**1. mysql如何使用索引**

       索引用于快速找出在某个列中有一特定值的行。对相关列使用索引是提高SELECT 操作性能的最佳途径。  
       查询要使用索引最主要的条件是查询条件中需要使用索引关键字，如果是多列 索引，那么只有查询条件使用了多列关键字最左边的前缀时（**前缀索引**），才可以使用索引，否则 将不能使用索引。

       下列情况下， Mysql 不会使用已有的索引：  
      1、如果 mysql 估计使用索引比全表扫描更慢，则不使用索引。例如：如果 key\_part 1均匀分布在 1 和 100 之间，下列查询中使用索引就不是很好：  
        SELECT \* FROM table\_name where key\_part1 > 1 and key\_part1 < 90  
      2、如果使用**heap 表**并且 where 条件中不用＝索引列，**其他 > 、 < 、 >= 、 <= 均不使 用索引（MyISAM和innodb表使用索引）**；

      3、使用or分割的条件，如果or前的条件中的列有索引，后面的列中没有索引，那么涉及到的索引都不会使用。  
      4、如果创建复合索引，如果条件中使用的列不是索引列的第一部分；**（不是前缀索引）  
      4、如果 like 是以％开始；**  
      5、对 where 后边条件为字符串的一定要加引号，字符串如果为数字 mysql 会自动转 为字符串，但是不使用索引。

**2. 查看索引使用情况**

        如果索引正在工作， Handler\_read\_key 的值将很高，这个值代表了一个行被索引值读的次数，很低的值表明增加索引得到的性能改善不高，因为索引并不经常使 用。  
Handler\_read\_rnd\_next 的值高则意味着查询运行低效，并且应该建立索引补救。这个值的含义是在数据文件中读下一行的请求数。如果你正进行大量的表扫描，  
该值较高。通常说明表索引不正确或写入的查询没有利用索引。

       语法：  
        **mysql> show status like 'Handler\_read%';**

**3. 具体优化查询语句**

**1. 查询进行优化，应尽量避免全表扫描**

    对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引

**.**    尝试下面的技巧以避免优化器错选了表扫描：

**·**  使用ANALYZE TABLE*tbl\_name*为扫描的表更新关键字分布。

**·** 对扫描的表使用FORCEINDEX告知MySQL，相对于使用给定的索引表扫描将非常耗时。

            SELECT \* FROM t1, t2 FORCE INDEX (*index\_for\_column*)   WHERE t1.*col\_name*=t2.*col\_name*；

**·** 用--max-seeks-for-key=1000选项启动**mysqld**或使用SET max\_seeks\_for\_key=1000告知优化器假设关键字扫描不会超过1,000次关键字搜索。

 1). 应尽量避免在 where 子句中对字段进行 null 值判断

        否则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 is null

        NULL对于大多数数据库都需要特殊处理，MySQL也不例外，它需要更多的代码，更多的检查和特殊的索引逻辑，有些开发人员完全没有意识到，创建表时NULL是默认值，但大多数时候应该使用NOT NULL，或者使用一个特殊的值，如0，-1作为默  认值。

        不能用null作索引，任何包含null值的列都将不会被包含在索引中。即使索引有多列这样的情况下，只要这些列中有一列含有null，该列    就会从索引中排除。也就是说如果某列存在空值，即使对该列建索引也不会提高性能。 任何在where子句中使用is null或is not null的语句优化器是不允许使用索引的。

        此例可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

         select id    from t where num=0

 2). 应尽量避免在 where 子句中使用!=或<>操作符

         否则将引擎放弃使用索引而进行全表扫描。  
         MySQL只有对以下操作符才使用索引：**<，<=，=，>，>=，BETWEEN，IN**，以及某些时候的LIKE。

         可以在LIKE操作中使用索引的情形是指另一个操作数不是以**通配符（%或者\_）开头**的情形。例如:  
         SELECT id FROM  t WHERE col LIKE 'Mich%'; #  这个查询将使用索引，  
         SELECT id FROM  t WHERE col  LIKE '%ike';   #这个查询不会使用索引。

 3). 应尽量避免在 where 子句中使用 or 来连接条件

        否则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=10 or num=20

        可以 使用UNION合并查询： select id from t where num=10 union all select id from t where num=20

       在某些情况下，or条件可以避免全表扫描的。

**1 .where 语句里面如果带有or条件, myisam表能用到索引， innodb不行。**

**2 .必须所有的or条件都必须是独立索引**

[mysql or条件可以使用索引而避免全表](http://blog.csdn.net/hguisu/article/details/7106159)

 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).下面的查询也将导致全表扫描：

        select id from t where name like '%abc%' 或者

        select id from t where name like '%abc' 或者

        若要提高效率，可以考虑全文检索。

        而select id from t where name like 'abc%' 才用到索引

 7). 如果在 where 子句中使用参数，也会导致全表扫描。

       因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推 迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

       select id from t where num=@num

       可以改为强制查询使用索引： select id from t with(index(索引名)) where num=@num

 8). 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

9). 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

      select id from t where datediff(day,createdate,'2005-11-30')=0--‘2005-11-30’

      生成的id 应改为:

      select id from t where name like 'abc%'

      select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'

10).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。

11). 索引字段不是复合索引的前缀索引

       例如 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。

**2 .其他一些注意优化：**

12). 不要写一些没有意义的查询，

        如需要生成一个空表结构：

        select col1,col2 into #t from t where 1=0

        这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样： create table #t(...)

13). 很多时候用 exists 代替 in 是一个好的选择：

       select num from a where num in(select num from b)

       用下面的语句替换：

       select num from a where exists(select 1 from b where num=a.num)

14). 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。

15). 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。

16).应尽可能的避免更新 clustered 索引数据列，

       因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17).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。

18).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。

19).最好不要使用"\*"返回所有： select \* from t ，

     用具体的字段列表代替“\*”，不要返回用不到的任何字段。

**3. 临时表的问题：**

20). 尽量使用表变量来代替临时表。

    如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

21).避免频繁创建和删除临时表，以减少系统表资源的消耗。

22).临时表并不是不可使用，

     适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

23).在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；

     如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

24). 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

**4. 游标的问题：**

25).尽量避免使用游标，

      因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26).使用基于游标的方法或临时表方法之前，

      应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27).与临时表一样，游标并不是不可使用。

     对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28).在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。

      无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

**5. 事务的问题：**

29).尽量避免大事务操作，提高系统并发能力。

**6. 数据量的问题**

30).尽量避免向客户端返回**[大数据](http://lib.csdn.net/base/hadoop" \t "_blank" \o "Hadoop知识库)**量，若数据量过大，应该考虑相应需求是否合理。

**7. COUNT优化：**

31) count(\*) 优于count(1)和count(primary\_key)

　　很多人为了统计记录条数，就使用 count(1) 和 count(primary\_key) 而不是 count(\*) ，他们认为这样性能更好，其实这是一个误区。对于有些场景，这样做可能性能会更差，应为数据库对 count(\*) 计数操作做了一些特别的优化。

32）count(column) 和 count(\*) 是不一样的

　　这个误区甚至在很多的资深工程师或者是 DBA 中都普遍存在，很多人都会认为这是理所当然的。实际上，count(column) 和 count(\*) 是一个完全不一样的操作，所代表的意义也完全不一样。

　　count(column) 是表示结果集中有多少个column字段不为空的记录

　　count(\*) 是表示整个结果集有多少条记录

1)innodb引擎在统计方面和myisam是不同的，Myisam内置了一个计数器，

 Count(\*)在没有查询条件的情况下使用 select count(\*) from table 的时候，Myisam直接可以从计数器中取出数据。而innodb必须全表扫描一次方能得到总的数量

2. 但是当有查询条件的时候，两者的查询效率一致。

4. 主键索引count(\*)的时候之所以慢

InnoDB引擎:

[1]     数据文件和索引文件存储在一个文件中，主键索引默认直接指向数据存储位置。

[2]     二级索引存储指定字段的索引，实际的指向位置是主键索引。当我们通过二级索引统计数据的时候，无需扫描数据文件；而通过主键索引统计数据时，由于主键索引与数据文件存放在一起，所以每次都会扫描数据文件，所以主键索引统计没有二级索引效率高。

[3]     由于主键索引直接指向实际数据，所以当我们通过主键id查询数据时要比通过二级索引查询数据要快。

l  MyAsm引擎

[1]     该引擎把每个表都分为几部分存储，比如用户表，包含user.frm，user.MYD和user.MYI。

[2]     User.frm负责存储表结构

[3]     User.MYD负责存储实际的数据记录，所有的用户记录都存储在这个文件中

[4]     User.MYI负责存储用户表的所有索引，这里也包括主键索引。

**8. 优化order by语句**

     基于索引的排序  
     MySQL的弱点之一是它的排序。虽然MySQL可以在1秒中查询大约15,000条记录，但由于MySQL在查询时最多只能使用一个索引。因此，如果WHERE条件已经占用了索引，那么在排序中就不使用索引了，这将大大降低查询的速度。我们可以看看如下的SQL语句:  
     SELECT \* FROM SALES WHERE NAME = “name” ORDER BY SALE\_DATE DESC;  
     在以上的SQL的WHERE子句中已经使用了NAME字段上的索引，因此，在对SALE\_DATE进行排序时将不再使用索引。为了解决这个问题，我们可以对SALES表建立复合索引:  
     ALTER TABLE SALES DROP INDEX NAME, ADD INDEX (NAME,SALE\_DATE)  
     这样再使用上述的SELECT语句进行查询时速度就会大副提升。但要注意，在使用这个方法时，要确保WHERE子句中没有排序字段，在上例中就是不能用SALE\_DATE进行查询，否则虽然排序快了，但是SALE\_DATE字段上没有单独的索引，因此查询又会慢下来。

     在某些情况中， MySQL可以使用一个索引来满足 ORDER BY子句，而不需要额外的排序。 where条件和order by使用相同的索引，并且order by 的顺序和索引顺序相 同，并且order by的字段都是升序或者都是降序。例如：下列sql可以使用索引。

     SELECT \* FROM t1 ORDER BY key\_part1,key\_part2,... ;  
     SELECT \* FROM t1 WHERE key\_part1=1 ORDER BY key\_part1 DESC, key\_part2 DESC;  
     SELECT \* FROM t1 ORDER BY key\_part1 DESC, key\_part2 DESC;  
    但是以下情况不使用索引：  
     SELECT \* FROM t1 ORDER BY key\_part1 DESC, key\_part2 ASC ； --order by 的字段混合 ASC 和 DESC  
     SELECT \* FROM t1 WHERE key2=constant ORDER BY key1 ；-- 用于查询行的关键字与 ORDER BY 中所使用的不相同  
     SELECT \* FROM t1 ORDER BY key1, key2 ；-- 对不同的关键字使用 ORDER BY ：

**9. 优化GROUP BY**

      默认情况下， MySQL 排序所有 GROUP BY col1 ， col2 ， .... 。查询的方法如同在查询中指定 ORDER BY col1 ， col2 ， ... 。如果显式包括一个包含相同的列的 ORDER BY  
子句， MySQL 可以毫不减速地对它进行优化，尽管仍然进行排序。如果查询包括 GROUP BY 但你想要避免排序结果的消耗，你可以指定 ORDER BY NULL禁止排序。  
例如 ：  
INSERT INTO foo  SELECT a, COUNT(\*) FROM bar GROUP BY a ORDER BY NULL;

**10. 优化 OR**

具体详解看：[mysql or条件可以使用索引而避免全表](http://blog.csdn.net/hguisu/article/details/7106159" \t "_blank)

**4. Explain解释说明**

explain显示了mysql如何使用索引来处理select语句以及连接表。可以帮助选择更好的索引和写出更优化的查询语句。   
使用方法，在select语句前加上explain就可以了：   
如：

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. explain **select** surname,first\_name form a,b **where** a.id=b.id

分析结果形式如下：   
**table |  type | possible\_keys | key | key\_len  | ref | rows | Extra   
EXPLAIN列的解释**：

**1 table:**

显示这一行的数据是关于哪张表的

**2 type:**

这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为：system、const、eg\_reg、ref、ref\_or\_null、 range、indexhe、 ALL。  
**system:**表仅有一行(=系统表)。这是const联接类型的一个特例  
      **const:(PRIMARY KEY或UNIQUE)**  
            表最多有一个匹配行，它将在查询开始时被读取。因为仅有一行，在这行的列值可被优化器剩余部分认为是常数。  
            const表很快，因为它们只读取一次！  
            const用于用常数值比较PRIMARY KEY或UNIQUE索引的所有部分时。  
            在下面的查询中，tbl\_name可以用于const表：

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** \* **from** tbl\_name **WHERE** primary\_key=1；

**eq\_reg:key**  
  对于每个来自于前面的表的行组合，从该表中读取一行。这可能是最好的联接类型，除了const类型。  
           它用在一个索引的所有部分被联接使用并且索引是UNIQUE或PRIMARY KEY。  
           eq\_ref可以用于使用= 操作符比较的带索引的列。比较值可以为常量或一个使用在该表前面所读取的表的列的表达式。  
  在下面的例子中，MySQL可以使用eq\_ref联接来处理ref\_tables：

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** \* **FROM** ref\_table,other\_table **WHERE** ref\_table.key\_column=other\_table.**column**;
2. **SELECT** \* **FROM** ref\_table,other\_table **WHERE** ref\_table.key\_column\_part1=other\_table.**column**
3. AND ref\_table.key\_column\_part2=1;

        **ref:key**  
 对于每个来自于前面的表的行组合，所有有匹配索引值的行将从这张表中读取。如果联接只使用键的最左边的前缀，  
          或如果键不是UNIQUE或PRIMARY KEY（换句话说，如果联接不能基于关键字选择单个行的话），则使用ref。  
          如果使用的键仅仅匹配少量行，该联接类型是不错的。  
          ref可以用于使用=或<=>操作符的带索引的列。  
 在下面的例子中，MySQL可以使用ref联接来处理ref\_tables：

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** \* **FROM** ref\_table **WHERE** key\_column=expr;
2. **SELECT** \* **FROM** ref\_table,other\_table **WHERE** ref\_table.key\_column=other\_table.**column**;
3. **SELECT** \* **FROM** ref\_table,other\_table **WHERE** ref\_table.key\_column\_part1=other\_table.**column**
4. AND ref\_table.key\_column\_part2=1;

**ref\_or\_null:Or Is null**  
该联接类型如同ref，但是添加了MySQL可以专门搜索包含NULL值的行。在解决子查询中经常使用该联接类型的优化。  
        在下面的例子中，MySQL可以使用ref\_or\_null联接来处理ref\_tables：

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** \* **FROM** ref\_table **WHERE** key\_column=expr OR key\_column **IS** NULL;

**range:=、<>、>、>=、<、<=、IS NULL、<=>、BETWEEN或者IN**  
 只检索给定范围的行，使用一个索引来选择行。key列显示使用了哪个索引。  
          key\_len包含所使用索引的最长关键元素。在该类型中ref列为NULL。  
当使用=、<>、>、>=、<、<=、IS NULL、<=>、BETWEEN或者IN操作符，用常量比较关键字列时，可以使用range：

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** \* **FROM** tbl\_name **WHERE** key\_column = 10;
2. **SELECT** \* **FROM** tbl\_name **WHERE** key\_column BETWEEN 10 and 20;
3. **SELECT** \* **FROM** tbl\_name **WHERE** key\_column IN (10,20,30);
4. **SELECT** \* **FROM** tbl\_name **WHERE** key\_part1= 10 AND key\_part2 IN (10,20,30);

**indexhe:**  
该联接类型与ALL相同，除了只有索引树被扫描。这通常比ALL快，因为索引文件通常比数据文件小。  
当查询只使用作为单索引一部分的列时，MySQL可以使用该联接类型。  
     **ALL：**  
对于每个来自于先前的表的行组合，进行完整的表扫描。如果表是第一个没标记const的表，  
        这通常不好，并且通常在它情况下很差。通常可以增加更多的索引而不要使用ALL，  
        使得行能基于前面的表中的常数值或列值被检索出。

**3 possible\_keys :**

   显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从WHERE语句中    
   选择一个合适的语句

**4 key ：**

 实际使用的索引。如果为NULL，则没有使用索引。很少的情况下，MYSQL会选择优化不足的索引  。  
 这种情况下，可以在SELECT语句中使用USEINDEX（indexname）来强制使用一个索引或者用IGNORE INDEX（indexname）来强制MYSQL忽略索引

**5key\_len:**

使用的索引的长度。在不损失精确性的情况下，长度越短越好

**6 ref**

显示索引的哪一列被使用了，如果可能的话，是一个常数

**7 rows**

MYSQL认为必须检查的用来返回请求数据的行数 (扫描行的数量)

**8 Extra**

 该列包含MySQL解决查询的详细信息  
 关于MYSQL如何解析查询的额外信息。将在表4.3中讨论，但这里可以看到的坏的例子是Using temporary和Using filesort，  
 意思MYSQL根本不能使用索引，结果是检索会很慢   
  
extra列返回的描述的意义   
  
**Distinct:**  
一旦MYSQL找到了与行相联合匹配的行，就不再搜索了   
**Not exists :**  
MYSQL优化了LEFT JOIN，一旦它找到了匹配LEFT JOIN标准的行， 就不再搜索了   
        面是一个可以这样优化的查询类型的例子：  
SELECT \* FROM t1 LEFT JOIN t2 ON t1.id=t2.id WHERE t2.id IS NULL；  
假定t2.id定义为NOT NULL。在这种情况下，MySQL使用t1.id的值扫描t1并查找t2中的行。  
        如果MySQL在t2中发现一个匹配的行，它知道t2.id绝不会为NULL，并且不再扫描t2内有相同的id值的行。  
        换句话说，对于t1的每个行，MySQL只需要在t2中查找一次，无论t2内实际有多少匹配的行。  
**Range checked for each Record（index map:#）**  
没有找到理想的索引，因此对于从前面表中来的每一个行组合，MYSQL检查使用哪个索引，并用它来从表中返回行。  
        这是使用索引的最慢的连接之一   
        MySQL没有发现好的可以使用的索引，但发现如果来自前面的表的列值已知，可能部分索引可以使用。  
        对前面的表的每个行组合，MySQL检查是否可以使用range或index\_merge访问方法来索取行。  
        关于适用性标准的描述参见7.2.5节，“范围优化”和7.2.6节，“索引合并优化”，  
        不同的是前面表的所有列值已知并且认为是常量。这并不很快，但比执行没有索引的联接要快得多。  
**Using filesort**  
看到这个的时候，查询就需要优化了。MYSQL需要进行额外的步骤来发现如何对返回的行排序。  
        它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行   
**Using index**  
列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，  
        这发生在对表的全部的请求列都是同一个索引的部分的时候   
**Using temporary**  
看到这个的时候，查询需要优化了。这里，MYSQL需要创建一个临时表来存储结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上   
**Using where**  
使用了WHERE从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行，  
        并且连接类型ALL或index，这就会发生，或者是查询有问题   
Impossible WHERE noticed after reading const table...

**5. SQL核心语句(非常实用的几个技巧)**

**1) 插入数据**

**批量插入:**

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **INSERT** mytable (first\_column,second\_column,third\_column)
2. **VALUES** ('some data','some more data','yet more data') ,
3. **VALUES** ('some data','some more data','yet more data') ,
4. **VALUES** ('some data','some more data','yet more data')

**2）.清空数据表**

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **TRUNCATE** **TABLE**  `mytable`

注意：删除表中的所有记录，应使用TRUNCATE TABLE语句。注意这里为什么要用TRUNCATE TABLE语句代替DELETE语句:当你使用TRUNCATE TABLE语句时，记录的删除是不作记录的。也就是说，这意味着TRUNCATE TABLE要比DELETE快得多。

**3）用SELECT创建记录和表**

　　INSERT语句与DELETE语句和UPDATE语句有一点不同，它一次只操作一个记录。然而，有一个方法可以使INSERT 语句一次添加多个记录。要作到这一点，你需要把INSERT语句与SELECT语句结合起来，象这样:

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **INSERT** mytable(first\_column,second\_column)
2. **SELECT** another\_first,another\_second  **FROM** anothertable **WHERE** another\_first='Copy Me!';

　　这个语句从anothertable拷贝记录到mytable.只有表anothertable中字段another\_first的值为'Copy Me!'的记录才被拷贝。

　　当为一个表中的记录建立备份时，这种形式的INSERT语句是非常有用的。在删除一个表中的记录之前，你可以先用这种方法把它们拷贝到另一个表中。

　　如果你需要拷贝整个表，你可以使用SELECT INTO语句。例如，下面的语句创建了一个名为newtable的新表，该表包含表mytable的所有数据:

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** \* **INTO** newtable **FROM** mytable;

　　你也可以指定只有特定的字段被用来创建这个新表。要做到这一点，只需在字段列表中指定你想要拷贝的字段。另外，你可以使用WHERE子句来限制拷贝到新表中的记录。下面的例子只拷贝字段second\_columnd的值等于'Copy Me!'的记录的first\_column字段。

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/5731629" \o "view plain) [copy](http://blog.csdn.net/hguisu/article/details/5731629" \o "copy)

[print](http://blog.csdn.net/hguisu/article/details/5731629" \o "print)[?](http://blog.csdn.net/hguisu/article/details/5731629" \o "?)

1. **SELECT** first\_column **INTO** newtable
2. **FROM** mytable
3. **WHERE** second\_column='Copy Me!';

　　使用SQL修改已经建立的表是很困难的。例如，如果你向一个表中添加了一个字段，没有容易的办法来去除它。另外，如果你不小心把一个字段的数据类型给错了，你将没有办法改变它。但是，使用本节中讲述的SQL语句，你可以绕过这两个问题。

　　例如，假设你想从一个表中删除一个字段。使用SELECT INTO语句，你可以创建该表的一个拷贝，但不包含要删除的字段。这使你既删除了该字段，又保留了不想删除的数据。

　　如果你想改变一个字段的数据类型，你可以创建一个包含正确数据类型字段的新表。创建好该表后，你就可以结合使用UPDATE语句和SELECT语句，把原来表中的所有数据拷贝到新表中。通过这种方法，你既可以修改表的结构，又能保存原有的数据。