Java MySQL系列课程05

------ MySQL优化之SQL和索引的优化

本文档包含以下内容：

1. 索引的执行分析
2. 索引失效
3. SQL和索引的优化
4. 优化总结

#### 前言：

**“索引什么时候会用到？”**

**“索引什么时候会失效？”**

**“索引是不是创建的越多越好？”**

**“索引的底层实现是什么？”**

**“多表联合查询怎么查？”**

**“你做过哪些SQL和索引的优化？”**

本章应该是面试中最复杂的一部分内容了，和大家平时做项目，积累的SQL优化、索引优化相关，一般项目在后期才做优化，或者到满足不了用户需求的时候再做优化，因此同学们学习过程中很少碰到需要优化的数据库的场景，但是既然作为考察的重点问题，还是要掌握一些优化的手段。在校招中有关MySQL优化的问题很多，这一章先来看看哪些情况下需要优化SQL和索引，以及具体该怎么去优化。

#### 一 索引的**执行**分析：

先来看看，一般我们做单表查询和多表联合查询是怎么使用索引的。

**1）单表查询索引的执行过程以及优化 - 普通查询**

**我们创建过一张student学生表**，如下：

mysql> **desc student;**

+-------+-------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+-------------+------+-----+---------+-------+

| id | int(11) | NO | PRI | NULL | |

| name | varchar(20) | YES | MUL | NULL | |

| score | double | YES | | 0 | |

+-------+-------------+------+-----+---------+-------+

3 rows in set (0.00 sec)

mysql> **select \* from student;**

+----+-------+-------+

| id | name | score |

+----+-------+-------+

| 1 | zhang | 60.5 |

| 2 | gao | 78.6 |

| 3 | liu | 89.2 |

| 4 | wang | 95.6 |

+----+-------+-------+

4 rows in set (0.00 sec)

可以看到student表里面有4条内容。

**当我们用id作为过滤条件进行查询的时候（注意id是主键）**，如下：

mysql> **select \* from student where id=3;**

+----+------+-------+

| id | name | score |

+----+------+-------+

| 3 | liu | 89.2 |

+----+------+-------+

1 row in set (0.00 sec)可以查询出一条记录。

**我们用explain分析一下上面SQL语句的执行计划**，如下：

mysql> **explain select \* from student where id=3\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: const

**possible\_keys: PRIMARY**

**key: PRIMARY**

key\_len: 4

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

1 row in set, 1 warning (0.00 sec)

可以看到，表里面虽然有4行数据，但是找到id=3的学生信息，**只查询了一行就找到了，因为使用了PRIARY主键索引（InnoDB会自动给主键字段创建主键索引树）**，根据前面学习的内容可知，当我们用id作为where子句的过滤条件时，首先从主键B+索引树上快速找到id，**InnoDB采用聚集索引（索引关键字和数据在一起存放）**，**因此找到id，就找到这一行的数据**，不用把整个表扫描一遍，效率很高。

**那如果是下面的查询呢？**

mysql> **select \* from student where name='liu';**

+----+------+-------+

| id | name | score |

+----+------+-------+

| 3 | liu | 89.2 |

+----+------+-------+

1 row in set (0.00 sec)

上面的SQL根据name字段查询信息，我们之前给该name字段创建过辅助索引，名字是name\_index。

**用explain分析一下上面的SQL执行过程**，如下：

mysql> **explain select \* from student where name='liu'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: ref

**possible\_keys: name\_index**

**key: name\_index**

key\_len: 63

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

1 row in set, 1 warning (0.00 sec)

从上面可以看到，根据name字段查询学生信息的时候，**用到了name\_index辅助索引**，还记得吗？**InnoDB的辅助索引叶子节点存储的是辅助索引字段值和对应该行记录的主键（此处就是存了name,id）**，也就是上面的这个SQL语句，**先查询name字段的辅助索引B+树，找到name=’liu’的节点，获取对应行的主键id=3，然后拿id=3在主键索引树去寻找数据**。

**可以看到，上面根据name查询学生信息，总共搜索了两次B+索引树，那么下面的这个SQL语句呢？**

mysql> **explain select id from student where name='liu'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: ref

**possible\_keys: name\_index**

**key: name\_index**

key\_len: 63

ref: const

**rows: 1**

filtered: 100.00

Extra: Using index

1 row in set, 1 warning (0.00 sec)

上面的**SQL使用到了name\_index辅助索引**，在辅助索引树上根据name找到对应行的主键，**由于上面的select语句选择的就是id，因此在辅助索引树上就已经得到了，不用再去主键索引树上去搜索了**。

**再变换一下上面的SQL语句，**如下：

mysql> **explain select score from student where name='liu'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: ref

**possible\_keys: name\_index**

**key: name\_index**

key\_len: 63

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

1 row in set, 1 warning (0.00 sec)

上面的SQL是根据name查询成绩score，由于name字段上创建了辅助索引，因此**首先根据name=’liu’查询辅助索引树，找到对应的行主键id=3**，由于SQL上需要选择score，**为了得到score字段的值，还需要拿着id=3去主键索引树上去搜索这一行记录，然后拿到score。**所以，知道了索引的执行过程，现在你知道怎么优化上面这个SQL语句吗？

对了，如果SQL上经常根据name查询score，那么**可以创建一个（name，score）的联合索引**，再进行上面的SQL查询时（**select score from student where name='liu'**），**根据name=’liu’直接查询辅助索引树，就找到score了**，**不用再去查找主键索引树了（此时辅助索引树的叶子节点存储的是name,score,id），性能提高**！但是注意，由于用name作为where的过滤条件，创建联合索引的时候，**name列在前，score列在后**。

**2）单表查询索引的执行过程以及优化 - 普通查询+排序或者分组**

**根据上面的例子，对于索引的执行过程应该有一些深入的了解了，那么继续看看下边的例子：**

生成一个订单表，包含用户id，商品id，以及入购物车日期，如下：

mysql> **create table orderlist(userid int not null,**

**-> productid int not null,**

**-> date timestamp);**

Query OK, 0 rows affected (0.56 sec)

**给表中添加一组数据，如下：**

mysql> **select \* from orderlist;**

+--------+-----------+---------------------+

| userid | productid | date |

+--------+-----------+---------------------+

| 1 | 1 | 2018-08-31 10:52:46 |

| 2 | 1 | 2018-08-30 15:32:56 |

| 2 | 5 | 2018-07-30 18:12:56 |

| 1 | 2 | 2018-07-25 21:38:21 |

| 1 | 3 | 2018-07-28 13:31:24 |

| 1 | 9 | 2018-07-29 17:30:20 |

+--------+-----------+---------------------+

6 rows in set (0.00 sec)

**执行下面的SQL查询语句：**

mysql> **select \* from orderlist where userid=1 order by date;**

+--------+-----------+---------------------+

| userid | productid | date |

+--------+-----------+---------------------+

| 1 | 2 | 2018-07-25 21:38:21 |

| 1 | 3 | 2018-07-28 13:31:24 |

| 1 | 9 | 2018-07-29 17:30:20 |

| 1 | 1 | 2018-08-31 10:52:46 |

+--------+-----------+---------------------+

4 rows in set (0.00 sec)

可以看到，上面是查询id号位1的用户的购物信息，按日期升序排序。

**用explain分析一下上面SQL的执行计划，如下：**

mysql> **explain select \* from orderlist where userid=1 order by date\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: orderlist

partitions: NULL

type: ALL

**possible\_keys: NULL**

**key: NULL**

key\_len: NULL

ref: NULL

**rows: 6**

filtered: 16.67

Extra: Using where; **Using filesort**

1 row in set, 1 warning (0.00 sec)

根据explain分析的结果，可以看到，上面的SQL查询做了整表查询，效率相当低，由于查询完userid=1的所有行记录后，还要按date字段进行升序排序，所以出现了**Using filesort，这涉及文件排序，效率相当低，一般遇见Using filesort一定要进行优化！**

**那么给userid和date分别创建索引怎么样，如下：**

mysql> **create index userid\_index on orderlist (userid);**

Query OK, 0 rows affected (0.82 sec)

Records: 0 Duplicates: 0 Warnings: 0

mysql> **create index date\_index on orderlist (date);**

Query OK, 0 rows affected (0.63 sec)

Records: 0 Duplicates: 0 Warnings: 0

重复上面的explain查询计划，看看和上面有什么不同：

mysql> **explain select \* from orderlist force index(userid\_index) where userid=1 order by date\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: orderlist

partitions: NULL

type: ref

**possible\_keys: userid\_index**

**key: userid\_index**

key\_len: 4

ref: const

**rows: 4**

filtered: 100.00

Extra: Using index condition; **Using filesort**

1 row in set, 1 warning (0.00 sec)

可见这次用到了索引，但是只用到了userid\_index，**但是还是出现了Using filesort**，**上面的SQL是从查询userid的辅助索引树，找到主键id，然后再拿这些主键id，到主键索引树上去搜索数据，搜索到了以后，需要根据date字段进行文件排序，效率还是很低**！

**force index(userid\_index)**这是什么？这是**强制索引**，为什么要使用强制索引呢，这是MySQL在5.6版本后加入的新特性，SQL优化器会在索引存在的情况下，**通过符合RANGE范围的条数和总数的比例来选择是使用索引还是进行全表遍历，**MySQL Server认为使用索引，还不如整表搜索来得快，它就不会使用索引了，**强制索引是一定会使用指定的索引的**。

大家可能会问，既然我们给date也创建了索引，那为什么还要对date进行文件排序**Using filesort呢？要注意，一个SQL查询一张表，一次只能使用一个索引，因此userid\_index的索引使用过了，date的索引就使用不到了！**

**继续优化，根据上面的分析，我们删除userid和date的索引，创建一个联合索引，由于userid是where的过滤条件，因此联合索引userid在前，date在后，如下：**

mysql> **create index userid\_date\_index on orderlist (userid, date);**

Query OK, 0 rows affected (0.54 sec)

Records: 0 Duplicates: 0 Warnings: 0

**继续做上面的explain分析，如下：**

mysql> **explain select \* from orderlist force index(userid\_date\_index) where userid=1 order by date\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: orderlist

partitions: NULL

type: ref

**possible\_keys: userid\_date\_index**

**key: userid\_date\_index**

key\_len: 4

ref: const

**rows: 4**

filtered: 100.00

Extra: Using index condition

1 row in set, 1 warning (0.00 sec)

我们给userid和date创建联合索引后，就不会再出现using filesort了，**根据userid=1查询辅助索引树，找到的数据也已经按date排序好了**，然后在去主键索引树上去整行数据就完了，效率很高！

之所以使用force index（userid\_date\_index），是防止SQL优化器优化索引的使用，因为我们的测试数据量较少，有时候使用索引不一定比整表搜索快！

如果上面的SQL只是select date，那么就不用再往主键索引树上查找了，在辅助索引树上就能找到date，**所以select后面选择哪些列也是有讲究的**。

1. **多表查询-连接查询索引的执行过程以及优化**

**重新举一个例子，我们创建两张表，分别是runoob\_tbl和tcount\_tbl，如下：**

mysql> **select \* from runoob\_tbl;**

+-----------+---------------+---------------+-----------------+

| runoob\_id | runoob\_title | runoob\_author | submission\_date |

+-----------+---------------+---------------+-----------------+

| 1 | 学习 PHP | 菜鸟教程 | 2017-04-12 |

| 2 | 学习 MySQL | 菜鸟教程 | 2017-04-12 |

| 3 | 学习 Java | RUNOOB.COM | 2015-05-01 |

| 4 | 学习 Python | 谭浩强 | 2016-03-06 |

| 5 | 学习 C | FK | 2017-04-05 |

| 6 | 学习 C++ | 234 | 2017-04-05 |

+-----------+---------------+---------------+-----------------+

6 rows in set (0.00 sec)

mysql> **select \* from tcount\_tbl;**

+---------------+--------------+

| runoob\_author | runoob\_count |

+---------------+--------------+

| 菜鸟教程 | 11 |

| 学习 C | 12 |

+---------------+--------------+

2 rows in set (0.00 sec)

上面的两张表，**都给runoob\_author字段创建了辅助索引**！

**那么考虑一下，下面这个SQL语句，是怎么使用索引的**，如下：

mysql> **select \* from runoob\_tbl a inner join tcount\_tbl b on a.runoob\_author = b.runoob\_author;**

+-----------+--------------+---------------+-----------------+---------------+--------------+

| runoob\_id | runoob\_title | runoob\_author | submission\_date | runoob\_author | runoob\_count |

+-----------+--------------+---------------+-----------------+---------------+--------------+

| 1 | 学习 PHP | 菜鸟教程 | 2017-04-12 | 菜鸟教程 | 11 |

| 2 | 学习 MySQL| 菜鸟教程 | 2017-04-12 | 菜鸟教程 | 11 |

+-----------+--------------+---------------+-----------------+---------------+--------------+

在使用多表连接查询的时候，MySQL会首先判断**runoob\_tbl（别名表a）和tcount\_tbl（别名表b）**哪个表小，**这里表小主要指的是行数少，很显然b表小，MySQL会对b表进行整表遍历，然后在a表上根据runoob\_author 字段进行查询**，所以b表就是小表，无论如何都是要整表遍历的，是使用不到索引的，但是大表a表的**runoob\_author** 字段创建索引，就能使用到了！

所在在连接查询的时候，**小表总是要整表搜索的，建索引没有用，大表创建索引是能提高查询效率的，小表决定查询次数，大表决定查询时间**！

**用explain分析上面SQL的执行计划，**如下：

mysql> **explain select \* from runoob\_tbl a inner join tcount\_tbl b on a.runoob\_author = b.runoob\_author\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: b **小表**

partitions: NULL

type: ALL

**possible\_keys: tcountt\_author\_name\_index**

**key: NULL 没有用到小表的索引**

key\_len: NULL

ref: NULL

rows: 2

filtered: 100.00

Extra: NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: a

partitions: NULL

type: ref

**possible\_keys: runoob\_author\_name\_index**

**key: runoob\_author\_name\_index 大表的索引用到了**

key\_len: 122

ref: example.b.runoob\_author

**rows: 1**

filtered: 100.00

Extra: Using index condition

2 rows in set, 1 warning (0.00 sec)

从上面的explain结果可以很清楚的看到，先整表搜索了b表，然后在a表的辅助索引树上查找！**b表没用到索引，a表用到了索引**。

**给上面的连接查询添加where条件过滤子句**，如下：

mysql> **explain select \* from runoob\_tbl a inner join tcount\_tbl b on a.runoob\_author = b.runoob\_author where a.runoob\_author='FK'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: a

partitions: NULL

type: ref

**possible\_keys: runoob\_author\_name\_index**

**key: runoob\_author\_name\_index 这里用所以主要是因为where条件过滤子句**

key\_len: 122

ref: const

rows: 1

filtered: 100.00

Extra: NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

**table: b 这次b表成大表了，因为a表根据where条件过滤后，没有b表多，a成小表了**

partitions: NULL

type: ref

**possible\_keys: tcountt\_author\_name\_index**

**key: tcountt\_author\_name\_index**

key\_len: 767

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

2 rows in set, 1 warning (0.00 sec)

从上面可以看到，a表用**where a.runoob\_author='FK'**作为条件滤过，由于a表的runoob\_author创建了索引，因此a表使用了索引进行条件过滤，**过滤完就1行数据，所以这次a成小表，b成大表了，b的索引就能够使用上**，如上面的信息所示，大家仔细分析一下！

所以，**在连接查询中，大表小表的角色是不一定的，没有where子句，那么就按表的行数来定，如果有where子句，那么就按条件过滤完的行数来定大小表**！

1. **多表查询-连接查询代替子查询**

子查询在多表联合查询时的效率是很低的，因为要产生中间表，多表查询最好优化成连接查询，效率很高！

**比如拿下面的两张表做子查询操作：**

mysql> **select \* from student;**

+----+-------+

| id | name |

+----+-------+

| 1 | zhang |

| 2 | gao |

| 3 | liu |

| 4 | wang |

| 5 | wei |

| 6 | liang |

| 7 | yang |

| 8 | hu |

+----+-------+

8 rows in set (0.00 sec)

mysql> **select \* from grade;**

+--------+---------+

| userid | average |

+--------+---------+

| 1 | 80.5 |

| 2 | 90.6 |

| 3 | 55.5 |

+--------+---------+

3 rows in set (0.00 sec)

mysql> **explain select \* from student where id in (select userid from grade where average>60.0)\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

**table: <subquery2> 子查询有中间表的产生和销毁，效率低**

partitions: NULL

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: NULL

filtered: 100.00

Extra: Using where

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: eq\_ref

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 4

ref: <subquery2>.userid

rows: 1

filtered: 100.00

Extra: NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 2

select\_type: MATERIALIZED

table: grade

partitions: NULL

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 3

filtered: 33.33

Extra: Using where

3 rows in set, 1 warning (0.00 sec)

**上面的子查询会产生中间表，效率没有连接查询的效率高，把子查询优化成连接查询**，如下：

mysql> **explain select \* from student a inner join grade b on a.id = b.userid where b.average>60.0\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: b

partitions: NULL

type: ALL

**possible\_keys: NULL**

**key: NULL where子句过滤b表的average，因此给b表average字段建立联合索引会提高查询效率！**

key\_len: NULL **大家可以自行创建测试**

ref: NULL

rows: 3

filtered: 33.33

Extra: Using where

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: a

partitions: NULL

type: eq\_ref

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 4

ref: usermanager.b.userid

rows: 1

filtered: 100.00

Extra: NULL

2 rows in set, 1 warning (0.00 sec)

**可以看到，连接查询不会产生中间表！**

#### 二 索引失效：

上面第一部分列举了很多SQL语句，并分析了索引是怎么应用，怎么执行的，对于单表查询和多表连接查询中，索引的使用一定要牢牢掌握，这一部分看看哪些情况下，索引会失效！

1. **like后面的通配符在前面，索引会失效。**

mysql> **select \* from runoob\_tbl;**

+-----------+---------------+---------------+-----------------+

| runoob\_id | runoob\_title | runoob\_author | submission\_date |

+-----------+---------------+---------------+-----------------+

| 1 | 学习 PHP | 菜鸟教程 | 2017-04-12 |

| 2 | 学习 MySQL | 菜鸟教程 | 2017-04-12 |

| 3 | 学习 Java | RUNOOB.COM | 2015-05-01 |

| 4 | 学习 Python | 谭浩强 | 2016-03-06 |

| 5 | 学习 C | FK | 2017-04-05 |

| 6 | 学习 C++ | 234 | 2017-04-05 |

+-----------+---------------+---------------+-----------------+

runoob\_author字段已经创建过索引

mysql> **explain select \* from runoob\_tbl where runoob\_author like 'F%'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: runoob\_tbl

partitions: NULL

type: range

**possible\_keys: runoob\_author\_name\_index**

**key: runoob\_author\_name\_index**

key\_len: 122

ref: NULL

**rows: 1**

filtered: 100.00

Extra: Using index condition

1 row in set, 1 warning (0.00 sec)

**能用到索引！**

mysql> **explain select \* from runoob\_tbl where runoob\_author like '%k'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: runoob\_tbl

partitions: NULL

type: ALL

**possible\_keys: NULL**

**key: NULL**

key\_len: NULL

ref: NULL

**rows: 6**

filtered: 16.67

Extra: Using where

1 row in set, 1 warning (0.00 sec)

**用不到索引！**

1. **没有使用联合索引的第一列，not in，!=，使用MySQL函数，类型转换，or等都无法用到索引。**

mysql> **select \* from runoob\_tbl;**

+-----------+---------------+---------------+-----------------+

| runoob\_id | runoob\_title | runoob\_author | submission\_date |

+-----------+---------------+---------------+-----------------+

| 1 | 学习 PHP | 菜鸟教程 | 2017-04-12 |

| 2 | 学习 MySQL | 菜鸟教程 | 2017-04-12 |

| 3 | 学习 Java | RUNOOB.COM | 2015-05-01 |

| 4 | 学习 Python | 谭浩强 | 2016-03-06 |

| 5 | 学习 C | FK | 2017-04-05 |

| 6 | 学习 C++ | 234 | 2017-04-05 |

+-----------+---------------+---------------+-----------------+

mysql> **explain select \* from runoob\_tbl where runoob\_author in('FK')\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: runoob\_tbl

partitions: NULL

type: ref

**possible\_keys: runoob\_author\_name\_index**

**key: runoob\_author\_name\_index in使用了索引！**

key\_len: 122

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

1 row in set, 1 warning (0.00 sec)

mysql> **explain select \* from runoob\_tbl where runoob\_author not in('FK')\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: runoob\_tbl

partitions: NULL

type: ALL

**possible\_keys: runoob\_author\_name\_index not in没有使用索引！**

**key: NULL**

key\_len: NULL

ref: NULL

**rows: 6**

filtered: 83.33

Extra: Using where

1 row in set, 1 warning (0.00 sec)

mysql> **explain select \* from runoob\_tbl where runoob\_author = 234\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: runoob\_tbl

partitions: NULL

type: ALL

**possible\_keys: runoob\_author\_name\_index**

**key: NULL 上面输入了234整数，强转成字符串类型，不能使用索引！**

key\_len: NULL

ref: NULL

rows: 6

filtered: 16.67

Extra: Using where

1 row in set, 3 warnings (0.00 sec)

mysql> **explain select \* from runoob\_tbl where runoob\_author = '234'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: runoob\_tbl

partitions: NULL

type: ref

**possible\_keys: runoob\_author\_name\_index**

**key: runoob\_author\_name\_index 上面‘234’用的就是字符串，没有类型转换，可以使用索引！**

key\_len: 122

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

1 row in set, 1 warning (0.00 sec)

**其它的种类能否使用索引，大家可以自行验证！**

#### 三 SQL和索引的优化：

回答诸如**“你对SQL和索引做过哪些优化？”**此类问题的时候，还是要提醒大家，面试过程回答问题，总是要把问题的起因，经过，结果都描述清楚。当然小问题的回答套路会少一些，但是诸如优化类的问题，我们最好从问题的应用场景开始回答，也就是我碰见过什么问题，发现哪些SQL执行效率低了，然后我想着去优化它们，最后具体怎么优化的，不要一上来，就照本宣科的开始背书上的各种优化措施，听起来很假，这样只能给自己挖坑，能不能填的上就看你自己的本事了！

在应用的的开发过程中，由于初期数据量小，开发人员写 SQL 语句时更重视功能上的实现，但是当应用系统正式上线后，随着生产数据量的急剧增长，很多 SQL 语句开始逐渐显露出性能问题，对生产的影响也越来越大，此时这些有问题的 SQL 语句就成为整个系统性能的瓶颈， 因此我们必须要对它们进行优化

我们介绍三种方法，怎么去定位哪些SQL操作效率低，然后根据实际情况再想优化的措施。

1. **慢查询日志（很重要）**

MySQL可以设置慢查询日志，当SQL执行的时间超过我们设定的时间，那么这些SQL就会被记录在慢查询日志当中，然后我们通过查看日志，用explain分析这些SQL的执行计划，来判定为什么效率低下，是没有使用到索引？还是索引本身创建的有问题？或者是索引使用到了，但是由于表的数据量太大，花费的时间就是很长，那么此时我们可以把表分成n个小表，比如订单表按年份分成多个小表等。

慢查询日志相关的参数如下所示：

mysql> **show variables like '%slow\_query%';**

+---------------------+--------------------------+

| Variable\_name | Value |

+---------------------+--------------------------+

| **slow\_query\_log | ON**  | **说明慢查询日志已经打开**

| **slow\_query\_log\_file | XB-20150705TIRN-slow.log | 慢查询日志的名称**

+---------------------+--------------------------+

2 rows in set, 1 warning (0.05 sec)

慢查询日志记录了包含所有执行时间超过参数 long\_query\_time（单位：秒）所设置值的 SQL语句的日志，在MySQL上用命令可以查看，如下：

mysql> **show variables like 'long%';**

+-----------------+-----------+

| Variable\_name | Value |

+-----------------+-----------+

| **long\_query\_time | 10.000000** | **表示超过10秒的SQL都会被记录在慢查询日志当中！**

+-----------------+-----------+

1 row in set, 1 warning (0.01 sec)

这个值当然是可以修改的，如下：

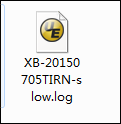
mysql> **set long\_query\_time = 1;**  #注意，单位是秒

Query OK, 0 rows affected (0.00 sec)

现在修改成**超过1秒的SQL都会被记录在慢查询日志当中！可以设置为0.01秒，表示10毫秒。**

慢查询日志，默认名称是host\_name-slow.log，存放在MySQL的数据路径下，在我的电脑上就是这个数据路径：

C:\ProgramData\MySQL\MySQL Server 5.7\Data

这里面存放的就是慢查询日志，内容格式显示大致如下：

# Query\_time: 0.012000 Lock\_time: 0.000000 Rows\_sent: 0 Rows\_examined: 139

use tuluneducation;

SET timestamp=1534527397;

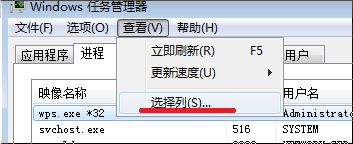
**select id,author from subject where content like '%linux%' and title like '%c++ linux%';**

通过查询慢查询日志，发现项目运行过程中，上面这条SQL语句的执行时间超过了设定的慢查询时间，那么接下来就需要用explain分析一下该SQL的执行计划了，根据具体情况找出SQL和索引该怎么去优化。

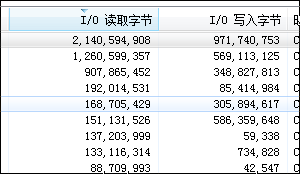
**log\_queries\_not\_using\_indexes**参数设置为ON，可以捕获到所有未使用索引的SQL语句。

1. **查看磁盘I/O读写的数据量**

数据库毕竟是磁盘存储，我们可以通过项目运行过程中，检测磁盘I/O读写的数据量，来定位效率低下的SQL。在windows上打开任务管理器就可以检测，Linux上通过命令可以检测，这里以windows为示例：



选择I/O读写查看列，检测我们Java程序运行过程中，执行哪些SQL时花费的磁盘I/O太大，读写的字节数过多，一般不加索引，或者索引创建不正确没有使用到索引，查询花费的磁盘I/O字节数都会上M，而添加了索引的查询一般消耗在K级别。



1. **show status**

MySQL提供了show status命令，查看MySQL Server的运行参数，可以查看select，insert，delete，update语句的执行频率，慢查询次数，事务的提交和回滚的次数，如下：

mysql> **show status like 'Com\_%';**

**打印的信息比较多，主要关注如下：**

Com\_select：执行 select 操作的次数，一次查询只累加 1。

Com\_insert：执行 INSERT 操作的次数， 对于批量插入的 INSERT 操作， 只累加一次。

Com\_update：执行 UPDATE 操作的次数。

Com\_delete：执行 DELETE 操作的次数。

**针对InnoDB存储引擎的参数：**

**mysql> show status like 'InnoDB\_%';**

Innodb\_rows\_read：select 查询返回的行数。

Innodb\_rows\_inserted：执行 INSERT 操作插入的行数。

Innodb\_rows\_updated：执行 UPDATE 操作更新的行数。

Innodb\_rows\_deleted：执行 DELETE 操作删除的行数。

**还有以下几个参数，同样可以用show status打印查看：**

Connections：试图连接 MySQL 服务器的次数。

Uptime：服务器工作时间。

Slow\_queries：慢查询的次数。

Com\_commit 和 Com\_rollback： 可以了解事务提交和回滚的情况，对于回滚操作非常频繁的数据库，可能意味着应用编写存在问题。

**更多具体的SQL和索引优化措施，请看以下书籍：**

**《MySQL入门很简单》 第18章 《深入浅出MySQL》 第18章 《高性能MySQL》 第6章**

#### 四 优化总结：

**上面的内容已经就SQL和索引的优化做了详细的讲解，那么在这里总结一下，我们常用的优化方式：**

1. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where和order by 涉及的列上建立索引。
2. 注意前面罗列的会使索引失效的那些运算符，这样SQL是无法使用索引的。
3. 应尽量避免在 where 子句中使用or 来连接条件，否则将导致MySQL放弃使用索引而进行全表扫描，如：select id from t where num=10 or num=20可以这样查询：select id from t where num=10 union all select id from t where num=20
4. in和 not in 也要慎用，否则会导致全表扫描，如：select id from t where num in(1,2,3) 对于连续的数值，能用 between 就不要用 in 了：select id from t where num between 1 and 3
5. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where num/2=100应改为:select id from t where num=100\*2
6. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where substring(name,1,3)='abc' ，name以abc开头的id，应改为:select id from t where name like 'abc%'
7. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。
8. 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。
9. 并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用，这个是属于MySQL的SQL优化器对索引的一种优化。
10. 索引并不是越多越好，索引固然可 以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。
11. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。
12. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

**优化措施不能死记硬背，需要在MySQL Server上多实践，逐渐积累优化经验！**