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 «Программное обеспечение ЭВМ
и информационные технологии»

АЛГОРИТМ ОПРЕДЕЛЕНИЯ НЕОБХОДИМЫХ ИНДЕКСОВ ДЛЯ ОПТИМИЗАЦИИ ЗАПРОСОВ С СОЕДИНЕНИЕМ ДВУХ ТАБЛИЦ В СУБД MYSQL (INNODB)

Исполнитель: студент ИУ7-81 Петухов И. С. _____

Руководитель: преподаватель ИУ7 Просуков Е. А. _____

Москва, 2017

Содержание

Введение

В работе рассматривается решение прикладной задачи, возникающей при работе с базами данных, которые уже спроектированы, используются в эксплуатации и имеют таблицы с несколькими тысячами записей. Проблема оптимизации работы базы данных решается с помощью индексирования.

Рассматривается структура хранения данных в СУБД MySQL (InnoDB). Описываются кластерные и вторичные индексы. Правила применения индексов для запросов по одной таблице.

Для SQL запроса, имеющего операторы JOIN (INNER, LEFT, RIGHT), WHERE, ORDER BY предлагается два алгоритма. Первый алгоритм определяет, по какой из двух таблиц оператора JOIN будет осуществлено полное сканирование. Второй алгоритм показывает, какие индексы необходимо построить для оптимизации выполнения этого запроса. Оба алгоритма могут иметь программную реализацию. На примерах показывается использование этих алгоритмов.

1 Аналитический раздел

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 (при необходимости);
в) выявление медленных запросов, их анализ;
г) корректировка запросов;
д) создание индексов.

Основным приемом увеличения производительности выполнения запросов к базе данных является индексирование. Индексы представляют собой структуры, которые помогают MySQL 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]

Индексы увеличивают производительность путем оптимизации обращений к дисковой памяти.

Индексы следует создавать по мере обнаружения медленных запросов. В этом поможет *slow log* в MySQL. 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.

Задачу выбора индекса для конкретного sql запроса на БД, заполненной реальными данными, решают СУБД. На текущий момент даже сами БД не справляются с этой задачей. Но если абстрагироваться от реальных данных, которыми заполнена СУБД и статистики запросов, то зная правила работы СУБД с индексами можно сделать предположения, какие индексы могла бы использовать СУБД. В данной работе предлагаются алгоритмы, которые по заданному sql запросу вернут список индексов, которые могли бы использоваться СУБД при выполнении этого запроса. Дальше, администратор БД примет решение, какой именно индекс построить из предложенного списка.

1.1 Структура индексов в MySQL InnoDB

Индексы представляют собой структуры, которые помогают эффективно извлекать данные. В СУБД MySQL InnoDB используются Btree индексы, основанные на структуре данных B+tree Рисунок 1.1.

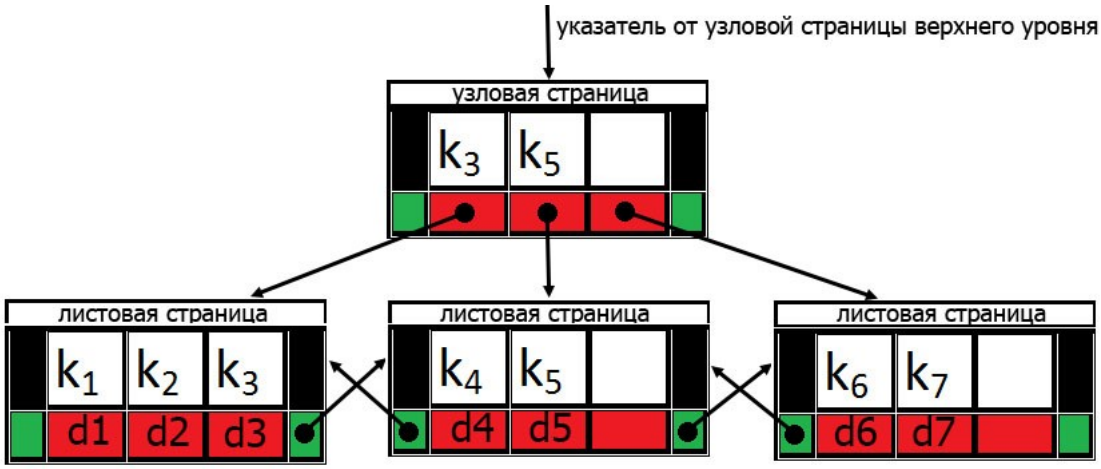

Рисунок 1.1 — Структура данных B+tree.

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

Сложность поиска в линейной структуре данных $O(N)$, а в B+tree - $log(K) + N_k$, 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.

Далее в качестве примера будет рассматриваться таблица $poet(poet\_id, last\_name, first\_name, dob, country)$

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 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 1.1.5.

По рисуноку 1.1, где $k_1 \dots k_7$ - первичные ключи ($poet\_id$), $k_i = i$; $d_1 \dots d_7$ - данные.

d_1 = “Блок“, “Александр“, 1880, “ru”;

d_2 = “Фет“, “Афанасий“, 1820, “ru”;

...

Если вы не задаёте первичный ключ, то MySQL неявно добавит скрытое поле и кластерезует данные по нему, но доступа к нему не даст. Отсюда следует, что первичный ключ желательно явно создавать самим.

Характерной проблемой кластерного индекса является то, что при вставке в 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. 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 ($poet\_id$); $p_i = i$.

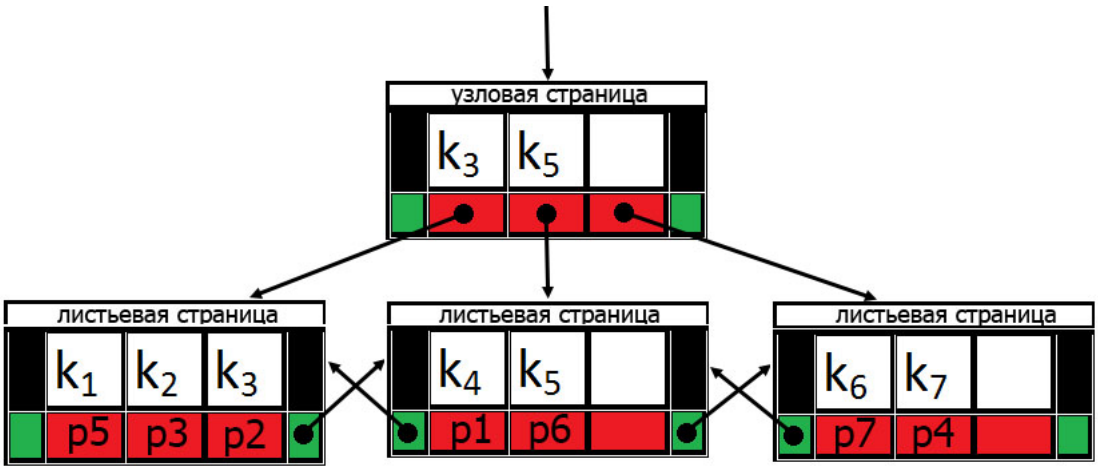

Рисунок 1.2 — Структура вторичного индекса

1.1.3 Составной индекс

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. Составной индекс может использоваться частично, но только по самому левому префиксу.

Если построить индекс по полям $(dob, last\_name)$, то для рисунка 1.2 k_1 = “1799Пушкин”, k_2 = “1814Лермонтов”, k_3 = “1820Фет”, . . .

Составной индекс может использоваться как **частичный индекс**, являющийся левым префиксом составного индекса.

1.1.4 Селективность индексов

Чем меньше строк войдет в выборку, тем быстрее будет работать поиск по ней. Если рассматривать таблицы с равномерно распределенными значениями в полях, то условие = более селективно, чем условия >, <=, >=, <.

1.1.5 Покрывающий индекс

Индекс называется покрывающим, если в нем есть все поля, используемые в запросе. Покрывающие индексы позволяют имитировать кластерные индексы. Индексы, как правило, находятся в памяти, и не потребуется долгих операций ввода/-вывода с диска.

Для запроса на листинге 1.1 строим индекс $(last\_name, dob)$.

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

Чтобы построить индексы для простых запросов (т.е. без соединения таблиц) необходимо воспользоваться простыми правилами.

1.2.1 Индексы для WHERE

Для запроса на листинге 1.2 строим индекс $(dob, last\_name)$ и если не учитывать селективность, то порядок полей не важен.

Листинг 1.2 — запрос для index-on-where

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

Index-where правила

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

Примеры работы индекса (a, b, c):

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

б) $a = 5$ (т.к. a - левый префикс индекса)

в) $a > 5$ (т.к. a - левый префикс индекса)

г) $a = 5\ AND\ b = 10$ (т.к. a, b - левый префикс индекса)

д) $a = 5\ AND\ b = 10\ AND\ c\ LIKE\ "Hello\ w\%"$ (т.к. $LIKE$ использует левый префикс столбца)

е) $a = 5\ AND\ b\ IN$ (2,3) (т.к. IN - рассматривается как поиск по диапазону)

Примеры, когда индекс (a, b, c) не работает:

а) $b = 5$ (т.к. b - не левый префикс индекса)

б) $a = 5\ AND\ b = 10\ AND\ c\ LIKE\ "\%world"$ (т.к. $LIKE$ использует не левый префикс столбца)

в) $a = 5\ AND\ c = 10$ (т.к. пропущен столбец b)

Но с помощью индекса (a, b, c), MySQL сделает выборку по условию $a = 5$, а дальше будут просмотрены все строки из этой выборки. Это будет частичным индексом.

г) $a > 5\ AND\ b = 2$ (т.к. по столбцу a используется условие поиска по диапазону, то индекс сработает только частично)

1.2.2 Индексы при сортировке

Чтобы получить отсортированную последовательность данных, MySQL достаточно будет пройтись по дереву, т.к. в индексе данные хранятся в отсортированном виде.

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

```
1 SELECT *
2 FROM poet
3 WHERE country="ru"
4 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

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

Index-order правила

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

Примеры работы индекса (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, когда индекс (a, b) не работает:

а) $ORDER\ BY\ b$ (т.к. b - не левый префикс индекса)
б) $WHEREa > 5\ ORDER\ BY\ B$

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) $WHERE a\,IN\,(1,2)\;ORDER\,BY\,b$ (тоже самое, что и в предыдущем запросе)

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

```
1 SELECT MAX(dob)
2 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

Построение индексов для запроса с соединением таблиц часто вызывает сложности, так как требует от разработчика знания тонкостей работы СУБД при соединении таблиц, и тонкостей работы с индексами. Раньше не предлагалось общего алгоритма построения индексов на запросы с соединением таблиц. В данной работе предлагается такой алгоритм, который может иметь программную реализацию.

2 Конструкторский раздел

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

Листинг 2.1 — Вид sql запроса

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

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.

2.1.1 Примеры работы алгоритма

Рассмотрим работу fullscan 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}
```

Условие соединения - $LEFTJOIN$. В условии $WHERE$ исключается возможность равенства поля $t2.c$ значению $null$, значит изменим запрос на

```
1 query = {t1  INNERJOIN 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

```
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}
```

Условие соединения - $LEFTJOIN$. 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 - $LEFTJOIN$. 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 $ORDERBYt2.c, \ldots$. 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. Лучшая из множества пар индексов выбирается по селективности, исходя из данных, которыми заполнены таблицы БД.

Введем некоторые обозначения:

а) $index(t)$ - индекс-множество для таблицы t, под которым подразумевается конечное множество индексов. Например, если $index(t) = t(a,\ b,\ c',\ d,\ e)$, то $index(t) = t(a,b,c,d,e), t(b,a,c,d,e), t(a,b,c,e,d), t(b,a,c,e,d)$. Апостроф означает фиксированное поле в индексе. Нефиксированные поля можно менять местами только между фиксированными полями, началом и концом;

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

в) $index(T) \leftarrow deleteFieldsFromIndex(index(T), [fields])$ - из $index(T)$ удалить поля, перечисленные в массиве $fields$;

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

д) $index(T) \leftarrow createIndexSet(query)$ - 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 indexjoinAlg

$block1$: пусть $T = t2$, $query = \{\dots ORDERBYt2.z, t2.y, t1.z, t1.y, t2.x \dots\}$, тогда $query1 = \{\dots\ ORDER\ BY\ t2.z,\ t2.y\ \dots\}$

$block1$: пусть $T = t2$, $query = \{\dots ORDER\ BY\ t1.x,\ t2.x\ \dots\}$, тогда отбрасываются все поля (т.е. в $query1$ не будет сортировки)

$block2$: пусть $query1 = \{\dots ONt1.a = t2.aWHEREt1.b = Z, t2.b = Y \dots\}$, тогда $query21 = \{\dots\ ONt1.a = const1WHEREt1.b = Z\ \dots\}$

$block3$: пусть $T = t1$, $index(t1) = t1(a, b)$, $index(t2) = t2(c, a)$, $query0 = \{\dots ON\ t1.a = t2.a \dots\}$, тогда $index(t1) = t1(b)$, $index(t2) = t2(c, a)$

$block4$: пусть $index(t1) = t1(a, b, c)$, $index(t2) = t2(a, b, c)$, $query1 = \{\dots ONt1.a = t2.a \dots\}$, 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}
```

$$T \leftarrow null$$

$$query1 \leftarrow query$$

$$query21 \leftarrow ONt1.a = constWHEREt1.b > 1$$

$$query22 \leftarrow ONt2.a = constWHEREt2.c = 5$$

$$index(t1) \leftarrow t1(a,\ b')$$

$$index(t2) \leftarrow t2(a, c)$$

$$index1(t1) \leftarrow t1(a,\ b')$$

$$index2(t2) \leftarrow t2(c)$$

$$index1(t1) \leftarrow t1(b)$$

$$index2(t2) \leftarrow t2(a, c)$$

$$(t1(a,\ b'),\ t2(c)) \equiv (t1(a,\ b),\ t2(c))$$

$$(t1(b),\ t2(a,\ c)) \equiv (t1(b),\ t2(a,\ c)),\ (t1(b),\ t2(c,\ a))$$

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

```
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}
```

$$T \leftarrow t1$$

$$query1 \leftarrow FROMt1LEFTJOINt2ONt1.a = t2.aWHEREt1.b = 5000ANDt1.c > 3$$

$$query21 \leftarrow ONt1.a = constWHEREt1.b = 5000ANDt1.c > 3$$

$$query22 \leftarrow ONt2.a = const$$

$$index(t1) \leftarrow t1(a,\ b,\ c')$$

$$index(t2) \leftarrow t2(a)$$

$$index(t1) \leftarrow t1(b,\ c')$$

$$index(t2) \leftarrow t2(a)$$

$$(t1(b,\ c'),\ t2(a)) \equiv (t1(b,\ c),\ t2(a))$$

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

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}
```

$$T \leftarrow t2$$

$$query1 \leftarrow query$$

$$query21 \leftarrow ONt1.a = const$$

$$query22 \leftarrow ONt2.a = constWHEREt2.b = 5000ANDt2.c > 3ORDERBYt2.c, t2.d$$

$$index(t1) \leftarrow t1(a)$$

$$index(t2) \leftarrow t2(a,b,c',d')$$

$$index(t1) \leftarrow t1(a)$$

$$index(t2) \leftarrow t2(b,c',d')$$

$$(t1(a),\ t2(b,\ c',\ d')) \equiv (t1(a),\ t2(b,\ c,\ d))$$

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

Пример №4

```
query = {FROM t1 LEFT JOIN t2 ON t2.a = t1.a ORDER BY t2.b}
```

$$T \leftarrow t1$$

$$query1 \leftarrow t1 LEFT JOIN t2 ON t2.a = t1.a$$

$$query21 \leftarrow ON t1.a = const$$

$$query22 \leftarrow ON t2.a = const$$

$$index(t1) \leftarrow t1(a)$$

$$index(t2) \leftarrow t2(a)$$

$$index(t1) \leftarrow t1()$$

$$index(t2) \leftarrow t2(a)$$

$$(t1(), t2(a)) \equiv (t1(), t2(a))$$

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

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 ORDERBY A.a, ... then
18:            return таблица A
19:        else if есть сортировка ORDERBY B.a, ... then
20:            return таблица B
21:        end if
22:    end if

23:    return null
24: end function
```

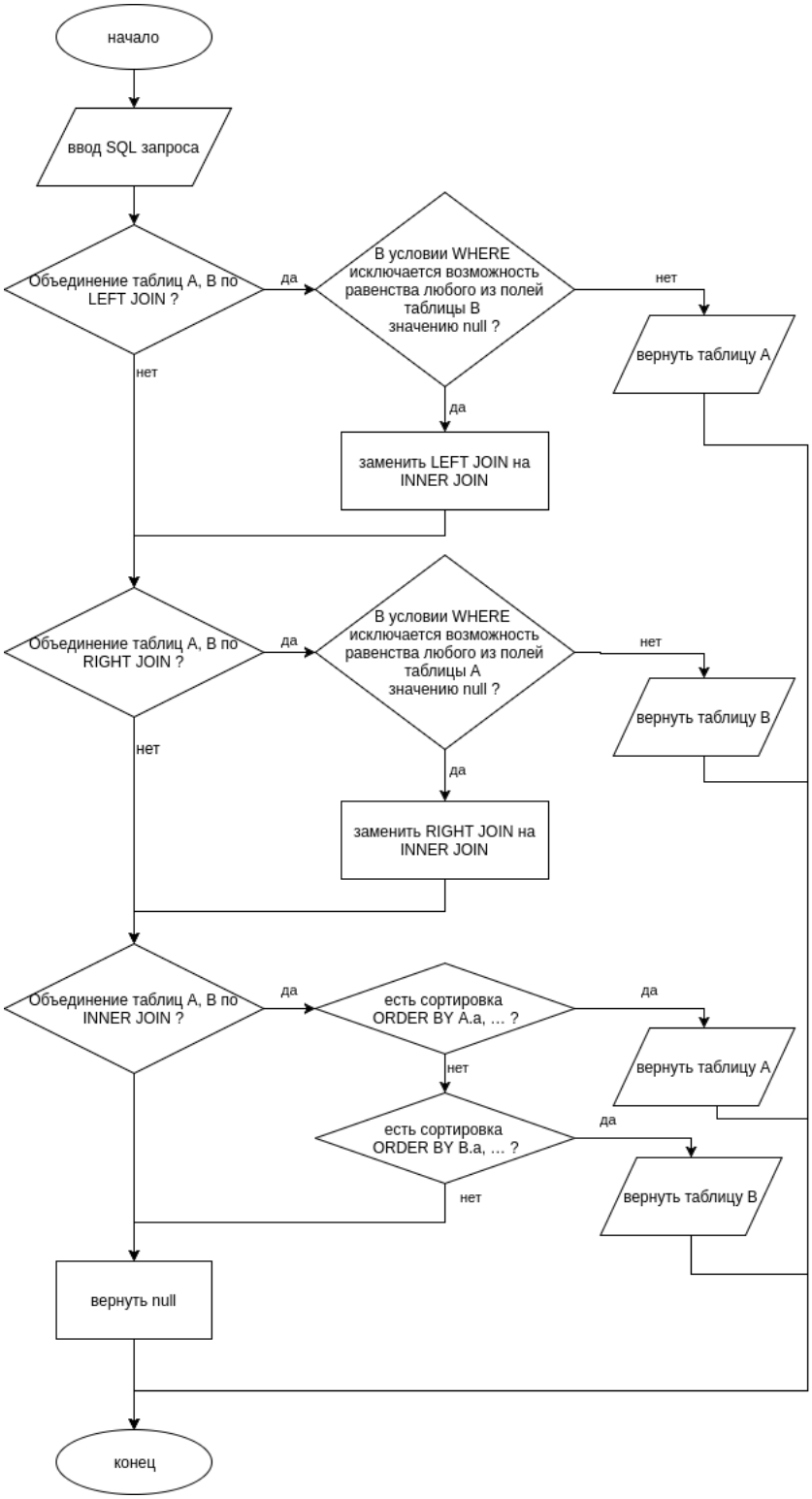

Рисунок 2.1 — Блок-схема fullscan-алгоритма.

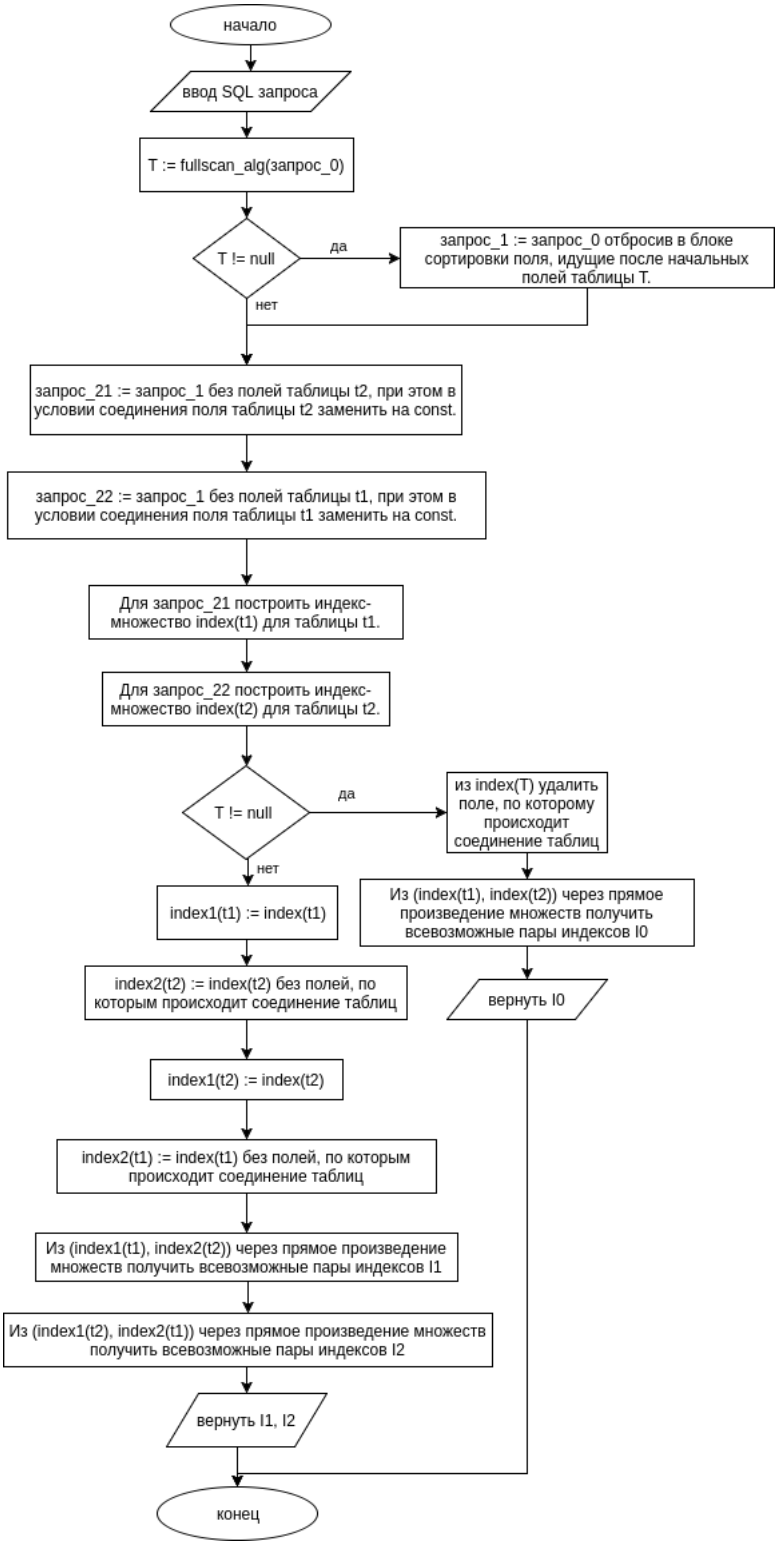

Рисунок 2.2 — Блок-схема indexjoin алгоритма.

Algorithm 2 Indexjoin алгоритм

1: **function** INDEXJOINALG(query)
2: $T \leftarrow$ FULLSCANALG($query$)
▷ —begin block1—
3: **if** $T \neq null$ **then**
4: $query1 \leftarrow query$ отбросив в блоке сортировки поля, идущие после начальных полей таблицы T
5: **else**
6: $query1 \leftarrow query$
7: **end if** ▷ —end block1—
▷ —begin block2—
8: $query21 \leftarrow query1$ без полей таблицы $t2$, при этом в условии соединения поля таблицы $t2$ заменить на *const*
9: $query22 \leftarrow query1$ без полей таблицы $t1$, при этом в условии соединения поля таблицы $t1$ заменить на *const* ▷ —end block2—
10: $index(t1) \leftarrow createIndexSet(query21)$
11: $index(t2) \leftarrow createIndexSet(query22)$
12: $joinFields \leftarrow getJoinFields(query)$
▷ —begin block3—
13: **if** $T \neq null$ **then**
14: $index(T) \leftarrow deleteFieldsFromIndex(index(T), joinFields)$
15: **return** $cartesianProduct(index(t1), index(t2))$
16: **end if** ▷ —end block3—
▷ —begin block4—
17: $index1(t1) \leftarrow index(t1)$
18: $index2(t2) \leftarrow deleteFieldsFromIndex(index(t2), joinFields)$
19: $index1(t2) \leftarrow index(t2)$
20: $index2(t1) \leftarrow deleteFieldsFromIndex(index(t1), joinFields)$
▷ —end block4—
21: **return** $cartesianProduct(index1(t1), index2(t2)), cartesianProduct(index1(t2), index2(t1))$
22: **end function**

3 Технологический раздел

Заключение

В данной работе предлагается алгоритм определения необходимых индексов по заданному запросу с соединением двух таблиц. Данный алгоритм позволяет решить задачу, возникающую в высоконагруженных приложениях, использующих реляционную базу данных MySQL (InnoDB). Для представленных алгоритмов получена программная реализация.

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].