Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1.5

По дисциплине «Системное программное обеспечение»

Вариант №2

Реляционные таблицы + JSON

Студент:
Михайлова Алена Андреевна

Группа:
P34112

Преподаватель:
Кореньков Юрий Дмитриевич

Санкт-Петербург

2021 г.

1. **Цель задания**

Разработать способ организации данных в файле, позволяющий хранить, выбирать и гранулярно обновлять наборы записей общим объёмом от 10GB соответствующего варианту вида. Реализовать модуль или библиотеку для работы с ним в режиме курсора.

Используя данный способ сериализации, воспользоваться существующей библиотекой для описания схемы и реализации модуля, обеспечивающего функционирование протокола обмена запросами создания, выборки, модификации и удаления данных, и результатами их выполнения.

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора некоторого подмножества языка запросов по выбору в соответствии с вариантом формы данных. Должна быть обеспечена возможность описания команд создания, выборки, модификации и удаления данных.

Используя созданные модули разработать в виде консольного приложения две программы: клиентскую и серверную части. Серверная часть –получающая по сети запросы и операции описанного формата и выполняющая их над файлом, организованным в соответствии с разработанным способом. Имя фала данных для работы получать с аргументами командной строки, создавать новый в случае его отсутствия. Клиентская часть –получающая от пользователя команду, пересылающая её на сервер, получающая ответ и выводящая его в человекопонятном виде.

2. **Описание работы**

Консольный вариант программы:

- Для запуска серверной части исполняемый файл нужно запустить с двумя аргументам: server <filename>, где параметр filename отвечает за название файла табличного хранилища
- Для запуска клиентской части исполняемый файл нужно запустить с аргументом client

Для клиента программа выглядит следующим образом:

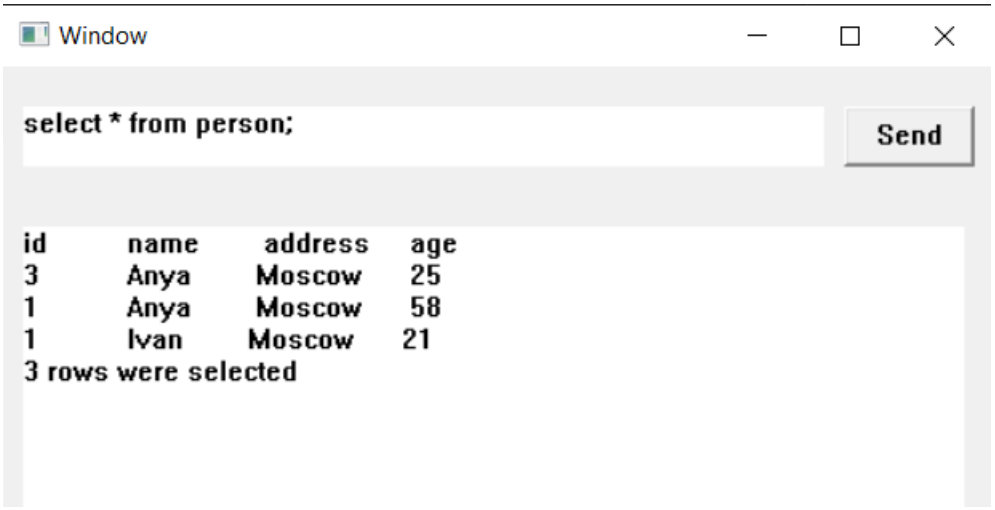
```
Socket successfully created..
connected to the server..
> select * from person;
Syntax Correct
id          name        address     age
3           Anya        Moscow      25
1           Anya        Moscow      58
1           Ivan        Moscow      21
3 rows were selected
> select * from person
The syntax of command is invalid
> select * from fffldllf;
Syntax Correct
operation failed: table with the specified name does not exist
>
```

Пользователю предлагается ввести запрос к табличному хранилищу, а затем ему выводится ответ от сервера.

Вариант программы с графическим интерфейсом:

- Для запуска серверной части исполняемый файл нужно запустить с двумя аргументам: server <filename>, где параметр filename отвечает за название файла табличного хранилища
- Для запуска клиентской части отдельный клиентский исполняемый файл (лежит в ветке gui) нужно запустить без аргументов

Внешний вид интерфейса:



Интерфейс является довольно простым и по своей сути повторяет псевдоинтерфейс программы в консольном режиме. В первом инпуте пользователю предлагается ввести запрос, затем нажать кнопку Send, после чего получить ответ в инпуте в нижней части экрана.

Модули программы:

- Модуль, отвечающий за операции с хранилищем (storage.c)
- Модули, отвечающие за работу с JSON (json_deserialization_module.c и json_serialization_module.c)
- Модуль анализатора SQL (sql_grammar.yacc и sql_lexer.lex)
- Модули для реализации взаимодействия клиента и сервера (client-linux.c и server-linux.c)

Список поддерживаемых операций:

Операции для управления таблицами

1) CREATE TABLE *table_name* (
   column1 datatype,
   column2 datatype,
   column3 datatype,
   ....
   );

2) DROP TABLE *table_name*;

Операции для управления записями в таблицах

1) INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)
   VALUES (*value1*, *value2*, *value3*, ...);

2) INSERT INTO *table_name*
   VALUES (*value1*, *value2*, *value3*, ...);

3) SELECT *column1*, *column2*, ...
   FROM *table_name*;

4) SELECT * FROM *table_name*;

5) SELECT *column1*, *column2*, ...
   FROM *table_name*
   WHERE *condition*;

6) UPDATE *table_name*
   SET *column1* = *value1*, *column2* = *value2*, ...
   WHERE *condition*;

7) DELETE FROM *table_name* WHERE *condition*;

Возможные типы данных колонок при создании таблицы:

1) INT/int
2) UINT/uint
3) NUMBER/number (число с плавающей точкой)
4) TEXT (строковое значение)

Поддерживаемые типы условий (для WHERE):

1) >
2) <
3) =

Примеры входных и выходных данных:

1) Некорректно введенный запрос:

```
> оводлдв
The syntax of command is invalid
>
```

2) Операция создания таблицы:

```
> create table person (id int, name text, city text, age uint, weight number);
Syntax Correct
the table was successfully created
```

3) Операция выборки из таблицы:

```
> select * from person;
Syntax Correct
```

id	name	city	age	weight
10	Petya	Moscow	8	39.9
9	Kate	StPetersburg	18	77.1
8	Loki	Moscow	534	90.8
7	Asgard	Sochi	33	56.7
6	Maria	Saransk	37	33.1
5	Rostislav	Tver	40	45.8
4	Bogdan	Moscow	50	60.5
3	Olya	VelikiyNovgorod	29	90.1
2	Katya	StPetersburg	21	60.5
1	Ivan	Moscow	25	70.8

```
10 rows were selected
```

4) Операция выборки из несуществующей таблицы:

```
>  select * from person123;
Syntax Correct
operation failed: table with the specified name does not exist
```

5) Операция выборки из таблицы с условием:

```
> select * from person where id > 5;
Syntax Correct
```

id	name	city	age	weight
10	Petya	Moscow	8	39.9
9	Kate	StPetersburg	18	77.1
8	Loki	Moscow	534	90.8
7	Asgard	Sochi	33	56.7
6	Maria	Saransk	37	33.1

```
5 rows were selected
```

6) Операция добавления строки в таблицу:

```
> INSERT INTO person VALUES (8, Loki, Moscow, 534, 90.8);
Syntax Correct
the row was successfully inserted
```

7) Операция добавление строки в таблицу со значением, не подходящим по типу данных:

```
> INSERT INTO person VALUES (8, Asgard, Sochi, jfjfkdl, 56.7);
Syntax Correct
operation failed: value for column with name age (uint) has wrong type str
```

8) Операция удаления строки из таблицы:

```
> DELETE FROM person WHERE city = Moscow;
Syntax Correct
4 rows were deleted
```

9) Операция обновления строки:

```
> UPDATE person SET city = Moscow WHERE name = Maria;
Syntax Correct
1 rows were updated
```

10) Операция обновления строки со значением, не подходящим по типу данных:

```
> UPDATE person SET city = Moscow WHERE name = 7171;
Syntax Correct
operation failed: types str and uint are not comparable
```

11) Удаление таблицы:

```
> drop table person;
Syntax Correct
DROP person
the table was successfully dropped
```

3. Аспекты реализации и результаты

Основная часть реализации табличного хранилища состояла в выборе структур, которые будут отражать необходимые объекты (таблицы, строки, колонки) и записываться в файл. Разберем, например, такую структуру для таблиц:

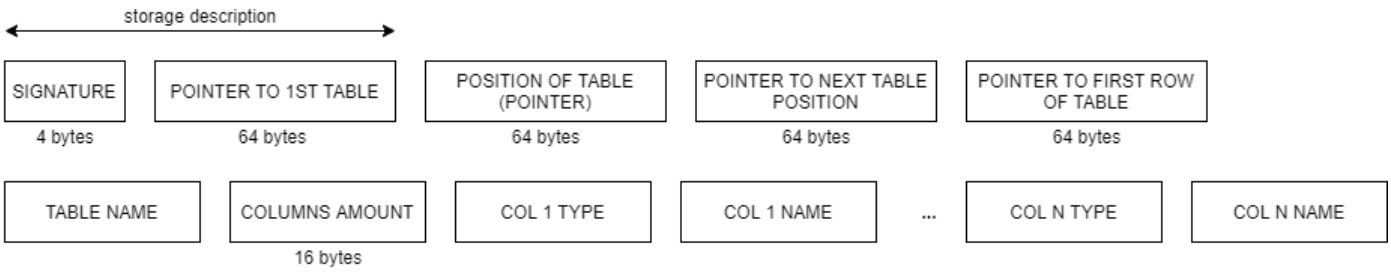
```
struct storage_table {
    struct storage * storage;

    uint64_t position;
    uint64_t next;

    uint64_t first_row;
    char * name;

    struct {
        uint16_t amount;
        struct storage_column * columns;
    } columns;
}
```

То есть в файл таблица будет записана в следующем виде:



При добавлении новой таблицы она записывается в конец файла, указатель на следующую таблицу устанавливается равным указателю на первую текущую таблицу, а указатель на новую таблицу присваивается значению структуры storage «first_table», то есть новая таблица становится первой.

При создании новой таблицы указатель на первую строку выставляется в 0.

Перейдем теперь к реализации парсинга SQL. Для описания лексики был использован win_flex, а для парсинга грамматики и создания выходной структуры был использован win_bison. В результате парсинга команды, которую ввел пользователь, возвращалось дерево, состоящее из узлом структуры sql_node.

```
struct sql_node * new_sql_node (
    int node_type,
    char * text,
    struct sql_node * first,
    struct sql_node * next
);
```

, где
node_type – это номер токена
text – текстовое представление узла
first – ссылка на ребенка,
next – ссылка на сиблинга

В итоге парсинга запроса на создание таблицы можно было получить дерево следующего вида:

```
create table bla (id number, name text, text1 text);
```

```
|_query 258
  |_create 262
    |_table 272
      |_bla 272
      |_columns 286
        |_column_def 285
          |_column_name 279
            |_id 287
          |_column_type 283
            |_number 278
        |_column_def 285
          |_column_name 279
            |_name 287
          |_column_type 283
            |_text 277
        |_column_def 285
          |_column_name 279
            |_text1 287
          |_column_type 283
            |_text 277
```

Далее данное дерево парсится и превращается в JSON объект для передачи клиенту.

Сперва была составлена JSON-схема объектов для данной лабораторной работы следующего вида:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "sql.query.schema.json",
```

```
  "title": "SQL query",
  "description": "A sql query in json",
  "type": "object",
  "properties": {
    "query-type": {
      "description": "The query type",
        "enum": ["CREATE", "DROP", "INSERT", "SELECT", "UPDATE", "DELETE"]
    },
      "table-name": {
      "description": "The name of the table which is going to be manipulated (CREATE,
DROP, SELECT, INSERT, UPFATE, DELETE)",
        "type": "string"
    },
      "selected-columns": {
      "description": "Array of columns which are going to be manipulated (INSERT,
SELECT)",
        "type": "array",
        "items": {
            "type": "string"
        }
    },
      "condition": {
      "description": "The description of condition (after WHERE in SELECT, UPDATE,
DELETE)",
        "type": "object",
        "properties": {
            "condition-type": { "type": "string", "pattern": "+|<|>"},
            "left-param": { "type": "string" },
            "right-param": { "type": "number" }
         }
    },
      "column-defs": {
      "description": "The definition of columns (column name and column type) (CREATE
TABLE)",
        "type": "array",
        "items":{
            "type": "object",
            "properties": {
                  "name":{"type": "string"},
                  "type":{"type": "string", "pattern":
"COLUMN_TYPE_TEXT|COLUMN_TYPE_INT"}
            }
        }
    },
      "values": {
      "description": "Values which are going to be inserted in table (INSERT)",
        "type": "array",
        "items": {
            "type": "string"
        }
    },
      "set-values":{
            "description": "Array of objects that represent pairs of name of the colum
which is going to be updated ans its' new value (UPDATE)",
            "type": "array",
            "items":{
            "type": "object",
            "properties": {
                  "col-name":{"type": "string"},
```

```
                "value":{"type": "string"}
            }
          }
        },
  },
  "required": [ "query-type", "table-name" ],
  "anyOf": [
    {
      "properties": {
        "query-type": { "const": "CREATE" }
      },
      "required": ["column-defs"]
    },
    {
      "properties": {
        "query-type": { "const": "INSERT" }
      },
      "required": ["values"]
    },
      {
      "properties": {
        "query-type": { "const": "SELECT" }
      },
      "required": ["selected-columns"]
    },
      {
      "properties": {
        "query-type": { "const": "UPDATE" }
      },
      "required": ["set-values", "condition"]
    },
      {
      "properties": {
        "query-type": { "const": "DELETE" }
      },
      "required": ["set-values", "condition"]
    }
  ]
}
```

Для работы с JSON была выбрана библиотека wjelement (https://github.com/netmail-open/wjelement). И это, пожалуй, была самая большая ошибка, сделанная мною в ходе выполнения данной лабораторной работы.

Мало того, что документация очень и очень скудная, так еще и в ней представлены функции, которые в самой библиотеки не реализованы.

Более того, очень неудобно сделана работа с массивами – нельзя заранее узнать его длину, при десериализации JSON-а, также нельзя узнать, элемент какого типа лежит в массиве – строка, число или еще что-то. Поэтому пришлось все рассматривать как строку, а потом уже на стороне сервера пытаться узнать, содержит ли строка целое число, дробное или просто строку.

В общем, эта худшая библиотека, с которой я работала за всю жизнь…

Теперь перейдем к реализации клиент – серверного взаимодействия с помощью сокетов. Это была самая приятная часть работы. Пример из интернета заработал сразу же, все, что оставалось – это понять, как можно понять, что все ожидаемое сообщение передано другой стороной, и можно больше не ждать (без закрытия сокета отправляющей стороной). Выбранное решение состояло в том, чтобы сначала высылать количество байт, которое будет содержаться в сообщении, а потом уже само сообщение. Таким образом, клиент будет знать, сколько байт ему нужно считать.

```
void readXBytes(int socket, unsigned int x, void *buffer) {
    int bytesRead = 0;
    int result;
    while (bytesRead < x) {
        result = read(socket, buffer + bytesRead, x - bytesRead);
        if (result < 1) {
        }
        bytesRead += result;
    }
}
```

4. Выводы

Самым большим открытием данной лабораторной работы стало существование такого понятия, как JSON-схема, которая описывает структура JSON-сообщения. Мало того, что она позволяет в голове лучше понимать, как будет выглядеть JSON-сообщение, которое ты пытаешься создать, так еще и с ее помощью можно это сообщение валидировать.

Вторым открытием стало то, что в файле внутри структуры можно хранить ссылку на другую структуру посредствам указателя (от начала файла).

Третьим открытием стало то, что при использовании сокетов можно использовать следующий подход: сначала отправить количество байт, которое будет содержаться в сообщении, а потом уже отправлять сообщение. Это позволит принимающей стороне узнать, сколько байт информации ожидать, прочитать это количество байт и больше не ожидать поступления до заполнения буфера.

И, наконец, четвертое открытие – НИКОГДА не используйте библиотеку wjelement.