Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Московский государственный технический университет

имени Н.Э. Баумана

(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовой работе по дисциплине
«Базы данных»
на тему:

Реализация дополнительных индексов и фильтра Блума в Mill-DB

Студент группы ИУ9-61 _____ В.В.Мельников

(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ А.В.Коновалов

(Подпись, дата) (И.О.Фамилия)

Москва, 2019

АННОТАЦИЯ

Цель работы – реализовать дополнительные индексы и фильтр Блума в NoSQL базе данных Mill-DB.

В качестве результата получено расширение генератора NoSQL баз данных, позволяющее генерировать таблицы с фильтрами Блума и дополнительными индексами на необходимых столбцах.

Работа состоит из 26 страниц, содержит 22 рисунка и 1 таблицу. При написании работы использовалось 4 источника литературы.

Содержание

ВВЕДЕНИЕ

В базах данных таблицы могут иметь колоссальное количество строк, поиск по которым необходимых элементов занимает много времени. Для решения задачи оптимизации поиска по данным в таблицах используются специальные объекты – индексы, которые упорядочивают данные в порядке, значительно ускоряющим скорость поиска целевых объектов.

В помощь индексам также может использоваться другая структура – фильтр Блума, которая сильно уменьшает время до ответа от базы при запросах на несуществующие строки таблиц.

Целью данной работы является расширение генератора баз данных Mill-DB путем добавления в него функционала по созданию фильтра Блума и дополнительных индексов.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 База данных

База данных – это хранилище информации, в котором данные представлены в той или иной форме [1]. Эта форма различается в зависимости от целей, для которой была создана эта база данных.

Например, для хранения данных о пользователях социальной сети понадобится база, на которой оптимизированы операции изменения и удаления данных, в то время как для баз, использующихся для хранения логов веб-серверов этими операциями можно вовсе пренебречь, ограничившись лишь вставкой и поиском.

Некоторые типы баз:

- Реляционная база данных – база данных, которая хранит связанные между собой данные в виде двумерных таблиц. Строки таблицы соответствуют элементу данных (например, товар в таблице со списком товаров), а столбцы – атрибутам этих элементов (например, доступное количество товара на складе). Связи между строками различных таблиц хранятся в самой структуре базы данных (Рисунок 1).

- Документоориентированная база данных – база данных, в которой данные хранятся в виде JSON (или JSON-подобных) документах [2]. Этот тип баз позволяет хранить данные с различной структурой в одной коллекции (аналог таблиц из реляционных баз данных). Данный тип не подразумевает связей между элементами коллекций на уровне базы.

- Графовая база данных – база данных, предназначенная главным образом для хранения взаимосвязей между объектами [3]. Данные

хранятся в виде графов, где узлы представляют собой объекты сущностей, а ребра – взаимосвязи (Рисунок 2).

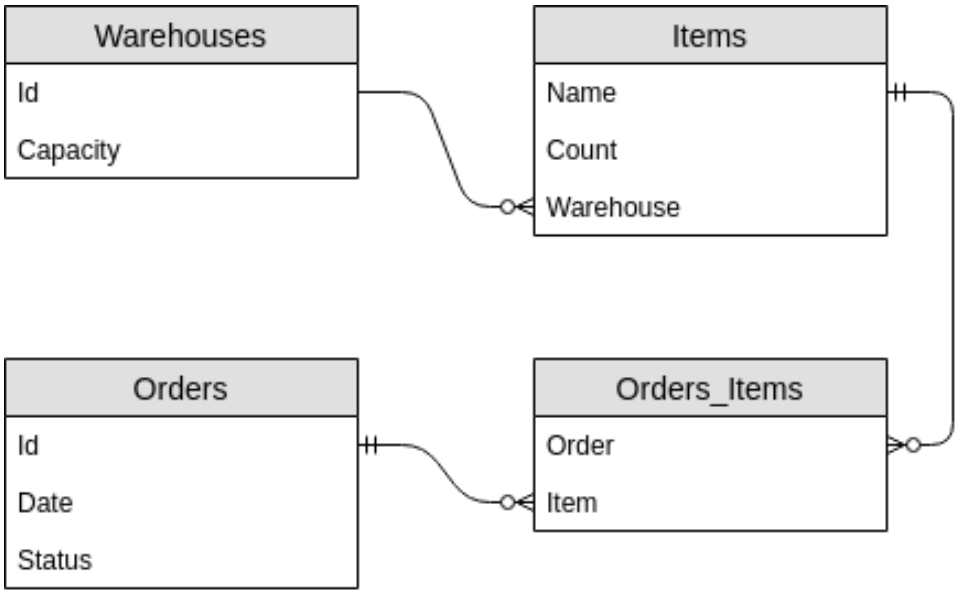

Рисунок 1 *– Реляционная база данных.*

Также базы данных могут различаться по видам хранимых в них данных, например:

- геоданные – данные о географических единицах (точки, полигоны, расстояния);
- мультимедийные данные – аудиозаписи, видеозаписи, изображения, графическая информация;
- полнотекстовые данные – данные, состоящие из текста, поиск на которых происходит по содержимому.

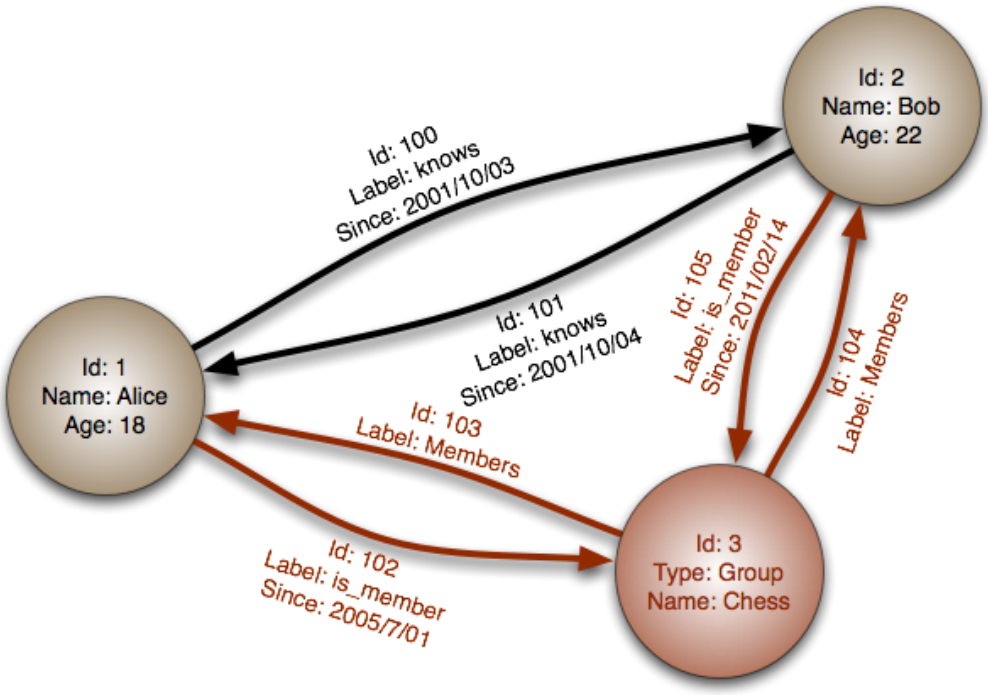

Рисунок 2 *– Графовая база данных.*

1.2 Обзор генератора баз данных Mill-DB

Mill-DB – генератор высокопроизводительных баз данных с C-интерфейсом.

Сгенерированная база может обладать операциями записи и выборки данных. Операции выборки данных из базы можно проводить только после проведения всех операций записи – запись на диск происходит только во время удаления дескриптора базы на запись, после чего дальнейшая запись в базу становится невозможной.

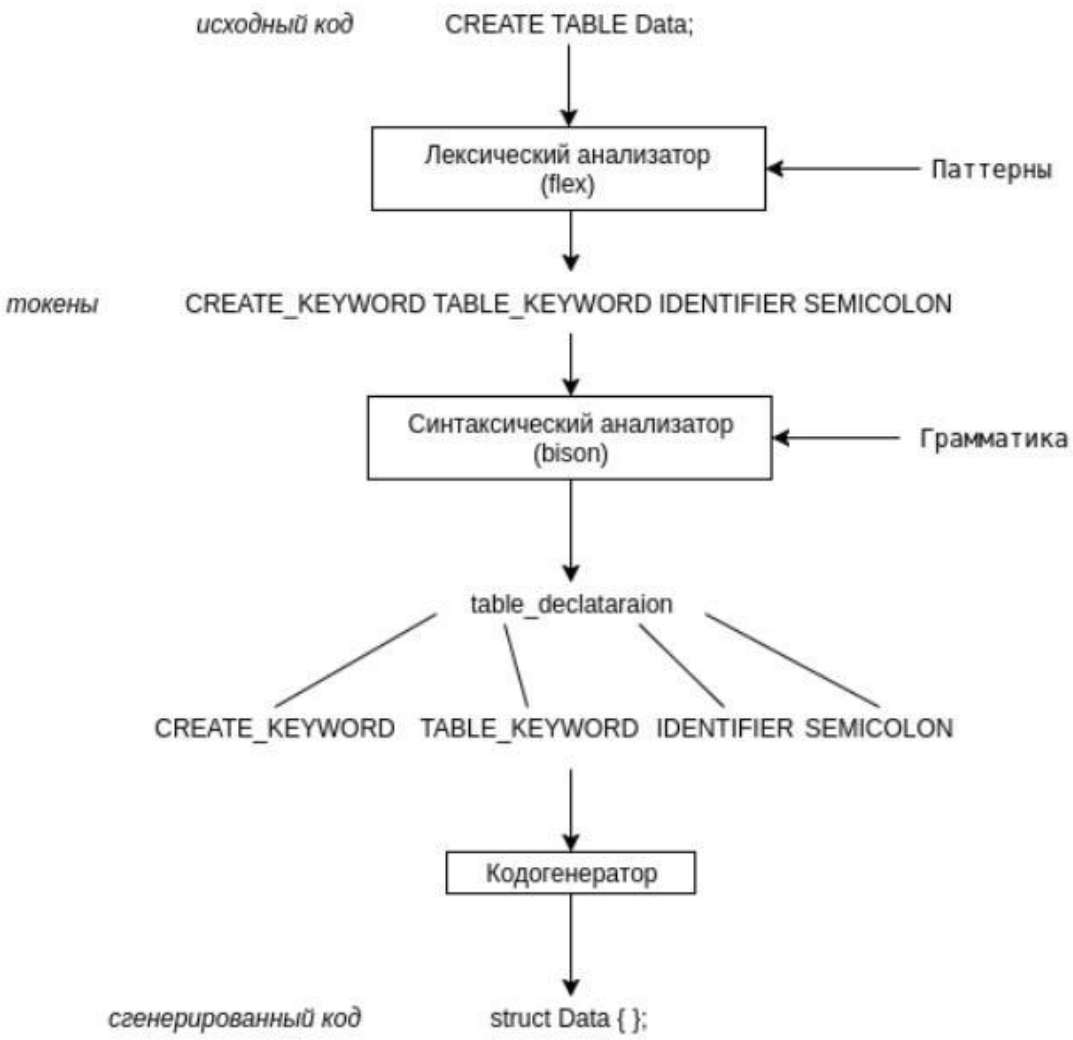

Рисунок 3 *– Схема работы анализатора входного языка.*

Mill-DB пренебрегает операциями изменения и удаления данных, благодаря чему и была получена высокая производительность на поддерживаемых операциях.

Генерация целевой базы (Рисунок 3) происходит следующим образом:

1. На вход подается описание базы данных в форме SQL-подобных команд, (создание таблиц, описание их полей, а также создание необходимых процедур).

2. Входящее описание базы обрабатывается лексическим анализатором, сгенерированным Flex.

3. Далее последовательность токенов проходит через парсер, полученный при помощи генератора синтаксических анализаторов Bison.

4. После этого происходят операции кодогенерации с последующей записью исходного кода в файлы DBNAME.c и DBNAME.h с C-реализацией полученных на вход методов.

1.3 Грамматика входного языка Mill-DB

Входной язык генератора баз Mill-DB является диалектом языка SQL. Грамматика описания таблицы представлена на рисунке 4. На рисунке 5 представлена грамматика описания процедур.

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

Рисунок 4 *– Описание таблицы.*

```
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 5 *– Описание процедур.*

Паттерны лексического анализатора предоставлены на рисунке 6.

```
LPAREN      "("
RPAREN      ")"
SEMICOLON   ";"
COMMA       ","
AT          "@"
EQ          "="

TABLE_KEYWORD       ?i:"table"
JOIN_KEYWORD        ?i:"join"

SEQUENCE_KEYWORD    ?i:"sequence"
NEXTVAL_KEYWORD     ?i:"nextval"
CURRVAL_KEYWORD     ?i:"currval"

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"
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}+
DIGIT               [[:alnum:]]

COMMENT_START       "--"

NEWLINE             \n
WHITESPACE          [ \t\n]+
```

Рисунок 6 *– Паттерны лексера.*

```
CREATE TABLE employee (
    name char(5) pk,
    post char(6),
    salary int
);

CREATE PROCEDURE add_employee(@name char(5) in, @post char(6) in, @salary int in)
BEGIN
    INSERT TABLE employee VALUES (@name, @post, @salary);
END;

CREATE PROCEDURE get_employee(@post char(6) in, @salary int in, @name char(5) out)
BEGIN
    SELECT name SET @name FROM employee WHERE post = @post and salary = @salary;
END;
```

Рисунок 7 *– Пример создания базы.*

Пример создания базы с таблицей работников предоставлен на рисунке 7. Таблица employee содержит три столбца – имя работника (name), являющееся первичным ключом, должность (post) и зарплата (salary). Также для этой базы определено две процедуры – add_employee и get_employee. Add_employee принимает на вход имя, должность и зарплату нового работника и добавляет его в таблицу. Get_employee возвращает имена всех работников из таблицы employee, занимающих определенную должность и получающих определенную зарплату.

1.4 Файл данных Mill-DB

Файл данных, с которым работает сгенерированная при помощи Mill-DB база, делится на несколько частей (Рисунок 8):

1. Заголовок – располагается в самом начале файла и хранит в себе общую информацию о базе: количество записей в каждой из таблиц, а также смещения относительно начала файла остальных секторов.

2. Часть файла с данными – здесь хранятся все строки таблицы. Для каждой таблицы создается своя часть с данными.

3. Индекс – содержит информацию об индексном дереве для таблицы и располагается сразу после каждой части с данными. Узлы индекса содержат первичные ключи и смещения для каждой записи, расположенной первой в части с данными и выгружаются в оперативную память каждый раз при открытии файла данных на чтение.

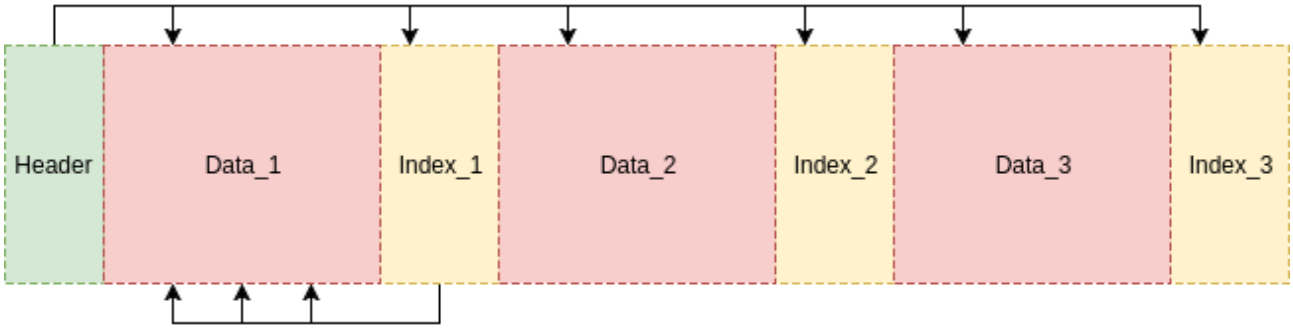

Рисунок 8 – *Схема файла базы данных.*

Индекс реализован как Б-дерево (Рисунок 9).

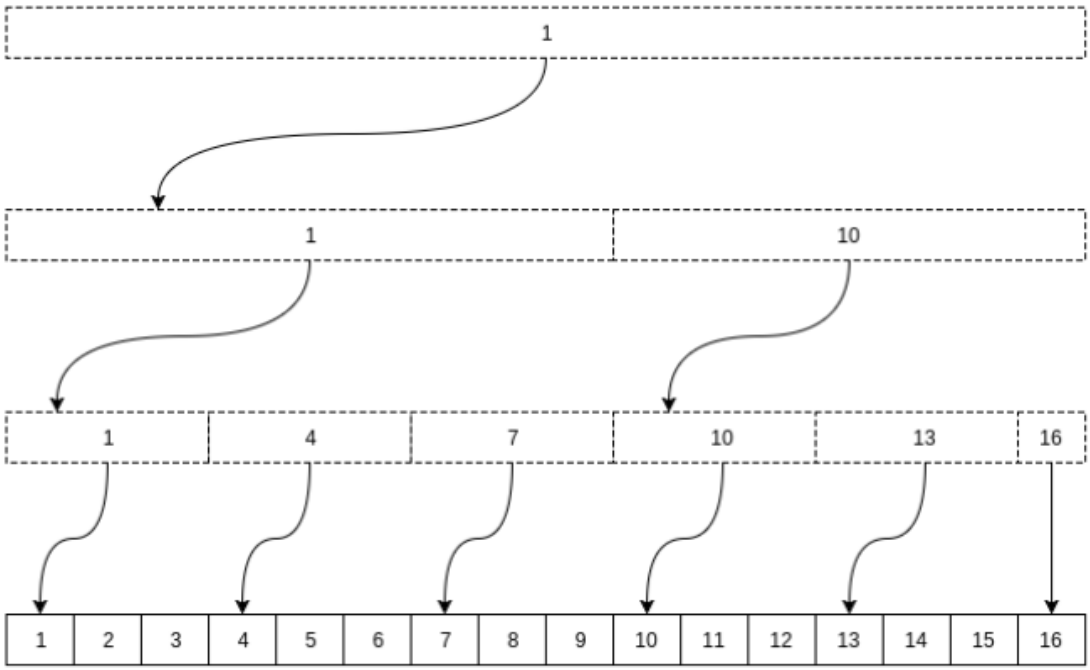

Рисунок 9 – *Структура Б-дерева.*

Индекс ускоряет запросы с поиском по первичному ключу – вместо просмотра всей части файла с данными при помощи индекса можно найти смещение от начала файла необходимой страницы данных, на которой располагается необходимая запись.

1.5 Фильтр Блума

Фильтр Блума – одна из реализаций вероятностного множества, которую придумал Бёртон Блум в 1970 году.

Вероятностное множество – множество с операциями добавления элемента и запроса принадлежности элемента к множеству. Операция запроса на принадлежность возвращает не привычные определенные ответы «да» или «нет», а либо «возможно», либо «нет». «Возможно» означает, что элемент может принадлежать, а может не принадлежать множеству, в то время, как «нет» определенно сообщает о том, что элемента в этом множестве нет.

Данные в фильтре Блума хранятся в виде битового массива из m элементов, а также набора из k хеш-функций. Каждая хеш-функция отображает входящие элементы на одну из позиций битового массива. При инициализации фильтра Блума все его элементы равны нулю. Для добавления элемента необходимо подать его на вход каждой из хеш-функций и обратить каждый элемент битового массива, соответствующий номерам из выходных значений, в единицу. Для проверки на принадлежность элемента к множеству нужно проверить соответствующие биты массива, и если все из них равны единице, то элемент «возможно» находится во множестве, в ином случае – элемента точно нет.

Ложные срабатывания могут происходить по вине случайной установки соответствующих битов элемента в единицу в процессе добавления других элементов.

На рисунке 10 представлен пример фильтра Блума с битовым массивом, состоящим из девяти элементов, тремя хеш-функциями и двумя добавленными во множество элементами (A и B). При проверке на принадлежность множеству третьего элемента (C) возникает ложноположительное срабатывание из-за коллизий.

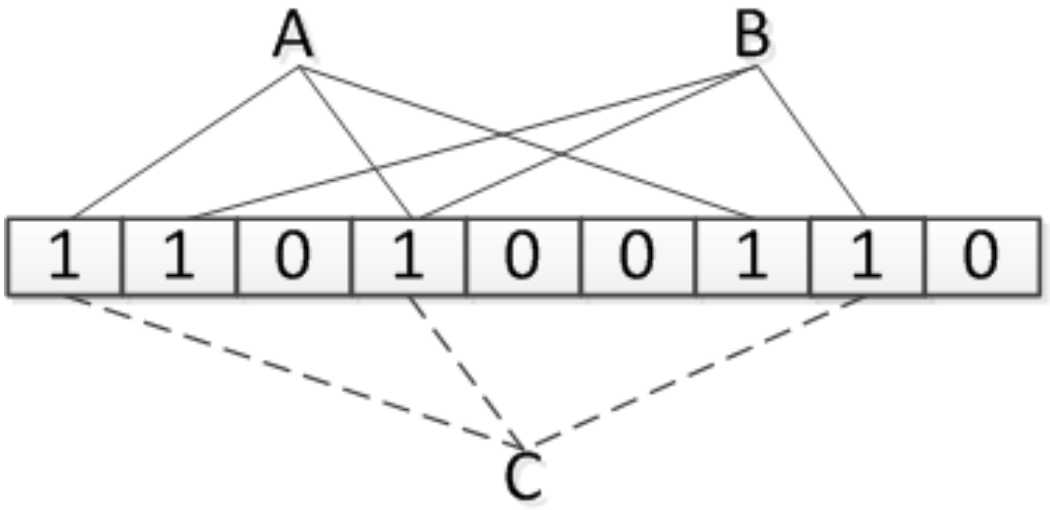

Рисунок 10 *– Пример фильтра Блума с двумя добавленными элементами и ложным положительным срабатываем для третьего.*

2. РАЗРАБОТКА

2.1 Программный интерфейс фильтра Блума

Для фильтра Блума был разработан интерфейс, показанный на рисунке 11.

```
struct bloom_filter;

struct bloom_filter *new_bf(size_t set_size, double fail_share);
void delete_bf(struct bloom_filter *bf);
void add_bf(struct bloom_filter *bf, void *item, size_t item_size);
int check_bf(struct bloom_filter *bf, void *item, size_t item_size);
```

Рисунок 11 – *Программный интерфейс фильтра Блума.*

Описание функций:

- new_bf – выделяет место в динамической памяти для фильтра Блума и возвращает указатель на него. Первым аргументом идет ожидаемое количество элементов во множестве, вторым – вероятность ложных положительных срабатываний при выполнении операции определения принадлежности элемента к множеству.

- delete_bf – освобождает место от фильтра блума в динамической памяти. На вход принимает указатель на фильтр Блума, который необходимо удалить.

- add_bf – добавляет элемент к множеству. Первым аргументом принимает указатель на фильтр Блума, вторым – указатель на добавляемый элемент, третьим – размер этого элемента в памяти (в байтах).

- check_bf – операция на принадлежность элемента ко множеству. Первым аргументом принимает указатель на фильтр Блума, вторым

– указатель на проверяемый элемент, третьим – размер этого элемента в памяти (в байтах).

2.2 Реализация интерфейса фильтра Блума

Выбранная хеш-функция показана на рисунке 12. Первые два параметра – начальные значения для генерации хеша – одно уникально для функции, через которую прогоняется входящий элемент, и генерируется при создании фильтра Блума, а второе – для самого элемента и генерируется перед добавлением или проверкой элемента. Третий параметр – указатель на входной элемент, а четвертый – его размер в байтах.

```
size_t _hash_str(size_t seed1, size_t seed2, size_t mod, const char *str, size_t str_size) {
    unsigned int res = 1;
    for (int i = 0; i < (int)str_size; i++) {
        res = (seed1 * res + seed2 * str[i]) % mod;
    }
    return res;
}
```

Рисунок 12 *– Хеш-функция.*

Для расчета оптимального размера битового массива необходимо воспользоваться формулой [4]:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

где p — вероятность ложных положительных срабатываний при выполнении операции определения принадлежности элемента к множеству, n — ожидаемое количество элементов во множестве.

Для расчета оптимального количества хеш-функций [4]:

$$k = \frac{m}{n} \ln 2$$

Получившийся код конструктора (new_bf) можно увидеть на рисунке 13.

```
struct bloom_filter *new_bf(size_t set_size, double fail_share) {
    fail_share = fail_share < 0.01 ? 0.01 :
                 fail_share > 0.99 ? 0.99 : fail_share;

    size_t cell_size = (-1.0 * (double)set_size * log(fail_share)) / pow(log(2),2);
    cell_size = cell_size < 1 ? 1 : cell_size;
    size_t hashes_size = log(2) * (double)cell_size / set_size;
    hashes_size = hashes_size < 1 ? 1 : hashes_size;

    struct bloom_filter *bf = calloc(1, sizeof(struct bloom_filter));

    size_t buf_size = cell_size/8;
    if (cell_size%8 != 0) {
        buf_size++;
    }

    bf->cell = calloc(buf_size, sizeof(char));
    bf->cell_size = cell_size;

    bf->seeds = calloc(hashes_size, sizeof(size_t));
    bf->seeds_size = hashes_size;

    for (int i = 0; i < (int)hashes_size; i++) {
        bf->seeds[i] = i * 2.5 + cell_size / 10;
    }

    return bf;
}
```

Рисунок 13 – *Конструктор фильтра Блума.*

Сначала рассчитываются значения размера битового массива и количества хеш-функций, затем выделяется память и рассчитываются начальные значения для вычисления хешей для каждой хеш-функции.

На рисунке 14 показана реализация функций add_bf и check_bf. Во время выполнения функции add_bf рассчитывается начальное значение для вычисления хеша добавляемого элемента, а затем элемент проходит через каждую из функций и соответствующие биты массива устанавливаются в единицу. Функция check_bf сначала также рассчитывает начальное значение, после чего прогоняет элемент через все функции. Если обнаружилось, что один из битов равен нулю – возвращается false, иначе – true.

```c
void add_bf(struct bloom_filter *bf, void *item, size_t item_size) {
    size_t seed_str = _generate_seed_str(item, item_size);

    for (int i = 0; i < bf->seeds_size; i++) {
        _set_bool(bf->cell, _hash_str(bf->seeds[i], seed_str, bf->cell_size, item, item_size));
    }
}

int check_bf(struct bloom_filter *bf, void *item, size_t item_size) {
    size_t seed_str = _generate_seed_str(item, item_size);

    for (int i = 0; i < bf->seeds_size; i++) {
        if (!_get_bool(bf->cell, _hash_str(bf->seeds[i], seed_str, bf->cell_size, item, item_size))) {
            return 0;
        }
    }

    return 1;
}
```

Рисунок 14 *– Добавление и проверка на принадлежность элемента.*

2.3 Добавление фильтра Блума в Mill-DB

При открытии файла на чтение необходимо считать значения необходимых полей и добавить в фильтр Блума, после чего добавить фильтр Блума в структуру дескриптора базы (рисунок 15, поле TABLENAME_FIELDNAME_bloom).

При запросе на чтение (функция PROCEDURENAME_init) нужно прогонять через фильтры Блума все аргументы, для которых они создавались, и если хотя бы один из фильтров дал отрицательный ответ – сразу заканчивать выполнение функции, так как уже известно, что в базе элемента с такими параметрами нет.

```
struct my_base_handle {
    FILE* file;
    int mode;
    struct MILLDB_header* header;

    struct employee_node* employee_root;
    struct employee_post_node* employee_post_root;
    struct bloom_filter *employee_name_bloom;
    struct bloom_filter *employee_post_bloom;
};
```

Рисунок 15 *– Дескриптор базы.*

2.4 Разработка дополнительных индексов

Для хранения индексов было решено расширить файл данных (Рисунок 16) следующим образом: данные о дополнительных индексах хранятся сразу после первичных. Каждый такой кусок данных о дополнительных индексах делится на три части (Рисунок 17):

1. Массив структур с информацией о расположении элементов с определенным ключом (Рисунок 18). Содержит в себе количество таких элементов и адрес, по которому хранятся смещения от начала файла элементов таблицы.

2. Дерево для ускорения поиска по первой (описанной выше) части индекса. Устроено аналогично дереву индексов по первичным ключам.

3. Смещения элементов, с которыми и работает первая часть.

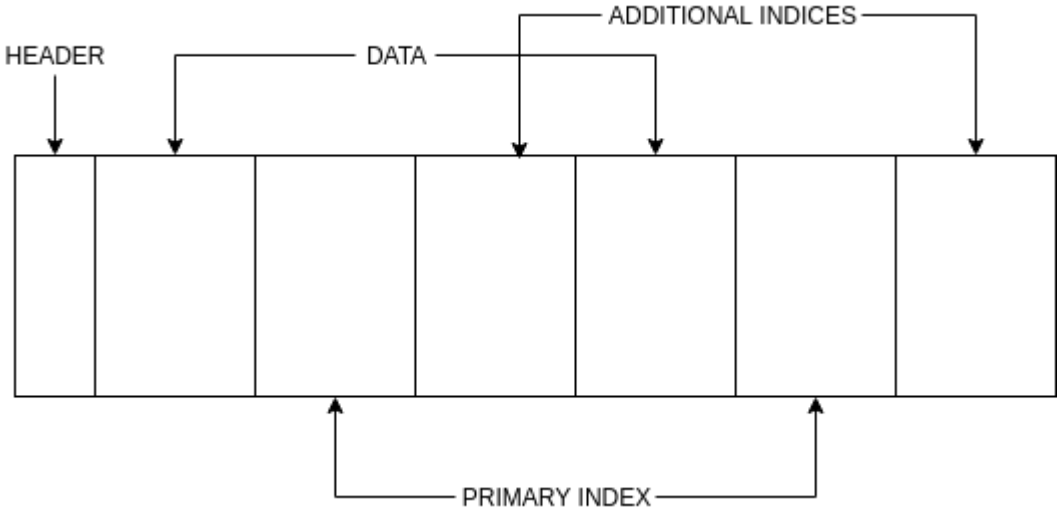

Рисунок 16 *– Новая структура файла.*

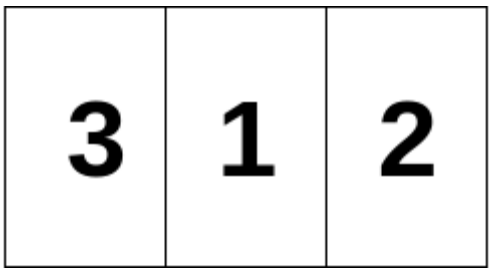

Рисунок 17 *– Структура дополнительного индекса в файле.*

```
struct employee_post_index_item {
    char key[6];
    uint64_t count;
    uint64_t offset;
};
```

Рисунок 18 *– Структура с информацией о расположении элементов с определенным ключом.*

При открытии файла на чтение для полей, на которых определен индекс, выгружаются деревья, сохраняются в дескрипторе (поле TABLENAME_FIELDNAME_root, рисунок 15) и создаются фильтры Блума. Поддерживается обратная совместимость, то есть файлы, созданные для старой версии Mill-DB, будут работать и на новой версии.

При запросе базы на чтение, если в аргументах, подающихся на вход, нет первичных ключей и есть индексированные поля, то сначала происходит поиск по одному из этих полей. По дереву ищется страница со структурой, в которой содержится информация об элементах, эта страница выгружается в память, по ней идет поиск необходимого элемента и берутся смещения от начала файла найденных элементов.

2.5 Расширение грамматики

Теперь кроме модификатора «pk» на поле можно определить также «indexed» или «bloom», что означает, что для поля будет создан фильтр Блума или дополнительный индекс.

Также можно установить необходимую вероятность ложных положительных срабатываний, поставив после модификатора «pk», «indexed» или «bloom» вещественное положительное число от 0.01 до 0.99 (0.2 по умолчанию).

Новая грамматика показана на рисунке 19.

```
CREATE TABLE table-name (
{column-name data-type {PK {float} | INDEXED {float} | BLOOM {float}}}
);
```

Рисунок 19 – *Новая грамматика создания таблицы.*

3. ТЕСТИРОВАНИЕ

Для тестирования используется база, определяемая на рисунке 7. В качестве клиентской части выступает программа, показанная на рисунке 20.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "my_base.h"

int main() {
    char name[4][6]={"Maria", "Sasha", "Natal", "Tanya"};
    char post[4][7]={"DevOps", "Develo", "DevOps", "DevOps"};
    int salarity[4]={120000, 120000, 130000, 120000};
    my_base_open_write("TESTODB");
    for (char c1 = '0'; c1 <= '9'; c1++) {
        for (char c2 = 'a'; c2 <= 'z'; c2++) {
            for (int i = 0; i < 4; i++) {
                name[i][3] = c1;
                name[i][4] = c2;
                if (c1 % 2) {
                    post[i][3] = c1;
                    post[i][4] = c2;
                }
                add_employee(name[i], post[i], salarity[i]);
            }
        }
    }
    my_base_close_write();
    struct my_base_handle* handle1 = my_base_open_read("TESTODB");
    struct get_employee_out iter1;
    struct timespec spec;
    long j = 0, sum = 0;
    for (char c1 = '0'; c1 <= '9'; c1++) {
        for (char c2 = 'a'; c2 <= 'z'; c2++) {
            for (int i = 0; i < 4; i++) {
                clock_gettime(CLOCK_REALTIME, &spec);
                long begin = spec.tv_nsec;
                post[i][3] = c1;
                post[i][4] = c2;
                get_employee_init(&iter1, handle1, post[i], 120000);
                clock_gettime(CLOCK_REALTIME, &spec);
                long end = spec.tv_nsec;
                sum += end - begin;
                j++;
            }
        }
    }
    my_base_close_read(handle1);
    printf("%ld\n", sum / j);
    return 0;
}
```

Рисунок 20 – *Тестовый клиент.*

Результаты тестирования находятся в таблице 1. В ходе тестирования сравнивалось среднее время выполнения одной операции (треть запросов с выборкой несуществующих строк) при:

1. Выборе данных по полю без фильтра Блума и индекса.
2. Выборке данных по полю с фильтром Блума, но без индекса (Рисунок 21).
3. Выборке данных по полю с индексом (Рисунок 22).

Таблица 1 *– Результаты тестов.*

№	Среднее время выполнения операции, нс.
1	24856
2	12355
3	6821

```
CREATE TABLE employee (
    name char(5) pk,
    post char(6) bloom,
    salary int
);
```

Рисунок 21 *– Таблица с полем с фильтром Блума.*

```
CREATE TABLE employee (
    name char(5) pk,
    post char(6) indexed,
    salary int
);
```

Рисунок 22 *– Таблица с индексированным полем.*

По результатам тестирования можно сделать вывод, что запросы к базе после добавления в Mill-DB фильтра Блума и дополнительных индексов стали более оптимизированы по времени выполнения.

ЗАКЛЮЧЕНИЕ

В итоге курсовой работы было разработано расширение генератора высокопроизводительных NoSQL баз данных, которое позволяет оптимизировать запросы к базе на чтение по времени. Также была сохранена обратная совместимость.

СПИСОК ЛИТЕРАТУРЫ

1. Пирогов В. // Организационные системы и базы данных: организация и проектирование. – 2009.
2. Черемнов Д. // Профессиональные компетенции разработки программного обеспечения. – 2019.
3. Клеппман М. // Высоконагруженные приложения. Программирование, масштабирование, поддержка. – 2018.
4. Луридас П. // Алгоритмы для начинающих. Теория и практика для разработчика. — 2017.

ПРИЛОЖЕНИЕ 1

Сгенерированная функция выборки (для таблицы employee, без фильтра Блума):

```
void get_employee_1(struct get_employee_out* iter, const char* post, int32_t salary)
{
//table employee    cond: post = @post
//table employee    cond: salary = @salary
        struct my_base_handle* handle = iter->service.handle;
        struct get_employee_out_data* inserted = malloc(sizeof(struct
get_employee_out_data));
//TABLE employee
        uint64_t offset = 0;


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


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


                for (uint64_t i = 0; i < employee_CHILDREN; i++) {
                        const char* p_name= page.items[i].name;
                        const char* c_name= page.items[i].name;
                        const char* c_post= page.items[i].post;
                        int32_t c_salary= page.items[i].salary;
                        if (offset + i * sizeof(struct employee) >= handle->header-
>index_offset[employee_header_count]) {
                                free(inserted);
                                return;
                        }
                        if (1) {
                                if (strcmp(c_post , post)!=0)
```

```
                        continue;

                    if (!(c_salary == salary))
                        continue;

                    memcpy(inserted->name, c_name, 5); inserted->name[5] =
'\0';
                    get_employee_add(iter, inserted);
                }
            }
            offset += employee_CHILDREN * sizeof(struct employee);
        }

}

void get_employee_init(struct get_employee_out* iter, struct my_base_handle*
handle, const char* post, int32_t salary) {
    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_employee_1(iter, post, salary);
}
```

С фильтром Блума (после многоточия код не меняется):

```
void get_employee_1(struct get_employee_out* iter, const char* post, int32_t salary)
{
//table employee    cond: post = @post
//table employee    cond: salary = @salary
```

```
        if (!is_employee_post_bloom(iter->service.handle, post)) {
                return;
        }
…
```

С индексированным полем:

```
void get_person_1(struct get_person_out* iter, const char* qwer, int32_t name) {
//table person        cond: qwer = @qwer
//table person        cond: name = @name
        if (!is_person_qwer_bloom(iter->service.handle, qwer)) {
                return;
        }
        struct my_base_handle* handle = iter->service.handle;
        struct get_person_out_data* inserted = malloc(sizeof(struct
get_person_out_data));
//TABLE person

   uint64_t info_offset;

   struct person_qwer_node* node = handle->person_qwer_root;
   uint64_t i = 0;
   while (1) {
      if (!strncmp(node->data.key, qwer, 5) || node->childs == NULL) {
         info_offset = node->data.offset;
         break;
      }
      if (strncmp(node->childs[i]->data.key, qwer, 5) > 0 && i > 0) {
         node = node->childs[i-1];
         i = 0;
         continue;
      }
```

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

    uint64_t *offsets;
    uint64_t off_count = 0;

    int break_flag = 0;
    while (1) {
        fseek(handle->file, info_offset, SEEK_SET);
        struct person_qwer_index_item items[person_qwer_CHILDREN];
        uint64_t size = fread(items, sizeof(struct person_qwer_index_item),
person_qwer_CHILDREN, handle->file); if (size == 0) return;

        for (uint64_t i = 0; i < person_qwer_CHILDREN; i++) {
            if (strncmp(items[i].key, qwer, 5) > 0 || info_offset + i * sizeof(struct
person_qwer_index_item) >= handle->header-
>add_index_tree_offset[person_qwer_index_count]) {
                free(inserted);
                return;
            }
            if (!strncmp(items[i].key, qwer, 5)) {
                off_count = items[i].count;
                offsets = malloc(sizeof(uint64_t) * off_count);

                fseek(handle->file, items[i].offset, SEEK_SET);
                fread(offsets, sizeof(uint64_t), off_count, handle->file);

                break_flag = 1;
```

```
            break;
        }
    }
    if (break_flag) {
        break;
    }
    info_offset += person_qwer_CHILDREN * sizeof(struct
person_qwer_index_item);
  }

  for (uint64_t i = 0; i < off_count; i++) {
    struct person *item = malloc(sizeof(struct person));
    fseek(handle->file, offsets[i], SEEK_SET);
    fread(item, sizeof(struct person), 1, handle->file);

    if (item->name != name) {
        free(item);
        continue;
    }

    inserted->id = item->id;
    get_person_add(iter, inserted);
  }

  free(inserted);
}
…
```