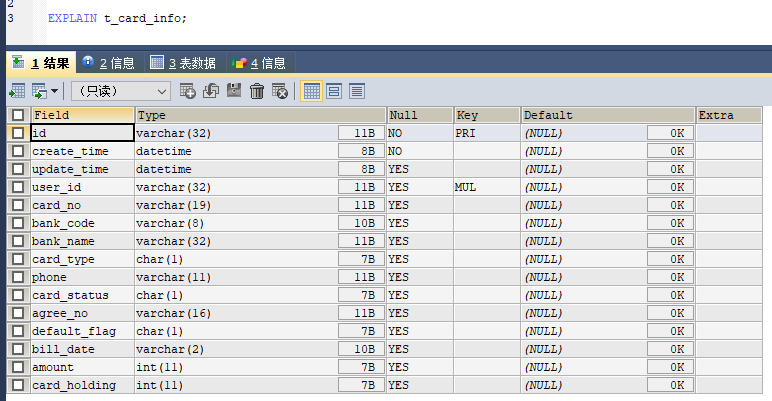
## 一、执行计划

<http://blog.csdn.net/xifeijian/article/details/19773795>

1. EXPLAIN table\_name; // 列出表的字段结构



1. EXPLAIN select\_sql; // 给出查询SQL执行的相关索引信息

展示的各字段有：

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

各字段含义：

**id**：select查询的序列号

**select\_type**：select查询的类型，区分普通查询、联合查询、子查询等，可能的值有：

1) SIMPLE: 查询中不包含子查询或者UNION

2) 查询中若包含任何复杂的子部分，最外层查询则被标记为：PRIMARY

3) 在SELECT或WHERE列表中包含了子查询，该子查询被标记为：SUBQUERY

4) 在FROM列表中包含的子查询被标记为：DERIVED（衍生）

5) 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

6) 从UNION表获取结果的SELECT被标记为：UNION RESULT

**table** :输出的行所引用的表。

**type** :联合查询所使用的类型，表示MySQL在表中找到所需行的方式

结果值从好到坏依次是：

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL ，一般来说，得保证查询至少达到range级别，最好能达到ref.

1. ALL: 扫描全表
2. index: 扫描全部索引树
3. range: 扫描部分索引，索引范围扫描，对索引的扫描开始于某一点，返回匹配值域的行，常见于between、<、>等的查询
4. ref: 非唯一性索引扫描，返回匹配某个单独值的所有行。常见于使用非唯一索引进行的查找
5. eq\_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描
6. const, system: 当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量。system是const类型的特例，当查询的表只有一行的情况下， 使用system。
7. NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引。

**possible\_keys**:指出MySQL能使用哪个索引在该表中找到行。查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用。如果是空的，没有相关的索引。这时要提高性能，可通过检验WHERE子句，看是否引用某些字段，或者检查字段不是适合索引。

**key** :显示MySQL实际决定使用的键。如果没有索引被选择，键是NULL。

**key\_len**:显示MySQL决定使用的键长度。表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。如果键是NULL，长度就是NULL。文档提示特别注意这个值可以得出一个多重主键里mysql实际使用了哪一部分。

**ref**:显示哪个字段或常数与key一起被使用。

**rows**:这个数表示mysql要遍历多少数据才能找到，表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数，在innodb上可能是不准确的。

**Extra**:包含不适合在其他列中显示但十分重要的额外信息。

1. Only index，这意味着信息只用索引树中的信息检索出的，这比扫描整个表要快。
2. using where是使用上了where限制，表示MySQL服务器在存储引擎收到记录后进行“后过滤”（Post-filter），如果查询未能使用索引，Using where的作用只是提醒我们MySQL将用where子句来过滤结果集。
3. impossible where 表示用不着where，一般就是没查出来啥。
4. Using filesort（MySQL中无法利用索引完成的排序操作称为“文件排序”）当我们试图对一个没有索引的字段进行排序时，就是filesoft。它跟文件没有任何关系，实际上是内部的一个快速排序。
5. Using temporary（表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询），使用filesort和temporary的话会很吃力，WHERE和ORDER BY的索引经常无法兼顾，如果按照WHERE来确定索引，那么在ORDER BY时，就必然会引起Using filesort，这就要看是先过滤再排序划算，还是先排序再过滤划算。

## 二、索引

<https://www.cnblogs.com/manmanlu/p/6274487.html>

<https://yq.aliyun.com/articles/38328>

<http://blog.jobbole.com/86594/>

## 3.索引创建的几个原则

（1）适合索引的列是出现在WHERE 子句中的列  
最适合索引的列是出现在WHERE 子句中的列，或连接子句中指定的列，而不是出现在SELECT 关键字后的选择列表中的列。  
（2）使用惟一索引  
考虑某列中值的分布。对于惟一值的列，索引的效果最好，而具有多个重复值的列，其索引效果最差。例如，存放年龄的列具有不同值，很容易区分 各行。而用来记录性别的列，只含有“ M”和“F”，则对此列进行索引没有多大用处。  
（3）使用短索引  
如果对串列进行索引，应该指定一个前缀长度，只要有可能就应该这样做。例如，如果有一个CHAR(200) 列，如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。对前10 个或20 个字符进行索引能够节省大量索引空间，也可能会使查询更快。较小的索引涉及的磁盘I/O 较少，较短的值比较起来更快。更为重要的是，对于较短的键值，索引高速缓存中的块能容纳更多的键值，因此，MySQL也可以在内存中容纳更多的值。这增加 了找到行而不用读取索引中较多块的可能性。（当然，应该利用一些常识。如仅用列值的第一个字符进行索引是不可能有多大好处的，因为这个索引中不会有许多不 同的值。）  
（4）利用最左前缀  
在创建一个n 列的索引时，实际是创建了MySQL可利用的n 个索引。多列索引可起几个索引的作用，因为可利用索引中最左边的列集来匹配行。这样的列集称为最左前缀。（这与索引一个列的前缀不同，索引一个列的前缀是利用该的前n 个字符作为索引值。）  
（5）不要过度索引  
不要以为索引“越多越好”，什么东西都用索引是错的。每个额外的索引都要占用额外的磁盘空间，并降低写操作的性能，这一点我们前面已经介绍 过。在修改表的内容时，索引必须进行更新，有时可能需要重构，因此，索引越多，所花的时间越长。如果有一个索引很少利用或从不使用，那么会不必要地减缓表 的修改速度。此外，MySQL在生成一个执行计划时，要考虑各个索引，这也要费时间。创建多余的索引给查询优化带来了更多的工作。索引太多，也可能会使 MySQL选择不到所要使用的最好索引。只保持所需的索引有利于查询优化。如果想给已索引的表增加索引，应该考虑所要增加的索引是否是现有多列索引的最左 索引。如果是，则就不要费力去增加这个索引了，因为已经有了。  
（6）考虑在列上进行的比较类型。  
索引可用于“ <”、“ < = ”、“ = ”、“ > =”、“ > ”和BETWEEN 运算。在模式具有一个直接量前缀时，索引也用于LIKE 运算。如果只将某个列用于其他类型的运算时（如STRCMP( )），对其进行索引没有价值。

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。

## 4.创建索引的时机

**（1）什么时候创建索引**  
较频繁地作为查询条件的字段，也就是说最适合索引的列是出现在WHERE 子句中的列，或连接子句中指定的列，而不是出现在SELECT 关键字后的选择列表中的列。  
**（2）什么时候不创建索引**  
**表记录太少：**  
如果一个表只有5条记录，采用索引去访问记录的话，那首先需访问索引表，再通过索引表访问数据表，一般索引表与数据表不在同一个数据块，这种情况下ORACLE至少要往返读取数据块两次。而不用索引的情况下ORACLE会将所有的数据一次读出，处理速度显然会比用索引快。  
**唯一性太差的字段：**如状态字段、类型字段。那些只存储固定几个值的字段，例如用户登录状态、消息的status等。

这个涉及到了索引扫描的特性。例如：通过索引查找键值为A和B的某些数据，通过A找到某条相符合的数据，这条数据在X页上面，然后继续扫描，又发现符合A的数据出现在了Y页上面，那么存储引擎就会丢弃X页面的数据，然后存储Y页面上的数据，一直到查找完所有对应A的数据，然后查找B字段，发现X页面上面又有对应B字段的数据，那么他就会再次扫描X页面，等于X页面就会被扫描2次甚至多次。以此类推，所以同一个数据页可能会被多次重复的读取，丢弃，在读取，这无疑给存储引擎极大地增加了IO的负担。

**更新太频繁地字段不适合创建索引：**  
当你为这个字段创建索引时候，当你再次更新这个字段数据时，数据库会自动更新他的索引，所以当这个字段更新太频繁地时候那么就是不断的更新索引。  
如果一个字段同一个时间段内被更新多次，那么不能为他建立索引。

## 5.索引失效的几种情况

（1）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   
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）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。   
（3）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，

|  |  |
| --- | --- |
| 1 | select id from t where num=10 or num=20 |

可以这样查询：

|  |  |
| --- | --- |
| 1  2  3 | select id from t where num=10  union all  select id from t where num=20 |

（4）in 和 not in 也要慎用，否则会导致全表扫描，如：

|  |  |
| --- | --- |
| 1 | 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：

|  |  |
| --- | --- |
| 1 | select id from t where num between 1 and 3 |

下面的查询也将导致全表扫描：   
select id from t where name like '%abc%'   
若要提高效率，可以考虑全文检索。

（5）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：

|  |  |
| --- | --- |
| 1  2 | select id from t where substring(name,1,3)='abc'--name以abc开头的id  select id from t where datediff(day,createdate,'2005-11-30')=0--‘2005-11-30’生成的id |

应改为:

|  |  |
| --- | --- |
| 1  2 | select id from t where name like 'abc%'  select id from t where createdate>='2005-11-30' and createdate<'2005-12-1' |

索引失效的情况还有很多，其他的还有使用<>或者单独的>,<；当变量采用的是times变量，而表的字段采用的是date变量时等

## 6.索引类型

A）聚集索引，表数据按照索引的顺序来存储的。对于聚集索引，叶子结点即存储了真实的数据行，不再有另外单独的数据页。  
B）非聚集索引，表数据存储顺序与索引顺序无关。对于非聚集索引，叶结点包含索引字段值及指向数据页数据行的逻辑指针，该层紧邻数据页，其行数量与数据表行数据量一致。

在一张表上只能创建一个聚集索引，因为真实数据的物理顺序只可能是一种。如果一张表没有聚集索引，那么它被称为“堆集”（Heap）。这样的表中的数据行没有特定的顺序，所有的新行将被添加的表的末尾位置。

主键是聚集索引

## 7．聚集索引

在聚集索引中，叶结点也即数据结点，所有数据行的存储顺序与索引的存储顺序一致。

1）聚集索引与查询操作

如上图，我们在名字字段上建立聚集索引，当需要在根据此字段查找特定的记录时，数据库系统会根据特定的系统表查找的此索引的根，然后根据指针查找下一个，直到找到。例如我们要查询“Green”，由于它介于[Bennet,Karsen]，据此我们找到了索引页1007，在该页中“Green”介于[Greane, Hunter]间，据此我们找到叶结点1133（也即数据结点），并最终在此页中找以了目标数据行。

此次查询的IO包括3个索引页的查询（其中最后一次实际上是在数据页中查询）。这里的查找可能是从磁盘读取(Physical Read)或是从缓存中读取(Logical Read)，如果此表访问频率较高，那么索引树中较高层的索引很可能在缓存中被找到。所以真正的IO可能小于上面的情况。

2）聚集索引与插入操作

最简单的情况下，插入操作根据索引找到对应的数据页，然后通过挪动已有的记录为新数据腾出空间，最后插入数据。

如果数据页已满，则需要拆分数据页（页拆分是一种耗费资源的操作，一般数据库系统中会有相应的机制要尽量减少页拆分的次数，通常是通过为每页预留空间来实现）：  
A）在该使用的数据段（extent）上分配新的数据页，如果数据段已满，则需要分配新段。  
B）调整索引指针，这需要将相应的索引页读入内存并加锁。  
C）大约有一半的数据行被归入新的数据页中。  
D）如果表还有非聚集索引，则需要更新这些索引指向新的数据页。

特殊情况：  
A）如果新插入的一条记录包含很大的数据，可能会分配两个新数据页，其中之一用来存储新记录，另一存储从原页中拆分出来的数据。  
B）通常数据库系统中会将重复的数据记录存储于相同的页中。  
C）类似于自增列为聚集索引的，数据库系统可能并不拆分数据页，页只是简单的新添数据页。

3）聚集索引与删除操作

删除行将导致其下方的数据行向上移动以填充删除记录造成的空白。

如果删除的行是该数据页中的最后一行，那么该数据页将被回收，相应的索引页中的记录将被删除。如果回收的数据页位于跟该表的其它数据页相同的段上，那么它可能在随后的时间内被利用。如果该数据页是该段的唯一一个数据页，则该段也被回收。

对于数据的删除操作，可能导致索引页中仅有一条记录，这时，该记录可能会被移至邻近的索引页中，原索引页将被回收，即所谓的“索引合并”。

每个InnoDB表都有一个称为聚簇索引的特殊索引，其中存储了行的数据。 通常，聚簇索引与主键同义。 要从查询，插入和其他数据库操作中获得最佳性能，您必须了解InnoDB如何使用聚簇索引来优化每个表的最常见查找和DML操作。

在表上定义PRIMARY KEY时，InnoDB将其用作聚簇索引。为您创建的每个表定义主键。 如果没有逻辑唯一且非空列或一组列，请添加一个新的自动增量列，其值将自动填充。

如果没有为表定义PRIMARY KEY，MySQL将找到第一个UNIQUE索引，其中所有键列都是NOT NULL，而InnoDB将它用作聚簇索引。

如果表没有PRIMARY KEY或合适的UNIQUE索引，InnoDB会在包含行ID值的合成列内部生成名为GEN\_CLUST\_INDEX的隐藏聚簇索引。 这些行按InnoDB分配给此类表中的行的ID排序。 行ID是一个6字节的字段，在插入新行时会单调增加。 因此，由行ID排序的行在物理上处于插入顺序。

聚簇索引如何加快查询速度？

通过聚簇索引访问行很快，因为索引搜索直接指向包含所有行数据的页面。 如果表很大，则与使用与索引记录不同的页面存储行数据的存储组织相比，聚簇索引体系结构通常会保存磁盘I/O操作。

二级索引如何与聚集指数相关联？

除聚集索引之外的所有索引都称为辅助索引。 在InnoDB中，辅助索引中的每个记录都包含该行的主键列以及为辅助索引指定的列。 InnoDB使用此主键值来搜索聚簇索引中的行。

如果主键很长，则二级索引使用更多空间，因此使用短主键是有利的。

## 8．非聚集索引

非聚集索引与聚集索引相比：  
A）叶子结点并非数据结点  
B）叶子结点为每一真正的数据行存储一个“键-指针”对  
C）叶子结点中还存储了一个指针偏移量，根据页指针及指针偏移量可以定位到具体的数据行。  
D）类似的，在除叶结点外的其它索引结点，存储的也是类似的内容，只不过它是指向下一级的索引页的。

聚集索引是一种稀疏索引，数据页上一级的索引页存储的是页指针，而不是行指针。而对于非聚集索引，则是密集索引，在数据页的上一级索引页它为每一个数据行存储一条索引记录。

对于根与中间级的索引记录，它的结构包括：  
A）索引字段值  
B）RowId（即对应数据页的页指针+指针偏移量）。在高层的索引页中包含RowId是为了当索引允许重复值时，当更改数据时精确定位数据行。  
C）下一级索引页的指针

对于叶子层的索引对象，它的结构包括：  
A）索引字段值  
B）RowId

1）非聚集索引与查询操作

针对上图，如果我们同样查找“Green”，那么一次查询操作将包含以下IO：3个索引页的读取+1个数据页的读取。同样，由于缓存的关系，真实的IO实际可能要小于上面列出的。

2）非聚集索引与插入操作

如果一张表包含一个非聚集索引但没有聚集索引，则新的数据将被插入到最末一个数据页中，然后非聚集索引将被更新。如果也包含聚集索引，该聚集索引将被用于查找新行将要处于什么位置，随后，聚集索引、以及非聚集索引将被更新。

3）非聚集索引与删除操作

如果在删除命令的Where子句中包含的列上，建有非聚集索引，那么该非聚集索引将被用于查找数据行的位置，数据删除之后，位于索引叶子上的对应记录也将被删除。如果该表上有其它非聚集索引，则它们叶子结点上的相应数据也要删除。

如果删除的数据是该数所页中的唯一一条，则该页也被回收，同时需要更新各个索引树上的指针。

由于没有自动的合并功能，如果应用程序中有频繁的随机删除操作，最后可能导致表包含多个数据页，但每个页中只有少量数据。

## 三、事务

MySQL的数据是绑定到存储引擎中的，由存储引擎提供支持。InnoDB存储引擎默认支持事务。

事务的ACID特性：原子性、一致性、隔离性、持久化

## 四、锁

## 五、查询优化

Query 语句的优化思路和原则主要提现在以下几个方面：

1. 优化更需要优化的Query；

2. 定位优化对象的性能瓶颈；

3. 明确的优化目标；

4. 从 Explain 入手；

5. 多使用profile

6. 永远用小结果集驱动大的结果集；

7. 尽可能在索引中完成排序；

8. 只取出自己需要的Columns；

9. 仅仅使用最有效的过滤条件；

10. 尽可能避免复杂的Join和子查询；

**JOIN的实现原理**

MySQL 中，只有一种 Join 算法，就是大名鼎鼎的 Nested Loop Join，他没有其他很多数据库所提供的 Hash Join，也没有 Sort Merge Join。

顾名思义， Nested Loop Join 实际上就是通过驱动表的结果集作为循环基础数据，然后一条一条的通过该结果集中的数据作为过滤条件到下一个表中查询数据，然后合并结果。如果还有第三个参与 Join，则再通过前两个表的 Join 结果集作为循环基础数据，再一次通过循环查询条件到第三个表中查询数据，如此往复。

**JOIN语句的优化**

1.尽可能减少 Join 语句中的 Nested Loop 的循环总次数：有效的办法只有一个，那就是让驱动表的结果集尽可能的小，即“ 永远用小结果集驱动大的结果集”。

2.优先优化Nested Loop 的内层循环：内层循环是循环中执行次数最多的，每次循环节约很小的资源，在整个循环中就能节约很大的

3. 保证 Join 语句中被驱动表上 Join 条件字段已经被索引：这也正是优化内层循环的实际优化方法。

4. 当无法保证被驱动表的 Join 条件字段被索引且内存资源充足的前提下，不要太吝惜 Join Buffer 的设置：

**GROUP BY 的实现与优化**

Group by需要先进行排序，然后再进行分组。

尽可能让MySQL利用索引来完成group by操作；无法使用索引时，由于需要使用到临时表和filesort，必须有足够的sort\_buffer\_size，否则会出现将临时表数据copy到磁盘上再进行操作。

**Distinct实现和优化**

与Group By实现类似，只不过是在Group By之后每组中只取出一条记录。

思路与Group By一致。

## 六、其他知识点

### 6.1插入更新 ON DUPLICATE KEY UPDATE

使用场景：如果存在则更新,如果不存在则新增

使用前提：待操作的表需要有：unique索引或者 primary key

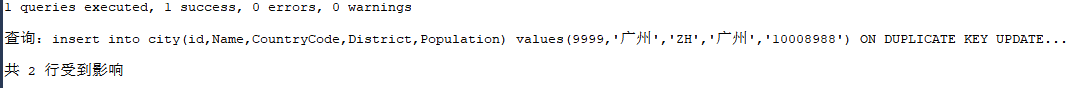
如果没有前提条件，每次执行语句都会进行数据插入

使用例子：INSERT INTO city(id,NAME,CountryCode,District,Population) VALUES(9999,'广州','ZH','广州','10008988') ON DUPLICATE KEY UPDATE District = '广州223';

使用说明：

1）有唯一性表示的列，必须出现在Insert语句中

2）不存在记录时，会根据唯一键值执行Update语句的更新，并且受影响的行数为2行



### 6.2 替换更新：Replace Into

使用场景：存在记录，则先删除再插入

使用前提：需要有：unique索引或者 primary key

如果没有前提条件，Replace Into和Insert Into是一样的

使用例子：REPLACE INTO city(id,NAME,CountryCode,District,Population) VALUES(9999,'广州456','ZH','广州','10008988');

使用说明：

1. 为了能够使用REPLACE，必须同时拥有表的INSERT和DELETE权限
2. REPLACE语句会返回一个数，来指示受影响的行的数目。该数是被删除和被插入的行数的和