MySQL索引和查询优化 - 棂枫

对于任何DBMS，索引都是进行优化的最主要的因素。对于少量的数据，没有合适的索引影响不是很大，但是，当随着数据量的增加，性能会急剧下降。
如果对多列进行索引(组合索引)，列的顺序非常重要，MySQL仅能对索引最左边的前缀进行有效的查找。

例如：
假 设存在组合索引it1c1c2(c1,c2)，查询语句select * from t1 where c1=1 and c2=2能够使用该索引。查询语句select * from t1 where c1=1也能够使用该索引。但是，查询语句select * from t1 where c2=2不能够使用该索引，因为没有组合索引的引导列，即，要想使用c2列进行查找，必需出现c1等于某值。

索引是快速搜索的关键。MySQL索引的建立对于MySQL的高效运行是很重要的。

下面介绍几种常见的MySQL索引类型：
在数据库表中，对字段建立索引可以大大提高查询速度。假如我们创建了一个 mytable表：

```
1. CREATE TABLE mytable( ID INT NOT NULL, username VARCHAR(16) NOT NULL );
```

我们随机向里面插入了10000条记录，其中有一条：5555, admin。

在查找username="admin"的记录

```
1. SELECT * FROM mytable WHERE username='admin';
```

时，如果在username上已经建立了索引，MySQL无须任何扫描，即准确可找到该记录。相反，MySQL会扫描所有记录，即要查询10000条记录。

索引分单列索引和组合索引。单列索引，即一个索引只包含单个列，一个表可以有多个单列索引，但这不是组合索引。组合索引，即一个索包含多个列。

MySQL索引类型包括：(1)普通索引
这是最基本的索引，它没有任何限制。它有以下几种创建方式：

创建索引

```
1. CREATE INDEX indexName ON mytable(username(length));
```

如果是CHAR，VARCHAR类型，length可以小于字段实际长度;如果是BLOB和TEXT类型，必须指定 length，下同。

修改表结构

```
1. ALTER mytable ADD INDEX [indexName] ON (username(length))
```

创建表的时候直接指定

```
1. CREATE TABLE mytable( ID INT NOT NULL,
   username VARCHAR(16) NOT NULL, INDEX [indexName] (username(length)) );
```

删除索引的语法：

```
1. DROP INDEX [indexName] ON mytable;
```

(2)MySQL索引类型：唯一索引

它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

创建索引

```
1. CREATE UNIQUE INDEX indexName ON mytable(username(length))
```

修改表结构

```
1. ALTER mytable ADD UNIQUE [indexName] ON (username(length))
```

创建表的时候直接指定

```
1. CREATE TABLE mytable( ID INT NOT NULL,
   username VARCHAR(16) NOT NULL, UNIQUE [indexName] (username(length)) );
```

(3)MySQL索引类型：主键索引

它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引：

```
1. CREATE TABLE mytable( ID INT NOT NULL, username VARCHAR(16) NOT NULL, PR
   IMARY KEY(ID) );
```

当然也可以用 ALTER 命令。记住：一个表只能有一个主键。

(4)组合索引

为了形象地对比单列索引和组合索引，为表添加多个字段：

```
1. CREATE TABLE mytable( ID INT NOT NULL, username
```

VARCHAR(16) NOT NULL, city VARCHAR(50) NOT NULL, age INT NOT NULL);

为了进一步榨取MySQL的效率，就要考虑建立组合索引。就是将 name, city, age建到一个索引里：

1. ALTER TABLE mytable ADD INDEX name_city_age (name(10),city,age);

建表时，usernname长度为 16，这里用 10。**这是因为一般情况下名字的长度不会超过10，这样会加速索引查询速度，还会减少索引文件的大小，提高INSERT的更新速度。**

如果分别在 usernname，city，age上建立单列索引，让该表有3个单列索引，查询时和上述的组合索引效率也会大不一样，远远低于我们的组合索引。虽然此时有了三个索引，但MySQL只能用到其中的那个它认为似乎是最有效率的单列索引。

建立这样的组合索引，其实是相当于分别建立了下面三组组合索引：

1. usernname , city, age usernname , city usernname

以上的相关内容就是对MySQL索引类型的部分内容的介绍，望你能有所收获。

使用索引的注意事项

使用索引时，有以下一些技巧和注意事项：

索引不会包含有NULL值的列

只要列中包含有NULL值都将不会被包含在MySQL索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

索引列排序

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

like语句操作

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%”

不会使用MySQL索引而like “aaa%”可以使用索引。

不要在列上进行运算

1. select * from users where YEAR(adddate)<2007;

将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成

1. select * from users where adddate<‘2007-01-01’;

不使用NOT IN和<>操作

索引优化 http://blog.codinglabs.org/articles/theory-of-mysql-index.html

最左前缀原理与相关优化

高效使用索引的首要条件是知道什么样的查询会使用到索引，这个问题和B+Tree中的“最左前缀原理”有关，下面通过例子说明最左前缀原理。

这里先说一下联合索引的概念。在上文中，我们都是假设索引只引用了单个的列，实际上，MySQL中的索引可以以一定顺序引用多个列，这种索引叫做联合索引，一般的，一个联合索引是一个有序元组<a1, a2, …, an>，其中各个元素均为数据表的一列，实际上要严格定义索引需要用到关系代数，但是这里我不想讨论太多关系代数的话题，因为那样会显得很枯燥，所以这里就不再做严格定义。另外，单列索引可以看成联合索引元素数为1的特例。

以employees.titles表为例，下面先查看其上都有哪些索引：

```
SHOW INDEX FROM employees.titles;
+--------+------------+----------+--------------+-------------+--
---------+-------------+------+------------+
| Table  | Non_unique | Key_name | Seq_in_index | Column_name |
Collation | Cardinality | Null | Index_type |
+--------+------------+----------+--------------+-------------+--
---------+-------------+------+------------+
| titles |          0 | PRIMARY  |            1 | emp_no      | A
|         NULL |       | BTREE      |
| titles |          0 | PRIMARY  |            2 | title       | A
|         NULL |       | BTREE      |
```

```
| titles |          0 | PRIMARY  |            3 | from_date   | A
|      443308 |      | BTREE      |
| titles |          1 | emp_no   |            1 | emp_no      | A
|      443308 |      | BTREE      |
+--------+------------+----------+--------------+-------------+--
---------+------------+------+------------+
```

从结果中可以到titles表的主索引为<emp_no, title, from_date>，还有一个辅助索引<emp_no>。为了避免多个索引使事情变复杂（MySQL的SQL优化器在多索引时行为比较复杂），这里我们将辅助索引drop掉：

```
ALTER TABLE employees.titles DROP INDEX emp_no;
```

这样就可以专心分析索引PRIMARY的行为了。
情况一：全列匹配。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND
title='Senior Engineer' AND from_date='1986-06-26';
+----+-------------+--------+-------+---------------+---------+--
-------+-------------------+------+-------+
| id | select_type | table  | type  | possible_keys | key     |
key_len | ref               | rows | Extra |
+----+-------------+--------+-------+---------------+---------+--
-------+-------------------+------+-------+
|  1 | SIMPLE      | titles | const | PRIMARY       | PRIMARY |
59      | const,const,const |    1 |       |
+----+-------------+--------+-------+---------------+---------+--
-------+-------------------+------+-------+
```

很明显，当按照索引中所有列进行精确匹配（这里精确匹配指“=”或“IN”匹配）时，索引可以被用到。这里有一点需要注意，理论上索引对顺序是敏感的，但是由于MySQL的查询优化器会自动调整where子句的条件顺序以使用适合的索引，例如我们将where中的条件顺序颠倒：

```
EXPLAIN SELECT * FROM employees.titles WHERE from_date='1986-06-
26' AND emp_no='10001' AND title='Senior Engineer';
+----+-------------+--------+-------+---------------+---------+--
-------+-------------------+------+-------+
```

```
| id | select_type | table  | type  | possible_keys | key     |
key_len | ref               | rows | Extra |
+----+-------------+--------+-------+---------------+---------+--
-------+-------------------+------+-------+
|  1 | SIMPLE      | titles | const | PRIMARY       | PRIMARY |
59      | const,const,const |    1 |       |
+----+-------------+--------+-------+---------------+---------+--
-------+-------------------+------+-------+
```

效果是一样的。
情况二：最左前缀匹配。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001';
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------+
| id | select_type | table  | type | possible_keys | key     |
key_len | ref   | rows | Extra |
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------+
|  1 | SIMPLE      | titles | ref  | PRIMARY       | PRIMARY | 4
| const |    1 |       |
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------+
```

当查询条件精确匹配索引的左边连续一个或几个列时，如<emp_no>或<emp_no, title>，所以可以被用到，但是只能用到一部分，即条件所组成的最左前缀。上面的查询从分析结果看用到了PRIMARY索引，但是key_len为4，说明只用到了索引的第一列前缀。
情况三：查询条件用到了索引中列的精确匹配，但是中间某个条件未提供。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND
from_date='1986-06-26';
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------------+
| id | select_type | table  | type | possible_keys | key     |
key_len | ref   | rows | Extra       |
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------------+
```

```
|  1 | SIMPLE      | titles | ref  | PRIMARY       | PRIMARY | 4
| const |    1 | Using where |
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------------+
```

此时索引使用情况和情况二相同，因为title未提供，所以查询只用到了索引的第一列，而后面的from_date虽然也在索引中，但是由于title不存在而无法和左前缀连接，因此需要对结果进行扫描过滤from_date（这里由于emp_no唯一，所以不存在扫描）。如果想让from_date也使用索引而不是where过滤，可以增加一个辅助索引<emp_no, from_date>，此时上面的查询会使用这个索引。除此之外，还可以使用一种称之为“隔离列”的优化方法，将emp_no与from_date之间的“坑”填上。

首先我们看下title一共有几种不同的值：

```
SELECT DISTINCT(title) FROM employees.titles;
+--------------------+
| title              |
+--------------------+
| Senior Engineer    |
| Staff              |
| Engineer           |
| Senior Staff       |
| Assistant Engineer |
| Technique Leader   |
| Manager            |
+--------------------+
```

只有7种。在这种成为“坑”的列值比较少的情况下，可以考虑用“IN”来填补这个“坑”从而形成最左前缀：

```
EXPLAIN SELECT * FROM employees.titles
WHERE emp_no='10001'
AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior
Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
AND from_date='1986-06-26';
+----+-------------+--------+-------+---------------+---------+--
------+------+------+-------------+
| id | select_type | table  | type  | possible_keys | key     |
```

```
key_len | ref  | rows | Extra       |
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
|  1 | SIMPLE      | titles | range | PRIMARY       | PRIMARY |
59      | NULL |    7 | Using where |
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
```

这次key_len为59，说明索引被用全了，但是从type和rows看出IN实际上执行了一个range查询，这里检查了7个key。看下两种查询的性能比较：

```
SHOW PROFILES;
+----------+------------+-----------------------------------------
---------------------------------------+
| Query_ID | Duration   | Query
|
+----------+------------+-----------------------------------------
---------------------------------------+
|       10 | 0.00058000 | SELECT * FROM employees.titles WHERE
emp_no='10001' AND from_date='1986-06-26'|
|       11 | 0.00052500 | SELECT * FROM employees.titles WHERE
emp_no='10001' AND title IN ...          |
+----------+------------+-----------------------------------------
---------------------------------------+
```

“填坑”后性能提升了一点。如果经过emp_no筛选后余下很多数据，则后者性能优势会更加明显。当然，如果title的值很多，用填坑就不合适了，必须建立辅助索引。
情况四：查询条件没有指定索引第一列。

```
EXPLAIN SELECT * FROM employees.titles WHERE from_date='1986-06-
26';
+----+-------------+--------+------+---------------+------+------
---+------+--------+-------------+
| id | select_type | table  | type | possible_keys | key  |
key_len | ref  | rows   | Extra       |
+----+-------------+--------+------+---------------+------+------
---+------+--------+-------------+
|  1 | SIMPLE      | titles | ALL  | NULL          | NULL | NULL
```

```
| NULL | 443308 | Using where |
+----+-------------+--------+------+---------------+------+------
---+------+--------+-------------+
```

由于不是最左前缀，索引这样的查询显然用不到索引。
情况五：匹配某列的前缀字符串。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND
title LIKE 'Senior%';
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
| id | select_type | table  | type  | possible_keys | key     |
key_len | ref  | rows | Extra       |
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
|  1 | SIMPLE      | titles | range | PRIMARY       | PRIMARY |
56      | NULL |    1 | Using where |
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
```

此时可以用到索引，但是如果通配符不是只出现在末尾，则无法使用索引。（原文表述有误，如果通配符%不出现在开头，则可以用到索引，但根据具体情况不同可能只会用其中一个前缀）
情况六：范围查询。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no < '10010' and
title='Senior Engineer';
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
| id | select_type | table  | type  | possible_keys | key     |
key_len | ref  | rows | Extra       |
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
|  1 | SIMPLE      | titles | range | PRIMARY       | PRIMARY | 4
| NULL |   16 | Using where |
+----+-------------+--------+-------+---------------+---------+--
-------+------+------+-------------+
```

范围列可以用到索引（必须是最左前缀），但是范围列后面的列无法用到索引。同时，索引最多用于一个范围列，因此如果查询条件中有两个范围列则无法全用到索引。

```
EXPLAIN SELECT * FROM employees.titles
WHERE emp_no < '10010'
AND title='Senior Engineer'
AND from_date BETWEEN '1986-01-01' AND '1986-12-31';
+----+-------------+--------+-------+---------------+---------+---
-------+------+------+-------------+
| id | select_type | table  | type  | possible_keys | key     |
key_len | ref  | rows | Extra       |
+----+-------------+--------+-------+---------------+---------+---
-------+------+------+-------------+
|  1 | SIMPLE      | titles | range | PRIMARY       | PRIMARY | 4
| NULL |   16 | Using where |
+----+-------------+--------+-------+---------------+---------+---
-------+------+------+-------------+
```

可以看到索引对第二个范围索引无能为力。这里特别要说明MySQL一个有意思的地方，那就是仅用explain可能无法区分范围索引和多值匹配，因为在type中这两者都显示为range。同时，用了“between”并不意味着就是范围查询，例如下面的查询：

```
EXPLAIN SELECT * FROM employees.titles
WHERE emp_no BETWEEN '10001' AND '10010'
AND title='Senior Engineer'
AND from_date BETWEEN '1986-01-01' AND '1986-12-31';
+----+-------------+--------+-------+---------------+---------+---
-------+------+------+-------------+
| id | select_type | table  | type  | possible_keys | key     |
key_len | ref  | rows | Extra       |
+----+-------------+--------+-------+---------------+---------+---
-------+------+------+-------------+
|  1 | SIMPLE      | titles | range | PRIMARY       | PRIMARY |
59      | NULL |   16 | Using where |
+----+-------------+--------+-------+---------------+---------+---
-------+------+------+-------------+
```

看起来是用了两个范围查询，但作用于emp_no上的“BETWEEN”实际上相当于“IN”，也

就是说emp_no实际是多值精确匹配。可以看到这个查询用到了索引全部三个列。因此在MySQL中要谨慎地区分多值匹配和范围匹配，否则会对MySQL的行为产生困惑。
情况七：查询条件中含有函数或表达式。

很不幸，如果查询条件中含有函数或表达式，则MySQL不会为这列使用索引（虽然某些在数学意义上可以使用）。例如：

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND
left(title, 6)='Senior';
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------------+
| id | select_type | table  | type | possible_keys | key     |
key_len | ref   | rows | Extra       |
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------------+
|  1 | SIMPLE      | titles | ref  | PRIMARY       | PRIMARY | 4
| const |    1 | Using where |
+----+-------------+--------+------+---------------+---------+---
------+-------+------+-------------+
```

虽然这个查询和情况五中功能相同，但是由于使用了函数left，则无法为title列应用索引，而情况五中用LIKE则可以。再如：

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no - 1='10000';
+----+-------------+--------+------+---------------+------+------
---+------+--------+-------------+
| id | select_type | table  | type | possible_keys | key  |
key_len | ref  | rows   | Extra       |
+----+-------------+--------+------+---------------+------+------
---+------+--------+-------------+
|  1 | SIMPLE      | titles | ALL  | NULL          | NULL | NULL
| NULL | 443308 | Using where |
+----+-------------+--------+------+---------------+------+------
---+------+--------+-------------+
```

显然这个查询等价于查询emp_no为10001的函数，但是由于查询条件是一个表达式，MySQL无法为其使用索引。看来MySQL还没有智能到自动优化常量表达式的程度，因此在写查询语句时尽量避免表达式出现在查询中，而是先手工私下代数运算，转换为无表达式

的查询语句。
索引选择性与前缀索引

既然索引可以加快查询速度，那么是不是只要是查询语句需要，就建上索引？答案是否定的。因为索引虽然加快了查询速度，但索引也是有代价的：索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，MySQL在运行时也要消耗资源维护索引，因此索引并不是越多越好。一般两种情况下不建议建索引。

第一种情况是表记录比较少，例如一两千条甚至只有几百条记录的表，没必要建索引，让查询做全表扫描就好了。至于多少条记录才算多，这个个人有个人的看法，我个人的经验是以2000作为分界线，记录数不超过 2000可以考虑不建索引，超过2000条可以酌情考虑索引。

另一种不建议建索引的情况是索引的选择性较低。所谓索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数（#T）的比值：

```
Index Selectivity = Cardinality / #T
```

显然选择性的取值范围为(0, 1]，选择性越高的索引价值越大，这是由B+Tree的性质决定的。例如，上文用到的employees.titles表，如果title字段经常被单独查询，是否需要建索引，我们看一下它的选择性：

```
SELECT count(DISTINCT(title))/count(*) AS Selectivity FROM
employees.titles;
+-------------+
| Selectivity |
+-------------+
|      0.0000 |
+-------------+
```

title的选择性不足0.0001（精确值为0.00001579），所以实在没有什么必要为其单独建索引。

有一种与索引选择性有关的索引优化策略叫做前缀索引，就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。下面以employees.employees表为例介绍前缀索引的选择和使用。

从图12可以看到employees表只有一个索引<emp_no>，那么如果我们想按名字搜索一个人，就只能全表扫描了：

```
EXPLAIN SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido';
+----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+
| id | select_type | table     | type | possible_keys | key  | key_len | ref  | rows   | Extra       |
+----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+
|  1 | SIMPLE      | employees | ALL  | NULL          | NULL | NULL    | NULL | 300024 | Using where |
+----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+
```

如果频繁按名字搜索员工，这样显然效率很低，因此我们可以考虑建索引。有两种选择，建<first_name>或<first_name, last_name>，看下两个索引的选择性：

```
SELECT count(DISTINCT(first_name))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.0042 |
+-------------+
SELECT count(DISTINCT(concat(first_name, last_name)))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.9313 |
+-------------+
```

<first_name>显然选择性太低，<first_name, last_name>选择性很好，但是first_name和last_name加起来长度为30，有没有兼顾长度和选择性的办法？可以考虑用first_name和last_name的前几个字符建立索引，例如<first_name,

left(last_name, 3)>，看看其选择性：

```
SELECT count(DISTINCT(concat(first_name, left(last_name,
3))))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.7879 |
+-------------+
```

选择性还不错，但离0.9313还是有点距离，那么把last_name前缀加到4：

```
SELECT count(DISTINCT(concat(first_name, left(last_name,
4))))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.9007 |
+-------------+
```

这时选择性已经很理想了，而这个索引的长度只有18，比<first_name, last_name>短了接近一半，我们把这个前缀索引 建上：

```
ALTER TABLE employees.employees
ADD INDEX `first_name_last_name4` (first_name, last_name(4));
```

此时再执行一遍按名字查询，比较分析一下与建索引前的结果：

```
SHOW PROFILES;
+----------+------------+----------------------------------------
----------------------------------------+
| Query_ID | Duration   | Query
|
+----------+------------+----------------------------------------
----------------------------------------+
|       87 | 0.11941700 | SELECT * FROM employees.employees WHERE
first_name='Eric' AND last_name='Anido' |
|       90 | 0.00092400 | SELECT * FROM employees.employees WHERE
```

```
first_name='Eric' AND last_name='Anido' |
+----------+------------+-----------------------------------------
----------------------------------------+

性能的提升是显著的，查询速度提高了120多倍。

前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于Covering index（即当索引本身包含查询所需全部数据时，不再访问数据文件本身）。
```

补充该节中的"范围查询"说明:

Mysql对于范围查询range分的优化为单字段优化和多元素优化:

单元素索引范围条件的定义如下：

· 对于BTREE和HASH索引，当使用=、<=>、IN、IS NULL或者IS NOT NULL操作符时，关键元素与常量值的比较关系对应一个范围条件,即const范围。

· 对于BTREE索引，当使用>、<、>=、<=、BETWEEN、!=或者<>，或者LIKE 'pattern'(其中 'pattern'不以通配符开始)操作符时，关键元素与常量值的比较关系对应一个范围条件。

· 对于所有类型的索引，多个范围条件结合OR或AND则产生一个范围条件。

前面描述的“常量值”系指：· 查询字符串中的常量· 同一联接中的const或system表中的列· 无关联子查询的结果· 完全从前面类型的子表达式组成的表达式多元素索引的范围条件:

1.----------

对于BTREE索引，区间可以对结合AND的条件有用，其中每个条件用一个常量值通过=、<=>、IS NULL、>、<、>=、<=、!=、<>、BETWEEN或者LIKE 'pattern' (其中 'pattern'不以通配符开头)比较一个关键元素。区间可以足够长以确定一个包含所有匹配条件(或如果使用<>或!=，为两个区间)的记录的单一的关键元组。例如，对于条件：

key_part1 = 'foo' AND key_part2 >= 10 AND key_part3 > 10

2.----------

对于HASH索引，可以使用包含相同值的每个区间。

key_part1 cmp const1 AND key_part2 cmp const2

AND ... AND key_partN cmp constN;

这里，const1，const2，...为常量，cmp是=、<=>或者IS NULL比较操作符之一，条件包括所有索引部分。(也就是说，有N 个条件，每一个对应N-元素索引的每个部

分）。

3. ----------

如果包含区间内的一系列记录的条件结合使用OR，则形成包括一系列包含在区间并集的记录的一个条件。如果条件结合使用了AND，则形成包括一系列包含在区间交集内的记录的一个条件。例如，对于两部分索引的条件：

(key_part1 = 1 AND key_part2 < 2) OR (key_part1 > 5)

区间为：

(1, -inf) < (key_part1, key_part2) < (1, 2)

(5, -inf) < (key_part1, key_part2)

Mysql检索时间查询 (版本要求: 5.0.37或以上)

```
 开启profile
1    mysql> set profiling=1;
2    Query OK, 0 rows affected (0.00 sec)
eg：
1    mysql> select * from test_1;
2    mysql> show profiles;
3    +----------+------------+----------------------+
4    | Query_ID | Duration   | Query                |
5    +----------+------------+----------------------+
6    |        1 | 0.84718100 | select * from test_1 |
7    +----------+------------+----------------------+
8    1 row in set (0.00 sec)
```