1. **概述**

Innodb中的常见索引如下：

* B+树索引
* 全文索引
* 哈希索引

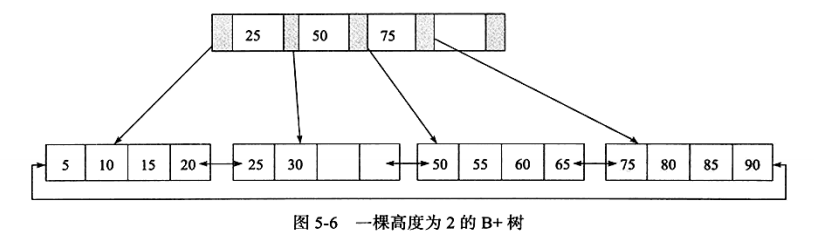
Innodb支持的hash所有是自适应的，会根据实际使用情况自动生成，不用人为干预。

B+树索引是传统意义上的所有，这是目前关系型数据库中查找最为常用和最有效的所有。其构造类似于平衡多叉搜索树，根据键值（key value）快速找到数据（logN）。B+树中的B是balance。

另一个需要注意的是，B+树不能直接定位到给定键值的数据行，只能定位到数据行所在的页（通常为16kb一页），然后数据库把页读入内存，再在内存中查找想要的数据。

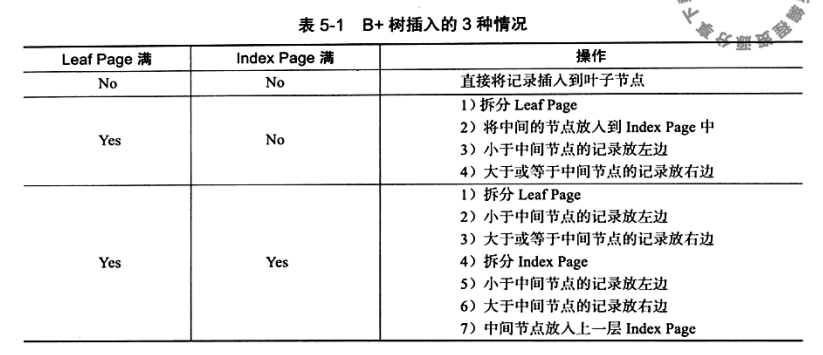
1. **B+树**

简单来说，B+树是为磁盘或其他直接存储设备设计的一种平衡查找树。在B+树中，所有的记录节点都是按照key的大小顺序存放在同一层的叶子节点上，各个叶子节点由双向指针连接。非叶子节点用于标识key的范围。下图为高度为2，每页存放4条记录的B+树：

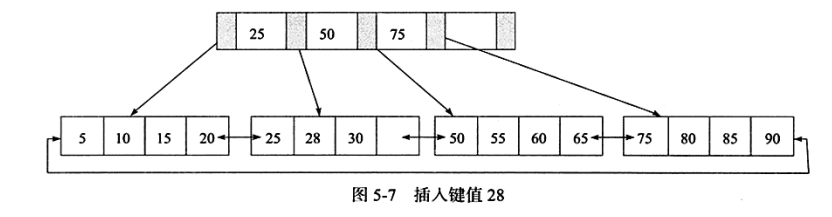


1. **插入**

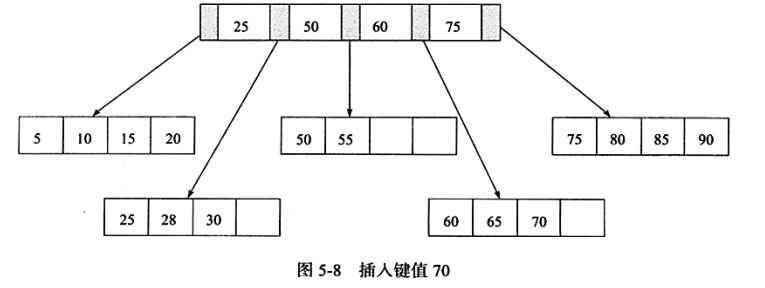
B+树的插入删除和《11-234树》非常类似。会有节点分裂的过程：



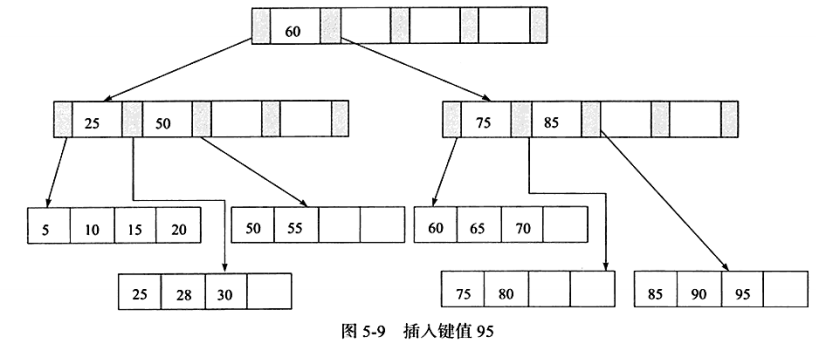
Case 1: 对于5-6的B+树，先插入28，则可以直接插入：



Case2: 再插入70，原先Leaf page满了，index page未满，符合表5-1中的第二种情况，此时会将中间的60查分叶子节点，得到如下图：



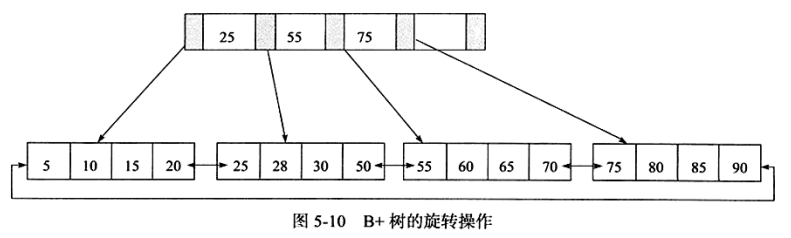
Case3: 最后插入95，此时Leaf page和index Page都满了，符合5-1表中的第三种情况：



可以看出，B+树是平衡树，但是查分页的操作意味着磁盘操作，所以应该尽可能少的减少也的查分。所以B+树统一提供了红黑树的旋转功能。

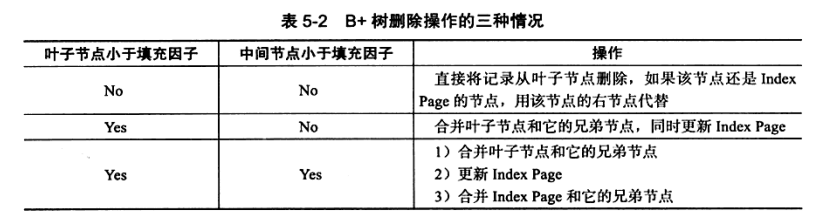
旋转发生在Leaf page已满，但其左右兄弟节点未满，此时B+树不会急于拆分页，而是将记录转移到兄弟节点上。通常情况下，左兄弟节点先用来做旋转。

Case4: 再来看5-7，若插入70，B+树不拆分leaf page，而是旋转，得到如下：

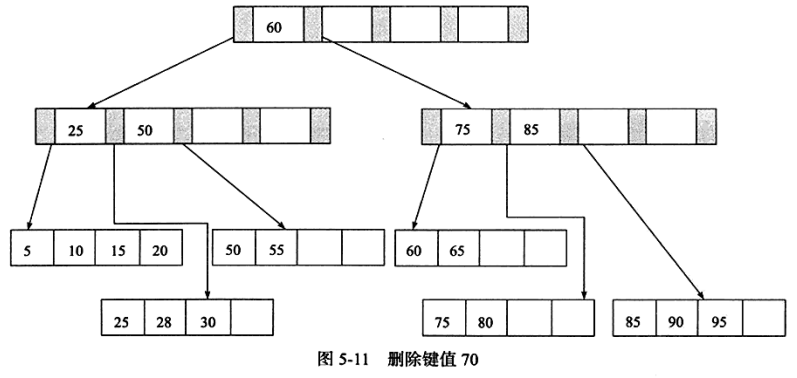


1. **删除**

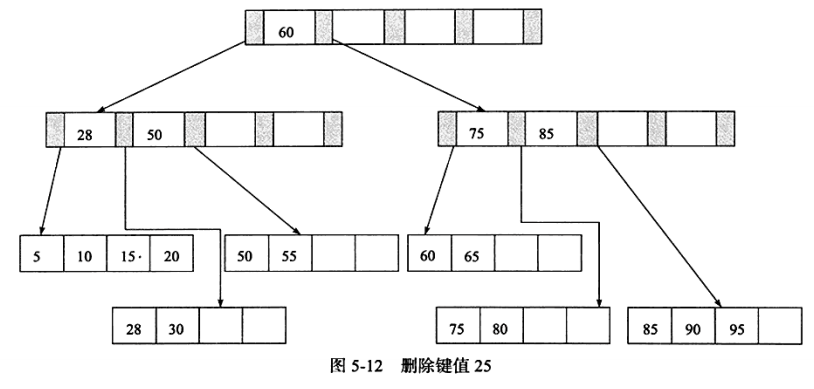
B+树使用填充引子(fill factor)来控制树的删除变化，一般来说，填充引子最小50%。



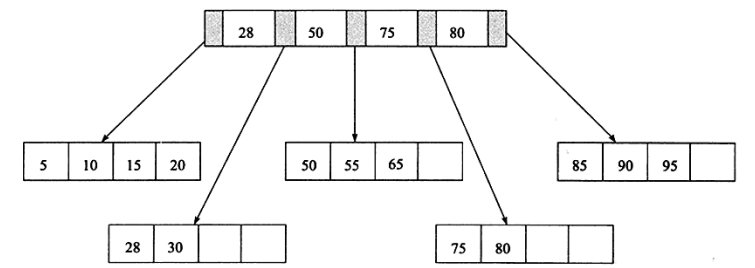
Case1: 根据5-9来删除70这个记录，符合表5-2的第一种情况：



Case2: 删除25，此时已然符合符合表5-2的第一种情况，但是需要删除index page中的记录，并将其右兄弟节点28更新到page index中：



Case3: 删除60。由于对应的Leaf page删除60后，fill factor小于50%。则需要进行页合并操作：



1. **B+树索引**

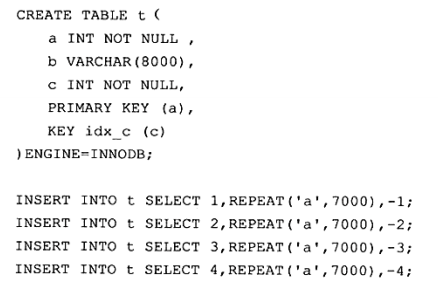
B+树索引的本质就是B+树在数据库中的实现。一般在DB中B+树的高度在2-4层左右。也就意味着只需要2-4次的IO操作即可。而现在的磁盘每秒差不多在100次IO左右，2-4次意味着查询时间只需0.02-0.04秒。

数据库中B+树可以分为聚集索引(clustered index)和辅助索引(secondary index)，这两种都是B+树，即高度平衡，叶子节点存放数据。聚集索引与辅助索引不同的是，叶子节点存在一整行数据。

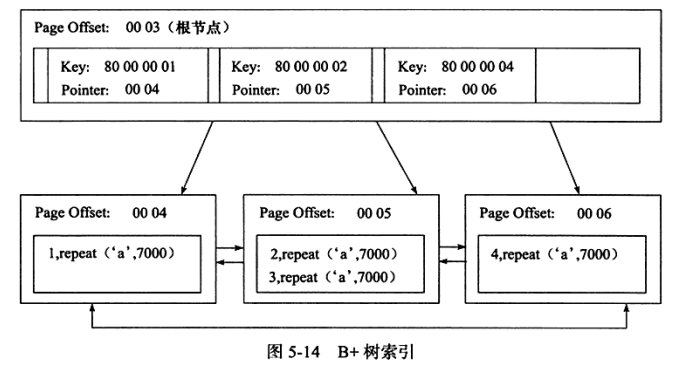
1. **聚集索引**

Innodb存储引擎表示索引组织表，即表中数据按照逐渐顺序存放。而聚集索引就是按照每张表的主键构造一棵B+树，叶节点存放整张表的行记录数据页。聚集索引这个特性决定了索引组织表中数据也是索引的一部分。

下面我们创建一张表，其中b列长度为7000个字节，而一个数据页为16kb，所以一页只能存储2行数据。

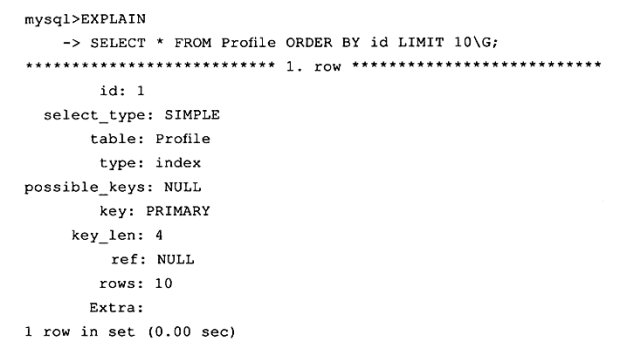


此时构成的聚集索引如下。我们发现，数据页上存放的是完整的行数据，而在索引页中，存放的是索引key和指向数据页的偏移量（指针）。



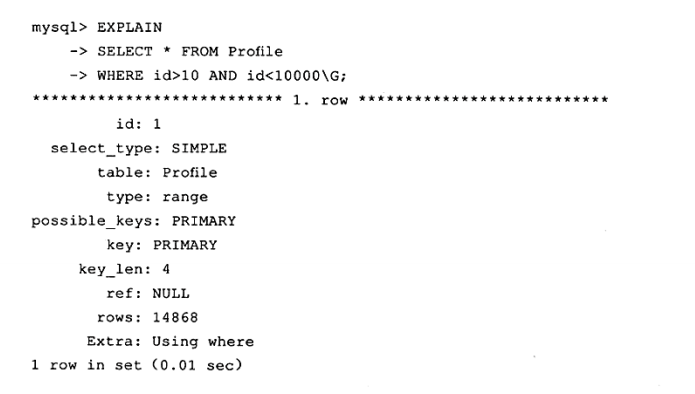
聚集索引不一定按照物理地址存储数据，因为按照特点顺序存放物理记录维护成本很高。所以聚集索引在物理上不是连续的，但是在逻辑上是连续的。

聚集索引的另一个好处是，对于主键的排序查找和范围查找非常快。叶子节点的数据就是全部数据，如果用户需要查询一张注册表，查询最后注册的10位用户，用于B+树是双向链表，则可以快速找到最后一页数据，根据链表从最后向前取出十条即可。使用explain如下：



虽然sql中使用了order by，但并没有进行file sort操作，这就是聚集索引的特点。

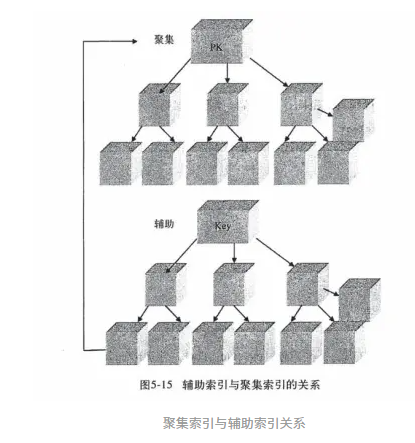
另一个就是范围查询，如果根据主键查找某分为内的数据，通过叶子节点的上层中间节点就可以找到页的范围，之后直接读取数据页即可。执行explain如下：



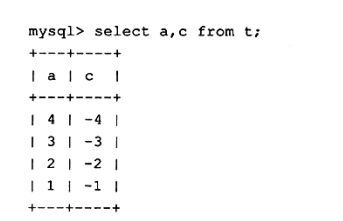
执行计划中的rows都是估计数量。

1. **辅助索引**

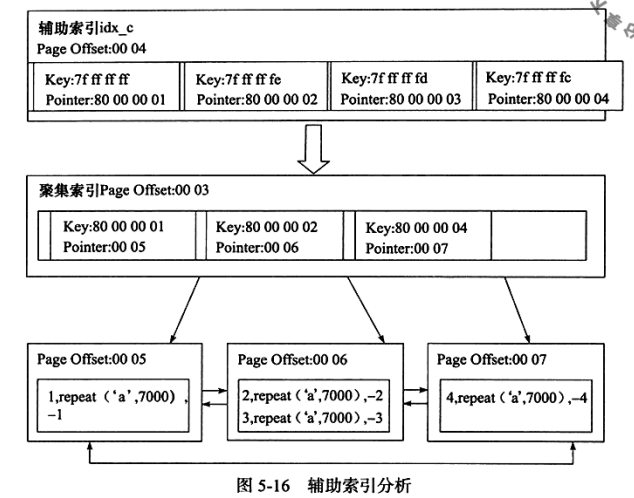
对于辅助索引，叶子节点除了包含索引key以外，还包含了聚集索引key，即我们还需要拿着聚集索引key（主键）再去聚集索引中回表才能获得全部数据。



在5-14图中的表再新建一个c列，并其设为辅助索引，其中c的值如下：



此时我们根据辅助索引c查出全部数据的索引结构如下（7f ff ff ff代表-1）：



辅助索引可以支持索引覆盖，即从辅助索引中既可以查询数据记录。在特定sql（如字段为索引字段，count）时，索引覆盖可以提升查询效率，不需要从聚集索引中回表二次查询。

特别是count查询时，不会走聚集索引，辅助索引比聚集索引的大小小的多，选择辅助索引可以减少io操作。

此时userId是table辅助索引，id是聚集索引select count(1) from table比如走的是辅助索引。

1. **索引管理**

我们通过show index from tableName可以查看tableName表下的索引信息，下面说几个比较关键的：

1. Seq\_in\_index

索引中该列的位置，联合索引最为直观

1. Collation

列以什么方式存储在索引中，可以是A或NULL。B+树都是A，即排序。而heap存储引擎，并建立了hash索引，则会显示NULL。

1. Cardinality

非常关键的值，表示索引中唯一值数目的抽样估计。Cardinality/N（N为该表总行数）越接近于1，则说明该索引的效率越高，如果非常小，sql在执行时可能就不会走改索引，所以此时需要考虑删除该索引。

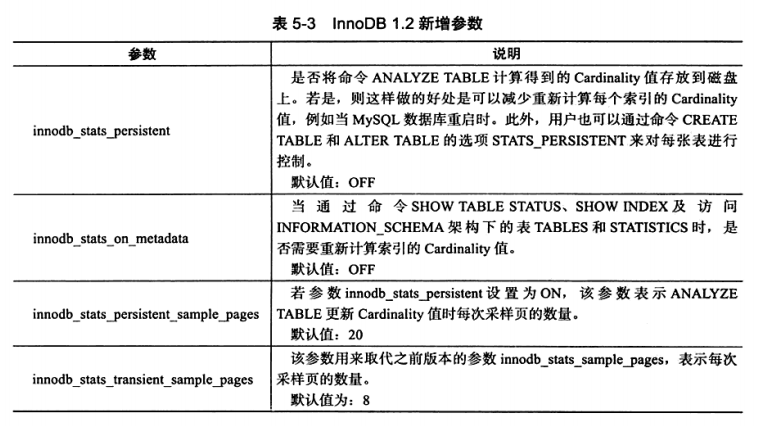
例如我们给性别一列建立索引，那么大部分情况下，该索引的Cardinality=2.此时完全没有必要设为索引。

Cardinality的更新只发生在以下2中情况的某一种：

* 表中1/16的数据发生过变化
* stat\_modified\_counter(总修改次数) > 2 000 000 000

对于Cardinality的更新是通过采样来完成，默认会对8个叶子节点Leaf page进行采样，设总叶子节点个数为A，8个叶子节点中索引key的不同记录的个数为P1,P2…P8，那么Cardinality=(P1 + P2 + … + P8) / 8 \* A。

我们在执语句analyze table、show table status、show index、optimize 时可能会重新计算Cardinality。



optimize table时一定重新计算Cardinality

1. **Fast index creation**

Mysql5.5之前创建索引或删除这样的DDL操作，数据库操作过程为：

* 创建临时表，表结构通过命令alter table新定义的结构；
* 把原表数据导入临时表；
* 删除原表；
* 将临时表名改为原表名。

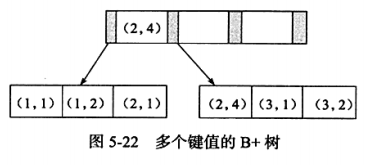
可以发现，对一张大表进行索引新增或删除，需要很长时间，且存在大量事务阻塞，服务不可用。

Innodb在新版本支持Fast index creation，此时会对索引创建的表上加一个share锁，允许读操作，不允许写操作。删除操作就可以简单，只需要更新内部视图，并将辅助索引的空间标记为可用，同时删除mysql数据库内部视图上对该表的索引定义即可。

1. **联合索引**

联合索引是只对表上多个列进行索引。联合索引与普通B+树索引一样，只不过其中的key是多个索引的排序组合，按照从左到右的顺序。

例如(a,b)构成的联合索引如下图：



在叶子节点，其顺序为先对a排序，在a排序的基础上再对b排序。显然我们不能只根据b作为条件来查询。因为b列本身并不是有序的。使用联合索引在查询时对第二个key进行了排序，即对于：

Select a,b from table where a=’1’ order by b desc

此时sql的执行计划中不会有filesort的排序。

1. **索引查询优化**
2. **优化器不使用索引**

在执行范围查询和join时，我们explain发现并没有通过辅助索引去查找数据，而是通过聚集索引全表扫描。因为辅助索引需要会再次查询聚集索引，此时在磁盘上就是大量随机查询，当数据量大时，不如直接通过顺序查询聚集索引扫描全表。

当然我们可以通过force index来强制执行索引查询。

例如EXPLAIN SELECT \* FROM `trade` t where t.create\_time BETWEEN '2019-07-17 19:43:50' and '2020-07-01 19:43:50'

C:\Users\ofcard\AppData\Local\Temp\1592467180(1).png

我们缩小范围有，此时就走索引了。

EXPLAIN SELECT \* FROM `trade` t where t.create\_time BETWEEN '2019-07-17 19:43:50' and '2019-07-17 19:45:50'

C:\Users\ofcard\AppData\Local\Temp\1592467161(1).png

使用force index如下：

EXPLAIN SELECT \* FROM `trade` t force index(idx\_c\_t\_c\_t) where t.create\_time BETWEEN '2019-07-17 19:43:50' and '2020-07-17 19:45:50'

C:\Users\ofcard\AppData\Local\Temp\1592467133(1).png

1. **Multi-Range Read优化（Extra出现MRR）**

对于范围查询和JOIN，mysql5.6开始支持MRR优化。工作方式如下：

* 将查询得到辅助索引的key放入一个缓存中，此时缓存中的数据是根据辅助索引的key排序的；
* 将缓存中的key根据RowId排序（聚集索引顺序）；
* 根据RowId的顺序来实际访问数据文件。

MRR可以减少随机读，增加顺序读操作，可以将随机读减少到最大。例如：

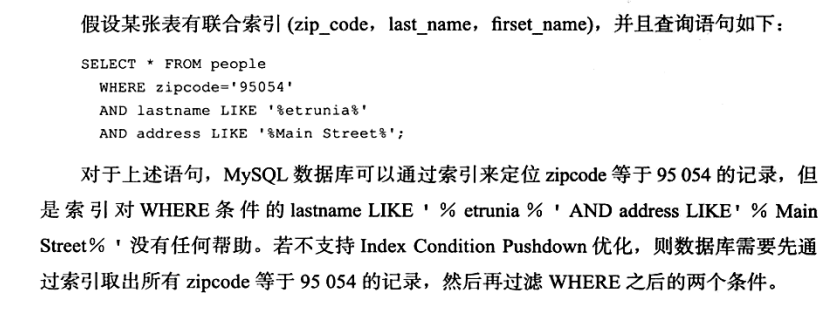
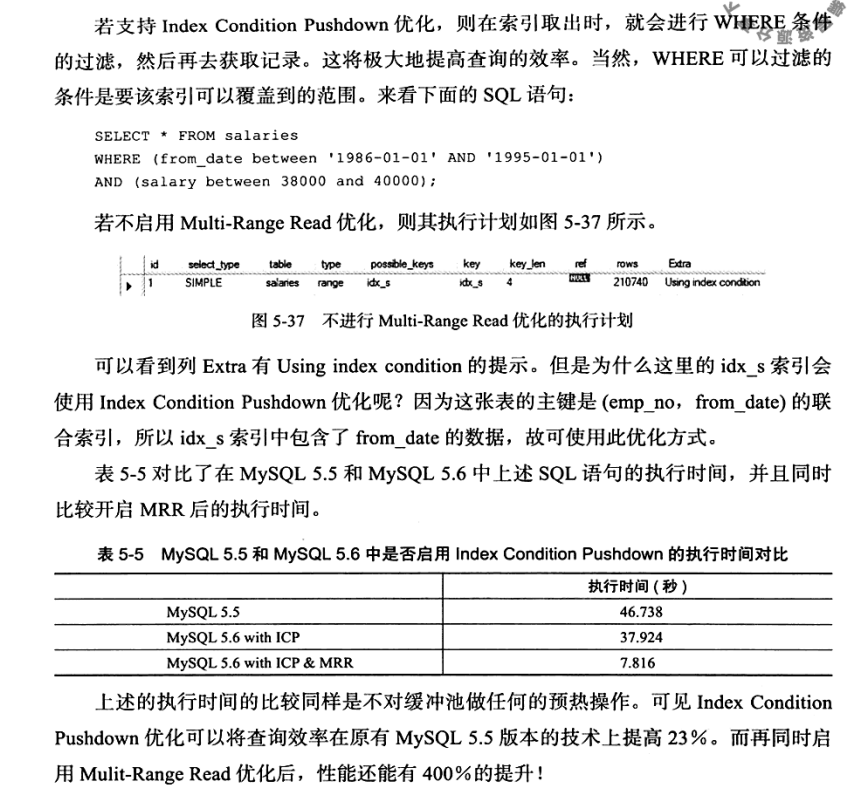


可以发现，MRR可以提升查询效率10倍以上，但是我在自己库中（开了MRR）却没有发现extra中出现MRR提示。

1. **Index Condition Pushdown优化（Extra出现index condition）**

Mysql5.6提供了ICP优化，在没有ICP时，sql先根据索引查询出全部数据，然后再根据where过滤。在支持ICP后，mysql库在取出索引的同时，判断是否可以进行where 条件过滤，也就是将where的部分过滤操作放在了innodb存储引擎层，而不是sql层，这将大大较少上次sql层对记录（全部数据）的索取，从而提升整体性能。要使用ICP，此时where条件必须是辅助索引（联合索引）中的一个key。

ICP优化支持范围查询range、ref、eq\_ref等类型。

1. **索引优化实例**

原sql为：

*Select t.tid, t.seller from trade t where t.createTime between ‘2020-07-20’ and ‘2020-07-21’ order by t.seller limit 0,100*

tid为主键，且createTime是辅助索引，此时由于查询条件中含有seller，显然会回表然后根据seller进行排序查询，查询时间接近900ms

由于tid也有时间且基本有序，为了减少回表，直接根据tid过滤：

*Select t.tid, t.seller from trade t where t.tid between ‘T20200720’ and ‘T20200721’ order by t.seller limit 0,100*

优化后时间为190ms。

此外，如果执行如下分页查询：select \* from table limit offset,size，

mysql会以物理存储顺序显示，当进行增删改之后，存储顺序改变，可能会造成数据重复或丢失，解决办法就是加order by。

1. **Hash索引**
2. **hash算法**

参见《14-哈希表》。Innodb中的hash算法在处理碰撞时使用链表扩展法。

1. **innodb hash索引**

Innodb中使用自适应hash算法来进行查询，对于缓冲池页的哈希表来说，其表中的每个槽都记录了key值所在的缓存池中的page页。也就是说，hash索引的最小粒度也是page(16kb)，与B+树索引一致。缓冲池中Page页都有一个chain指针，指向hashcode相同的页构成链表。

哈希表的大小是大于2倍缓存池页数量的质数，如innodb\_buffer\_pool\_size为10Mb，则共有640个16Kb的页。那么我们需要比1280大的质数，即1399作为哈希表的大小（mod）。为什么是2倍2倍缓存池页数量的质数内，这是希望是哈希表的槽数大写，能够尽量避免hash碰撞。

1. **自适应hash索引**

一般**只有=查询**才会创建自适应hash索引。

Select \* from table where a = ‘xx’

我们使用show engine innodb status可以看到，描述了自适应hash索引的使用情况。

*-------------------------------------*

*INSERT BUFFER AND ADAPTIVE HASH INDEX*

*-------------------------------------*

*Ibuf: size 1, free list len 59883, seg size 59885, 3391431 merges*

*merged operations:*

*insert 4024996, delete mark 133549692, delete 3852389*

*discarded operations:*

*insert 0, delete mark 0, delete 0*

*1541.20 hash searches/s, 1541.72 non-hash searches/s*

1. **全文检索**

我们知道，对于select \* from table where content like ‘xxx%’，只要content添加为索引，就可以通过索引来查询。

而select \* from table where content like ‘%xxx%’显然无法通过索引查询。全文检索full text search通过构建倒排索引，则可以帮助我们来完成查询和统计。基于mysql的全文检索比较罕见，es更加实用。

1. **innodb表最大数据量**

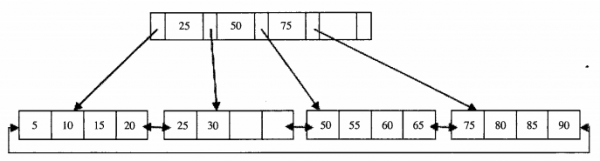
理论上来说，一张表可以存储操作系统运行范围下的单个文件的最大数据（例如单文件大小限制在4GB）。但是innodb索引组织表的B+树索引有一个特点是高扇出性，因此在数据库中，B+树的高度一般在2到3层。也就是说查找某一键值的记录，最多只需要2到3次IO开销。按磁盘每秒100次IO来计算，查询时间只需0.02到0.03秒。

**因此为了性能，通常需要限制B+的高度在3-4层！**

那么此时一张表最多可以存储多少行数据呢？

InnoDB存储引擎默认一个数据页大小为16kb，非叶子节点存放（key，pointer），pointer为6个字节，key如果是8个字节，即非叶子节点能存放16kb/14左右的key，pointer，而叶子节点如果一条数据大小为100字节，那一个叶子节点大约可存放160条数据。

如果高度为3，则可存放数据为：16kb/14 \* 16kb/14 \* 160大约1亿多数据。



参考

1. 《MySQL技术内幕：InnoDB存储引擎》第五章 索引和算法
2. 表的最大数据数量 <https://cloud.tencent.com/developer/article/1494725>
3. innodb b+树的高度问题。https://www.cnblogs.com/tangchuanyang/p/6634581.html