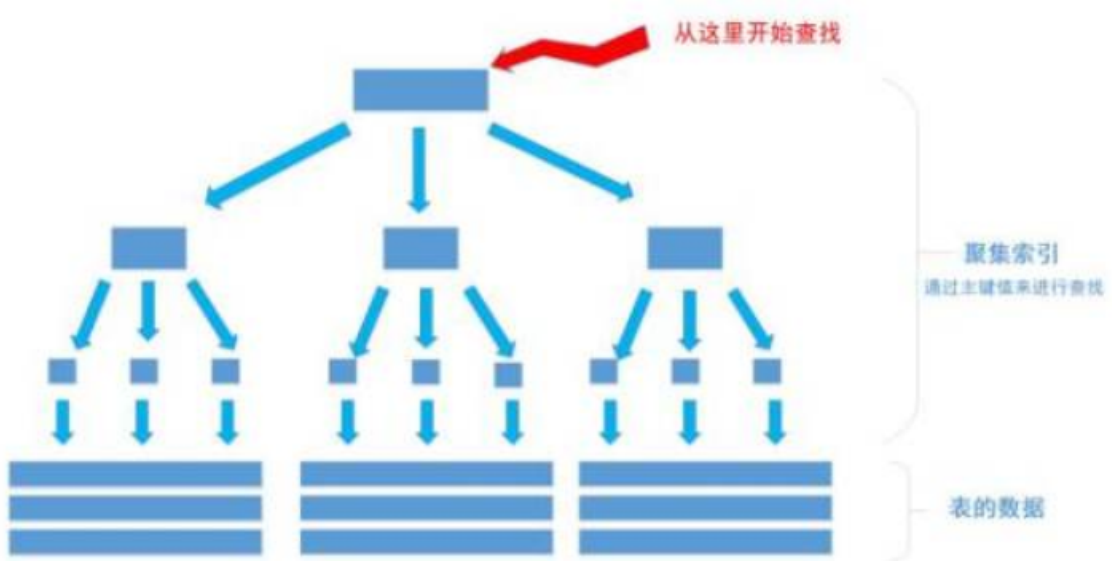
MySql - 索引

1.为什么要给表加上主键

一个重要概念：平衡树，平衡因子的绝对值不能超过1
MyISAM和InnoDB存储引擎只支持BTREE索引，即将平衡树当做数据表默认的索引数据结构

#1.平时建表的时候都会为表加上主键，在某些关系数据库中,如果建表时不指定主键，数据库会拒绝建表的语句执行。
#2.一个没加主键的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐， 跟我认知中的'表'很接近
#3.事实上， 一个加了主键的表，并不能被称之为'表',如果给表上了主键，那么表在磁盘上的存储结构就由整齐排列的结构转变成了树状结构，也就是上面说的「平衡树」结构,换句话说，就是整个表就变成了一个索引 -->聚集索引

主键的作用就是把「表」的数据格式转换成「索引（平衡树）」的格式放置。（如下图）



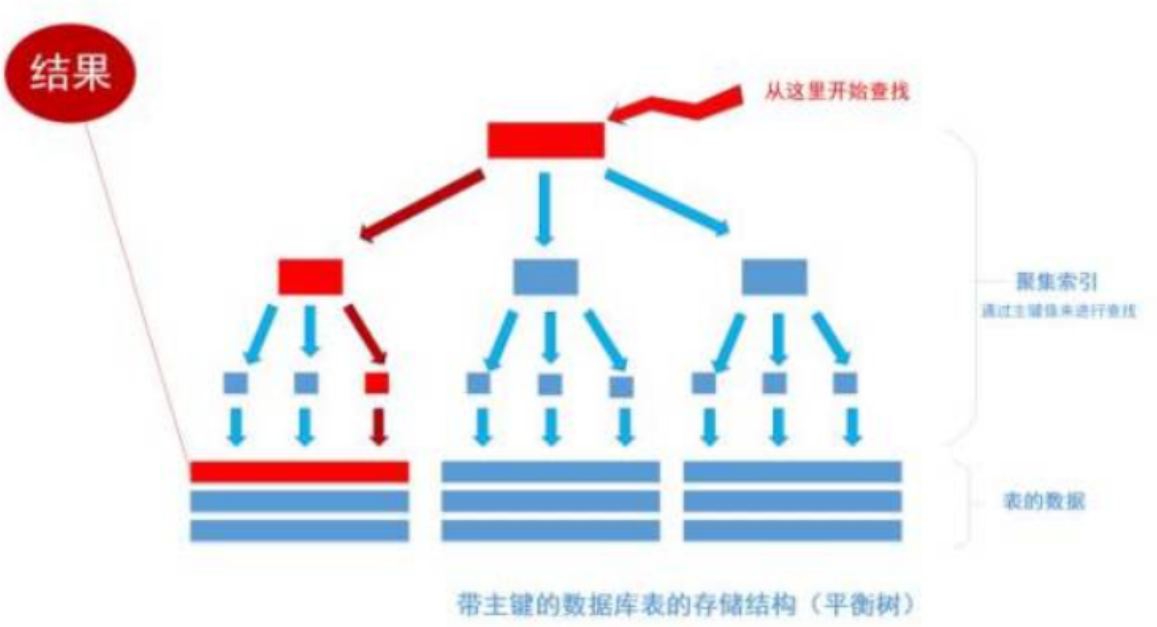
带主键的数据库表的存储结构（平衡树）

上图就是带有主键的表（聚集索引）的结构图
其中树的所有结点（底部除外）的数据都是由主键字段中的数据构成，即节点只存key，也就是通常我们指定主键的id字段。最下面部分是真正表中的数据。 假如我们执行一个SQL语句：

```
    select * from table where id = 1256;
```

#首先根据索引定位到1256这个值所在的叶结点，然后再通过叶结点取到id等于1256的数据行。 这里不讲解平衡树的运行细节， 但是从上图能看出，树一共有三层， 从根节点至叶节点只需要经过三次查找就能得到结果。如下图

#另外在数据之间是一个双链表的结构
select * from table where id > 1256;那么据不需要再一层一层往上面找，直接找他后边的数据就行



带主键的数据库表的存储结构（平衡树）

2.为什么加索引后会使查询变快?

#不使用索引 O(N)

假如一张表有一亿条数据 ，需要查找其中某一条数据，按照常规逻辑， 一条一条的去匹配的话， 最坏的情况下需要匹配一亿次才能得到结果，用大O标记法就是O(n)最坏时间复杂度，这是无法接受的，而且这一亿条数据显然不能 一次性读入内存供程序使用， 因此， 这一亿次匹配在不经缓存优化的情况下就是一亿次IO开销，以现在磁盘的IO能 力和CPU的运算能力， 有可能需要几个月才能得出结果。

#使用索引

如果把这张表转换成平衡树结构（一棵非常茂盛和节点非常多的树），假设这棵树有10层，那么只需要10次IO开销就能查找到所需要的数据， 速度以指数级别提升，用大O标记法就是O(logn)

因此，利用索引会使数据库查询有惊人的性能提升

3.为什么加索引后会使写入、修改、删除变慢?

然而,事物都是有两面的,索引能让数据库查询数据的速度上升而使写入数据的速度下降，原因很简单的，

#因为平衡树这个结构必须一直维持在一个正确的状态:

增删改数据都会改变平衡树各节点中的索引数据内容，破坏树结构

因此，在每次数据改变时，DBMS必须去重新梳理树（索引）的结构以确保它的正确，这会带来不小的性能开销，也就是为什么索引会给查询以外的操作带来副作用的原因。

4.当我给表加上主键后，根据非主键字段查找时怎么加快，要是设置索引，查询机制是什么?

#引子：当我们设置id为主键，当我们执行下列查询语句:

```
select * from table where name = alex:
```

遍历树结构，一个个找，同样很慢，因此需要如果将name也创建索引，就是常规索引即非聚集索引

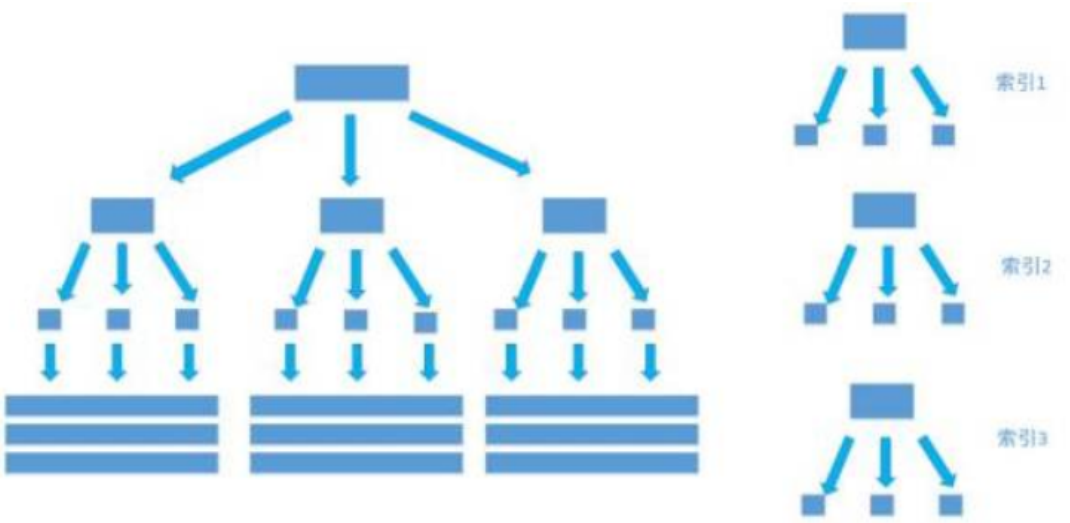
#非聚集索引和聚集索引一样，同样是采用平衡树作为索引的数据结构。

索引树结构中各节点的值来自于表中的索引字段

假如给user表的name字段加上索引，那么索引就是由name字段中的值构成，也就是说此数的节点就是name

在数据改变时， DBMS需要一直维护索引结构的正确性。如果给表中多个字段加上索引，那么就会出现多个独立的索引结构，每个索引（非聚集索引）互相之间不存在关联。

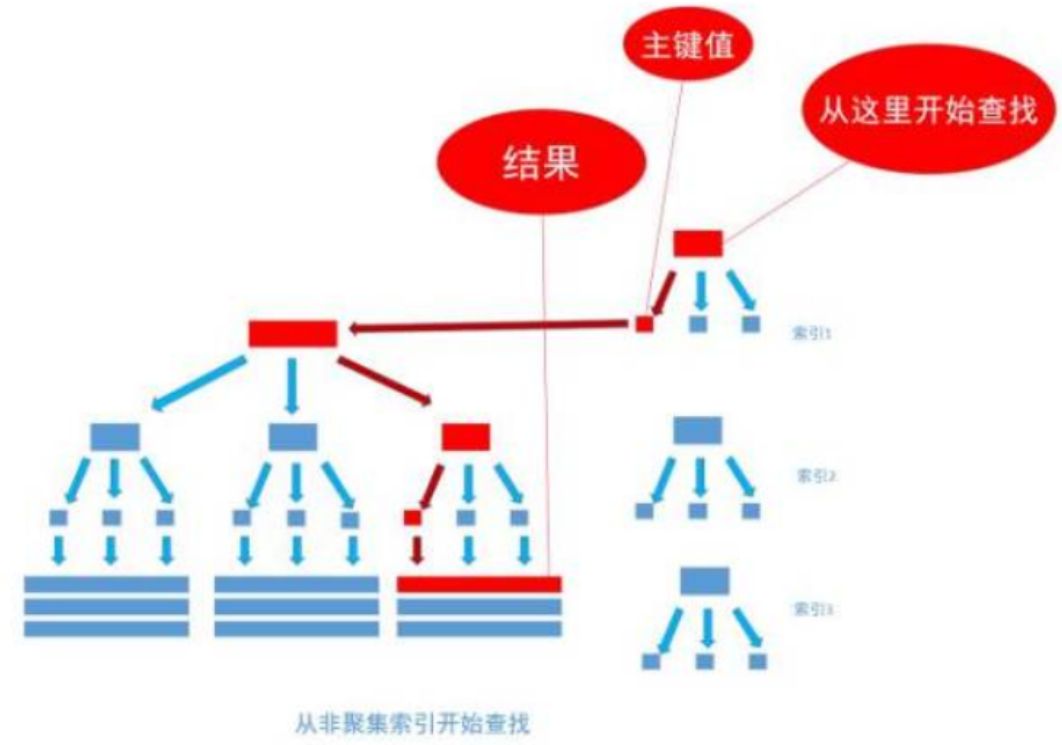
每次给字段建一个新索引，字段中的数据就会被复制一份出来， 用于生成索引。 因此给表添加索引，会增加表的体积，占用磁盘存储空间



带有主键和三个非聚集索引的表的存储结构

#非聚集索引和聚集索引的区别在于，通过聚集索引可以查到需要查找的数据，而通过非聚集索引可以查到记录对应的主键值，再使用主键的值通过聚集索引查找到需要的数据，如下图

不管以任何方式查询表， 最终都会利用主键通过聚集索引来定位到数据， 聚集索引（主键）是通往真实数据所在的唯一路径



5.什么情况下要同时在两个字段上建索引?

#有一种例外可以不使用聚集索引就能查询出所需要的数据，这种非主流的方法 称之为「覆盖索引」查询
即的复合索引或者多字段索引查询
当为字段建立索引以后，字段中的内容会被同步到索引之中，如果为一个索引指定两个字段，那么这个两个字段的内容都会被同步至索引之中。

#先看下面这个SQL语句

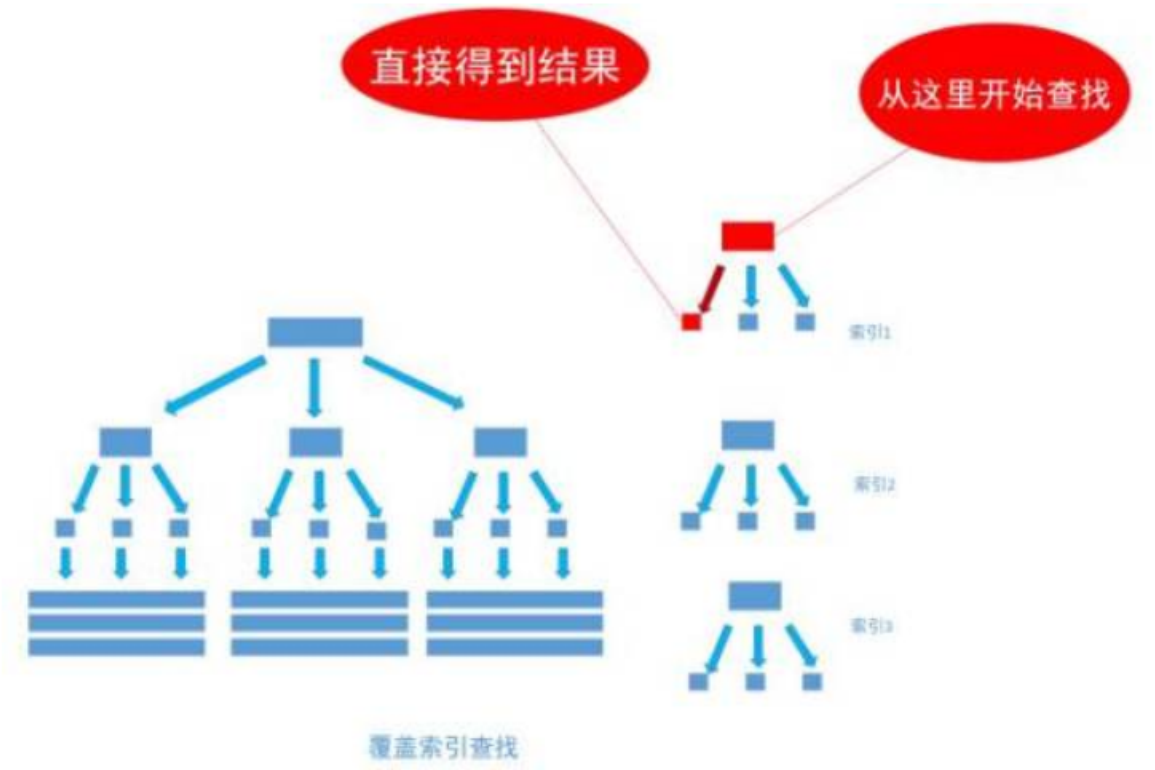
```
create index index_age on user_info(age);
select user_name from user_info where age = '20'
```

这句SQL语句的执行过程如下
首先，通过非聚集索引index_age查找age等于19的所有记录的主键ID值
然后，通过得到的主键ID值执行聚集索引查找，找到主键ID值对就的真实数据（数据行）存储的位置
最后， 从得到的真实数据中取得user_name字段的值返回， 也就是取得最终的结果

#我们把birthday字段上的索引改成双字段的覆盖索引

```
create index index_age_and_user_name on user_info(age, user_name);
```

#这句SQL语句的执行过程就会变为
通过非聚集索引index_age_and_user_name查找age等于19的叶节点的内容
叶节点中除了有user_name表主键ID的值以外,user_name字段的值也在里面,因此不需要通过主键ID值的查找数据行的真实所在,直接取得叶节点中user_name的值返回即可
通过这种覆盖索引直接查找的方式,可以省略不使用覆盖索引查找的后面两个步骤,大大的提高了查询性能，如下图



6.什么是最左前缀的规则

```
#使用联合索引时候,如果想要命中索引必须遵循最左前缀原则
#最左匹配原则中，有如下说明:
    最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如    果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

    =和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式
```

7.如何命中索引

```
1.注意范围问题,或者说条件不明确， 条件中出现这些符号或关键字：>、>=、<、<=、!= 、between.and.、like
2.尽量使用区分度高 的列设置索引
3.索引列不能在条件中参与计算
4.使用最左前缀匹配原则
5.不要使用函数  select * from tb1 where reverse(email) = 'egon';sql语句出现函数时候,索引也未命中
6.注意类型一致
7.条件为索引，则select字段必须也是索引字段，否则无法命中 select name from s1 order by email desc;
```

8.如何创建索引：

```
#方式一
create table t1(
    id int,
    name char,
    age int,
```

```
    sex enum('male','female'),
    unique key uni_id(id),
    index ix_name(name) #index没有key
);
create table t1(
    id int,
    name char,
    age int,
    sex enum('male','female'),
    unique key uni_id(id),
    index(name) #index没有key
);

#方式二
create index ix_age on t1(age);

#方式三
alter table t1 add index ix_sex(sex);
alter table t1 add index(sex);

#查看
mysql> show create table t1;
| t1    | CREATE TABLE `t1` (
  `id` int(11) DEFAULT NULL,
  `name` char(1) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `sex` enum('male','female') DEFAULT NULL,
  UNIQUE KEY `uni_id` (`id`),
  KEY `ix_name` (`name`),
  KEY `ix_age` (`age`),
  KEY `ix_sex` (`sex`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```