

共享锁和排他所 --- 读锁和写锁

读锁是共享的，写锁是排他的

表锁：

开销最小，一个用户对表进行写操作（插入、删除、更新）前，需要先获得写锁，这会阻塞其他用户对该表的所有读写操作。只有没有写锁时，其他读取的用户才能获取读锁，读锁之间是不相互阻塞的。另外写锁也比读锁有更高的优先级，因此一个写锁可能会被插入到读锁队列的前面，反之，读锁则不能插入到写锁的前面。

行锁：

可以最大程度地支持并发处理，但是也带来了最大的锁开销。行锁只在存储引擎层实现，而Mysql服务层没有实现。服务层完全不了解存储引擎中的锁实现。

未提交读：

事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读。这个级别会导致很多问题。

提交读：

一个事务开始时，只能“看见”已经提交的事务所做的修改。也就是，一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。大多数数据库系统默认的隔离级别是提交读，但MySQL不是。这个级别有时候也叫做不可重复读，因为两次执行同样的插叙，可能会得到不一样的结果。

可重复读：

解决了脏读的问题。该级别保证在同一事务中多次读取同样记录的结果是一致的。但是无法解决幻读的问题。所谓幻读，指的是当某个事务在读取某个范围的记录时，另一个事务又在该范围内插入新的记录，当之前的事务再次读取该范围内的记录时，会产生幻行。

可串行化：最高的隔离级别，通过强制事务串行化，避免了幻读的问题。

SQL的隔离级别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 隔离级别 | 脏读可能性 | 不可重复读可能性 | 幻读可能性 | 加锁读 |
| READ UNCOMMITTED | YES | YES | YES | NO |
| READ COMMITTED | NO | YES | YES | NO |
| REPEATABLE READ | NO | NO | YES | NO |
| SERIALIZABLE | NO | NO | NO | YES |

死锁：

死锁是指两个或者多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环的现象。当多个事务试图以不同的顺序锁定资源时，就可能产生死锁。多个事务同时锁定同一个资源时，也会产生死锁。解决办法：死锁检测和死锁超时检测。

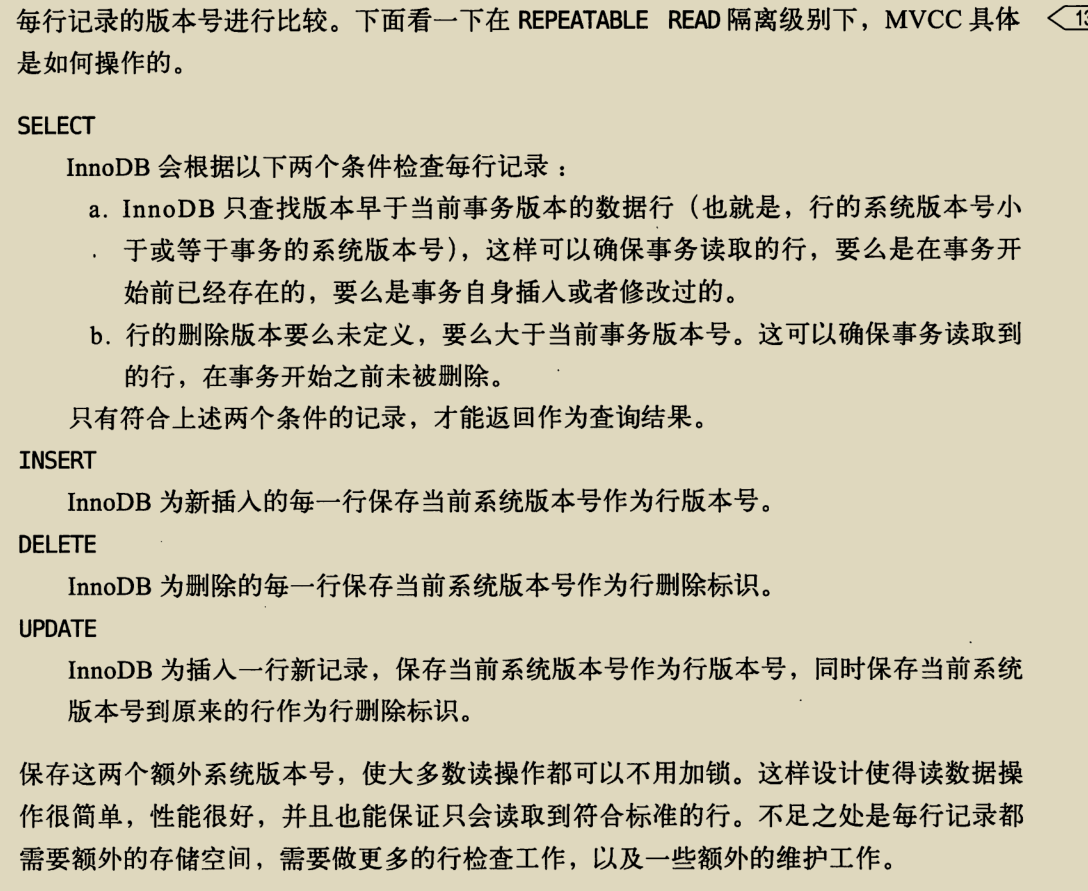
锁的行为和顺序是和存储引擎相关的。以同样的顺序执行语句，有些存储引擎会产生死锁，有些则不会。死锁的产生有双重原因：有些是因为真正的数据冲突，这种情况通常很难避免，但有些则完全是由于存储引擎的实现方式导致的。死锁发生以后，只有部分或者完全回滚其中一个事务，才能打破死锁。

MYSQL中的事务：

1. 自动提交
2. 在事务中混合使用存储引擎

多版本并发控制（MVCC-行级锁的变种）：

通过在每行记录后面保存两个隐藏的列来实现。这两个列，一个保存了创建时间，一个保存了过期时间（或者删除时间）。当然储存的并不是实际的时间值，而是系统的版本号。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本会作为事务的版本号，用来和查询到的每行记录的版本进行比较。



两种性能剖析：

1. 基于执行时间的分析：什么任务的执行时间最长
2. 基于等待的分析：判断任务在什么地方被阻塞的时间最长

剖析MySql查询

1.剖析单条查询语句

SHOW PROFILES

SHOW PROFILE FOR QUERY 54

SHOW STATUS

诊断间隙性问题

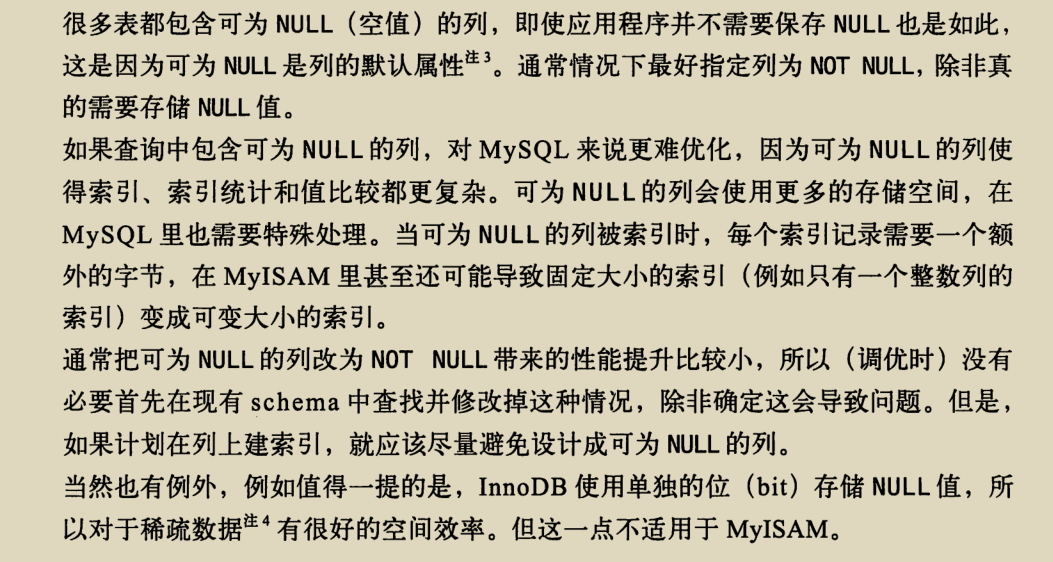
4.Schema与数据类型优化

4.1 选择优化的数据类型

更小的通常更好

简单就好

尽量避免NULL值：



1.LEFT（str,length）从左截取指定长度的字符串。

2.ON DUPLICATE KEY UPDATE：当insert已经存在的记录时，执行Update。

3.IF(expr1,expr2,expr3)：expr1为true，返回expr2,否则返回expr3

4.sql/92标准可以使用using关键字来简化连接查询，但是只是在查询满足下面两个条件时，才能使

用using关键字进行简化。

1.查询必须是等值连接。

2.等值连接中的列必须具有相同的名称和数据类型。

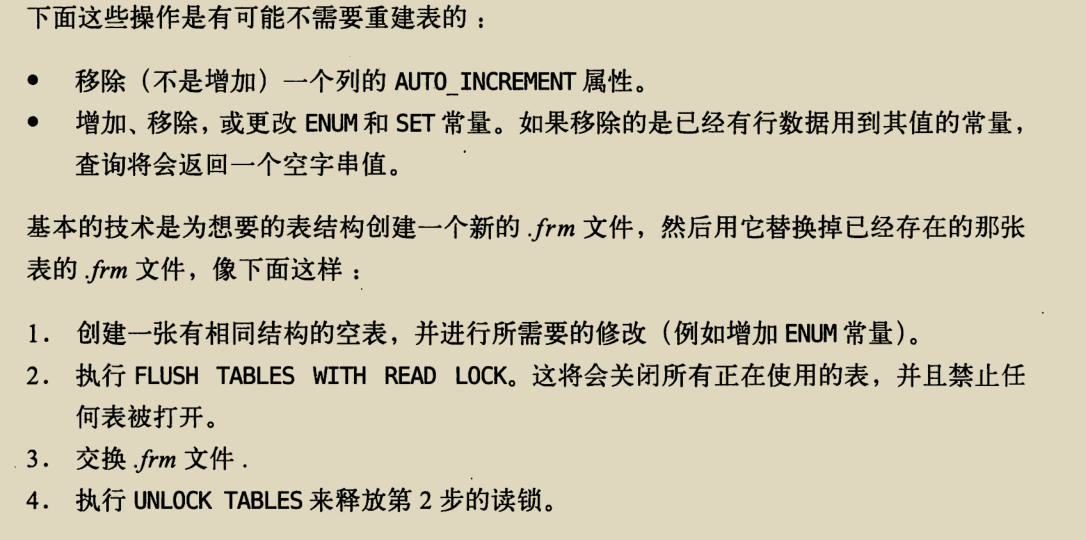
4.5加快alter table 的操作速度

常见的场景，能使用的技巧只有两种：

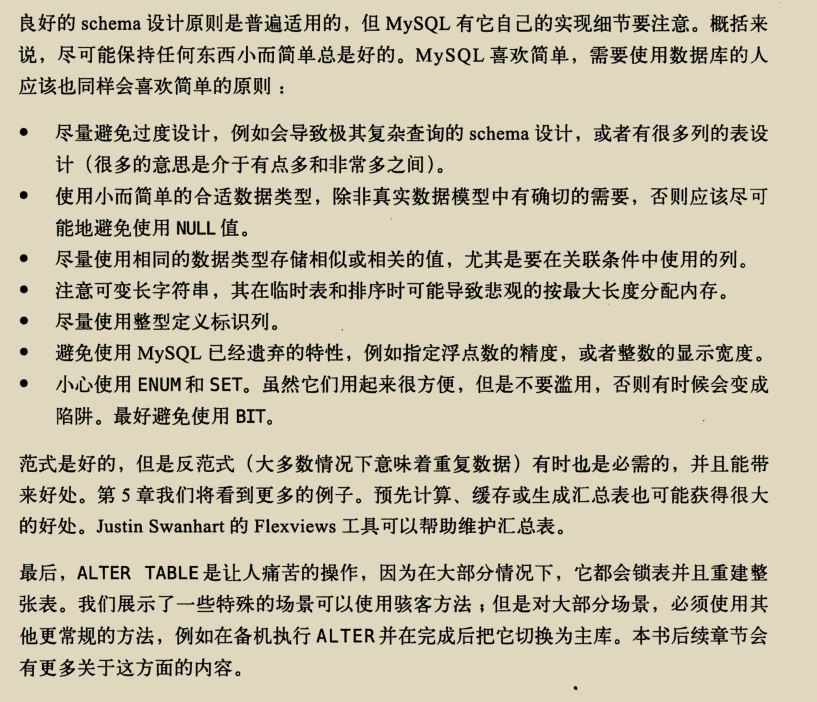
1.先是在一台不提供服务的机器上执行alter table操作，然后和提供服务的主库进行切换

2.影子拷贝：用要求的表结构创建一张和源表无关的新表，然后通过重命名和删除操作交换两张表。

4.5.1 只修改.frm文件



4.6 总结



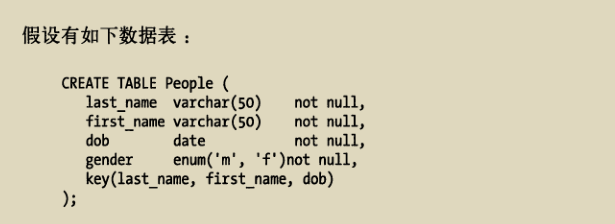
5 创建高性能的索引

索引（在mysql中也叫做键-key）是存储引擎用于快速找到记录的一种数据结构。

索引可以包含一个或者多个列的值。如果索引包含多个列，那么列的顺序也十分重要，因为MySQL只能高效地使用索引最前缀列。创建一个包含两个列的索引，和创建两个包含一列的索引是大不相同的。

5.1 .1索引的类型

在MySQL 中，索引是在存储引擎层而不是服务器实现的。所以并没有统一的标准：不同的存储引擎的索引的工作方式并不一样，也不是所有的存储引擎都支持所有类型的索引。即使多个存储引擎支持统一类型的索引，其底层的实现也可能不同。



B-Tree索引

通常意味着所有的值都是按照顺序存储的，每一个叶子页到根的距离相同。B-Tree索引能够快速加快访问数据的速度，因为存储引擎不再需要进行全表扫描来获取需要的数据，取而代之的是从索引的根节点开始进行搜索。根节点的槽中存放了指向子节点的指针，存储引擎根据这些指针向下层查找。通过比较节点页的值和要查找的值，可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限，最终存储引擎要么找不到对应的值，要么记录不存在。

叶子节点比较特别，它们的指针指向的是被索引的数据，而不是其他的节点页（不同引擎的“指针”类型不同）。

注意：索引对多个值进行排序的依据是CREATE TABLE语句中定义索引时列的顺序。

B-Tree索引适合用于全键值、键值范围或键前缀查找。其中键前缀查找只适合用于根据最左前缀的查找。

全值匹配

全值匹配指的是和索引中的所有列进行匹配，例如前面提到的索引可用于查找姓名为xx、出生于1991年的人。

匹配最左前缀

前面提到的索引可用于查找所有姓为Allen的人，即只使用索引的第一列。

匹配列前缀

也可以只匹配某一列的值的开头部分。例如前面提到的索引可用于查找所有以J开头的姓的人。这里也只使用了索引的第一列。

匹配范围值

例如前面提到的索引可用于查找姓在Allen和Barrymore之间的人。这里也只使用了索引的第一列。

精确匹配某一列并范围匹配另外一列

例如索引也可以用于查找所有姓为Allen,并且名字是字母K开头的人。即第一列是last\_name的全匹配，第二列是first\_name的范围匹配。

只访问索引的查询

B-Tree通常可以支持“只访问索引的查询”，即查询只需要访问索引，而无须访问数据行。

因为索引树中的节点是有序的，所以除了按值查找之外，索引还可以用于查询中的ORDER BY操作（按顺序查找）。一般来说，如果B-Tree可以按照某种方式查找到值，那么也可以按照这种方式用于排序。

B-Tree的一些限制：

1.如果不是按照索引的最左列开始查找，则无法使用索引。例如索引无法用于查找姓为Bill的人，也无法查找某个特定生日的人，因为这两列都是不最左数据列。

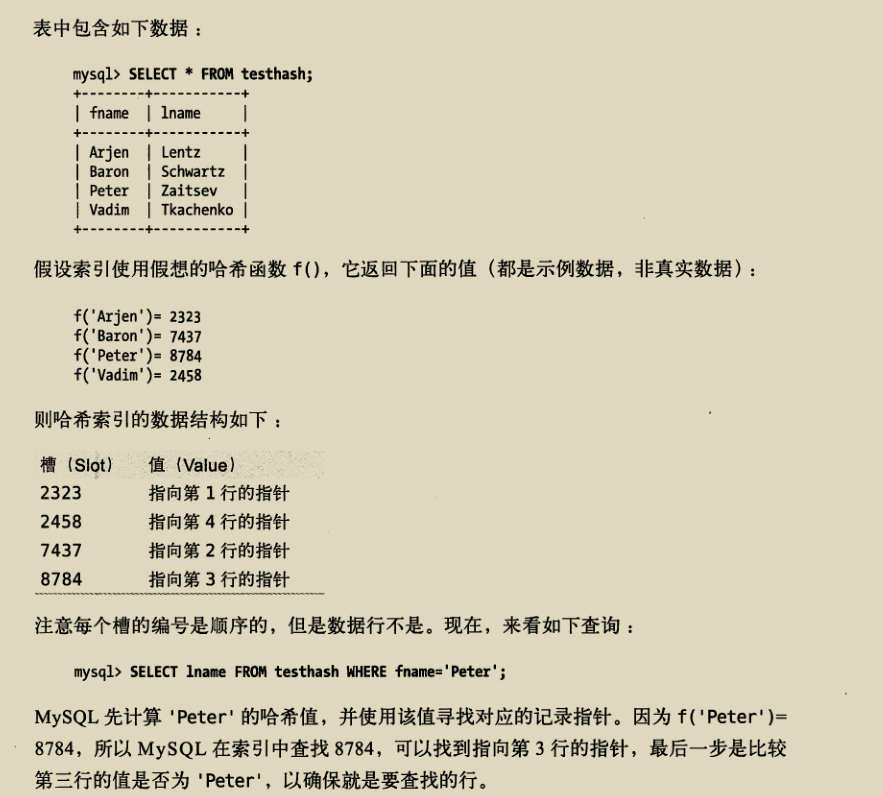
2.不能跳过索引中的列。即前面所述的索引无法用于查找姓为Smith并且在某个特定日期出生的人。如果不指定名（first\_name）,则MySQL只能使用索引的第一列。

3.如果查询中有某个列的范围查找，则右边所有列都无法使用索引优化查找。例如有查询：

Where last\_name = ‘Smith’AND first\_name Like ‘J%’AND dob = ‘1991-10-1’,这个查询只能使用索引的前两列，因为这里Like是一个范围条件。如果范围查询列值的数量有限，那么可以通过使用多个等于条件来替代范围条件。

哈希索引

哈希索引（hash index）基于哈希表实现，只有精确匹配索引所有列的查询才有效。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），哈希吗是一个较小的值，并且不同键值的行计算出来的哈希码也不一样。哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。如果多个列的哈希值相同，索引会以链表的方式存放多个记录指针到同一个哈希条目中。



哈希索引的限制：

1.哈希索引值包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行的速度很快，所以大部分情况下这一点对性能的影响并不明显。

2.哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序。

3.哈希索引也不支持部分索引列匹配查找，因为哈希索引始终是使用列的全部内容来计算哈希值的。

4.哈希索引只支持等值比较查询，包括=、IN()、<=>（注意<>和<=>是不同的操作）。也不支持任何范围查找，例如where price > 100

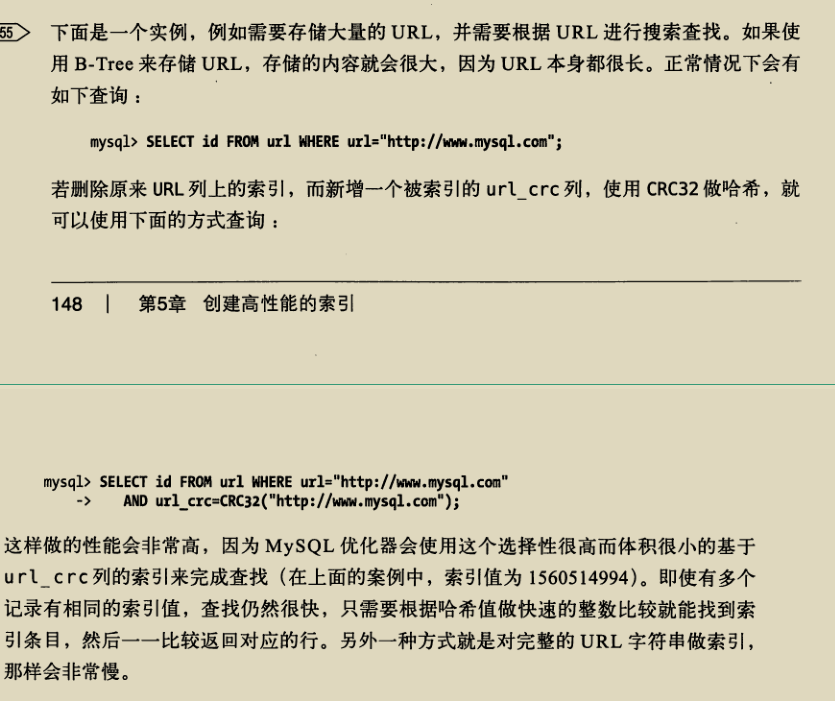
注：<>-不等于；<=>-安全的等于，不会返回 UNKNOWN

5.访问哈希索引的数据非常快，除非有很多哈希冲突（不同的索引列值却有着相同的哈希值）。存储引擎必须遍历链表中所有的行指针，逐行比较，直到找到所有符合条件的行。

6.如果哈希冲突很多的话，一些索引维护操作的代价也会很高。例如，如果在某个选择性很低（哈希冲突很多）的列上建立索引，那么当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行，找到并删除对应行的引用，冲突越多，代价越大。

InnoDB 引擎中有一个特殊的功能叫做“自适应哈希索引（adaptive hash index）”。当InnoDB注意到某些索引值被使用的非常频繁时，它会在内存中基于B-Tree索引之上再创建一个哈希索引，这样就让B-Tree索引也具有哈希索引的一些优点。

创建自定义的哈希索引



空间数据索引

全文索引

5.2 索引的优点

1.索引大大减少服务器需要扫描的数据量。

2.索引可以帮助服务器避免排序和零时表。

3.索引可以将随机I/O变为顺序I/O.

如何评价一个索引是否适合某个查询：

三星系统-three-star system

索引将相关的记录放在一起则获得一星；

如果索引中的数据顺序和查找中的排列顺序一致则获得二星；

如果索引中的列包含了查询中需要的全部列则获得三星。

5.3 高性能的索引策略

5.3.1 独立的列

如果查询中的列不是独立的，则MySQL就不会使用索引。“独立的列”是指索引列不能是表达式的一部分，也不是函数的参数。

5.3.2 前缀索引和索引选择性

通常索引开始的部分字符串，这样可以大大节约索引空间，从而提高索引效率。但是这样会降低索引的选择性。索引的选择性是指：不重复的索引值（也称基数）和数据表的记录总数（T）的比值，范围从1/T到1之间。索引的选择性越高则查询的效率越高，因为选择性高的索引可以让MySQL在查找时过滤更多的行。唯一索引的选择性是1。

一般情况下某个列前缀的选择性也是足够高的，足以满足查询性能。对于BLOB、text或者很长的VARCHAR类型的列，必须使用前缀索引，因为MySQL不允许索引这些列的完整长度。

关键在于选择足够长的前缀以保证较高的选择性，同时又不能太长。前缀应该足够长，以使得前缀索引的选择性接近于索引整个列。换句话说，前缀的“基数”应该接近于完整列的“基数”（完整列的基数：某一列不重复的行数与数据表的总行数的比值）。

前缀索引的缺点：MySQL无法使用前缀索引做ORDER BY和GROUP BY操作，也无法使用前缀索引做覆盖扫描。

5.3.3 多列索引

常见的错误就是：为每个列创建独立的索引，或者按照错误的顺序创建多列索引。

在多个列上建立独立的单列索引大部分情况下并不能提高MySQL的查询性能。Mysq5.0和更新版本引入一种叫“索引合并”（index merge）的策略，一定程度上可以使用表上的多个单列索引来定位指定的行。

索引合并策略有时候是一种优化的结果，但实际上更多时候说明表上的索引建得很糟糕：

1.当出现服务器对多个索引做相交操作时（通常有多个AND条件），通常意味着需要一个包含所有相关列的多列索引，而不是多个独立的单列索引。

2.当服务器对多个索引锁联合操作时（通常有多个OR条件），通常需要耗费大量CPU和内存资源在算法的缓存、排序和合并操作上。特别是当其中有些索引的选择性不高，需要合并返回的大量数据的时候。

3．更重要的是，优化器不会把这些计算到“查询成本”中，优化器只关系随机页面读取。这会使得查询的成本被“低估”。导致该执行计划还不如直接全表扫描。

5.3.4 选择合适的索引顺序（看案例明白索引的选择性）

正确的顺序依赖于使用该索引的查询，并且同时需要考虑如何更好地满足排序和分组的需要。在一个多列B-Tree索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列，等等。故索引可以按照升序或者降序进行扫描，以满足精确符合列顺序的ORDER BY、GROUP BY和DISTINCT等子句的查询需求。

如何选择索引的列顺序有一个经验法则：将选择性最高的列放在索引最前面。当不需要考虑排序和分组时，将选择性最高的列放在最前面通常是很好的。这时候的索引作用只是用于优化where条件的查找。

5.3.5 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据结构。但InnoDB的聚簇索引实际上在同一结构中保存了B-Tree索引和数据行。

当表中有聚簇索引时，它的数据行实际上存放在索引的叶子页（leaf page）中。聚簇表示数据行和相邻的键值紧凑地存储在一起。因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。？？？

InnoDB通过主键聚集数据，如果没有定义主键，InnoDB会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。InnoDB只聚集在同一页面中的记录。包含相邻键值的页面可能会相距甚远。？？？

聚簇索引的优点：

1.可以把相关数据保存在一起

2.数据访问更快。聚簇索引将索引和数据保存在同一个B-Tree中，因此从聚簇索引中获取数据通常比在非聚簇索引中查找更快。

3.使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

聚簇索引的缺点：

1．聚簇索引最大限度地提高了I/O密集型应用的性能，但是如果数据全部都放在内存中，则访问数据的顺序就没有那么重要了，聚簇索引就没什么优势了。

2.插入速度严重依赖于插入顺序。按照主键的顺序插入是加载数据到InnoDB表中速度最快的方式。但是如果不是按照主键顺序加载数据到InnoDB表中，那么在加载完成以后最好使用OPTIMIZE TABLE命令重新组织一下表。

3.更新聚簇索引列的代价很高，因为会强制InnoDB将每个被更新的行移到新的位置。

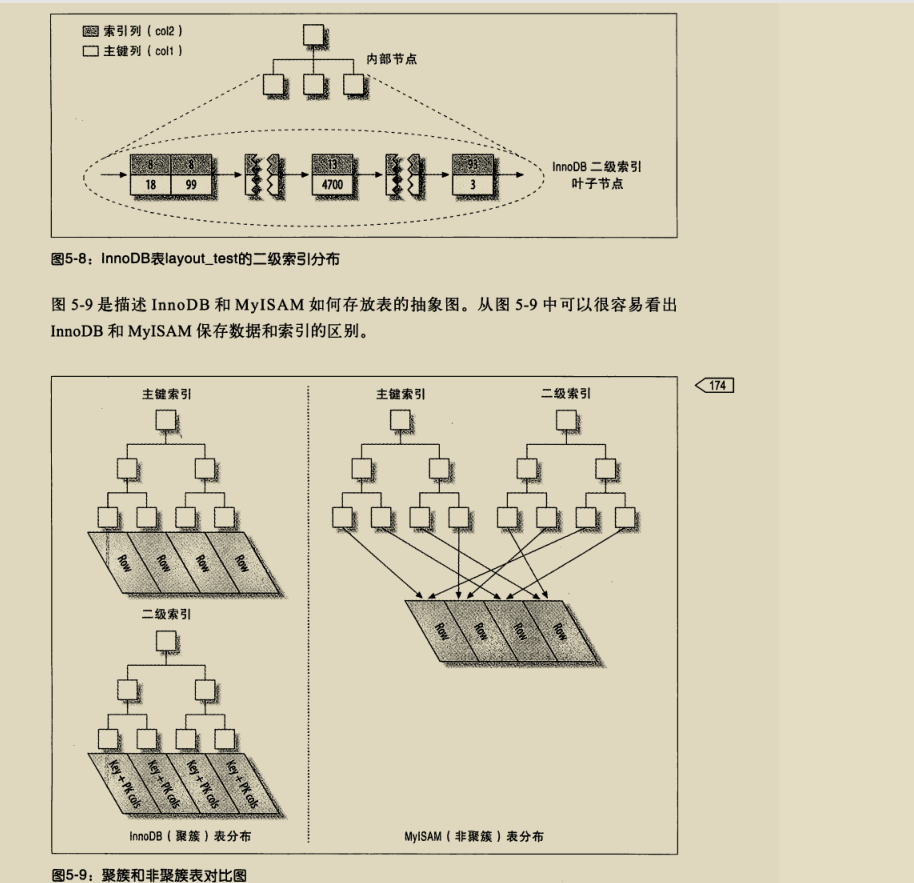
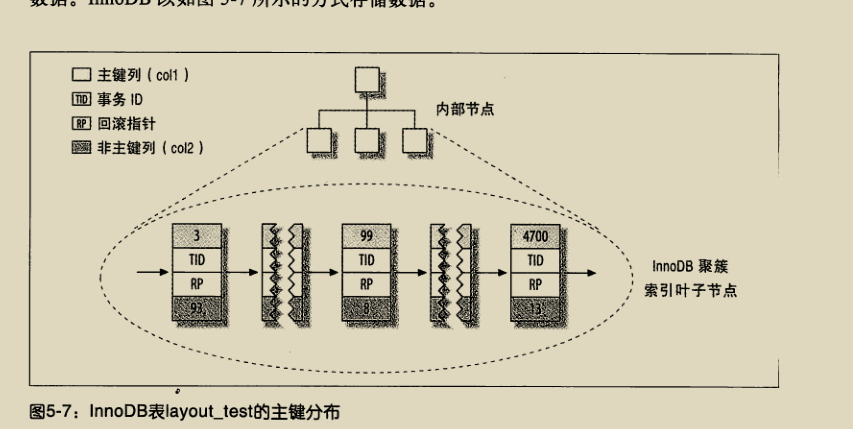
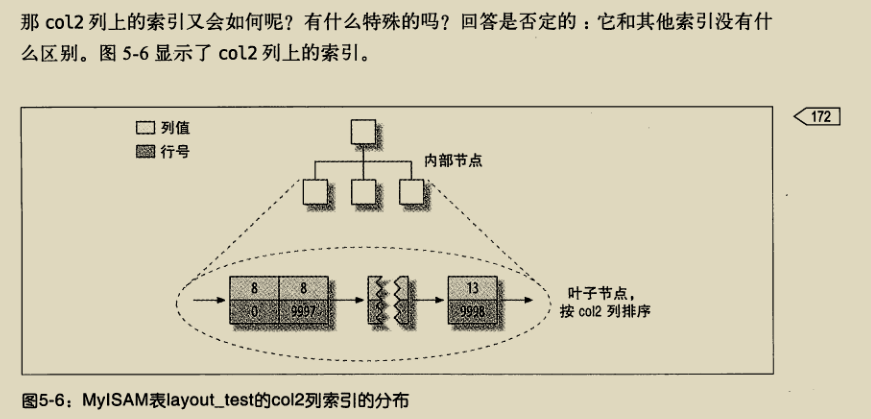
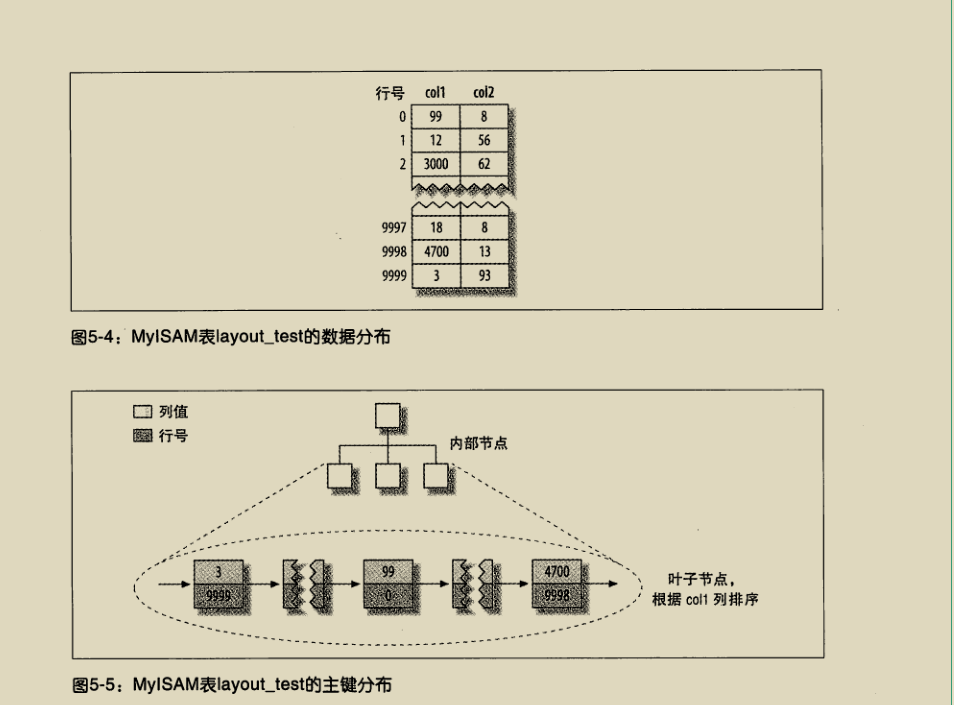
4.基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临“页分裂（page split）”的问题。当行的主键要求必须将这一行插入到某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳该行，这就是一次页分裂操作。页分裂操作会导致表占用更多的磁盘空间。？？？

5.聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候。

6.二级索引（非聚簇索引）可能比想象中的要更大，因为在二级索引中的叶子节点包含了引用行的主键列。？

7.二级索引访问需要两次索引查找，而不是一次。为什么二级索引需要两次索引查找，原因是：二级索引中保存的“行指针”的实质。要记住，二级索引叶子节点保存的不是指向行的物理位置的指针，而是行的主键值。

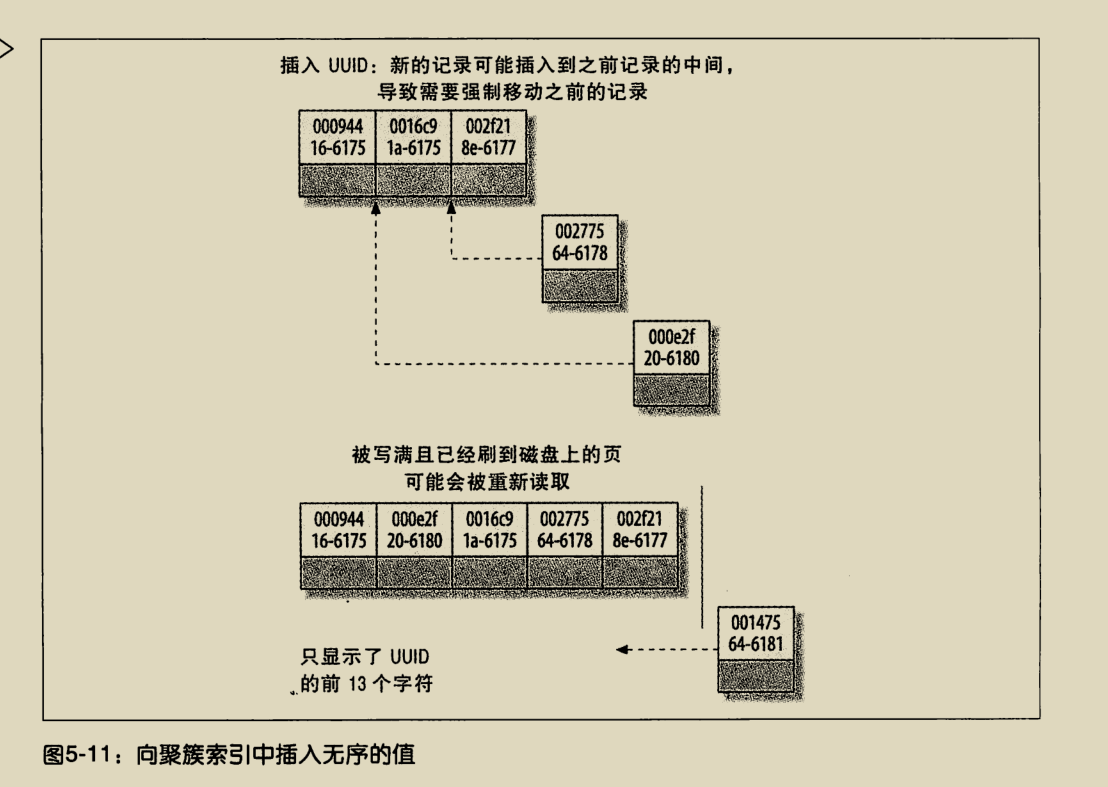
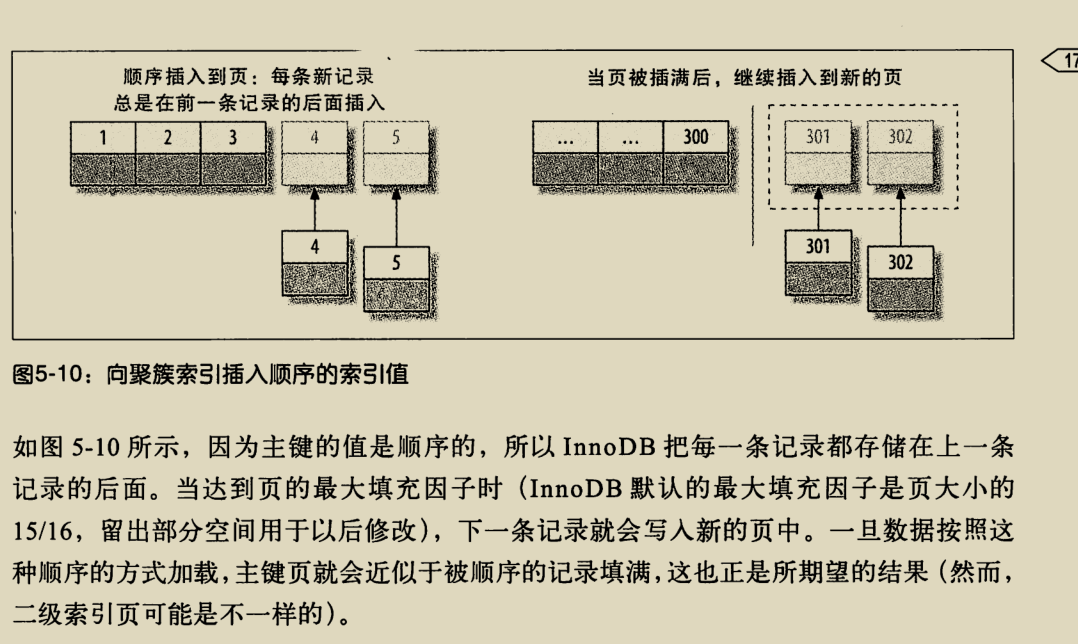
InnoDB和MyISAM的数据数据分布对比：



聚簇索引的每一个叶子节点都包含了主键值、事务ID和MVCC的回滚指针以及所有的剩余列。如果主键是一个列前缀索引，InnnoDB也会包含完整的主键列和剩余的其他列。

InnoDB的二级索引和聚簇索引很不相同。InnoDB二级索引的叶子节点中存储的不是“行指针”，而是主键值，并以此作为指向行的“指针”。这样的策略减少了当出现行移动或者数据分页时二级索引的维护工作。使用主键值当作指针会让二级索引占用更多的空间，换来的好处是，InnoDB在移动行时无须更新二级索引中的这个“指针”。

在InnoDB表中按主键顺序插入行



无序插入，InnoDB无法简单地总是把新行插入到索引的最后，而是需要为新的行寻找合适的位置-通常是已有数据的中间位置-并且分配空间。无序插入的缺点如下：

1.写入的目标页可能已经刷到磁盘上并从缓存中移除，或者是还没有被加载到缓存中，InnoDB在插入之前不得不先找到并从磁盘读取目标页到内存中。这将导致大量的随机I/O。

2.因为写入时乱序的，InnoDB不得不频繁地做页分裂操作，以便为新的行分配空间。页分裂会导致移动大量数据，一次插入最少需要修改三个页面而不是一个页。

3.由于频繁的页分裂，页会变得稀疏并被不规则地填充，所以最终数据会有碎片。

顺序主键何时会有坏结果：

对于高并发工作负载，在InnoDB中按主键顺序插入可能会造成明显的争用。

5.3.6 覆盖索引

通常我们会根据查询的where条件来创建合适的索引，不过这只是索引优化的一个方面。设计优秀的索引应该考虑到整个查询，而不单单是where条件部分。如果一个索引包含（或者覆盖）所有需要查询的字段的值，则称之为“覆盖索引”。

覆盖索引的好处：

1.索引的条目通常远小于数据行的大小，所以如果只需读取索引，那么MySQL就会极大地减少数据访问量。这对缓存的负载非常重要，因为这种情况下响应时间大部分花费在数据拷贝上。覆盖索引对于I/O密集型的应用也有帮忙，因为索引比数据更小，更容易全部放在内存中。

2.因为索引是按照列值顺序存储的（至少在单个页内是如此），所以对于密集型的范围查询会比随机从磁盘读取每一行数据的I/O要少得多。

3.一些存储引擎如MyISAM在内存中只缓存索引，数据则依赖于操作系统来缓存，因此要访问数据需要一次系统调用。这可能会导致严重的性能问题，尤其是那些系统调用占了数据访问中的最大开销的场景。

4.由于InnoDB的聚簇索引，覆盖索引对InnoDB表特别有用。InnoDB的二级索引在叶子节点中保存了行的主键值，索引如果二级主键能够覆盖查询，则可以避免对主键索引的二次查询。？？？

在发起一个被索引覆盖的查询时，在EXPLAIN的Extra列可以看到“Using index”。

5.3.7 使用索引扫描来做排序

MySQL有两种方式可以生成有序的结果：通过排序操作；或者按索引顺序扫描。

如果EXPLAIN出来的type列的值为“index”，则说明MySQL使用了索引扫描来做排序。

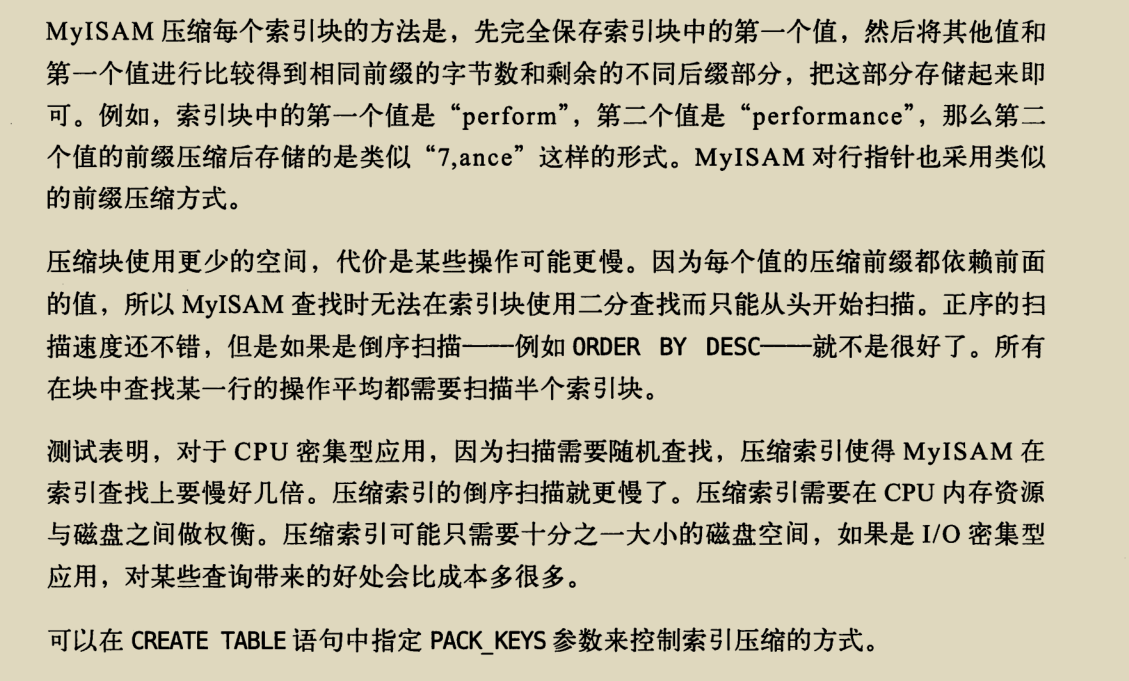
满足使用索引排序的几种情况：

1.只有当索引的列顺序和ORDER BY子句的顺序完全一致，并且所有列的排序方向（正序或者倒序）都一样。

2.如果查询需要关联多张表，则只有当ORDER BY子句引用的字段全部为第一个表时，才能使用索引做排序。ORDER BY子句和查找型查询的限制是一样：满足索引的最左前缀的要求；否则，MySQL都需要执行排序操作，而无法利用索引排序。

3.有一种情况下ORDER BY子句可以不满足索引的最左前缀的要求，就是前导列为常量的时候。如果where子句或者join子句中对这些列指定了常量，就可以“弥补”索引的不足。

5.3.8 压缩（前缀索引）索引

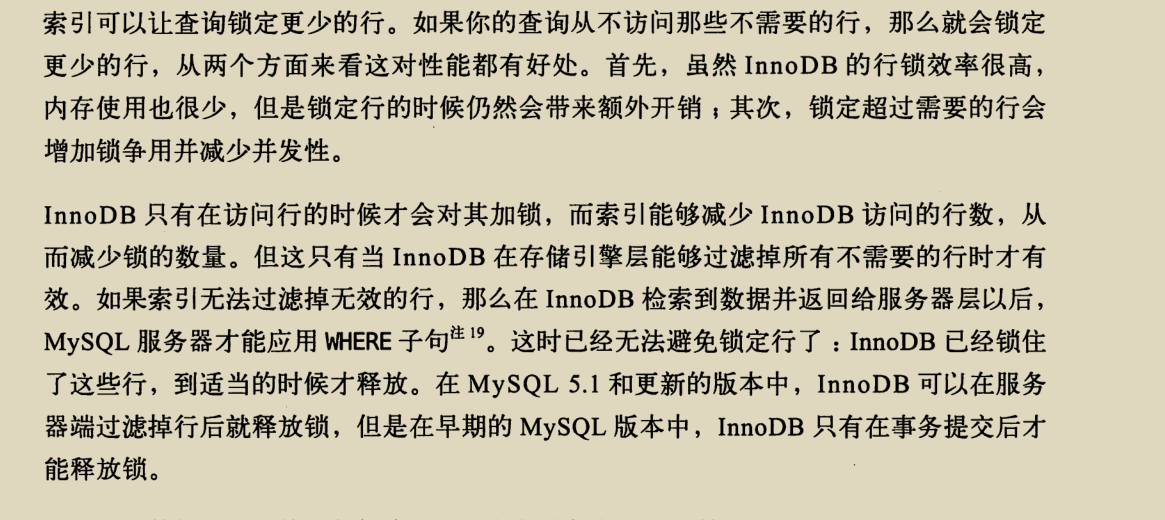


5.3.9 冗余和重复索引

5.3.10 未使用的索引

5.3.11 索引和锁

使用EXPLAIN时，Extra列出现了“Using where”,这表示MySQL服务器将存储引擎返回行以后再应用where过滤条件。



5.4 索引案例学习

第一件需要考虑的事情就是需要使用索引来排序，还是先检索数据在排序。使用索引排序会严格限制索引和查询的设计。

5.4.1 支持多种过滤条件

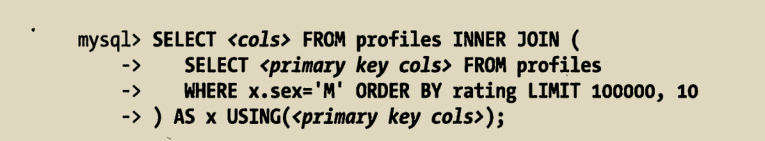
Where条件中age BETWEEN 18 AND 25，无法使用索引，转换的技巧：使用IN列出所有的取值。但如果列有太多的不同值，这样就不行了。

5.4.2 避免多个范围条件

对于范围条件查询，MySQL无法使用范围列后的其他索引了，但是对于“多个等值条件查询”则没有这个限制。

5.4.3 优化排序

优化索引排序的较好策略是使用延迟关联，通过使用覆盖索引查询返回需要的主键，在根据这些主键关联原表获得需要的行。这样可以减少MySQL扫描那些需要丢弃的行数。



MyISAM存储引擎的特点：

1.不支持事务操作（OLTP）

2.缓冲池值缓存索引文件，而不缓冲数据文件