

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 . . . . .	4
1 Аналитический раздел . . . . .	5
1.1 Структура индексов в MySQL InnoDB . . . . .	5
1.1.1 Индексы для простых запросов . . . . .	6
1.1.2 Индексы для сложных запросов . . . . .	6
2 Конструкторский раздел . . . . .	8
2.1 Fullscan алгоритм . . . . .	8
2.1.1 Примеры работы алгоритма . . . . .	8
2.2 Indexjoin алгоритм . . . . .	9
2.2.1 Примеры для участков кода indexjoinAlg . . . . .	10
2.2.2 Примеры работы алгоритма . . . . .	10
3 Технологический раздел . . . . .	17
Заключение . . . . .	18
Список использованных источников . . . . .	19

## 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. 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 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, 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

В СУБД MySQL (InnoDB) используются Btree индексы, основанные на структуре данных B+tree Рисунок 1.1.

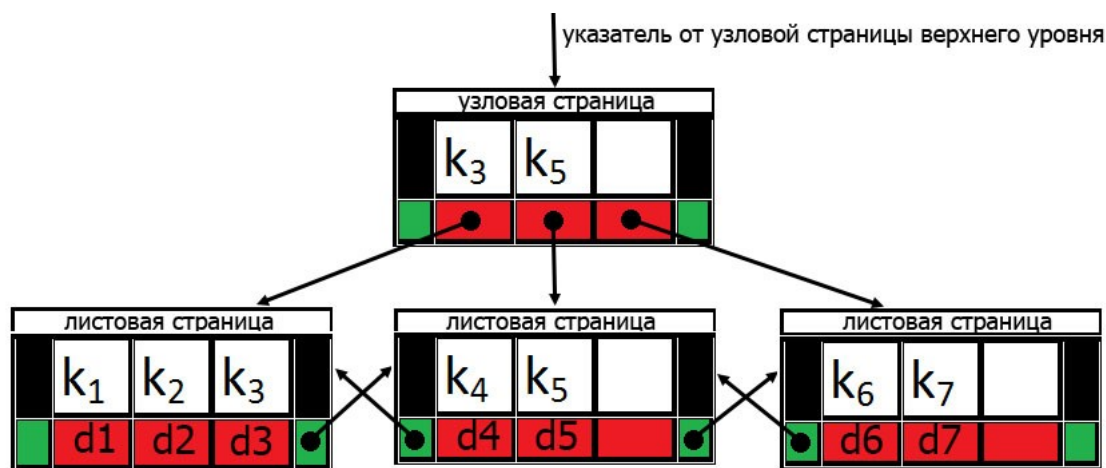


Рисунок 1.1 — Структура данных B+tree.

Где  $k_i$  - ключи ( $k_i < k_{i+1}$ ),  $d_i$  - данные.

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

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.

### 1.1.1 Индексы для простых запросов

Чтобы построить индексы для запросов без соединения таблиц, необходимо воспользоваться простыми правилами.

### 1.1.2 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 обще-

го 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 (А и В), для каждой строки одной таблицы А будет происходить сканирование другой таблицы В. Такое сканирование таблицы А 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 - *LEFT JOIN*. 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 - *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

```
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}
```

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

```
1 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 *ORDERBY t2.c, ...*. Вернуть *t2*.

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

## 2.2 Indexjoin алгоритм

**Indexjoin алгоритм** - алгоритм, для определения индексов, которые нужно построить для оптимизации выполнения join-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.

### 2.2.1 Примеры для участков кода `indexjoinAlg`

*block1*: пусть  $T = t2$ ,  $query = \{\dots ORDERBY t2.z, t2.y, t1.z, t1.y, t2.x \dots\}$ , 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  $query1 = \{\dots ON t1.a = t2.a WHERE t1.b = Z, t2.b = Y \dots\}$ , тогда  $query21 = \{\dots ON t1.a = const1 WHERE t1.b = Z \dots\}$

*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  $index(t1) = t1(a, b, c)$ ,  $index(t2) = t2(a, b, c)$ ,  $query1 = \{\dots ON t1.a = t2.a \dots\}$ , тогда  $index1(t1) = t1(a, b, c)$ ,  $index2(t2) = t2(b, c)$ ,  $index2(t1) = t1(b, c)$ ,  $index1(t2) = t2(a, b, c)$

### 2.2.2 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}`

$T \leftarrow null$

$query1 \leftarrow query$

$query21 \leftarrow ON t1.a = const WHERE t1.b > 1$

$query22 \leftarrow ON t2.a = const WHERE t2.c = 5$

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

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

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

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

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

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

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

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

**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

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}

$$T \leftarrow t1$$

$$query1 \leftarrow FROM t1 LEFT JOIN t2 ON t1.a = t2.a WHERE t1.b = 5000 AND t1.c > 3$$

$$query21 \leftarrow ON t1.a = const WHERE t1.b = 5000 AND t1.c > 3$$

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

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

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

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

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

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

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

### Пример №3

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}

$$T \leftarrow t2$$

$$query1 \leftarrow query$$

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

$$query22 \leftarrow ON t2.a = const WHERE t2.b = 5000 AND t2.c > 3 ORDER BY t2.c, t2.d$$

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

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

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

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

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

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

### Пример №4

1 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  $A \text{ LEFT JOIN } B$  then
3:     if в условии WHERE исключается возможность равенства любого из полей
        таблицы  $B$  значению null then
4:       заменить  $\text{LEFT JOIN}$  на  $\text{INNER JOIN}$ 
5:     else
6:       return таблица  $A$ 
7:     end if
8:   end if

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

16:  if условие соединения  $A \text{ INNER JOIN } B$  then
17:    if есть сортировка  $\text{ORDERBY } A.a, \dots$  then
18:      return таблица  $A$ 
19:    else if есть сортировка  $\text{ORDERBY } B.a, \dots$  then
20:      return таблица  $B$ 
21:    end if
22:  end if

23:  return null
24: end function
```

---

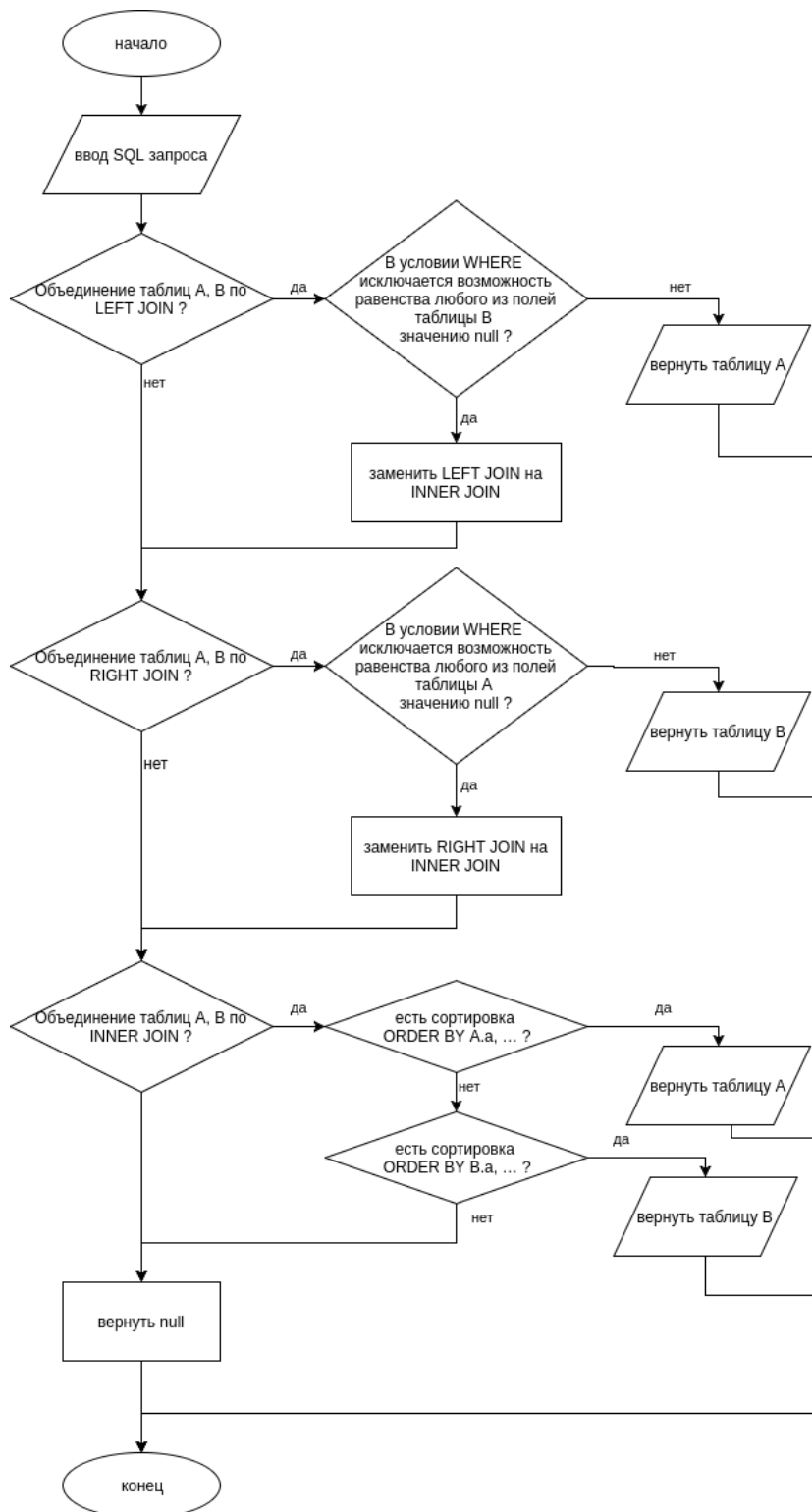


Рисунок 2.1 — Блок-схема fullscan-алгоритма.

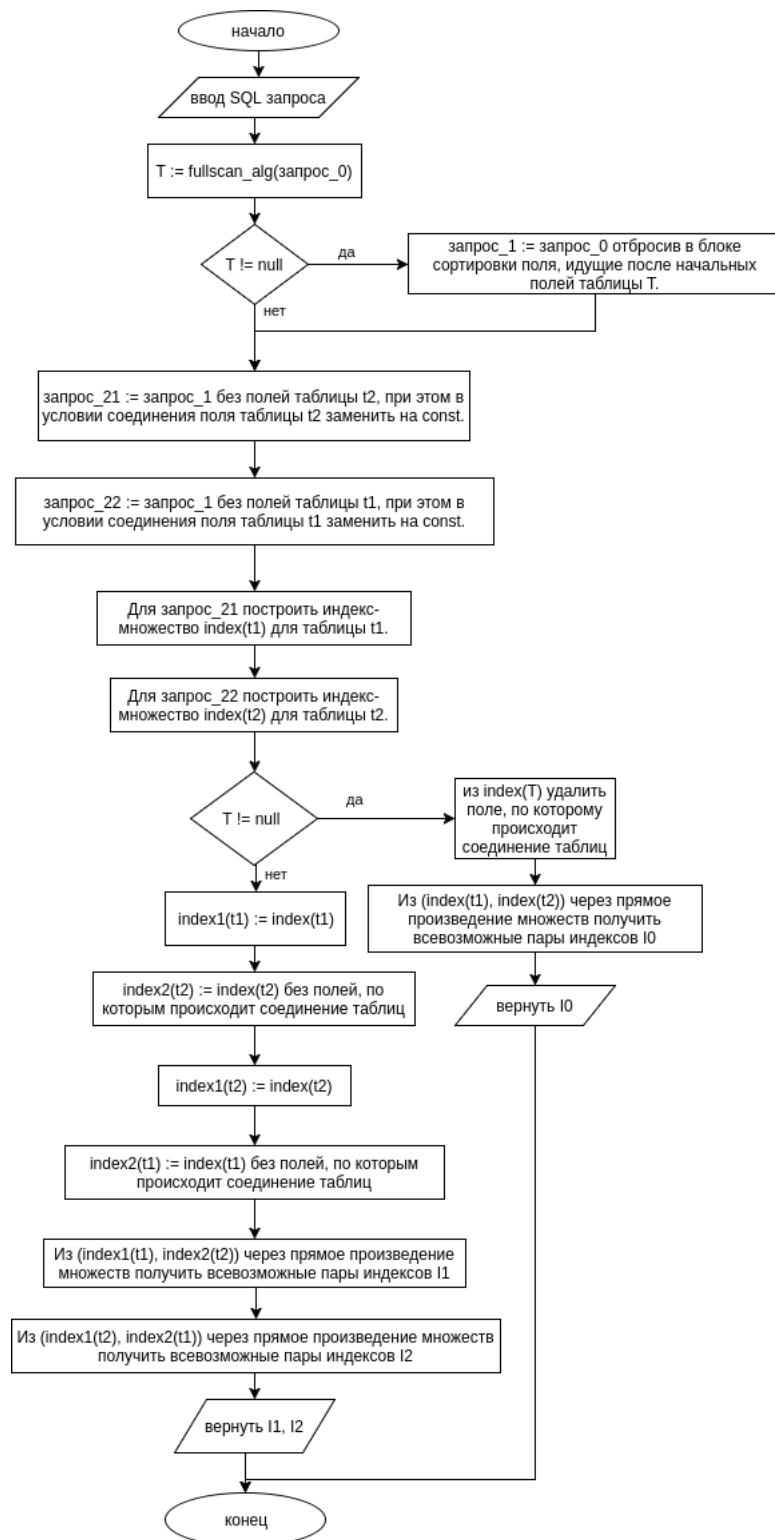


Рисунок 2.2 — Блок-схема indexjoin алгоритма.

---

**Algorithm 2** Indexjoin алгоритм

---

```
1: function INDEXJOINALG(query)
2:    $T \leftarrow \text{FULLSCANALG}(query)$ 
3:   if  $T \neq null$  then
4:      $query1 \leftarrow query$  отбросив в блоке сортировки поля, идущие после началь-
       ных полей таблицы  $T$ 
5:   else
6:      $query1 \leftarrow query$ 
7:   end if
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  $const$ 
10:   $index(t1) \leftarrow createIndexSet(query21)$ 
11:   $index(t2) \leftarrow createIndexSet(query22)$ 
12:   $joinFields \leftarrow getJoinFields(query)$ 
13:  if  $T \neq null$  then
14:     $index(T) \leftarrow deleteFieldsFromIndex(index(T), joinFields)$ 
15:  return  $cartesianProduct(index(t1), index(t2))$ 
16:  end if
17:   $index1(t1) \leftarrow index(t1)$ 
18:   $index2(t2) \leftarrow deleteFieldsFromIndex(index(t2), joinFields)$ 
19:   $index1(t2) \leftarrow index(t2)$ 
20:   $index2(t1) \leftarrow deleteFieldsFromIndex(index(t1), joinFields)$ 
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\cyrn\T2A\cyri+\T2A\cyrv+MySQL>. — [Online; accessed 31-01-2017].