Министерство науки и высшего образования Российской Федерации

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ имени Н.Э. БАУМАНА

ФАКУЛЬТЕТ ИНФОРМАТИКИ И СИСТЕМ УПРАВЛЕНИЯ

КАФЕДРА «ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ»

Расчетно-пояснительная записка к курсовой работе на тему

«Расширение языка запросов в СУБД MillDB»

по дисциплине «*Базы данных»*

Студент ___ИУ9-62б__ _____ ___А.Б. Барлука__

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ __А.В. Коновалов_

(Подпись, дата) (И.О.Фамилия)

Москва 2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Невозможно представить современный мир без баз данных. Огромные массивы информации, которые приходится сохранять, обрабатывать и анализировать в различных областях: коммерческая деятельность, медицина, образование, обеспечение правопорядка, и так далее. Широкое применение получили аналитические базы данных – поддерживающие преимущественно операции записи и чтения хранилища для накопления статистических данных. Ограничение на множестве операций взаимодействий с БД обусловлено возможностью повысить эффективность накопления и анализа данных. Так, на кафедре «Теоретическая информатика и компьютерные технологии» в рамках выпускной квалификационной работы был разработан генератор высокопроизводительной «write-only» базы данных MillDB[1], эффективность которого была положительно отмечена при сравнительном тестировании с БД MySQL, а также аналогичными разработками.

Целями курсовой работы являются ознакомление с MillDB и развитие синтаксиса языка генератора, а именно: расширение синтаксиса логических инструкций: добавление поддержки ключевых слов «OR», «NOT», добавление операторов «<=», «=>», «<», «>», «<>» с дальнейшей оптимизацией операций чтения; а также круглых скобок для явного назначения приоритета операций.

1 Обзор MillDB

MillDB – генератор высокопроизводительной и компактной «write-only» базы данных. Под «write-only» следует понимать отсутствие поддержки операций модификации и удаления данных.

Рассмотрим основные компоненты ядра MillDB, представленные на Рисунке 1.

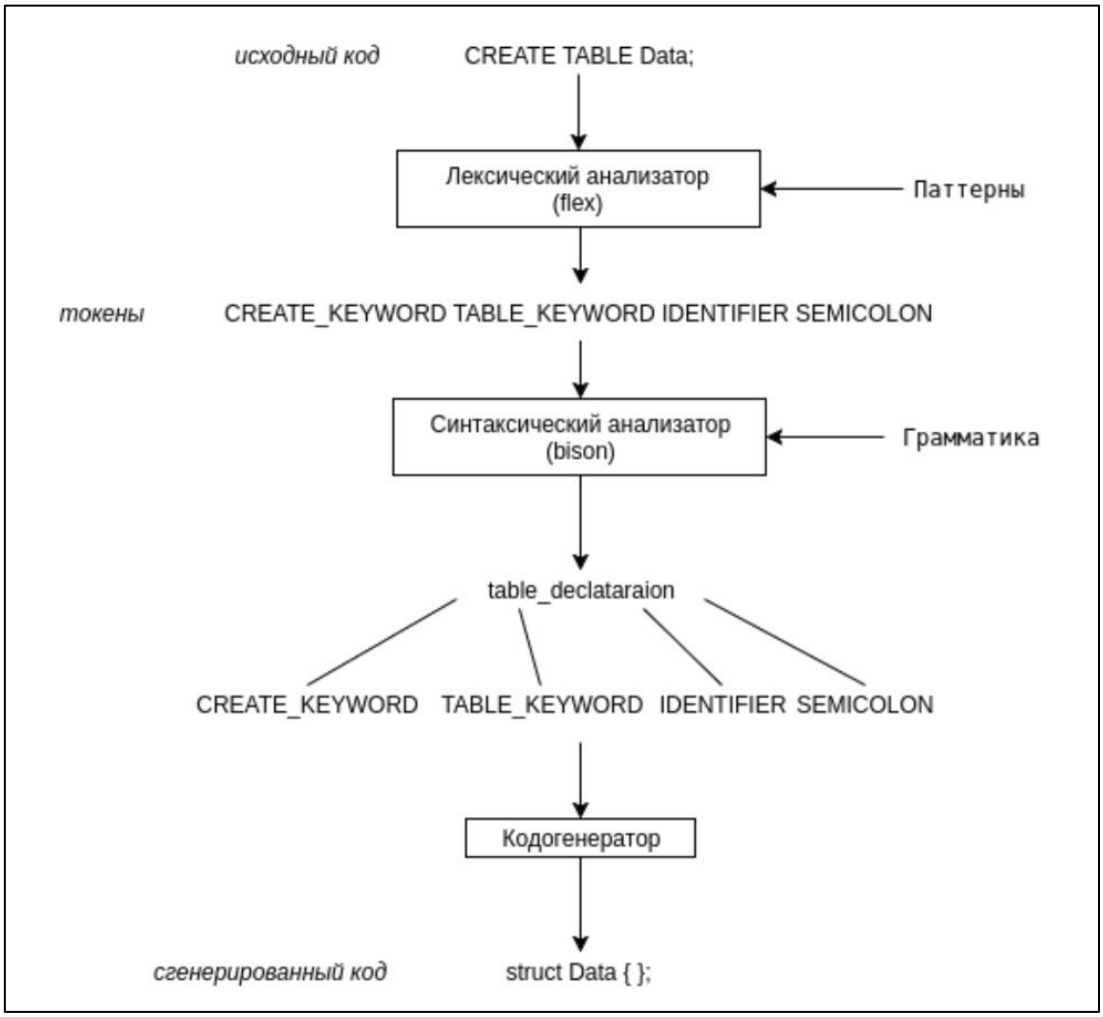

Рисунок 1 – Основные компоненты MillDB

1.1 Основные компоненты

Принцип работы MillDB следующий: на вход программе подается файл с описанием таблиц и процедур чтения/записи («исходный код»).

Исходный код проходит обработку лексическим анализатором с целью разбиения на токены для дальнейшего анализа.

Далее набор распознанных токенов передается в синтаксический анализатор и проверяется на соответствие заданной грамматике.

В случае, если грамматика была успешно распознана, происходит генерация целевого кода созданной базы данных.

Затем пользователи генератора могут подключить сгенерированные библиотеки, состоящие из заголовочного файла *.h и файла реализации *.c, и использовать предоставленные структуры и функции для открытия/закрытия базы данных, чтения из базы, запись в базу.

Сборка ядра MillDB, а также генерация БД производится с использованием утилиты Make[2], что значительно упрощает работу с генератором.

1.2 Спецификация

MillDB имеет собственную SQL-подобную спецификацию, в которой определяются таблицы и процедуры для работы с ними. Имеется поддержка последовательностей («CREATE SEQUENCE»).

Описание таблиц состоит из названия таблицы и описания колонок, включая типы (Рисунок 2):

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

Рисунок 2 – Описание таблиц на диалекте MillDB

Процедуры записи и чтения задаются следующим образом (Рисунок 3):

```
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;
```

Рисунок 3 – Описание процедур на диалекте MillDB

Пример создания базы данных с процедурами добавления и удаления записей. (Рисунок 4):

```
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 создания БД на диалекте MillDB

Также в MillDB была реализована возможность создания связанных таблиц (Рисунок 4):

```
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;
```

Рисунок 4 – Описание связанных таблиц на диалекте MillDB

1.3 Лексический анализ

Для распознавания исходного кода используется лексический анализатор Flex[3]. Flex – это быстрый генератор лексического анализатора. Это инструмент для создания программ, которые выполняют сопоставление шаблонов на текст. Flex – это свободная (но не GNU) реализация оригинальной программы Unix lex.

Слова, состоящие из заглавных букв, является ключевыми. Для распознавания остальных лексем используются регулярные выражения. На Рисунке 5 представлены примеры паттернов, по которым производится распознавание токенов.

```
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"
```

Рисунок 5 – Примеры паттернов для Flex

1.4 Синтаксический анализ

Полученный набор токенов передается на вход синтаксическому анализатору в MillDB – GNU Bison[4]. Bison использует восходящий метод разбора «перенос-свертка». Грамматика разборка задается в виде правил, как показано на Рисунке 6:

```
column_declaration_list :   column_declaration| column_declaration_list
                            COMMA column_declaration ;
```

Рисунок 6 – Пример грамматического правила в Bison

Здесь в левой части «column_declaration_list» – имя правила. После двоеточия перечислены все возможные варианты свертки. Каждое ветвление разделяется знаком «|». «column_declaration» – имя другого правила, а «column_declaration_list» в правой части означает рекурсию. Точка с запятой означает конец правила. Данное правило определяет задание списка колонок при описании таблицы.

1.5 Генерирование базы данных

В случае успешного распознавания грамматики запускается процесс генерации базы данных.

Для части синтаксический правил генерируются объекты соответствующих классов C++. Например, для правила с Рисунка 7 создается экземпляр класса Column. Класс содержит тип и название колонки таблицы, а также индикатор того, является ли колонка первичным ключом.

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

Рисунок 7 – Пример грамматического правила в Bison

Для файла спецификации <file_name>.sql будет создан заголовочный файл <file_name>.h, а также файл реализации <file_name>.c.

Так, для открытия/закрытия базы данных «X» сгенерируется следующий набор функций:

- void X_open_write(const char* filename);
- void X_close_write(void);
- struct X_handle* X_open_read(const char* filename);
- void X_close_read(struct X_handle* handle);

Пример генерируемого заголовочного файла представлен на рисунке А.1.

1.6 Выполнение запросов

Для осуществлений операций чтения или записи необходимо подключить сгенерированные библиотеки. Каждой описанной во входном файле спецификации процедуре соответствует функция на языке С стандарта ISO/IEC 9899:1999. Пример использования С-функций для выполнения запросов на Рисунке 8:

```c
#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;
}
```

Рисунок 8 – Пример выполнения запроса

2 Расширение возможностей MillDB

2.1 Оператор «NOT»

Первым этапом было решено добавить поддержку оператора «NOT». В большинстве диалектов SQL данный оператор имеет синтаксис, представленный на Рисунке 9. Функция «NOT» тривиальна – это добавление отрицания на условие «condition».

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

Рисунок 9 – Синтаксис «NOT» в распространенных диалектах SQL

Первым делом необходимо было расширить набор лексических правил (Рисунок 10).

```
NOT_KEYWORD          ?i:"not"
```

Рисунок 10 – Паттерн для распознавания ключевого слова «NOT»

Далее были обновлены правила грамматики для синтаксического анализатора (Рисунок 11).

```
search_cond_not: condition_simple {

                        $$ = new condition_tree_node();
                        $$->mode = Condition::NONE;
                        $$->value = $1;
                        $$->value->has_not = false;
                }
        | NOT_KEYWORD condition_simple {
                        debug("NOT condition");

                        $$ = new condition_tree_node();
                        $$->mode = Condition::NONE;
                        $$->value = $2;
                        $$->value->has_not = true;
                }
        ;
```

Рисунок 11 – Правило грамматики для оператора «NOT»

При кодогенерации проверяется добавленное в класс «Condition» поле «has_keyword_not» типа «bool», и если его значение равно «true» (истина), то в генерируемом коде оператор «==» (равно) заменяется на «!=» (не равно).

Алгоритм работает верно и в случае сравнения строк (т.к. для сравнения строк используется функция strcmp из стандартной библиотеки языка C, возвращающая int).

Реализация следующего оператора потребовала более интересных размышлений.

2.2 Оператор «OR»

Изначально MillDB поддерживал в условиях лишь оператор «AND». В связи с этим в памяти иерархия условий представлялась не в виде дерева, а в виде списка (соответственно, реализовано как std::vector<Condition*>).

Поэтому при реализации оператора «OR» было необходимо добавить дерево условий. Был реализован класс ConditionTreeNode, представляющий узел в дереве условий. ConditionTreeNode имеет поле mode, означающее тип узла («NONE» – обычное условие Condition, «AND» – конъюнкция условий, «OR» – дизъюнкция условий).

Если mode равно «AND» или «OR», класс имеет непустой список дочерних вершин childen, имеющий тип std::vector<ConditionTreeNode*>, не имея при этом условия Condition. На Рисунке 12 представлена пример разбора выражения.

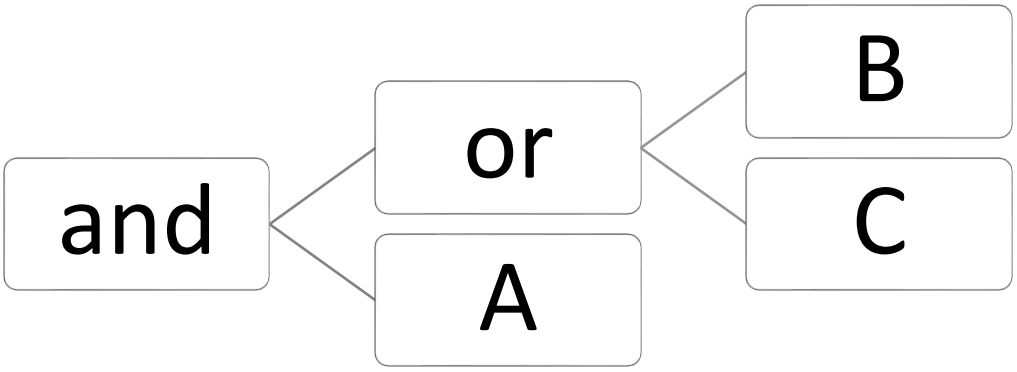

Рисунок 12 – Пример разбора деревом условий. Выражение «A and (B or C)»

Дополнение множества правил лексических паттернов и расширение грамматики проводились аналогично, как для оператора «NOT» (Рисунок 13).

```
search_cond_not OR_KEYWORD condition_list {
                debug("condition_list 3 BEGIN");

                if ($3->mode == Condition::OR) {
                        $3->children.insert($3->children.begin(), $1);
                        $$ = $3;
                } else {
                        $$ = new condition_tree_node();
                        $$->mode = Condition::OR;
                        $$->children.push_back($1);
                        $$->children.push_back($3);
                }

                debug("condition_list 3 END");
        }
```

Рисунок 13 – Синтаксическое правило для оператора «OR»

2.3 Скобочные последовательности

Имея в инструментарии два оператора с разными приоритетами («AND» имеет больший вес, нежели «OR»), необходимо уметь вручную расставлять приоритеты. Сделать это можно, например, с помощью скобочных последовательностей.

Была реализована возможность расставлять в условных выражениях парные круглые скобки в произвольном количестве (Рисунок 14)

```
CREATE PROCEDURE get_person_name(@id int in, @name char(100) out)
BEGIN
    SELECT name SET @name FROM person
    WHERE (((id = @id) AND ((id >= @id) OR (id <= @id))));
END;
```

Рисунок 14 – Условие, содержащее круглые скобки для переопределения приоритета операций

На Рисунке 15 отражены синтаксические правила (семантические действия опущены в целях сокращения объема):

```
| LPAREN condition_list RPAREN AND_KEYWORD condition_list {
| LPAREN condition_list RPAREN OR_KEYWORD condition_list {
| LPAREN condition_list RPAREN {
```

Рисунок 15 – Синтаксическое правила для круглых скобок

2.4 Операторы «<=», «=>», «<», «>», «<>»

Также были реализованы операции сравнения: «меньше или равно», «больше или равно», «меньше», «больше», «не равно». Пример использования на Рисунке 13.

Как и в случае с оператором «NOT», алгоритм для новых операторов сравнения един для целых чисел и строк.

Правила, определенные для лексического анализатора, выглядят следующим образом (Рисунок 16):

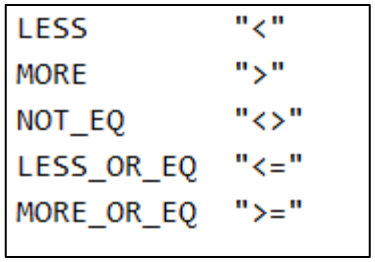

Рисунок 16 – Лексические правила для операторов «<=», «=>», «<», «>», «<>»

2.5 Оптимизация запросов поиска по первичному ключу

Изначально при запросах с условием на первичный ключ сканировалась вся БД, начиная с id=0. По достижении записи с искомым id поиск останавливался. В ходе работы были добавлены некоторые оптимизации при поиске.

Так, если вершина дерева условного выражения – это Condition, содержащий любой из операторов «<=», «=>», «<», «>», «=», то генерируется код для вычисления диапазона возможных значений id.

Например, для условия из процедуры с Рисунка 17 будет сгенерированы границы, отраженные на Рисунке 18. id_bound_l и id_bound_u – нижняя и верхняя границы данного диапазона соответственно. Пример сгенерированного кода с вычислением границ условий представлен на Рисунке А.3.

```
CREATE PROCEDURE get_people_name_with_id
(
        @id1 int in,
        @id2 int in,
        @id3 int in,
        @name char(100) out
)
BEGIN
        SELECT name SET @name FROM person
        WHERE id >= @id1 AND id < @id2 AND id > @id3;
END;
```

Рисунок 17 – Пример процедуры

```
int32_t id_bound_l = MAX(id1,id3+1);
int32_t id_bound_u = id2-1;
```

Рисунок 18 – Генерируемый код для диапазона значений id

MIN и MAX – макросы для вычисления минимального и максимального значения, соответственно (Рисунок 19).

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
```

Рисунок 19 – Макросы для вычисления минимальных и максимальных значений

Если число граничных условий больше двух, то макросы MIN и MAX будут применены рекурсивно.

3 Тестирование

Описание базы данных (Рисунок 20):

```
CREATE TABLE person (
        id int pk,
        age int,
        name char(100)
);

CREATE PROCEDURE add_person(@id int in, @name char(100) in, @age int in)
BEGIN
        INSERT TABLE person VALUES (@id, @age, @name);
END;

CREATE PROCEDURE get_people_name_older_than_age(@age int in, @name char(100) out)
BEGIN
        SELECT name SET @name FROM person WHERE age > @age;
END;
```

```
CREATE PROCEDURE get_people_name_with_id(@id1 int in, @id2 int in, @id3 int in, @id4 int in, @name char(100) out)
BEGIN
       SELECT name SET @name FROM person WHERE id >= @id1 AND id < @id2 AND id > @id3 AND id <= @id4;
END;
```

```
CREATE PROCEDURE get_people_name_with_id_2(@id int in, @name char(100) out)
BEGIN
        SELECT name SET @name FROM person WHERE id <= @id;
END;

CREATE PROCEDURE get_people_name_with_id_3(@id int in, @name char(100) out)
BEGIN
        SELECT name SET @name FROM person WHERE (NOT id >= @id) OR (id = @id AND id >= @id);
END;
```

Рисунок 20 – Тестовое описание БД

Тестовый код на C (Рисунок 21):

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

#define FILE_NAME "FILE_TEST"

int main() {
    // Write sample people with their name and ages to file.
    char people[4][100] = {"Ivan", "Sergey", "Nikolas", "Daniel"};
    test_logic_open_write(FILE_NAME);
    add_person(0, people[0], 18);
    add_person(1, people[1], 19);
    add_person(2, people[2], 20);
    add_person(3, people[3], 18);
    test_logic_close_write();
    struct test_logic_handle *handle = test_logic_open_read(FILE_NAME);
```

```
// Test
printf("age > 18:\n");
struct get_people_name_older_than_age_out iter1;
get_people_name_older_than_age_init(&iter1, handle, 18);
while (get_people_name_older_than_age_next(&iter1)) {
    printf("%s\n", iter1.data.name);
}
printf("\n");

// Test
printf("id >= 0 and id < 3 and id > 0 and id <= 3:\n");
struct get_people_name_with_id_out iter2;
get_people_name_with_id_init(&iter2, handle, 0, 3, 0, 3);
while (get_people_name_with_id_next(&iter2)) {
    printf("%s\n", iter2.data.name);
}
printf("\n");

// Test
printf("id <= 2:\n");
struct get_people_name_with_id_2_out iter3;
get_people_name_with_id_2_init(&iter3, handle, 2);
while (get_people_name_with_id_2_next(&iter3)) {
    printf("%s\n", iter3.data.name);
}
printf("\n");
```

```
    // Test
    printf("not id >= 1 or (id = 1 and id >= 1):\n");
    struct get_people_name_with_id_3_out iter4;
    get_people_name_with_id_3_init(&iter4, handle, 1);
    while (get_people_name_with_id_3_next(&iter4)) {
        printf("%s\n", iter4.data.name);
    }
    printf("\n");

    test_logic_close_read(handle);
    return 0;
}
```

Рисунок 21 – Тестовый код на ISO/IEC 9899-1999

Сгенерированный код представлен в приложении А.

Результат работы программы:

age > 18:
Sergey
Nikolas

id >= 0 and id < 3 and id > 0 and id <= 3:
Sergey
Nikolas

id <= 2:
Ivan
Sergey
Nikolas

not id >= 1 or (id = 1 and id >= 1):
Ivan
Sergey

ЗАКЛЮЧЕНИЕ

Таким образом, были достигнуты поставленные в начале работы цели: изучены возможности генератора MillDB, сделан обзор. Был расширен язык, использованы технологии для сближения диалекта MillDB с множеством SQL-диалектов. Реализованы оптимизации для эффективной выборки данных из БД. Отработаны навыки построения лексических и синтаксических анализаторов. Произведено тестирование добавленных возможностей.

СПИСОК ЛИТЕРАТУРЫ

1. Репозиторий MillDB: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/mill-db/. (Дата обращения: 29.09.2019).
2. GNU Make Manual: [Электронный ресурс]. URL: https://www.gnu.org/software/make/manual/. (Дата обращения: 29.09.2019).
3. Flex: [Электронный ресурс]. URL: https://www.gnu.org/software/flex/. (Дата обращения: 29.09.2019).
4. Bison - GNU Project - Free Software Foundation: [Электронный ресурс]. URL: https://www.gnu.org/software/bison/. (Дата обращения: 29.09.2019).

ПРИЛОЖЕНИЕ А

В этом приложении приведен листинг кода библиотеки, генерируемой MillDB: заголовочный файл test_logic.h и файл реализации test_logic.c.

```c
#ifndef TEST_LOGIC_H
#define TEST_LOGIC_H

#include <stdint.h>

#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))

struct test_logic_handle;

void add_person(int32_t id, const char* name, int32_t age);

struct get_people_name_older_than_age_out_data {
    char name[101];
};

struct get_people_name_older_than_age_out_service {
    struct test_logic_handle* handle;
    struct get_people_name_older_than_age_out_data* set;
    int size;
    int length;
    int count;
};
```

```
struct get_people_name_older_than_age_out {
    struct get_people_name_older_than_age_out_service service;
    struct get_people_name_older_than_age_out_data data;
};

void get_people_name_older_than_age_init(struct get_people_name_older_than_age_out *iter,
                                         struct test_logic_handle *handle, int32_t age);

int get_people_name_older_than_age_next(struct get_people_name_older_than_age_out *iter);

struct get_people_name_with_id_out_data {
    char name[101];
};

struct get_people_name_with_id_out_service {
    struct test_logic_handle *handle;
    struct get_people_name_with_id_out_data *set;
    int size;
    int length;
    int count;
};

struct get_people_name_with_id_out {
    struct get_people_name_with_id_out_service service;
    struct get_people_name_with_id_out_data data;
};
```

```
void
get_people_name_with_id_init(struct get_people_name_with_id_out *iter,
                             struct test_logic_handle *handle, int32_t id1,
                             int32_t id2, int32_t id3, int32_t id4);

int get_people_name_with_id_next(struct get_people_name_with_id_out *iter);

struct get_people_name_with_id_2_out_data {
    char name[101];
};

struct get_people_name_with_id_2_out_service {
    struct test_logic_handle *handle;
    struct get_people_name_with_id_2_out_data *set;
    int size;
    int length;
    int count;
};
```

```
struct get_people_name_with_id_2_out {
    struct get_people_name_with_id_2_out_service service;
    struct get_people_name_with_id_2_out_data data;
};

void get_people_name_with_id_2_init(struct get_people_name_with_id_2_out *iter,
                                    struct test_logic_handle *handle,
                                    int32_t id);

int get_people_name_with_id_2_next(struct get_people_name_with_id_2_out *iter);

struct get_people_name_with_id_3_out_data {
    char name[101];
};

struct get_people_name_with_id_3_out_service {
    struct test_logic_handle *handle;
    struct get_people_name_with_id_3_out_data *set;
    int size;
    int length;
    int count;
};
```

```
struct get_people_name_with_id_3_out {
    struct get_people_name_with_id_3_out_service service;
    struct get_people_name_with_id_3_out_data data;
};

void get_people_name_with_id_3_init(struct get_people_name_with_id_3_out *iter,
                                    struct test_logic_handle *handle,
                                    int32_t id);

int get_people_name_with_id_3_next(struct get_people_name_with_id_3_out *iter);

void test_logic_open_write(const char *filename);

void test_logic_close_write(void);

struct test_logic_handle *test_logic_open_read(const char *filename);

void test_logic_close_read(struct test_logic_handle *handle);

#endif
```

Рисунок А.1 – Сгенерированный код: заголовочный файл

```c
void get_people_name_older_than_age_1(struct get_people_name_older_than_age_out *iter, int32_t age) {
//table person  cond: age > @age
    struct test_logic_handle *handle = iter->service.handle;
    struct get_people_name_older_than_age_out_data *inserted = malloc(
            sizeof(struct get_people_name_older_than_age_out_data));
//TABLE person
    uint64_t offset = 0;

    offset += handle->header->data_offset[person_header_count];
    int32_t id_bound_l = 0;
    int32_t id_bound_u = 2147483647;
    offset += id_bound_l * sizeof(struct person);

    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;
            int32_t c_age = page.items[i].age;
            const char *c_name = page.items[i].name;
            if (offset + i * sizeof(struct person) >= handle->header->index_offset[person_header_count]) {
                free(inserted);
                return;
            }
            if (1) {
                if (!((c_age > age)))
                    continue;
                memcpy(inserted->name, c_name, 100);
                inserted->name[100] = '\0';
                get_people_name_older_than_age_add(iter, inserted);
            }
        }
        offset += person_CHILDREN * sizeof(struct person);
    }

}
```

Рисунок А.2 – Сгенерированный код: генерация условия для выборки

```
void get_people_name_with_id_1(struct get_people_name_with_id_out *iter,
                               int32_t id1, int32_t id2, int32_t id3,
                               int32_t id4) {
//table person  cond: id >= @id1
//table person  cond: id < @id2
//table person  cond: id > @id3
//table person  cond: id <= @id4
    struct test_logic_handle *handle = iter->service.handle;
    struct get_people_name_with_id_out_data *inserted = malloc(sizeof(struct get_people_name_with_id_out_data));
//TABLE person
    uint64_t offset = 0;

    struct person_node *node = handle->person_root;
    uint64_t i = 0;
    while (1) {
        if (node->data.key == id1 || node->childs == NULL) {
            offset = node->data.offset;
            break;
        }
        if (node->childs[i]->data.key > id1 && 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[person_header_count];
       int32_t id_bound_l = MAX(id1, id3 + 1);
       int32_t id_bound_u = MIN(id2 - 1, id4);
       offset += id_bound_l * sizeof(struct person);

       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;
               int32_t c_age = page.items[i].age;
               const char *c_name = page.items[i].name;
              if (offset + i * sizeof(struct person) >
                  handle->header->data_offset[person_header_count] + id_bound_u * sizeof(struct person)){
                   free(inserted);
                   return;
               }
               if (1) {
                   if (!(((c_id >= id1) && (c_id < id2) && (c_id > id3) && (c_id <= id4))))
                       continue;

                   memcpy(inserted->name, c_name, 100);
                   inserted->name[100] = '\0';
                   get_people_name_with_id_add(iter, inserted);
               }
           }
           offset += person_CHILDREN * sizeof(struct person);
       }

}
```

Рисунок А.3 – Сгенерированный код: генерация условия для выборки