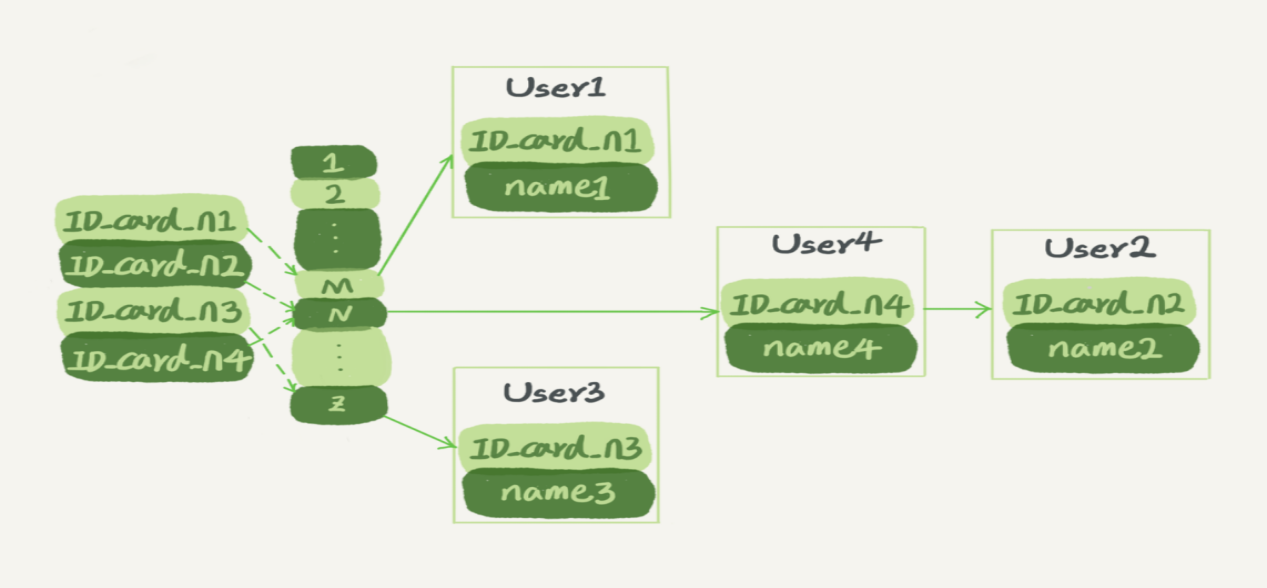
4| 索引（上）--常见索引结构

索引常见模型

1. 哈希表是一种键-值（key-value）存储的数据结构，只要输入key,就能找到对应的值value，把值放在数组里，用一个哈希函数把key换算成一个确定的位置，然后把value放在数组的这个位置。

不可避免的多个key值经过哈希函数算法，会出现同一个值得情况，处理这种情况的一种方法是拉出一个链表

假设，现在维护一个身份证信息和姓名的表，需要身份证号查找对应的名字，这是对应的哈希索引示意图如下所示：

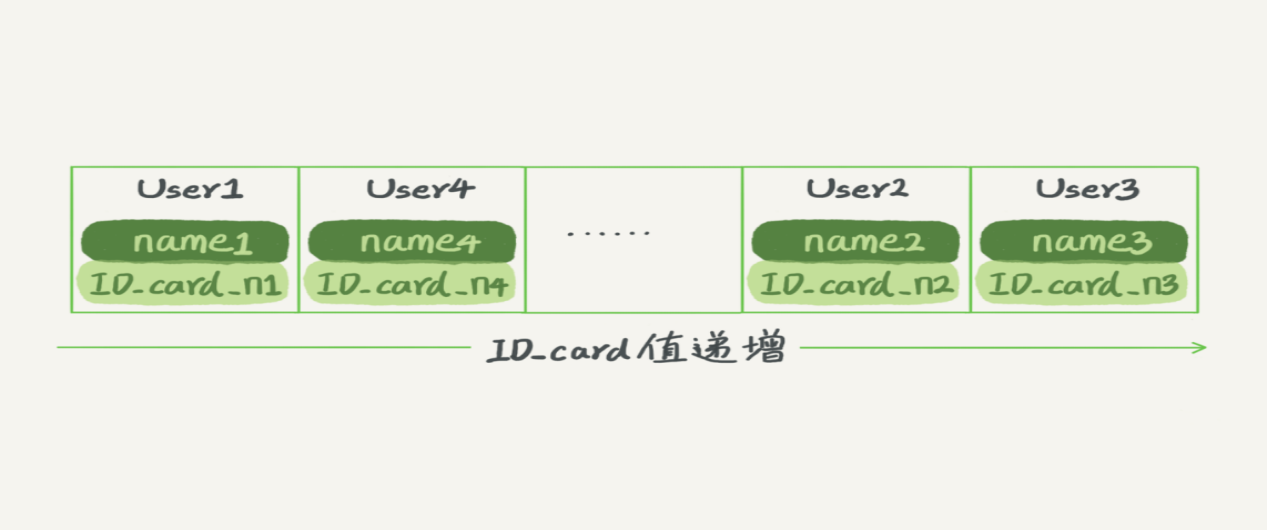


途中User2和User4身份证号算出来是N,但没关系，后面跟一个链表，假设这时要查ID\_card\_n2对应的名字是什么，处理步骤就是:首先，将ID\_card\_n2通过哈希函数算出N；然后按顺序遍历找到User2,

需要注意的是，图中四个 ID\_card\_n 的值并不是递增的，这样是新增的新的User是速度会很快，只需要往后追加，但缺点是，因为不是有序的，所以哈希索引做区间查询的速度很慢

所以，**哈希索引结构只适合用于等值查询的场景**

**有序数组在等值查询和范围查询场景中的性能就都非常优秀，**还是上面的身份证号查名字的例子，



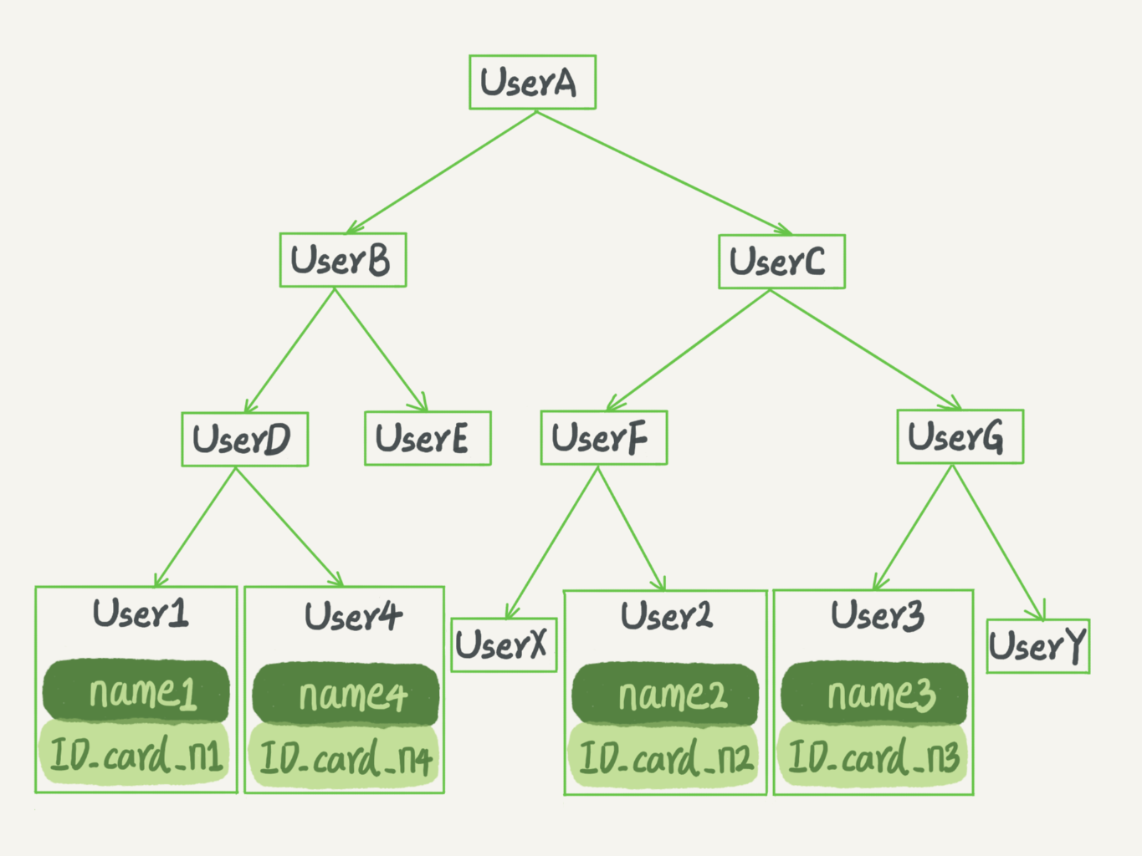
这里身份证号没有重复，这个数组就是按照身份证号递增的顺序保存，这个时候如果要查ID\_card\_n2对应的名字，用二分法就可以快速得到，这个时间复杂度是O(log(N))

如果要查询[ID\_card\_X,ID\_card\_Y]区间的值User,可以先用二分法找到ID\_card\_X（如果不存在，就找到大于ID\_card\_x的第一个user）然后向右遍历，直到查到第一个大于ID\_card\_Y的值，退出循环。

但是如果更新有序数组花费的成本就比较高，往中间插入一个值，就必须挪动后面所有的记录。

**有序数组索引值使用于静态存储引擎，**比如要存储2017年某个城市所有人口信息，这类不会修改的数据

**二叉搜索树也是经典的数据结构。**



**InnoDB的索引模型**

在InnoDB中，表都是根据主键顺序以索引的形式存放的，这种存储方式的表称为索引组织表，

**每一个索引在InnoDB里面对应一个棵B+树**

假设有个主键Id的表，表中有字段k,并且在k上有索引

Create table T(

Id int primary key,

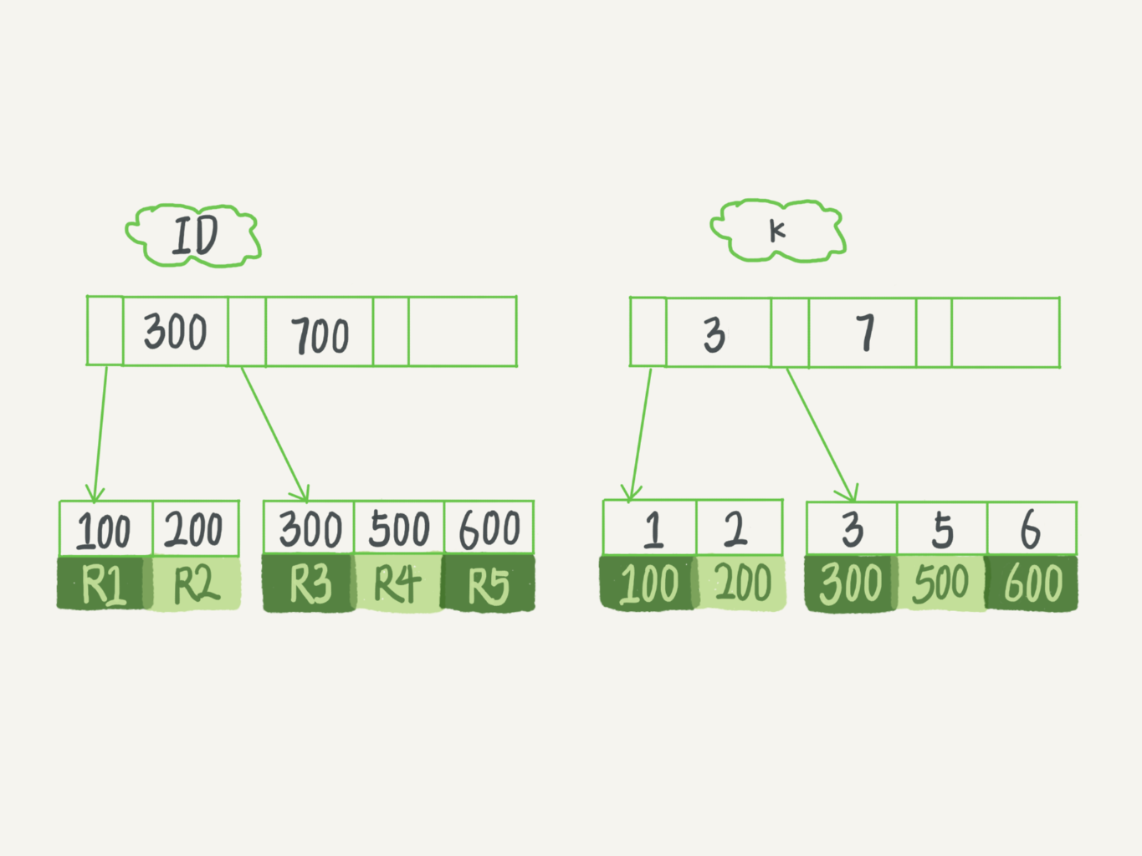
K int not null,

Name varchar(16),

Index(k)

)engine=InnoDB;

表中 R1~R5 的 (ID,k) 值分别为 (100,1)、(200,2)、(300,3)、(500,5) 和 (600,6)，两棵树的示例示意图如下。



根据叶子节点的内容，索引类型分为主键索引和非主键索引

主键索引的叶子节点存的是整行数据，在InnoDB里，主键索引也被称为**聚蔟索引**

非主键索引的叶子节点的内容是主键的值，在InnoDB里非主键索引也被称为**二级索引**

**基于主键索引和非主键索引的查询有什么区别？**

1 如果语句 select \* from T where ID=500 即主键方式查询，则只需搜索ID这棵B+树。

2 如果是 select \* from T where K= 5,即普通索引查询方式，则需要先搜索K索引树，得到ID的值为500，在到ID索引树搜索一次，这个过程称为**回表**

也就是说非主键索引的查询需要多扫描一次索引树，因此，我们在应用中应该尽量使用主键查询。

**索引的维护**

**B+**树维护了索引有序性，在插入值的时需要维护，以上面的图为例，如果插入行的ID为700，则只需要在R5的记录后面插入一个新纪录，如果插入的ID为400，就相对麻烦了，需要逻辑上挪动后面的数据，空出位置

而更糟的情况是，如果R5所在的数据页已经满了，根据B+树的算法，此时需要申请一个新的数据页，然后挪动部分数据过去，这个过程称为页分裂，在这种情况下，性能自然会受影响。

除了性能外，页分裂操作还影响数据页的利用率，原本放在一个页的数据，现在分到两个中，整体空间利用率降低50%。

当然后分裂就有合并，当相邻的两个页由于删除了数据，利用率很低之后，会将数据页做合并，合并的过程是分裂的逆过程。

讨论一个案例

可能在一些建表规范里面见到类似的描述，要求检表语句里一定要有自增主键，事无绝对，我们来分析一下那些场景应该使用自增主键，而那些场景下不应该

自增主键是值自增列上定义的主键，在建表语句中一般是这么定义的，not null primary key autoincrement

插入新纪录的时候不指定ID值，系统会自动获取当前ID最大值加1作为下一条记录的ID值。

也就是说自增的插入数据模式，正符合我们前面提到的递增插入的场景，每次插入一条新纪录，都是追加操纵，都不涉及到挪动其他记录，也不会触发叶子节点的分裂。

而有业务逻辑的字段为主键，则往往不容易保证有序插入，这样写数据成本相对较高

除了考虑性那个能外，我们可以从存储空间角度来看，假设你的表中确实有一个唯一字段，比如字符串类型的身份证号，那应该用身份证号做主键，还是自增字段做主键呢？

由于每非主键索引的叶子节点都是主键的值，如果用身份证号做主键，那么每个二级索引的叶子节点占用20个字节，而如果用整型主键，只要4个字节，如果是长整型（bigint）则是8个字节。

**显然，主键长度小，普通索引的叶子节点就越小，普通索引占用的空间也就越小**

所有，从性能和空间角度看，自增主键往往是更合理的选择。

**有没有什么场景适合业务字段直接做主键呢？比如有些业务的场景需求是这样的**

1. **只有一个索引**
2. **该索引必须是唯一索引**

**这是典型的k-v场景**

**由于没有其他的索引，所以也就不考虑其他索引的叶子节点大小的问题**

**此时我们就要优先考虑“尽量使用主键的查询”原则，直接给这个索引设置为主键，可以避免每次查询需要搜索两棵树。**

**讨论题**

对于上面的例子表T,如果要重建索引k，你的两个SQL语句，

Alter table T drop index k;

Alter table T add index(k);

如果重建主键索引

Alter table T drop primary key;

Alter table T add primary key(id);对于上面的重建索引的理解，是否有不合适的地方，为什么，更好的方法是什么？

参考答案：所以可能因为删除，或者页分裂等原因，导致数据页有空洞，重建索引的过程会创建一个新的索引，把数据按顺序插入，这样页面的利用率高，也就是缩影更加紧凑，更节省空间

重建索引k是合理的，可以达到节省空间的目的，但是重建主键索引过程不合理，不论删除主键还是创建主键，都会将整个表重建，所以执行这个两个语句的，第一个语句就是白做了，这两个语句可以用 alter table T engine=InnoDB代替

**问题**

**如果没有主键的表，有一个普通索引，怎么回表？**

没有主键的表，innodb会默认创建一个rowid做主键。

一颗100万的平衡而二叉树，树高20，一次查询可能访问20个数据块，为什么树高20就是20个数据块？

每个叶子节点就是一个块，每个块包含两个数据，块之间通过链接方式。树高20的，就要遍历20个块，因为二叉树结构，每次指针查找很大概率是触发随机读磁盘（比如很难刚好碰上一个节点和他的左右儿子节点刚好相邻）