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 КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Реализация связанных таблиц в СУБД MillDB»

Студент ИУ9-62 _____ Атымханова М.
(Подпись, дата) (И.О.Фамилия)

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

2018 г.

Оглавление

Оглавление 2
Введение. Постановка задачи. 3
1. Обзор генератора баз данных 4
1.1. Архитектура СУБД MillDB 4
1.2. Построение запроса. Лексический анализ. Грамматика 7
1.3 Классы окружения. Ядро MillDB 14
1.5 Построение плана выполнения запроса 16
1.6 Выполнение запроса. Использование сгенерированной библиотеки 21
2. Реализация связанных таблиц 22
2.1 Sequence 22
2.1.1 Расширение синтаксиса входного языка 23
2.1.2 Пример использования 25
Текст запроса: 25
2.2 Оператор JOIN 26
2.2.1 Пример использования 27
3. Тестирование 28
Заключение 30
Список использованных источников 31
Приложение 1 32

Введение. Постановка задачи.

Реляционные базы данных (РБД) используются повсюду. Они бывают самых разных видов, от маленьких и полезных SQLite до мощных Teradata. В рамках выпускной квалификационной работы бакалавра на кафедре "Теоретической информатики и компьютерных технологий" был разработан генератор высокопроизводительной базы данных MillDB, берущий на себя производство библиотеки обслуживающей базу данных и выполняющей основные функции системы управления базой данных. MillDB позволяет его пользователю использовать техники высокопроизводительной базы данных в собственных проектах, направленных на решение проблем эффективной записи и выборки данных. MillDB осуществляет эффективную запись и выборку данных по заранее заданным спецификацией шаблонам, минимизируя при этом накладные расходы. Для удовлетворения самых различных нужд такая программа генерируется автоматически из поданного на вход программы спецификации, описывающей параметры данных, их типы, параметры процедур вставки и выборки по условиям.

Целью данной работы является расширение возможностей СУБД MillDB, ознакомление с архитектурой СУБД путем внесения новшеств и изменений. Необходимо реализовать связанные таблицы, что означает добавление оператора JOIN и элемента, связывающего таблицы, по которому можно составить предикат функции поиска по связанным таблицам.

Этапы работы над курсовым проектом:

1) Обзор генератора баз данных

2) Расширение синтаксиса входного языка

3) Разработка шаблонов кода для нового синтаксиса

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].

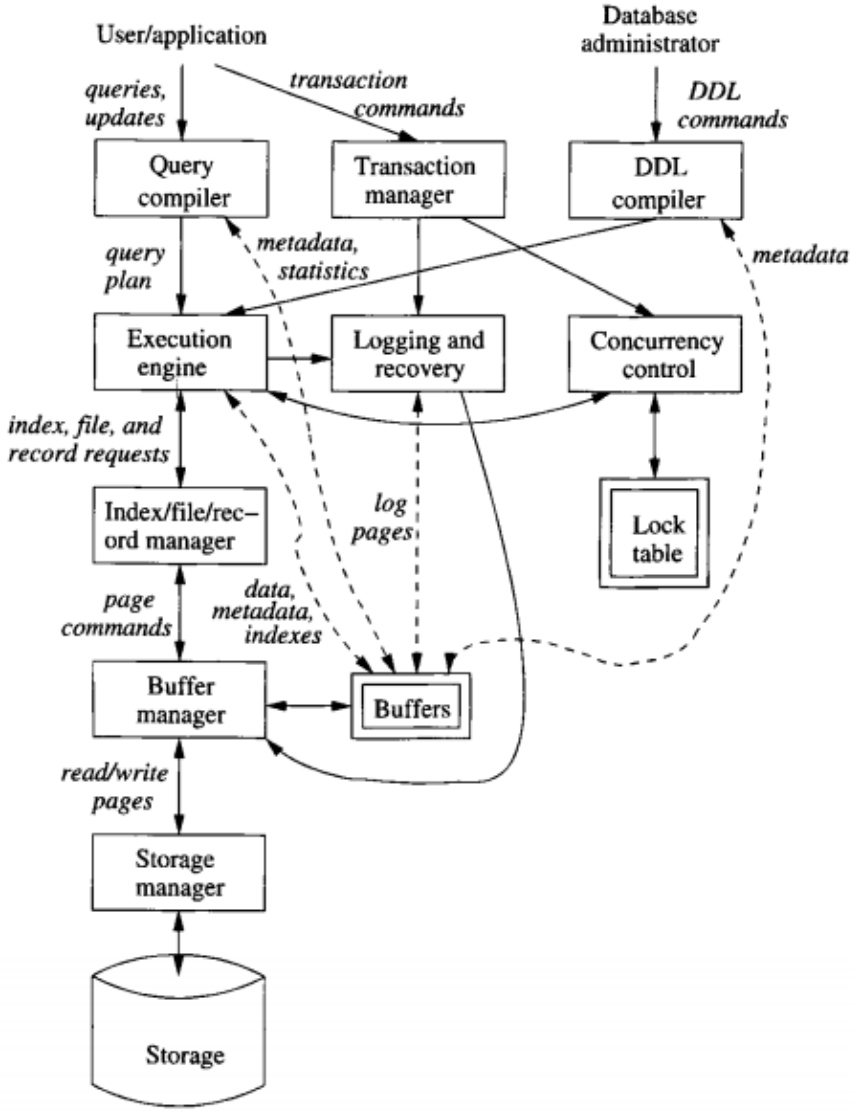


Рисунок 1 Компоненты СУБД [3]

MillDB - генератор баз данных, используемый для построения библиотеки высокопроизводительной и компактной базы данных.

Позволительно записывать и считывать данные по заранее известной схеме данных. При этом, MillDB пренебрегает поддержкой удаления и модификации данных, что увеличивает эффективность работы целевых операций вставки и выборки. В то же время, полученная база стремится быть максимально удобной в сопровождении.

Взаимодействие с СУБД начинаются с построения запроса. MillDB принимает на вход запрос со спецификацией на собственном диалекте SQL, описывающим набор таблиц и процедуры чтения и записи.

Запрос на первой стадии обрабатывается лексером milldb.l, полученным с применением генератора лексических анализаторов Flex.

Лексер используется совместно с генератором синтаксических анализаторов Bison. Парсер осуществляет разбор потока токенов в соответствие с грамматикой SQL подобного диалекта методом восходящего анализа перенос-свертка.

На каждом этапе свертки генерируется код, соответствующий текущему правилу грамматики. Генерируемый код нацелен на построение классов окружения, описывающих архитектуру базы данных, описанной в запросе.

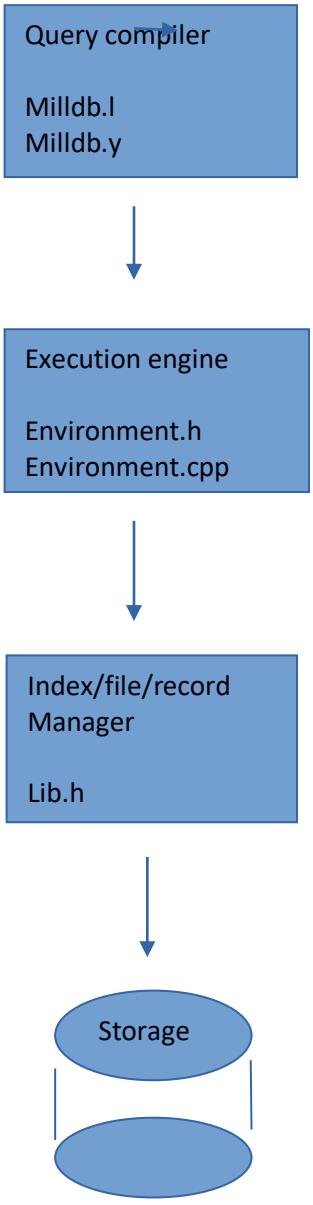
Сформированное окружение вызывает функцию кодогенерации заголовочного файла *.h и файла с исходным кодом *.c, содержащими реализацию таблиц и функций из запроса, открытия базы данных на чтение, запись, закрытие базы данных.

```
sample.sql → sample.h sample.c
```

Рисунок 2 Иллюстрация результата

С использованием сгенерированной библиотеки можно осуществлять процедуры вставки и извлечения данных из таблиц базы данных в соответствие с описанием функций и таблиц в запросе.

Компоненты MillDB:



1.2. Построение запроса. Лексический анализ. Грамматика

На вход менеджеру запросов подается спецификация на собственном диалекте SQL, описывающая набор таблиц и процедуры чтения и записи.

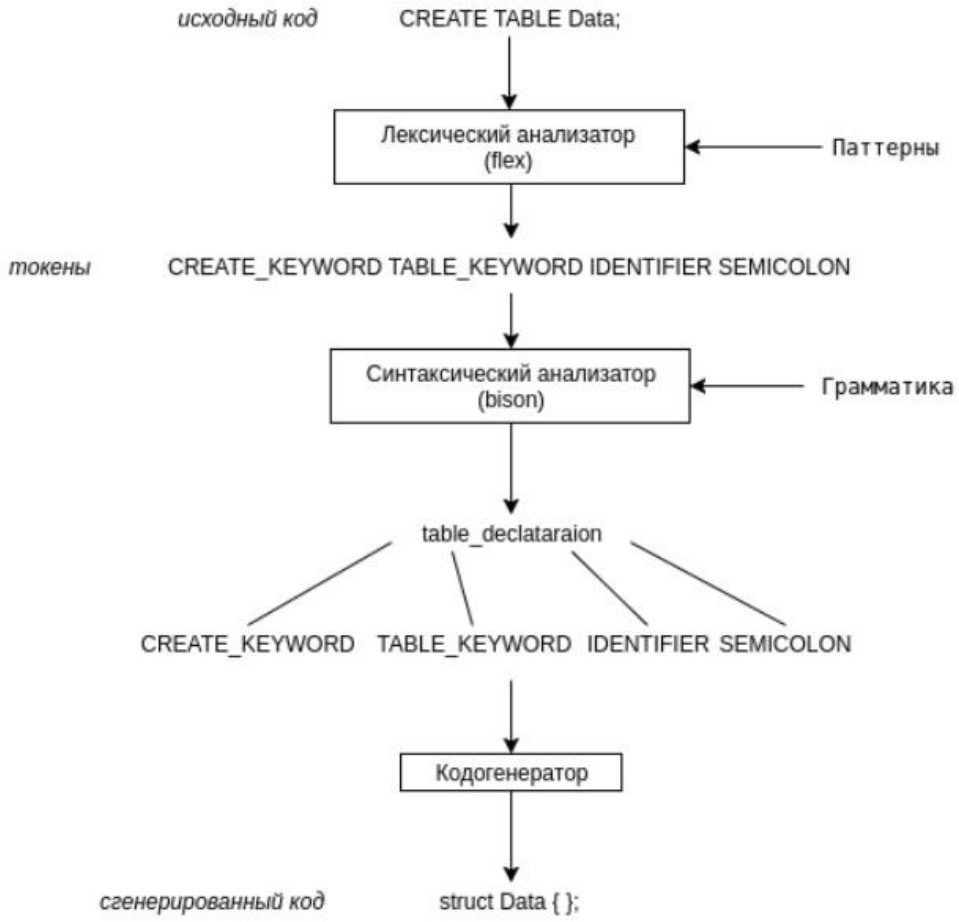


Рисунок 3 Схема работы анализатора спецификации

Описание структуры таблиц включает в себя название таблицы, описание колонок и их типы.

```
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;
```

```
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

Слова, прописанные заглавными буквами, являются ключевыми, для распознания остальных лексем используются регулярные выражения. Реализация лексера на Flex. Паттерны лексического анализатора:

```
LPAREN                  "("
RPAREN                  ")"
SEMICOLON               ";"
COMMA                   ","
POINT                   "."
AT                      "@"
EQ                      "="
TABLE_KEYWORD           ?i:"table"
CREATE_KEYWORD          ?i:"create"
PK_KEYWORD              ?i:"pk"
SELECT_KEYWORD          ?i:"select"
FROM_KEYWORD            ?i:"from"
WHERE_KEYWORD           ?i:"where"
INSERT_KEYWORD          ?i:"insert"
VALUES_KEYWORD          ?i:"values"
PROCEDURE_KEYWORD       ?i:"procedure"
BEGIN_KEYWORD           ?i:"begin"
```

```
END_KEYWORD             ?i:"end"
IN_KEYWORD              ?i:"in"
OUT_KEYWORD             ?i:"out"
SET_KEYWORD             ?i:"set"
ON_KEYWORD              ?i:"on"
AND_KEYWORD             ?i:"and"
INT_KEYWORD             ?i:"int"
FLOAT_KEYWORD           ?i:"float"
DOUBLE_KEYWORD          ?i:"double"
CHAR_KEYWORD            ?i:"char"
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
     LPAREN INTEGER RPAREN ;

parameter_mode : IN_KEYWORD|OUT_KEYWORD ;
```

1.3 Классы окружения. Ядро MillDB

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
Condition	Условие для процедуры Select проверки на равенство одной из колонок таблицы по которой производится

	выборка и параметра
Selection	Набор полей необходимых для извлечения данных

1.5 Построение плана выполнения запроса

Рассмотрим создание библиотеки для следующего запроса.

```
CREATE TABLE Person (
      id int pk,
      name char(6)
);

CREATE PROCEDURE add_Person(@id int in, @name char(6) in)
BEGIN
      INSERT TABLE Person VALUES (@id,@name);
END;

CREATE PROCEDURE get_Person_by_id(@person_id int in, @person_name
char(6) out)
BEGIN
      SELECT name SET @person_name FROM Person WHERE id=@person_id;
END;
```

В рамках этого курсового проекта необходимо рассмотреть генерацию не всех участков кода. Стоит рассмотреть структуру Person:

```
struct Person {
      int32_t id;
      char name[6];
};
```

Для второй процедуры генерируется следующий целевой код: структура с выборкой нужных полей из таблицы, структура со сформированной выборкой по которой можно итерироваться также сгенерированным итератором.

```
struct get_Person_by_id_out_data {
	char person_name[7];
};

struct get_Person_by_id_out_service {
	struct sample_handle* handle;
	struct get_Person_by_id_out_data* set;
	int size;
	int length;
	int count;
};

struct get_Person_by_id_out {
	struct get_Person_by_id_out_service service;
	struct get_Person_by_id_out_data data;
};
```

Наибольший интерес представляет функция формирования выборки.

Функция, осуществляющая формирование выборки, принимает аргумент с модификатором IN. Знание того, по какому полю будет проходить сравнение, в момент кодогенерации извлекается из набора объектов Condition объекта SelectStatment. Сформированная функция определяет является ли это поле первичным ключом.

Здесь необходимо показать как хранятся данные на диске и необходимые метаданные в оперативной памяти.

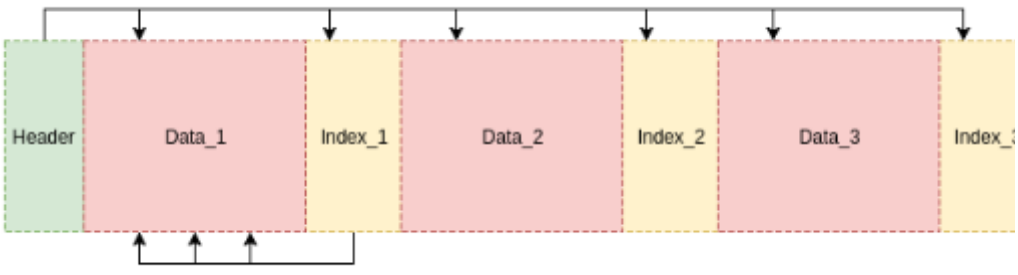


Рисунок 5: Файл базы данных

Доступ к данным осуществим через дескриптор файла, который содержит заголовок файла, набор ссылок на корни индексных деревьев. Извлечь данные по которым составлен индекс является более оптимальным подходом по сравнению с линейный просмотром всего множества на соответствие предикату из запроса. Индекс имеет структуру Б-дерева:

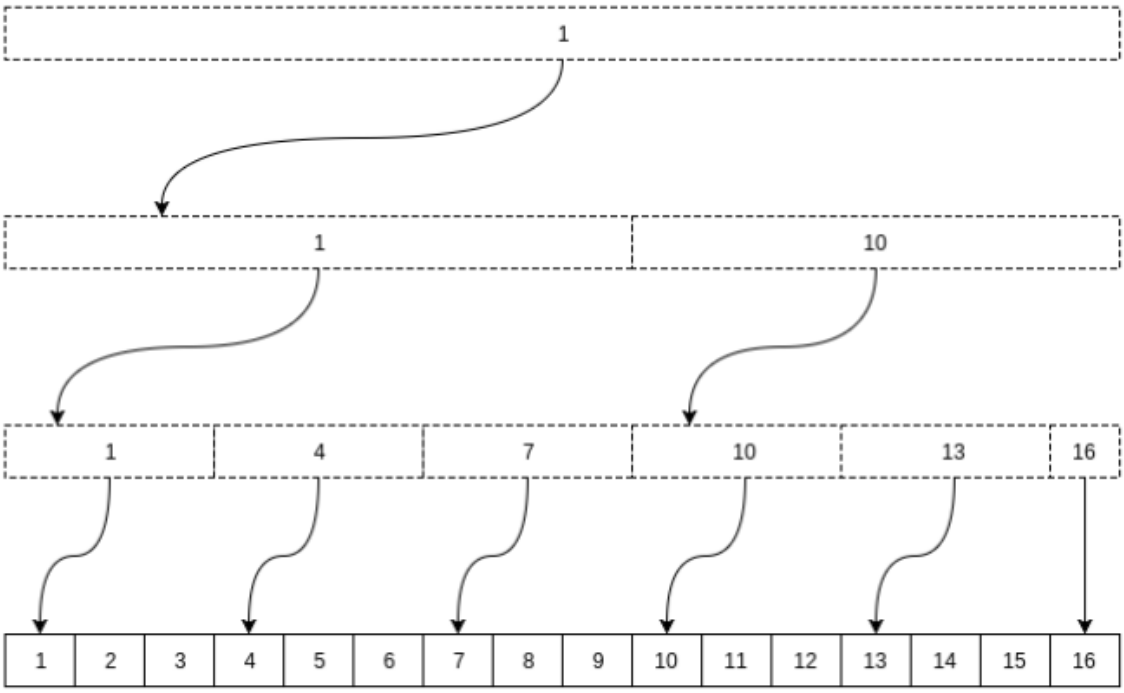


Рисунок 6: Устройство Б-дерева

Поиск нужного элемента в дереве осуществляется от корня дерева, в котором как и в других узлах содержится ключ самого левого значения из множества на которое он ссылается. Если значение ключа больше значения того узла в дереве в котором сейчас осуществляется поиск но меньше чем в соседнем узле того же уровня, то поиск переходит к потомкам данного узла. В листьях поиск завершается либо удачно, либо нет.

```
void get_Person_by_id(struct get_Person_by_id_out* iter, int32_t
person_id) {
     struct sample_handle* handle = iter->service.handle;
     struct get_Person_by_id_out_data* inserted =
malloc(sizeof(struct get_Person_by_id_out_data));
```

```
uint64_t offset = 0;

struct Person_node* node = handle->Person_root;
uint64_t i = 0;
while (1) {
	if (node->data.key == person_id || node->childs == NULL) {
		offset = node->data.offset;
		break;
	}
	if (node->childs[i]->data.key > person_id && i > 0) {
		node = node->childs[i-1];
		i = 0;
		continue;
	}
	if (i == node->n-1) {
		node = node->childs[i];
		i = 0;
		continue;
	}
	i++;
}
```

Здесь node — представляет узел индексного дерева в ОЗУ. Целью прохода по дереву является поиск смещения, на котором в области данных располагаются нужные данные. Если поиск осуществляется не по ключевому полю таблицы, то эта часть кода в шаблоне функции отсутствует.

Проход по области памяти осуществляется до тех пор пока не достигнута область где хранится индекс для данной таблицы, располагающийся сразу после области данных. Если поиск осуществляется по первичному ключу, добавляется проверка условия на непревышение значения этого поля, что имеет смысл, так как данные на диске записаны в отсортированным по первичному ключу порядке. Производится поиск данных удовлетворяющих условиям запроса.

```
offset += handle->header->data_offset[Person_header_count];

     while (1) {
          fseek(handle->file, offset, SEEK_SET);
          union Person_page page;
          uint64_t size = fread(&page, sizeof(struct Person),
                         Person_CHILDREN, handle->file);
          if (size == 0) return;

          for (uint64_t i = 0; i < Person_CHILDREN; i++) {
               const char* p_name= page.items[i].name;
               int32_t c_id= page.items[i].id;
               const char* c_name= page.items[i].name;
               if (c_id > person_id ||
                   offset + i * sizeof(struct Person) >=
                handle->header->index_offset[Person_header_count]) {
                    free(inserted);
                    return;
               }
               if (c_id == person_id) {
                    memcpy(inserted->person_name, c_name, 6);
                     inserted->person_name[6] = '\0';
                    get_Person_by_id_add(iter, inserted);
               }
          }
          offset += Person_CHILDREN * sizeof(struct Person);
     }
```

Рассмотрены шаблоны генерации целевого кода, приведя в исполнение который данная библиотека будет функционировать как база данных, выполняющая спецификацию запроса.

1.6 Выполнение запроса. 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;
}
```

Первым делом необходимо открыть файл базы данных для записи. Осуществить запись данных с применением сгенерированной функции вставки в таблицу. Необходимо закрыть запись в файл, в этот момент данные из оперативной памяти перестроятся и будут записаны на диск в соответствие с рассмотренным ранее устройством хранения данных, будут построены индексы для таблиц. Необходимо объявить структуру итератор и инициализировать вспомогательную структуру с нужным списком аргументов из запроса. Начать итерироваться по выборке. Закрыть дескриптор файла.

2. Реализация связанных таблиц

Связанные таблицы расширяют функционал MillDB. Данные из таких таблиц могут быть обработаны оператором JOIN с условием равенства по двум полям таблицы. Связь таблиц будет осуществляться вставкой в данные одной таблицы значения другого поля второй таблицы. Автоматизация вставки равных значений может быть реализована объектом последовательности Sequence.

2.1 Sequence

Последовательность CREATE SEQUENCE – это объект базы данных, который генерирует целые числа в соответствии с правилами, установленными во время его создания. Для последовательности можно указывать как положительные, так и отрицательные целые числа. В системах баз данных последовательности применяют для самых разных целей, но в основном для автоматической генерации первичных ключей. Тем не менее к первичному ключу таблицы последовательность никак не привязана, так что в некотором смысле она является еще и объектом коллективного пользования. Если первичный ключ нужен лишь для обеспечения уникальности, а не для того, чтобы нести определенный смысл, последовательность является отличным средством.

Последовательность создается командой CREATE SEQUENCE.[4]

Первое обращение к **NEXTVAL** возвращает начальное значение последовательности. Последующие обращения к **NEXTVAL** изменяют значение последовательности на приращение, которое было определено, и возвращают новое значение. Любое обращение к **CURRVAL** всегда возвращает текущее значение последовательности, а именно, то значение, которое было возвращено последним обращением к **NEXTVAL**. Прежде чем обращаться к **CURRVAL** в текущем сеансе работы, необходимо хотя бы один раз выполнить обращение к **NEXTVAL**.

В MillDB в поле первой таблицы значение последовательности NEXTVAL и в поле второй значение CURRVAL будут связывать таблицы.

2.1.1 Расширение синтаксиса входного языка

Создание объекта Sequence:

Добавление ключевых слов в milldb.l

```
SEQUENCE_KEYWORD                ?i:"sequence"
NEXTVAL_KEYWORD                 ?i:"nextval"
CURRVAL_KEYWORD                 ?i:"currval"
<INITIAL>{SEQUENCE_KEYWORD}     { debugl("SEQUENCE_KEYWORD");
                                return SEQUENCE_KEYWORD; }
<INITIAL>{NEXTVAL_KEYWORD}      { debugl("NEXTVAL_KEYWORD"); return NEXTVAL_KEYWORD; }
<INITIAL>{CURRVAL_KEYWORD}      {debugl("CURRVAL_KEYWORD");
                                 return CURRVAL_KEYWORD; }
```

Расширение правил грамматики milldb.y:

1) объявление последовательности и добавление в окружение объекта Sequence

```
program_element:  sequence_declaration {
            Sequence* sequence = Environment::get_instance()->
                                    find_sequence($1->get_name());
            if (sequence != nullptr) {
                  string msg("sequence ");
                  msg+=$1->get_name();
                  msg+=" already exists";
                  delete $1;
                  throw logic_error(error_msg(msg));
            }
            Environment::get_instance()->add_sequence($1);
      }
;
```

Правило распознавания объявления последовательности, создание экземпляра класса Sequence

```
sequence_declaration: CREATE_KEYWORD SEQUENCE_KEYWORD sequence_name
SEMICOLON {
            debug("sequence_declaration BEGIN");
```

```
            $$ = new Sequence(*$3);
            delete $3;
            debug("sequence_declaration END");
      };
```

Расширение вариаций аргумента для процедуры. Теперь она может принимать значения последовательности в поля предназначенные для связи таблиц.

```
argument: CURRVAL_KEYWORD LPAREN sequence_name RPAREN {
                  $$ = new pair<string, Argument::Type>(*$3,
                                    Argument::SEQUENCE_CURR);
                  delete $3;
            }
      | NEXTVAL_KEYWORD LPAREN sequence_name RPAREN {
                  $$ = new pair<string, Argument::Type>(*$3,
                                    Argument::SEQUENCE_NEXT);
                  delete $3;
            }
      ;
sequence_name: IDENTIFIER {
                  debug("sequence_name");
                  $$ = new string($1);
            }
      ;
```

Класс	Функциональность
Sequence	Кодогенерирует глобальную переменную В значения аргументов функций вставки передает инкремент себя либо текущее значение.

2.1.2 Пример использования

Текст запроса:

```
CREATE SEQUENCE Pet_sequence;
CREATE TABLE owner (
      oid int pk,
      oname char(6),
      pet_id int
);
CREATE TABLE pet (
      pid int pk,
      pname char(6)
);
CREATE PROCEDURE add_owner_pet(@oid int in,@oname char(6) in,@pname
char(6) in)
BEGIN
      INSERT TABLE owner VALUES (@oid, @oname,NEXTVAL(Pet_sequence));
      INSERT TABLE pet VALUES (CURRVAL(Pet_sequence),@pname);
END;
```

Фрагмент сгенерированного кода библиотеки, создающий и обрабатывающий обращение к последовательности:

```
pet_by_owner.h:
uint64_t Pet_sequence = 0 ;
pet_by_owner.c:
//функция, соответствующая процедуре вставки данных
void add_owner_pet(int32_t oid, const char* oname, const char* pname)
{
      add_owner_pet_1(oid, oname);
      add_owner_pet_2(pname);
}
void add_owner_pet_1(int32_t oid, const char* oname) {
      struct owner* inserted = owner_new();
      inserted->oid = oid;
      memcpy(inserted->oname, oname, 6);
      inserted->pet_id = ++Pet_sequence;
      owner_buffer_add(inserted);
}
```

```
void add_owner_pet_2(const char* pname) {
      struct pet* inserted = pet_new();
      inserted->pid = Pet_sequence;
      memcpy(inserted->pname, pname, 6);
      pet_buffer_add(inserted);
}
```

2.2 Оператор JOIN

Первым шагом в реализации в MillDB оператора JOIN станут расширение синтаксиса входного языка.

```
JOIN_KEYWORD                          ?i:"join"
<INITIAL>{JOIN_KEYWORD}               {debugl("JOIN_KEYWORD");
                                      return JOIN_KEYWORD;}
```

В классы окружения не пришлось внедрять новый класс. Пришлось изменить класс SelectStatement в соответствие с измененным правилом грамматики:

```
select_statement: SELECT_KEYWORD selection_list FROM_KEYWORD table_lst
                  WHERE_KEYWORD condition_list SEMICOLON
```

Теперь объект SelectStatement для процедуры хранит не одну таблицу для выборки, а их набор.

Сопутствующее правило грамматики с правилами для кодогенерации в узле дерева разбора:

```
table_lst: table_lst JOIN_KEYWORD table_name {
           $$=$1;
           Table* table = find_table(*$3);
           $$->push_back(table);
           }
     | table_name{
           $$=new vector<Table*>();
           Table* table = find_table(*$1);
           $$->push_back(table);
           }
     ;
```

Переписан метод класса SelectStatement реализующий кодогенерацию в конечную библиотеку. Теперь метод итерируется по таблицам выражения, осуществляя проверку соответствующему предикату для этой таблицы и выносу ее полей в локальные переменные для того, что бы можно было осуществить проверку условием для двух таблиц на более глубоком уровне вложенности поиска. На каждом этапе итерации порождается код для случая выполнения во внешней таблице всех соответствующих ей условий, который содержит поиск нужных данных в этой таблице по соответствующим предикатам запроса. В случае когда поиск осуществляется по первичному ключу, производится позиционирование к нужной странице данных, что оптимизирует связывание таблиц алгоритмом построения декартового произведения.

Условия WHERE запроса процедуры SELECT с применением оператора JOIN для связанных таблиц сортируются в порядке обращения к таблицам в функции составления выборки, и каждая проверка условию осуществляется на той итерации цикла, которая относится к нужной таблице. Такой подход оптимизирует поиск в сравнение с составлением полной записи из двух таблиц и последующей проверкой всех условий.

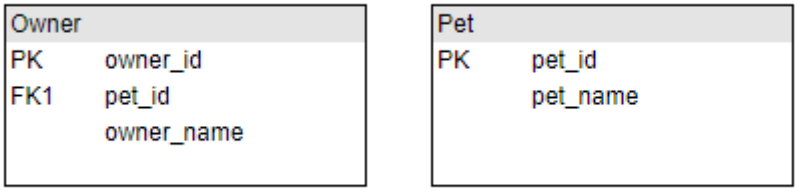
Сгенерированный код функции составления выборки для процедуры SELECT для связанных таблиц для примера из раздела тестирование демонстрируется в Приложении 1.

2.2.1 Пример использования

```
CREATE PROCEDURE get_pet_by_owner(@oname char(6) in,
                                  @pname char(6) out)
BEGIN
     SELECT pname SET @pname
     FROM owner JOIN pet WHERE oname=@oname and pet_id=pid;
END;
```

3. Тестирование

База данных:



Запрос pet_by_owner.txt:

```
CREATE SEQUENCE Pet_sequence;
CREATE TABLE owner (
      oid int pk,
      oname char(6),
      pet_id int
);
CREATE TABLE pet (
      pid int pk,
      pname char(6)
);
CREATE PROCEDURE add_owner_pet(@oid int in,@oname char(6) in,@pname
char(6) in)
BEGIN
      INSERT TABLE owner VALUES (@oid, @oname,NEXTVAL(Pet_sequence));
      INSERT TABLE pet VALUES (CURRVAL(Pet_sequence),@pname);
END;

CREATE PROCEDURE get_pet_by_owner(@oname char(6) in, @pname char(6)
out)
BEGIN
      SELECT pname SET @pname
      FROM owner JOIN pet WHERE oname=@oname and pet_id=pid;
END;
```

Пример использования сгенерированной библиотеки pet_by_owner.h pet_by_owner.c :

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

int main() {
	char owners[4][6]={"Maria","Sasha","Natal","Tanya"};
	pet_by_owner_open_write("FILE2");
	add_owner_pet(100,"Maria","Shuia");
	add_owner_pet(1001,"Sasha","Kawai");
	add_owner_pet(4,"Natal","Kazem");
	add_owner_pet(54,"Tanya","Charl");
	pet_by_owner_close_write();
	struct pet_by_owner_handle* handle1 =
pet_by_owner_open_read("FILE2");
	struct get_pet_by_owner_out iter1;
	int i;
	for (i=0;i<4;i++){
		printf("%s\t",owners[i]);
		get_pet_by_owner_init(&iter1,handle1,owners[i]);
		if (get_pet_by_owner_next(&iter1))
		printf("%s\n",iter1.data.pname);
	}
	pet_by_owner_close_read(handle1);

	return 0;
}
```

Заключение

В ходе работы над курсовым проектом произошло ознакомление с архитектурой системы управления базами данных, алгоритмами оптимизации поиска, способом хранения данных в памяти. Также применены навыки построения лексического и синтаксического анализаторов.

Результатом работы стала расширенная версия генератора баз данных MillDB, который прошел тестирование, показавшее пригодность и работоспособность продукта.

Список использованных источников

1. Когаловский М.Р.*Энциклопедия технологий баз данных*. — : Финансы и статистика, 2002. — 800 с. — ISBN 5-279-02276-4.

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. Garcia-Molina, H. *Database systems: the complete book* / H. Garcia-Molina, J.D.Ullman, J.Widom —Hardcover, Pearson, 2008 — 1203 p. - ISBN 9780131873254

4. Язык запросов SQL. Режим доступа: https://sql-language.ru/create-sequence.html (дата обращения 28.10.2018)

Приложение 1

Сгенерированная функция формирования выборки для процедуры SELECT запроса из раздела тестирование:

```
void get_pet_by_owner_1(struct get_pet_by_owner_out* iter, const char*
                                                                oname)
{
   struct pet_by_owner_handle* handle = iter->service.handle;
   struct get_pet_by_owner_out_data* inserted = malloc(sizeof(struct
                                          get_pet_by_owner_out_data));
   //TABLE owner
   uint64_t offset = 0;

   offset += handle->header->data_offset[owner_header_count];

   while (1) {
      fseek(handle->file, offset, SEEK_SET);
      union owner_page page;
      uint64_t size = fread(&page, sizeof(struct owner),
                            owner_CHILDREN, handle->file);
      if (size == 0) return;

      for (uint64_t i = 0; i < owner_CHILDREN; i++) {
         int32_t c_oid= page.items[i].oid;
         const char* c_oname= page.items[i].oname;
         int32_t c_pet_id= page.items[i].pet_id;
         if (offset + i * sizeof(struct owner) >=
                handle->header->index_offset[owner_header_count]) {
            free(inserted);
            return;
         }
         if (1) {
            if (strcmp(c_oname , oname)!=0)
               continue;

            //TABLE pet
            uint64_t offset = 0;

            struct pet_node* node = handle->pet_root;
            uint64_t i = 0;
            while (1) {
               if (node->data.key == c_pet_id || node->childs == NULL)
               {
                  offset = node->data.offset;
                  break;
               }
               if (node->childs[i]->data.key > c_pet_id && i > 0) {
                  node = node->childs[i-1];
                  i = 0;
                  continue;
               }
```

```
                if (i == node->n-1) {
                    node = node->childs[i];
                    i = 0;
                    continue;
                }
                i++;
            }

            offset += handle->header->data_offset[pet_header_count];

            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++) {
                    const char* p_pname= page.items[i].pname;
                    int32_t c_pid= page.items[i].pid;
                    const char* c_pname= page.items[i].pname;
                    if (c_pid > c_pet_id ||
                        offset + i * sizeof(struct pet) >=
                        handle->header->index_offset[pet_header_count])
                    {
                        free(inserted);
                        return;
                    }
                    if (c_pid == c_pet_id) {
                        memcpy(inserted->pname, c_pname, 6);
                        inserted->pname[6] = '\0';
                        get_pet_by_owner_add(iter, inserted);
                    }
                }
                offset += pet_CHILDREN * sizeof(struct pet);
            }

        }
    }
    offset += owner_CHILDREN * sizeof(struct owner);
}

}

void get_pet_by_owner_init(struct get_pet_by_owner_out* iter, struct
pet_by_owner_handle* handle, const char* oname) {
    memset(iter, 0, sizeof(*iter));
    iter->service.handle = handle;
    iter->service.set = NULL;
    iter->service.size = 0;
    iter->service.count = 0;
    iter->service.length = 0;
```

```
   get_pet_by_owner_1(iter, oname);
}

int get_pet_by_owner_next(struct get_pet_by_owner_out* iter) {
   if (iter == NULL)
      return 0;

   struct get_pet_by_owner_out_service* service = &(iter->service);

   if (service->set != NULL && service->count < service->length) {
      memcpy(&iter->data, &(service->set[service->count]),
sizeof(struct get_pet_by_owner_out_data));
      service->count++;
      return 1;
   } else {
      free(service->set);
   }

   return 0;
}
```