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

Факультет «Информатика и системы управления»
Кафедра «Программное обеспечение ЭВМ и информационные
технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ НА ТЕМУ:

АВТОМАТИЗАЦИЯ ПОДБОРА ИНДЕКСОВ ДЛЯ SQL ЗАПРОСОВ В СУБД MYSQL

Студент ИУ7-81 _____ И. С. Петухов

Руководитель ВКР _____ Е. А. Просуков

Консультант _____ _____

Консультант _____ _____

Нормоконтролер _____ _____

Москва, 2017

Содержание

Реферат

Индексирование решает проблему оптимизации выполнения SQL запросов в базах данных, которые уже спроектированы, используются в эксплуатации и имеют таблицы с более чем тысячи записей. В работе делается попытка облегчения работы администратора БД при построении индексов.

Рассматривается структура хранения данных в СУБД MySQL (InnoDB). Описываются кластерные, вторичные, составные, покрывающие индексы. Рассмотрены примеры применения индексов для запросов по одной таблице.

Для сложного SQL запроса (с соединением двух таблиц) предлагается алгоритм, который показывает, какие индексы необходимо построить для оптимизации выполнения этого запроса. На примерах показывается использование этого алгоритма.

Для алгоритма создана программная реализация.

Введение

Одной из главных причин низкой скорости обработки запросов является работа с базой данных (БД), которая не оптимизирована: нерациональные запросы, некорректное использование индексов, неоптимальные значения параметров конфигурации.

Оптимизация производительности сводится к следующим задачам:

а) корректировка параметров СУБД;
б) денормализация данных;
в) выявление медленных запросов и их анализ:
    1) корректировка запросов;
    2) создание индексов.

Основным приемом увеличения производительности выполнения запросов к базе данных является индексирование. Индексы представляют собой структуры данных, которые помогают СУБД эффективно извлекать данные. Они критичны для достижения хорошей производительности, но многие часто забывают о них или плохо понимают их смысл, поэтому индексирование является главной причиной проблем с производительностью в реальных условиях. [1]

СУБД используют индексы для увеличения производительности выполнения SQL запросов путем оптимизации обращений к дисковой памяти (индексы в основном находятся в памяти).

Индексы следует создавать по мере обнаружения медленных запросов. В этом может помочь лог медленных запросов. Запросы, которые выполняются более 1 секунды, являются первыми кандидатами на оптимизацию. [2] Конечно, говорить о том, сколько секунд считать медленным запросом зависит от конкретной задачи. В основном такие запросы используют JOIN оператор для соединения двух и более таблиц.

СУБД решают задачу выбора индекса для выполнения конкретного SQL запроса на работающей БД. Чтобы понять, какой индекс необходимо построить для оптимизации выполнения SQL запроса администратору БД необходимо учитывать:

а) реальные данные, которыми заполнена БД
б) статистику запросов;
в) как СУБД применяет индексы;
г) как СУБД соединяет таблицы (для сложных запросов).

Администратору БД необходимо учитывать эти факторы, чтобы построить индекс для SQL запроса.

Если абстрагироваться от реальных данных и не учитывать статистику запросов, то зная как СУБД применяет индекса можно сделать предположения, какие индексы могли бы быть использованы для конкретного запроса.

В данной работе предлагается алгоритм, возвращающий список индексов, которые СУБД могла бы использовать при выполнении этого запроса. Программа, реализующая предложенный алгоритм, облегчит работу администраторов БД, которым останется только принять решение, какой именно индекс построить из предложенного списка возможных индексов.

1 Аналитический раздел

1.1 Структура индексов в MySQL InnoDB

Индексы представляют собой структуры данных, которые помогают эффективно извлекать данные. В СУБД MySQL InnoDB используются Btree индексы [3], основанные на структуре данных B+tree (рисунок 1.1).

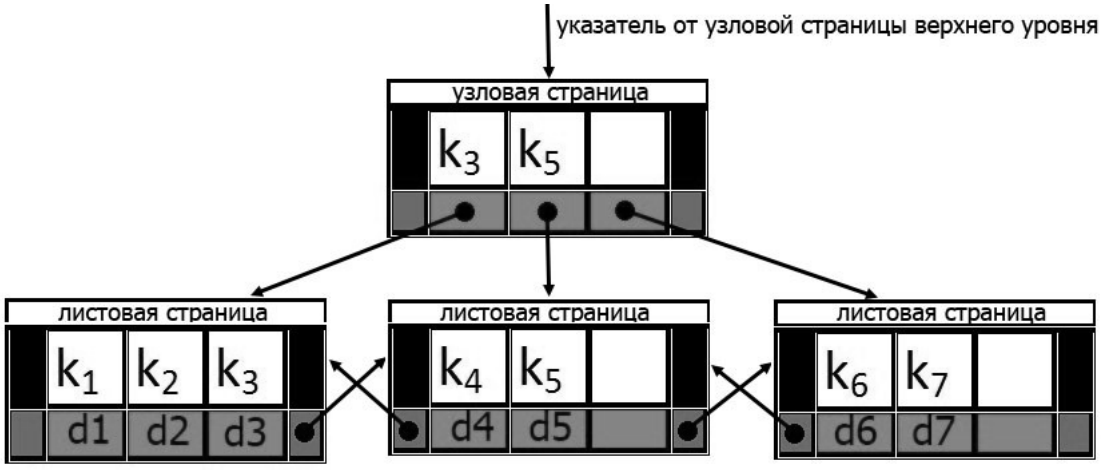

Рисунок 1.1 — Структура данных B+tree.

Где k_i - ключи ($k_i < k_i + 1$), d_i - данные.

Сложность поиска в линейной структуре данных $O(N)$, а в B+tree $O(log(k)+N_k)$, где k — количество уровней, а N_k — количество элементов в узле. Поэтому индексы, использующие структуру данных B+tree, эффективны для поиска данных. Для оптимизации поиска по диапазону в листовых страницах есть указатели на следующую и предыдущую листовую страницу.

В качестве примера рассмотрим таблицу 1.1, заголовок которой $poet(poet\_id, last\_name, first\_name, dob, country)$

Таблица 1.1 — Таблица поэтов

poet_id	last_name	first_name	dob	country
1	"Блок"	"Александр"	1880	"ru"
2	"Фет"	"Афанасий"	1820	"ru"
3	"Лермонтов"	"Михаил"	1814	"ru"
4	"Ильф"	"Илья"	1897	"ru"
5	"Пушкин"	"Александр"	1799	"ru"
6	"Булгаков"	"Михаил"	1891	"ru"
7	"Есенин"	"Сергей"	1895	"ru"

1.1.1 Кластерный индекс

В InnoDB данные хранятся в структуре данных B+tree, где в узловых страницах хранятся первичные ключи, а в листовых страницах хранятся данные. Такое дерево называется кластерным индексом. Над таблицей можно построить только один кластерный индекс, поскольку невозможно хранить одну и ту же запись одновременно в двух местах. Однако часть или всю запись можно хранить в нескольких местах, что будет использоваться в покрывающих индексах 1.1.6.

На рисунке 1.1, $k_1 \dots k_7$ - первичные ключи ($poet\_id$), $k_i = i$, $d_1 \dots d_7$ - данные, т.е. d_1 = Блок, Александр, 1880, ru; d_2 = Фет, Афанасий, 1820, ru; ...

Характерной проблемой кластерного индекса является то, что при вставке в листовую страницу, если в ней закончилось место, создается новая пустая страница. Из-за этого, со временем появляется много пустующего места и таблицы InnoDB начинают занимать много места. В MySQL данная проблема решается командой *OPTIMAZE TABLE table_name*

1.1.2 Вторичный индекс

Для оптимизации конкретных запросов используются *вторичные индексы* (далее просто *индексы*). В узловых страницах индексов хранятся поля, по которым создан этот индекс, а в листовых страницах хранится значение первичного ключа. Для каждой таблицы в одном запросе используется только один индекс. При использовании в запросах индекса, сначала будет найдено значение первичного ключа, затем по этому значению будут найдены данные в кластерном индексе. Поэтому при создании вторичного индекса, в конец неявно добавляется первичный ключ.

На рисунке 1.2 показан индекс по полю (dob). Где $k_1 \dots k_7$ - ключи, по которым построен индекс; $k_1 = 1799$, $k_2 = 1814$, $k_3 = 1820$, ...; $p_1 \dots p_7$ - первичные ключи ($poet\_id$); $p_i = i$.

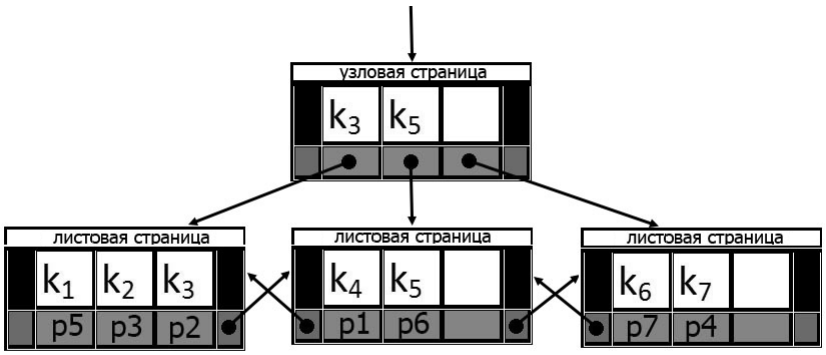

Рисунок 1.2 — Структура вторичного индекса

1.1.3 Составной индекс

Составной индекс - это индекс, построенный по нескольким полям. Чтобы правильно использовать составные индексы, необходимо понять структуру их хранения. Все работает точно так же, как и для обычного индекса, но для значений используются значения всех входящих колонок сразу, по которым строится индекс.

Если построить индекс по полям *(dob, last_name)*, то для рисунка 1.2 $k_1 =$ 1799Пушкин, $k_2 =$ 1814Лермонтов, $k_3 =$ 1820Фет, ...

1.1.4 Частичный индекс

Составной индекс может использоваться не полностью, но только как левый префикс. Такое использование индекса наызвается *частичным индексом.*

Оператор *LIKE* с использованием знака % в конце указываемого шаблона рассматривается как левый префикс.

1.1.5 Селективность индексов

Чем меньше строк войдет в выборку, тем быстрее будет работать поиск по ней. Индекс дающий наименьшую выборку называется *более селективным.* Если СУБД может применить несколько индексов к данному SQL запросу, то использоваться будет более селективный.

1.1.6 Покрывающий индекс

Индекс называется *покрывающим*, если в нем есть все поля, используемые в запросе. Для того чтобы вернуть результат запроса при использовании покрывающего индекса СУБД не нужно обращаться к кластерному индексу. Покрывающие индексы позволяют имитировать кластерные индексы.

Для запроса на листинге 1.1 можно построить индекс $(last\_name, dob)$, который будет считаться покрывающим.

Листинг 1.1 — запрос для covering-index

```
1 SELECT last_name, dob
2 FROM poet
3 WHERE last_name = "Пушкин"
```

1.2 Индексы для простых запросов

Рассмотрим примеры использования индексов для простых запросов (по одной таблице).

1.2.1 Индексы для WHERE

Для запроса на листинге 1.2 можно построить индексы *(dob, last_name)* и *(last_name, dob)*.

Листинг 1.2 — запрос для index-on-where

```
1 SELECT *
2 FROM poet
3 WHERE last_name = "Пушкин"
4     AND dob = 1799
```

Рассмотрим работу индексов на примере индекса *(a, b, c)*, где *a, b* - числа, *c* - строка.

Примеры запросов, к которым применяется индекс *(a, b, c)*

а) *a = 5 AND b = 10 AND c="Hello world"*

б) *a = 5*

в) *a > 5*

г) *a = 5 AND b = 10*

д) *a = 5 AND b = 10 AND c LIKE "Hello w%"*

е) *a = 5 AND b IN (2,3)* (*IN* - рассматривается как поиск по диапазону)

Примеры запросов, к которым не применяется индекс *(a, b, c)*

а) *b = 5*

б) *a=5 AND b=10 AND c LIKE "% world"*

в) *a=5 AND c=10*

г) *a>5 AND b=2*

1.2.2 Индексы при сортировке

В индексе данные хранятся в отсортированном виде, поэтому дополнительно сортировать данные выборки не потребуется.

Для запроса на листинге 1.3 строим индекс *(dob)*.

Листинг 1.3 — запрос для index-order

```
1 SELECT *
2 FROM poet
3 ORDER BY dob
```

Для запроса на листинге 1.4 строим индекс *(country, dob)*. В индексе находим строки, удовлетворяющие условию *country="ru"*, а в этой выборке строки уже отсортированы по *dob*.

Листинг 1.4 — запрос для index-order

```
SELECT *
FROM poet
WHERE country="ru"
ORDER BY dob
```

Для запроса на листинге 1.5 строим индекс *(dob)*. *GROUP BY* возьмет уже отсортированные строки из индекса и уберет повторы, а т.к. строки уже отсортированные - *ORDER BY* всего лишь задаст, в каком порядке выводить данные.

Листинг 1.5 — запрос для index-order

```
SELECT *
FROM poet
GROUP BY dob
ORDER BY dob
```

Рассмотрим работу индексов на примере индекса *(a, b)*, где *a, b* - числа.

Примеры запросов, к которым применяется индекс *(a, b)*

а) сортировка по первой колонке

б) первая колонка в условии *WHERE*, и сортировка по второй колонке

в) первая колонка в условии *WHERE* и сортировка по первой колонке

г) сортировка по двум колонкам и обе в одну сторону

Примеры запросов, к которым не применяется индекс *(a, b)*

а) *ORDER BY b*

б) *WHERE a>5 ORDER BY b*

Пример данных в индексе *(a, b): a=6, b=2; a=6, b=3; a=7, b=0; a=8, b=1*. MySQL сделает выборку по условию *a>5*, но в этой выборке строки не отсортированы по *b*.

в) *WHERE a IN (1,2) ORDER BY b* (аналогично предыдущему запросу)

г) сортировка разных столбцов в разных направлениях (чтобы обойти это, можно сделать виртуальную колонку, например, перед числом поставить минус)

1.2.3 Агрегирующие функции MIN, MAX

Так как данные в индексе отсортированы, то для нахождения минимального или максимального значения достаточно взять крайнее значение.

Для запроса на листинге 1.6 строим индекс (*dob*).

Листинг 1.6 — запрос для index-aggr

```
SELECT MAX(dob)
FROM poet;
```

Для запроса на листинге 1.7 строим индекс $(first\_name, dob)$.

Листинг 1.7 — запрос для index-aggr

```
1 SELECT MAX(dob)
2 FROM poet
3 GROUP BY first_name
```

1.3 Индексы для сложных запросов

1.3.1 Общие правила

а) в одном запросе для каждой таблицы используется максимум один индекс;

б) в выражениях *GROUP BY* и *ORDER BY* поля только из одной таблицы.

2 Конструкторский раздел

Под *join запросами* будем понимать запросы, вида листинг 2.1.

Листинг 2.1 — Вид join запроса

```
t1 {INNER | LEFT | RIGHT} JOIN t2 ON conditional_expr
    [WHERE where_definition]
    [ORDER BY col_name [ASC | DESC], ...]

where_definition:
    where_expr or where_expr [AND | OR] where_expr

where_expr:
    column_name [> | >= | = | <> | <= | < ]
    column_name_or_constant or
    column_name LIKE column_name_or_constant or
    column_name IS NULL or column_name IS NOT NULL or (where_definition)
```

Разрабатываемый алгоритм должен определить, какие индексы необходимо построить для оптимизации выполнения *join запроса*.

Последовательность действий:

а) определить, по какой из двух таблиц оператора *JOIN* будет осуществлено полное сканирование (учитывая, как СУБД соединяет таблицы)

б) разбить сложный запрос на два подзапроса

в) построить индексы для каждого подзапроса (учитывая правила СУБД построения индексов)

г) объединить индексы подзапросов в индекс запроса

2.1 Fullscan алгоритм

При соединении двух таблиц (A и B), для каждой строки одной таблицы A будет происходить сканирование другой таблицы B. Такое сканирование таблицы A назовем *полным сканированием*.

Fullscan алгоритм - алгоритм для определения таблицы, по которой будет осуществлено полное сканирование. Алгоритм на вход принимает текст запроса и возвращает название таблицы, по которой будет полное сканирование или NULL, если таблица не определена (в таком случае MySQL после выбора подходящих индексов выберет, по какой таблице будет происходить полное сканирование).

В качестве fullscan алгоритма для join-запроса можно предложить алгоритм, при работе которого происходит анализ исследуемого запроса. Псевдокод представлена в алгоритме 1. Блок-схема представлена на рисунке 2.1.

Algorithm 1 Fullscan алгоритм

```
1: function FULLSCANALG(query)
2:     if условие соединения A LEFT JOIN B then
3:         if в условии WHERE исключается возможность равенства любого из полей таблицы B значению NULL then
4:             заменить LEFT JOIN на INNER JOIN
5:         else
6:             return таблица A
7:         end if
8:     end if

9:     if условие соединения A RIGHT JOIN B then
10:        if в условии WHERE исключается возможность равенства любого из полей таблицы A значению NULL then
11:            заменить RIGHT JOIN на INNER JOIN
12:        else
13:            return таблица B
14:        end if
15:    end if

16:    if условие соединения A INNER JOIN B then
17:        if есть сортировка ORDER BY A.a, ... then
18:            return таблица A
19:        else if есть сортировка ORDER BY B.a, ... then
20:            return таблица B
21:        end if
22:    end if

23:    return NULL
24: end function
```

2.1.1 Примеры работы алгоритма

Рассмотрим работу fullscan алгоритма по обработке заданного SQL запроса на конкретных практических примерах.

Пример №1

```
1 query = {FROM t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t1.b > 1 AND t2.c = 5}
```

Условие соединения - *LEFT JOIN*. В условии *WHERE* исключается возможность равенства поля *t2.c* значению *NULL*, значит изменим запрос на

```
query = {FROM t1 INNER JOIN t2 ON t1.a = t2.a WHERE t1.b > 1 AND t2.c = 5}
```

Условие соединения - *INNER JOIN*. Сортировок нет, значит вернуть *NULL*.

Ответ: по какой таблице будет осуществлено полное сканирование на данном этапе не определено.

Пример №2

```
query = {FROM t1 LEFT JOIN t2  ON t1.a = t2.a WHERE t1.b = 5000 AND t1.c >
    3 ORDER BY t2.c, t2.d}
```

Условие соединения - *LEFT JOIN*. В условии *WHERE* не исключается возможность равенства любого поля таблицы *t2* значению *NULL*, значит вернуть *t1*.

Ответ: по таблице *t1* будет осуществлено полное сканирование.

Пример №3

```
query = {FROM t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t2.b = 5000 AND t2.c >
    3 ORDER BY t2.c, t2.d}
```

Условие соединения - *LEFT JOIN*. В условии *WHERE* исключается возможность равенства поля *t2.b* и *t2.c* значению *NULL*, значит изменим запрос на

```
query = {FROM t1 INNER JOIN t2 ON t1.a = t2.a WHERE t2.b = 5000 AND t2.c >
    3 ORDER BY t2.c, t2.d}
```

Условие соединения - *INNER JOIN*. Есть сортировка *ORDER BY t2.c, ...*. Вернуть *t2*.

Ответ: по таблице *t2* будет осуществлено полное сканирование.

2.2 Indexjoin алгоритм

Indexjoin алгоритм - алгоритм, для определения индексов, которые могут быть использованы СУБД для оптимизации выполнения SQL запроса. Алгоритм на вход принимает *join запрос* и возвращает множество пар индексов. В каждой паре первый индекс - индекс для первой таблицы, второй индекс - индекс для второй таблицы. Если в индексе нет полей, значит по таблице не нужно строить индекс.

Введем некоторые обозначения:

а) *index(T)* - индекс-множество для таблицы T, под которым подразумевается конечное множество индексов. Знак восклицания (!) означает фиксированное поле в индексе. Нефиксированные поля можно менять местами только между фиксированными полями, началом и концом. Например *T(a, b, c!, d, e)* $\equiv$ *t(a, b, c, d, e), t(b, a, c, d, e), t(a, b, c, e, d), t(b, a, c, e, d)*;

б) *[joinFields]* $\leftarrow$ *getJoinFields(query)* - получить поля, по которым происходит соединение таблиц по SQL запросу *query*;

в) *index(T)* ← *deleteFieldsFromIndex(index(T), [fields])* - из *index(T)* удалить поля, перечисленные в массиве *[fields]*;

г) *[(x, y)]* ← *cartesianProduct(X, Y)* - Из пар индекс-множеств *(X, Y)* через прямое произведение множеств *X, Y* получить всевозможные пары индексов *(x, y)*;

д) *index(T)* ← *createIndexSet(query)* - построить индекс-множество по SQL запросу *query*.

В качестве *indexjoin алгоритма* для *join запроса* можно предложить алгоритм, при работе которого происходит анализ исследуемого запроса. Псевдокод представлена в алгоритме 2. Блок-схема представлена на рисунке 2.2.

Примеры для участков кода *block1*

Пусть *T = t2, query = ... ORDER BY t2.z, t2.y, t1.z, t1.y, t2.x ...*

тогда *query1 = ... ORDER BY t2.z, t2.y*

Пусть *T = t2, query = ... ORDER BY t1.x, t2.x ...*

тогда отбрасываются все поля (т.е. в *query1* не будет сортировки)

Примеры для участков кода *block2*

Пусть *query1 = ... ON t1.a = t2.a WHERE t1.b = Z, t2.b = Y ...*

тогда *query21 = ... ON t1.a = const1 WHERE t1.b = Z ...*

Примеры для участков кода *block3*

Пусть *T = t1, index(t1) = t1(a, b), index(t2) = t2(c, a), query0 = ... ON t1.a = t2.a ...*

тогда *index(t1) = t1(b), index(t2) = t2(c, a)*

Примеры для участков кода *block4*

Пусть *index(t1) = t1(a, b, c), index(t2) = t2(a, b, c), query1 = ... ON t1.a = t2.a ...*

тогда *index1(t1) = t1(a, b, c), index2(t2) = t2(b, c), index2(t1) = t1(b, c), index1(t2) = t2(a, b, c)*

2.2.1 Примеры работы алгоритма

Рассмотрим работу indexjoin алгоритма по обработке заданного SQL запроса на конкретных практических примерах.

Пример №1

```
1 query := t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t1.b > 1 AND t2.c = 5
2
3 T := NULL
4
5 query1 := query
6
7 query21 := ON t1.a = const WHERE t1.b > 1
```

```
8  query22 := ON t2.a = const WHERE t2.c = 5
9
10 index(t1) :=t1(a ,b!)
11 index(t2) := t2(a, c)
12
13 index1(t1) :=t1(a ,b!)
14 index2(t2) :=t2(c)
15
16 index1(t2) := t2(a, c)
17 index2(t1) :=t1(b)
```

(t1(a, b!), t2(c)) ≡ (t1(a, b), t2(c))

(t1(b), t2(a, c)) ≡ (t1(b), t2(a, c)), (t1(b), t2(c, a))

Ответ: *(t1(a, b), t2(c)), (t1(b), t2(a, c)), (t1(b), t2(c, a))* - множество пар индексов, наиболее подходящих данному запросу.

Пример №2

```
1  query := FROM t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t1.b = 5000 AND t1.c >
       3 ORDER BY t2.c, t2.d
2
3  T := t1
4
5  query1 := FROM t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t1.b = 5000 AND t1.c >
       3
6
7  query21 := ON t1.a = const WHERE t1.b = 5000 AND t1.c > 3
8  query22 := ON t2.a = const
9
10 index(t1) :=t1(a ,b ,c!)
11 index(t2) := t2(a)
12
13 index(t1) :=t1(b ,c!)
```

(t1(b, c!), t2(a)) ≡ (t1(b, c), t2(a))

Ответ: *(t1(b, c), t2(a))* - множество пар индексов, наиболее подходящих данному запросу.

Пример №3

```
1  query := FROM t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t2.b = 5000 AND t2.c >
       3 ORDER BY t2.c, t2.d
2
3  T := t2
4
5  query1 := query
6
7  query21 := ON t1.a = const
```

```
8  query22 := ON t2.a = const WHERE t2.b = 5000 AND t2.c > 3 ORDER BY t2.c,
       t2.d
9
10 index(t1) := t1(a)
11 index(t2) := t2(a, b, c!, d!)
12
13 index(t2) := t2(b, c!, d!)
```

(t1(a), t2(b, c!, d!)) ≡ (t1(a), t2(b, c, d))

Ответ: *(t1(a), t2(b, c, d))* - множество пар индексов, наиболее подходящих данному запросу.

Пример №4

```
1  query := FROM t1 LEFT JOIN t2 ON t2.a = t1.a ORDER BY t2.b
2
3  T := t1
4
5  query1 := t1 LEFT JOIN t2 ON t2.a = t1.a
6
7  query21 := ON t1.a = const
8  query22 := ON t2.a = const
9
10 index(t1) :=t1(a)
11 index(t2) := t2(a)
12
13 index(t1) :=t1()
```

(t1(), t2(a)) ≡ (t1(), t2(a))

Ответ: *(t1(), t2(a))* - множество пар индексов, наиболее подходящих данному запросу.

Пример №5

```
1  query :=
2      t1 INNER JOIN t2
3          ON t1.a = t2.a
4      WHERE t1.b = const
5      ORDER BY t1.c
```

Ответ: *(t1(b, c), t2(a))* - множество пар индексов, наиболее подходящих данному запросу.

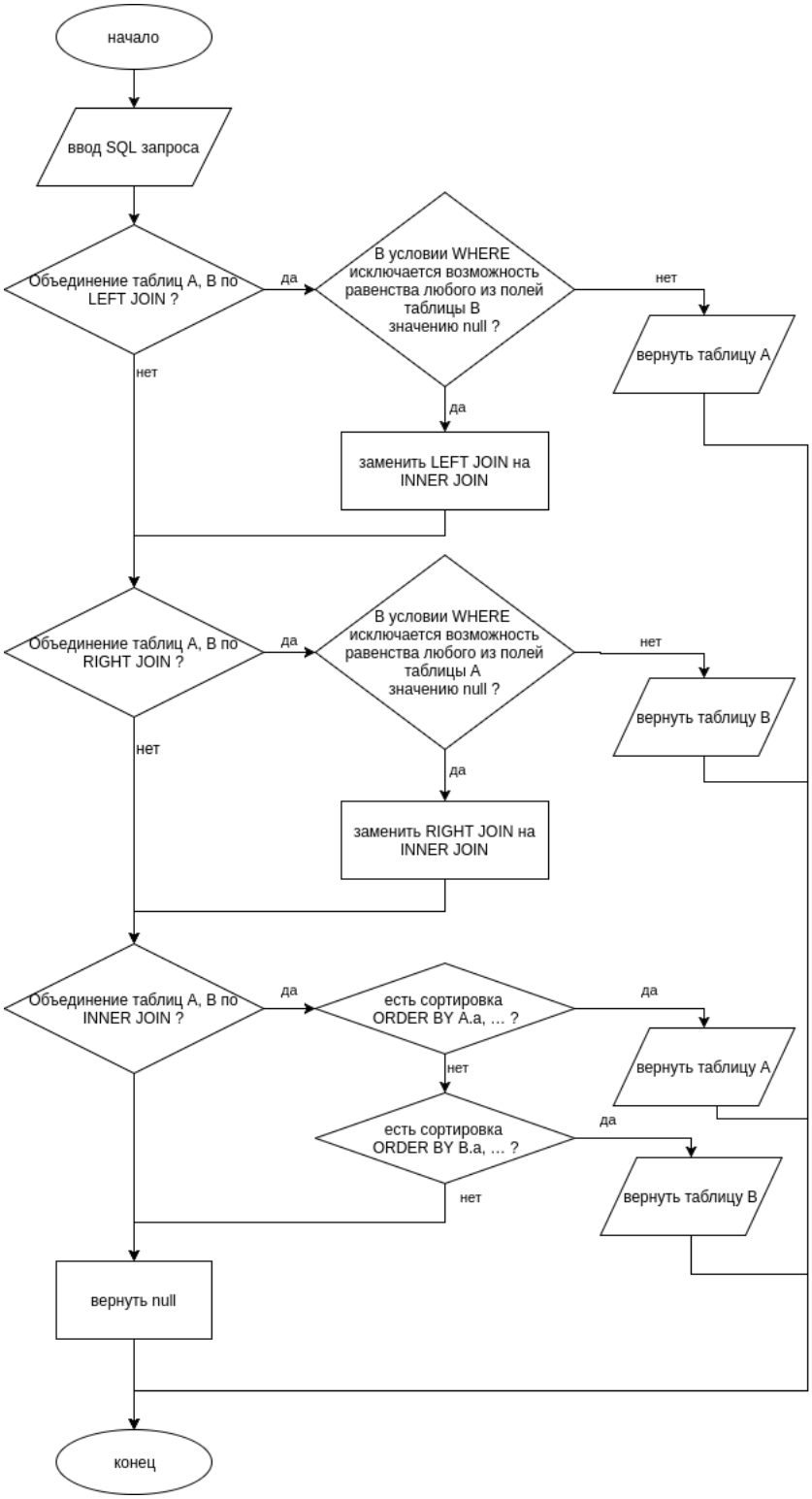

Рисунок 2.1 — Блок-схема fullscan-алгоритма.

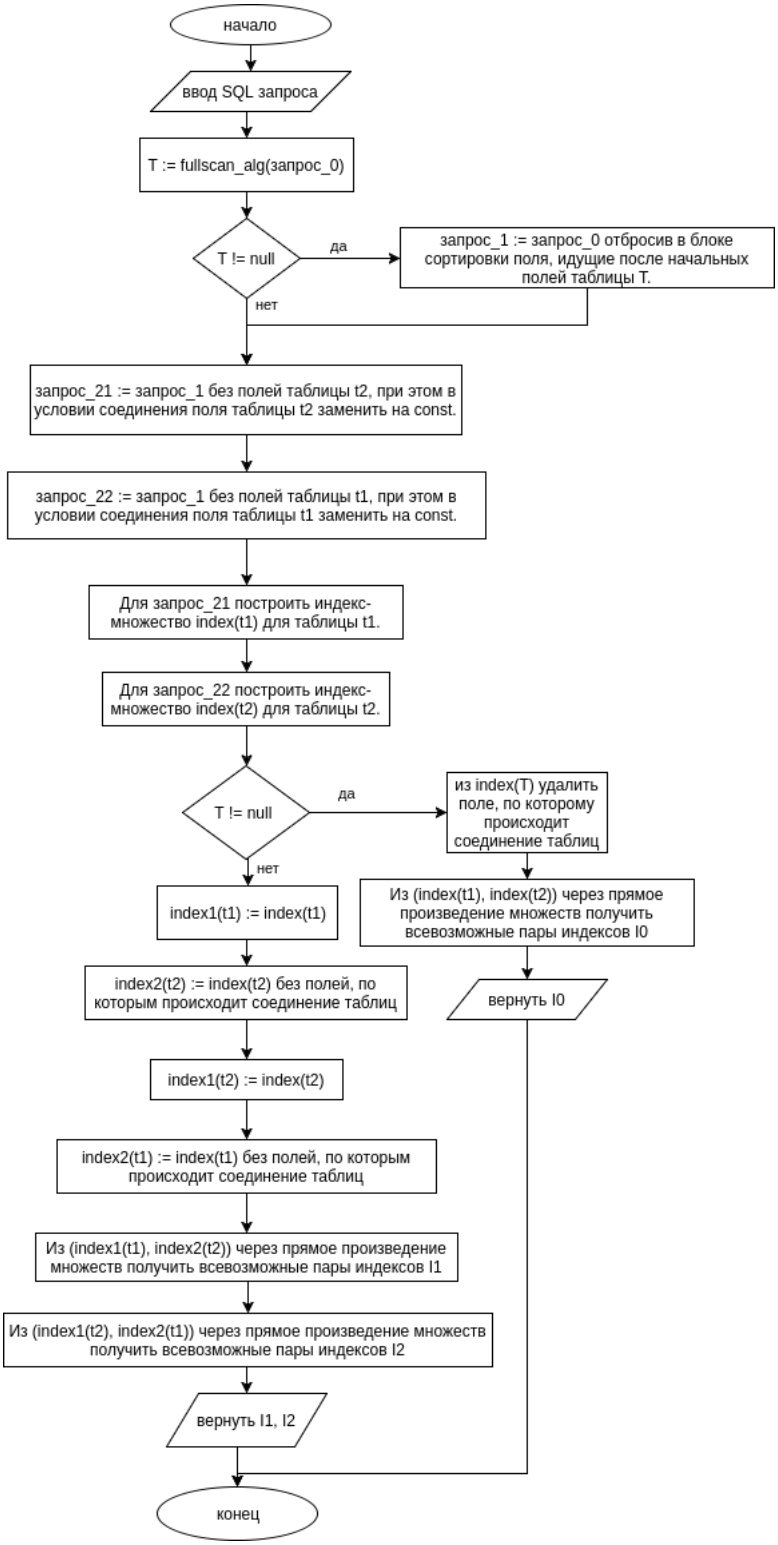

Рисунок 2.2 — Блок-схема indexjoin алгоритма.

Algorithm 2 Indexjoin алгоритм

```
1: function INDEXJOINALG(query)
2:     T ← FULLSCANALG(query)
                                                    ▷ —begin block1—
3:     if T ≠ NULL then
4:         query1 ← query (отбросив в блоке сортировки поля, идущие после начальных полей таблицы T)
5:     else
6:         query1 ← query
7:     end if                                       ▷ —end block1—
                                                    ▷ —begin block2—
8:     query21 ← query1 (без полей таблицы t2, при этом в условии соединения поля таблицы t2 заменить на CONST)

9:     query22 ← query1 (без полей таблицы t1, при этом в условии соединения поля таблицы t1 заменить на CONST   ▷ —end block2—

10:    index(t1) ← createIndexSet(query21)
11:    index(t2) ← createIndexSet(query22)
12:    joinFields ← getJoinFields(query)

13:    if T ≠ NULL then
                                                    ▷ —begin block3—
14:        index(T) ← deleteFieldsFromIndex(index(T), joinFields)

15:        result ← cartesianProduct(index(t1), index(t2))
                                                    ▷ —end block3—
16:    else
                                                    ▷ —begin block4—
17:        index1(t1) ← index(t1)
18:        index2(t2) ← deleteFieldsFromIndex(index(t2), joinFields)

19:        index1(t2) ← index(t2)
20:        index2(t1) ← deleteFieldsFromIndex(index(t1), joinFields)

21:        result ← cartesianProduct(index1(t1), index2(t2)), cartesianProduct(index1(t2), index2(t1))
                                                    ▷ —end block4—
22:    end if

23:    return result
24: end function
```

3 Технологический раздел

Заключение

В данной работе предлагается алгоритм определения необходимых индексов по заданному запросу с соединением двух таблиц. Данный алгоритм облегчает работу администраторов баз данных высоконагруженных приложений.

Список использованных источников

1. *Шварц Б. Зайцев П., Ткаченко В.* MySQL. Оптимизация производительности. / Ткаченко В. Шварц Б., Зайцев П. — Символ-Плюс, 2010.

2. *Web-приложений, Оптимизация и масштабирование.* Индексы в MySQL. — http://ruhighload.com/post/\T2A\CYRR\T2A\cyra\T2A\cyrb\T2A\cyro\T2A\cyrt\T2A\cyra+\T2A\cyrs+\T2A\cyri\T2A\cyrn\T2A\cyrd\T2A\cyre\T2A\cyrk\T2A\cyrs\T2A\cyra\T2A\cyrm\T2A\cyri+\T2A\cyrv+MySQL. — [Online; accessed 31-01-2017].

3. How MySQL Uses Indexes. — https://dev.mysql.com/doc/refman/5.7/en/mysql-indexes.html. — [Online; accessed 18-05-2017].