Java MySQL系列课程03

------ 索引以及底层原理

本文档包含以下内容：

1. 索引的分类
2. 索引的创建和删除
3. 索引的执行过程
4. 索引的底层实现原理
5. 主键索引，辅助索引，聚集索引，非聚集索引
6. 索引的设计原则

#### 前言：

索引就像书籍的目录，当我们要搜索想看的内容的时候，先从目录搜索，然后找到内容的页数，再翻到相应的页去查看就可以了，比我们一页页翻着查找要快很多，所以索引的核心就是加快SQL的查询。

当我们表中的数据量上万，几十万甚至上百万的时候，一个SQL查询所花费的时间就会很长，那么此时就需要用到索引来加速SQL查询，具体内容请看下面对索引的描述。

但由于索引也是需要存储成索引文件的，因此对于索引的使用，也涉及大量的磁盘I/O操作，如果索引创建过多，使用不当，会造成SQL查询时，进行大量无用的磁盘I/O操作，降低了SQL查询效率，适得其反，因此掌握索引的执行过程和实现原理，非常重要！

#### 一 索引的分类：

索引是创建在表上的，是对数据库表中一列或者多列的值进行排序的一种结果。**索引的核心是提高查询的速度！**

**索引的优点： 提高查询效率**

**索引的缺点： 索引并非越多越好，过多的索引会导致CPU使用率居高不下，由于数据的改变，会造成索引文 件的改动，过多的磁盘I/O造成CPU负荷太重**

**1.普通索引：**没有任何限制条件，可以给任何类型的字段创建普通索引

**2.唯一性索引：**使用UNIQUE修饰的字段，值不能够重复，主键索引就隶属于唯一性索引

**3.主键索引：**使用Primary Key修饰的字段会自动创建索引

**4.单列索引：**在一个字段上创建索引

**5.多列索引：**在表的多个字段上创建索引

**6.全文索引：**使用FULLTEXT参数可以设置全文索引，只支持CAHR，VARCHAR和TEXT类型的字段上，

常用语数据量较大的字符串类型上，可以提高查询速度；只有MyISAM存储引擎支持

#### 二 索引的创建和删除：

**创建表的时候指定索引字段：**

**CREATE TABLE index1(id INT,**

**name VARCHAR(20),**

**sex ENUM('male', 'female'),**

**INDEX(id));**

**在已经创建的表上添加索引：**

**CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX 索引名**

**ON 表名 （属性名 [ASC | DESC]）;**

**删除索引：**

**DROP INDEX 索引名 ON 表名;**

#### 三 索引的执行过程：

根据第二步的内容，大家已经知道怎么在表上创建索引和删除索引，那么索引具体有没有产生作用，也就是索引是否生效，我们通过什么方法来查看呢？下面我们创建一张简单的表，测试如下：

先创建student这张表，如下：

mysql> **create table student(id int primary key,**

**-> name varchar(20),**

**-> score double default 0.0);**

Query OK, 0 rows affected (0.89 sec)

然后插入一些数据，如下：

mysql> **insert into student values(1, 'zhang', 60.5),(2,'gao',78.6),(3,'liu',89.2),(4,'wang',95.6);**

Query OK, 4 rows affected (0.13 sec)

Records: 4 Duplicates: 0 Warnings: 0

查看表的内容，如下：

mysql> **select \* from student;**

+----+-------+-------+

| id | name | score |

+----+-------+-------+

| 1 | zhang | 60.5 |

| 2 | gao | 78.6 |

| 3 | liu | 89.2 |

| 4 | wang | 95.6 |

+----+-------+-------+

4 rows in set (0.00 sec)

现在想找名字为liu的同学的成绩，SQL如下：

mysql> **select score from student where name='liu';**

+-------+

| score |

+-------+

| 89.2 |

+-------+

1 row in set (0.00 sec)

用explain查看一下上面SQL语句的执行计划，如下：

mysql> **explain select score from student where name='liu'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: ALL

**possible\_keys: NULL**

**key: NULL**

key\_len: NULL

ref: NULL

**rows: 4**

filtered: 25.00

Extra: Using where

1 row in set, 1 warning (0.00 sec)

从上面的打印可以看到，在student表中查询liu同学的成绩时，是做了整表查询操作，把表中的所有行（rows：4）全部扫描了一遍，才找到liu同学的信息，那假设这张表有4百万行数据，那么为了查找liu同学的信息，要搜索4百万行数据，这效率实在是太低了！

因此，给student表中的name字段添加一个索引，如下：

mysql> **create index name\_index on student (name);**

Query OK, 0 rows affected (0.73 sec)

Records: 0 Duplicates: 0 Warnings: 0

此时再用explain分析一个上面的SQL查询语句，信息如下：

mysql> **explain select score from student where name='liu'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: student

partitions: NULL

type: ref

**possible\_keys: name\_index**

**key: name\_index**

key\_len: 63

ref: const

**rows: 1**

filtered: 100.00

Extra: NULL

1 row in set, 1 warning (0.03 sec)

注意，这次查询liu同学的成绩信息，只在表中**查询一行数据**就得到了，而且从上面显示信息可以看到，这个SQL语句的查询，使用了名叫name\_index的索引，这就是我们上面给name字段创建的索引，可以发现，**使用索引对SQL语句查询效率的提升是非常大的！所以，请记住explain命令，可以查看SQL执行计划，分析该SQL语句是否能够正确使用索引！**

**至于索引是怎么样提高查询效率的，请看下一节索引的底层实现原理！**

#### 四 索引的底层实现原理：

MySQL支持两种索引，一种的B-树索引，一种是哈希索引，大家知道，B-树和哈希表在数据查询时的效率是非常高的。

这里我们主要讨论一下MySQL InnoDB存储引擎，**基于B-树（但实际上MySQL采用的是B+树结构）**的索引结构。

1. 树是一种m阶平衡树，叶子节点都在同一层，由于每一个节点存储的数据量比较大，索引整个B-树的层数是非常低的，基本上不超过三层。

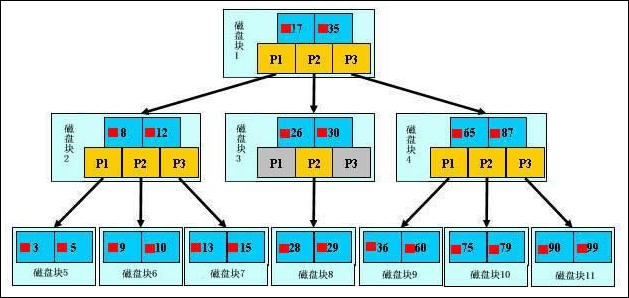
由于磁盘的读取也是按block块操作的（内存是按page页面操作的），**因此B-树的节点大小一般设置为和磁盘块大小一致**，这样一个B-树节点，就可以通过一次磁盘I/O把一个磁盘块的数据全部存储下来，所以当使用B-树存储索引的时候，磁盘I/O的操作次数是最少的（MySQL的读写效率，主要集中在磁盘I/O上）。

**那么MySQL最终为什么要采用B+树存储索引结构呢，那么看看B-树和B+树在存储结构上有什么不同？**

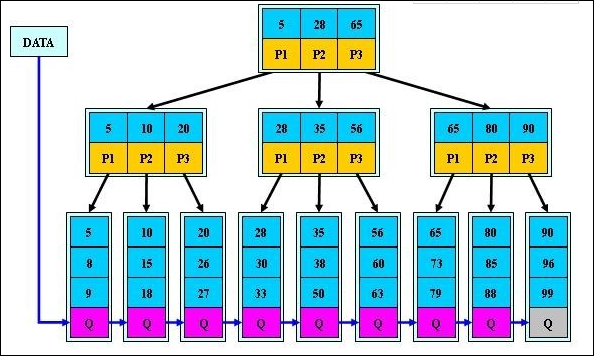
1. B-树的每一个节点，存了关键字和对应的数据地址，而B+树的非叶子节点只存关键字，不存数据地址。因此B+树的每一个非叶子节点存储的关键字是远远多于B-树的，B+树的叶子节点存放关键字和数据，因此，从树的高度上来说，**B+树的高度要小于B-树，使用的磁盘I/O次数少，因此查询会更快一些。**
2. B-树由于每个节点都存储关键字和数据，因此离根节点进的数据，查询的就快，离根节点远的数据，查询的就慢；B+树所有的数据都存在叶子节点上，因此**在B+树上搜索关键字，找到对应数据的时间是比较平均的，没有快慢之分。**
3. 在B-树上如果做区间查找，遍历的节点是非常多的；B+树所有叶子节点被连接成了有序链表结构，**因此做整表遍历和区间查找是非常容易的。**

**哈希索引**当然是由哈希表实现的，哈希表对数据并不排序，**因此不适合做区间查找**，效率非常低，需要搜索整个哈希表结构。

**B-树的结构图如下：**



**B+树的结构图如下：**



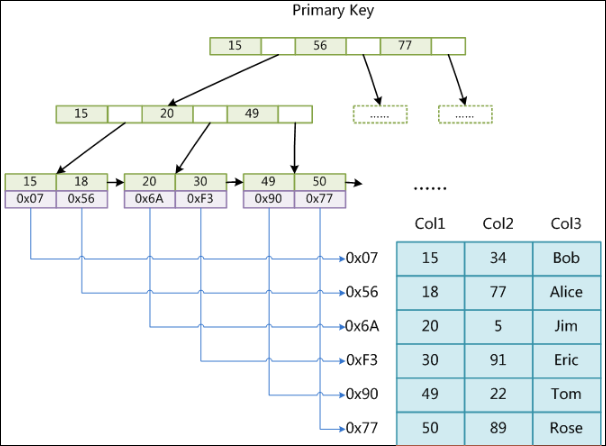
**从上面的两个图，请验证前面给出的MySQL最终采用了B+树存储索引的结论。请仔细理解索引的底层实现原理（B+树），这个是面试经常会问到的问题！**

#### 五 主键索引，辅助索引，聚集索引，非聚集索引：

我们主要学习一下MySQL两个重要的存储引擎，**MyISAM和InnoDB存储引擎的索引结构。**

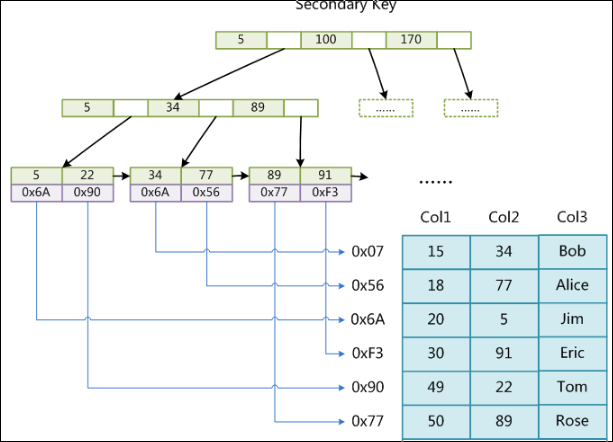
1. **MyISAM存储引擎 - 主键索引**

MyISAM引擎使用B+树作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM主键索引的原理图：



1. **MyISAM存储引擎 - 辅助索引**

在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复，如果给其它字段创建辅助索引，结构图如下：



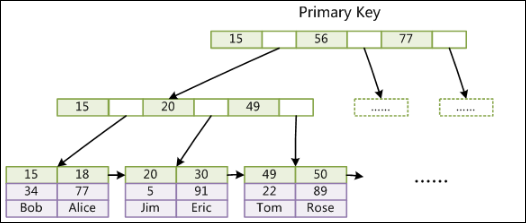
根据上面两张图，首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

可以看到，MyISAM存储引擎，索引结构叶子节点存储关键字和数据地址，也就是说索引关键字和数据没有在一起存放，体现在磁盘上，就是索引在一个文件存储，数据在另一个文件存储，例如一个user表，会在磁盘上存储三个文件 **user.frm（表结构文件） user.MYD（表的数据文件） user.MYI（表的索引文件）。**

**MyISAM的索引方式也叫做非聚集索引，之所以这么称呼是为了与InnoDB的聚集索引区分！**

1. **InnoDB存储引擎 - 主键索引**

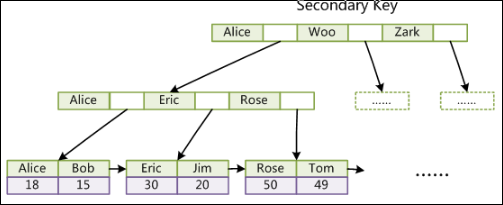
InnoDB存储引擎的主键索引，叶子节点中，索引关键字和数据是在一起存放的，如图：



可以看到，索引关键字和数据在叶节点上，在一起存储。

1. **InnoDB存储引擎 - 辅助索引**

InnoDB的辅助索引，叶子节点上存放的是索引关键字和对应的主键，如图：



辅助索引的B+树，先根据关键字找到对应的主键，再去主键索引树上找到对应的行记录数据。

从索引树上可以看到，InnoDB的索引关键字和数据都是在一起存放的，体现在磁盘存储上，例如创建一个user表，在磁盘上只存储两种文件，**user.frm（存储表的结构），user.ibd（存储索引和数据）。**

**InnoDB的索引树叶节点包含了完整的数据记录，这种索引叫做聚集索引。**因为InnoDB的数据文件本身要按主键聚集，所以**InnoDB要求表必须有主键（MyISAM可以没有）**，如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，**如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。**

#### 六 索引的设计原则：

从前面的内容可以看到，索引固然很好，但是给表创建过多的索引，效率反而会降低，因此在给表设计索引的时候，**需要遵循以下的设计原则：**

**1.给区分度高的字段创建索引**

**2.给经常需要排序，分组和多表联合操作的字段创建索引**

**3.给常作为查询条件的字段创建索引**

**4.索引的数目不宜过多**

**5.使用数据量少的索引（如前缀索引，主要针对字符串类型，字符串类型尽量创建前缀索引）**

**6.对于多列索引，优先指定最左边的列集**

**7.删除不再使用或者很少使用的索引**

**以上7点是设计索引必须要遵守的，你能够根据索引的使用和底层实现原理理解它们吗？**