**MySQL索引优化分析**

为什么你写的sql查询慢？为什么你建的索引常失效？通过本章内容，你将学会MySQL性能下降的原因，索引的简介，索引创建的原则，explain命令的使用，以及explain输出字段的意义。助你了解索引，分析索引，使用索引，从而写出更高性能的sql语句。还在等啥子？撸起袖子就是干！

**案例分析**

我们先简单了解一下**非关系型数据库**和**关系型数据库**的区别。  
MongoDB是NoSQL中的一种。NoSQL的全称是Not only SQL，非关系型数据库。它的特点是**性能高**，**扩张性强**，**模式灵活**，在高并发场景表现得尤为突出。但目前它还只是关系型数据库的补充，它在数据的一致性，数据的安全性，查询的复杂性问题上和关系型数据库还存在一定差距。  
MySQL是关系性数据库中的一种，**查询功能强**，**数据一致性高**，**数据安全性高**，**支持二级索引**。但性能方面稍逊与MongoDB，特别是百万级别以上的数据，很容易出现查询慢的现象。这时候需要分析查询慢的原因，一般情况下是程序员sql写的烂，或者是没有键索引，或者是索引失效等原因导致的。  
公司ERP系统数据库主要是MongoDB（最接近关系型数据的NoSQL），其次是Redis，MySQL只占很少的部分。现在又重新使用MySQL，归功于阿里巴巴的奇门系统和聚石塔系统。考虑到订单数量已经是百万级以上，对MySQL的性能分析也就显得格外重要。

我们先通过两个简单的例子来入门。后面会详细介绍各个参数的作用和意义。  
说明：需要用到的sql已经放在了github上了，喜欢的同学可以点一下star，哈哈。<https://github.com/ITDragonBlog/daydayup/tree/master/MySQL/>

**场景一：订单导入，通过交易号避免重复导单**

业务逻辑：订单导入时，为了避免重复导单，一般会通过交易号去数据库中查询，判断该订单是否已经存在。

**最基础的sql语句**

mysql> select \* from itdragon\_order\_list where transaction\_id = "81X97310V32236260E";

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

| id | transaction\_id | gross | net | stock\_id | order\_status | descript | finance\_descript | create\_type | order\_level | input\_user | input\_date |

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

| 10000 | 81X97310V32236260E | 6.6 | 6.13 | 1 | 10 | ok | ok | auto | 1 | itdragon | 2017-08-18 17:01:49 |

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

mysql> explain select \* from itdragon\_order\_list where transaction\_id = "81X97310V32236260E";

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | ALL | NULL | NULL | NULL | NULL | 3 | 33.33 | Using where |

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

查询的本身没有任何问题，在线下的测试环境也没有任何问题。可是，功能一旦上线，查询慢的问题就迎面而来。几百上千万的订单，用全表扫描？啊？哼!  
怎么知道该sql是全表扫描呢？通过explain命令可以清楚MySQL是如何处理sql语句的。打印的内容分别表示：  
**id** : 查询序列号为1。  
**select\_type** : 查询类型是简单查询，简单的select语句没有union和子查询。  
**table** : 表是 itdragon\_order\_list。  
**partitions** : 没有分区。  
**type** : 连接类型，all表示采用全表扫描的方式。  
**possible\_keys** : 可能用到索引为null。  
**key** : 实际用到索引是null。  
**key\_len** : 索引长度当然也是null。  
**ref** : 没有哪个列或者参数和key一起被使用。  
**Extra** : 使用了where查询。  
因为数据库中只有三条数据，所以rows和filtered的信息作用不大。这里需要重点了解的是type为ALL，全表扫描的性能是最差的，假设数据库中有几百万条数据，在没有索引的帮助下会异常卡顿。

**初步优化：为transaction\_id创建索引**

mysql> create unique index idx\_order\_transaID on itdragon\_order\_list (transaction\_id);

mysql> explain select \* from itdragon\_order\_list where transaction\_id = "81X97310V32236260E";

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | const | idx\_order\_transaID | idx\_order\_transaID | 453 | const | 1 | 100 | NULL |

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

这里创建的索引是唯一索引，而非普通索引。  
唯一索引打印的type值是const。表示通过索引一次就可以找到。即找到值就结束扫描返回查询结果。  
普通索引打印的type值是ref。表示非唯一性索引扫描。找到值还要继续扫描，直到将索引文件扫描完为止。(这里没有贴出代码)  
显而易见，const的性能要远高于ref。并且根据业务逻辑来判断，创建唯一索引是合情合理的。

**再次优化：覆盖索引**

mysql> explain select transaction\_id from itdragon\_order\_list where transaction\_id = "81X97310V32236260E";

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | const | idx\_order\_transaID | idx\_order\_transaID | 453 | const | 1 | 100 | Using index |

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

这里将select \* from 改为了 select transaction\_id from 后  
Extra 显示 Using index，表示该查询使用了覆盖索引，这是一个非常好的消息，说明该sql语句的性能很好。若提示的是Using filesort(使用内部排序)和Using temporary(使用临时表)则表明该sql需要立即优化了。  
根据业务逻辑来的，查询结构返回transaction\_id 是可以满足业务逻辑要求的。

**场景二，订单管理页面，通过订单级别和订单录入时间排序**

业务逻辑：优先处理订单级别高，录入时间长的订单。  
既然是排序，首先想到的应该是order by， 还有一个可怕的 Using filesort 等着你。

**最基础的sql语句**

mysql> explain select \* from itdragon\_order\_list order by order\_level,input\_date;

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | ALL | NULL | NULL | NULL | NULL | 3 | 100 | Using filesort |

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

首先，采用全表扫描就不合理，还使用了文件排序Using filesort，更加拖慢了性能。  
MySQL在4.1版本之前文件排序是采用双路排序的算法，由于两次扫描磁盘，I/O耗时太长。后优化成单路排序算法。其本质就是用空间换时间，但如果数据量太大，buffer的空间不足，会导致多次I/O的情况。其效果反而更差。与其找运维同事修改MySQL配置，还不如自己乖乖地建索引。

**初步优化：为order\_level,input\_date 创建复合索引**

mysql> create index idx\_order\_levelDate on itdragon\_order\_list (order\_level,input\_date);

mysql> explain select \* from itdragon\_order\_list order by order\_level,input\_date;

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | ALL | NULL | NULL | NULL | NULL | 3 | 100 | Using filesort |

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

创建复合索引后你会惊奇的发现，和没创建索引一样？？？都是全表扫描，都用到了文件排序。是索引失效？还是索引创建失败？我们试着看看下面打印情况

mysql> explain select order\_level,input\_date from itdragon\_order\_list order by order\_level,input\_date;

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | index | NULL | idx\_order\_levelDate | 68 | NULL | 3 | 100 | Using index |

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

将select \* from 换成了 select order\_level,input\_date from 后。type从all升级为index，表示（full index scan）全索引文件扫描，Extra也显示使用了覆盖索引。可是不对啊！！！！检索虽然快了，但返回的内容只有order\_level和input\_date 两个字段，让业务同事怎么用？难道把每个字段都建一个复合索引？  
MySQL没有这么笨，可以使用force index 强制指定索引。在原来的sql语句上修改 force index(idx\_order\_levelDate) 即可。

mysql> explain select \* from itdragon\_order\_list force index(idx\_order\_levelDate) order by order\_level,input\_date;

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

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

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

| 1 | SIMPLE | itdragon\_order\_list | NULL | index | NULL | idx\_order\_levelDate | 68 | NULL | 3 | 100 | NULL |

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

**再次优化：订单级别真的要排序么？**

其实给订单级别排序意义并不大，给订单级别添加索引意义也不大。因为order\_level的值可能只有，低，中，高，加急，这四种。对于这种重复且分布平均的字段，排序和加索引的作用不大。  
我们能否先固定 order\_level 的值，然后再给 input\_date 排序？如果查询效果明显，是可以推荐业务同事使用该查询方式。

mysql> explain select \* from itdragon\_order\_list where order\_level=3 order by input\_date;

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

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

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

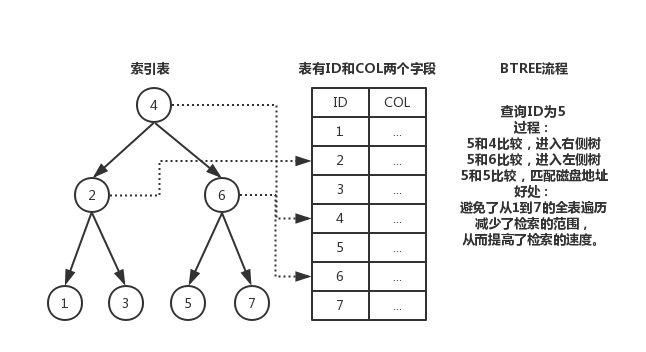
| 1 | SIMPLE | itdragon\_order\_list | NULL | ref | idx\_order\_levelDate | idx\_order\_levelDate | 5 | const | 1 | 100 | Using index condition |

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

和之前的sql比起来，type从index 升级为 ref(非唯一性索引扫描)。索引的长度从68变成了5，说明只用了一个索引。ref也是一个常量。Extra 为Using index condition 表示自动根据临界值，选择索引扫描还是全表扫描。总的来说性能远胜于之前的sql。

上面两个案例只是快速入门，我们需严记一点：优化是基于业务逻辑来的。绝对不能为了优化而擅自修改业务逻辑。如果能修改当然是最好的。

**索引简介**

官方定义：索引（Index） 是帮助MySQL高效获取数据的数据结构。  
大家一定很好奇，索引为什么是一种数据结构，它又是怎么提高查询的速度？我们拿最常用的二叉树来分析索引的工作原理。看下面的图片：  


创建索引的优势  
1 提高数据的检索速度，降低数据库IO成本：使用索引的意义就是通过缩小表中需要查询的记录的数目从而加快搜索的速度。  
2 降低数据排序的成本，降低CPU消耗：索引之所以查的快，是因为先将数据排好序，若该字段正好需要排序，则真好降低了排序的成本。

创建索引的劣势  
1 占用存储空间：索引实际上也是一张表，记录了主键与索引字段，一般以索引文件的形式存储在磁盘上。  
2 降低更新表的速度：表的数据发生了变化，对应的索引也需要一起变更，从而减低的更新速度。否则索引指向的物理数据可能不对，这也是索引失效的原因之一。  
3 优质索引创建难：索引的创建并非一日之功，也并非一直不变。需要频繁根据用户的行为和具体的业务逻辑去创建最佳的索引。

**索引分类**

我们常说的索引一般指的是BTree（多路搜索树）结构组织的索引。其中还有聚合索引，次要索引，复合索引，前缀索引，唯一索引，统称索引，当然除了B+树外，还有哈希索引（hash index）等。

**单值索引**：一个索引只包含单个列，一个表可以有多个单列索引  
**唯一索引**：索引列的值必须唯一，但允许有空值  
**复合索引**：一个索引包含多个列，实际开发中推荐使用  
实际开发中推荐使用复合索引，并且单表创建的索引个数建议不要超过五个

基本语法：  
创建：

create [unique] index indexName on tableName (columnName...)

alter tableName add [unique] index [indexName] on (columnName...)

删除：

drop index [indexName] on tableName

查看：

show index from tableName

哪些情况需要建索引：  
1 主键，唯一索引  
2 经常用作查询条件的字段需要创建索引  
3 经常需要排序、分组和统计的字段需要建立索引  
4 查询中与其他表关联的字段，外键关系建立索引

哪些情况不要建索引：  
1 表的记录太少，百万级以下的数据不需要创建索引  
2 经常增删改的表不需要创建索引  
3 数据重复且分布平均的字段不需要创建索引，如 true,false 之类。  
4 频发更新的字段不适合创建索引  
5 where条件里用不到的字段不需要创建索引

**性能分析**

**MySQL 自身瓶颈**

MySQL自身参见的性能问题有磁盘空间不足，磁盘I/O太大，服务器硬件性能低。  
1 CPU：CPU 在饱和的时候一般发生在数据装入内存或从磁盘上读取数据时候  
2 IO：磁盘I/O 瓶颈发生在装入数据远大于内存容量的时候  
3 服务器硬件的性能瓶颈：top,free,iostat 和 vmstat来查看系统的性能状态

**explain 分析sql语句**

使用explain关键字可以模拟优化器执行sql查询语句，从而得知MySQL 是如何处理sql语句。

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

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

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

**id**

select 查询的序列号，包含一组可以重复的数字，表示查询中执行sql语句的顺序。一般有三种情况：  
第一种：id全部相同，sql的执行顺序是由上至下；  
第二种：id全部不同，sql的执行顺序是根据id大的优先执行；  
第三种：id既存在相同，又存在不同的。先根据id大的优先执行，再根据相同id从上至下的执行。

**select\_type**

select 查询的类型，主要是用于区别普通查询，联合查询，嵌套的复杂查询  
**simple**：简单的select 查询，查询中不包含子查询或者union  
**primary**：查询中若包含任何复杂的子查询，最外层查询则被标记为primary  
**subquery**：在select或where 列表中包含了子查询  
**derived**：在from列表中包含的子查询被标记为derived（衍生）MySQL会递归执行这些子查询，把结果放在临时表里。  
**union**：若第二个select出现在union之后，则被标记为union，若union包含在from子句的子查询中，外层select将被标记为：derived  
**union result**：从union表获取结果的select

**partitions**

表所使用的分区，如果要统计十年公司订单的金额，可以把数据分为十个区，每一年代表一个区。这样可以大大的提高查询效率。

**type**

这是一个非常重要的参数，连接类型，常见的有：all , index , range , ref , eq\_ref , const , system , null 八个级别。  
性能从最优到最差的排序：system > const > eq\_ref > ref > range > index > all  
对java程序员来说，若保证查询至少达到range级别或者最好能达到ref则算是一个优秀而又负责的程序员。  
**all**：（full table scan）全表扫描无疑是最差，若是百万千万级数据量，全表扫描会非常慢。  
**index**：（full index scan）全索引文件扫描比all好很多，毕竟从索引树中找数据，比从全表中找数据要快。  
**range**：只检索给定范围的行，使用索引来匹配行。范围缩小了，当然比全表扫描和全索引文件扫描要快。sql语句中一般会有between，in，>，< 等查询。  
**ref**：非唯一性索引扫描，本质上也是一种索引访问，返回所有匹配某个单独值的行。比如查询公司所有属于研发团队的同事，匹配的结果是多个并非唯一值。  
**eq\_ref**：唯一性索引扫描，对于每个索引键，表中有一条记录与之匹配。比如查询公司的CEO，匹配的结果只可能是一条记录，  
**const**：表示通过索引一次就可以找到，const用于比较primary key 或者unique索引。因为只匹配一行数据，所以很快，若将主键至于where列表中，MySQL就能将该查询转换为一个常量。  
**system**：表只有一条记录（等于系统表），这是const类型的特列，平时不会出现，了解即可

**possible\_keys**

显示查询语句可能用到的索引(一个或多个或为null)，不一定被查询实际使用。仅供参考使用。

**key**

显示查询语句实际使用的索引。若为null，则表示没有使用索引。

**key\_len**

显示索引中使用的字节数，可通过key\_len计算查询中使用的索引长度。在不损失精确性的情况下索引长度越短越好。key\_len 显示的值为索引字段的最可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，并不是通过表内检索出的。

**ref**

显示索引的哪一列或常量被用于查找索引列上的值。

**rows**

根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数，值越大越不好。

**extra**

**Using filesort**： 说明MySQL会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。MySQL中无法利用索引完成的排序操作称为“文件排序” 。出现这个就要立刻优化sql。  
**Using temporary**： 使用了临时表保存中间结果，MySQL在对查询结果排序时使用临时表。常见于排序 order by 和 分组查询 group by。 出现这个更要立刻优化sql。  
**Using index**： 表示相应的select 操作中使用了覆盖索引（Covering index），避免访问了表的数据行，效果不错！如果同时出现Using where，表明索引被用来执行索引键值的查找。如果没有同时出现Using where，表示索引用来读取数据而非执行查找动作。  
覆盖索引（Covering Index） ：也叫索引覆盖，就是select 的数据列只用从索引中就能够取得，不必读取数据行，MySQL可以利用索引返回select 列表中的字段，而不必根据索引再次读取数据文件。  
**Using index condition**： 在5.6版本后加入的新特性，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。  
**Using where**： 表明使用了where 过滤  
**Using join buffer**： 表明使用了连接缓存  
**impossible where**： where 语句的值总是false，不可用，不能用来获取任何元素  
**distinct**： 优化distinct操作，在找到第一匹配的元组后即停止找同样值的动作。

**filtered**

一个百分比的值，和rows 列的值一起使用，可以估计出查询执行计划(QEP)中的前一个表的结果集，从而确定join操作的循环次数。小表驱动大表，减轻连接的次数。

通过explain的参数介绍，我们可以得知:  
1 表的读取顺序(id)  
2 数据读取操作的操作类型(type)  
3 哪些索引被实际使用(key)  
4 表之间的引用(ref)  
5 每张表有多少行被优化器查询(rows)

**性能下降的原因**

从程序员的角度  
1 查询语句写的不好  
2 没建索引，索引建的不合理或索引失效  
3 关联查询有太多的join  
从服务器的角度  
1 服务器磁盘空间不足  
2 服务器调优配置参数设置不合理

**总结**

1 索引是排好序且快速查找的数据结构。其目的是为了提高查询的效率。  
2 创建索引后，查询数据变快，但更新数据变慢。  
3 性能下降的原因很可能是索引失效导致。  
4 索引创建的原则，经常查询的字段适合创建索引，频繁需要更新的数据不适合创建索引。  
5 索引字段频繁更新，或者表数据物理删除容易造成索引失效。  
6 擅用 explain 分析sql语句  
7 除了优化sql语句外，还可以优化表的设计。如尽量做成单表查询，减少表之间的关联。设计归档表等。

到这里，MySQL的索引优化分析就结束了，有什么不对的地方，大家可以提出来。如果觉得不错可以点一下推荐。