数据库优化案例

目录

[SQL篇： 2](#_Toc312166142)

[mysql子查询： 2](#_Toc312166143)

[原理介绍： 2](#_Toc312166144)

[真实案例： 2](#_Toc312166145)

[使用mysql 伪‘loose index scan’优化max/min/count（distinct）： 5](#_Toc312166146)

[原理介绍： 5](#_Toc312166147)

[真实案例： 6](#_Toc312166148)

[order by desc/asc limit的优化： 9](#_Toc312166149)

[优化原理： 9](#_Toc312166150)

[真实优化案例： 11](#_Toc312166151)

[延迟查找在分页和排序中的优化： 15](#_Toc312166152)

[优化原理： 15](#_Toc312166153)

[真实案例： 15](#_Toc312166154)

[Mysql中分析函数的实现： 17](#_Toc312166155)

[真实案例 17](#_Toc312166156)

[Oracle的时间隐式转换： 20](#_Toc312166157)

[原理介绍： 20](#_Toc312166158)

[真实案例 21](#_Toc312166159)

[Oracle提示符—/\*+ordered use\_nl\*/ 28](#_Toc312166160)

[真实案例 28](#_Toc312166161)

[主键在myisam中的使用误区： 31](#_Toc312166162)

[真实案例： 31](#_Toc312166163)

[tc读库查询回表案例： 32](#_Toc312166164)

[真实案例： 32](#_Toc312166165)

[使用analyze分析表统计信息解决tc读库索引走错： 35](#_Toc312166166)

[真实案例 35](#_Toc312166167)

[优化oracle的虚拟函数：str2numlist BITXOR\_TB, str2varList： 38](#_Toc312166168)

[BITXOR\_TB: 38](#_Toc312166169)

[str2numList： 40](#_Toc312166170)

[str2varList 41](#_Toc312166171)

[JDBC篇： 43](#_Toc312166172)

[Mysql-jdbc批量提交的优化： 43](#_Toc312166173)

[线上项目优化篇: 46](#_Toc312166174)

[案例一：Snsbask淘分享db优化： 46](#_Toc312166175)

[案例二：结合业务巧妙优化count(\*); 49](#_Toc312166176)

[工具使用： 50](#_Toc312166177)

[mk-query-digest：日志分析工具利器 50](#_Toc312166178)

[diffsqltables.pl 日常环境与性能环境、沙箱环境同步脚本 53](#_Toc312166179)

[主备一致校验工具：mk-table-checksum 54](#_Toc312166180)

# SQL篇：

## mysql子查询：

### 原理介绍：

mysql优化器在处理子查询的时候，会将将子查询改写。通常情况下，我们希望由内到外，

先完成子查询的结果，然后在用子查询来驱动外查询的表，完成查询例如：

Select \* from test where test\_id in(select fk\_test\_id from sub\_test where group\_id=10)

通常我们会想到该sql的执行顺序为：



sub\_test表中根据group\_id取得fk\_test\_id(2,3,4,5,6),然后在到test中，带入test\_id=2,3,4,5,6取得查询数据，但是实际mysql的处理方式为：

Select \* from test where exists (

select \* from sub\_test where group\_id=10 and sub\_test.test\_id=test.id)

mysql将会扫描test中的所有数据，每条数据将会传到子查询中与sub\_test关联，子查询不能首先被执行，如果外部表越大，那么与内查询关联的次数将会越多，因此性能会出现瓶颈，所以这时候需要改写sql，将子查询改写为关连查询：

select t1.\* from test t1,sub\_test t2

where t2.group\_id=10 and t1.test\_id=t2.id;

### 真实案例：

定投轮播系统在国庆期间由于子查询sql，导致应用hang主：

select item\_id, sum(sold) as sold from statistic\_item\_detail where item\_id in (select item\_id from statistic\_item\_detail where Gmt\_create >= '2011-10-07 12:09:00') group by item\_id;

该sql用于统计轮播商品的销量：比如用户选定了10-07，10-07之后有轮播数据的商品共有100个，查这100个商品所有时间里的轮播数据；由于历史原因，这条sql遗留到了现在，在国庆期间执行量增加

表的索引结构为：



原始sql:

root@provide 11:04:27>explain select item\_id, sum(sold) as sold

-> from statistic\_item\_detail

-> where item\_id in (select item\_id

-> from statistic\_item\_detail

-> where Gmt\_create >= '2011-10-07 12:09:00')

-> group by item\_id\G;

执行计划：

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

id: 1

select\_type: PRIMARY

table: statistic\_item\_detail

type: index

possible\_keys: NULL

key: ind\_statistic\_item\_detail\_item

key\_len: 11

ref: NULL

rows: 856474

Extra: Using where

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

id: 2

select\_type: DEPENDENT SUBQUERY

table: statistic\_item\_detail

type: index\_subquery

possible\_keys: ind\_statistic\_item\_detail\_item,ind\_statistic\_item\_detail\_gmt

key: ind\_statistic\_item\_detail\_item

key\_len: 8

ref: func

rows: 24

Extra: Using where

改写后：

SELECT t2.item\_id, SUM(t2.sold) AS sold

FROM (SELECT distinct item\_id

FROM statistic\_item\_detail

WHERE gmt\_create >= '2011-10-05 16:15:00'

and item\_id in (22432677, 22432677)) t1,

statistic\_item\_detail t2

WHERE t1.item\_id = t2.item\_id

GROUP BY t2.item\_id\G;

执行计划：

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

id: 1

select\_type: PRIMARY

table: <derived2>

type: system

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 1

Extra:

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

id: 1

select\_type: PRIMARY

table: t2

type: ref

possible\_keys: ind\_statistic\_item\_detail\_item

key: ind\_statistic\_item\_detail\_item

key\_len: 8

ref: const

**rows: 280**

Extra:

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

id: 2

select\_type: DERIVED

table: statistic\_item\_detail

type: range

possible\_keys: ind\_statistic\_item\_detail\_item,ind\_statistic\_item\_detail\_gmt

key: ind\_statistic\_item\_detail\_item

key\_len: 8

ref: NULL

**rows: 280**

Extra: Using where

## 使用mysql 伪‘loose index scan’优化max/min/count（distinct）：

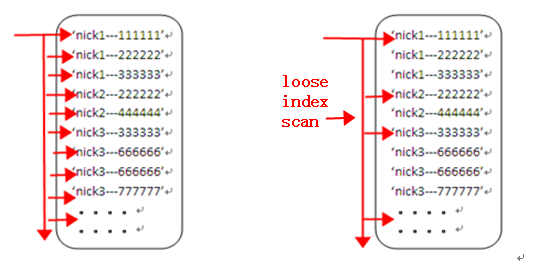
### 原理介绍：

我们知道索引的值是按照索引字段升序的，比如我们对（nick，appkey）两个字段做了索引，那么在索引中的则是按照nick，appkey的升序排列；

如果我们现在的一条sql：

select count(distinct nick) from top\_xdc\_nickapp\_period\_09\_29;

用于查询统计某天日志表中的UV，优化器则是直接从nick1开始一条条扫描下来，直到扫描到最后一个nick\_n,那么中间过程会扫描很多重复的nick，如果我们能够跳过中间重复的nick，则性能会优化非常多（在oracle中，这种扫描技术为loose index scan，但是在mysql中还不能直接支持这种优化技术）：



所以需要通过改写sql来达到伪loose index scan：

root@topxray2 09:41:30>

select count(\*) from ( select distinct(nick) from top\_xdc\_nickapp\_period\_09\_29)t ;

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

| count(\*) |

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

|   806934 |

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

Sql中先选出不同的nick，最后在外面套一层，就可以得到nick的distinct值总和；

最重要的是在子查询中：select distinct(nick) 实现了上图中的伪loose index scan，优化器在这个时候的执行计划为Using index for group-by ，

需要注意的是mysql把distinct优化为group by，它首先利用索引来分组，然后扫描索引，对需要的nick只扫描一条记录；

### 真实案例：

#### <1>优化distinct

在开放平台Xray-dc数据监控中心中，top\_xdc\_nickapp\_period\_xx\_xx这张报表的每天入库的数据在15G左右，而我们的内存只有11G，在计算每隔时段的uv sql中：

select count(distinct nick) from top\_xdc\_nickapp\_period\_09\_29;

即使在top\_xdc\_nickapp\_period\_xx\_xx上加上nick的索引，通过查看执行计划，也为全索引扫描，sql在执行的时候，会对整个服务器带来抖动；

root@topxray2 09:00:12>select count(distinct nick) from top\_xdc\_nickapp\_period\_09\_29;

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

| count(distinct nick) |

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

|               806934 |

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

1 row in set (52.78 sec)

执行一次sql需要花费52.78s，已经非常的慢了，那么按照上面的优化思路对改UV统计sql进行改写：

root@topxray2 09:41:30>

select count(\*) from ( select distinct(nick) from top\_xdc\_nickapp\_period\_09\_29)t ;

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

| count(\*) |

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

|   806934 |

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

1 row in set (5.81 sec)

由52.78秒降至5.81秒，速度提升了差不多10倍；

查看sql的执行计划分别为：

优化写法：

**root@topxray2 09:41:10>explain select distinct(nick) from top\_xdc\_nickapp\_period\_09\_29**

    -> ;

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

| id | select\_type | table                        | type  | possible\_keys | key                             | key\_len | ref  | rows    | Extra                    |

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

|  1 | SIMPLE      | top\_xdc\_nickapp\_period\_09\_29 | range | NULL          | ind\_top\_xdc\_nickapp\_period\_nick | 67      | NULL | **2124695** | **Using index for group-by** |

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

原始写法：

**root@topxray2 09:42:55>**

**explain select count(distinct nick) from top\_xdc\_nickapp\_period\_09\_29;**

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

| id | select\_type | table                        | type  | possible\_keys | key                        | key\_len | ref  | rows     | Extra       |

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

|  1 | SIMPLE      | top\_xdc\_nickapp\_period\_09\_29 | index | NULL          | ind\_top\_xdc\_nickapp\_period | 177     | NULL | **19546123 | Using index** |

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

#### <2>优化max/min

有时候我们会遇到以下的应用场景：

SELECT MAX(log\_time)

FROM log\_table

WHERE log\_machine IN ($machines)

CREATE TABLE log\_table (

id INT NOT NULL PRIMARY KEY,

log\_machine VARCHAR(20) NOT NULL,

log\_time DATETIME NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE INDEX ix\_log\_machine\_time ON log\_table (log\_machine, log\_time);

我们建立的索引为：(log\_machine,log\_time),当我们传入单个machine的时候，速度很快，但是当我们传入多个machines的时候，查询速度会一下子就降下来；

首先我们看看下面的查询：

root@test 10:21:15>explain select max(log\_time) from log\_table;

+----+-------------+-------+------+---------------+------+---------+------+----| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | Select tables optimized away |

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

当然我们的索引时建立在log\_time上的单列索引，这个时候优化器发现不用扫描所有的叶子节点，而直接到最右叶子节点的尾部就可以得到最大的log\_time；

同理当我们在in list中传入单个值的时候索引为 (log\_machine, log\_time)：

root@test 10:16:18>explain SELECT MAX(log\_time)

-> FROM log\_table

-> WHERE log\_machine IN ('Machine 1')

-> ;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | Select tables optimized away |

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

从执行计划中我们中，我们可以看到端倪，优化器和上面一样能够直接找到满足条件的最大log\_time，不过这次优化器需要首先在索引中定位到Machine 1，接着在这一系列前缀值都为machine 1的记录中，在直接在定位到最后到一条记录；

在mysql 5.5以前，mysql对[loose index scan](http://dev.mysql.com/doc/refman/5.5/en/loose-index-scan.html)并不支持，这就意味着mysql的索引扫描通常需要一个确定的起点和终点，即使查询只需要其中一些不连续的行，mysql也会扫描起点和终点范围内的所有行；那么由于不支持loose index scan，现在我们传入的是一个list，mysql不得不把所有满足在list中的machine的记录查询出来，然后在这些记录中得到最大的log\_time；

root@test 10:47:10>explain SELECT log\_machine, MAX(log\_time)

-> FROM log\_table

-> WHERE log\_machine IN ('Machine 1','Machine 2','Machine 3','Machine 4');

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | log\_table | range | ix\_log\_machine\_time | ix\_log\_machine\_time | 62 | NULL | **569160** | Using where; Using index |

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

root@test 10:47:15>SELECT log\_machine, MAX(log\_time)

-> FROM log\_table

-> WHERE log\_machine IN ('Machine 1','Machine 2','Machine 3','Machine 4');

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

| log\_machine | MAX(log\_time) |

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

| Machine 1 | 2010-05-07 23:59:13 |

1 row in set (0.65 sec)

我们看到有569160 行参加了运算；

这个时候如果我们换一种想法，把每个machine的最大log\_time计算出来，然后在计算一次所有machine的中最大的log\_time，这样不仅可以利用优化器能够直接得到每个machine的最大log\_time的优化特点，而且还可大大减少参与计算的行，这样就可以明显提升性能：

root@test 10:47:17>SELECT log\_machine, MAX(log\_time)

-> FROM log\_table

-> WHERE log\_machine IN ('Machine 1','Machine 2','Machine 3','Machine 4')

-> group by log\_machine;

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

| log\_machine | MAX(log\_time) |

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

| Machine 1 | 2010-05-07 23:59:13 |

| Machine 2 | 2010-05-07 23:58:42 |

| Machine 3 | 2010-05-07 23:58:41 |

| Machine 4 | 2010-05-07 23:56:29 |

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

root@test 10:51:44>SELECT log\_machine, MAX(log\_time) max\_log\_time

-> FROM log\_table

-> WHERE log\_machine IN ('Machine 1','Machine 2','Machine 3','Machine 4')

-> group by log\_machine

-> order by max\_log\_time desc

-> limit 1;

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

| log\_machine | max\_log\_time |

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

| Machine 1 | 2010-05-07 23:59:13 |

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

root@test 10:52:21>explain SELECT log\_machine, MAX(log\_time) max\_log\_time

-> FROM log\_table

-> WHERE log\_machine IN ('Machine 1','Machine 2','Machine 3','Machine 4')

-> group by log\_machine ;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-----------+-------+---------------------+---------------------+---------+------+------+--------| 1 | SIMPLE | log\_table | range | ix\_log\_machine\_time | ix\_log\_machine\_time | 62 | NULL | 18 | **Using where; Using index for group-by** |

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

执行时间从0.65s降到了0.00s，参与计算的row也只有18行，这个优化速度是非常明显的；

从执行计划中我们可以清楚看到优化器使用到了using index for group by，这样就可以让优化器使用“伪松散索引扫描”，最终达到优化的目的；

## order by desc/asc limit的优化：

### 优化原理：

CREATE TABLE `goods` (

`cat\_id` int(10) unsigned NOT NULL,

`seller\_id` int(10) unsigned NOT NULL,

`price` decimal(10,2) NOT NULL,

KEY `cat\_id` (`cat\_id`,`price`),

KEY `cat\_id\_2` (`cat\_id`,`seller\_id`)

)

mysql> explain select \* from goods where cat\_id=5 and seller\_id=1 order by price desc limit 10 \G

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

id: 1

select\_type: SIMPLE

table: goods

type: ref

possible\_keys: cat\_id,cat\_id\_2

key: cat\_id\_2

key\_len: 8

ref: const,const

rows: 296338

Extra: Using where; Using filesort

mysql> explain select \* from goods force index(cat\_id) where cat\_id=5 and seller\_id=1 order by price desc limit 10 \G

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

id: 1

select\_type: SIMPLE

table: goods

type: ref

possible\_keys: cat\_id

key: cat\_id

key\_len: 4

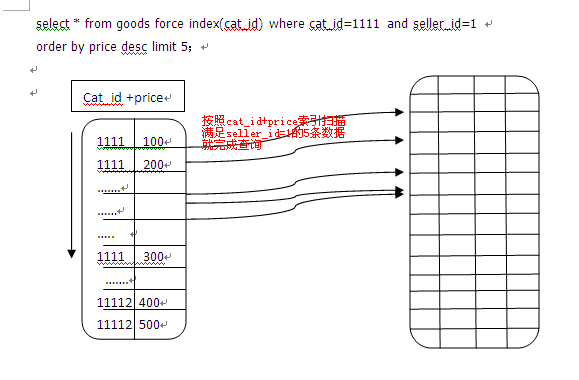
ref: const

rows: 989171

Extra: Using where

A、mysql优化器选择过滤性字段最多的索引cat\_id\_2,然后在根据price排序，这种情况对seller\_id选择性很高的时候，排序的代价是非常小的；

B、当我们强制走第二个索引的cat\_id的时候，优化器估计的rows远远比走第一个索引要多很多，但是sql中order by price能够利用索引的有序性，同时只取出10条数据，那么在执行sql的时候，只要按照索引顺序扫描，当达到满足条件的10条数据后，就退出查询；所以在seller\_id的过滤性不高的时候，这种优化技术是十分有用的



### 真实优化案例：

#### 案例一：

优化前的sql：

root@snsoption 01:53:23>

explain select POLL\_ID,USER\_ID,POLL\_COUNT,TYPE,END\_TIME,GMT\_CREATE,GMT\_MODIFIED,NUMBER\_1 from poll\_hot where END\_TIME > now() order by GMT\_CREATE desc,POLL\_COUNT desc limit 12, 12;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | poll\_hot | range | ind\_hot\_endtime | **ind\_hot\_endtime** | 9 | NULL | **113549** | Using where; Using filesort |

查看相关索引：**ind\_hot\_endtime（END\_TIME，poll\_count）**

root@snsoption 01:52:45>show index from poll\_hot;

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

| Table | Non\_unique | Key\_name | Seq\_in\_index | Column\_name | Collation | Cardinality | Sub\_part | Packed | Null | Index\_type | Comment |

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

| poll\_hot | 0 | PRIMARY | 1 | poll\_id | A | 227035 | NULL | NULL | | BTREE | |

| poll\_hot | 1 | **ind\_hot\_endtime** | **1 | end\_time**  | A | 227035 | NULL | NULL | YES | BTREE | |

| poll\_hot | 1 | **ind\_hot\_endtime** |  **2 | poll\_count**  | A | 227035 | NULL | NULL | YES | BTREE | |

| poll\_hot | 1 | idx\_ph\_num | 1 | number\_1 | A | 12 | NULL | NULL | YES | BTREE | |

| poll\_hot | 1 | idx\_ph\_num | 2 | end\_time | A | 227035 | NULL | NULL | YES | BTREE | |

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

执行时间：

root@test 02:00:44>select POLL\_ID,USER\_ID,POLL\_COUNT,TYPE,END\_TIME,GMT\_CREATE,GMT\_MODIFIED,NUMBER\_1 from poll\_hot where END\_TIME > now() order by GMT\_CREATE desc,POLL\_COUNT desc limit 12, 12;

…………………………………………………

…………………………………………………

**12 rows in set (0.49 sec)**

调整索引：

root@test 02:01:06>alter table poll\_hot add index ind\_gmt\_create(gmt\_create,POLL\_COUNT);

Query OK, 211945 rows affected (6.71 sec)

root@test 02:01:35>select POLL\_ID,USER\_ID,POLL\_COUNT,TYPE,END\_TIME,GMT\_CREATE,GMT\_MODIFIED,NUMBER\_1 from poll\_hot where END\_TIME > now() order by GMT\_CREATE desc,POLL\_COUNT desc limit 12, 12;

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

…………………………..

…………………………...

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

12 rows in set (0.00 sec)

root@test 02:01:42>explain select POLL\_ID,USER\_ID,POLL\_COUNT,TYPE,END\_TIME,GMT\_CREATE,GMT\_MODIFIED,NUMBER\_1 from poll\_hot where END\_TIME > now() order by GMT\_CREATE desc,POLL\_COUNT desc limit 12, 12;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | poll\_hot | index | ind\_hot\_endtime | **ind\_gmt\_create** | 14 | NULL |  **48 |** Using where |

#### 案例二：

######################################

优化前：

root@test 03:05:18>explain select \*

-> from poll\_poll

-> where ranges = 0

-> and status &4 <= 0

-> and status &1 <= 0

-> and module\_id = 2

-> and gmt\_create >= '2009-12-20 09:45:16'

-> and gmt\_create <= '2009-12-28 09:45:16'

-> order by record\_count desc limit 30;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | poll\_poll | range | ind\_poll\_gmt\_create | **ind\_poll\_gmt\_create** | 8 | NULL | **193242** | Using where; Using filesort |

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

查看索引：

**ind\_poll\_gmt\_create(gmt\_create)**

root@test 03:05:25>show index from poll\_poll;

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

| Table | Non\_unique | Key\_name | Seq\_in\_index | Column\_name | Collation | Cardinality | Sub\_part | Packed | Null | Index\_type | Comment |

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

| poll\_poll | 0 | PRIMARY | 1 | poll\_id | A | 531481 | NULL | NULL | | BTREE | |

| poll\_poll | 1 **| ind\_poll\_gmt\_create** |  **1 | gmt\_create**  | A | 531481 | NULL | NULL | | BTREE | |

| poll\_poll | 1 | ind\_poll\_endtime | 1 | end\_time | A | 33217 | NULL | NULL | | BTREE | |

| poll\_poll | 1 | ind\_poll\_userid | 1 | user\_id | A | 265740 | NULL | NULL | | BTREE | |

| poll\_poll | 1 | ind\_poll\_userid | 2 | gmt\_create | A | 531481 | NULL | NULL | | BTREE | |

| poll\_poll | 1 | idx\_pp\_num | 1 | number\_1 | A | 17 | NULL | NULL | YES | BTREE | |

| poll\_poll | 1 | idx\_pp\_num | 2 | gmt\_create | A | 531481 | NULL | NULL | | BTREE | |

| poll\_poll | 1 | idx\_pp\_num | 3 | end\_time | A | 531481 | NULL | NULL | | BTREE | |

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

查询满足条件的数据有多少：

root@test 03:38:36> select count(\*)

-> from poll\_poll

-> where ranges = 0

-> and status &4 <= 0

-> and status &1 <= 0

-> and module\_id = 2

-> and gmt\_create >= '2011-12-05 00:00:00'

-> and gmt\_create <= '2011-12-09 23:59:59';

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

| count(\*) |

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

**| 598523 |**

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

调整索引：

root@test 03:33:38>alter table poll\_poll add index ind\_poll\_poll\_module(module\_id,record\_count);

查看执行计划：

root@test 03:37:53>explain select \*

-> from poll\_poll

-> where ranges = 0

-> and status &4 <= 0

-> and status &1 <= 0

-> and module\_id = 2

-> and gmt\_create >= '2011-12-05 00:00:00'

-> and gmt\_create <= '2011-12-09 23:59:59'

-> order by record\_count desc limit 1;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

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

| 1 | SIMPLE | poll\_poll | ref | ind\_poll\_gmt\_create,ind\_poll\_poll\_module | **ind\_poll\_poll\_module** | 1 | const | **874200** | Using where |

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

root@test 03:37:56>select \*

-> from poll\_poll

-> where ranges = 0

-> and status &4 <= 0

-> and status &1 <= 0

-> and module\_id = 2

-> and gmt\_create >= '2011-12-05 00:00:00'

-> and gmt\_create <= '2011-12-09 23:59:59'

-> order by record\_count desc limit 1;

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

…………………………

……………………………

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

**1 row in set (0.00 sec)--🡪调优前执行时间为2s左右**

######################################

## 延迟查找在分页和排序中的优化：

### 优化原理：

所谓的延迟查找讲的就是根据innodb聚簇表的原理，在分页,排序中先取出主表的id，对id进行排序,分页，延迟取出表中的数据排序,分页；

我们线上分页写法已经广泛的使用先分页取出id，在同主表关联相关的记录，这样就可以减少一些不必要的数据取出,这也是延时查找的应用范围；

那么在排序的应用上，延迟查找有作用吗？答案是肯定的：

### 真实案例：

select t.\*, o.\*

from ali\_order o, ali\_trade t, ali\_wl\_product p

WHERE p.item\_id = o.item\_id

and o.trade\_id = t.id

and p.ware\_house\_id = 12

and t.gmt\_pay < '2011-09-07 16:00:00'

and t.Pay\_Status = 2

and o.delivery\_Status = 0

and o.refund\_status = 9

order by o.buyer\_flag desc limit 0, 20;

**Empty set (1.53 sec)**

改写后：

select t3.\*, t2.\*

from (select t.id tid, o.id oid

from ali\_order o, ali\_trade t, ali\_wl\_product p

WHERE p.item\_id = o.item\_id

and o.trade\_id = t.id

and p.ware\_house\_id = 12

and t.gmt\_pay < '2011-09-07 16:00:00'

and t.Pay\_Status = 2

and o.delivery\_Status = 0

and o.refund\_status = 9

order by o.buyer\_flag desc limit 0, 20) t1,

ali\_order t2,

ali\_trade t3

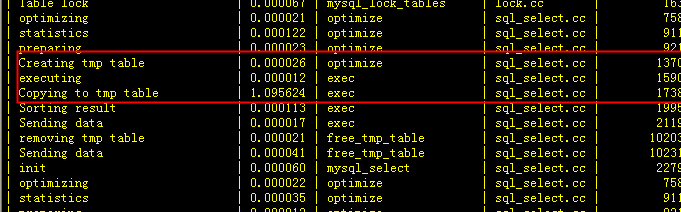
where t3.id = tid

and t2.id = oid;

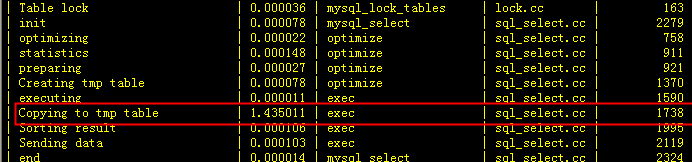
**Empty set (1.11 sec)**

在没有调整索引的情况下，执行时间降低差不多1/3，优化效果还是挺明显的，查看相关的profiles：

优化写法：



原始写法：



Copying to tmp table降低了许多，性能提升的关键原因在于我们排序的字段为id，而不是查询中的所有字段，这样就减小在排序缓存中的排序数据

## Mysql中分析函数的实现：

### 真实案例

今天收到同事的一需求，要求实现以下功能:

drop table test;

create table test(name varchar(10),name1 varchar(10),count bigint);

delete from test;

insert into test values(’1′,’a',2);

insert into test values(’1′,’b',1);;

insert into test values(’1′,’c',4);

insert into test values(’1′,’d',5);

insert into test values(’1′,’e',7);

insert into test values(’1′,’f',8);

insert into test values(’2′,’g',9);

insert into test values(’2′,’h',0);

insert into test values(’2′,’i',21);

insert into test values(’2′,’j',3);

insert into test values(’2′,’k',4);

insert into test values(’2′,’l',56);

insert into test values(’3′,’m',67);

insert into test values(’3′,’n',89);

insert into test values(’3′,’o',12);

insert into test values(’3′,’p',22);

insert into test values(’3′,’q',23);

insert into test values(’3′,’r',42);

insert into test values(’3′,’s',26);

根据name字段分组，取出改组内的前4项，并且按照count字段进行降序排序，由于mysql没有oracle中的分析函数，看上去很简单的需求，但是折腾了许久，还是没有实现，于是乎在网上收罗了一下mysql分析函数是怎么[实现的](http://tech.it168.com/a2009/0512/275/000000275999.shtml) ，找到了mysql分析函数的解决办法，学习了一下，于是乎把同事的功能实现了；

select name, name1, count

from (select b.name,

b.name1,

b.count,

if(@name = b.name, @rank := @rank + 1, @rank := 1) as rank, @name := b.name

from (select name, name1, count

from test

order by name asc, count desc) b,

(select @rownum := 0, @name := null, @rank := 0) a) result

where rank < 5;

| name | name1 | count |

*+——+——-+——-+*

*| 1    | f     |     8 |*

*| 1    | e     |     7 |*

*| 1    | d     |     5 |*

*| 1    | c     |     4 |*

*| 2    | l     |    56 |*

*| 2    | i     |    21 |*

*| 2    | g     |     9 |*

*| 2    | k     |     4 |*

*| 3    | n     |    89 |*

*| 3    | m     |    67 |*

*| 3    | r     |    42 |*

*| 3    | s     |    26 |*

*+——+——-+——-+*

*12 rows in set (0.02 sec)*

如果上面的sql初次看到有些让人摸不着头脑的话，你可以看看他的执行计划,然后从执行计划得到一些执行流程，该sql中最核心的技术点为使用自定义变量来保存sql执行过程中的值：

if(@name = b.name, @rank := @rank + 1, @rank := 1) as rank,@name:=b.name

这条判断语句对下面结果进行处理，并生成rank，由于下面查询的结果中对name做了排序，所以@name:=b.name使的相同name值的rank能够递增；

*+—-+————-+————+——–+—————+——+———+——+——+—————-+*

*| id | select\_type | table      | type   | possible\_keys | key  | key\_len | ref  | rows | Extra          |*

*+—-+————-+————+——–+—————+——+———+——+——+—————-+*

*|  1 | PRIMARY     | <derived2> | ALL    | NULL          | NULL | NULL    | NULL |   19 | Using where    |*

*|  2 | DERIVED     | <derived4> | system | NULL          | NULL | NULL    | NULL |    1 |                |*

*|  2 | DERIVED     | <derived3> | ALL    | NULL          | NULL | NULL    | NULL |   19 |                |*

*|  4 | DERIVED     | NULL       | NULL   | NULL          | NULL | NULL    | NULL | NULL | No tables used |*

*|  3 | DERIVED     | test       | ALL    | NULL          | NULL | NULL    | NULL |   19 | Using filesort |*

如果你对下面的select @rownum := 0, @name := null, @rank := 0看不太明白，可以改写一下sql：

select name, name1, count  from (select b.name, b.name1,b.count,

if(@name = b.name, @rank := @rank + 1, @rank := 1) as rank,@name:=b.name

from (select name, name1, count,@rownum := 0, @name := null, @rank := 0

from test order by name asc, count desc) b) result where rank<4;

*+—-+————-+————+——+—————+——+———+——+——+—————-+*

*| id | select\_type | table      | type | possible\_keys | key  | key\_len | ref  | rows | Extra          |*

*+—-+————-+————+——+—————+——+———+——+——+—————-+*

*|  1 | PRIMARY     | <derived2> | ALL  | NULL          | NULL | NULL    | NULL |   19 | Using where    |*

*|  2 | DERIVED     | <derived3> | ALL  | NULL          | NULL | NULL    | NULL |   19 |                |*

*|  3 | DERIVED     | test       | ALL  | NULL          | NULL | NULL    | NULL |   19 | Using filesort |*

*+—-+————-+————+——+—————+——+———+——+——+—————-+*

## Oracle的时间隐式转换：

### 原理介绍：

java.sql定义时间类型包括三个类，date, time, 和 timestamp，分别用来表示日期（无时间信息，eg: yyyy-mm-dd），时间（只处理时间，无日期部分, eg: hh:mm:ss）和时间戳（精确到纳秒级别）。在它们都继承自java.util.date。而oracle与jdbc之间的类型映射：

date  java.sql.date

date  java.sql.time

timestamp    java.sql.timestamp

oracle数据库字段类型主要有date、timestamp。

在9i以后、11g以前的oracle jdbc驱动中存在一个会丢失date类型字段的时间信息的bug，原因是其jdbc驱动将oracle的date类型处理为java.sql.date 类型，这就丢失了时间部分；事实上，如果是使用ibatis，pojo属性的类型设置为java.util.date，确保 jdbctype不为 date或者time，则避免了这个bug。因为此时ibatis会以java.sql.timestamp来处理该字段；

简而言之，oracle在处理ibatis传入到数据库的变量时候，解析为timestamp类型，如果数据库中时间字段定义为date类型，由于timestamp类型的精度比date类型的精度高，所以oracle会做出隐身转换，将date类型转换为timestamp类型，那么创建在date类型上的索引将不会使用到；

### 真实案例

select count(\*)  from order t  where (o\_type = 2 or o\_type = 3)   and p\_type = 1   and t.o\_status = :1   and t.gmt\_create >= :2   and t.gmt\_create <= :3    and t.m\_type = :4   and t.a\_id = :5   and t.a\_item\_id = :6

——————————————————————————————

| operation                      | phv/object name               |  rows | bytes|   cost |

——————————————————————————————

|select statement                |———- 1348628516 ———|       |      |      3 |

|sort aggregate                  |                               |     1 |   74 |        |

**| filter** |                               |       |      |        |

|  index range scan              |ind\_o\_artid            |     1 |   74 |      3 |

——————————————————————————————

索引idx\_artid(a\_id, p\_type, gmt\_create, o\_type, m\_type, a\_item\_id, o\_status)

跟踪一下绑定变量的值：

child\_number   position name  datatype\_string   max\_length   last\_captured   value\_string

5                             1           :1         number                    22         2011-04-25 16:30:07       3

**5                            2            :2    timestamp 11         2011-04-25 16:30:07**

**5                           3             :3    timestamp 11        2011-04-25 16:30:07**

5                          4              :4    number                       22      2011-04-25 16:30:07          6

5                          5              :5    number                        22      2011-04-25 16:30:07         940

5                          6               :6    number                       22       2011-04-25 16:30:07      1791

可以看到oracle将gmt\_create绑定为timestamp， 奇怪的是我们没有在value\_string中看到有gmt\_create的值；从执行计划上我们看到filter，原表中gmt\_create为date数据类型，这里可以断定是由于隐式转换导致了时间字段在索引中排序没有作用了，所以还要过滤filter，在索引中的gmt\_create字段没有起到过滤的作用，从而导致了大量的逻辑读；

验证如下：

11:24:02 bss2@ crm>select /\*+index(t idx\_artid)\*/ count(\*)

11:24:25   2    from order t

11:24:25   3   where (o\_type = 2 or o\_type = 3)

11:24:25   4     and p\_type = 1

11:24:25   5     and t.o\_status = 3

11:24:25   6     and t.gmt\_create >= ’2011-04-10 11:20:22′

11:24:25   7    and t.gmt\_create <= ’2011-04-13 11:20:22′—变量值传入时间的字符串

11:24:25   8     and t.m\_type = 6

11:24:25   9     and t.a\_id = 965

11:24:25  10     and t.a\_item\_id =1863;

1 row selected.

elapsed: 00:00:00.01

execution plan

———————————————————

plan hash value: 3276776702

—————————————————————————————–

| id  | operation         | name                | rows  | bytes | cost (%cpu)| time     |

—————————————————————————————–

|   0 | select statement  |                     |     1 |    74 |     2   (0)| 00:00:01 |

|   1 |  sort aggregate   |                     |     1 |    74 |            |          |

|\*  2 |   index range scan| idx\_artid |     1 |    74 |     2   (0)| 00:00:01 |

—————————————————————————————–

predicate information (identified by operation id):

—————————————————

2 – access(“t”.”a\_id”=965 and “p\_type”=1 and

**“t”.”gmt\_create”>=to\_date(‘ 2011-04-10 11:20:22′, ‘yyyy-mm-dd hh24:mi:ss’) and**

“t”.”m\_type”=6 and “t”.”a\_item\_id”=1863 and “t”.”o\_status”=3 and

“t”.”gmt\_create”<=to\_date(‘ 2011-04-13 11:20:22′, ‘yyyy-mm-dd hh24:mi:ss’))

filter((“o\_type”=2 or “o\_type”=3) and “t”.”m\_type”=6 and

“t”.”a\_item\_id”=1863 and “t”.”o\_status”=3)

statistics

———————————————————-

1  recursive calls

0  db block gets

**6  consistent gets**

0  physical reads

0  redo size

11:28:11 bss2@ crm>select /\*+index(t idx\_artid)\*/ count(\*)

11:28:12   2    from order t

11:28:12   3   where (o\_type = 2 or o\_type = 3)

11:28:12   4     and p\_type = 1

11:28:12   5     and t.o\_status = 3

11:28:12   6     and t.gmt\_create >= to\_date(’2011-04-10 11:20:22′,’yyyy-mm-dd hh24:mi:ss’)

**11:28:12   7     and t.gmt\_create <= to\_date(’2011-04-13 11:20:22′,’yyyy-mm-dd hh24:mi:ss’)–用to\_date转换一下**

11:28:12   8     and t.m\_type = 6

11:28:12   9     and t.a\_id = 965

11:28:12  10     and t.a\_item\_id =1863;

1 row selected.

elapsed: 00:00:00.01

execution plan

———————————————————-

plan hash value: 3276776702

—————————————————————————————–

| id  | operation         | name                | rows  | bytes | cost (%cpu)| time     |

—————————————————————————————–

|   0 | select statement  |                     |     1 |    74 |     2   (0)| 00:00:01 |

|   1 |  sort aggregate   |                     |     1 |    74 |            |          |

|\*  2 |   index range scan| idx\_artid |     1 |    74 |     2   (0)| 00:00:01 |

—————————————————————————————–

predicate information (identified by operation id):

—————————————————

2 – access(“t”.”a\_id”=965 and “p\_type”=1 and

“t”.”gmt\_create”>=to\_date(‘ 2011-04-10 11:20:22′, ‘syyyy-mm-dd hh24:mi:ss’) and

“t”.”m\_type”=6 and “t”.”a\_item\_id”=1863 and “t”.”o\_status”=3 and

“t”.”gmt\_create”<=to\_date(‘ 2011-04-13 11:20:22′, ‘syyyy-mm-dd hh24:mi:ss’))

filter((“o\_type”=2 or “o\_type”=3) and “t”.”m\_type”=6 and

“t”.”a\_item\_id”=1863 and “t”.”o\_status”=3)

statistics

———————————————————-

0  recursive calls

0  db block gets

**6  consistent gets**

0  physical reads

0  redo size

11:24:26 bss2@ crm>select /\*+index(t idx\_artid)\*/ count(\*)

11:26:31   2    from order t

11:26:31   3   where (o\_type = 2 or o\_type = 3)

11:26:31   4     and p\_type = 1

11:26:31   5     and t.o\_status = 3

**11:26:31   6     and t.gmt\_create >= to\_timestamp(’2011-04-10 11:20:22′,’yyyy-mm-dd hh24:mi:ss’)**

**11:26:31   7     and t.gmt\_create <= to\_timestamp(’2011-04-13 11:20:22′,’yyyy-mm-dd hh24:mi:ss’)–模仿没有经过处理的变量值，ibatis转换为timestamp**

11:26:31   8     and t.m\_type = 6

11:26:31   9     and t.a\_id = 965

11:26:31  10     and t.a\_item\_id =1863;

1 row selected.

elapsed: 00:00:00.20

execution plan

———————————————————-

plan hash value: 2438882768

——————————————————————————————

| id  | operation          | name                | rows  | bytes | cost (%cpu)| time     |

——————————————————————————————

|   0 | select statement   |                     |     1 |    74 |     3   (0)| 00:00:01 |

|   1 |  sort aggregate    |                     |     1 |    74 |            |          |

**|\*  2 |   filter           |                     |       |       |            |          |**

|\*  3 |    index range scan| idx\_artid |     1 |    74 |     3   (0)| 00:00:01 |

——————————————————————————————

predicate information (identified by operation id):

—————————————————

2 – filter(to\_timestamp(’2011-04-10 11:20:22′,’yyyy-mm-dd

hh24:mi:ss’)<=to\_timestamp(’2011-04-13 11:20:22′,’yyyy-mm-dd hh24:mi:ss’))

3 – access(“t”.”a\_id”=965 and “p\_type”=1 and “t”.”m\_type”=6 and

“t”.”a\_item\_id”=1863 and “t”.”o\_status”=3)

filter((“o\_type”=2 or “o\_type”=3) and “t”.”m\_type”=6 and

“t”.”a\_item\_id”=1863 and internal\_function(“t”.”gmt\_create”)>=to\_timestamp(‘

2011-04-10 11:20:22′,’yyyy-mm-dd hh24:mi:ss’) and

internal\_function(“t”.”gmt\_create”)<=to\_timestamp(’2011-04-13

11:20:22′,’yyyy-mm-dd hh24:mi:ss’) and “t”.”o\_status”=3)

statistics

———————————————————-

1  recursive calls

0  db block gets

**698  consistent gets**

0  physical reads

0  redo size

可以看到当我们采用to\_timestatmp转换gmt\_create传入的值后，其执行计划是和没有做任何处理查询变量的是一样的，

所以接下来的两种方案可以为：

使用cast(:2 as date)转换一下；

变量定义为字符串，在采用to\_date(:2,’yyyy-mm-dd hh24:mi:ss’);

程序中绑定为date：gmt\_create<#2:date#；

结论：在处理时间类型的时候，我们需要特别的谨慎，避免隐式转换的发生.

## Oracle提示符—/\*+ordered use\_nl\*/

### 真实案例

今天早上10点左右，misc负载上升，发现有大量物流的对账sql在执行，同时联系云湛对tim是否有影响，发现部分接口调用的较慢，sql为:

SELECT */\*+use\_nl(a,t)\*/*

count(ID)

  FROM table(str2numlist(:1)) a, TOP\_AUTHORIZE t

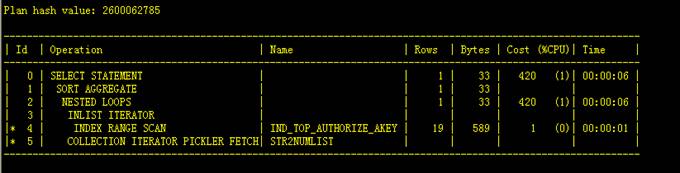
where t.APP\_KEY = :2

   and t.USER\_ID = a.column\_value

   and (t.STATUS = 1 or t.STATUS = 2)

**AND AUTHORIZE\_TYPE IN (3, 7)**

**执行计划为：**

****

走的索引为ind\_top\_authorize\_akey：app\_key，status，user\_id，gmt\_create，AUTHORIZE\_TYPE

执行计划可以简单的理解为根据传入的appkey+status+authorize\_type值走索引**ind\_top\_authorize\_akey**过滤后，在和虚拟表str2numlist做一次嵌套循环；

Authori\_type为topnotify新发布日常加上去的查询条件，之前已经在索引中冗余了该字段，避免回表：

在联系物流停掉对账的sql后，misc的负载降低了许多，但是tim总体还是异常，tim很多sql执行时间较长，由于top\_authorize表中有大量status=0的数据

是可以删除的，于是采取了数据订正，删除了该表中200w数据，删除数据后，该sql的执行时间还是较长，于是联系吴迪和墨灵回滚掉相关发布后，tim恢复正常；

由于在没有传入**AUTHORIZE\_TYPE**前的执行计划是正常的，所以查看前一天该sql的执行计划：

SELECT */\*+use\_nl(a,t)\*/*

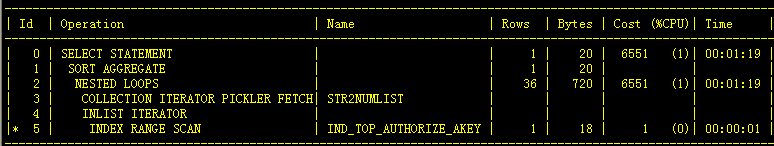
count(ID)

  FROM table(str2numlist(:1)) a, TOP\_AUTHORIZE t

where t.APP\_KEY = :2

   and t.USER\_ID = a.column\_value

   and (t.STATUS = 1 or t.STATUS = 2)

****

发现是驱动表的顺序是先扫描虚拟表中的user\_id，在根据查询索引ind\_top\_authorize\_akey得到的结果集做嵌套循环；

那为什么之前以查询ind\_top\_authorize\_akey索引得到的结果集作为驱动表，是不正确的执行计划？

在咨询了传入虚拟表的user\_id数量为最多20个user\_id，如果查询索引ind\_top\_authorize\_akey的结果集有热点appkey的时候，那么这条sql的

循环次数将会非常大；

20:29:51 TOP@ misc>select app\_key,count(\*) from top\_authorize group by app\_key having count(\*)>10000;

APP\_KEY                COUNT(\*)

--------------------          ----------

12011906                  17031

12016086                 257725

12018268                  62726

12029422                  39737

12087192                  31687

12178237                  20183

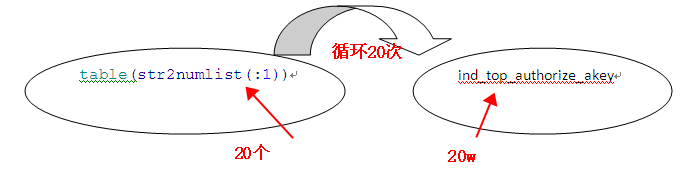
12245347                  11889

12267317                  23088

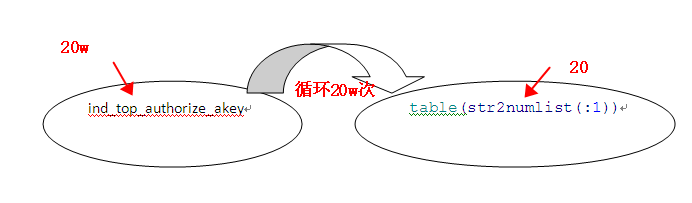
12275531                  10001

12364270                  11755

正确驱动顺序执行计划：



错误驱动顺序执行计划：



所以在提示符中加入ordered，强制sql按照虚拟表作为驱动表：

SELECT */\*+ ordered use\_nl(a,t)\*/*

count(ID)

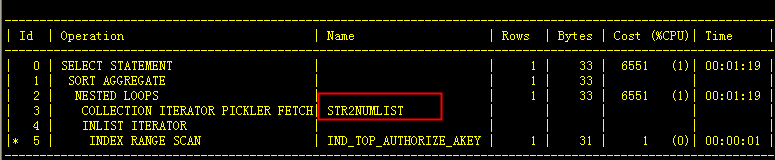
  FROM table(str2numlist(:1)) a, TOP\_AUTHORIZE t

where t.APP\_KEY = :2

   and t.USER\_ID = a.column\_value

   and (t.STATUS = 1 or t.STATUS = 2)

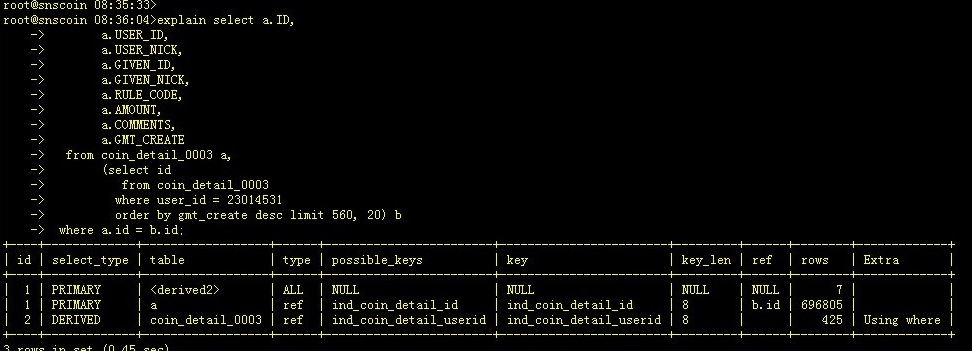
   AND AUTHORIZE\_TYPE IN (3, 7)

****

## 主键在myisam中的使用误区：

### 真实案例：

首先查看Sql执行计划（coin\_detail\_xxxx为myisam存储引擎的表）



可以发现id=2出出现了using where，表名查询回表了，再看看索引的情况：



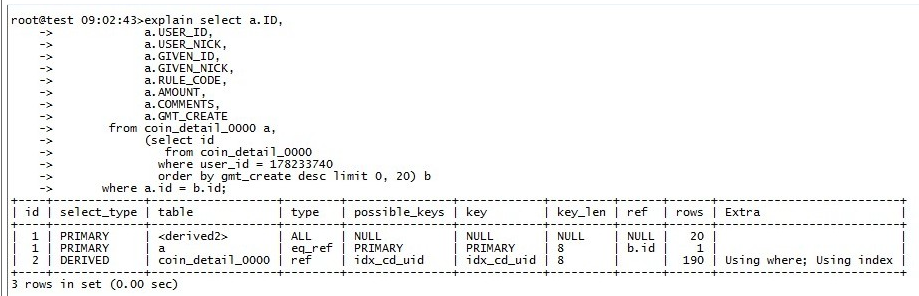
对应innodb存储引擎的来说，由于id是存储到索引的最后面，子查询完全可以通过覆盖索引来完成查询，不用再回表：

ind\_coin\_detail\_userid:user\_id+gmt\_create+amount+id

而对于myisam存储引擎来说，id也是一个普通的二级索引，其索引后面加入的不是主键id，而是行的行号；

ind\_coin\_detail\_userid:user\_id+gmt\_create+amount+rownum

所以这里就解释了using where出现的原因；所以调整索引：ind\_coin\_detail\_userid:user\_id+gmt\_create+amount+id



显示的将id加入到索引的创建中；

## tc读库查询回表案例：

### 真实案例：

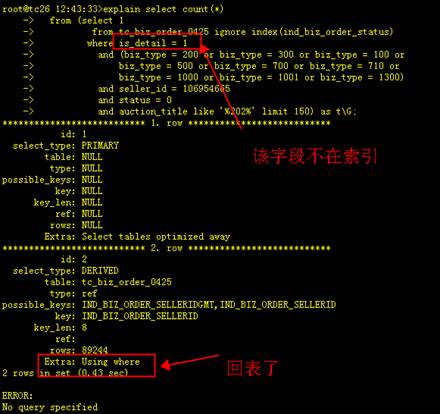
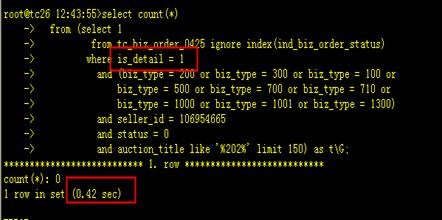
tc读库load较高，通过查看慢日志sql的执行计划，发现tc

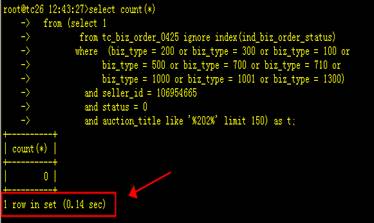
的索引IND\_BIZ\_ORDER\_PARENTID，没有is\_detail字段，导致回表造成大量的随机io：

索引字段为：

seller\_id, gmt\_create, is\_main, pay\_status, biz\_type, status, logistics\_status, sub\_biz\_type, refund\_status, seller\_rate\_status, buyer\_rate\_status, end\_time, trade\_tag, shop\_id, auction\_title

而查询中是有is\_detail的查询：



上面是执行多次后的执行时间，可以看到从执行时间上看减小3倍，

当第一次执行的时候，由于内存中没有缓存数据，还会更慢：



所以最终原因是索引中没有冗余is\_detail字段导致回表导致问题；

## 使用analyze分析表统计信息解决tc读库索引走错：

### 真实案例

今早上登上tc的读库和主库上执行，发现该表在主库和备库的统计信息不一致：

A.表的统计信息：



-分析后的表统计信息:



采用analyze table重新分析统计信息后，执行计划正常了；

root@tc30 09:46:15>

explain select count(\*) from tc\_biz\_order\_0480  where  is\_main = 1 and (biz\_type=100 or biz\_type=200 or biz\_type=300 or biz\_type=500) and seller\_id=13048288 and status=0 and gmt\_modified>=date\_format('2011-12-05 20:11:33','%Y-%m-%d %T') and gmt\_modified<=date\_format('2011-12-05 20:19:09','%Y-%m-%d %T')\G       ;

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

  id: 1

 select\_type: SIMPLE

 table: tc\_biz\_order\_0480

 type: ref

possible\_keys:IND\_BIZ\_ORDER\_SELLERIDGMT,IND\_BIZ\_ORDER\_GMT,ind\_biz\_order\_status,IND\_BIZ\_ORDER\_SELLERID

**key: IND\_BIZ\_ORDER\_SELLERIDGMT**

key\_len: 8

ref: const

 rows: 2966

Extra: Using where

但是注意了一下show tables status ,show index也会导致innodb重新收集表和索引的统计信息

**show index from tc\_biz\_order\_0480;**

**show table  status like '%tc\_biz\_order\_0480%';**

root@tc30 09:46:29>

explain select count(\*) from tc\_biz\_order\_0480  where  is\_main = 1 and (biz\_type=100 or biz\_type=200 or biz\_type=300 or biz\_type=500) and seller\_id=13048288 and status=0 and gmt\_modified>=date\_format('2011-12-05 20:11:33','%Y-%m-%d %T') and gmt\_modified<=date\_format('2011-12-05 20:19:09','%Y-%m-%d %T')\G       ;

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

    id: 1

  select\_type: SIMPLE

    table: tc\_biz\_order\_0480

    type: range

possible\_keys:

IND\_BIZ\_ORDER\_SELLERIDGMT,IND\_BIZ\_ORDER\_GMT,ind\_biz\_order\_status,IND\_BIZ\_ORDER\_SELLERID

**key: IND\_BIZ\_ORDER\_GMT**

    key\_len: 8

      ref: NULL

      rows: 2604

      Extra: Using where

这个时候需要关闭 innodb\_stats\_on\_metadata=on可以关闭相关统计信息的收集

## 优化oracle的虚拟函数：str2numlist BITXOR\_TB, str2varList：

### BITXOR\_TB:

|  |
| --- |
| 原函数代码 |
| create or replace function BITXOR\_TB(x in number, y in number) return number is    Result number;   begin    if **BITAND**(x,y)<>0 then---多次调用bitand函数       Result:=**BITOR**(x,y) - **BITAND**(x,y);--- bitand原理优化    else        return x;    end if;    return(Result);  end BITXOR\_TB;    优化点：  用变量保存bitand计算的值，以免多次调用；  **BITXOR(x,y) = BITOR(x,y) - BITAND(x,y) = (x + y) - BITAND(x, y) \* 2;**  优化前：  escape Time :95  escape Time :94  escape Time :95  escape Time :95  cpu使用率：40%-60%波动 |

|  |
| --- |
| 优化后效果： |
| CREATE OR REPLACE FUNCTION BITXOR\_tb2(x in number, y in number) return number is     Result number;     bitandV number := BITAND(x,y);  begin     if bitandV <> 0 then  **Result := (x + y) - bitandV \* 2;**     else         return x;     end if;     return(Result);  end BITXOR\_TB2;    优化后：  escape Time :65  escape Time :65  escape Time :64  escape Time :65  cpu使用率：在30%-40%波动 |
|  |

### str2numList：

|  |
| --- |
| 原函数代码 |
| create or replace function str2numList(p\_string in varchar2 )     return numTableType  as  v\_str long default p\_string || ',';  v\_n number;  v\_data numTableType := numTableType();  begin  loop     v\_n := instr( v\_str, ',' );     exit when (nvl(v\_n,0) = 0);     v\_data.extend;     v\_data( v\_data.count ) := ltrim(rtrim(substr(v\_str,1,v\_n-1)));                                     ---ltrim和rtrim两个函数用trim函数代替       v\_str := substr( v\_str, v\_n+1 );  end loop;  return v\_data;  end;  优化点：  Ltrim和rtrim可用trim函数替换：  优化前：Elaspe Time :118  Elaspe Time :118  Elaspe Time :119  Elaspe Time :118  Elaspe Time :118 |

**优化后：**

|  |
| --- |
| 优化后代码 |
| 优化后：  create or replace function str2numlist3(p\_string varchar)  return numTableType  as     v\_str long default p\_string || ',';     v\_n number;     v\_data numTableType := numTableType();     v\_i number := 1;  begin      loop          v\_n := instr( v\_str, ',' ,v\_i);          exit when (v\_n = 0);          v\_data.extend;          v\_data( v\_data.count ) := trim(substr(v\_str, v\_i , v\_n - v\_i));          v\_i := v\_n + 1;          end loop;          return v\_data;  end;  优化后：Elaspe Time :91  Elaspe Time :92  Elaspe Time :91  Elaspe Time :92  Elaspe Time :91    **Cpu使用率：**  **优化前在45%-67%**  **优化后在35%-50%** |  |

### str2varList

|  |
| --- |
| 原函数代码 |
| create or replace function str2varList( p\_string in varchar2) return VarTableType  as  v\_str long default p\_string || ',';  v\_n varchar2(2000);----v\_n为数字，但是被定义为了字符串  v\_data VarTableType := VarTableType();  begin  loop  **v\_n :=instr( v\_str, ',' );**  exit when (nvl(v\_n,0) = 0);  v\_data.extend;  v\_data( v\_data.count ) := ltrim(rtrim(substr(v\_str,1,v\_n-1)));  v\_str := substr( v\_str, v\_n+1 );  end loop;  return v\_data;  end;  除了上面所讲的优化点外,请注意：  V\_n被定义为了varchar(2000)，改为定义number；  优化前：  Elaspe Time :173  Elaspe Time :173  Elaspe Time :173  Elaspe Time :173  Elaspe Time :173 |

优化后：

|  |
| --- |
| 优化后代码 |
| create or replace function str2varList2(p\_string varchar)  return varTableType  as     v\_str long default p\_string || ',';  **v\_n number;**     v\_data varTableType := varTableType();     v\_i number := 1;  begin          loop  **v\_n := instr( v\_str, ',' ,v\_i);**  **exit when (v\_n = 0);**              v\_data.extend;              v\_data( v\_data.count ) := trim(substr(v\_str, v\_i , v\_n - v\_i));              v\_i := v\_n + 1;          end loop;          return v\_data;  end str2varList2;  优化后：  Elaspe Time :100  Elaspe Time :100  Elaspe Time :99  Elaspe Time :100  Elaspe Time :100    **Cpu使用：优化前为：40%-67%**  **优化后为：30%-60%** |

# JDBC篇：

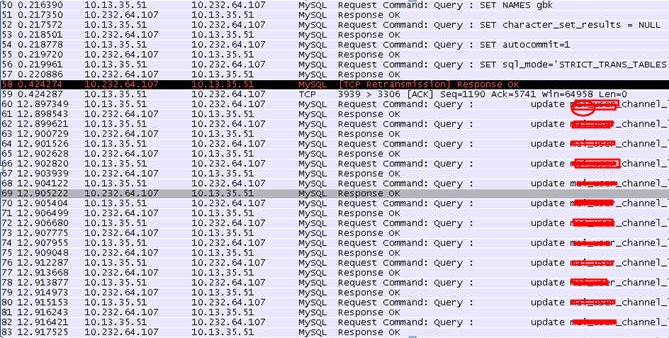
## Mysql-jdbc批量提交的优化：

.**背景**

用户修改布局时，需要批量更新mysql的xxxx\_layout\_xxxx表。批量操作的数据量是2-30条/次。批量操作是这次项目在技术上比较关键的一个点，之前批量操作做过性能上的测试，mysql端问题不大，7000+tps，Java端的效率有些差，有优化空间。

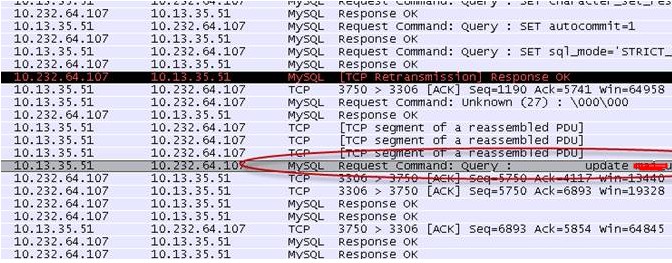
**对批量的性能进行了测试，优化。过程如下**。

经测试，批量更新30条记录的时间是35ms。由于数据在mysql服务端中会有内存缓存，批量更新30条的时间用了35ms，感觉有些长，试图找出原因。使用截包工具（这里用的ethereal），抓取mysql的数据包，下面是一次批量更新的数据包:

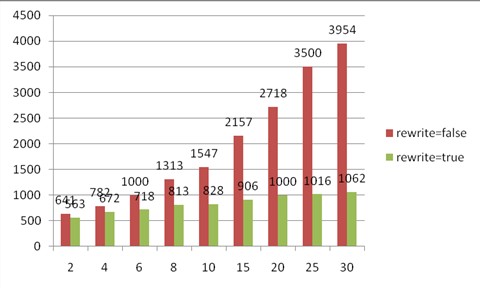
[](http://hidba.org/wp-content/uploads/2011/07/commit1.jpg)可以看出，批量更新时，每条update语句都去mysql请求了一次。并没有打包发给mysql。这种批量的效率肯定不会高。同样方法试了下oracle数据库，oracle驱动做的就很好，一次批量是打包在同一个请求中，是真正的批量提交，效率自然比mysql高。

找了些资料，发现mysql默认情况确实是不支持batch。为了解决上面的问题，需要给JDBC连接加上参数rewriteBatchedStatements=true，并且jdbc driver需要升级到5.1.8以上才支持这个参数。

增加参数rewriteBatchedStatements=true，driver版本升到5.1.17后，再次测试，批量更新30条的时间从35ms降到了11ms。截包后，可以看出底层的机制，已经变成批量提交：

[](http://hidba.org/wp-content/uploads/2011/07/commit2.jpg)

查看包的内容可以发现，这条请求里，封装了30条update语句

[](http://hidba.org/wp-content/uploads/2011/07/commit3.jpg)

 横坐标： 一次批量更新的条数。纵坐标：更新100次所用时间（ms），可见，当批量条数增加时，rewriteBatchedStatements=true的性能有很大优势。即使数量少时，也还是有一定优势。

**结论**：

使用rewriteBatchedStatements=true参数，对批量操作，性能有较大提高，从官方解释上看，对普通操作没有影响。 从网上资料和自己的测试上看，暂时没有发现rewriteBatchedStatements=true参数Driver版本5.1.17的问题。 因此，本项目中计划采取下面优化措施：

JDBC Driver版本从5.0.4升级到5.1.17。

连接属性中加入rewriteBatchedStatements=true参数

附：

测试环境：

mysql JDBC 3.0.4/3.1.17。

客户端: 普通PC机。

连接池数: 1-10。

10线程并发，批量更新30条记录（索引有效），循环更新100次。

批量更新主要代码：

mmpSqlMapClient.startTransaction(); // 使用事务

mmpSqlMapClient.startBatch(); // 批量提交

for (ChannelLayoutDO channelLayout: userChannelLayoutList) {              mmpSqlMapClient.update(“UserChannelLayoutDAO.updateSort”, channelLayout);

}

mmpSqlMapClient.executeBatch();

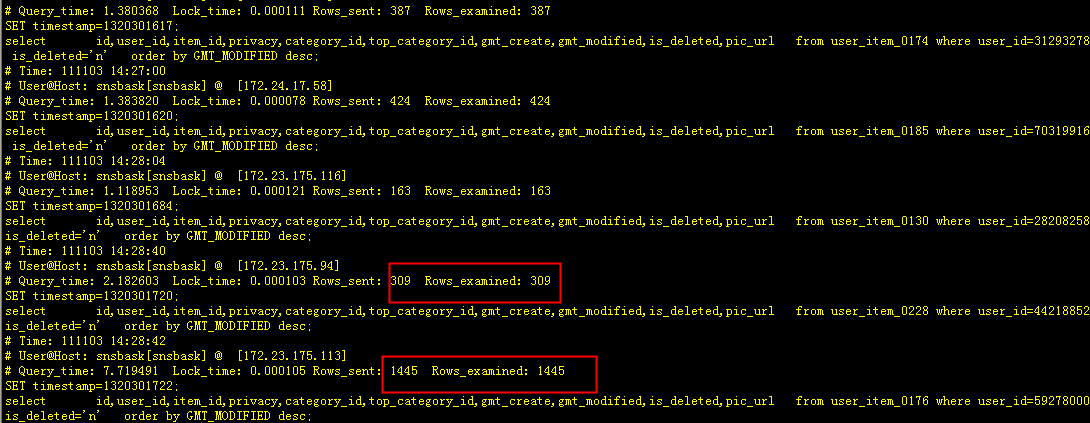
mmpSqlMapClient.commitTransaction();

# 线上项目优化篇:

## 案例一：Snsbask淘分享db优化：

snsbask优化点

A.慢日志



B．页面功能点：

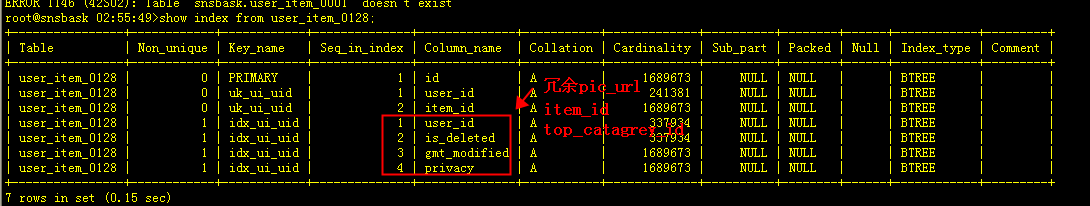


1.rows\_sent=row\_exmained，选取了过多的数据2.id,user\_id,item\_id,privacy,category\_id,top\_category\_id,gmt\_create,gmt\_modified,is\_deleted,pic\_url ---查询了许多不必要的字段

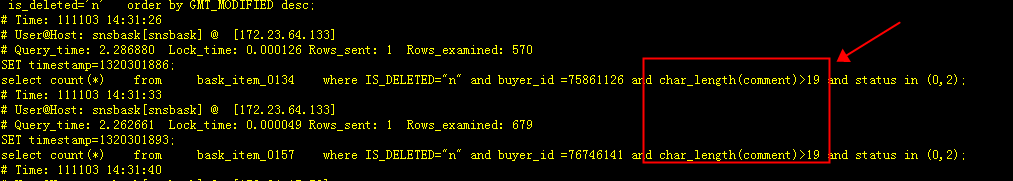
优化措施：

----改用数据库分页；

----只选出item\_id和pic\_url,调整索引，使用覆盖索引来完成查询；



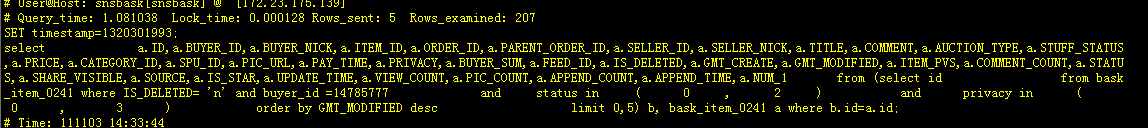
B．对大字段就行计算，导致命中率下降，无法使用覆盖索引完成查询，回表导致随机io：

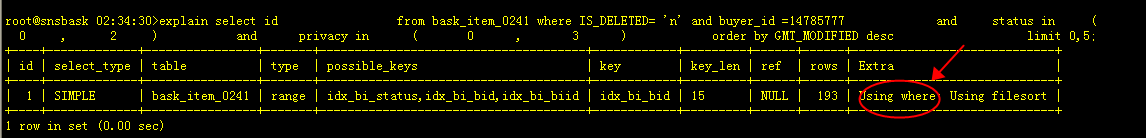


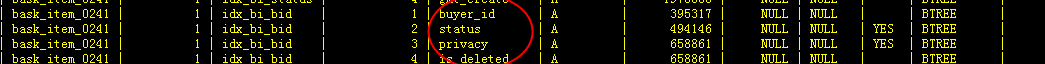
-减小comment的长度，将comment冗余到索引中？

-应用端对传入的comment字段做的计算，增加一个flag标记，来表示是否已经到达19个字符；

C.使用覆盖索引

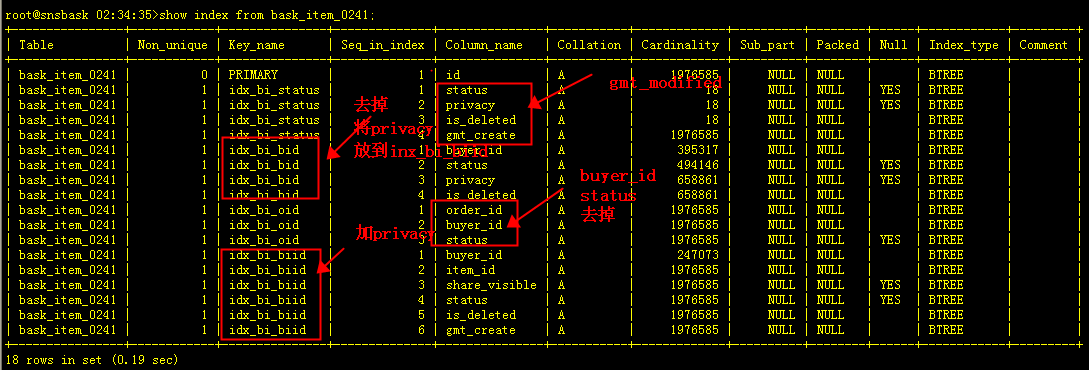






---调整索引，将gmt\_modified冗余到索引中；

D．优化表的索引：



优化点：

---第一个索引inx\_bi\_status提供dw备库增量搜索使用，剩余的status，privacy，is\_deleted没有必要加入到索引中，搜索条件中会必传gmt\_create和gmt\_modified

所以只建一个索引gmt\_modified即可（gmt\_modified会有更新）

--->第二个索引inx\_bi\_bid的字段和索引ind\_bi\_biid的字段有大部分相同，并且有相同的前导列，所以可以将该索引与最后一个索引合并，在最后一个索引中冗余privacy；

--->第三个索引ind\_bi\_oid由于order\_id基本可以定位一条记录，所以可以将buyer\_id和status去掉；

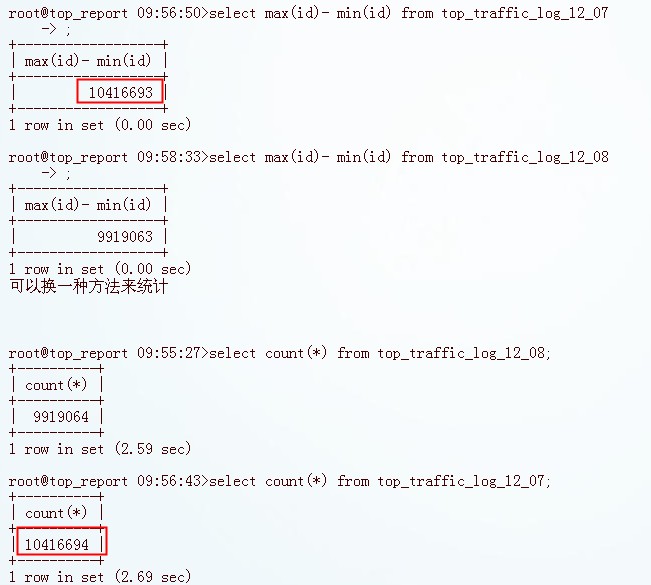
## 案例二：结合业务巧妙优化count(\*);

Topreport报表中心统计top\_traffic\_log\_xx\_xx的app流量：

需要定时执行select count(\*) from top\_traffic\_log\_xx\_xx

由于单表的数据量在千万以上，所以每次count(\*)的时候都非常的慢，急需对其进行优化；对于mysiam存储引擎的表，由于内部视图有统计表的总和数，所以count（\*）非常的快，但是如果是innodb 存储引擎，由于是聚簇表，所以count(\*)的时候需要扫描这个聚簇索引；

怎么来优化这条sql，在分析了业务后，由于该表数据为递增写入，中间没有删除，所以表中的数据主键是连续的，怎样这个表的数据可以由最大id减去最小id，加上1后就是整个表的数据之和：



这样很巧妙的优化掉了查询统计慢的sql，响应时间一下子就提升上去了；

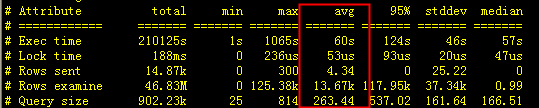
^\_^

# 工具使用：

## mk-query-digest：日志分析工具利器

(1)查找问题：

对于sql的定位可以使用mk-query-digest这个工具分析慢日志，根据这个工具所产生的profile来分析问题的瓶颈:

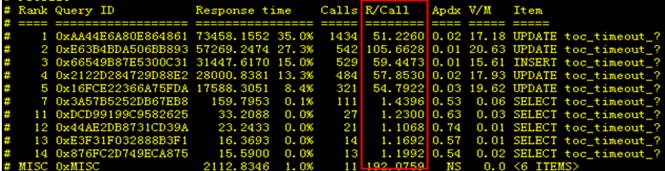


A从整体报告来看:

每条sql平均的执行时间为60S，lock time有53us，rows examie为13k，但是 rows sent只有4条，这样可以明白这个库的sql应该

为没有大量的lock 等待，而是有大量没有优化的sql导致全表扫描，或者索引建的不好，索引走错，导致大量数据的扫描（从rows sent的值很小，可以排除dump，或备份）；

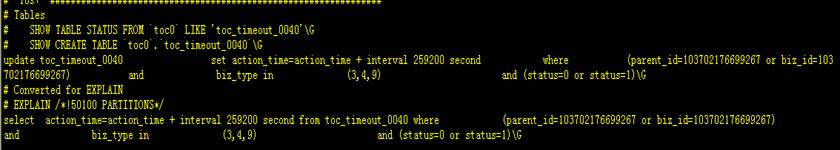
B接着看具体的详细报告：



这个报告将sql进行了归类，这里可以从response time以及每条sql的响应时间开始，看到前5条sql的执行时间都很异常，

特别是第二条sql，在进一步检查各条sql：

C.具体验证第二rank 2的sql：



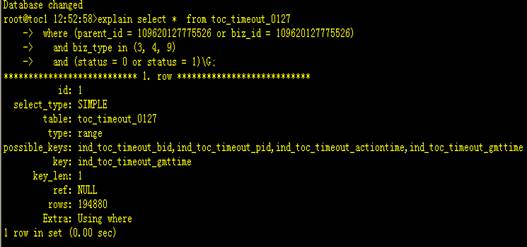
可以看到这个工具帮我们做了update到select的转换，进一步看：



可以从rows examine和row sent看出，这个库的row examin异常来源这条sql，并且从执行时间的范围分布来看，

这条sql的执行时间绝大部分一直都在10S以上，最终导致了将连接数占用完，导致应用拿不到连接。

D.查看执行计划：

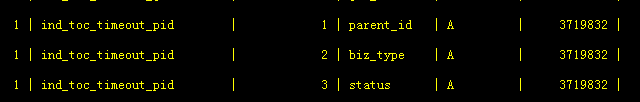


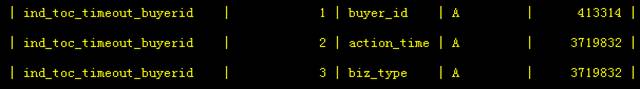
可以看到这条sql需要扫描194880行数据，使用的索引为gmttime(status,gmt\_create),从using where中可以出，从索引的status（0,1）并不会过滤

大部分的数据，然后在回表更新需要的字段，这样就会导致大量的随机读，导致buffer pool命中率下降，整个库的sql执行时间加长；

(2).完善sql：

这条sql的优化可以将两个or的条件拆开（原因为两个or条件的过滤性都很高，并且这两个条件上已经建有索引）





所以原始sql：

select \*  from toc\_timeout\_0127

 where (parent\_id = 109620127775526 or biz\_id = 109620127775526)

   and biz\_type in (3, 4, 9)

   and (status = 0 or status = 1)

改为：

select \*  from toc\_timeout\_0127

 where (parent\_id = 109620127775526 )

   and biz\_type in (3, 4, 9)

   and (status = 0 or status = 1)

   union

select \*  from toc\_timeout\_0127

 where (biz\_id = 109620127775526)

   and biz\_type in (3, 4, 9)

   and (status = 0 or status = 1)

执行时间对比：

原始sql：1 row in set (1 min 30.47 sec)

改造后sql:1 row in set (0.01 sec)

## diffsqltables.pl 日常环境与性能环境、沙箱环境同步脚本

在10.232.31.33

/home/xuancan/tools/diffsqltables/trunk/diffsqltables.pl

这个脚本可以同步两个环境的表结构，索引，数据库

需求：现在将日常环境的snsexchange数据库与性能环境的snsexchange数据库做表结构，索引同步

-dbsrc---为日常环境的DNS

-dbdst---为性能环境DNS

-user---日常环境和性能环境共同用户

-pw---为登陆日常环境和性能环境的密码

（如果没有共同用户和共同密码则：-usersrc--为日常环境用户，-userdst--为性能环境用户  -pwsrc---为登陆日常环境密码  -pwdst---为登陆性能环境密码）

(1)同步表结构，索引

A.源库和目标库用户名，密码相同：

./diffsqltables.pl [-dbsrc=snsexchange@10.232.31.14:3306](mailto:-dbsrc=snsexchange@10.232.31.14:3306) [-dbdst=snsexchange@10.232.21.195](mailto:-dbdst=snsexchange@10.232.21.195) -user=sns -pw=sns

B.源库和目标库用户名，密码不相同：  
./diffsqltables.pl [-dbsrc=snsexchange@10.232.31.14:3306](mailto:-dbsrc=snsexchange@10.232.31.14:3306) [-dbdst=snsexchange@10.232.21.195](mailto:-dbdst=snsexchange@10.232.21.195) -usersrc=sns\_src -userdst=sns\_dst -pwsrc=pw\_src -pwdst=pw\_dst

(2)同步数据：  
./diffsqltables.pl [-dbsrc=snsexchange@10.232.31.14:3306](mailto:-dbsrc=snsexchange@10.232.31.14:3306) [-dbdst=snsexchange@10.232.21.195](mailto:-dbdst=snsexchange@10.232.21.195) -user=sns -pw=sns -data -limit=0

## 主备一致校验工具：mk-table-checksum

$/u01/dbaperl/bin/perl /usr/bin/pt-table-checksum

h=172.24.65.29,u=superman,p=superman

h=172.24.65.46,u=superman,p=superman

-d snscoin\_06 -t coin\_detail\_activity\_0221

