Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет программной инженерии и компьютерной техники

Дисциплина: Low-level Programming

Отчёт по лабораторной работе №1

Вариант 3

Выполнил: Нгуен Нгок Дык

Студент группы: P33302

Преподаватель: Кореньков Юрий Дмитриевич

Санкт-Петербург

2023 г

1. **Objective:**

Create a module that implements **file storage** of data in the form of a graph of nodes with attributes with a total volume of **10GB** or more.

2. **Problem**

- Design data structures to represent information in RAM
- Design a schema representation for a data file and implement basic operations to work with it:
  - Data scheme operations (creating and delete elements)
  - Basic operations on data elements (insert, select, update, delete)
  - Implement a public interface for the above operations
  - Implement a test program to demonstrate the functionality of the solution
- Implement a public interface for the above operations
- Implement a test program to demonstrate the functionality of the solution

3. **Implementation**

The project consists of the following modules:

1. *graphdb* (contains all data structures and functionalities of the graph database)
2. *test* (test all features in specific scripts)
3. *benchmark* (estimate time of execution, file size, memory consumption, …)

Overview:

1. The database is stored in a file, up to 10GB in size. The database is built on graphs by **nodes** and **edges**.
2. The database object is **Node**. Nodes have different **types**. For example, node of type *movie* is associated with node of type *actor*.
3. **Edges** are used to present the connection between nodes.
4. Each node has a set of **attributes**, each of which contains a record with *name, type, and a pointer to the next attribute.*
5. To create a database, we first need to create a **database schema** that contains pointers to the first and last node types. After database has been created, the function returns a pointer to a data structure representing the created database.
6. When a new node instance is created by calling the **createNode** function, it is written to the database. To do this, a string is created (after **its offset** from the beginning of the file will be returned), an instance of the node with all attributes is written to it. And this line is written to the file.

Working with Database:

```
graph_db * create_new_graph_db_by_scheme(db_scheme * scheme, char * file_name);

// Create DB connect with file storage
db = create_new_graph_db_by_scheme(scheme, "storage.txt");


void close_db(graph_db *db);

// close DB
close_db(db);
```

Database Structures:

```
typedef struct {
   db_scheme * scheme;
   FILE * file_storage;
   char * write_buffer;
   int n_write_buffer;
```

```
    char * read_buffer;
    int n_read_buffer;
    int i_read_buffer;
} graph_db;

typedef struct {
    scheme_node * first_node;
    scheme_node * last_node;
} db_scheme;

typedef struct scheme_node{
    struct attr * first_attr;
    struct attr * last_attr;
    struct scheme_node * next_node;
    node_relation * first_node_relation;
    node_relation * last_node_relation;
    char * type;
    int root_off_set;
    int first_off_set;
    int last_off_set;
    int added;
    int n_buffer;
    char * buffer;
    int prev_offset;
    int this_offset;
} scheme_node;

typedef struct attr {
    char * name_attr;
    unsigned char type_attr;
    struct attr * next;
} attr;

typedef struct node_relation{
    struct scheme_node * node;
    struct node_relation * next_node_relation;
} node_relation;

typedef struct node_set_item{
    scheme_node * node;
    int prev_offset;
    int this_offset;
    struct node_set_item * next;
    struct node_set_item * prev;
} node_set_item;

typedef struct {
    unsigned char operand_type;
```

```
   union {
       struct condition * op_condition;
       char * op_string;
       float op_int_bool_float;
       char * op_attr_name;
   };
} condition_operation;



typedef struct condition{
   unsigned char operation_type;
   condition_operation * operand_1;
   condition_operation * operand_2;
} condition;
```

```
enum attr_type{AT_INT32, AT_FLOAT, AT_STRING, AT_BOOLEAN};
enum operand_type{OPRD_INT_BOOL_FLOAT, OPRD_STRING, OPRD_ATTR_NAME, OPRD_CONDITION};
enum operation_type{OP_EQUAL, OP_NOT_EQUAL, OP_LESS, OP_GREATER, OP_NOT, OP_AND,
OP_OR};
enum record_type{R_EMPTY, R_STRING, R_NODE};
```

Definition functionalities

```
// headers
db_scheme * create_new_scheme();
scheme_node * add_node_to_scheme(db_scheme * scheme, char * type_name);
attr * add_attr_to_node(scheme_node * node, char * name, char type);
node_relation * add_node_relation(scheme_node * node, scheme_node *
next_related_node);
void del_node_relation(scheme_node * node, scheme_node * to_delete_node);
void del_node_from_scheme(db_scheme * scheme, scheme_node * node);
void del_attr_from_node(scheme_node * node, attr * to_delete_attr);

————————————————————————————————————————————————————————————————————————
// example to create database structure
   scheme = create_new_scheme();

   // create CITY node
   movie_node = add_node_to_scheme(scheme, "Movie");
   add_attr_to_node(movie_node, "Title", AT_STRING);
   add_attr_to_node(movie_node, "Year", AT_INT32);
   // create DELETED node
   deleted_node = add_node_to_scheme(scheme, "Deleted");
   add_attr_to_node(deleted_node, "Signature", AT_INT32);
   // ADD RELATION
   add_node_relation(movie_node, deleted_node);
   // DELETE ATTR FROM NODE
```

```
    int i;
    del_attr_from_node(actor_node, search_attr_by_name(actor_node, "toDelete", &i));
    db = create_new_graph_db_by_scheme(scheme, "storage.txt");
```

Insert data functionalities

```
    create_node_for_db(db, movie_node);
    set_value_for_attr_of_node(db, movie_node, "Title", create_string_for_db(db,
Titles[i/2]));
    set_value_for_attr_of_node(db, movie_node, "Year", 2000 + i);
    post_node_to_db(db, movie_node);
```

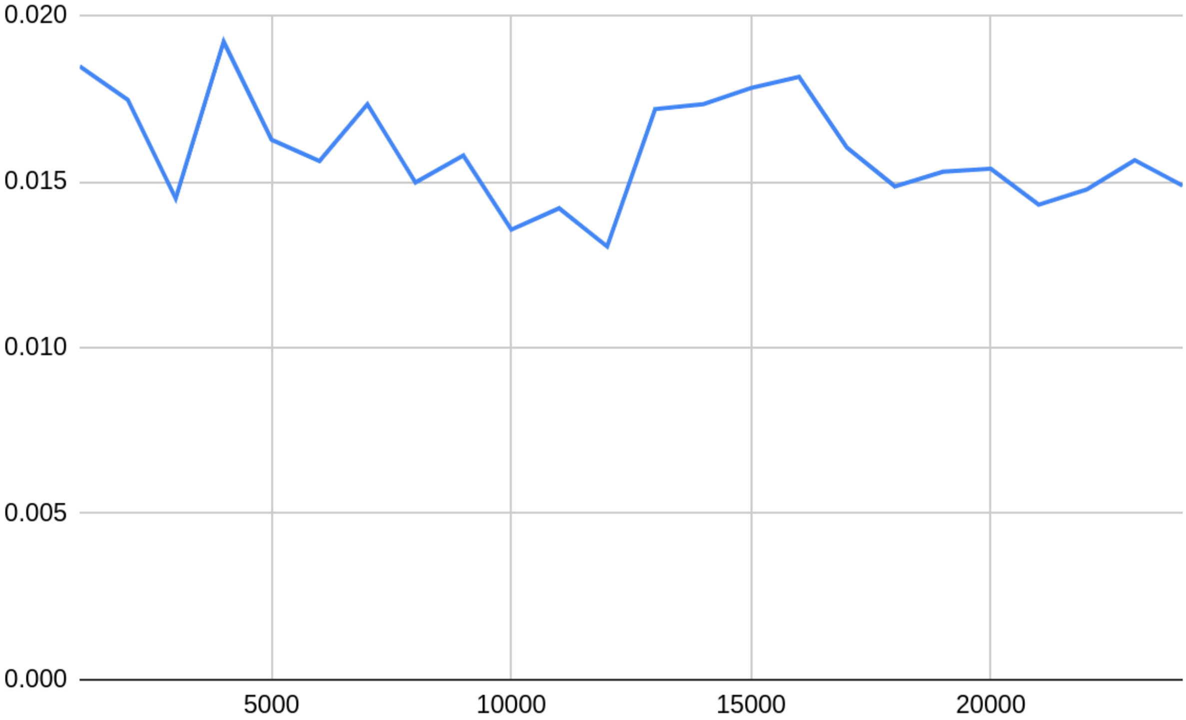
Query functionalities (written in Cypher language**)**

```
    // MATCH (j:Movie) - [:DIRECTED]->(a:Actor) Where (j.Year < 2004) AND (a.Family
!= "Pitt") AND (a.Family != "Hamatova") return a
    ns2 = query_cypher_style(db, 2, movie_node, cond, actor_node, cond2);
    ns12 = ns2;
    i = 0;
    printf("MATCH (j:Movie)-[:DIRECTED]->(a:Actor) WHERE (j.Year < 2004) AND
(a.Family != 'Pitt') AND (a.Family != 'Hamatova') RETURN a;  =>\n");
    while (ns12 != NULL){
        navigate_by_node_set_item(db, ns12);
        if (open_node_to_db(db, actor_node)){
            char * Family = get_string_from_db(db, get_attr_value_of_node(actor_node,
"Family"));
            printf("%s [%i]\n", Family, (int) get_attr_value_of_node(actor_node,
"Year_of_birthday"));
            register_free(strlen(Family)+1);
            free(Family);
            cancel_editing_node(actor_node);
        } else
            printf("Can't open actor node!\n");
        ns12 = ns12->next;
        i++;
    }
    free_node_set(db, ns2);
    printf("%i actors selected!\n", i);
```

4. Result

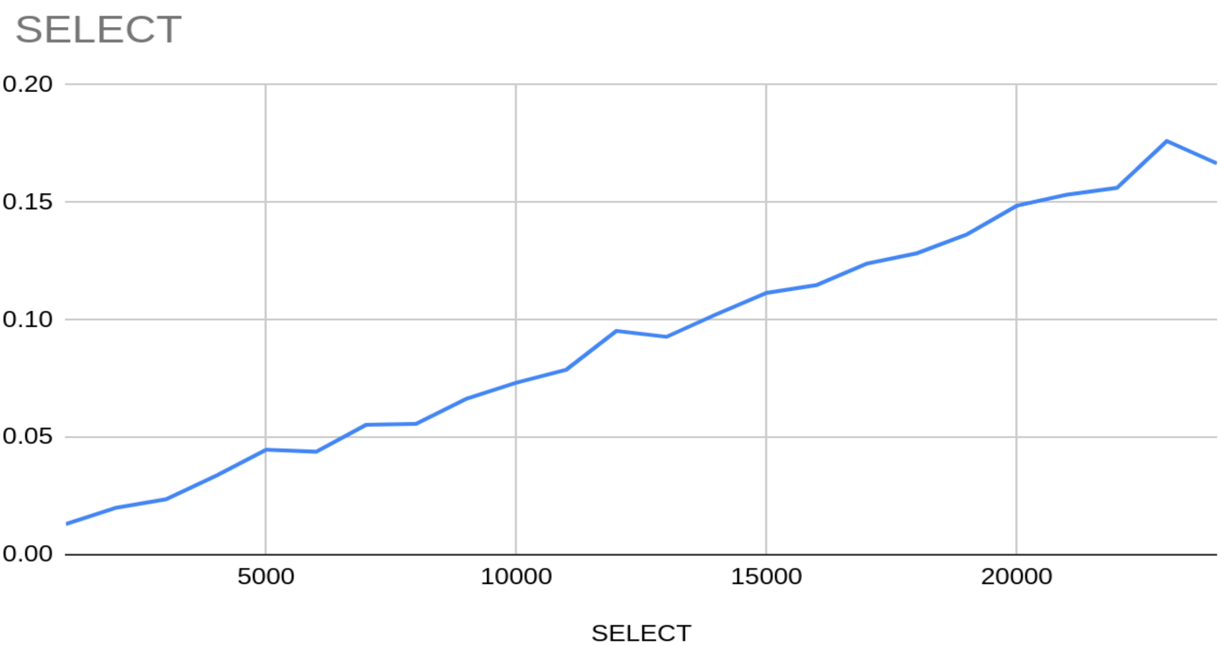
- **Insertion**

  Benchmark insert queries from 0 to 25000 times and the result chart proves that it works in constant time complexity O(1), regardless of the size of the data present in the file.

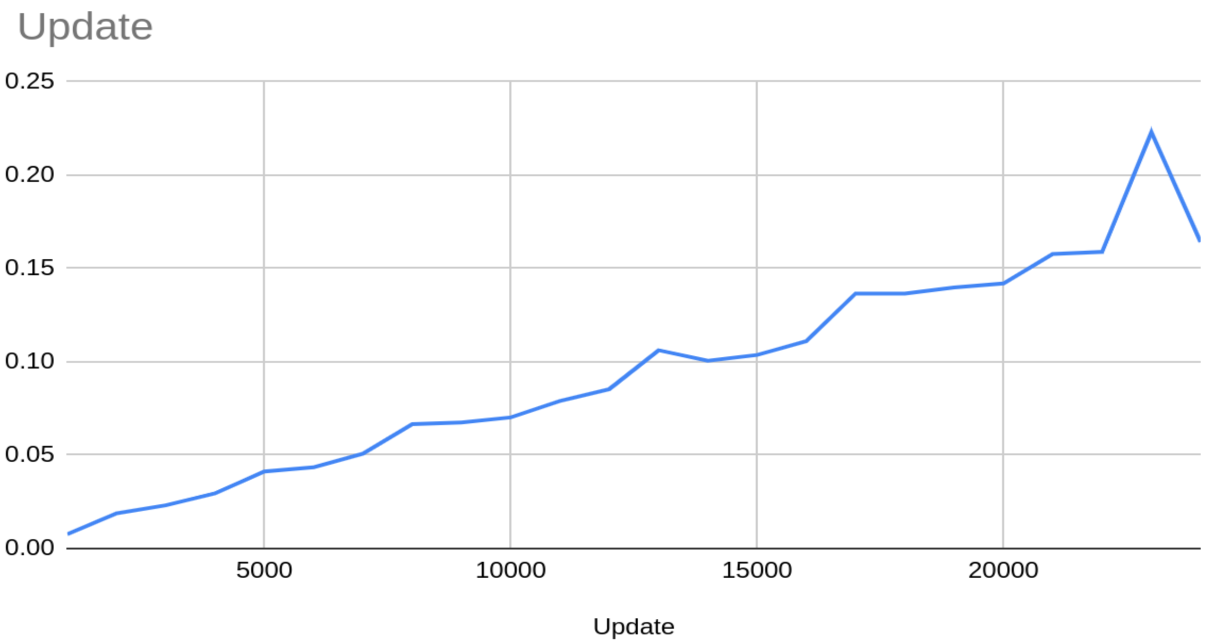


- **Select**

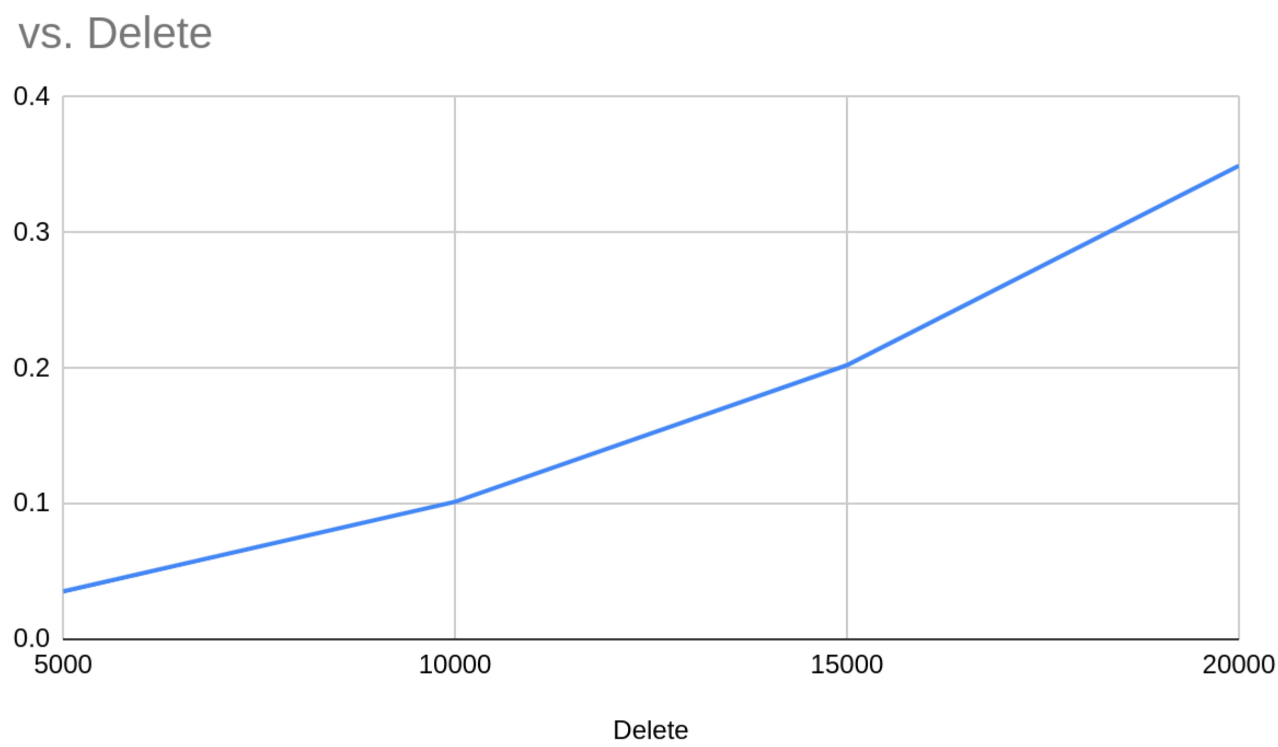
  Benchmark simple select queries (without relation) in data set with size from 0 to 25000 records and the result shows that it works in linear time O(n) - n is number of records



- **Update** - work linear time O(N)



- **Delete** - work in linear time O(N)



5. **Runbook**
    1. git clone git@github.com:ndwannafly/graph_database_file_storage.git
    2. cd graph_database_file_storage
    3. make
    4. ./build/test
    5. ./build/benchmark
6. **Conclusion:**

    During the project implementation, we have designed the data structures and functionalities for the graph database, which supports the basic operations such as: Select, Update, Delete, Create. In general, the graph is similar to a linked list in terms of "fast" insert, "slow" update, delete, select. The benchmark results also show that all time complexities are as expected:

    - [ ] Insert - O(1)
    - [ ] Select O(n)
    - [ ] Update O(n)
    - [ ] Delete O(n)