Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана» (МГТУ им. Н.Э.Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Поддержка строк переменной длины в генераторе баз данных Mill-DB»

Студент ИУ9-62 _____ Орлов В. А.
(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ Коновалов А.В.
(Подпись, дата) (И.О.Фамилия)

2019 г.

Оглавление

Введение. Постановка задачи.

В современном быстроразвивающемся мире с каждым годом накапливание информации происходит все быстрее и быстрее. Для хранения таковой в большом количестве используются реляционные базы данных. Они используются в самых различных сферах жизни человека: в финансовых, торговых и промышленных организациях. Базы данных в зависимости от области их использования имеют различные виды. В небольших организациях, таких как интернет-магазин, целесообразнее использовать небольшие SQLite базы, а в более крупных, например, Facebook, разумно использование более мощных баз данных подобных Teradata.

В самом общем смысле, база данных - это набор файлов и записей, которые структурированы оптимальным образом для хранения и быстрого доступа к нужной информации среди огромного количества другой. Они позволяют правильно хранить данные, делают доступ к информации максимально удобным, а саму информацию достоверной. Тем самым удовлетворяя все потребности компании.

В рамках курсовой работы на кафедре “Теоретичесой информатики и компьютерных технологии” необходимо было расширение функционала генератора высокопроизводительной базы данных MillDB, создающего библиотеку, обслуживающую базу данных и выполняющую основные функции системы управления баз данных. С помощью MillDB пользователь может использовать техники высокопроизводительной базы данных в своих проектах для решения некоторых проблем выборки и эффективной записи данных. Данная СУБД осуществляет эффективную запись и выборку данных по заранее заданным шаблонам, тем самым минимизируя накладные расходы.

Целью данной курсовой работы является расширение возможностей MillDB, с помощью введение нового типа данных Text. В котором возможно было бы хранить любое неограниченное количество символов.

Этапы работы над курсовым проектом:

1) Изучение архитектуры и исходного кода генератора баз данных.

2) Расширение синтаксиса генератора: лексический анализ, синтаксический анализ, семантический анализ.

3) Изменение формата файла базы данных для поддержки строк переменной длины.

4) Тестирование

1. Обзор генератора баз данных

1.1. Архитектура СУБД MillDB

Система управления базами данных, *сокр.* СУБД (англ. *Database Management System*, сокр. DBMS) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных[1].

СУБД — комплекс программ, позволяющих создать базу данных (БД) и манипулировать данными (вставлять, обновлять, удалять и выбирать). Система обеспечивает безопасность, надёжность хранения и целостность данных, а также предоставляет средства для администрирования БД[2].

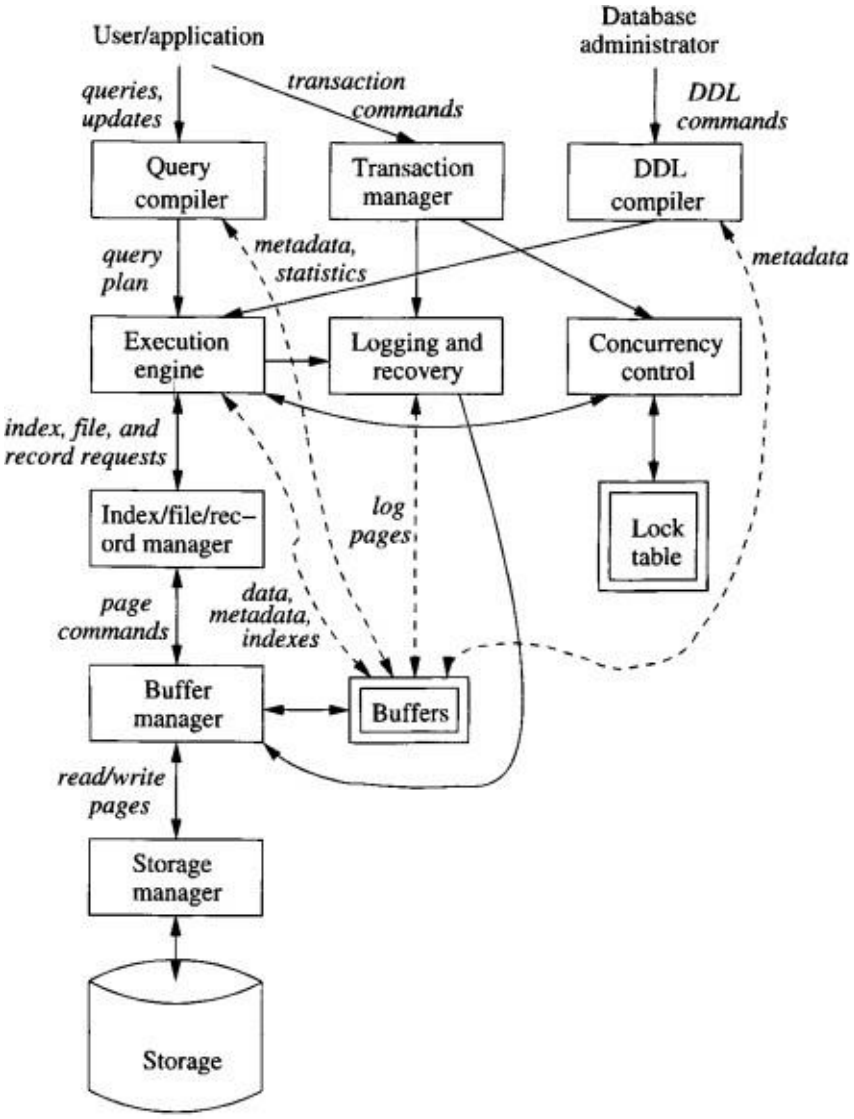

Рисунок 1 Компоненты СУБД [3]

MillDB – генератор СУБД, который используется для построения библиотеки базы данных. Позволяет записывать и считывать данные по заранее известной схеме. Процедура может быть только двух типов: на чтение и на запись. Если в объявлении процедуры присутствует хотя бы один идентификатор “OUT”, то процедура считается “на чтение”, если “IN”, то “на запись”. Но нет процедур для удаления или модификации уже существующих записей. Это сделано для увеличения эффективности работы целевых операции вставки и выбора. В то же время полученная база максимальная удобна в сопровождении.

Взаимодействие с MillDB начинается с построения запроса со спецификацией на собственном диалекте SQL, описывающий набор таблиц и процедур чтения и записи. Далее, происходит обработка спецификации лексером milldb.l, полученным с помощью генератора лексических анализаторов Flex. Совместно с лексером используется генераторо синтаксических анализаторов Bison. Парсер занимается разбором потока токенов методом восходящего анализа перенос-свертка. После окончания работы синтаксического анализатора создается окружение, описывающее структуру базы данных, которая была описана в SQL запросе. Генерируются заголовочный файл *.h и файл с исходным кодом *.c, содержащий реализацию функции из запроса, открытия базы данных для чтения данных, запись и закрытия сгенерированной БД.

Например, если на вход MillDB был подан SQL файл sample.sql, то на выходе будут получены sample.h и sample.c. В этих файлах реализованы все процедуры и таблицы из запроса. Далее, пользователь может использовать эти функции в своих проектах.

Компоненты MillDB:

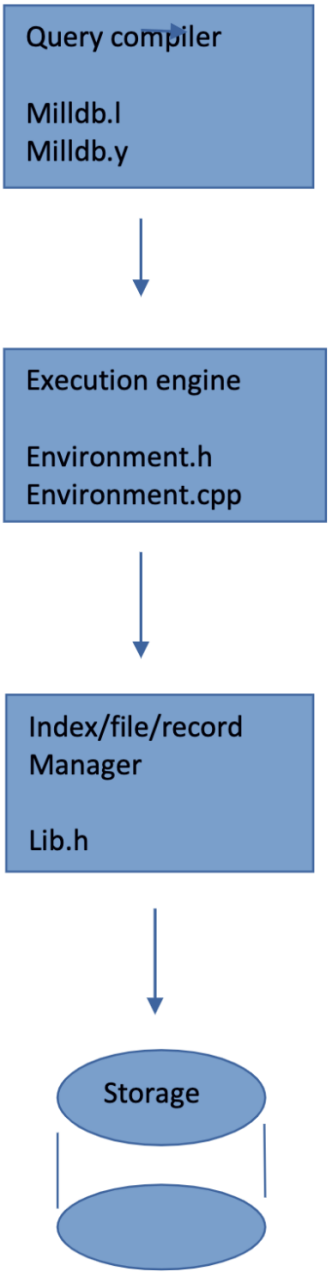

Рисунок 2 Компоненты MillDB

1.2. Построение запроса. Лексический анализ. Грамматика

На вход MillDB подается запрос, описанный на собственном диалекте SQL. Он должен содержать как описание создаваемых в БД таблиц, так и процедуры для выборки и вставки записей.

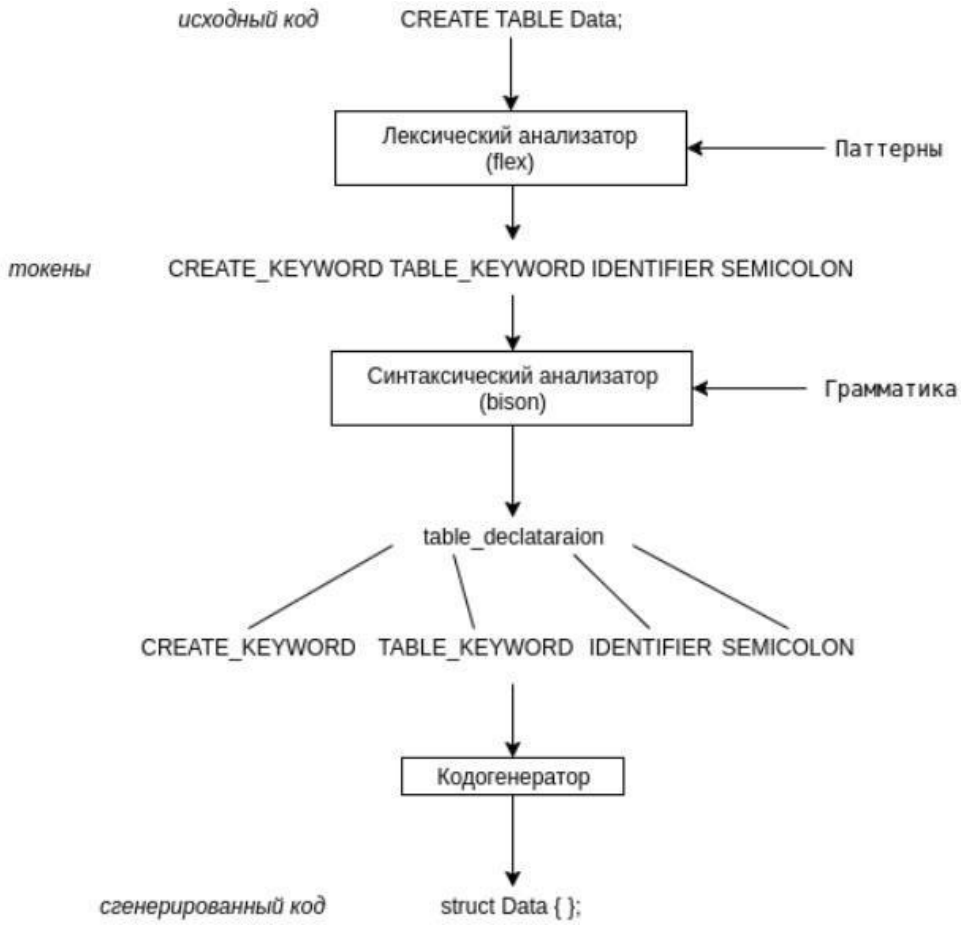

Рисунок 3 Схема работы анализатора спецификации

Описание структуры таблиц состоит из названия таблицы, описания колонок и их типа.

```
CREATE TABLE table-name ({ column-name data-type });
```

Описание процедур записи и поиска:

```
CREATE PROCEDURE procedure-name ({ parameter-name data-type IN | OUT })
BEGIN
{ [ INSERT TABLE table-name VALUES ({argument}); ] |
[ SELECT { column-name SET parameter-name } FROM table-name
WHERE { condition }; ] }
END;
```

Пример входной SQL спецификации:

```
CREATE TABLE Person (id int pk, name char(32));
CREATE PROCEDURE add_person
        (@id int in, @name char(32) in)
BEGIN
        INSERT TABLE Person VALUES (@id, @name);
END;
CREATE PROCEDURE get_person
        (@id int in, @name char(32) out)
BEGIN
        SELECT name SET @name FROM Person WHERE id = @id;
END;
```

Рисунок 4 Пример спецификации базы данных

Описания лексических доменов в виде регулярных выражений:

LPAREN	“(“
RPAREN	“)”
SEMICOLON	“;”
COMMA	“,”
POINT	“.”
AT	“@”
EQ	“=”
TABLE_KEYWORD	“table”
CREATE_KEYWORD	“create”
PK_KEYWORD	“pk”
SELECT_KEYWORD	“select”
FROM_KEYWORD	“from”
WHERE_KEYWORD	“where”
INSERT_KEYWORD	“insert”

VALUES_KEYWORD	“values”
PROCEDURE_KEYWORD	“procedure”
BEGIN_KEYWORD	“begin”
END_KEYWORD	“end”
IN_KEYWORD	“in”
OUT_KEYWORD	“out”
SET_KEYWORD	“set”
ON_KEYWORD	“on”
AND_KEYWORD	“and”
INT_KEYWORD	“int”
FLOAT_KEYWORD	“float”
DOUBLE_KEYWORD	“double”
CHAR_KEYWORD	“char”
TEXT_KEYWORD	“text”
IDENTIFIER	{IDENTIFIER_START}{IDENTIFIER_PART}*
IDENTIFIER_START	[[:alpha:]]
IDENTIFIER_PART	[[:alnum:]_]
PARAMETER	{AT}{IDENTIFIER}
INTEGER`	{DIGIT}+
FLOAT	{DIGIT}+{POINT}{DIGIT}+
DIGIT	[[:alnum:]]
COMMENT_START	“--“
NEWLINE	\n
WHITESPACE	[\t\n]+

Далее полученный поток токенов передается на вход синтаксическому анализатору, который в данном случае был сгенерирован при помощи программы bison. Bison преобразовывает предоставленную программистом грамматику языка в синтаксической анализатор.

Первый набор правил грамматики представляет листья в дереве разбора восходящего метода анализа «перенос-свертка»:

```
table_name:              IDENTIFIER ;
column_name :            IDENTIFIER ;
procedure_name:          IDENTIFIER ;
parameter_name:          PARAMETER ;
```

```
data_type :              INT_KEYWORD|FLOAT_KEYWORD|DOUBLE_KEYWORD|
                         CHAR_KEYWORD LPAREN INTEGER RPAREN ;
parameter_mode :         IN_KEYWORD| OUT_KEYWORD ;
```

При свертке по следующему правилу грамматики генерируется код инициализирующий объект класса Argument, который может иметь тип `enum Type { PARAMETER, VALUE, SEQUENCE_CURR,SEQUENCE_NEXT }` в зависимости от того какой объект передается как аргумент в функцию-прототип процедуры запроса.

```
argument_list :          argument| argument_list COMMA argument ;
argument :               parameter_name ;
```

При свертке по следующему правилу грамматики генерируется код инициализирующий объект класса Column. Класс содержит тип и название колонки таблицы, индикатор того является ли она первичным ключом.

```
column_declaration_list :      column_declaration|
                               column_declaration_list
                               COMMA column_declaration ;

column_declaration :           column_name data_type | column_name
                               data_type PK_KEYWORD ;
```

Данное правило грамматики описывает фильтр для процедуры выборки из таблицы. Будет инициализирован массив экземпляров класса Condition. Экземпляр содержит для сравнения на равенство объекты типа Column и Parameter.

```
condition_list :      condition| condition_list AND_KEYWORD condition;

condition :           column_name EQ parameter_name ;
```

При свертке по этому правилу будет создан список экземпляров класса Selection. Класс описывает те колонки(Column), которые в спецификации являются параметрами м модификатором OUT, то есть те колонки данные из которых необходимы процедуре выборки.

```
selection_list :            selection| selection_list COMMA selection ;

selection :                 column_name SET_KEYWORD parameter_name ;
```

Правило, генерирующее объекты Parameter. Логическое предназначение класса описывать тип параметра, модификатор {IN,OUT}. Список параметров — это входные данные для процедуры.

```
parameter_declaration_list :    parameter_declaration |
                                parameter_declaration_list
                                COMMA parameter_declaration ;

parameter_declaration :       parameter_name data_type parameter_mode ;
```

При применении первого из этих двух правил грамматики будет сконструирован объект InsertStatement, содержащий таблицу и набор аргументов.

При применении второго правила грамматики будет создан объект SelectStatement, содержащий таблицу и наборы условий (Condition) и выборочных полей таблицы.

```
insert_statement :     INSERT_KEYWORD TABLE_KEYWORD table_name
                       VALUES_KEYWORD LPAREN argument_list RPAREN
                       SEMICOLON ;

select_statement :     SELECT_KEYWORD     selection_list     FROM_KEYWORD
                       table_name  WHERE_KEYWORD  condition_list  SEMICOLON
                       ;
```

Все предложения из содержимого процедуры будут сгруппированы в набор выражений вставки или выборки.

```
statement_list :        statement| statement_list statement ;

statement :             insert_statement| select_statement ;
```

Формообразующие участки кода выполняются при свертке предложения по этому правилу. А именно, создается объект Table со всеми надлежащими атрибутами класса, во втором правиле объект Procedure.

```
table_declaration :         CREATE_KEYWORD TABLE_KEYWORD
                            table_name
                            LPAREN column_declaration_list RPAREN
                            SEMICOLON ;

procedure_declaration :     CREATE_KEYWORD PROCEDURE_KEYWORD
                            procedure_name LPAREN
                            parameter_declaration_list RPAREN
                            BEGIN_KEYWORD statement_list END_KEYWORD
                            SEMICOLON ;
```

Корнем дерева разбора становится правило program, описывающее текст программы как набор из объявления таблиц и процедур.

```
program :               program_element_list ;

program_element_list :  program_element| program_element_list
                        program_element ;

program_element :       table_declaration| procedure_declaration ;
```

Грамматика:

```
program : program_element_list ;
program_element_list: program_element| program_element_list
     program_element ;
program_element : table_declaration| procedure_declaration ;
table_declaration : CREATE_KEYWORD TABLE_KEYWORD table_name LPAREN
     column_declaration_list RPAREN SEMICOLON ;
procedure_declaration : CREATE_KEYWORD PROCEDURE_KEYWORD
     procedure_name LPAREN parameter_declaration_list RPAREN
     BEGIN_KEYWORD statement_list END_KEYWORD SEMICOLON ;
statement_list : statement| statement_list statement ;
statement : insert_statement| select_statement ;

insert_statement : INSERT_KEYWORD TABLE_KEYWORD table_name
     VALUES_KEYWORD LPAREN argument_list RPAREN SEMICOLON ;

select_statement : SELECT_KEYWORD selection_list FROM_KEYWORD
     table_name WHERE_KEYWORD condition_list SEMICOLON ;

parameter_declaration_list:parameter_declaration|
     parameter_declaration_list COMMA parameter_declaration ;

parameter_declaration : parameter_name data_type parameter_mode ;

selection_list : selection| selection_list COMMA selection ;

selection : column_name SET_KEYWORD parameter_name ;

condition_list : condition|condition_list AND_KEYWORD condition ;

condition : column_name EQ parameter_name ;

column_declaration_list : column_declaration|column_declaration_list
     COMMA column_declaration ;

column_declaration : column_name data_type| column_name data_type
     PK_KEYWORD ;
```

```
argument_list : argument|argument_list COMMA argument ;

argument : parameter_name ;

table_name : IDENTIFIER ;

column_name : IDENTIFIER ;

procedure_name : IDENTIFIER ;

parameter_name : PARAMETER ;

data_type : INT_KEYWORD| FLOAT_KEYWORD| DOUBLE_KEYWORD| CHAR_KEYWORD|
      TEXT_KEYWORD LPAREN INTEGER RPAREN ;

parameter_mode : IN_KEYWORD|OUT_KEYWORD ;
```

1.3 Классы окружения.

В данном разделе представлены классы, формирующие конечные файлы библиотеки и исходного кода для обслуживания желаемой базы данных. Каждый класс имеет функцию генерации в два файла кода, соответствующего функционалу класса и предназначению.

Класс	Функциональность
Environment	Содержит все экземпляры таблиц и процедур запроса Кодогенерирует в результирующую библиотеку подключение заголовков библиотек, шаблоны структур данных таблиц, функции открытия файла базы данных на чтение, запись, закрытие соответственно, организацию представления данных в оперативной памяти и на диске. Сгенерированный код выполняет функционал диспечтера памяти. Является реализацией паттерна Singleton
Table	Состоит из экземпляров, реализующих колонки в таблице, а также индекса. Реализует функции кодогенерации шаблонов записи таблицы на диск, построения индекса, выгрузку данных, выгрузку индекса и представление его в оперативной памяти.
Column	Содержит тип и название колонки, индикатор того является ли она первичным ключом

DataType	Содержит возможные варианты типов данных {INT, FLOAT, DOUBLE, CHAR}, функции распечатки каждого из типов при кодогенерации параметров и других атрибутов таблицы.
Procedure	Процедура бывает двух типов: на чтение и на запись. Это определяется по типу параметров процедуры которые содержат модификатор режима «на вход» или «на выход». В зависимости от типа строятся разные шаблоны кода, которые обслуживают правильную запись или выборку данных.
Parameter	Класс параметра для процедуры Каждый экземпляр содержит поле модификатора enum Mode {IN, OUT}
Statement	Объявляет функции кодогенерации для переопределения в классах-наследниках.
InsertStatement	Отвечает за кодогенерацию функций вставки в таблицу.
SelectStatement	Реализует структуру данных содержащую выбранные из запроса поля, функцию инициализации и итерирования по составленной выборке. Процесс создания выборки и фильтрации будет описан в следующих главах.

Argument	Кодогенерация типа и имени параметра и его значения в качестве аргумента
Selection	Набор полей необходимых для извлечения данных
Condition	Условие для процедуры Select проверки на равенство одной из колонок таблицы по которой производится

1.4 Хранение данных

Для быстрой записи и считывания информации из базы необходимо грамотно структурировать данные. То есть нужна такая структура данных, которая позволила бы эффективно вставлять и находить нужные данных. Обычно для этих целей используется сбалансированные АВЛ деревья, которые позволяют эффективно вставлять, удалять, модифицировать данные. Но такой подход имеет некоторые затраты в виду того, что данные хранятся не в оперативной памяти, а на внешнем устройстве, и если в базе 1 миллион записей, то необходимо будет около 20 обращений к внешнему устройству, что является довольно-таки дорогостоящим действием.

Так как в случае с MillDB пренебрегаются операции удаления и модификации, то можно улучшить реализацию традиционных подходов к хранению данных и формированию индексов.

Чтобы снизить количество обращений к внешнему устройству несколько вершин хранятся в отдельном блоке, но тогда возникает необходимость производить поиск в самих блоках. Тем не менее это менее затратно, нежели постоянно обращаться к внешнему устройству.

Эффективной структурой данных для этой цели является Б-дерево. Главное отличие Б-дерева от бинарного дерево поиска является то что в Б-дереве у родителя могут быть несколько потомков, в то время как у дерево поиска количество потомков ограничено только двумя. Так как никогда не производится модификация элементов, то достаточно сформировать индексное дерево единожды при записи его на диск.

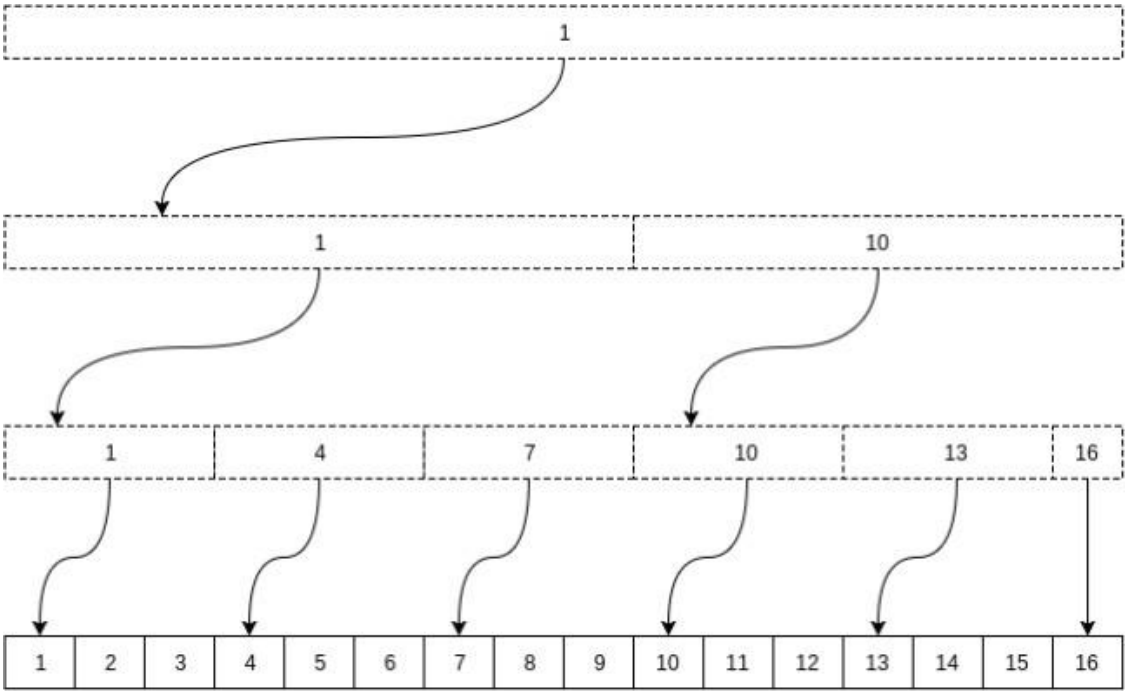

Рисунок 6: Устройство Б-дерева

Данные на диске хранятся следующим образом. В начале файла располагается заголовок фиксированного размера, содержащий информацию о количестве записей в каждой таблице и смещениях. Затем идут блоки из данных и индексов.

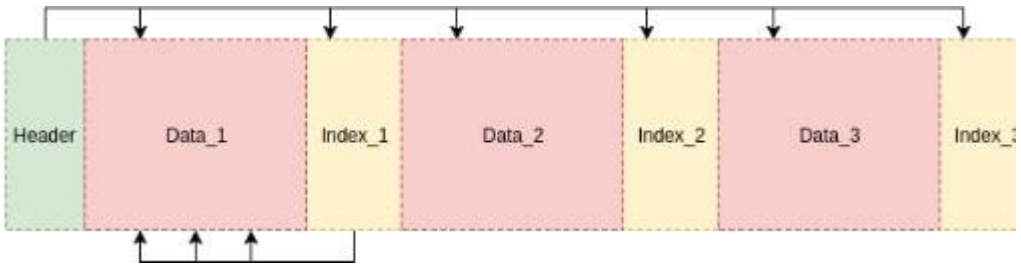

Рисунок 5: Файл базы данных

1.5 Использование сгенерированной библиотеки

```
#include <stdio.h>
#include <time.h>
#include "sample.h"
#include "sample.c"

int main() {
        sample_open_write("FILE");
        add_Person(1,"Adam");
        sample_close_write();
        struct sample_handle* handle1 = sample_open_read("FILE");
        struct get_Person_by_id_out iter1;
        get_Person_by_id_init(&iter1,handle1,1);
        if (get_Person_by_id_next(&iter1))
        printf("%s\n",iter1.data.person_name);
        sample_close_read(handle1);
        return 0;
}
```

Так как сгенерированная библиотека рассчитана на чтение или запись, перед началом работы мы должны вызвать одну из функций, соответствующую нашим целям.

В соответствии с используемой спецификацией SQL, которую мы рассмотрели выше для генерации библиотеки СУБД, в нашем распоряжении две процедуры, одна – на чтение, другая – на запись.

После открытия файла базы данных на чтение или запись, мы применяем нужные нам функции, и в конце обязательно вызываем функцию завершения работы нашей библиотеки.

Использование функции, сгенерированной для вставки записей, является тривиальным и представляет вызов функции с именем определенной в спецификации процедуры с нужными параметрами.

При работе с процедурой выборки мы должны открыть файл БД на чтение. Используя возвращенный обработчик, вызвать функцию инициализации “имя_процедуры”_init, и далее использовать “имя_процедуры”_next для получение каждого следующего члена выборки.

2. Реализация строк переменной длины

Задача этой курсовой работы состоит в добавлении нового типа данных, который называется ‘text’, для создания колонок в БД, хранящих строки неопределенного размера.

После анализа проблемы, были выделены основные этапы модернизации программы:

1. Добавление нового вида токена для лексического анализа
2. Интеграция токена в правила грамматики
3. Анализ возможных архитектурных решений для хранения типа данных неопределенного размера в бинарном файле.
4. Изменение процесса чтения структур из бинарного файла
5. Добавление особого вида записи и чтения записей при существовании в таблице колонки типа ‘text’

Первые два пункта не представляют никаких сложностей, стоит лишь отметить, что в данной реализации невозможно сделать тип ‘text’ первичным ключом, что было спроецировано в данных строчках ‘milldb.y’:

```
column_declaration: column_name data_type {
                        debug("column_declaration 1 BEGIN");

                        $$ = new Column(*$1, $2, false);
                        delete $1;

                        debug("column_declaration 1 END");
                }
        | column_name data_type PK_KEYWORD {
                        debug("column_declaration 2 BEGIN");

                        if ($2->get_typecode() == DataType::TEXT) {
                                throw  logic_error(error_msg("type  'text'  can't  be
primary key"));
                        }

                        $$ = new Column(*$1, $2, true);
                        delete $1;

                        debug("column_declaration 2 END");
        };
```

После, были рассмотрены возможные подходы к хранению строк переменной длины.

Пусть мы имеем таблицу со следующим описанием типов:

```
CREATE TABLE example (
        id int pk,
        description text,
        value int
);
```

Со строкой переменного размера нет никак проблем, пока она находится в оперативной памяти компьютера, но когда такую структуру нужно записать в бинарный файл, требуется совершить некоторые манипуляции.

Представим, что пользователь закончил добавление новых объектов в таблицу *example* и хочет сохранить результат.

Раньше, Mill-db бы просто проитерировался по всем записям, и поля каждой записались бы последовательно в память. Но когда мы хотим положить строку неопределенного размера, у нас возникнут сложности на этапе чтение такой записи. Сначала нам нужно узнать, сколько же занимает записанная строка, и только после этого мы корректно прочитаем набор байт, представляющий из себя бинарный файл.

При записи других полей, имеющих определенный размер, мы должны просто перенести данные в файл. Но если поле имеет тип ‘text’, запишем вместо неё размер данной строки, а после записи всех остальных полей, в конец добавим нашу строку. Таким образом, в шестнадцатеричном виде, запись в таблицу ‘example’ значений (1, ‘hello’, 13) будет выглядеть следующим образом:

01 00 00 00 0F 00 0D 00 00 00 68 65 6c 6c 6f

01 00 00 00 – единица в формате int

0F 00 – длина строки в формате int16

0D 00 00 00 – число 13 в формате int

68 65 6c 6c 6f – наша строка ‘hello’

С помощью именно такого формата записи в бинарном файле происходит хранение колонки таблицы типа ‘text’ в реализации данной курсовой работы.

Чтобы реализовать чтение такой структуры, потребовалось изменить способ чтения записей с постраничного на потоковый.

Было:

```
while (1) {
        fseek(handle->file, offset, SEEK_SET);
        union pet_page page;
        uint64_t  size  =  fread(&page,  sizeof(struct  pet),  pet_CHILDREN,  handle-
>file);  if (size == 0) return;

        for (uint64_t i = 0; i < pet_CHILDREN; i++) {
                . . . обработка объекта
        }
        offset += pet_CHILDREN * sizeof(struct pet);
}
```

Стало:

```
for (int i = 0;; i++) {
        struct pet item;
        uint64_t size = fread(&item, sizeof(struct pet), 1, handle->file);
        if (size == 0) return;

        . . . обработка объекта
}
```

3. Тестирование

При тестировании использовался следующий файл конфигурации SQL:

```
CREATE SEQUENCE Pet_sequence;

CREATE TABLE owner (
        oid int pk,
        oname text,
        pet_id int
);

CREATE TABLE pet (
        pid int pk,
        pname text
);

CREATE PROCEDURE add_owner_pet(@oid int in, @name text in, @pname text in)
BEGIN
        INSERT TABLE owner VALUES (@oid, @name, NEXTVAL(Pet_sequence));
        INSERT TABLE pet VALUES (CURRVAL(Pet_sequence),@pname);
END;

CREATE PROCEDURE get_pet_by_pid(@id int in, @out text out)
BEGIN
    SELECT oname SET @out FROM owner WHERE oid=@id;
    SELECT pname SET @out FROM pet WHERE pid=@id;
END;
```

Программа, использующая сгенерированную на основе конфигурации библиотеку:

```
#include <stdio.h>
#include <time.h>
#include "pet_by_owner.h"

int main() {
    int owners[6] = {5, 2, 1, 4, 3, 6};
    pet_by_owner_open_write("FILE2");
    add_owner_pet(1, "Vova",  "Ni");
    add_owner_pet(2, "Petya", "Chi");
    add_owner_pet(3, "Maria", "Shuia");
    add_owner_pet(4, "Sasha", "Kawai");
    add_owner_pet(5, "Natal", "Kazem");
    add_owner_pet(6, "Tanya", "Charl");
    pet_by_owner_close_write();
    struct pet_by_owner_handle *handle1 = pet_by_owner_open_read("FILE2");
    struct get_pet_by_pid_out iter1;
    int i;
    for (i = 0; i < 6; i++) {
        printf("%d\n", owners[i]);
        get_pet_by_pid_init(&iter1, handle1, owners[i]);

        int flag = 1;
        while (get_pet_by_pid_next(&iter1)) {
            printf("%s\n", iter1.data.out);
            flag = 0;
        }
```

```
        if (flag) {
            printf("\n");
        }
    }
    pet_by_owner_close_read(handle1);

    return 0;
}
```

Вывод программы в результате работы:

```
5
Natal
Kazem
2
Petya
Chi
1
Vova
Ni
4
Sasha
Kawai
3
Maria
Shuia
6
Tanya
Char
```

Заключение

Таким образом, в ходе работы над курсовым проектом были изучены различные структуры данных, методы и подходы к эффективному хранению информации в базах данных, произошло ознакомление с архитектурой систем управления базами данных. Также были применены навыки построения лексического и синтаксического анализаторов, а если точнее, то был разработан новый тип данных text, позволяющий хранить строки переменного размера, что позволило расширить возможности хранения информации в базе данных.

Список использованных источников

1. Кузнецов С. Д. Базы данных. Модели и языки / С. Д. Кузнецов. - М.: Бином-Пресс, 2013. - 720 с.
2. *Кузнецов С. Д.* Основы баз данных. — 2-е изд. — : Интернет- университет информационных технологий; БИНОМ. Лаборатория знаний, 2007. — 484 с. — ISBN 978-5-94774-736-2.
3. Кириллов, В. В. Введение в реляционные базы данных / В. В. Кириллов. - М.: БХВ-Петербург, 2016. 318 с.
4. Мартишин, С. А. Базы данных. Практическое примечание СУБД SQL и NoSQL. Учебное пособие / С. А. Мартишин, В. Л. Симонов, М. В. Храпченко. - М.: Форум, Инфра-М, 2016ю - 368 с