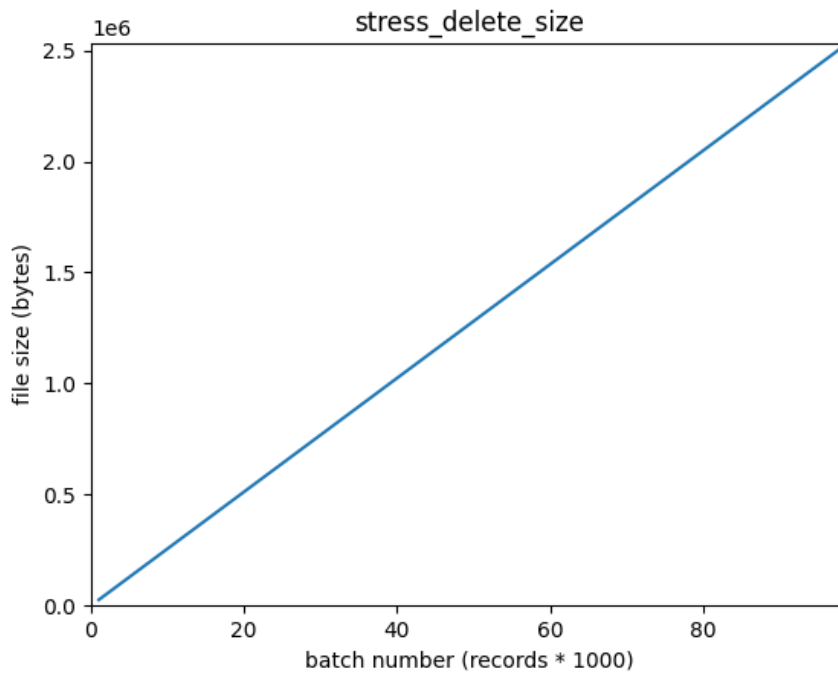
## Обновление N записей

Данный тест выполняет обновление в файле, содержащем N записей в рассматриваемой таблице, по условию. Он показывает, что время обновления по условию пропорционально общему числу записей в таблице.



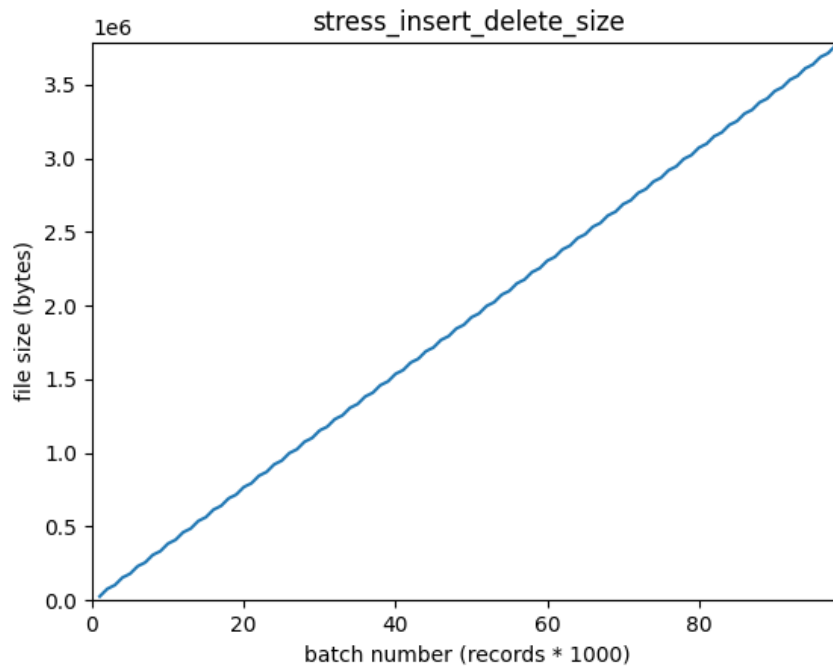
## Удаление N записей

Данный тест выполняет удаление в файле, содержащем N записей в рассматриваемой таблице, по условию. Он показывает, что время удаления по условию пропорционально общему числу записей в таблице.



## Вставка-удаление записей

Данный тест выполняет вставку  $N$  записей в таблицу, удаление половины из них, и вставку  $N/2$  записей после. Цель этого теста показать, что после таких операций размер файла пропорционален получившемуся числу записей в нём.



# Выводы

В результате выполнения задания был создан модуль, реализующий хранение, доступ и модификацию информации в файле. По результатам тестов можно судить, что временная сложность выполнения системой запросов соответствует требуемой. Размеры файла амортизировано пропорциональны размеру данных в нём.

В ходе выполнения данного задания я научился проектировать с нуля архитектуру приложения на языке C и научился пользоваться парадигмами полиморфизма, наследования и инкапсуляции в контексте языка C.