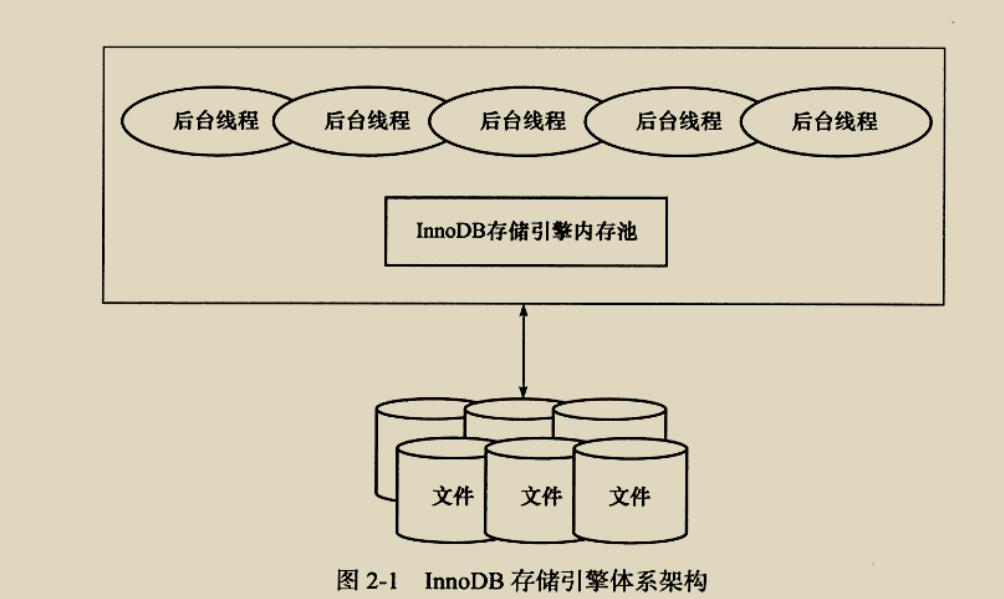
# InnoDB体系架构



后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存是最近的数据。此外将已经修改的数据文件刷新到磁盘文件，同时保证在数据库发生异常的情况下InnoDB能够恢复发到正常运行状态。

后台线程主要有以下几个：

1.Master Thread

2.IO Theread

3.Purge Thread

事务被提交以后，其所使用的undolog可能不再需要，因此需要Purge Thread来回收已经使用并分配的undo页。

4.Page Cleaner Thread

将之前版本中脏页的刷新操作都放入到单独的线程中来完成。

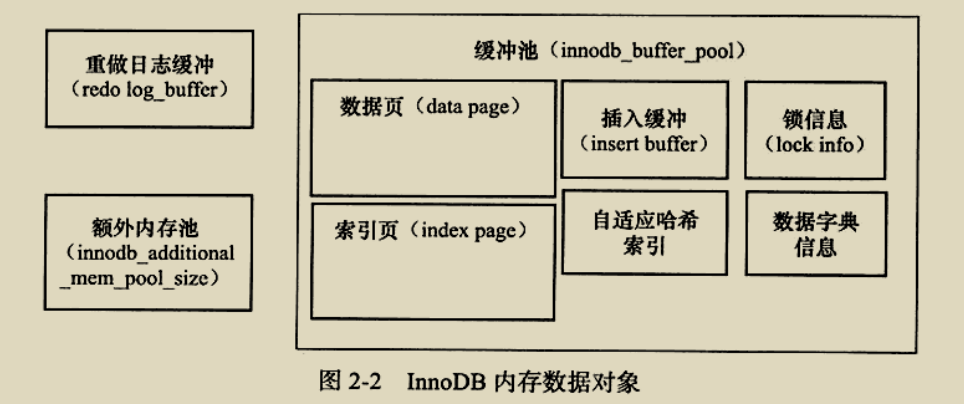
内存：

1.缓冲池

由于CPU速度和磁盘速度之间的鸿沟，基于磁盘的数据库系统通常使用缓冲池技术来提高数据库的整体性能。

缓冲池是一块内存区域，通过内存的速度来弥补磁盘速度较慢对数据库性能的影响。页从缓冲池刷新回磁盘的操作并不是在每次发生更新时触发，而是通过一种称为Checkpoint的机制刷新回磁盘。

缓冲池中缓存的数据页类型有：索引页、数据页、undo页、插入缓冲（insert buffer）、自适应哈希索引（adaptive hash index）、InnoDB存储的锁信息（lock info）、数据字典信息。不能简单的认为，缓冲池只是缓存索引页和数据页。



2. LRU List、Free List和Flush List

（1）LRU（Latest Recent Used）-最少使用算法

最频繁使用的页在LRU列表的前端，而最少使用的页放在LRU列表的尾端。当缓冲池不能存放新读取的到的页时，将首先释放LRU列表中尾端的页。

InnoDB对朴素的LRU算法进行了优化，在LRU算法中加入了midpoint位置。新读取到的页，虽然是最新访问的页，但并不是直接插入到LRU列表的首部，而是放在LRU列表的midpoint位置。因为某些SQL可能会是缓冲池中的页被刷新出，这类操作需要访问表中的许多页，甚至是全部的页，而这些通常来说又仅在这次查询操作中需要，并不是热点数据。

3.重做日志缓冲-redo log buffer

InnoDB存储引擎首先将重做日志信息放入到这个缓冲区，然后以一定频率将其刷新到重做日志文件。重做日志缓冲一般需要设置的很大，因为一般情况下每一秒会将重做日志缓冲刷新到日志文件。

重做日志缓冲刷新到日志文件的三种情况：

（1）Master Thread 每一秒将重做日志缓冲刷新到重做日志文件；

（2）每个事务提交时会将重做日志缓冲刷新到重做日志文件；

（3）当重做日志缓冲池剩余空间小于1/2时，重做日志缓冲刷新到重做日志文件。

4.额外的内存池

对一些数据结构本身的内存进行分配时，需要冲“额外的内存池”中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。因此，在申请了很大的InnoDB缓冲池时，也应该考虑相应地增加“额外的内存池”的大小。

5.Checkpoint技术

为了避免发生数据丢失的问题，当前事务数据库系统都采用了Write Ahead Log 策略，即当事务提交时，先写重做日志，在修改页。可以在发生宕机时，通过重做日志来完成数据的恢复。然而实际中由于缓冲池的内存大小有限，重做日志也不可能无限增大，最终导致恢复数据的代价很大。

因此Checkpoint（检查点）主要解决以下问题：

（1）缩短数据库的恢复时间；

（2）缓冲池不够用时，将脏页刷新到磁盘；

（3）重做日志不可用时，刷新脏页。

两种Checkpoint：

（1）Sharp Checkpoint

发生在数据库关闭时将所有的脏页都刷新回磁盘，这是默认的工作方式。在InnoDB存储引擎内部使用的是Fuzzy Checkpoint进行页的刷新。

（2）Fuzzy Checkpoint

a.Master Thread Checkpoint –每隔1秒或10秒的速度刷新

b.FLUSH\_LRU\_LIST Checkpoint - LRU中没有足够的空闲页则刷新

c.Async/Sync Flush Checkpoint – 重做日志不可用时，强制刷新

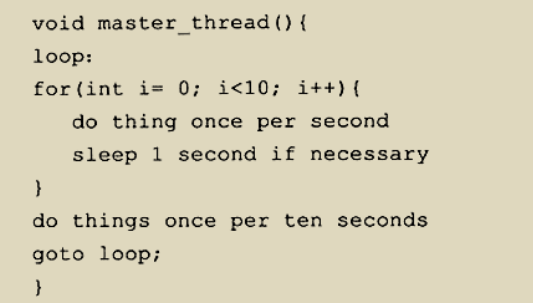
d.Dirty Page too much Checkpoint –脏页太多，为了保证缓冲池中足够的可用页

6.Master Thread 工作方式

主要线程具有最高的优先级，其内部有多个循环组成：

（1）主循环-loop

两大操作-每秒钟的操作和每10秒钟的操作。伪代码：



每秒一次的操作：

a.日志缓冲操作刷新到磁盘，即使这个事务没有提交（总是）

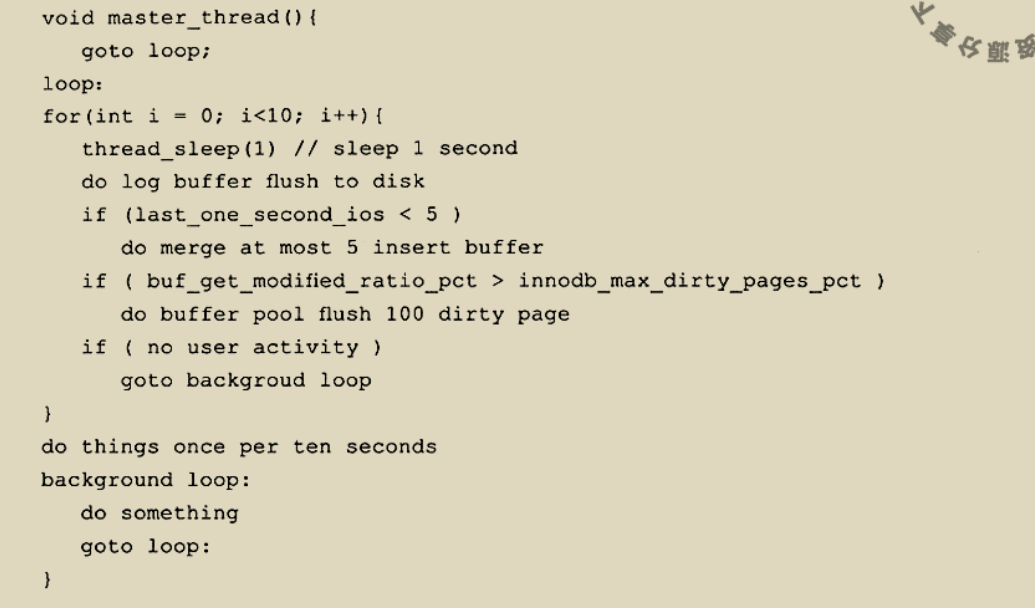
b.合并插入缓冲（可能）

根据前一秒发生的IO次数有关，大于某个值，则不执行。

c.至多刷新100个InnoDB的缓冲池的脏页到磁盘（可能）

d.如果当前没有用户活动，则切换到backgroup loop。

伪代码进一步具体化：



每10秒的操作：

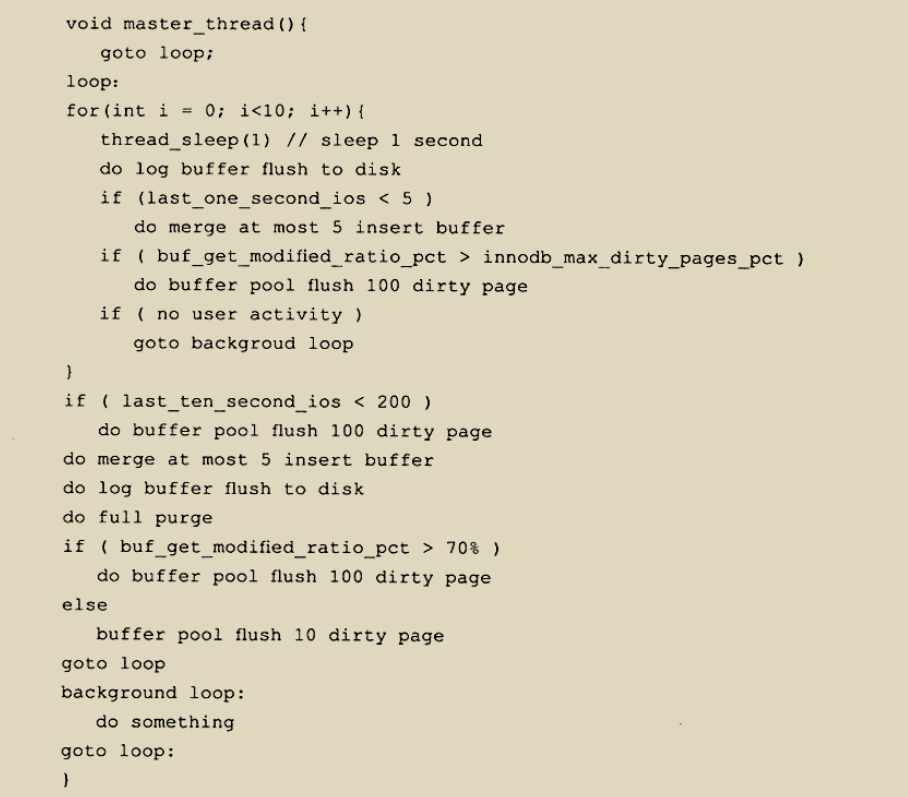
a.刷新100个脏页到磁盘（可能）

b.合并至多5个插入缓冲（总是）

c.将日志缓冲刷新到磁盘（总是）

d.删除无用的undo页（总是）

e.刷新100个或10脏页到到磁盘（总是）



（2）后台循环-backgroup loop

若当前用户活动（数据库空闲时）或者数据库关闭，就会切换到这个循环。

a.删除无用的undo页（总是）；

b.合并插入20个插入缓冲（总是）；

c.跳到主循环（总是）；

d.不断刷新100个页直到符合条件（可能，跳转到flush loop完成）。

（3）刷新循环-flush loop

最终完整的伪代码：



（4）暂停循环-suspend loop

6.InnoDB的关键技术

（1）插入缓冲- Insert Buffer

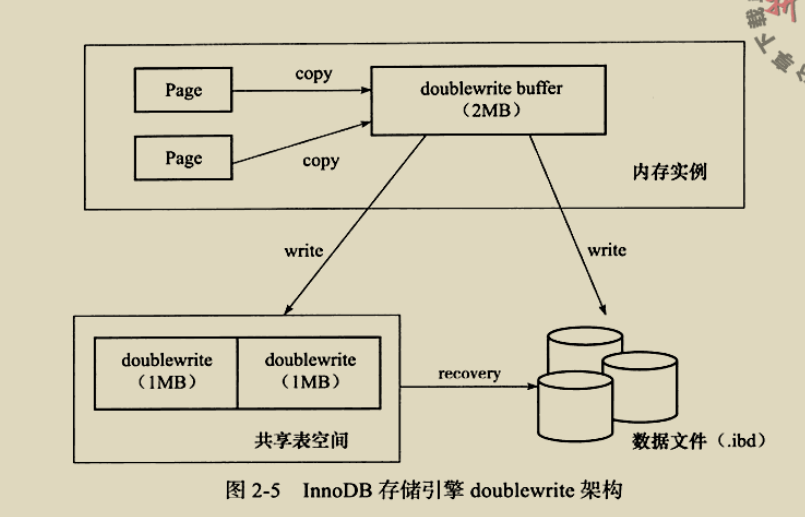
对于非聚簇索引的插入和更新操作，不是每次直接插入到索引页，而是先判断插入的非聚簇索引是否在缓冲中，若在，则直接插入；若不在，则先放在一个Insert Buffer对象中。然后以一定的频率进行Insert Buffer和辅助索引叶子节点的merge（合并）操作，这时通常能将多个插入合并到一个操作中，大大提高非聚簇索引的插入的性能。

Insert Buffer的使用需要同时满足两个条件：

a.索引是辅助索引

b.索引不是唯一的

（2）两次写



（3）自适应哈希索引

（4）异步IO

（5）刷新临接页

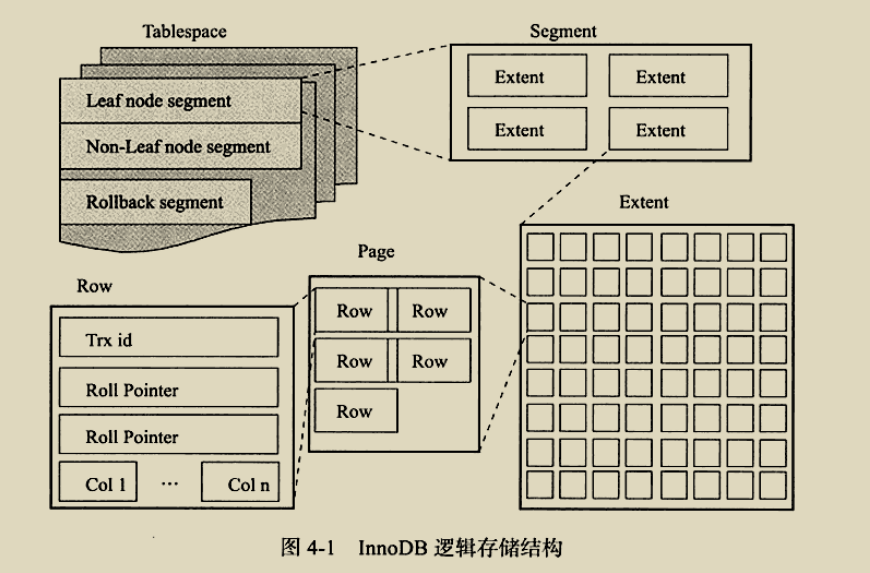
# 表

4.1 索引组织表

主键的选择根据的是定义索引的顺序，而不是建表时列的顺序。

4.2InnoDB逻辑存储结构

所有的数据都被逻辑地存放在一个空间中，称为表空间。表空间又由段，区，页组成。



4.2.1 表空间

所有的数据存在表空间中，默认情况下所有的数据都放在一个表空间内，也可以为每张表内的数据单独设置表空间。

4.2.2 段

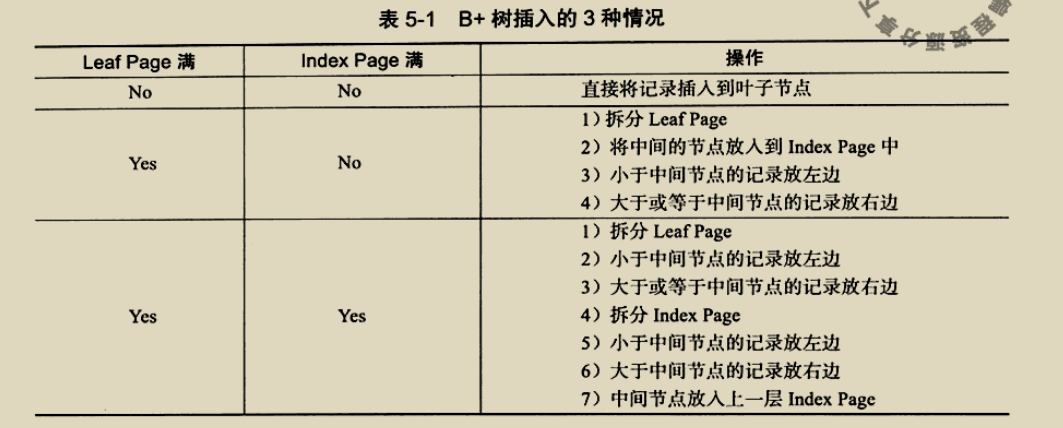
4.2.3 区

区是连续页组成的空间，在任何情况下每个区的大小为1MB。为了保证区中页的连续性，InnoDB存储引擎一次从磁盘申请4-5个区。在默认情况下，InnoDB存储引擎页的大小为16k,即一个区中一共有64连续的页。

B+树索引并不能找到一个给定键值的行。B+树索引能找到的只是被查数据行所在的页。然后数据库通过把页读入到内存，再在内存中进行查找，最后得到要查找的数据。

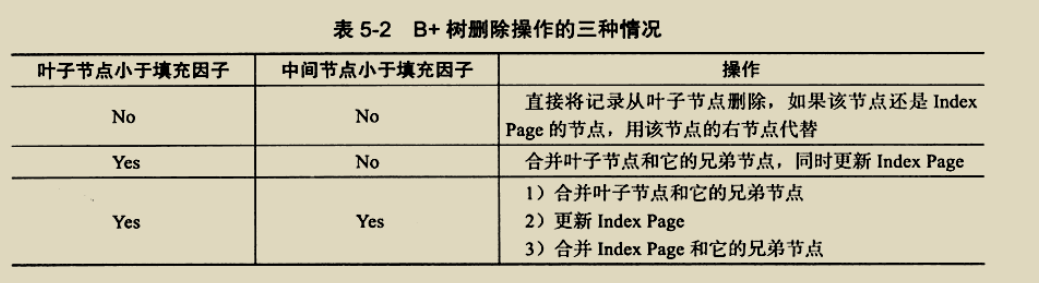
平衡二叉树：左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一颗平衡二叉树。

B+树的插入操作：插入必须保证叶子节点中的记录依然排序，同时需要考虑插入到B+树的三种情况，每种情况都可能导致不同的插入算法。



不管怎么变化，B+树总是会保持平衡。但是为了保持平衡对新插入的键值可能需要做大量的拆分页的操作，所以应该在可能的情况下尽可能减少页的拆分操作。

B+树的删除操作：B+树使用填充因子（fill factor）来控制树的删除变化，50%是填充因子可设的最小值。B+树的删除操作同样必须保证删除后叶子节点中的记录依然顺序，同插入一样B+树的插入操作需要考虑三种情况。



为了保证删除后叶子节点中的记录依然排序，可能需要做大量的合并操作。

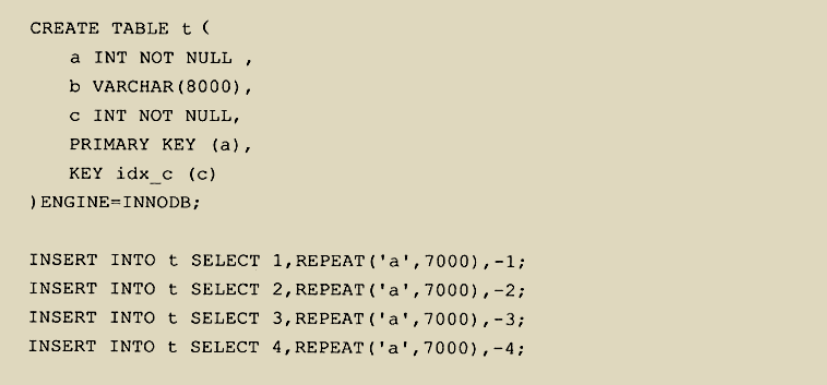
B+树索引

在数据库中，B+树的高度一般都在2-4层。数据库中的B+树索引可以分为聚集索引和辅助索引。聚集索引与辅助索引不同的是，叶子节点存放的是否是一整行的信息。

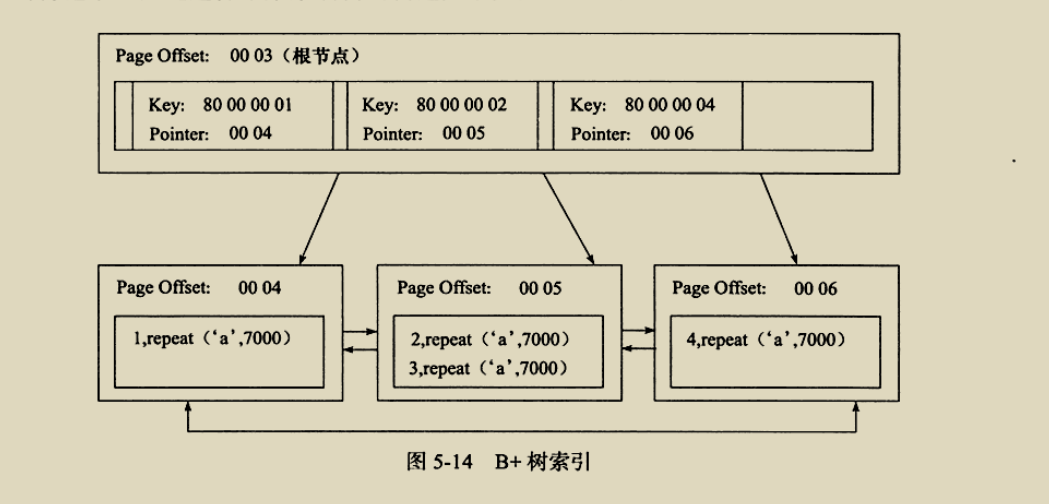
聚集索引

聚集索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的即为整张表的行记录数据，也将聚集索引中的叶子节点称为数据页。聚集索引的这个特性决定了索引组织表中数据也是索引的一部分。同B+树结构一样，每个数据页都通过一个双向链表来进行链接。由于实际的数据只能按照一棵B+树进行排序，因此每张表只能拥有一个聚集索引。

接下来看一张表，这里人为的方式让其每个页只能存放两行记录。



通过某些工具分析可以得出：数据页上存放的是完整的每行的记录，而在非数据页的索引页中，存放的仅仅是键值以及指向数据页的偏移量，而不是一个完整的行记录。因此这颗聚集树的构造大致如下：

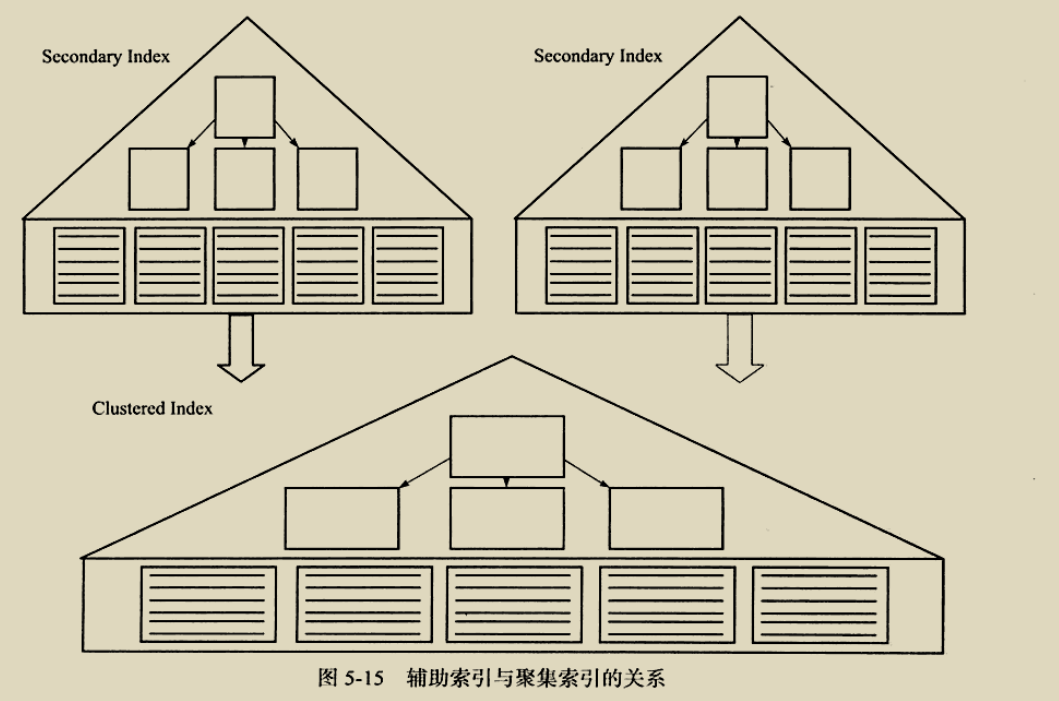


聚集索引的存储并不是物理上连续的，而是逻辑上连续的。这其中有两点：一是前面说过的页通过双向链表链接，页按照主键的顺序排序；另一个点是每个页中的记录也是通过双向链表进行维护的，物理存储上可以同样不按照主键存储。

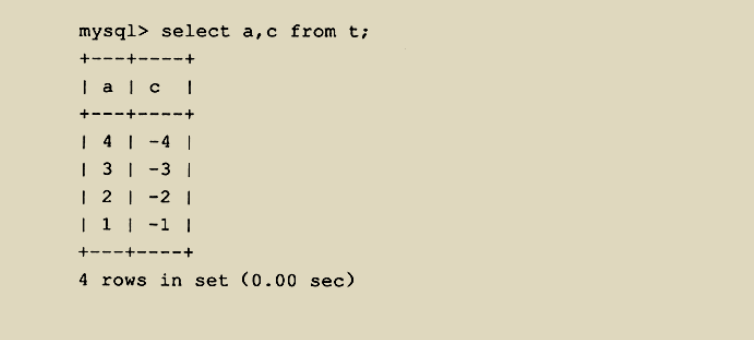
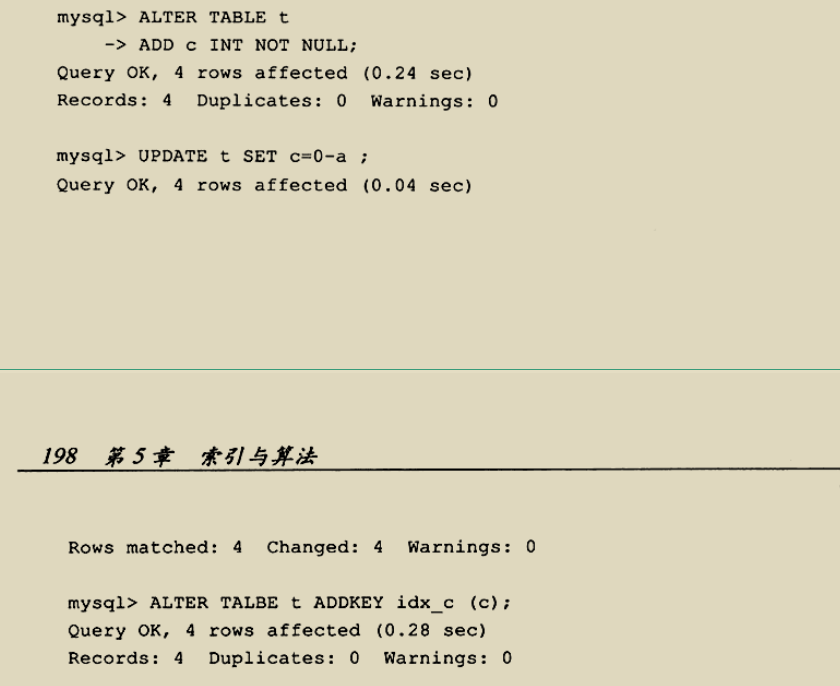
辅助索引

叶子节点并不包含行记录的全部数据。叶子节点除了包含键值以外，每个叶子节点中的索引行还包括一个书签（bookmark）。该书签来告诉InnoDB存储引擎哪里可以找到与索引相对应的行数据。由于InnoDB存储引擎表时索引组织表，因此InnoDB存储引擎的辅助索引的书签就是相应行数据的聚集索引。

辅助索引的存在并不影响数据在聚集索引中的组织，因此每张表上可以有多个辅助索引。当通过辅助索引来寻找数据时，InnoDB存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后再通过主键索引来找到一个完整的行的记录。

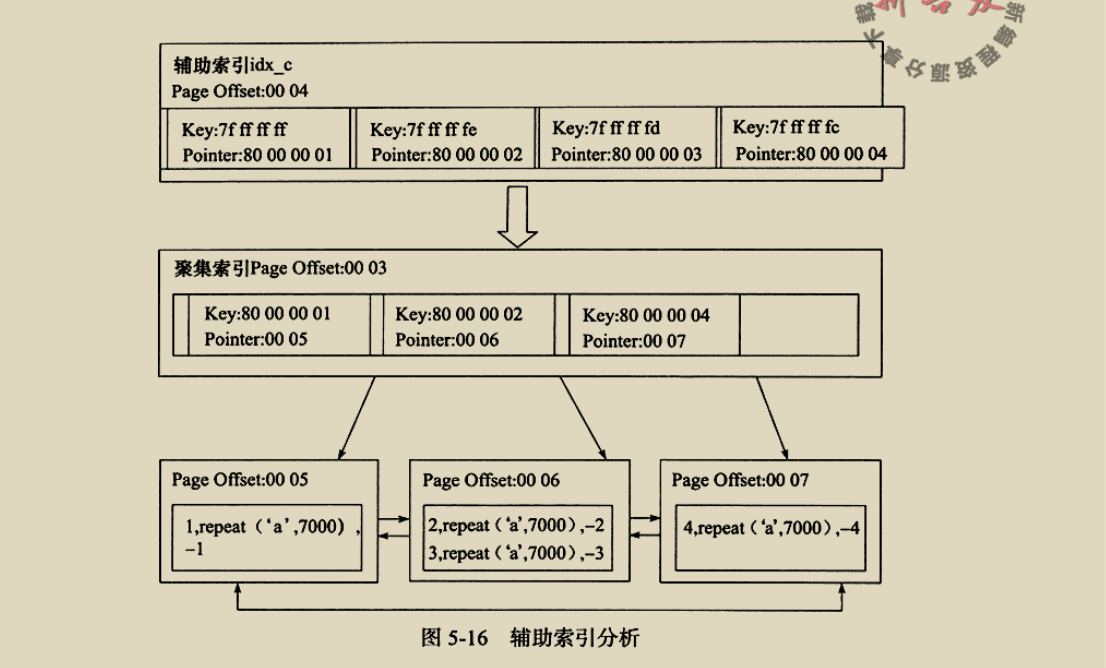


在上一个表t中，再建立一个列c，并对列c创建非聚集索引：



通过某些工具分析可得：下图显示了表t中辅助索引idx\_c和聚簇索引的关系。可看出辅助索引的叶子节点中包含列c的值和主键的值。-1内部以7f ff ff ff的方式进行内存存储。

7(0111)最高位0，代表负值，实际的值应该取反后加1，即得-1。



B+树索引的分裂

为了节省空间

5.6 B+树索引的使用

5.6.1 不同应用中B+树索引的使用

OLTP-在线事务处理

查询操作只从数据库中获取一小部分数据，一般可能10条记录一下，甚至很多时候只取一条记录，如通过主键获取用户信息，根据订单号获取订单详细信息。这些都是典型OLTP应用。此种场景适合使用B+树索引。

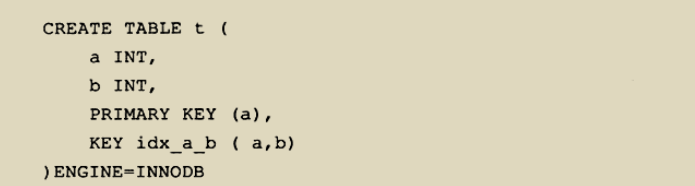
OLAP-在线分析处理

在OLAP应用中，都需要访问表中大量的数据，根据这些数据来产生查询的结果，这些查询多是面向分析的查询，目的是为决策者提供支持。因此OLAP中的索引的添加根据的应该是宏观的信息，而不是微观，因为最终要得到的结果是提供给决策者的。不过在OLAP应用中，通常会需要对时间字段进行索引，这是因为大多数统计需要根据时间维度来进行数据的筛选。

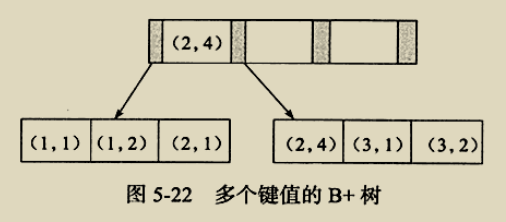
5.6.2 联合索引

联合索引是指对表上的多个列进行索引。

例如，以下代码创建了一张t表，并且索引idx\_a\_b是联合索引，联合的列为(a,b)。



联合索引的内部结果。从本质上来说，联合索引也是一棵B+树，不同的是联合索引的键值的数量不是1，而是大于等于2。假定两个键值的名称分别为a、b,如下图所示。



从图中可以看出，键值都是排序的，通过叶子节点可以逻辑上顺序地读取所有数据，就上面的例子来说，即(1,1)、(1,2)、(2,1)、(2,4)、(3,1)、(3,2)。数据按（a,b）的顺序进行了存放。

因此，对于查询select \* from table where a=xxx and b =xxx,显然是可以使用（a,b）这个联合索引的。对于单个的a列查询select \* from table where a=xxx，也可以使用这个索引（a,b）。但对于b列的查询select \* from table where b=xxx，则不可以使用这棵B+树索引。可以发现叶子节点上的b值为1、2、1、4、1、2，显示不是排序的，因此对于b列的查询使用不到（a,b）的索引。

联合索引的第二好处是已经对第二键值进行了排序处理。使用联合索引可以避免多一次排序操作，因为索引本身在叶子节点已经拍好序了。

5.6.2 覆盖索引-索引覆盖

即从辅助索引中就可以得到查询的记录，而不需要查询聚簇索引中的记录。使用覆盖索引的一个好处是辅助索引不包含整行的所有信息，故其大小要远小于聚簇索引，因此可以减少大量的IO操作。

覆盖索引的另一个好处是对于某些统计问题而言。在通常情况下，诸如(a,b)的联合索引，一般是不可以选择b中所谓的查询条件。但是如果是统计操作，并且是覆盖索引的，则优化器进行选择。

select count(\*) from buy\_log where buy\_date >= ‘2011-01-01’ and buy\_date< ‘2011-02-01’