Университет ИТМО

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.01 Информатика и вычислительная техника

Лабораторная работа №1

по дисциплине «Низкоуровневое программирование»
Вариант №3

Выполнил:
Студент группы P33302
Овчаренко А.А.

Преподаватель:
Кореньков Юрий Дмитриевич

г. Санкт-Петербург
2023

Содержание

Цели

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

Задачи

1. Спроектировать структуры данных для представления информации в оперативной памяти
    a. Для порции данных, состоящий из элементов определённого рода (см форму данных), поддержать тривиальные значения по меньшей мере следующих типов: четырёхбайтовые целые числа и числа с плавающей точкой, текстовые строки произвольной длины, булевские значения
    b. Для информации о запросе
2. Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним:
    a. Операции над схемой данных (создание и удаление элементов схемы)
    b. Базовые операции над элементами данных в соответствии с текущим состоянием схемы (над узлами или записями заданного вида)
        i. Вставка элемента данных
        ii. Перечисление элементов данных
        iii. Обновление элемента данных
        iv. Удаление элемента данных
3. Используя в сигнатурах только структуры данных из п.1, реализовать публичный интерфейс со следующими операциями над файлом данных:
    a. Добавление, удаление и получение информации об элементах схемы данных, размещаемых в файле данных, на уровне, соответствующем виду узлов или записей
    b. Добавление нового элемента данных определённого вида
    c. Выборка набора элементов данных с учётом заданных условий и отношений со смежными элементами данных (по свойствам/полями/атрибутам и логическим связям соответственно)
    d. Обновление элементов данных, соответствующих заданным условиям
    e. Удаление элементов данных, соответствующих заданным условиям
4. Реализовать тестовую программу для демонстрации работоспособности решения
    a. Параметры для всех операций задаются посредством формирования соответствующих структур данных

b. Показать, что при выполнении операций, результат выполнения которых не отражает отношения между элементами данных, потребление оперативной памяти стремится к O(1) независимо от общего объёма фактического затрагиваемых данных
c. Показать, что операция вставки выполняется за O(1) независимо от размера данных, представленных в файле
d. Показать, что операция выборки без учёта отношений (но с опциональными условиями) выполняется за O(n), где n – количество представленных элементов данных выбираемого вида
e. Показать, что операции обновления и удаления элемента данных выполняются не более чем за O(n*m) > t -> O(n+m), где n – количество представленных элементов данных обрабатываемого вида, m – количество фактически затронутых элементов данных
f. Показать, что размер файла данных всегда пропорционален количеству фактически размещённых элементов данных
g. Показать работоспособность решения под управлением ОС семейств Windows и *NIX

5. Результаты тестирования по п.4 представить в составе отчёта, при этом:
   a. В части 3 привести описание структур данных, разработанных в соответствии с п.1
   b. В части 4 описать решение, реализованное в соответствии с пп.2-3
   c. В часть 5 включить графики на основе тестов, демонстрирующие амортизированные показатели ресурсоёмкости по п. 4

Описание работы

Структуры для хранения информации

Есть основные структуры в графовой базе данных: узел, отношение, атрибут. Используются для работы с сущностями графовой базы данных. Поле type нужно для разделения сущностей на типы, что нужно для оптимизации выполнения операций.

```
typedef enum {
  NODE = 0,
  RELATIONSHIP = 1,
  PROPERTY = 2,
} TypeOfElement;

typedef struct __attribute__((packed)) {
    uint32_t id;
    uint32_t name_length;
    char type[NAME_TYPE_LENGTH + 1];
    char *name;
} Node;

typedef struct __attribute__((packed)) {
    uint32_t id;
    uint32_t parent_id;
    uint32_t child_id;
    char type[NAME_TYPE_LENGTH + 1];
    char parent_type[NAME_TYPE_LENGTH + 1];
    char child_type[NAME_TYPE_LENGTH + 1];
} Relationship;

typedef struct __attribute__((packed)) {
    ValueType value_type;
    uint32_t subject_id;
    uint32_t value_length;
    char type[NAME_TYPE_LENGTH + 1];
    char subject_type[NAME_TYPE_LENGTH + 1];
    void *value;
} Property;

typedef enum {
```

```
  INT = 0,
  FLOAT = 1,
  STRING = 2,
  BOOL = 3,
  VOID = 4,
} ValueType;
```

Структуры для работы с файлом

Файл поделен на страницы фиксированного размера. В начале каждой страницы есть заголовок PageHeader, который используется для навигации. Структура Cursor содержит мета информацию, которая не храниться в файле. Она высчитывается во время выполнения программы. Страницы образуют связанный однонаправленный лист. Для каждого типа объекта свой список. Таким образом

```
typedef struct {
    FILE *file;
    uint64_t file_length;
} File;
```

Структура хранит информацию о файле, она используется для осуществления чтения из файла и записи в него.

```
typedef struct __attribute__((packed)) {
    TypeOfElement element_type;
    ValueType value_type;
    uint32_t first_page;
    uint32_t last_page;
    uint32_t next_id;
    char type[NAME_TYPE_LENGTH + 1];
} Entity;

typedef enum {
  NODE = 0,
  RELATIONSHIP = 1,
  PROPERTY = 2,
} TypeOfElement;
```

Тип содержится в мета-информации о сущности.

```
typedef struct __attribute__((packed)) {
    uint32_t block_number;
```

```
    uint32_t next_block;
    uint32_t offset;
} PageHeader;

typedef struct {
    PageHeader *page_header;
    char *page_body;
} Page;

typedef struct {
    File *file;
    uint32_t last_entity_block;
    uint32_t number_of_pages;
    Stack *empty_pages;
} Cursor;
```

Структура Cursor используется для инкапсулирования информации, нужно для работы с файлом. first_entity_block и last_entity_block нужны для поиска нужной информации о сущности.

```
typedef struct {
    bool (*condition)(const void*, const void*);
    uint64_t (*get_size_of_element)(const void*);
    uint64_t (*get_min_size_of_element)(void*);
    void (*write_element_to_file)(Cursor*, Page*, Entity*, const void*, const uint32_t*);
    void *(*read_element_from_file)(const Cursor*, Page*, const uint64_t *);
    void (*memcpy_element)(const void*, char*, uint64_t*);
    void (*free_element)(void*);
    uint32_t (*get_id)(const void*);
} FunctionHelper;
```

Структура FunctionHelper используется для инкапсулирования логики работы с конкретным типом сущности: запись в файл, чтение из файла, копирование из одного места в памяти в другое. Основные метод у меня универсальны. Они не знаю, с каким конкретно тип сущности работают.

```
typedef struct {
    const Cursor *cursor;
    Entity *entity;
    const FunctionHelper *function_helper;
```

```
    uint32_t *read_block_;
    uint64_t *offset_;

    void *element;
    const void *helper;
} Iterator;
```

Структура Iterator используется для прохождения по файлу и последовательной обработки нужных элементов. Это позволяет не хранить все элементы в памяти, когда выполняется операция выборки элементов.

```
typedef struct{
    const Cursor *cursor;
    Entity *entity;
    Page *page;
    uint64_t *offset_;
    Iterator *iterator;
} EntityIterator;
```

Структура EntityIterator используется для прохождения по файлу по всем типам последовательно. Так как разные типы образуют разные связные списки, то для нахождения всех элементов из всех таблиц, нужно пройтись по всем элементам одного типа и перейти на следующий тип

Основные методы взаимодействия с базой данных

```
bool create_element(
    Cursor *cursor, const void *element,
    TypeOfElement element_type, const char *type,
    const FunctionHelper *function_helper,
    const uint32_t *id
);
```

Cursor - используется для работы с файлом, element_type и type используются для поиска необходимых мета данных. element – элемент, который нужно создать. id – под каким id создать элемент. Если id == NULL, то берется следующий доступный id для соответствующего типа

```
void *find_element(
```

```
    const Cursor *cursor, const Page *page,
    uint64_t size_of_element_malloc,
    const void *find_elem, uint64_t *offset_,
    const FunctionHelper *function_helper
);
```

Page - используется для возврата информации о странице на которой был найден элемент, size_of_element_malloc – отвечает за алокацию места в памяти нужного размера для элемента, offset_ – нужен для возврата отступа от начала страницы, где находится искомый элемент. Элемент ищется при помощи функции comparator внутри function_helper .

```
bool delete_element(
    const Cursor *cursor, const void *element,
    uint64_t size_of_sturcture,
    const void *type, TypeOfElement element_type,
    const FunctionHelper *function_helper
);

bool update_element(
    Cursor *cursor, const void *old_element,
    const void *new_element, uint64_t size_of_sturcture,
    const void *type, TypeOfElement element_type,
    const FunctionHelper *function_helper
);

Iterator *select_element(
    const Cursor *cursor,
    TypeOfElement element_type, const char *type,
    uint64_t size_of_element_malloc,
    const void *helper,
    const FunctionHelper *function_helper
);
```

Вспомогательные методы

```
void remove_bid_element(
    const Cursor *cursor, const Page *page,
```

```
    Entity *entity, uint64_t size_of_element,
    const uint64_t *pointer
);

void remove_small_element(
        const Cursor *cursor, const Page *page,
        Entity *entity, uint32_t offset,
        uint64_t size_of_element, const uint64_t *pointer
);
```

Метод remove_bid_element используется для удаления больших элементов. Он отвечает за очистку не нужных блоков.
Когда удаляется элемент, нужно сохранить освободившиеся блоки в памяти, чтобы можно было их использовать повторно.
Для хранения блоков используется структура Stack

```
typedef struct {
    uint64_t size;
    uint64_t top;
    bool is_empty;
    uint64_t *items;
} Stack;
```

Метод remove_small_element используется для удаления маленьких элементов, которые помещаются на одной странице.

Публичный интерфейс для взаимодействия с базой данных

```
bool create_node(Cursor *cursor, Node *node);

bool create_relationship(Cursor *cursor, Relationship *relationship);

bool create_property(Cursor *cursor, Property *property);

bool delete_all_nodes(Cursor *cursor, Node *node);

bool delete_node_by_id(Cursor *cursor, Node *node);

bool delete_all_relationships(Cursor *cursor, Relationship *relationship);

bool delete_relationship_by_id(Cursor *cursor, Relationship *relationship);
```

```
bool delete_all_properties(Cursor *cursor, Property *property);

bool delete_property_by_subject(Cursor *cursor, Property *property);

bool update_all_nodes(Cursor *cursor, Node *old_node, Node *new_node);

bool update_all_relationships(Cursor *cursor, Relationship *old_relationship, Relationship *new_relationship);

bool update_all_properties(Cursor *cursor, Property *old_property, Property *new_property);

Iterator *select_node_by_id(Cursor *cursor, Node *node);

Iterator *select_relationship_by_id(Cursor *cursor, Relationship *relationship);

Iterator *select_property_by_subject(Cursor *cursor, Property *property);

EntityIterator *select_properties_by_node(Cursor *cursor, Node *node);

EntityIterator *select_relationships_by_node(Cursor *cursor, Node *node);

EntityIterator *get_entity_iter_by_node(Cursor *cursor, Node *node, TypeOfElement element_type);

EntityIterator *select_all_nodes(Cursor *cursor);

EntityIterator *select_all_relationships(Cursor *cursor);

EntityIterator *select_all_properties(Cursor *cursor);
```

При удалении элемента типа Node происходит, удаляются элементы типа Relationship Property, которые связаны с элементом Node.
При добавлении элемента типа Relationship или Property есть проверка на то, что существует элемент типа Node, с которым связан добавляемый элемент.

Описание основных операций интерфейс для взаимодействия с базой данных

Работа с мета таблицей

Для сущностей каждого типа и вида (тип – Node, Relationship, Property, вид – задается пользователем, отвечает за группировку сущностей одного вида) есть соответсвующая мета таблица. Все мета таблицы хранятся также в страницах, которые связаны в

список. Соответственно, для нахождения нужно мета таблицы мы итерируемся по списку, проверяем каждую мета таблицу на соответствие условий выбора.
Мета таблица хранит номер первого и последнего блока, следующий id. Номер первого блока нужен для прохода по всем страницам и поиска нужного элемента. Номер последнего блока нужен для записи нового элемента.

Работа с элементами
Для выполнения любой операции необходимо прежде всего найти соответствующую мета таблицу. После нахождения мета таблицы выполняется операция.

Операция вставки происходит следующим образом:
если размер элемента меньше, чем размер страницы, и на странице достаточно место, то происходит запись в текущую страницу. После этого обновляется отступ, который указывает на следующую свободную ячейку.
если размер элемента больше, чем размер страницы, то создается новая страница. На нее записывается часть элемента, следующая часть записывается в следующую страницы.

Операция удаления происходит в два этапа:

1. Поиск необходимого элемента. При этом определяется страница, на которой записан элемент, и ее параметры.
2. Элемент удаляется. Если образуются пустые страницы, то они сохраняются для повторного использования.

Операция обновления состоит из двух операций: удаление элемента и создание нового.

Выборка элементов происходит при помощи итератора. Итератор указывает на конец предыдущего считанного элемента. При вызове метода has_next() происходит поиск необходимого элемента и сохранение его в памяти. Таким образом потребление оперативной памяти имеет асимптотику O(1), так как в любой момент времени храним только один элемент и вспомогательные данные, которые не меняются.

Амортизированные показатели ресурсоемкости

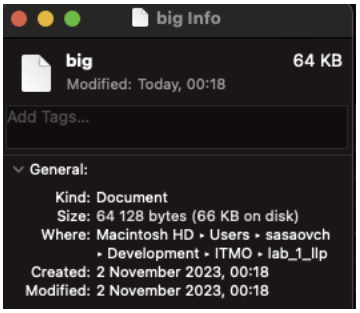
- Вставка элементов: по вертикали – время выполнения для n – элементов, по горизонтали – количество элементов. Видно, что график имеет асимптотику O(n)

- Обновление элементов. График схож с параболой. Асимптотика обновления O(n^2), где n – количество затрагиваемых элементов

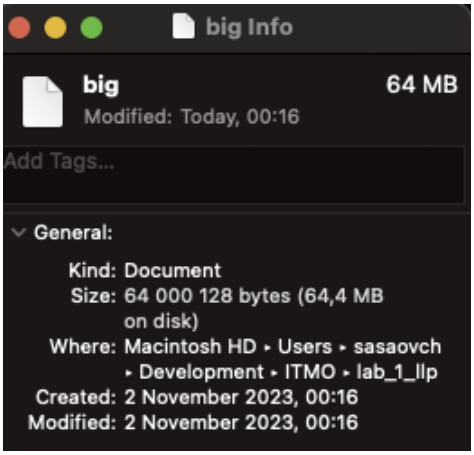
- Удаление элементов. График получается не ровным, не понятно почему.

- Вставка 200 элементов, удаление 100. По вертикали время для вставки 200 элементов

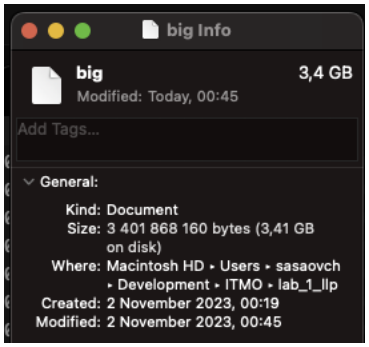
- Зависимость размера файла от количества элементов
  - 1000



  - 1000000

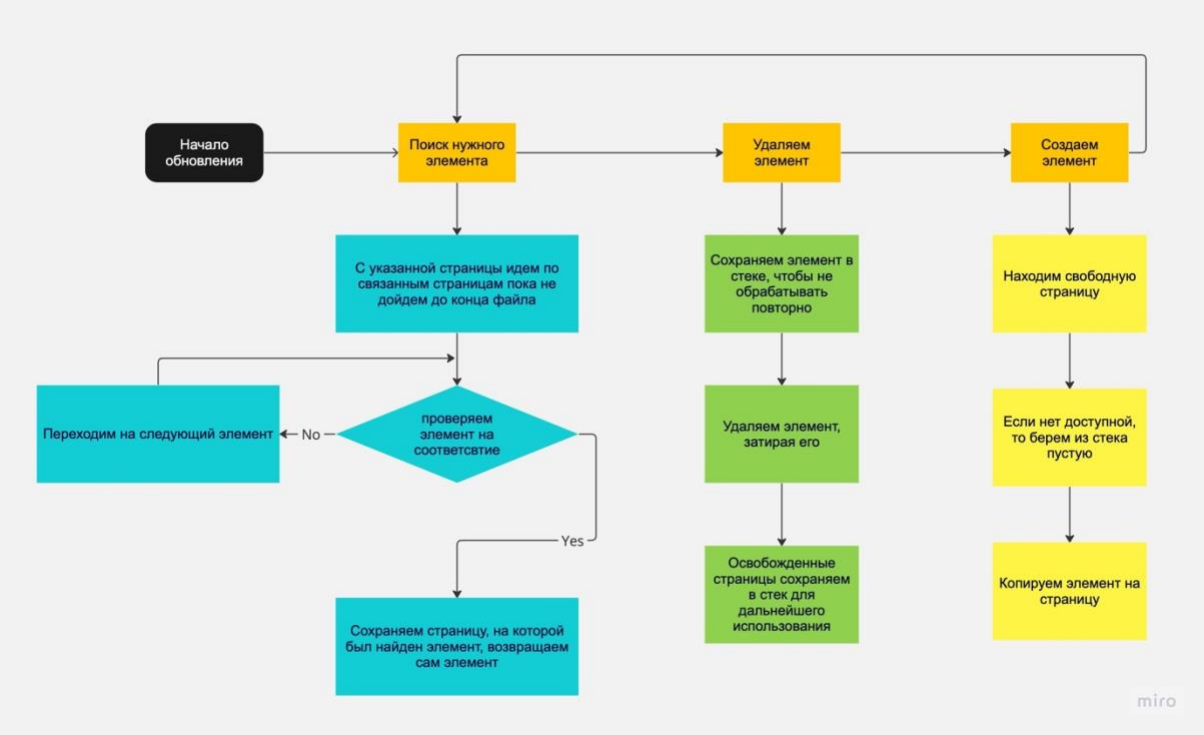


  - > 30000000


big Info
big
3,4 GB
Modified: Today, 00:45
Add Tags...
General:
Kind: Document
Size: 3 401 868 160 bytes (3,41 GB on disk)
Where: Macintosh HD ▸ Users ▸ sasaovch ▸ Development ▸ ITMO ▸ lab_1_llp
Created: 2 November 2023, 00:19
Modified: 2 November 2023, 00:45

Описание процесса update

1. В цикле ищем элемент в файле O(n):
    a. В цикле читаем блок
    b. Проходим по всем элементам блока
    c. Проверяем на условие соответствия, если элемент подходит, то возвращаем
2. Удаляем элемент O(1):
    a. Занульем блоки с этим элементом или затираем его
3. Создаем новый элемент O(1):
    a. Находим свободную страницу
    b. Записываем элемент



Результаты

- В ходе выполнения лабораторной работы были изучены графовые базы данных, рассмотрен принцип их работы, особенности, был произведен сравнительный анализ с реляционными базами данных.
- Была изучена работа neo4j база данных, какие структуры данных используются. Также были изучены postgresql и sqlite.
- Была разработана архитектура базы данных, определены способы взаимодействия с файлом.
- Были разработаны структуры для работы с данными: передача, обработка и хранение
- Были реализованы основные методы работы с данными
- Создан интерфейс для взаимодействия

Выводы

Грамотно разработанная архитектура значительно упрощает работу над разработкой приложения. Благодаря простой и продуманной структуре файла, корректно реализованным базовым методам работы с данными, было просто и быстро реализовать весь необходимый функционал. Удалось избежать дублирования кода во многих местах, переиспользовать методы. На мой взгляд, данная архитектура позволяет без значительных затрат, добавить условия на взаимоотношения различных типов элементов между собой, изменить структуру элементов (добавить/удалить поля), добавить новые типы.