

федеральное государственное автономное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ОТЧЕТ

по лабораторной работе №1

по дисциплине «**Низкоуровневое программирование**»

Вариант 11

Автор: Кулаков Н. В.

Факультет: ПИиКТ

Группа: Р33312

Преподаватель: Кореньков Ю.Д.



УНИВЕРСИТЕТ ИТМО

Санкт-Петербург 2022

Цели:

Создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное обновление) информации общим объёмом от 10GB соответствующего варианту вида.

-rw-r--r-- 1 nikit nikit 11G дек 14 21:14 .db-very-large.bin

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 со следующими операциями над файлом данных:

- Добавление, удаление и получение информации о элементах схемы данных, размещаемых в файле данных, на уровне, соответствующем виду узлов или записей
- Добавление нового элемента данных определённого вида
- Выборка набора элементов данных с учётом заданных условий и отношений со смежными элементами данных (по свойствам/полям/атрибутам и логическим связям соответственно).
- Обновление элементов данных, соответствующих заданным условиям.
- Удаление элементов данных, соответствующих заданным условиям.

4) Реализовать тестовую программу для демонстрации работоспособности решения.

5) Результаты тестирования по п.4 представить в составе отчёта. Включить графики на основе тестов, демонстрирующие амортизированные показатели ресурсоёмкости по п. 4.

Описание работы:

Программа представляет из себя библиотеку, предоставляющую возможность написания своих собственных запросов. Перейду к реализованным возможностям.

Создание, открытие, закрытие, удаление бд:

```
1 #pragma once
2
3 struct dbms *dbms_create(const char *fname);
4 struct dbms *dbms_open(const char *fname);
5 void dbms_close(struct dbms **dbms_ptr);
6 void dbms_remove(struct dbms **dbms_ptr);
```

Создание, удаление таблицы на уровне dto; добавление колонок, подсчет колонок в таблице; ограничение на колонки (is_null, is_unique не работают в плане попыток записи в бд, однако используются на уровне фильтров запросов, об этом далее):

```
3 enum dto_table_column_type {
4     DTO_COLUMN_INT32 = 0,
5     DTO_COLUMN_DOUBLE,
6     DTO_COLUMN_STRING,
7     DTO_COLUMN_BOOL
8 };
9
10 typedef struct dto_table_column_limits {
11     bool is_null;
12     bool is_unique;
13 } dto_table_column_limits;
14
15 struct dto_table *dto_table_construct(const char *name);
16 void dto_table_destruct(struct dto_table **table_ptr);
17 void dto_table_add_column(struct dto_table *table, const char *name,
18                           const enum dto_table_column_type type,
19                           const struct dto_table_column_limits lims);
20 int dto_table_column_cnt(const struct dto_table *table);
```

Создание, удаление таблиц, проверка существования таблицы:

```
7
8 bool table_exists(struct dbms *dbms, const char *name);
9 bool table_create(struct dbms *dbms, struct dto_table *table);
10 bool table_drop(struct dbms *dbms, struct dto_table *table);
```

Создание списка строк на уровне dto, добавление строк в этот список (копируются указатели, а не сами данные):

```
3 struct dto_row_list dto_row_list_construct();
4 void dto_row_list_destruct(struct dto_row_list *lst);
5 void dto_row_list_append(struct dto_row_list *lst, const void *row[]);
```

Вставка списка строк в бд (оптимизирована вставка сразу нескольких строк без перезагрузки блока):

```
8 int row_list_insert(struct dbms *dbms, const char *table_name,  
9 struct dto_row_list *list);
```

Сейчас перейдем к запросам на `select`, `update`, `delete`. Сам запрос (`plan`) строится из нод, каждая из которых выполняет свою функцию. Ниже перечислены основные типы `plan_nodes`. Для формирования запросы мы оборачиваем ноды.

[illegible]

Запросы пишутся в объектом lua стиле, ниже пример (можно было маллокнуть base, сделать тип приватный интерфейс для пользователя, однако не хочется лишний раз маллокать).

Пользователь может вызывать функции из примера ниже. Можно было разрешить запускать эти функции только для терминальных нод, но думаю ничего страшного нет, если я разрешил для всех).

```

57 // plan_source {{{
58 struct plan_source {
59     struct plan base;
60
61     struct dbms *dbms;
62     // iterator
63     struct tp_iter *iter;
64
65     INHERIT const struct plan_table_info *(*get_info)(void *self, size_t *size);
66     INHERIT struct tp_tuple **(*get)(void *self);
67     INHERIT bool (*end)(void *self);
68
69     OVERRIDE bool (*next)(void *self);
70     OVERRIDE void (*destruct)(void *self_ptr);
71 };

```

Вызов выглядит так:

```
// TEST select (passed)
{
    printf("Selection: \n");
    struct plan_select *select_table1 = plan_select_construct_move(
        | | plan_source_construct("table1", dbms), "temp-table1");

    size_t arr_size;
    const struct plan_table_info *table_info =
        | | select_table1->get_info(select_table1, &arr_size);

    select_table1->start(select_table1);

    while (!select_table1->end(select_table1)) {
        struct tp_tuple **tpt = select_table1->get(select_table1);
        print_table_tuple(tpt[0], table_info[0].dpt, table_info[0].col_info, dbms);

        select_table1->next(select_table1);
    }
    select_table1->destruct(select_table1);
}
```

Фильтры для запросов (where в sql) реализуются аналогичным образом. Пример абстрактного фильтра и const (константа колонки). Это приватных хедер, вызывается через plan_filter:

```
15
16 // fast {{{
17 struct fast {
18     enum fast_type type;
19     // result of calc
20     void *res;
21     enum table_column_type res_type;
22
23     void (*compile)(void *self, size_t pti_size, struct plan_table_info *info_arr);
24     void (*destruct)(void *self);
25
26     void *(*calc)(void *self); // void * - returned result
27     void (*pass)(void *self, const struct tp_tuple **tuple_arr);
28 };
29 // }}}
30
31 // fast_const {{{
32 struct fast_const {
33     struct fast base;
34
35     struct dbms *dbms;
36 };
```

Пользователю предоставляется возможность создавать и комбинировать фильтры, соответственно:

```

8 struct fast_const *fast_const_construct(enum dto_table_column_type col_type,
9                                         const void *value, struct dbms *dbms);
10
11 struct fast_column *fast_column_construct(const char *table_name,
12                                           const char *column_name, struct dbms *dbms);
13
14 struct fast_unop *fast_unop_construct(void *parent, struct fast_unop_func *fuf,
15                                       struct dbms *dbms);
16
17 struct fast_binop *fast_binop_construct(void *p_left, void *p_right,
18                                         struct fast_binop_func *fbf, struct dbms *dbms);

```

fast_unop_func, fast_binop_func задаются разработчиком, так как в них осуществляется доступ struct fast. Ниже представлены функции, которые реализованы, но можно конечно реализовать и больше:

```

5 #define EXT_UNOP(name) extern struct fast_unop_func name
6 #define EXT_BINOP(name) extern struct fast_binop_func name
7
8 struct fast_unop;
9 struct fast_binop;
10
11 // fast_unop {{{
12 struct fast_unop_func {
13     void (*func)(struct fast_unop *self, void *arg);
14     enum table_column_type ret_type;
15 };
16
17 EXT_UNOP(BOOL_NOT);
18 // }}}
19
20 // fast_binop {{{
21 struct fast_binop_func {
22     void (*func)(struct fast_binop *self, void *arg1, void *arg2);
23     enum table_column_type ret_type;
24 };
25
26 EXT_BINOP(DOUBLE_LARGER);
27 EXT_BINOP(DOUBLE_EQUALS);
28
29 EXT_BINOP(INT32_EQUALS);
30 EXT_BINOP(INT32_LARGER);
31
32 EXT_BINOP(BOOL_OR);
33 EXT_BINOP(BOOL_AND);
34
35 EXT_BINOP(STRING_EQUALS);

```

Примеры группировки через фильтры и написание полноценного запроса, конкретнее update:


```

struct plan_source *so1 = plan_source_construct("table1", dbms);
// struct plan_source *so2 = plan_source_construct("table2", dbms);
// struct plan_source *so3 = plan_source_construct("table1", dbms);
// struct plan_cross_join *j1 = plan_cross_join_construct_move(so1, so2);
// struct plan_cross_join *j2 = plan_cross_join_construct_move(j1, so3);

struct fast_column *fc_name = fast_column_construct("table1", "name", dbms);

struct fast_const *fc_nikit =
| | fast_const_construct(COLUMN_TYPE_STRING, "nikita", dbms);

struct fast_column *fc_weight = fast_column_construct("table1", "weight", dbms);
struct fast_binop *fb1 =
| | fast_binop_construct(fc_name, fc_nikit, &STRING_EQUALS, dbms);

const double min_val = 80;
struct fast_const *fc_weight80 =
| | fast_const_construct(COLUMN_TYPE_DOUBLE, &min_val, dbms);

struct fast_binop *fc_max_wight =
| | fast_binop_construct(fc_weight, fc_weight80, &DOUBLE_LARGER, dbms);

struct fast_binop *fc_and =
| | fast_binop_construct(fc_max_wight, fb1, &BOOL_AND, dbms);

// struct plan_filter *f = plan_filter_construct_move(j1, fb1);
struct column_value *cols = malloc(sizeof(struct column_value));
cols[0].column_name = "name";
cols[0].column_value = "perestaralsiya";

struct plan_filter *fi = plan_filter_construct_move(so1, fc_and);
struct plan_update *se = plan_update_construct_move(fi, 1, cols);

size_t ti_size;
struct plan_table_info ti = se->get_info(se, &ti_size)[0];

se->start(se);
while (!se->end(se)) {
    tp_tuple *tpt = se->get(se)[0];
    print_table_tuple(tpt, ti.dpt, ti.col_info, dbms);
    se->next(se);
}
se->destruct(se);

```

запрос аналогичен sql:

update from table1 set table1.name = «perestaralsiya» where table1.name = «nikita» and table1.weight > 80;

print_table_tuple — функция в util/printers.h

Другие примеры запросов и основных функций можно найти в директории tests (тесты запускаются через gtest), а в app/ для select, update, delete и фильтров.

Аспекты реализации:

Программа разделена на несколько уровней:

- уровень структур, итераторы по одной структуре
- уровень ввода/вывода
- уровень dbms (алгоритмы работы с файлами, реализация базовых операций, итераторы)
- уровень пользовательских функций (добавление, удаление, изменение записей в таблице, редактирование схемы)

1) Уровень структур (src/dbms/io/{page,meta}):

В начале файла (базы данных) хранится метайнформация, затем после нее следуют данные, которые упаковываются в блоки, которые можно конфигурировать.

1.1) Метаинформация представлена в виде (dbms/io/meta/meta.h):

```
/* Stored in file */
typedef struct meta {
    // file position (like heap position in malloc)
    fileoff_t pos_empty;

    // Database Page
    fileoff_t dp_last;

    // Last page of page allocator
    fileoff_t free_last;

    // Last pages of data distributor
    size_t slot_len;
    struct slot_page_entry slot_entries[];
} meta;
```

```
struct slot_page_entry {
    fileoff_t last;
    uint32_t slot_size;
    uint32_t slot_count;
};
```

dp_last — последняя таблица database_page.

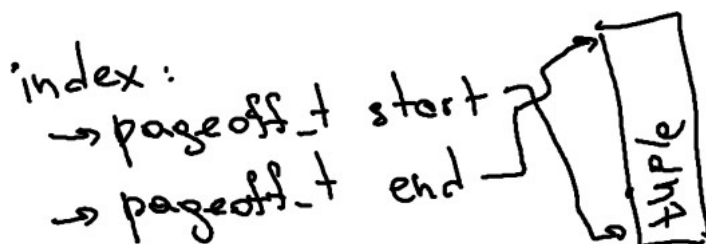
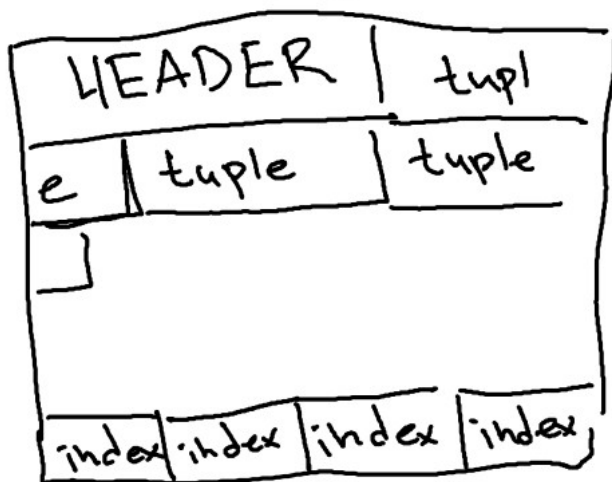
free_last — стек записей о пустых страницах вида struct {fileoff_t start — адрес с начала файла, pageoff_t size — размер страницы}.

slot_len, slot_entries (рассмотрены далее) — массив записей о различных страницах, предназначенных для данных нефиксированного размера (строк).

1.2) database_page (dbms/io/page/p_database.h)

Страница, в которой хранятся записи таблиц в бд. При добавлении таблицы запись добавляется в конец таблицы или создается новая таблица, которая линкуется с предыдущей и записывается таблица в нее. При удалении таблица помечается удаленной, никакие дырки не чистятся (ибо таблиц мало и это просто не рационально). Расположение:

database_page



tuple (aka dp_tuple)

```
// TYPE
typedef struct dpt_header {
    bool is_present;
    size_t cols;
    struct page_sso sso;

    fileoff_t tp_first; /* first and
    fileoff_t tp_last;

    fileoff_t td_last; /* pointer to
} dpt_header;

typedef struct dpt_col_limits {
    bool is_null;
} dpt_col_limits;

typedef struct dpt_column {
    int8_t type; /* table_column_type
    struct dpt_col_limits limits;
    struct page_sso sso;
} dpt_column;

typedef struct dp_tuple {
    struct dpt_header header;
    struct dpt_column columns[];
} dp_tuple;
```

tp_first, tp_last — первая и последняя страница таблицы соответственно,

is_null — может ли быть колонка null,

sso — имя таблицы или колонки, но хранится удивительным образом, об этом позднее,

is_present — является ли таблица удаленной или нет.

td_last — стек записей о страницах типа table_page, которые являются неполными.

1.3) `page_sso` — если строка размера $\leq \text{SSO_MXLEN}$, то она инлайнится, в противном случае создается запись `ssize` (размер строки) и `struct { fileoff_t — адрес страницы в файле, pageoff_t — смещение внутри страницы } po_ptr` (указатель, где лежит строка).

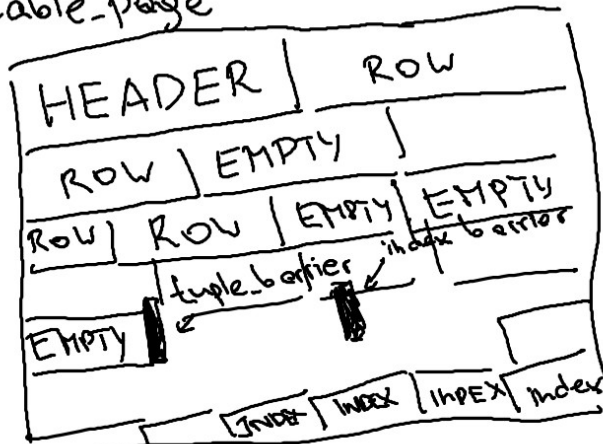
```
8 #define SSO_SSIZE_SIZE (sizeof(size_t) - sizeof(bool))
9 #define SSO_MXLEN (sizeof(struct _str_not_in))
10
11 /* Struct that stored if string is not inlined
12 * @ssize string size
13 * @ptr pointer to string in different page
14 */
15 #define STR_NOT_IN
16 {
17     unsigned char ssize[SSO_SSIZE_SIZE];
18     struct po_ptr po_ptr;
19 }
20 __attribute__((packed))
21 // Create non-anonymous class for c++ compatibility
22 struct _str_not_in STR_NOT_IN;
23
24 // Small string optimization struct with anonymous struct inside
25 typedef struct page_sso {
26     union {
27         struct STR_NOT_IN;
28         char name[SSO_MXLEN];
29     };
30     bool not_inline; // should be 0 if inline because string is null terminated
31 } __attribute__((packed)) page_sso;
32
33 size_t sso_to_size(const unsigned char *ssize);
34 void size_to_sso(size_t size, unsigned char *ssize);
35
36 #undef _STR_NOT_IN
```

Реализация конвертации `size_t` в `ssize` топорная, можно было на асике, но лень.

```
3 size_t sso_to_size(const unsigned char *ssize) {
4     size_t res = 0;
5     size_t mask = 0xff;
6     for (size_t i = 0; i != SSO_SSIZE_SIZE; ++i) {
7         res = (res << 8) | (mask & ssize[SSO_SSIZE_SIZE - 1 - i]);
8     }
9     return res;
10 }
11
12 void size_to_sso(const size_t size, unsigned char *ssize) {
13     size_t mask = 0xff;
14     for (size_t i = 0; i != SSO_SSIZE_SIZE; ++i) {
15         ssize[i] = (unsigned char)((size >> 8 * i) & mask);
16     }
17 }
```

1.4) `table_page` — страница, в которой хранятся строки

table-page

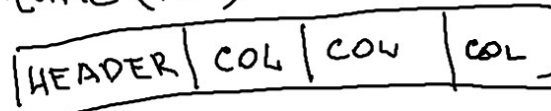


```
typedef struct slot_index {
    pageoff_t start;
} slot_index;

typedef struct table_header {
    struct base_header base;
    fileoff_t next;
    fileoff_t prev;

    pageoff_t tuple_barrier; // exclusive
    //
    pageoff_t index_barrier; // inclusive
    pageoff_t index_start; // if index_start == barrier then page is empty
} table_header;
```

tuple (row)



```
// NOTE: struct is packed inside tp_tuple that is why we need to add packed (else add
// columns not packed)
#define TPT_COLUMN_SIZE(enum_name) sizeof(struct tpt_column_##enum_name)
#define TPT_COLUMN_TYPE(enum_name) tpt_column_##enum_name
#define TPT_ENTRY_SIZE(enum_name)
    (TPT_COLUMN_SIZE(enum_name) - offsetof(struct tpt_column_##enum_name, entry))

// helpers
#define TPT_COLUMN_SIZE(enum_name) sizeof(struct tpt_column_##enum_name)
#define TPT_COL_TYPE(enum_name) tpt_column_##enum_name
#define TPT_ENTRY_SIZE(enum_name)
    (TPT_COLUMN_SIZE(enum_name) - offsetof(struct tpt_column_##enum_name, entry))

// define columns
TPT_COLUMN(bool, COLUMN_TYPE_BOOL)
TPT_COLUMN(int32_t, COLUMN_TYPE_INT32)
TPT_COLUMN(double, COLUMN_TYPE_DOUBLE)
TPT_COLUMN(struct page_sso, COLUMN_TYPE_STRING)

typedef struct tpt_header {
    bool is_present;
} tpt_header;

typedef struct tp_tuple {
    struct tpt_header header;
    struct tpt_column_base columns[]; // just the pointer to the first entry start (packed)
} tp_tuple;
```

Записи (tuple) фиксированного размера (заданы схемой), однако сами колонки могут быть разные по размеру если в них лежат разные типы. Однако можно явно найти, с какого адреса начинается та или иная колонка, если мы знаем «схему» данной таблицы, которая находится в `dp_tuple`.

`slot_index` — стек пустых слотов (tuple), куда можно добавить запись.

1.5) `slot_page` — страница, в которой лежат данные нефиксированного размера. Сами данные кладутся в слоты в соответствии с некоторым алгоритмом. Если данные не помещаются в максимальный слот, то в конце слота добавляется указатель и смещение (`ro_ptr`) на продолжение данных. Данные могут продолжаться как на этой же странице, та и на другой.

```
/*
Structure: data_page
|-----|
|HHHHHHH|TTTT| header.slot - size of header slot
|-----|
|      |TTTT|
|-----|
|      |      | I - empty slots
|-----|

*/

typedef struct slot_header {
    struct base_header base;
    size_t slot_size;
    size_t slot_count;
    pageoff_t slot_start; // slot number start
} slot_header;
```

```

5 // meta
6 static size_t DATA_PAGE_SLOT_SIZES[] = {64, 128, 256, 512, 1024};
7 static size_t DATA_PAGE_SLOT_COUNT[] = {32, 16, 8, 4, 2, 1};
8

```

Выше представлена конфигурация слотов: их размер и количество слотов в одной странице. Можно ограничить размер, сделав выравнивание, однако сделано как сделано, никто не жалуется.

Теперь вспомним, что было в meta, и поймем что это было за поле.

1.6) page_container — в нем хранятся записи о некоторых таблицах. В случае pagealloc (аллокатора страниц) — пустые страницы, в случае с таблицами — неполные страницы, куда можно добавлять записи.

```

/*
Structure: container_page
|-----|
|NNNNNNN| | H - header
|-----| | T - entries
|
|-----|
|          TTTTT|
|-----|
*/

```

```

typedef struct container_header {
    struct base_header base;
    fileoff_t prev;

    pageoff_t start; // points to the
} container_header;

```

```

typedef struct page_entry {
    fileoff_t start;
    pageoff_t size; // size of page
} page_entry;

```

Хранятся они с конца, потому что в случае, если запрашиваемая страница по размеру не подходит (слишком маленькая), то мы проходимся по всем страницам, которые лежат в стеке страниц container_page, и ищем подходящую. Можно было сделать проще: если последняя запись не подходит, то мы аллоцируем из файла (как malloc) новую страницу подходящую, однако что сделано, то сделано. Могу поменять, если под runtime не подходит.

2.1) Уровень IO, думаю пояснять не надо: просто операции с файлом.

Пример dbms/io/p_container.h:

```

PAGE_DEFAULT_SELECT(struct page_container, container)
PAGE_DEFAULT_CREATE(struct page_container, container)
PAGE_DEFAULT_ALTER(struct page_container, container)
PAGE_DEFAULT_DROP(struct page_container, container)

PAGE_DEFAULT_CONSTRUCT_SELECT(struct page_container, container)
PAGE_DEFAULT_ALTER_DESTRUCT(struct page_container, container)

```

Интересует что за defines — смотрите dbms/io/p_base.h, dbms/io/p_base.c.

3.1) Уровень dbms представляет из себя уровень с различными алгоритмами, использующими функции уровней ниже. Из интересного можно показать итераторы, а именно итераторы по страницам необходимой таблицы (dbms/iter.{c,h}). Итераторы могут не только читать, но и обновлять записи и удалять. Соответственно в результате удаления страница могла стать вместо полной неполной, и ее необходимо будет добавить в стек неполных страниц таблицы.

```

262 bool tp_iter_next(struct tp_iter *iter) {
263     if (!tp_tuple_iter_next(iter->tuple_iter)) {
264         tp_tuple_iter_destruct(&iter->tuple_iter);
265
266         while (tp_page_iter_next(iter->page_iter)) {
267             table_page *page = tp_page_iter_get(iter->page_iter);
268             iter->tuple_iter = tp_tuple_iter_construct(page, iter->tuple_size);
269
270             if (tp_tuple_iter_get(iter->tuple_iter) != NULL) {
271                 return true;
272             }
273         }
274         iter->tuple_iter = tp_tuple_iter_construct(NULL, iter->tuple_size);
275         return false;
276     }
277     return true;
278 }
279
280 // @return original tuple in case we want to update
281 struct tp_tuple *tp_iter_get(struct tp_iter *iter) {
282     return tp_tuple_iter_get(iter->tuple_iter);
283 }
284
285 void tp_iter_update(struct tp_iter *iter, tp_tuple *tpt_new, tpt_col_info *info) {
286     tp_tuple *tpt_old = tp_iter_get(iter);
287     tpt_erase(tpt_old, tpt_new, iter->page_iter->dpt, info, iter->page_iter->dbms);
288     tp_update_row_ptr(iter->page_iter->cur, tpt_new, iter->tuple_size, tpt_old);
289 }
290
291 void tp_iter_update_columns(struct tp_iter *iter, tp_tuple *tpt_new, tpt_col_info *info,
292                             size_t arr_size, size_t *idxs) {
293     tp_tuple *tpt_old = tp_iter_get(iter);
294     tpt_erase_columns(tpt_old, tpt_new, iter->page_iter->dpt, info, arr_size, idxs,
295                      iter->page_iter->dbms);
296     tp_update_row_ptr(iter->page_iter->cur, tpt_new, iter->tuple_size, tpt_old);
297 }
298
299 void tp_iter_remove(struct tp_iter *iter, tpt_col_info *info) {
300     tp_tuple *tpt_old = tp_iter_get(iter);
301     tpt_erase(tpt_old, NULL, iter->page_iter->dpt, info, iter->page_iter->dbms);
302     tp_remove_row_ptr(iter->page_iter->cur, tpt_old);
303 }
304

```

tp_erase_columns — препроцессинг апдейта и удаления: удаляем колонки строки, если они неинлайнены, чтобы утечки памяти не было. Также итератор по страницам:

```
146 static bool tp_page_iter_next(tp_page_iter *iter) {
147     if (iter->do_write) {
148         tp_page_td_post(iter);
149         tp_alter(iter->cur, iter->dbms->dbfile->file, iter->cur_loc);
150     }
151
152     fileoff_t next_loc = iter->cur->header.next;
153
154     tp_destruct(&iter->cur);
155
156     if (!fileoff_is_null(next_loc)) {
157         tp_page_iter_next_set(iter, dbms_tp_open(iter->dbms, next_loc), next_loc);
158         return true;
159     }
160     return false;
161 }
```

Если итератор открыт не только для чтения, но и для записи, то вызывается функция tp_page_td_post и осуществляется tp_alter (запись обновленной страницы в файл). Реализация — если страница была полной, а теперь она неполная, то добавляем ее в стек неполных страниц таблицы для переиспользования.

```
static void tp_page_td_post(tp_page_iter *iter) {
    if (iter->was_full && !tp_is_full(iter->cur)) {
        page_entry entry = {.size = iter->cur->header.base.size, .start = iter->cur_loc};

        fileoff_t td_last = iter->dpt->header.td_last;
        dbms_td_push_single(iter->dbms, &td_last, &entry);
        iter->dpt->header.td_last = td_last;
    }
}
```

Там есть еще много чего интересного, например pagerool аллокатор, который работает как (malloc, free), реализация data_dist — взаимодействие с данными, которые могут неинлайниться, table_dist — куда добавляются неполные страницы таблицы, и соответственно, функции для создания страниц в dbms/page.{h,c}.

4) Уровень пользовательских функций: только эти функции может дергать пользователь.

Пример реализации вставки, удаления страниц на уровне, использующий уровень ниже:


```

bool table_exists(struct dbms *dbms, const char *table_name) {
    fileoff_t fileoff;
    pageoff_t pageoff;
    return dbms_find_table(table_name, dbms, &fileoff, &pageoff);
}

bool table_create(struct dbms *dbms, struct dto_table *table) {
    if (table_exists(dbms, table->name)) {
        return false;
    } else {
        dbms_create_table(table, dbms);
        return true;
    }
}

bool table_drop(struct dbms *dbms, struct dto_table *table) {
    fileoff_t fileoff;
    pageoff_t pageoff;
    if (dbms_find_table(table->name, dbms, &fileoff, &pageoff)) {
        return dbms_drop_table(fileoff, pageoff, dbms);
    }
    return false;
}

```

4.1) Планы запросов (include/plan.h, src/plan.c):

Существую различные ноды планов:

```

enum plan_type {
    PLAN_TYPE_SOURCE,

    PLAN_TYPE_DELETED,

    PLAN_TYPE_SELECT,
    PLAN_TYPE_UPDATE,
    PLAN_TYPE_DELETE,

    PLAN_TYPE_CROSS_JOIN,

    PLAN_TYPE_FILTER,
};

```

Все планы «наследуются» от struct plan, который обладает следующими полями:

```
// plan {{{
struct plan {
    enum plan_type type;
    // info about what get returns
    size_t arr_size;
    struct plan_table_info *pti_arr;
    struct tp_tuple **tuple_arr;

    VIRTUAL const struct plan_table_info *(*get_info)(void *self, size_t *size);
    // get row with structure of plan_row_info arr
    VIRTUAL struct tp_tuple **(*get)(void *self);
    VIRTUAL bool (*next)(void *self);
    // Check if this element is last
    VIRTUAL bool (*end)(void *self);
    VIRTUAL void (*destruct)(void *self);

    // returns true if page if can row be located in db (if not virtual) else false
    VIRTUAL PRIVATE bool (*get_iter)(void *self, struct tp_iter **iter_out);
    VIRTUAL PRIVATE bool (*get_dbms)(void *self, struct dbms **dbms_out);
    VIRTUAL PRIVATE void (*start)(void *self, bool do_write);
};
```

- get — достаем текущую запись.
- next — запрос на следующий элемент (возвращает true если существует)
- end — дочитали ли таблицу до конца
- destruct — деструктор, вызывает все деревья рекурсивно

self — объект структуры, через который осуществляется вызов функции (крч как в lua).

В struct plan_table_info представлена различная информация о таблице:

```
struct plan_table_info {
    const char *table_name;
    struct dp_tuple *dpt; // only valid for current plan node
    size_t tpt_size;
    struct tpt_col_info *col_info;
};
```

tuple_arr и pti_arr являются массивами. В каждом индексе массива записана какая-то таблица. Таблиц может быть одна или несколько. Таблицы могут конвертироваться, например, в результате операции cross_join из 2 таблиц arr_size = 2, в первом индексе лежит первая таблица, во втором лежит вторая. Выполняем операцию select → 2 таблицы преобразуются в одну типую виртуальную таблицу, теперь arr_size = 1.

```

#define PLAN_PARENT
{
    struct plan base;
    struct plan *parent;
}

struct plan_parent {
    struct PLAN_PARENT;
};
// }}}

// plan_select{{{
// returns data like it is received from plan_source (smth like virtual table)
// but don't write data
struct plan_select {
    struct PLAN_PARENT;
    // methods
    INHERIT const struct plan_table_info *(*get_info)(void *self, size_t *size);
    INHERIT struct tp_tuple **(*get)(void *self);
    INHERIT bool (*end)(void *self);

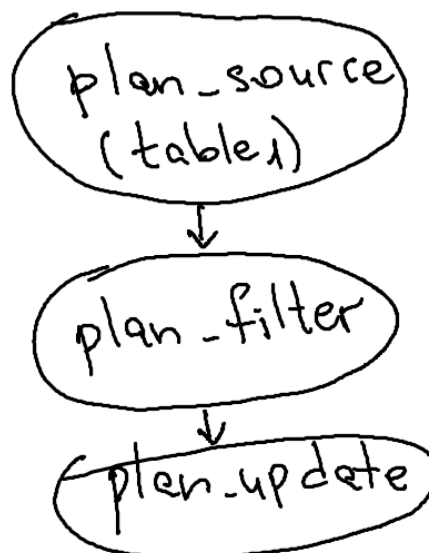
    OVERRIDE bool (*next)(void *self);
    OVERRIDE void (*destruct)(void *self);

    void (*start)(void *self); // only present in terminal plan_nodes
};

```

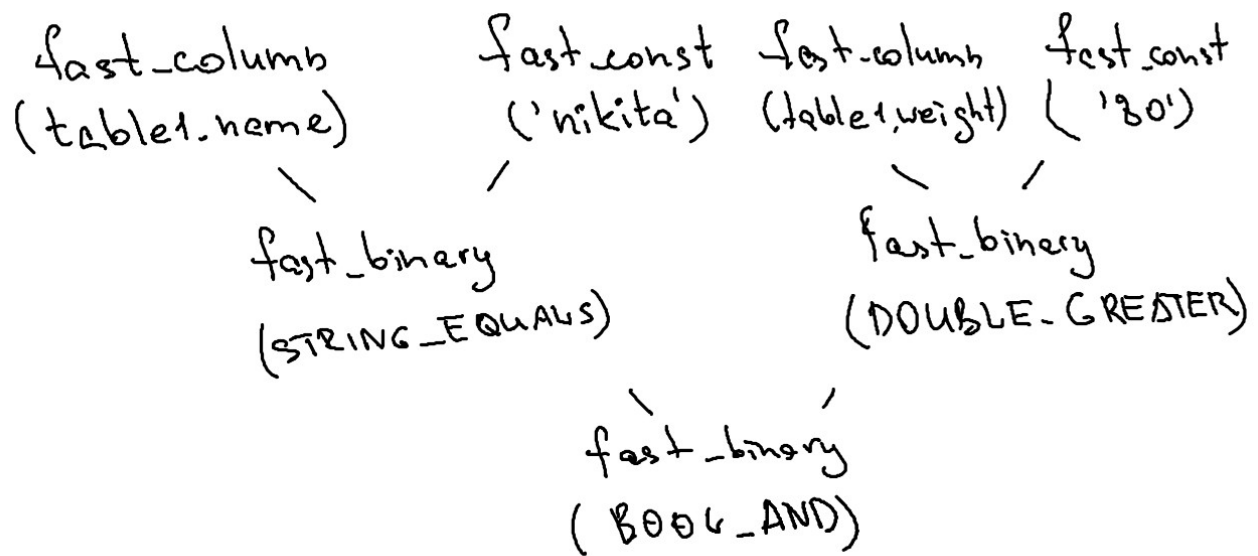
снаружи struct plan base лежат публичные методы, которые может быть вызваны. INHERIT, OVERRIDE — дефайны, чтобы самому не запутаться.

План запроса update from table1 set table1.name = «perestaralsiya» where table1.name = «nikita» and table1.weight > 80:



Фильтру мы передаем ограничения, в нашем примере это (table1.name = «nikita» and table1.weight > 80). Ограничения также строятся в виде дерева (подробнее в src/plan_filter.

{c,h}). Пользователь не может вызывать функции у фильтра и просматривать то, что внутри объекта. Дерево фильтра выглядит так:



Сначала у корня дерева фильтра мы вызываем метод `pass`, который устанавливает значения в ноды `fast_column`, а затем метод `calc`, который возвращает `tpt_column` типа `bool` с результатом `is_null` и `entry` (`true`, `false`). Да, здесь реализована поддержка `null`.

Можно реализовать любые унарные и бинарные операторы, которые вообще можно физически реализовать. Нет имплементации типа `column.value in (select ...)`, но спокойно можно добавить. Пример имплементации оператора и добавление в публичный хедер:

```

static void bool_not(struct fast_unop *self, void *arg_void) {
    struct TPT_COL_TYPE(COLUMN_TYPE_BOOL) * arg, *res;
    arg = arg_void;
    res = self->base.res;

    res->is_null = arg->is_null;
    res->entry = !arg->entry;
}

struct fast_unop_func BOOL_NOT = {.func = bool_not, .ret_type = COLUMN_TYPE_BOOL};

```

```

5 #define EXT_UNOP(name) extern struct fast_unop_func name
6 #define EXT_BINOP(name) extern struct fast_binop_func name
7
8 struct fast_unop;
9 struct fast_binop;
10
11 // fast_unop {{{
12 struct fast_unop_func {
13     void (*func)(struct fast_unop *self, void *arg);
14     enum table_column_type ret_type;
15 };
16
17 EXT_UNOP(BOOL_NOT);

```

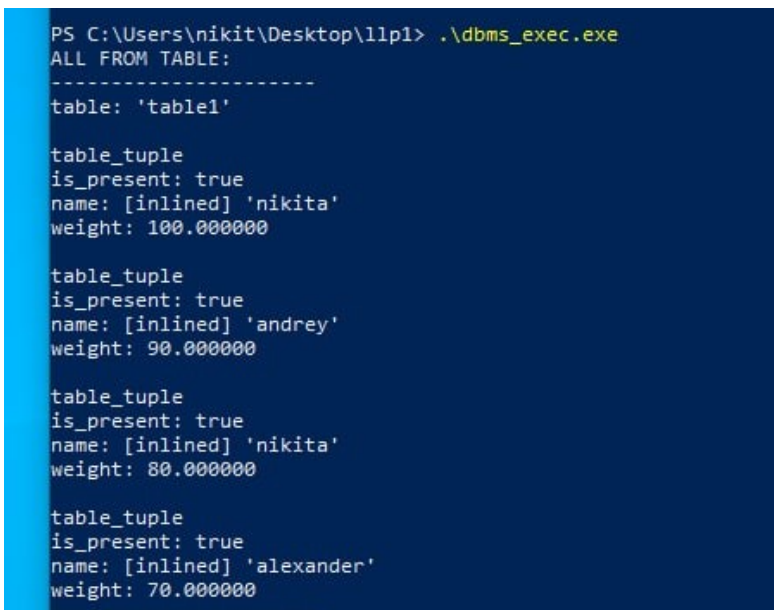
Результаты:

- 1) Были спроектированы структуры данных.
- 2) Было спроектировано представление схемы данных и описаны базовые операции работы над ними.
- 3) Был реализован публичный интерфейс с операциями добавления, обновления, удаления, выборки, фильтрации данных.
- 4) Были реализованы тестовые сценарии для демонстрации работоспособности и производительности программы.

Пример того, что программа работает на *nix, *win:

Сборка программы через кастомный скрипт для windows (crossdev):

```
./cmake.sh released gen reset -DCMAKE_C_COMPILER=i686-w64-mingw32-gcc -  
DCMAKE_CXX_COMPILER=i686-w64-mingw32-g++
```



```
PS C:\Users\nikit\Desktop\11p1> .\dbms_exec.exe  
ALL FROM TABLE:  
-----  
table: 'table1'  
  
table_tuple  
is_present: true  
name: [inlined] 'nikita'  
weight: 100.000000  
  
table_tuple  
is_present: true  
name: [inlined] 'andrey'  
weight: 90.000000  
  
table_tuple  
is_present: true  
name: [inlined] 'nikita'  
weight: 80.000000  
  
table_tuple  
is_present: true  
name: [inlined] 'alexander'  
weight: 70.000000  
-----
```

Сборка программы для *nix:

```
./cmake.sh released reset -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
```

```

table_tuple
is_present: true
name: [inlined] 'nikita'
weight: 100.000000

table_tuple
is_present: true
name: [inlined] 'andrey'
weight: 90.000000

table_tuple
is_present: true
name: [inlined] 'nikita'
weight: 80.000000

table_tuple
is_present: true
name: [inlined] 'alexander'
weight: 70.000000
-----

table_tuple
is_present: true
name: [inlined] 'perestarsiya'
weight: 100.000000
-----
After:
-----
table: 'table1'

table_tuple
is_present: true
name: [inlined] 'perestarsiya'
weight: 100.000000

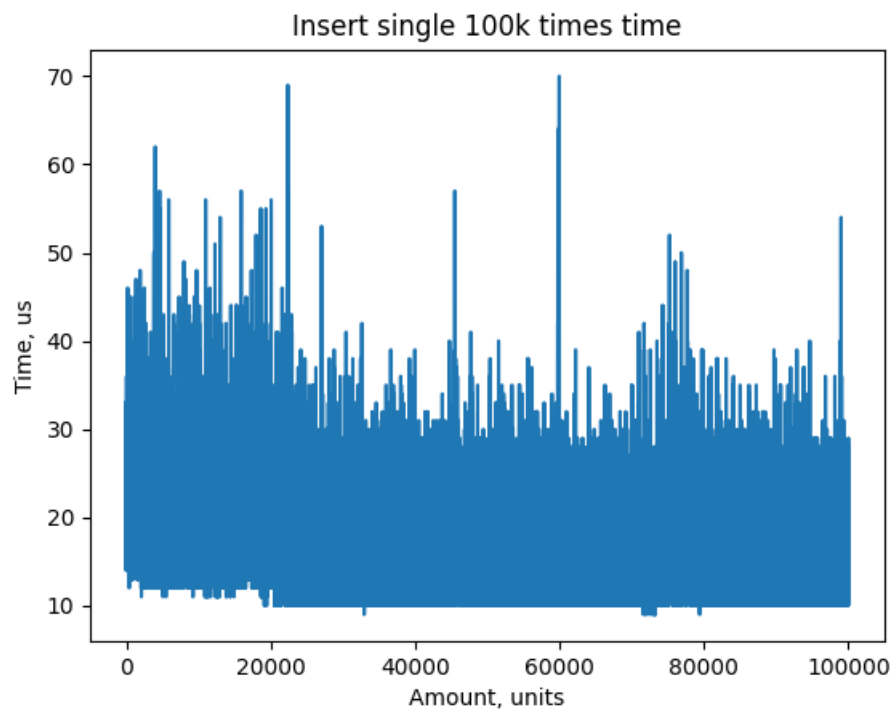
table_tuple
is_present: true
name: [inlined] 'andrey'
weight: 90.000000

table_tuple
is_present: true
name: [inlined] 'nikita'
weight: 80.000000

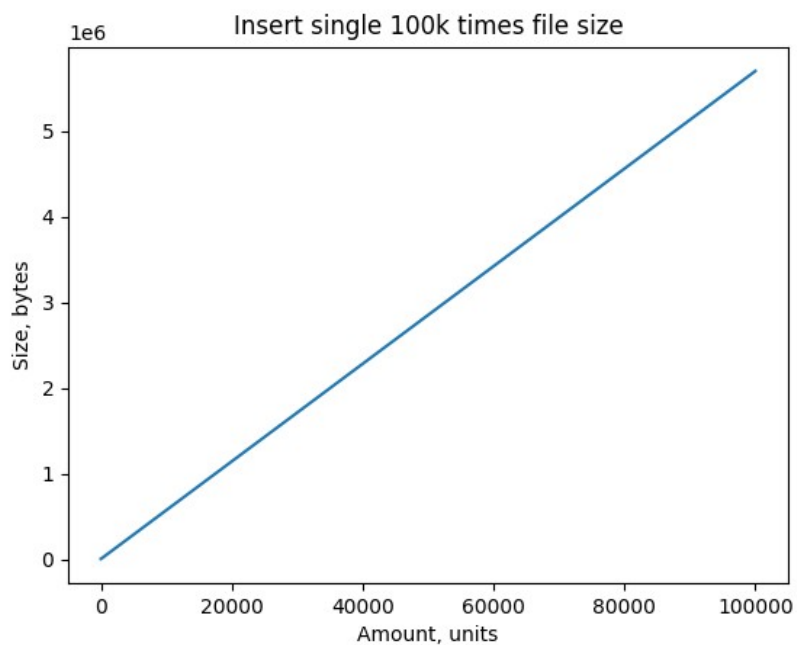
table_tuple
is_present: true
name: [inlined] 'alexander'
weight: 70.000000
-----

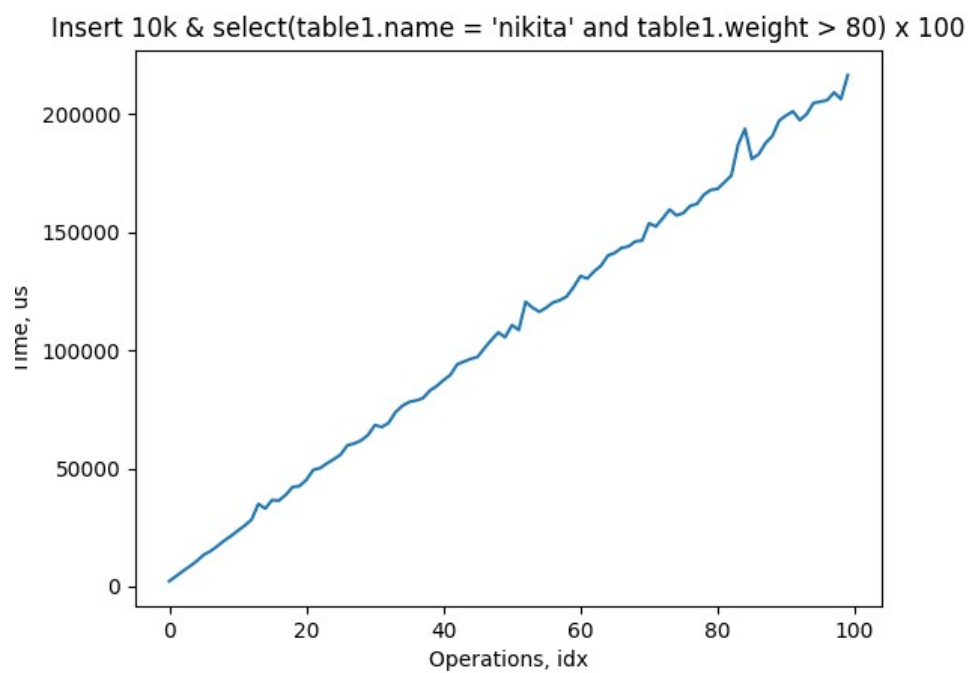
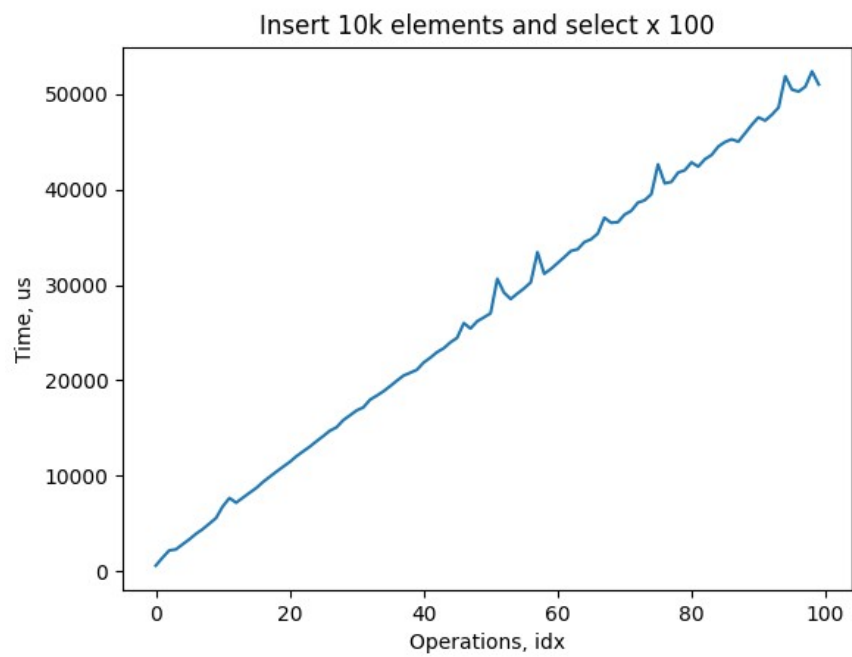
```

5.1) По результатам тестовых сценариев построены графики производительности (особо не парился с оптимизациями константы, также немного дурацкий алгоритм выбора страницы в malloce (в худшем случае переберет все страницы)):

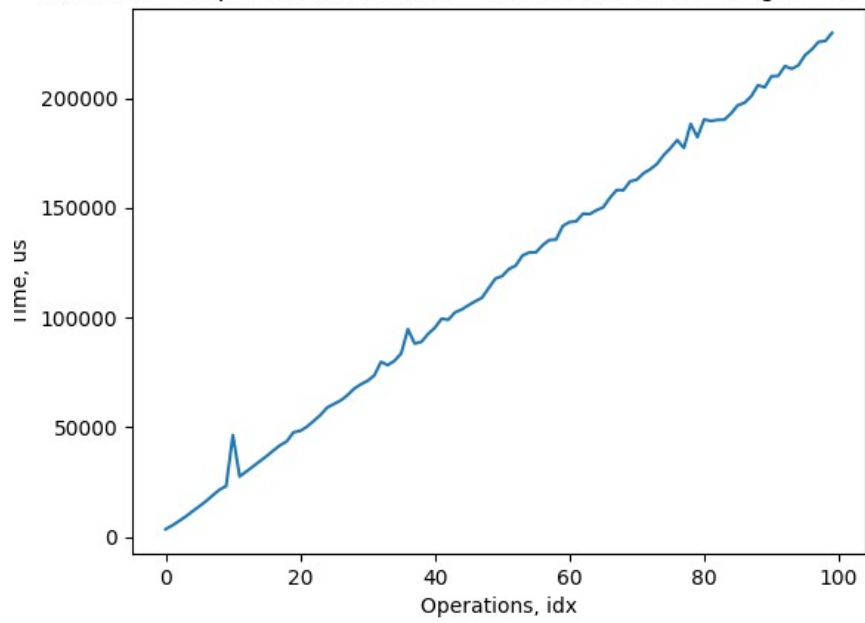


Константа вставки может быть еще меньше, если вставлять элементы не по одному, а несколько сразу, поскольку реализована вставка списка строк.

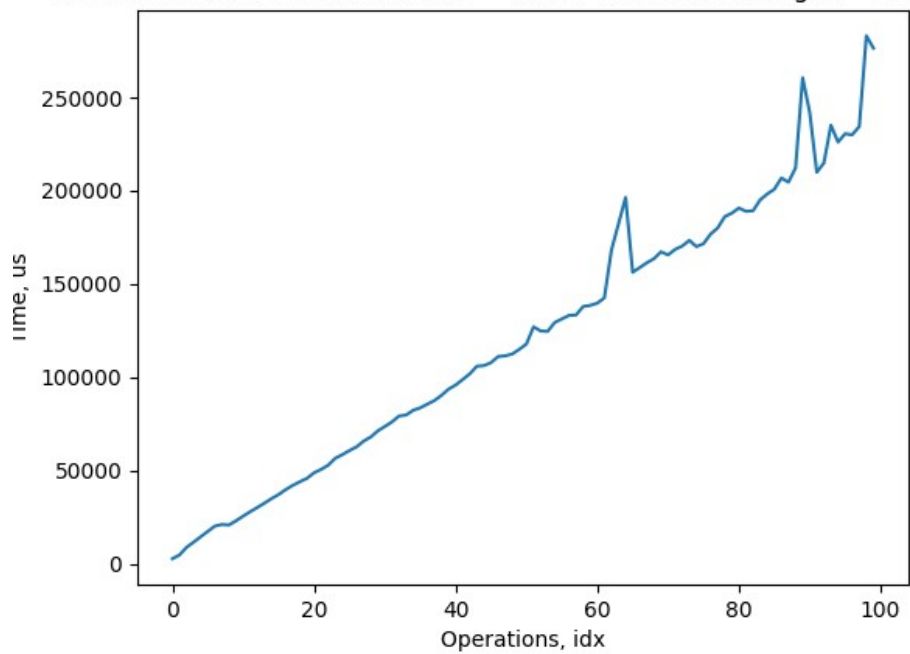


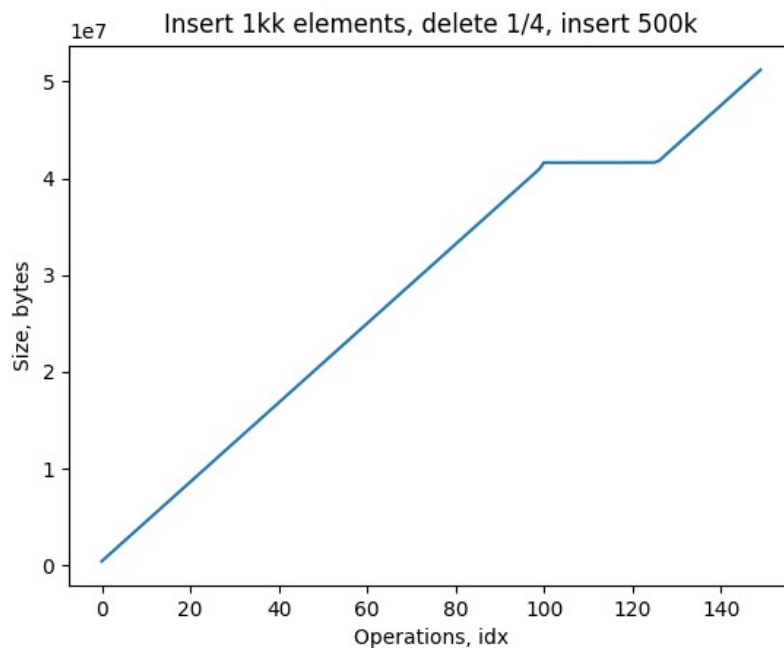


Insert 10k & update(table1.name = 'nikita' and table1.weight > 80) x 100



Insert 10k & delete(table1.name = 'nikita' and table1.weight > 80) x 100





5.2) Асан не ругается, valgrind в одном стеке ругается на write call (скорее всего байты непроинициализированы), с -O3 без -DNDEBUG работает замечательно (я просто в ассерты функционал позабиписывал, а он убирает в итоге код внутри ассертов, и все успешно падает).

Выводы:

Предложенная структура удовлетворяет требованиям производительности. Построены структуры, предоставляющие возможность интеграции с лабораторной работой 2. Сама лабораторная работа была переписана многочисленное число раз из-за того, что я раньше писал на Java, и благодаря своей концепции объекто-ориентированного программирования тяжело написать на этом Япе архитектурно очень плохой код, однако на си развязаны руки: ты можешь делать все что хочешь. Я сначала написал говно, потом опять написал говно, потом вроде бы осознал свои ошибки и постарался разбить программу на несколько слоев (типо как драйверы в linux), чтобы слои выше не могли дергать функции слоев выше и сильно ниже. Вот картинка моей активности при написании ЛР.

https://github.com/zubrailx/itmo_llp-lab1

В скором времени переедет сюда:

<https://github.com/zubrailx/University-ITMO/tree/main/Year-3/Low-level-programming/lab-1>