MySQL

Индексы

Введение

Есть множество способов, которые позволяют ускорить поиск по базе данных - разделение больших таблиц на несколько маленьких, попытки удержать данные в оперативной памяти компьютера, а также использование нескольких копий базы данных на разных компьютерах - т.е. если на один из компьютеров будут слишком сильная нагрузка, но запросы будут перенаправлены на другой. Однако основным инструментом, который позволяет оптимизировать запросы, является т.н. **индекс**.

Индексы представляют собой структуры данных, которые движки (InnoDB, MyISAM и т. д.) используют для быстрого нахождения строк (рядов таблицы). Самый простой способ понять, как работает индекс в MySQL, - представить себе алфавитный указатель в книге. Чтобы определить, где в книге обсуждается конкретная тема, вы в алфавитном указателе находите термин и номер страницы, на которой он упоминается. В MySQL индексы строятся не для всей таблицы, а для какого-то конкретного столбца или нескольких столбцов одновременно. Однако не следует думать, что для оптимизации нам надо создавать индексы для всех столбцов. Как мы еще убедимся, что это может не только не оказать вообще никакого положительного эффекта, но и привести к замедлению обновления и вставки данных.

Создание и удаление индексов

Автоматическое создание индексов

Когда мы рассматривали ограничения - первичного ключа (**PRIMARY KEY**), по уникальности (**UNIQUE**) и внешнего ключа (**FOREIGN KEY**), то мы уже упоминали, что для первичного и уникального столбца индекс создается автоматически, а для изначального столбца, на который ссылается столбец внешнего ключа, мы должны поставить индекс явно, иначе внешний ключ не будет установлен, а запрос вернет ошибку. Во всех других случаях никакой автоматической помощи и строгого контроля ожидать не стоит - если же хотим поставить индексы на неключевые столбцы, то это следует делать вручную, а не надеяться на СУБД.

Создание индексов для существующих таблиц (1) (основы)

```
CREATE [UNIQUE|FULLTEXT|SPATIAL]
    INDEX [название_индекса]
        [тип_индекса]
        ON название_таблицы (столбец_индекса,...)
        [опция_индекса]

опция_индекса: {
    тип_индекса
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
}

столбец_индекса:название_столбца [(length)] [ASC | DESC]

тип_индекса:USING {BTREE | HASH}
```

Между двумя первыми ключевыми словами **CREATE** и **INDEX** мы можем выбрать подвид индекса (хотя это необязательно). Уникальный индекс (**UNIQUE**) означает, что столбец, которому мы присваиваем должен хранить только уникальные значения - это позволит поиску остановиться при первом нахождении запрошенного значения. Полнотекстовый индекс (**FULLTEXT**) специально оптимизирован для поиска в значениях больших текстовых столбцов (**TEXT**, **LONGTEXT**). Пространственный индекс (**SPATIAL**) - должен использоваться для ускорения поиска по геометрическим типам столбцов, например таким как **GEOMETRY**.

Создание индексов для существующих таблиц (2) (описание параметров)

Тип индекса в MySQL представлен, во-первых, в виде специальной структуры данных, т.н. сбалансированного дерева (**BTREE**), которое является вариантом по умолчанию. Во-вторых, индекс может хранится в виде хеш-строки - сравнительное короткого значения, которое **в большинстве случаев** будет уникальным для текста, который длиннее этого значения. Используется только для движка **MEMORY**. В дальнейшем мы еще вернемся к анализу дерева и хеш-строки немного подробнее.

После ключевого слова **ON** и названия таблицы в скобках следует указать название того столбца (или нескольких столбцов), к которым будет относиться индекс.

В конце объявления индекса мы можем указать некоторые дополнительные опции для него. Если мы еще не объявляли тип индекса, то можно сделать это здесь. После ключевого слова **COMMENT** можно кратко описать, что же такое делает этот индекс. Затем мы может указать атрибут **VISIBLE** или **INVISIBLE**, который отвечает за то, будет ли индекс использоваться при выборках (надо отметить, что даже если использование будет выключено (т.е. если указан **INVISIBLE**), то индекс все равно будет обновляться и дополняться при вставках, модификациях и удалениях).

Создание индексов для существующих таблиц (3) (пример)

```
-- создание таблицы
CREATE TABLE workers (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL
);
```

```
-- создание индекса для таблицы

-- 1-й способ с доп. параметрами (описанный в схеме)
CREATE INDEX name_idx USING BTREE ON workers(name) VISIBLE;

-- 2-й способ без параметров (описанный в схеме)
CREATE INDEX name_idx_2 ON workers(name);

-- 3-й способ (альтернативный синтаксис)
ALTER TABLE workers ADD INDEX name_idx_3(name);
```

Создание индексов для новых таблиц (1) (с автоматическими названиями)

В самом простом случае индекс создается следующим образом - после перечисления всех столбцов ставится ключевое слово **INDEX** (также допускается псевдоним **KEY**), а сразу после него в скобках указываем название столбца (или нескольких столбцов через запятую):

```
CREATE TABLE products (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    code VARCHAR(255) UNIQUE NOT NULL,
    description TEXT NULL,
    type VARCHAR(255) NOT NULL,
    price DECIMAL(30,2),
    INDEX(name) -- объявление индекса
);
```

Просмотр существующих индексов

```
SHOW INDEXES FROM products;
```

Key_name	Column_name	Index_type	Visible
PRIMARY	id	BTREE	YES
code	code	BTREE	YES
name	name	BTREE	YES

Во-первых, произошло именно то, о чем мы говорили - мы не устанавливали индексы для первичного (**id**) и уникального ключа (**code**), но они были созданы автоматически. Во-вторых, всем индексам автоматически было установлено название - если с уникальным и первичным индексом это оправдано, то для индекса, созданного вручную, это может создать проблемы при удалении. В-третьих, мы могли заметить, что индексы получили тип **BTREE** и атрибут **Visible**, установленный как **YES**.

Создание индексов для новых таблиц (2) (с явно указанными названиями)

```
DROP TABLE IF EXISTS products;

CREATE TABLE products (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    code VARCHAR(255) UNIQUE NOT NULL,
    description TEXT NULL,
    type VARCHAR(255) NOT NULL,
    price DECIMAL(30,2),
    INDEX name_idx (name) -- создание индекса с явным указанием имени
);
```

```
SHOW INDEXES FROM products;
```

Key_name	Column_name	Index_type	Visible
PRIMARY	id	BTREE	YES
code	code	BTREE	YES
name	name_idx	BTREE	YES

Удаление индексов

Удаление осуществляется при помощи ключевой конструкции **DROP INDEX …** или **ALTER TABLE … DROP INDEX …**:

```
-- 1-й способ
DROP INDEX name_idx ON products;

-- 2-й способ
ALTER TABLE products DROP INDEX name_idx;
```

Префиксные индексы (теория)

Если нам нужен индекс, который индексирует не все значение столбца, а только некоторое количество его первых символов или байтов (т.н. префиксный индекс), то мы можем указать это количество при создании индекса. Однако перед тем, как это сделать, запомним несколько правил:

- Префиксные индексы могут создаваться только для текстовых и бинарных типов столбцов - для числовых и временных типов этого делать не следует.

- Для типов **TEXT** и **BLOCK** (и тех, которые могут больше по размеру) при создании индекса кол-во индексируемых данных надо указывать обязательно - т.е. в этих случаях префиксный индекс должен присутствовать всегда (но у нас есть возможность определять его длину).

- Для текстовых типов столбцов (**CHAR**, **VARCHAR**, **TEXT**) длина означает количество символов, а для бинарных (**BINARY**, **VARBINARY**, **BLOB**) - количество индексируемых байт.

- Максимально разрешенная длина данных отличается для разных движков и для разных настроек. Для InnoDB гарантированно разрешенная длина - 767 байт, а в свою очередь для MyISAM - 1000 байт.

- Очевидно, что префиксы могут помочь ускорить выборку, когда их столбцы используются в директиве **WHERE**, однако они бесполезны при при попытке ускорении сортировки (**ORDER BY**) и группировки (**GROUP BY**), т.к. в этих случая при сравнениях используется вся длина значения.

Префиксные индексы (создание)

Создадим префиксный индекс для столбца **description** (таблица **products**) - будем индексировать первые триста символов:

```
CREATE INDEX description_idx ON products( description(300) );
```

Многостолбцовые индексы (введение)

Иногда нам надо указать индекс для нескольких столбцов одновременно. Заметим, что создание двух индексов для двух столбцов и создание одного индекса для этих столбцов - разные вещи, т.к.

многостолбцовый индекс

```
CREATE TABLE IF NOT EXISTS single_index_products (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    code VARCHAR(255) UNIQUE NOT NULL,
    description TEXT NULL,
    type VARCHAR(255) NOT NULL,
    price DECIMAL(30,2),
    INDEX price_name_idx (price, name)
);
```

несколько индексов на разные столбцы

```
CREATE TABLE IF NOT EXISTS multiple_indices_products (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    code VARCHAR(255) UNIQUE NOT NULL,
    description TEXT NULL,
    type VARCHAR(255) NOT NULL,
    price DECIMAL(30,2),
    INDEX price_idx (price),
    INDEX name_idx (name)
);
```

Многостолбцовые индексы (проблема при раздельном применении)

Главная проблемы состоит в том, что во время запроса MySQL может полноценно использовать только один индекс на таблицу - рассмотрим пример для таблицы **multiple_indices_products**, которая содержит два индекса - **price_idx** и **name_idx**:

```
SELECT COUNT(*) FROM multiple_indices_products WHERE price = 500 AND name = 'Smartphone';
```

В данном случае интерпретатор СУБД сам решит, какой именно индекс из двух выбрать при выборке. Безусловно, современные версии MySQL умеют очень специфичным образом объединять сразу несколько индексов при выборке (с помощью объединения, пересечения или объединения пересечений), однако это никогда не даст нам гарантированного результата для всех случаев.

Многостолбцовые индексы (решение проблемы раздельного использования)

В отличие от предыдущего случая, гарантию ускорения запроса нам может дать многостолбцовый индекс - при следующем запросе в таблицу **single_index_products** у нас с максимальной эффективностью отработает индекс **price_name_idx**:

```
SELECT COUNT(*) FROM single_index_products WHERE price = 500 AND name = 'Smartphone';
```

Порядок столбцов в многостолбцовых индексах (1)

У многостолбцовых индексов существует еще одна особенность - порядок столбцов в таком индексе имеет значение. Однако если мы повторим предыдущий запрос, но поменяем в нем порядок столбцов в условии после **WHERE**, то это никак не скажется на работе индекса - выборка по прежнему будет эффективна. Интереператор MySQL достаточно умен, чтобы самому подогнать порядок столбцов условия **WHERE** под тот порядок, который используется в индексе.

```
SELECT COUNT(*) FROM single_index_products WHERE name = 'Smartphone' AND price = 500;
```

Так какое-же значение имеет порядок столбцов? Дело в том, что в MySQL индексирование происходит, условно говоря, слева направо. Это значит, что без использования тех столбцов, которые установлены в индексе сначала (слева), индекс перестанет быть таким же эффективным, каким он мог быть в противоположном случае.

Порядок столбцов в многостолбцовых индексах (2)

Для подтверждения тезиса о проблемности порядка полей перепишем предыдущий запрос, но оставим фильтрацию по столбцу name (который в индексе стоит вторым - т.е. условно справа):

```
-- индекс price_name_idx не будет полноценно отрабатывать (-)
SELECT COUNT(*) FROM single_index_products WHERE name = 'Smartphone';
```

Здесь индекс либо вообще перестанет работать, либо MySQL будет применять т.н. **skip scan** (доступен начиная с MySQL 8.0.13) - неявно сделает запрос для получения всех уникальных значений столбца **price**, а уже потом по ним сделает выборку по **name**, что позволит немного сократить количество обрабатываемых строк.

Данные странности происходят потому, что столбец name в индексе **price_name_idx**, перечислен вторым, а первый столбец индекса - **price** - в запросе вообще явно не используется. Мы можем сделать вывод, что все последующие столбцы индекса полноценно не работают без использования предыдущих столбцов. Наоборот же все работает прекрасно - если мы воспользуемся в запросе только первым столбцом индекса - **price**, и вообще не упомянем второй (**name**), то индекс отработает без всяких дополнительных уловок и поможет оптимизировать запрос.

```
-- индекс price_name_idx будет отрабатывать (+)
SELECT COUNT(*) FROM single_index_products WHERE price = 500;
```

Особенности использования индексов в контексте оператора LIKE (1)

Мы могли убедиться, что индексы очень хорошо взаимодействуют с фильтрацией данных при использовании директивы **WHERE** и операторов сравнения. Однако что будет происходить, если мы будет использовать дополнительный оператор **LIKE**, который используется для поиска строки не по конкретным значениям, а по части значения какого-либо текстового столбца? Чтобы ответить на этот вопрос, для начала вернемся к нашей оригинальной таблице **products**, и восстановим ранее удаленный индекс для столбца name:

```
CREATE INDEX name_idx ON products(name);
```

Когда мы начнем делать разные запросы, где будут участвовать оператор **LIKE** и столбец **name**, то убедимся, что индекс будет вести себя примерно так, как он вел себя в случае многостолбцовых индексов. Например, если нас интересуют все совпадения, где искомые символы должны находится в самом начале текста в столбце, то индекс будет отрабатывать идеально:

```
-- индекс name_idx будет полноценно отрабатывать (+)
SELECT COUNT(*) FROM products WHERE name LIKE 'John%';
```

Особенности использования индексов в контексте оператора LIKE (2)

Однако в том случае, если мы попытаемся найти столбец, некие символы которого находятся в конце или в середине текста, то индекс перестанет нам помогать ускорять запрос:

```
/*
   В обоих представленных ниже случаях индекс будет
   действовать неэффективно, т.к. поиск происходит не в
   начале строки, а в конце или в середине текста
*/
SELECT COUNT(*) FROM products WHERE name LIKE '%John';

SELECT COUNT(*) FROM products WHERE name LIKE '%John%';
```

Несколько индексов для одного столбца

В конце ответим на один казалось бы странный вопрос - может ли один и тот же столбец использоваться в разных индексах? Ответ - да! Мы можем даже сделать два одинаковых индекса (конечно же с разными именами) на одно и то же поле, однако это не имеет особого смысла и в будущих версиях MySQL будет запрещено. Смысл есть только в том случае, если столбец используется внутри многостолбцового индекса на любой позиции, отличной от первой, однако иногда нам нам использовать именно этот конкретный столбец для поиска по таблице:

```
-- индекс, где столбец code находится на второй позиции
CREATE INDEX price_code_idx ON products(price, code);

/*
   индекс, который поможет ускорить запрос, если в нем будет
   присутствовать фильтрация только по столбцу code
*/
CREATE INDEX code_idx ON products(code);
```

Структура B-TREE индексов

Введение и подготовка (1)

В предыдущей подглаве мы подробно проанализировали то, как индексы работают, а в этой подглаве попытаемся ответить на несколько более сложный вопрос - почему они вообще работают? Ответ на этот вопрос тесно связан с той структурой, которой обладают индексы - это довольно сложная тема, поэтому попробуем разобраться в ней начиная с самых простых вещей.

Мы уже упоминали, что MySQL для хранения индексов может применять два способа - использовать сбалансированное дерево и хеш-строку. Основным способом, который разрешен для всех движков, является использование дерева, поэтому начнем анализ структуры индексов именно с него. Как мы помним, в таблице products у нас есть индекс **price_idx** для столбца цены продукта (**price**) - возьмем для примера именно его. Но для начала занесем в таблицу 7 продуктов (число не имеет значения, оно выбрано просто для примера):

```
INSERT INTO multiple_indices_products(id, name, code, type, price) VALUES
(1, 'Pinocchio', 'a00c1', 'book', 150),
(2, 'Robinson Crusoe', 'a00c2', 'book', 125),
(3, 'Tom Sawyer', 'a00c3', 'book', 225),
(4, 'The Lord of the Rings', 'a00c4', 'book', 80),
(5, 'The Jungle Book', 'a00c6', 'book', 25),
(6, 'Treasure Island', 'a00c7', 'book', 220),
(7, 'Harry Potter', 'a00c8', 'book', 50);
```

Введение и подготовка (2)

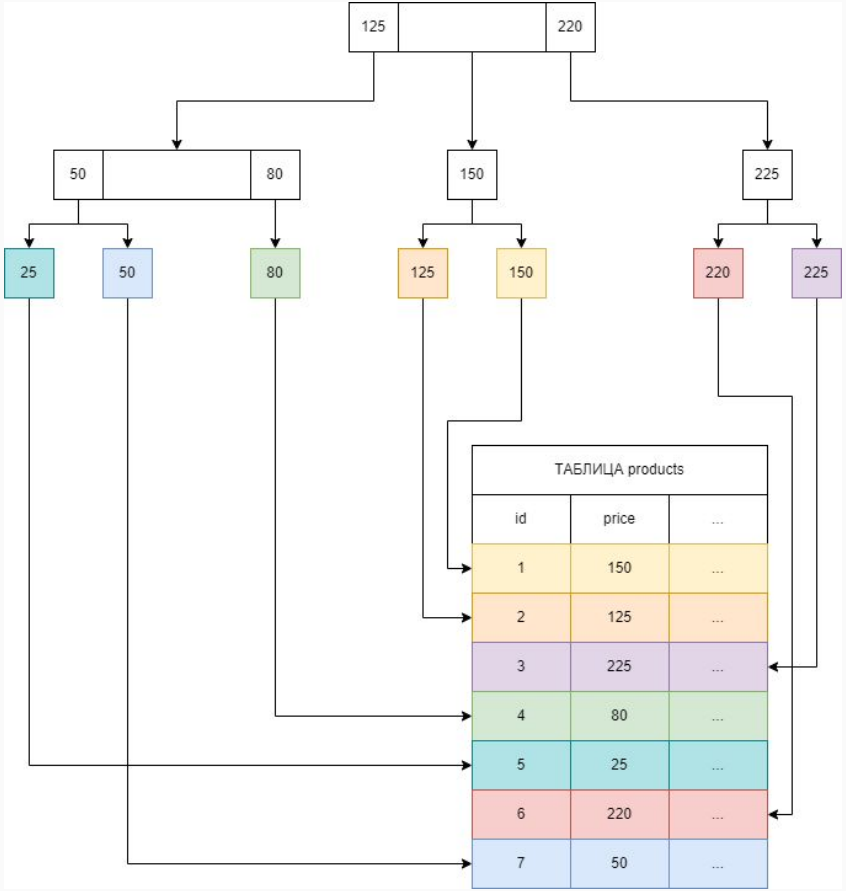
Содержимое таблицы multiple_indices_products:

id	name	code	description	type	price
1	Pinocchio	a00c1	NULL	book	150
2	Robinson Crusoe	a00c2	NULL	book	125
3	Tom Sawyer	a00c3	NULL	book	225
4	The Lord of the Rings	a00c4	NULL	book	80
5	The Jungle Book	a00c6	NULL	book	25
6	Treasure Island	a00c7	NULL	book	220
7	Harry Potter	a00c8	NULL	book	50

Визуальная модель B-TREE индекса price_idx

После того, как мы создали все наши продукты, MySQL построит следующее сбалансированное дерево и сохранит его в качестве структуры поиска для индекса **price_idx**.

Можно заметить, что дерево отсортировано особенным образом - ветви-потомки, которые находятся внизу слева от каких-либо узловых элементов, всегда меньше, чем значение узлового элемента (25 меньше 40), а ветви-потомки, которые находятся внизу справа - больше или равны узловому элементу (150 больше 125). Однако ни один потомок справа не может быть больше, чем тот узловой элемент, который находится слева и на одном уровне с родительским элементом (80 не может быть левее, чем 220). Также мы должны увидеть, что самые нижние ответвления дерева имеют ссылки на строки таблицы **multiple_indices_products**.



Алгоритм поиска в контексте B-TREE индекса

Так как же этот индекс работает? Представим, что нам надо найти продукт, цена (столбец price) которого равна 50 евро. MySQL обнаружит, что у этого столбца есть индекс и попытается его применить, чтобы найти необходимую строку:

1. Начиная с верхнего узла индекса интерпретатор начнет следующие действия: если значение меньше, чем значение узлового элемента - мы идем в нижний левый потомок, если больше или равно - в нижний правый потомок. В самом начале нас есть три варианта - 1) меньше 125, 2) больше или равно 125, но меньше 225, 3) больше или равно 225. 50 меньше, чем 125, поэтому идем в левый нижний потомок этого узла (50 - 80).

2. 50 равняется 50, но меньше, чем 80, поэтому мы переходим к узлу, который справа внизу от 50.

3. Узел, к которому мы перешли (опять 50) является самым крайним ответвлением дерева и имеет ссылку на строку в таблице с id = 7. Поиск успешно закончен.

Общая закономерность алгоритм поиска в контексте B-TREE индекса

Только что мы увидели, как в таблице из 7 элементов, необходимый нам элемент (который был последним) был найден за 3 шага. Если бы индекса не было, то интерпретатору пришлось сканировать всю таблицу и для этого потребовалось 7 шагов. Кажется, что экономия небольшая, но ниже представлено сравнение количества шагов, которое необходимо сделать в более крупных по количеству строк таблицах:

Кол-во строк	Шаги для поиска без индекса	Шаги для поиска с индексом
100	100	~7
10000	10000	~14
1000000	1000000	~20

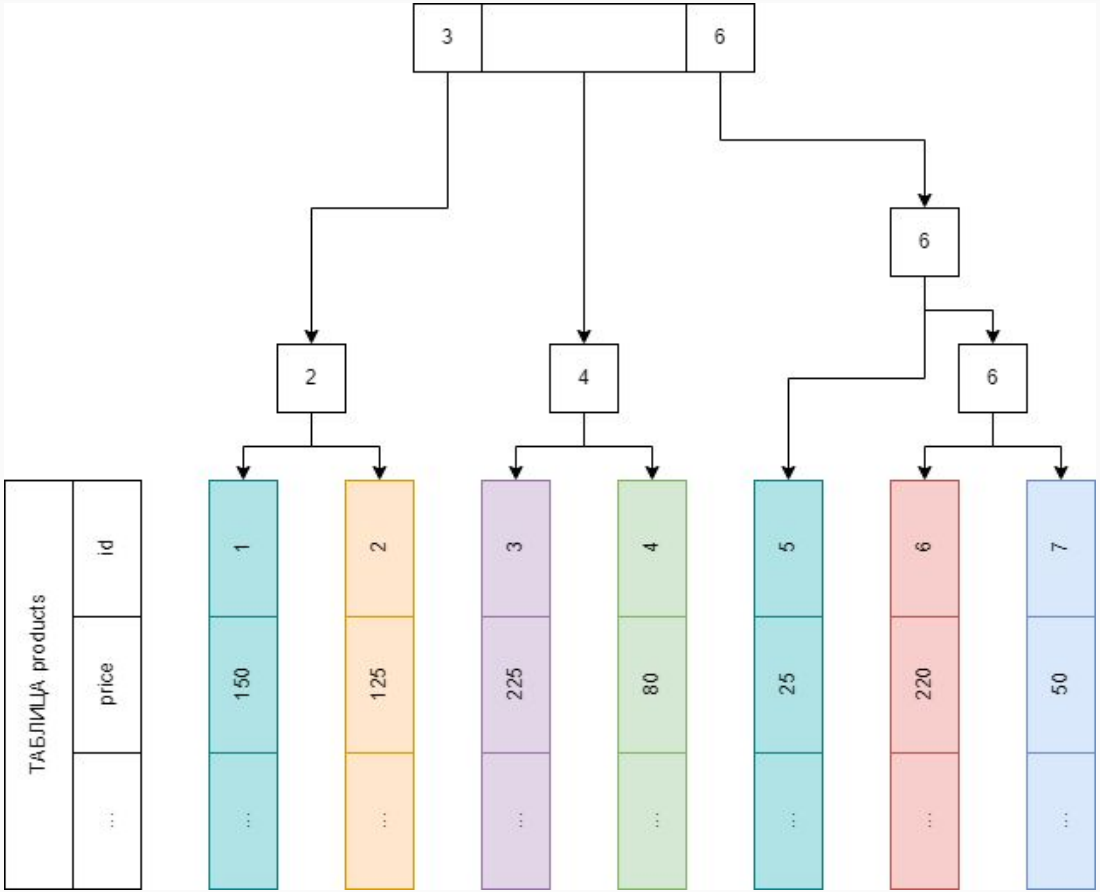
Можно понять, что чем больше строк в таблице, тем больше шагов индекс позволяет нам сэкономить. В последнем случае с миллионом строк при использовании индекса мы получаем ускорение поиска в 50000 (пятьдесят тысяч) раз, а сама сложность алгоритма при любых раскладах не будет превышать $\log_2 x$, где x - количество строк в таблице.

Кластерные индексы (InnoDB) (1)

Пример, который мы только что рассмотрели, довольно подробно объясняет индексацию в движке MyISAM, и касается некоторых случаев, которые связаны с InnoDB. Но он не совсем верно описывает т.н. кластерный индекс, который в MySQL используется только в InnoDB.

Стоит начать с того, что в случае использования InnoDB, в таблице всегда создается первичный ключ - даже в том случае, если мы его не указали при создании или позже (если мы создали сами, то берется наш). Если получиться, интерпретатор возьмет один из столбцов таблицы, если же ни один столбцов не подходит, то будет создан виртуальный числовой столбец, на который и будет установлен первичный ключ. После этого происходит главное - при создания InnoDB таблицы автоматически помещается в специальный индекс целиком, узлами которого будут значения столбца с первичным ключом. Это значит, что крайние узлы сбалансированного дерева индекса не просто будут ссылаться на строки таблицы - они будут содержать эти строки (продолжение на следующей странице).

Кластерные индексы (InnoDB) (2)



Кластерные индексы (InnoDB) (3)

Зачем же применяется такой необычный механизм? Дело в том, что даже операции перехода по ссылкам, а также операции чтения данных могут быть очень затратными по времени. Если же мы с помощью кластерного индекса нашли нужное нам значения, то нам больше не надо совершать никаких дополнительных манипуляций - данные доступны сразу же.

Очевидно, что кластерный индекс для каждой InnoDB таблицы может быть только один, а также должно быть понятно, что столбцом этого индекса является столбец первичного ключа. Если мы создадим для этой таблицы дополнительный индекс, то он будет выглядеть так же, как и в предыдущем обобщенном примере. Правда, когда он будет обращаться по ссылке в таблицу, то там тоже будет происходить поиск по индексу, ведь таблица в этом индексе и находится. Но, как мы уже поняли, этот поиск будет быстрым - практически мгновенным.

Структура HASH индексов

Введение и подготовка

Теперь проанализируем особенности хеш-индексов, которые работают только в таблицах на движке **MEMORY**, т.е. в таблицах, хранящихся в памяти компьютера. Для примера не будем изобретать велосипед - создадим и заполним данными таблицу **mem_products**, которая будет копией таблицы **products**. Будут только 2 отличия - во-первых, на столбец имени товара (**name**) будет установлен интересующий нас **hash**-индекс, а во-вторых будет отсутствовать столбец **description** (движок **MEMORY** не разрешает использования столбцов **TEXT** и **BLOB**):

```
CREATE TABLE mem_products (
   id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
   name VARCHAR(255) NOT NULL,
   code VARCHAR(255) UNIQUE NOT NULL,
   type VARCHAR(255) NOT NULL,
   price DECIMAL(30,2),
   INDEX code_idx USING HASH(name)
) ENGINE=MEMORY;

INSERT INTO mem_products (id, name, code, type, price) VALUES
(1, 'Pinocchio', 'a00c1', 'book', 150),
(2, 'Robinson Crusoe', 'a00c2', 'book', 125),
(3, 'Tom Sawyer', 'a00c3', 'book', 225),
(4, 'The Lord of the Rings', 'a00c4', 'book', 80),
(5, 'The Jungle Book', 'a00c6', 'book', 25),
(6, 'Treasure Island', 'a00c7', 'book', 220),
(7, 'Harry Potter', 'a00c8', 'book', 50);
```

Визуальная модель HASH индекса code_idx

В результате всех наших действий хеш-индекс будет сохранен не как сбалансированное дерево, а как таблица примерно следующего вида:

Цифровые значения в столбце name_hash получились следующим образом - MySQL берет значение индексируемого столбца (это значение может быть любого типа - текстовым, числовым, временным) и затем прогоняет его через специальную хеш-функцию, которая должна вернуть число в строго определенном диапазоне. Притом для одинаковых значений это число должно быть одинаковым. Затем эти числа заносятся в вышеуказанную таблицу в отсортированном порядке и к каждому числу добавляется ссылка на строку, с которой это число связано.

name_hash	link_to_table_row
0	*mem_products* row with id = 3
1	*mem_products* row with id = 7
2	*mem_products* row with id = 5
3	*mem_products* row with id = 6
4	*mem_products* row with id = 2
5	*mem_products* row with id = 1
6	*mem_products* row with id = 6

Алгоритм поиска в контексте HASH индекса

Затем приходит пора произвести выборку по индексированному столбцу, например, с помощью такого запроса:

```
SELECT * FROM mem_products WHERE name = 'Treasure Island';
```

В этом случае, запрашиваемое значение столбца name с помощью хеш функции преобразуется в число и, если оно есть в таблице индексов, мгновенно находится и мы получаем ссылку для чтения данных. Т.к. числа стоят в отсортированном порядке, надо всего лишь знать адрес в памяти, где начинается таблица, затем умножить пришедшее число на размер ячейки, хранящей индекс и сразу же туда перейти - сложность алгоритма - константная (т.е. не зависит от размера таблицы). Вот это и есть преимущество хеш-индексов - они крайне быстры, им не надо тратить время, чтобы обходить дерево - они умеют это делать фактически без поиска.

Недостатки HASH индексов

При всех преимуществах, у хеш-индексов есть и недостатки:

- При поиске используется точное соответствие (хеш запрошенного значения должен точно соответствовать хешу значения в таблице). Это значит, что мы не должны использовать операторы > =>, < =< и **LIKE**, т.к в этом случае хеш-индекс перестает работать.

- У нас действительно есть гарантия, что для одинаковых значений хеш-функция генерирует одинаковые числа, но допускается, что одинаковые числа могут быть сгенерированы для разных значений. Эти случаи называются коллизиями. MySQL умеет решать случаи, когда такие коллизии случаются, одна в том случае, если их будет слишком много, индекс перестанет ускорять запросы.

- Хеш-индексы не ускоряют сортировку при помощи **ORDER BY**.

- хеш-индексы не поддерживают поиск по частичному ключу (в многостолбцовых индексах), поскольку хеш-коды вычисляются для всего индексируемого значения.

Структура многостолбцовых индексов

Введение и подготовка

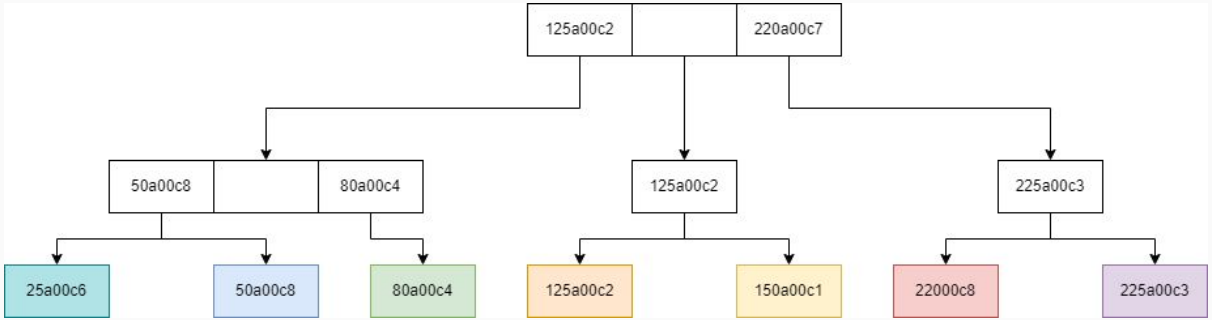
Мы должны уже знать, что порядок столбцов в многостолбцовом индексе имеет значение - наиболее эффективно оптимизация запроса происходит в том случае, если при фильтрации используются либо все столбцы индекса, либо те, которые стоят вначале. Попытаемся узнать, почему это происходим имеено так (в данном случае мы будем рассматривать **B-TREE** индексы, т.к. для хеш-индексов все очевидно - создание хеша для всех столбцов одновременно и невозможность поиска по части индекса - даже начальной).

Для наших экспериментов создадим в нашей таблице **multiple_indices_products** еще один индекс, который объединяет столбцы **price** и **code**:

```
CREATE INDEX price_code_idx ON multiple_indices_products(price, code);
```

Визуальная модель B-TREE индекса price_code_idx

После создания индекса для всех входящих в индекс столбцов будут созданы примерно следующие значения - можно заметить, что это склеенные значения оригинальных столбцов:



Теперь должно стать понятно, почему в многостолбцовых индексах первых столбцы имеют больший приоритет, чем те столбцы, которые стоят после них - данная структура полностью повторяет дерево, которые было посвящено ценам (без кодов). Ведь при сортировке дерева та часть, которая находится вначале (она получается из значений первого столбца) как раз и определяет положение узла индекса в дереве. Например, для нас не имеет значения, что 00c1 - наименьшее значение среди столбцов code, все равно общее индексное значение, в которое будет входить этот столбец (150a00c1) будет больше, чем 25a00c6 (для него значение кода - a00c6 - является как раз одним из самых больших). Значения последующих столбцов имеют значение только в том случае, если значения первых столбцов одинаковы.

Полнотекстовые индексы и полнотекстовый поиск

Введение

При правильной работе с индексами наши запросы могут быть ускорены в десятки и сотни раз. Однако мы столкнулись с одним печальным ограничением - нельзя ставить обычные индексы на столбцы, представляющие собой длинный текст (**TEXT**, **LONGTEXT** и т.д.), а префиксные индексы позволяют индексировать лишь небольшую часть значения в самом начале значения колонки. Значит ли это, что мы должны смириться с невозможностью быстрого поиска по крупным текстовым массивам? Конечно же нет, ведь в MySQL есть поистине изумительный инструмент, который позволяет не смотреть на длину текста - этот инструмент называется полнотекстовым индексом (**FULLTEXT INDEX**). Однако данная вещь доступна только для движков MyISAM и InnoDB, притом InnoDB начал поддерживаться такие индексы относительно недавно - начиная с версии MySQL 5.6.4.

Подготовка (создание таблицы) (1)

Создание полнотекстовых индексов мало чем отличается от создания обычных индексов - единственное дополнительное слово, которое следует использовать, это **FULLTEXT**. Создадим для этого таблицу **books**, с теми произведениями которые у нас уже использовались в таблице **mem_products**, однако уберем тип, код и цену, а добавим автора (**author**) и по одной строке из каждой книге в качестве содержимого (**content**). **Content** и будет тем столбцом, для которого будет установлен полнотекстовый индекс. Создавать таблицу будем на движке InnoDB:

```
CREATE TABLE books (
   id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
   title VARCHAR(255) NOT NULL,
   author VARCHAR(255) NOT NULL,
   content TEXT NOT NULL,
   FULLTEXT INDEX content_idx(content)
) ENGINE=InnoDB;
```

Подготовка (добавление данных) (2)

```
INSERT INTO books(id, title, author, content) VALUES

(1, 'Pinocchio', 'C. Collodi', 'How ridiculous I was as a Marionette! And how happy I am, now that I have become a real boy!'),

(2, 'Robinson Crusoe', 'D. Defoe', 'When I took leave of this island, I carried on board, for relics, the great goat-skin cap I had made, my umbrella, and one of my parrots'),

(3, 'Tom Sawyer', 'M. Twain', 'While Tom was eating his supper, and stealing sugar as opportunity offered, Aunt Polly asked him questions that were full of guile.'),

(4, 'The Lord of the Rings', 'J. R. R. Tolkien', 'As is told in The Hobbit, there came one day to Bilbo's door the great Wizard, Gandalf the Grey, and thirteen dwarves with him.'),

(5, 'The Jungle Book', 'J.R. Kipling', 'All this will show you how much Mowgli had to learn by heart, and he grew very tired of saying the same thing over a hundred times.'),

(6, 'Treasure Island', 'R. L. Stevenson', 'Of Silver we have heard no more. That formidable seafaring man with one leg has at last gone clean out of my life.'),

(7, 'Harry Potter', 'J. K. Rowling', 'Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much.');
```

Подготовка (дополнительный многостолбцовый индекс) (3)

Многотекстовые индексы могут быть установлены для нескольких столбцов сразу, т.е. они тоже могут быть составными. Создадим такой составной индекс, который будет ссылаться на название книги (**title**) и ее содержимое (**content**) одновременно. Создадим его уже после создания таблицы, поэтому применим немного другой синтаксис:

```
CREATE FULLTEXT INDEX title_content_idx ON books(title, content);
```

Особенности синтаксиса при полнотекстовом поиске

Первое, что бросается в глаза - полнотекстовый поиск имеет другой синтаксис. После **WHERE** необходимо вписать конструкцию **MATCH**, после которой в скобках следует указать столбец или столбцы (через запятую), по которым будет производиться выборка. Затем необходимо написать ключевое слово **AGAINST** и после него в скобках написать то текстовое значение, по которому мы хотим произвести поиск. Для примера найдем при помощи полнотекстового поиска все книги, где в содержимом есть слово ***Silver***:

```
SELECT title FROM books WHERE MATCH(content) AGAINST ('Silver');
```

Конечно же, Джон Сильвер - легендарный харизматичный пират из книги “Treasure Island”:

title
Treasure Island

Ограничение полнотекстового поиска

Полнотекстовый поиск разрешен только по тем столбцам и комбинациям столбцов, у которых есть полнотекстовый индекс (есть одно небольшое исключение, но мы поговорим о нем немного позднее). Как мы помним, обычный поиск может производиться независимо от того, есть ли на столбцах индексы или нет. Для примера осуществим многотекстовый поиск по столбцу **author**:

```
SELECT title FROM books WHERE MATCH(author) AGAINST ('Kipling');
```

В ответ получим закономерную ошибку:

```
ERROR 1191 (HY000): Can't find FULLTEXT index matching the column list
```

Список стоп-слов для полнотекстового поиска (введение)

У полнотекстового поиска и индекса может быть список т.н. стоп-слов, которые будут просто игнорироваться при поиске и индексации. Список стоп-слов по умолчанию для движка InnoDB можно узнать при помощи следующего запроса:

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_FT_DEFAULT_STOPWORD;
```

Если нам надо создать свой список стоп слов, то надо создать таблицу, идентичную **INNODB_FT_DEFAULT_STOPWORD**, заполнить ее необходимыми значениями, а затем установить для переменной **innodb_ft_aux_table** значение в следующем формате:

```
SET GLOBAL innodb_ft_aux_table = 'название_базы_данных/название_таблицы';
```

Для движка MyISAM стоп-слова определяются при помощи переменной **ft_stopword_file**, которую можно поместить в файл **my.ini** (раздел **[mysqld]**). В качестве значения этой переменной надо указать путь к файлу, где будут храниться интересующие нас слова (можно указать их, например, через запятую).

Список стоп-слов для полнотекстового поиска (использование)

Стоп-слова предназначены не для того, чтобы скрыть некие нецензурные выражения, а в первую очередь для того, чтобы отсечь малозначимые термины, которые потенциально могут присутствовать в каждой строке - такие, например, как английские артикли **the** и **a**, русские предлоги **но**, **и** и т.д. Для примера, сделаем запрос с артиклем **the** в нашу таблицу **books**:

```
SELECT COUNT(*) FROM books WHERE MATCH(title, content) AGAINST ('the');
```

В ответ мы получим закономерный ноль, т.к. слово **the** по умолчанию входит в список стоп-слов для движка InnoDB. Если же мы попытаемся сделать этот запрос обычным медленным способом, то увидим, что таких строк на самом деле четыре:

```
SELECT COUNT(*) FROM books WHERE title LIKE '%the%' OR content LIKE '%the%';
```

Подвиды полнотекстового поиска

У полнотекстового поиска есть целых три подвида - в режиме натурального языка (**IN NATURAL LANGUAGE MODE**), в логическом режиме (**IN BOOLEAN MODE**) и в режиме натурального языка с расширением запроса (**WITH QUERY EXPANSION**). В дальнейшем мы попытаемся подробно рассмотреть каждый перечисленный подвид.

Многотекстовый поиск IN NATURAL LANGUAGE MODE

Введение

Поиск в режиме натурального языка является подвидом поиска по умолчанию. Мы уже сталкивались с ним в предыдущей подглаве, когда, например, искали книгу со словом ***Silver***, просто мы делали это неявно. Однако можно сделать это явно:

```
-- эти два запроса являются идентичными

SELECT title FROM books WHERE MATCH(content) AGAINST ('Silver');

SELECT title FROM books WHERE MATCH(content) AGAINST ('Silver' IN NATURAL LANGUAGE MODE);
```

Пример продвинутого использования поиска IN NATURAL LANGUAGE MODE (1)

Как же происходит поиск в этом режиме? В скобках после слова **AGAINST** мы в кавычках указываем некий текст, по которому производится поиск. Однако для нахождения строк не требуется полного соответствия. При запросе интерпретатор разделяет текст в скобках на слова (разделителем являются пробелы, запятые, точки или двойные апострофы "), а уже затем ищет - если хотя бы одно слово целиком будет найдено в значении столбца - все строка попадет в результирующую выборку. Притом порядок слов не важен. Приведем пример - запросим все книги, в которых встречаются слова Gandalf или Hobbit:

```
SELECT title FROM books WHERE MATCH(content) AGAINST ('Gandalf Hobbit');
```

Несмотря на то, что запрашиваемые слова не стоят подряд, более того, в нашем тексте сначала идет слово Hobbit, а уже потом Gandalf, поиск успешно нашел книгу “Властелин колец”:

title
The Lord of the Rings

Пример продвинутого использования поиска IN NATURAL LANGUAGE MODE (2)

Теперь попробуем указать несколько слов, из которых только одно присутствует в нашей таблице - посмотрим, что выйдет:

```
SELECT title FROM books WHERE MATCH(content) AGAINST ('Akela Mowgli Shere Khan');
```

Запрос найдет один, результат, даже несмотря на то, что только слово Mowgli действительно может быть найдено:

title
The Jungle Book

Пример продвинутого использования поиска IN NATURAL LANGUAGE MODE (3)

Далее попробуем произвести поиск, в котором вернется несколько строк - для этого поищем по словам, которые казалось бы относятся только к “Острову сокровищ”:

```
SELECT title FROM books WHERE MATCH(content) AGAINST ('Silver one leg');
```

Безусловно, мы получили в ответ “Остров сокровищ”, однако слово one присутствовало также в книгах “Робинзон Крузо” и “Властелин колец” - поэтому они и попали в конечную таблицу:

title
Treasure Island
Robinson Crusoe
The Lord of the Rings

Пример продвинутого использования поиска IN NATURAL LANGUAGE MODE (4)

Пришла пора протестировать запрос, в котором будет присутствовать только часть слова, существующего в таблице - попробуем слово ***parrot***:

```
SELECT title FROM books WHERE MATCH(content) AGAINST ('parrot');
```

В ответ мы получим пустую таблицу - в книге “Robinson Crusoe” есть слово ***parrots***, но такие случаи не учитываются при использовании этого подвида полнотекстового поиска.

Пример продвинутого использования поиска IN NATURAL LANGUAGE MODE (5)

Упомянем еще одну особенность натурального поиска - дело в том, что он возвращает строки таблицы отсортированными, даже если мы явно не применяем сортировку (**ORDER BY**). Сортировка происходит по соответствию запрошенных данных найденной строке - чем больше соответствие, Тем выше будет находиться строка. Если же мы поставим в запросе нашу сортировку вручную, то данная сортировка по соответствию применятся не будет. Если мы хотим увидеть это соответствие своими глазами, то мы можем написать после **SELECT** то же самое, что мы пишем после **WHERE** - в примере ниже присвоим этому виртуальному столбцу псевдоним score:

```
SELECT title, MATCH(content) AGAINST ('Silver one leg') score
FROM books WHERE MATCH(content) AGAINST ('Silver one leg');
```

title	score
Treasure Island	1.5637884140014648
Robinson Crusoe	0.13540691137313843
The Lord of the Rings	0.13540691137313843

Многотекстовый поиск IN BOOLEAN MODE

Введение

Если мы будем использовать многотекстовый поиск в логическом режиме (**IN BOOLEAN MODE**) также, как в натуральном режиме, то мы не увидим между ними никакой разницы:

```
SELECT title FROM books WHERE MATCH(content) AGAINST ('were' IN BOOLEAN MODE);
```

title
Harry Potter
Tom Sawyer

Однако логическим режим предлагает несколько специальных модификаторов, которые можно ставить возле слова в запросе - эти модификаторов могут кардинально поменять смысл запроса и сильно помочь в некоторых случаях:

- **+** слово должно обязательно присутствовать
- **-** Слово должно обязательно отсутствовать
- ***** после слово могут присутствовать другие символы (т.е. для поиска подается усеченное слово)
- **"some words"** - поиск должен производиться по точному соответствию без разбиению на слова
- **(some expressions)** - позволяет сгруппировать выражения с другими модификаторами

Пример продвинутого использования поиска IN BOOLEAN MODE (1)

Применим сразу же модификатор *, чтобы решить проблему со словом ***parrot***:

```
SELECT title FROM books WHERE
MATCH(content) AGAINST ('parrot*' IN BOOLEAN MODE);
```

В этом случае мы наконец смогли найти книгу “Робинзон Крузо”, в которой было слово ***parrots*** (т.е. ***parrot*** и ***s***):

title
Robinson Crusoe

Пример продвинутого использования поиска IN BOOLEAN MODE (2)

Далее повторим запрос про одноногого Сильвера опять, но в этот раз потребуем, чтобы все слова обязательно присутствовали в значении столбца:

```
SELECT title FROM books WHERE
MATCH(content) AGAINST ('+Silver +one +leg' IN BOOLEAN MODE);
```

Запрос отработал безукоризненно и вернул только одну строку, ведь среди наших книг только “Остров сокровищ” обладает всей полнотой информации об одноногом пирате:

title
Treasure Island

Пример продвинутого использования поиска IN BOOLEAN MODE (3)

Применим знания о модификаторах, чтобы отсечь ненужные данные - представим, что нам надо получить все строки, в которых есть слово were, но в которых нет слова ***Dursley***:

```
SELECT title FROM books WHERE
MATCH(content) AGAINST ('+were -Dursley' IN BOOLEAN MODE);
```

Нам вернётся только книга о Томе Сойере, ведь несмотря на то, что книга о Гарри Поттере тоже содержит слово were, там присутствие слово ***Dursley***, которое мы не хотим видеть:

title
Tom Sawyer

Дополнительная информация о поиске IN BOOLEAN MODE

Остальные модификаторы мы не будем рассматривать, т.к. в одном случае они очевидны (поиск без разбиения на слова), а в другом применяются очень редко (группировка выражений). Еще есть две вещи которые стоит добавить в этой подглаве. Во-первых, в режиме логического поиска строки результирующей таблицы по соответствию автоматически не сортируются. Во-вторых, для движка MyISAM разрешается применение логического многотекстового поиска без создания многотекстового индекса (хотя это и будет крайне медленно).

Многотекстовый поиск IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION

Введение

Данный подвид многотекстового поиска является самым необычным из всех. Его полное название **IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION**, но гораздо чаще используется краткая форма **WITH QUERY EXPANSION**. Когда мы производим данный запрос, то внутри MySQL происходит процесс, который следует разделить на две части. Сначала выполняется запрос, который полностью идентичный запросу в режиме натурального языка. Затем из полученных строк отбираются ключевые слова, которые не были в оригинальном запросе и уже с ними происходит второй запрос, результаты которого объединяются с результатами первого запроса и таким образом попадают в финальную выборку. Таким образом, это не только запрос по ключевым словам, которые прислал клиент, но и по ключевым словам тех строк, которые будут найдены по пользовательскому запросу. Это и называется расширением запроса.

Пример продвинутого использования поиска WITH QUERY EXPANSION

Наша таблица **books** не сильно хорошо подходит для такого подвида запросов, но все же попытаемся получить хоть что-то - запросим строки по слову ***Dursley***:

```
SELECT title FROM books WHERE
MATCH(content) AGAINST ('Dursley' WITH QUERY EXPANSION);
```

В результате мы получим все книги, кроме одной - “Остров сокровищ”. Это означает, что единственная книга, которая попала под прямой запрос - “Harry Potter” не имеет сколько нибудь значимого количества слов, которые могут быть найдены в “Острове сокровищ”, а все остальные книги такие слова имеют:

title
Harry Potter
The Jungle Book
Tom Sawyer
Pinocchio
Robinson Crusoe
The Lord of the Rings

Особенности работы полнотекстовых составных индексов

Объединение данных при полнотекстовом многостолбцовом поиске

В начале нашей главы мы создали индекс **title_content_idx**, который объединяет сразу два столбца - **title** и **content**. Скажем о нем несколько слов. Дело в том, что при создании полнотекстового индекса из нескольких столбцов все ключевые слова столбцов просто объединяются. Это означает, что при составления запроса для полнотекстового поиска (любого его подвида) мы должны думать о всех этих столбцах, как об одном целом. Допустим, выполним логический поиск, в котором обязательно должно быть слово Pinocchio и Marionette (эти два слова есть в одной строке, но в разных столбцах):

```
SELECT title, author FROM books WHERE MATCH(title, content)
AGAINST ('+Pinocchio +Marionette' IN BOOLEAN MODE);
```

MySQL успешно найдет необходимую нам книгу за авторством Карло Коллоди:

title	author
Pinocchio	C. Collodi

Полезные практики при работе с индексами

Правильное использование индексов

В дополнение ко всему сказанному в предыдущих главах, отметим еще несколько моментов. Самое главное, что надо запомнить - индексы надо создавать только для тех столбцов, которые используются в фильтрации (в контексте **WHERE**), сортировке (**ORDER BY**), группировке (**GROUP BY**) и соединениях с другими таблицами (все виды **JOIN**). Просто так создавать индексы крайне вредно. Во-первых, они занимают место на диске - в отдельных случаях индексы могут занимать больше места, чем оригинальная таблица. Во-вторых, MySQL вынуждена обновлять индексы при каждой модификации таблицы (вставка, удаление, обновление) - можно понять, что индексы замедляют эти действия - особенно если надо перестраивать индекс для таблицы на миллионы строк.

Особенности работы индексов при соединении таблиц (JOIN)

В рамках запроса для одной таблицы полноценно отрабатывает только один индекс. Это значит, что при соединении (**JOIN**) нескольких таблиц, мы должны ожидать, что для каждой таблицы может отработать свой отдельный индекс. Это действительно так, поэтому те столбцы, которые имеют ограничение внешнего ключа, являются первыми кандидатами на получение индекса. Ведь именно эти столбцы практически всегда используются при соединения таблиц.

Анализ запросов

Для анализа использования индексов внутри MySQL есть прекрасный инструмент - анализатор выполнения запроса. Для использования анализатора нам всего лишь надо перед оригинальным запросом поставить слово **EXPLAIN** - в качестве результата мы получим не строки таблицы, а информацию о том, каким образом СУБД будет исполнять наш запрос, сколько строк будет отфильтровано, какие индексы вообще доступны и какие из них будут действительно применяться. Приведем пример с таблицей **books** из первой главы (покажем только самые интересные столбцы в результирующей таблице):

```
EXPLAIN SELECT * FROM products WHERE price = 225;
```

possible_keys	key	filtered
price_idx,price_name_idx,price_code_idx	price_idx	100.00

Полученная информация показывает, что в таблице есть три индекса - **price_idx**, **price_name_idx** и **price_code_idx**, которые могут быть использованы для оптимизации нашего запроса. Также видно, что реально использованным индексом оказался **price_idx**. Кроме того, указано, что из всех обработанных строк, 100% попадет в результирующую таблицу (это отличный результат).