1.

mysql|索引与innodb结构
一.innodb是如何存储数据的?
SQL对于新增是有一个规则->依赖于索引->mysql会根据索引定位数据在磁盘中的空间
即使一张表的数据量很大,也不会影响新增,而查询会受到影响

2.

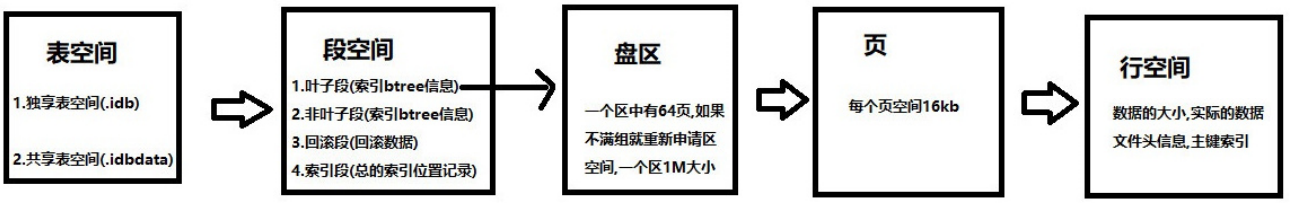
二.innodb的结构
1.表空间:
表空间分为了两种，这里简单的概括一下：
(1)独立表空间：每一个表都将会生成以独立的文件方式来进行存储，每一个表都有一个.frm表描述文件，还有一个.ibd文件。 其中这个文件包括了单独一个表的数据内容以及索引内容，默认情况下它的存储位置也是在表的位置之中。
(2)共享表空间： Innodb的所有数据保存在一个单独的表空间里面，而这个表空间可以由很多个文件组成，一个表可以跨多个文件存在，所以其大小限制不再是文件大小的限制，而是其自身的限制。从Innodb的官方文档中可以看到，其表空间的最大限制为64TB，也就是说，Innodb的单表限制基本上也在64TB左右了，当然这个大小是包括这个表的所有索引等其他相关数据。

2.分片段(Segment):
段是表空间文件中的主要组织结构，它是一个逻辑概念，用来管理物理文件，是构成索引、表、回滚段的基本元素。 创建一个索引（B+树）时会同时创建两个段，分别是内节点段和叶子段，内节点段用来管理（存储）B+树非叶子（页面）的数据，叶子段用来管理（存储）,B+树叶子节点的数据；也就是说，在索引数据量一直增长的过程中，所有新的存储空间的申请，都是从“段”这个概念中申请 的。

3.盘区:

4.页(page):
页是InnoDB磁盘管理的最小单位 对于innodb来说就是 16KB 不过如果是oracle 或者SQLserver就是4kb的大小。

3.



4.

三.索引介绍
在MySQL中，主要有4中类型的索引0，分别为：B-Tree索引，Hash索引、Fulltext索引和R-Tree索引。
常见的索引类型:
主键索引:就是我们的主键
唯一索引:一个唯一的字段的索引
单索引:单个字段的索引
联合索引:多个字段联合建立的索引
全文索引:针对于中进行分词的搜索
覆盖索引:是所有查询SQL中最求的索引效率的完美使用

5.

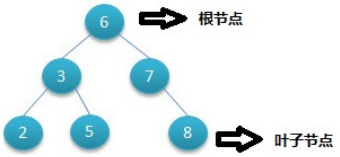
四.BTree和B+Tree详解
B+树索引是B+树在数据库中的一种实现，是最常见也是数据库中使用最为频繁的一种索引。B+树中的B代表平衡（balance），而不是二叉（binary），因为B+树是从最早的平衡二叉树演化而来的。在讲B+树之前必须先了解二叉查找树、平衡二叉树（AVLTree）和平衡多路查找树（B-Tree），B+树即由这些树逐步优化而来。

(1)二叉查找树
二叉树具有以下性质：左子树的键值小于根的键值，右子树的键值大于根的键值。
如下图所示就是一棵二叉查找树，

但是这棵二叉树的查询效率就低了。因此若想二叉树的查询效率尽可能高，需要这棵二叉树是平衡的，从而引出新的定义—平衡二叉树，或称AVL树。

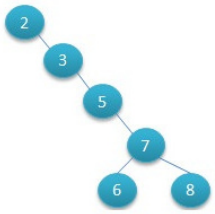
6.

如下图所示就是一棵二叉查找树，



对该二叉树的节点进行查找发现深度为1的节点的查找次数为1，深度为2的查找次数为2，深度为n的节点的查找次数为n，因此其平均查找次数为 (1+2+2+3+3+3) / 6 = 2.3次

二叉查找树可以任意地构造，同样是2,3,5,6,7,8这六个数字，也可以按照下图的方式来构造：



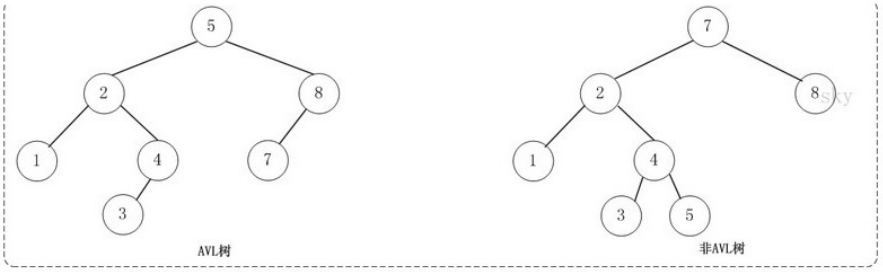
二插树

7.

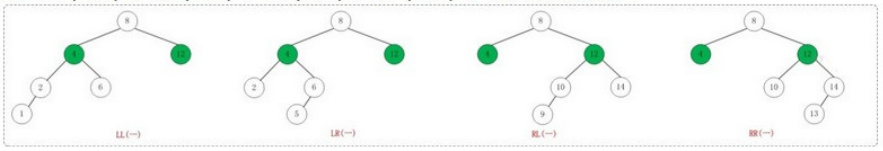
(2)平衡二叉树（AVL Tree）

平衡二叉树（AVL树）在符合二叉查找树的条件下，还满足任何节点的两个子树的高度最大差为1。下面的两张图片，左边是AVL树，它的任何节点的两个子树的高度差<=1；右边的不是AVL树，其根节点的左子树高度为3，而右子树高度为1；

8.



如果在AVL树中进行插入或删除节点，可能导致AVL树失去平衡，这种失去平衡的二叉树可以概括为四种姿态：LL（左左）、RR（右右）、LR（左右）、RL（右左）。它们的示意图如下：



9.

(3)平衡多路查找树（B-Tree）

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

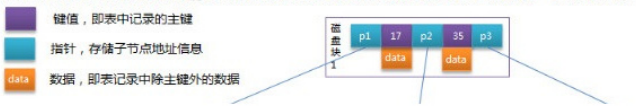
InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb_page_size将页的大小设置为4K、8K、16K，在MySQL中可通过如下命令查看页的大小：mysql> show variables like 'innodb_page_size';

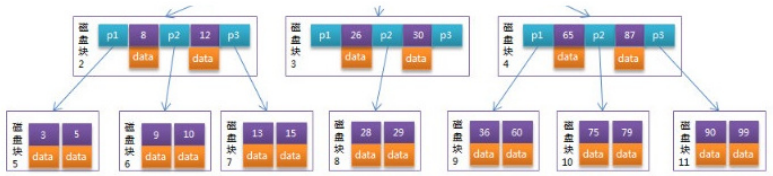
而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key，data] ，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

10.

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：





每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

B-tree

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

11.

(4)B+Tree

B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

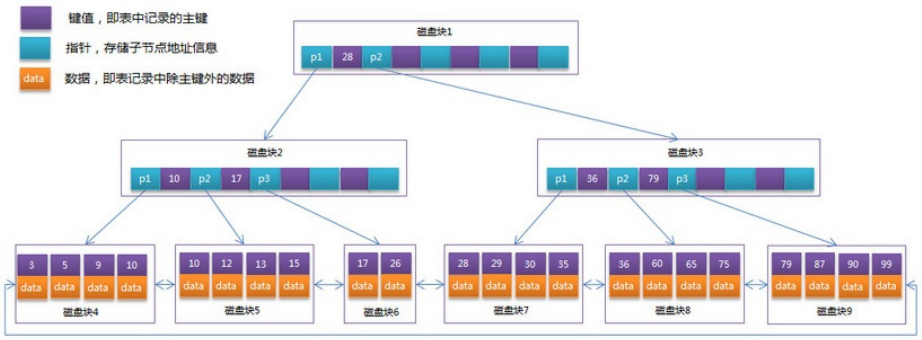
从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

非叶子节点只存储键值信息。
所有叶子节点之间都有一个链指针。
数据记录都存放在叶子节点中。

12.

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：



通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 * 10^3 * 10^3 = 10亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2~4层。mysql的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。