MySQL索引原理及慢查询优化

NeverMore

MySQL凭借着出色的性能、低廉的成本、丰富的资源，已经成为绝大多数互联网公司的首选关系型数据库。虽然性能出色，但所谓“好马配好鞍”，如何能够更好的使用它，已经成为开发工程师的必修课，我们经常会从职位描述上看到诸如“精通MySQL”、“SQL语句优化”、“了解数据库原理”等要求。我们知道一般的应用系统，读写比例在10:1左右，而且插入操作和一般的更新操作很少出现性能问题，遇到最多的，也是最容易出问题的，还是一些复杂的查询操作，所以查询语句的优化显然是重中之重。
本人从13年7月份起，一直在美团核心业务系统部做慢查询的优化工作，共计十余个系统，累计解决和积累了上百个慢查询案例。随着业务的复杂性提升，遇到的问题千奇百怪，五花八门，匪夷所思。本文旨在以开发工程师的角度来解释数据库索引的原理和如何优化慢查询。

一个慢查询引发的思考

```
select
   count(*)
from
   task
where
   status=2
   and operator_id=20839
   and operate_time>1371169729
   and operate_time<1371174603
   and type=2;
```

系统使用者反应有一个功能越来越慢，于是工程师找到了上面的SQL。
并且兴致冲冲的找到了我，“这个SQL需要优化，给我把每个字段都加上索引”
我很惊讶，问道“为什么需要每个字段都加上索引？”
“把查询的字段都加上索引会更快”工程师信心满满
“这种情况完全可以建一个联合索引，因为是最左前缀匹配，所以operate_time需要放到最后，而且还需要把其他相关的查询都拿来，需要做一个综合评估。”
“联合索引？最左前缀匹配？综合评估？”工程师不禁陷入了沉思。
多数情况下，我们知道索引能够提高查询效率，但应该如何建立索引？索引的顺序如何？许多人却只知道大概。其实理解这些概念并不难，而且索引的原理远没有想象的那么复杂。

MySQL索引原理

索引目的

索引的目的在于提高查询效率，可以类比字典，如果要查“mysql”这个单词，我们肯定需要定位到m字母，然后从下往下找到y字母，再找到剩下的sql。如果没有索引，那么你可能需要把所有单词看一遍才能找到你想要的，如果我想找到m开头的单词呢？或者ze开头的单词呢？是不是觉得如果没有索引，这个事情根本无法完成？

索引原理

除了词典，生活中随处可见索引的例子，如火车站的车次表、图书的目录等。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方式来锁定数据。
数据库也是一样，但显然要复杂许多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等等。数据库应该选择怎么样的方式来应对所有的问题呢？我们回想字典的例子，能不能把数据分成段，然后分段查询呢？最简单的如果1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段.....这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。但如果是1千万的记录呢，分成几段比较好？稍有算法基础的同学会想到搜索树，其平均复杂度是lgN，具有不错的查询性能。但这里我们忽略了一个关键的问题，复杂度模型是基于每次相同的操作成本来考虑的，数据库实现比较复杂，数据保存在磁盘上，而为了提高性能，每次又可以把部分数据读入内存来计算，因为我们知道访问磁盘的成本大概是访问内存的十万倍左右，所以简单的搜索树难以满足复杂的应用场景。

磁盘IO与预读

前面提到了访问磁盘，那么这里先简单介绍一下磁盘IO和预读，磁盘读取数据靠的是机械运动，每次读取数据花费的时间可以分为寻道时间、旋转延迟、传输时间三个部分，寻道时间指的是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下；旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是1/120/2 = 4.17ms；传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。那么访问一次磁盘的时间，即一次磁盘IO的时间约等于5+4.17 = 9ms左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十万百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。下图是计算机硬件延迟的对比图，供大家参考：

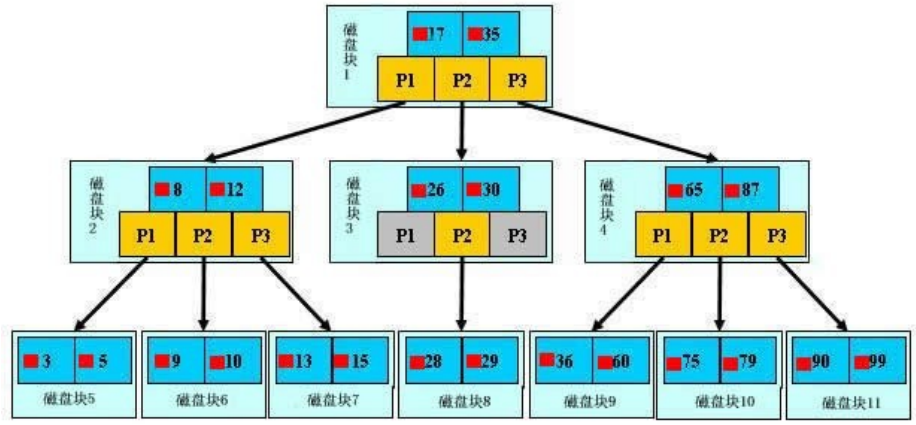
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

考虑到磁盘IO是非常高昂的操作，计算机操作系统做了一些优化，当一次IO时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候，实际上才发生了一次IO，这个理论对于索引的数据结构设计非常有帮助。

索引的数据结构

前面讲了生活中索引的例子，索引的基本原理，数据库的复杂性，又讲了操作系统的相关知识，目的就是让大家了解，任何一种数据结构都不是凭空产生的，一定会有它的背景和使用场景，我们现在总结一下，我们需要这种数据结构能够做些什么，其实很简单，那就是：每次查找数据时把磁盘IO次数控制在一个很小的数量级，最好是常数数量级。那么我们就想到如果一个高度可控的多路搜索树是否能满足需求呢？就这样，b+树应运而生。

详解b+树



如上图，是一颗b+树，关于b+树的定义可以参见B+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

b+树的查找过程

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

b+树性质

1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。
2.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

慢查询优化

关于MySQL索引原理是比较枯燥的东西，大家只需要有一个感性的认识，并不需要理解得非常透彻和深入。我们回头来看看一开始我们说的慢查询，了解完索引原理之后，大家是不是有什么想法呢？先总结一下索引的几大基本原则

建索引的几大原则

1.最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。
2.=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式
3.尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录
4.索引列不能参与计算，保持列“干净”，比如from_unixtime(create_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create_time = unix_timestamp(’2014-05-29’);
5.尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可

回到开始的慢查询

根据最左匹配原则，最开始的sql语句的索引应该是status、operator_id、type、operate_time的联合索引；其中status、operator_id、type的顺序可以颠倒，所以我才会说，把这个表的所有相关查询都找到，会综合分析；
比如还有如下查询

```
select * from task where status = 0 and type = 12 limit 10;
```

```
select count(*) from task where status = 0 ;
```

那么索引建立成(status,type,operator_id,operate_time)就是非常正确的，因为可以覆盖到所有情况。这个就是利用了索引的最左匹配的原则

查询优化神器 - explain命令

关于explain命令相信大家并不陌生，具体用法和字段含义可以参考官网explain-output，这里需要强调rows是核心指标，绝大部分rows小的语句执行一定很快（有例外，下面会讲到）。所以优化语句基本上都是在优化rows。

慢查询优化基本步骤

0.先运行看看是否真的很慢，注意设置SQL_NO_CACHE
1.where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高
2.explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）
3.order by limit 形式的sql语句让排序的表优先查
4.了解业务方使用场景
5.加索引时参照建索引的几大原则
6.观察结果，不符合预期继续从0分析

几个慢查询案例

下面几个例子详细解释了如何分析和优化慢查询

复杂语句写法

很多情况下，我们写SQL只是为了实现功能，这只是第一步，不同的语句书写方式对于效率往往有本质的差别，这要求我们对mysql的执行计划和索引原则有非常清楚的认识，请看下面的语句

```
select
   distinct cert.emp_id
from
   cm_log cl
inner join
   (
      select
         emp.id as emp_id,
         emp_cert.id as cert_id
      from
         employee emp
      left join
         emp_certificate emp_cert
            on emp.id = emp_cert.emp_id
      where
         emp.is_deleted=0
   ) cert
      on (
         cl.ref_table='Employee'
         and cl.ref_oid= cert.emp_id
      )
      or (
         cl.ref_table='EmpCertificate'
         and cl.ref_oid= cert.cert_id
      )
where
   cl.last_upd_date >='2013-11-07 15:03:00'
   and cl.last_upd_date<='2013-11-08 16:00:00';
```

0.先运行一下，53条记录 1.87秒，又没有用聚合语句，比较慢

```
53 rows in set (1.87 sec)
```

1.explain

```
+----+-------------+------------+-------+---------------------------------+-----------------------+---------+--
| id | select_type | table      | type  | possible_keys                   | key                   | key_len | r
+----+-------------+------------+-------+---------------------------------+-----------------------+---------+--
|  1 | PRIMARY     | cl         | range | cm_log_cls_id,idx_last_upd_date | idx_last_upd_date     | 8       | N
|  1 | PRIMARY     | <derived2> | ALL   | NULL                            | NULL                  | NULL    | N
|  2 | DERIVED     | emp        | ALL   | NULL                            | NULL                  | NULL    | N
|  2 | DERIVED     | emp_cert   | ref   | emp_certificate_empid           | emp_certificate_empid | 4       | m
+----+-------------+------------+-------+---------------------------------+-----------------------+---------+--
```

简述一下执行计划，首先mysql根据idx_last_upd_date索引扫描cm_log表获得379条记录；然后查表扫描了63727条记录，分为两部分，derived表示构造表，也就是不存在的表，可以简单理解成是一个语句形成的结果集，后面的数字表示语句的ID。derived2表示的是ID = 2的查询构造了虚拟表，并且返回了63727条记录。我们再来看看ID = 2的语句究竟做了写什么返回了这么大量的数据，首先全表扫描employee表13317条记录，然后根据索引emp_certificate_empid关联emp_certificate表，rows = 1表示，每个关联都只锁定了一条记录，效率比较高。获得后，再和cm_log的379条记录根据规则关联。从执行过程上可以看出返回了太多的数据，返回的数据绝大部分cm_log都用不到，因为cm_log只锁定了379条记录。
如何优化呢？可以看到我们在运行完后还是要和cm_log做join,那么我们能不能之前和cm_log做join呢？仔细分析语句不难发现，其基本思想是如果cm_log的ref_table是EmpCertificate就关联emp_certificate表，如果ref_table是Employee就关联employee表，我们完全可以拆成两部分，并用union连接起来，注意这里用union，而不用union all是因为原语句有“distinct”来得到唯一的记录，而union恰好具备了这种功能。如果原语句中没有distinct不需要去重，我们就可以直接使用union all了，因为使用union需要去重的动作，会影响SQL性能。
优化过的语句如下

```
select
   emp.id
from
   cm_log cl
inner join
   employee emp
      on cl.ref_table = 'Employee'
      and cl.ref_oid = emp.id
where
   cl.last_upd_date >='2013-11-07 15:03:00'
   and cl.last_upd_date<='2013-11-08 16:00:00'
```

```
   and emp.is_deleted = 0
union
select
   emp.id
from
   cm_log cl
inner join
   emp_certificate ec
      on cl.ref_table = 'EmpCertificate'
      and cl.ref_oid = ec.id
inner join
   employee emp
      on emp.id = ec.emp_id
where
   cl.last_upd_date >='2013-11-07 15:03:00'
   and cl.last_upd_date<='2013-11-08 16:00:00'
   and emp.is_deleted = 0
```

4.不需要了解业务场景，只需要改造的语句和改造之前的语句保持结果一致

5.现有索引可以满足，不需要建索引

6.用改造后的语句实验一下，只需要10ms 降低了近200倍！

```
+----+--------------+------------+--------+---------------------------------+-------------------+---------+----
| id | select_type  | table      | type   | possible_keys                   | key               | key_len | ref
+----+--------------+------------+--------+---------------------------------+-------------------+---------+----
|  1 | PRIMARY      | cl         | range  | cm_log_cls_id,idx_last_upd_date | idx_last_upd_date | 8       | NUL
|  1 | PRIMARY      | emp        | eq_ref | PRIMARY                         | PRIMARY           | 4       | mei
|  2 | UNION        | cl         | range  | cm_log_cls_id,idx_last_upd_date | idx_last_upd_date | 8       | NUL
|  2 | UNION        | ec         | eq_ref | PRIMARY,emp_certificate_empid   | PRIMARY           | 4       | mei
|  2 | UNION        | emp        | eq_ref | PRIMARY                         | PRIMARY           | 4       | mei
| NULL | UNION RESULT | <union1,2> | ALL    | NULL                            | NULL              | NULL    | N
+----+--------------+------------+--------+---------------------------------+-------------------+---------+----
53 rows in set (0.01 sec)
```

明确应用场景

举这个例子的目的在于颠覆我们对列的区分度的认知，一般上我们认为区分度越高的列，越容易锁定更少的记录，但在一些特殊的情况下，这种理论是有局限性的

```
select
   *
from
   stage_poi sp
where
   sp.accurate_result=1
   and (
      sp.sync_status=0
      or sp.sync_status=2
      or sp.sync_status=4
   );
```

0.先看看运行多长时间,951条数据6.22秒，真的很慢

```
951 rows in set (6.22 sec)
```

1.先explain，rows达到了361万，type = ALL表明是全表扫描

```
+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows    | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+
|  1 | SIMPLE      | sp    | ALL  | NULL          | NULL | NULL    | NULL | 3613155 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+
```

2.所有字段都应用查询返回记录数，因为是单表查询 0已经做过了951条

3.让explain的rows 尽量逼近951

看一下accurate_result = 1的记录数

```
select count(*),accurate_result from stage_poi  group by accurate_result;
+----------+-----------------+
| count(*) | accurate_result |
```

```
+----------+-----------------+
|     1023 |              -1 |
|  2114655 |               0 |
|   972815 |               1 |
+----------+-----------------+
```

我们看到accurate_result这个字段的区分度非常低，整个表只有-1,0,1三个值，加上索引也无法锁定特别少量的数据

再看一下sync_status字段的情况

```
select count(*),sync_status from stage_poi  group by sync_status;
+----------+-------------+
| count(*) | sync_status |
+----------+-------------+
|     3080 |           0 |
|  3085413 |           3 |
+----------+-------------+
```

同样的区分度也很低，根据理论，也不适合建立索引

问题分析到这，好像得出了这个表无法优化的结论，两个列的区分度都很低，即便加上索引也只能适应这种情况，很难做普遍性的优化，比如当sync_status 0、3分布的很平均，那么锁定记录也是百万级别的

4.找业务方去沟通，看看使用场景。业务方是这么来使用这个SQL语句的，每隔五分钟会扫描符合条件的数据，处理完成后把sync_status这个字段变成1,五分钟符合条件的记录数并不会太多，1000个左右。了解了业务方的使用场景后，优化这个SQL就变得简单了，因为业务方保证了数据的不平衡，如果加上索引可以过滤掉绝大部分不需要的数据

5.根据建立索引规则，使用如下语句建立索引

```
alter table stage_poi add index idx_acc_status(accurate_result,sync_status);
```

6.观察预期结果,发现只需要200ms，快了30多倍。

```
952 rows in set (0.20 sec)
```

我们再来回顾一下分析问题的过程，单表查询相对来说比较好优化，大部分时候只需要把where条件里面的字段依照规则加上索引就好，如果只是这种“无脑”优化的话，显然一些区分度非常低的列，不应该加索引的列也会被加上索引，这样会对插入、更新性能造成严重的影响，同时也有可能影响其它的查询语句。所以我们第4步调差SQL的使用场景非常关键，我们只有知道这个业务场景，才能更好地辅助我们更好的分析和优化查询语句。

无法优化的语句

```
select
   c.id,
   c.name,
   c.position,
   c.sex,
   c.phone,
   c.office_phone,
   c.feature_info,
   c.birthday,
   c.creator_id,
   c.is_keyperson,
   c.giveup_reason,
   c.status,
   c.data_source,
   from_unixtime(c.created_time) as created_time,
   from_unixtime(c.last_modified) as last_modified,
   c.last_modified_user_id
from
   contact c
inner join
   contact_branch cb
      on  c.id = cb.contact_id
inner join
   branch_user bu
      on  cb.branch_id = bu.branch_id
      and bu.status in (
         1,
      2)
   inner join
      org_emp_info oei
         on  oei.data_id = bu.user_id
         and oei.node_left >= 2875
         and oei.node_right <= 10802
```

```
          and oei.org_category = - 1
    order by
       c.created_time desc  limit 0 ,
       10;
```

还是几个步骤
0.先看语句运行多长时间，10条记录用了13秒，已经不可忍受

```
10 rows in set (13.06 sec)
```

1.explain

```
+----+-------------+-------+--------+-------------------------------------+-------------------------+---------+
| id | select_type | table | type   | possible_keys                       | key                     | key_len |
+----+-------------+-------+--------+-------------------------------------+-------------------------+---------+
|  1 | SIMPLE      | oei   | ref    | idx_category_left_right,idx_data_id | idx_category_left_right | 5       |
|  1 | SIMPLE      | bu    | ref    | PRIMARY,idx_userid_status           | idx_userid_status       | 4       |
|  1 | SIMPLE      | cb    | ref    | idx_branch_id,idx_contact_branch_id | idx_branch_id           | 4       |
|  1 | SIMPLE      | c     | eq_ref | PRIMARY                             | PRIMARY                 | 108     |
+----+-------------+-------+--------+-------------------------------------+-------------------------+---------+
```

从执行计划上看，mysql先查org_emp_info表扫描8849记录，再用索引idx_userid_status关联branch_user表，再用索引idx_branch_id关联contact_branch表，最后主键关联contact表。
rows返回的都非常少，看不到有什么异常情况。我们在看一下语句，发现后面有order by + limit组合，会不会是排序量太大搞的？于是我们简化SQL，去掉后面的order by 和 limit，看看到底用了多少记录来排序

```
select
  count(*)
from
   contact c
inner join
   contact_branch cb
      on  c.id = cb.contact_id
inner join
   branch_user bu
      on  cb.branch_id = bu.branch_id
      and bu.status in (
         1,
      2)
   inner join
      org_emp_info oei
         on  oei.data_id = bu.user_id
         and oei.node_left >= 2875
         and oei.node_right <= 10802
         and oei.org_category = - 1
+----------+
| count(*) |
+----------+
|   778878 |
+----------+
1 row in set (5.19 sec)
```

发现排序之前居然锁定了778878条记录，如果针对70万的结果集排序，将是灾难性的，怪不得这么慢，那我们能不能换个思路，先根据contact的created_time排序，再来join会不会比较快呢？
于是改造成下面的语句，也可以用straight_join来优化

```
select
c.id,
c.name,
c.position,
c.sex,
c.phone,
c.office_phone,
c.feature_info,
c.birthday,
c.creator_id,
c.is_keyperson,
c.giveup_reason,
c.status,
c.data_source,
from_unixtime(c.created_time) as created_time,
from_unixtime(c.last_modified) as last_modified,
```

```
c.last_modified_user_id
from
contact c
where
exists (
select
1
from
contact_branch cb
inner join
branch_user bu
on cb.branch_id = bu.branch_id
and bu.status in (
1,
2)
inner join
org_emp_info oei
on oei.data_id = bu.user_id
and oei.node_left >= 2875
and oei.node_right <= 10802
and oei.org_category = - 1
where
c.id = cb.contact_id
)
order by
c.created_time desc limit 0 ,
10;
```

验证一下效果 预计在1ms内，提升了13000多倍!

```sql
10 rows in set (0.00 sec)
```

本以为至此大工告成，但我们在前面的分析中漏了一个细节，先排序再join和先join再排序理论上开销是一样的，为何提升这么多是因为有一个limit！大致执行过程是：mysql先按索引排序得到前10条记录，然后再去join过滤，当发现不够10条的时候，再次去10条，再次join，这显然在内层join过滤的数据非常多的时候，将是灾难的，极端情况，内层一条数据都找不到，mysql还傻乎乎的每次取10条，几乎遍历了这个数据表!
用不同参数的SQL试验下

```
select
   sql_no_cache   c.id,
   c.name,
   c.position,
   c.sex,
   c.phone,
   c.office_phone,
   c.feature_info,
   c.birthday,
   c.creator_id,
   c.is_keyperson,
   c.giveup_reason,
   c.status,
   c.data_source,
   from_unixtime(c.created_time) as created_time,
   from_unixtime(c.last_modified) as last_modified,
   c.last_modified_user_id
from
   contact c
where
   exists (
      select
         1
      from
         contact_branch cb
      inner join
         branch_user bu
            on  cb.branch_id = bu.branch_id
            and bu.status in (
               1,
            2)
         inner join
            org_emp_info oei
               on  oei.data_id = bu.user_id
```

```
                and oei.node_left >= 2875
                and oei.node_right <= 2875
                and oei.org_category = - 1
          where
             c.id = cb.contact_id
       )
    order by
       c.created_time desc  limit 0 ,
       10;
 Empty set (2 min 18.99 sec)
```

2 min 18.99 sec！比之前的情况还糟糕很多。由于mysql的nested loop机制，遇到这种情况，基本是无法优化的。这条语句最终也只能交给应用系统去优化自己的逻辑了。
通过这个例子我们可以看到，并不是所有语句都能优化，而往往我们优化时，由于SQL用例回归时落掉一些极端情况，会造成比原来还严重的后果。所以，第一：不要指望所有语句都能通过SQL优化，第二：不要过于自信，只针对具体case来优化，而忽略了更复杂的情况。

慢查询的案例就分析到这儿，以上只是一些比较典型的案例。我们在优化过程中遇到过超过1000行，涉及到16个表join的“垃圾SQL”，也遇到过线上线下数据库差异导致应用直接被慢查询拖死，也遇到过varchar等值比较没有写单引号，还遇到过笛卡尔积查询直接把从库搞死。再多的案例其实也只是一些经验的积累，如果我们熟悉查询优化器、索引的内部原理，那么分析这些案例就变得特别简单了。

写在后面的话

本文以一个慢查询案例引入了MySQL索引原理、优化慢查询的一些方法论;并针对遇到的典型案例做了详细的分析。其实做了这么长时间的语句优化后才发现，任何数据库层面的优化都抵不上应用系统的优化，同样是MySQL，可以用来支撑Google/FaceBook/Taobao应用，但可能连你的个人网站都撑不住。套用最近比较流行的话：“查询容易，优化不易，且写且珍惜！”

参考

参考文献如下：
1.《高性能MySQL》
2.《数据结构与算法分析》

[2014-06-30 18:35 | mysql , database , slow query , btree , explain]

Presto实现原理和美团的使用实践 »