Университет ИТМО

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.04 Информатика и вычислительная техника
Дисциплина «Низкоуровневое программирование»

Отчет

По лабораторной работе №1

Вариант 1

Выполнил:
Степанов М.А.
Преподаватель:
Кореньков Ю. Д.

Санкт-Петербург, 2022 г.

Цель:

Создать модуль, реализующий хранение в одном файле большого количества данных (выборку, размещение и гранулярное обновление).

Задачи:

1. Спроектировать структуры данных для представления информации в оперативной памяти.
2. Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним.
3. Используя в сигнатурах только структуры данных из п.1, реализовать публичный интерфейс с операциями добавления, обновления, удаления данных.
4. Реализовать тестовую программу для демонстрации работоспособности решения
5. Построить графики зависимости скорости выполнения операций и размера файлов в зависимости от количество элементов в файле.

Описание работы:

В программе присутствует несколько модулей:

- *basic_types* — отвечает за определение базовых типов и операций crud, обозначающие используемые структуры данных.
- *basic_tools* — функции io и открытия файла, а также рассчета конечных размеров различных структур данных.
- *adv_types* — функции преобразований структур данных для записи их в файл и удобных манипуляций.
- *adv_tools* — высокоуровневые функция работы с файлом, направленные на удобство заполнение файла.
- *generator* — модуль генерации пустого файла.
- *crud_tools* — инкапсуляция манипуляций с данными, вынесенная в интерфейс в виде базовых crud функций.
- *commands* — команды для консольной работы с программой.
- *ui* — обработчик консольных команд.

При запуске программы, вы переходите в отдельную псевдоконсоль, в которой можете выполнять команды манипуляции с файлом.

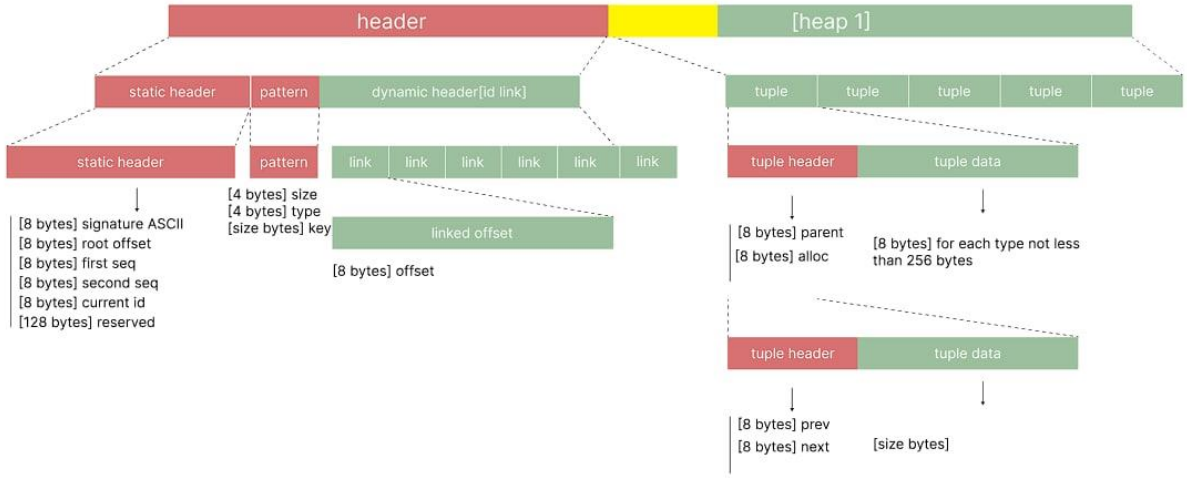
Программа может быть запущена, как и в обычном режиме, так и в режиме парсинга данных из файла.

```
Welcome to bigdata program!
File opened successfully!
Type 'help' for available commands info.
We notice, you haven't got initialized file yet.
So, you need to make file pattern...
Firstly, enter how many fields in each tuple do you need: 2
Then you need to describe each field:
--- Field 0   ---
Enter field name: name
0. Boolean
1. Integer
2. Float
3. String
Choose field type by entering number: 3
--- Field 1   ---
Enter field name: code
0. Boolean
1. Integer
2. Float
3. String
Choose field type by entering number: 1
>>> add
Enter fields of new tuple
name                :Mike
code                :72
Write parent id     :0
TUPLE   0 INSERTED
```

```
>>> find
Choose type of find
1. Get full tree
2. Find single by id
3. Find by parent
4. Find by condition
Type                :1
--- TUPLE   0 ---
name                : Mike
code                : 72
```

Аспекты реализации:

Предлагаемая модель данных:



Файл состоит из двух основных частей — секция с метаданными и секция с хранимыми структурами данных.

В секции с метаданными хранится информация о типе элементов, названии полей, и массив индексов с указателями на место в памяти, где хранятся соответствующие элементы.

В секции с хранимыми структурами данных хранятся элементы с их значениями. Здесь для оптимизации хранения используется разделение элементов данных на фундаментальные (которые описывают непосредственно вершины дерева) и строковые (которые описывают хранимые строки и ссылки на которые не хранится в метаданных). В фундаментальных элементах указывается лишь указатель на строки в файле, по которым их можно найти.

Особенности алгоритмов с примерами кода:

Для избежания фрагментации памяти в файле был использован следующий алгоритм удаления данных. Благодаря разделению элементов данных на фундаментальные и

строковые, стало возможным сделать структуры данных, хранимые в файле, равного размера, поэтому при удалении любого элемента нам необходимо лишь взять крайний элемент и переместить его на место удаляемого.

```
enum crud_operation_status swap_with_last(FILE *file, uint64_t offset, uint64_t size) {
    uint64_t full_size = size + sizeof(union tuple_header);
    fseek( stream: file,  off: 0,  whence: SEEK_END);
    fseek( stream: file,  off: -(int64_t) full_size,  whence: SEEK_END);
    return swap_tuple_to(file,  pos_from: ftell( stream: file),  pos_to: offset,  tuple_size: full_size);
}
```

При использовании данного алгоритма возникла необходимость изменения ссылок, которые ссылались на крайний элемент. Для фундаментальных элементов данная проблема решается тривиально — все ссылки на родителей у нас сделаны через индексы, поэтому требовалось лишь изменить указатель в массиве индексов. Для строк же пришлось применять иной подход. Элементы, описывающие строки имеют ссылки на прошлую и следующую часть строки. Поэтому ссылку на прошлую часть строку у первого строчного элемента мы указали на фундаментальный элемент, которому она принадлежит, поэтому при перестановке такого строчного элемента нам требовалось лишь перейти по ссылке и изменить ссылку в фундаментальном элементе.

```
enum crud_operation_status insert_string_tuple(FILE *file, char *string, size_t tuple_size, uint64_t par_pos, uint64_t *str_pos) {
    size_t len = strlen( s: string);
    size_t count = len / tuple_size + (len % tuple_size ? 1 : 0);
    struct tuple *temp_tuple = malloc( size: sizeof(struct tuple));
    char *temp_tuple_content = string;
    size_t pos = (size_t) ftell( stream: file);
    uint64_t fake_pos;
    fseek( stream: file,  off: 0,  whence: SEEK_END);
    *str_pos = ftell( stream: file);
    for (size_t iter = 0; count > iter; iter++) {
        if (count - 1 == iter) {
            temp_tuple->header.next = 0;
        } else {
            temp_tuple->header.next = pos + (tuple_size + sizeof(union tuple_header)) * (iter + 1);
        }
        if (0 == iter) {
            temp_tuple->header.prev = par_pos;
        } else {
            temp_tuple->header.prev = pos + (tuple_size + sizeof(union tuple_header)) * (iter - 1);
        }
        temp_tuple->data = (uint64_t *) (temp_tuple_content + tuple_size * iter);
        insert_new_tuple(file,  tuple: temp_tuple,  full_tuple_size: tuple_size + sizeof(union tuple_header),  tuple_pos: &fake_pos);
    }
    free( ptr: temp_tuple);
    return 0;
}
```

К тому же, соответственно, появилась необходимость различать строчные и фундаментальные элементы, но это проблема решалась легко — необходимо было лишь проверить положение элемента в файле на наличие его в массиве индексов.

Также для удаления нам был необходим рекурсивный алгоритм обхода всех дочерних элементов. Для этого я использовал алгоритм спуска в глубину, он показался более эффективным по сравнению с поиском в ширину, во-первых, потому что поиск в глубину будет хранить ссылки лишь на элементы, которые находятся в текущей ветке обхода, во-вторых, потому что не было необходимости хранить информацию о предыдущих этапах спуска.

```c
static enum crud_operation_status remove_recursive_tuple_with_values
        (FILE *file, uint64_t id, uint32_t *types, size_t pattern_size) {

    uint64_t size = get_real_tuple_size(pattern_size);
    uint64_t offset = remove_from_id_array(file, id);
    if (offset == NULL_VALUE) return CRUD_INVALID;
    struct uint64_list *childs = get_childs_by_id(file, id);
    for(struct uint64_list *iter = childs; iter != NULL; iter = iter->next) {
        remove_recursive_tuple_with_values(file, id: iter->value, types, pattern_size);
    }
    struct tuple *cur_tuple;
    fseek( stream: file, off: (int32_t) offset, whence: SEEK_SET);
    read_tuple( tuple: &cur_tuple, file, pattern_size);
    for(size_t iter = 0; iter < pattern_size; iter++) {
        if (types[iter] == STRING_TYPE) {
            remove_string_from_file(file, offset: cur_tuple->data[iter], size);
        }

    }
    free_uint64_list( result: childs);
    free_tuple( tuple: cur_tuple);
    return swap_with_last(file, offset, size);
}
```

Также возник вопрос о расширении массива индексов. Для этого было принято решение использовать тот же функционал, только теперь мы перемещали не элемент из конца в середину, а элемент из начала в конец (и соответственно расширение массива индексов).

```c
enum crud_operation_status append_to_id_array(FILE *file, uint64_t offset) {
    fseek( stream: file, off: 0, whence: SEEK_SET);
    struct tree_header *header = malloc( size: sizeof(struct tree_header));
    size_t pos;
    read_tree_header(header, file, fpos: &pos);
    if (!((header->subheader->cur_id + 1) % get_real_id_array_size( pattern_size: header->subheader->pattern_size, cur_id: header->subheader->cur_id))){
        uint64_t from = ftell( stream: file);
        fseek( stream: file, off: 0, whence: SEEK_END);
        swap_tuple_to(file, pos_from: from, pos_to: ftell( stream: file), tuple_size: get_real_tuple_size( pattern_size: header->subheader->pattern_size));
    }
    header->id_sequence[header->subheader->cur_id] = offset;
    header->subheader->cur_id++;
    write_tree_header(file, header);
    free_tree_header(header);
    return 0;
}
```

Оценка функций по ключевым параметрам:

- O(1) – добавление элемента.
- O(1) – поиск по индексу.
- O(n) – поиск по соответствию поля.
- O(m) – удаление элемента по индексу, от кол-ва зависимых элементов.
- O(1) – обновление элемента по индексу.
- O(n) – по количеству занимаемой памяти от кол-ва элементов

Результаты:

- Описаны ключевые структуры для работы программы и методы работы с ними.
- Был разработан консольный интерфейс для взаимодействия с программой.
- Были проведены тесты, оценивающие ключевые характеристики работы программы, результаты тестирования совпали с оценкой произведенной при проектировании модели данных.

Тестирование проводилось при помощи библиотеки time.

Листинг функций для тестирования:

```c
void time_add_wrapper(FILE *file, uint64_t *fields, uint64_t parent_id){
    clock_t cl = clock();
    add_tuple(file, fields, parent_id);
    printf("%f\n", (double) (clock() - cl) / CLOCKS_PER_SEC);
}

void size_add_wrapper(FILE *file, uint64_t *fields, uint64_t parent_id){
    add_tuple(file, fields, parent_id);
    fseek(file, 0, SEEK_END);
    printf("%ld\n", ftell(file));
}

void time_add_get_wrapper(FILE *file, uint64_t *fields, uint64_t parent_id, uint64_t id){
    add_tuple(file, fields, parent_id);
    clock_t cl = clock();
    get_tuple(file, &fields, id);
    printf("%f\n", (double) (clock() - cl) / CLOCKS_PER_SEC);
}
```

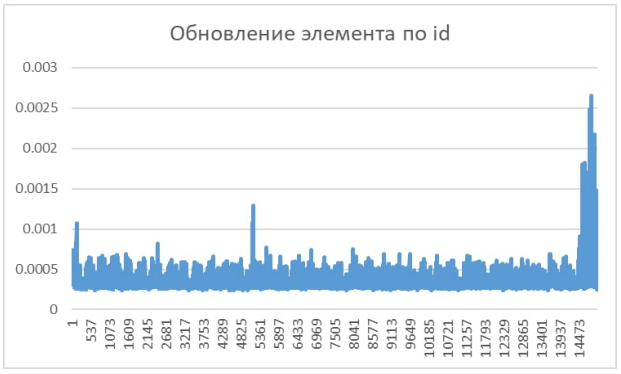
```c
void time_add_get_cond_wrapper(FILE *file, uint64_t *fields, uint64_t parent_id){
    add_tuple(file, fields, parent_id);
    uint64_t c = 3342;
    struct result_list_tuple *res;
    clock_t cl = clock();
    find_by_field(file, 1, &c, &res);
    printf("%f\n", (double) (clock() - cl) / CLOCKS_PER_SEC);
}

void time_add_remove_wrapper(FILE *file, uint64_t *fields, uint64_t parent_id, uint64_t id){
    add_tuple(file, fields, parent_id);
    struct result_list_tuple *res;
    clock_t cl = clock();
    remove_tuple(file, id);
    printf("%f\n", (double) (clock() - cl) / CLOCKS_PER_SEC);
}

void time_remove_wrapper(FILE *file, uint64_t id){
    clock_t cl = clock();
    remove_tuple(file, id);
    printf("%f\n", (double) (clock() - cl) / CLOCKS_PER_SEC);
}

void time_add_update_wrapper(FILE *file, uint64_t *fields, uint64_t parent_id, uint64_t id){
    add_tuple(file, fields, parent_id);
    uint64_t c = 3342;
    clock_t cl = clock();
    update_tuple(file, 1, &c, id);
    printf("%f\n", (double) (clock() - cl) / CLOCKS_PER_SEC);
}
```

Графики полученных данных по итогу тестирования:



Выводы:

1. Была реализована программа, способная работать с одним файлом, содержащим большое количество данных (большое количество данных не имеет какой-то конкретной оценки ключевым фактором является асимптотика времени выполнения операций над этими данными).
2. Тесты показали, что модель соответствует заявленным требования по производительности и способна выполнять необходимые функции при большем количестве данных при этом не увеличивая время выполнения или увеличивая его линейно.