# MySQL 索引深入剖析

数据结构演示网址： https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

## 索引是什么？

### 索引图解

维基百科上面对数据库索引的定义：

数据库索引，是数据库管理系统（DBMS）中一个排序的数据结构，以协助快速查询、更新数据库表中数据。

怎么理解？


首先我们的数据是以文件的形式存放在磁盘上面的，每一行数据都有它的磁盘地址。如果没有索引的话，我们要从 500 万行数据里面检索一条数据，只能依次遍历这张表的全部数据，直到找到这条数据。

但是有了索引之后，只需要在索引里面去检索这条数据就行了，因为它是一种特殊的专门用来快速检索的数据结构，我们找到数据存放的磁盘地址以后，就可以拿到数据了。

### 索引类型与方法

如何创建索引？


第一个是索引的名称，第二个是索引的列，比如我们是要对 id 创建索引还是对 name 创建索引。

在 InnoDB 里面，索引类型有三种：

普通（Normal）：也叫非唯一索引，是最普通的索引，没有任何的限制。

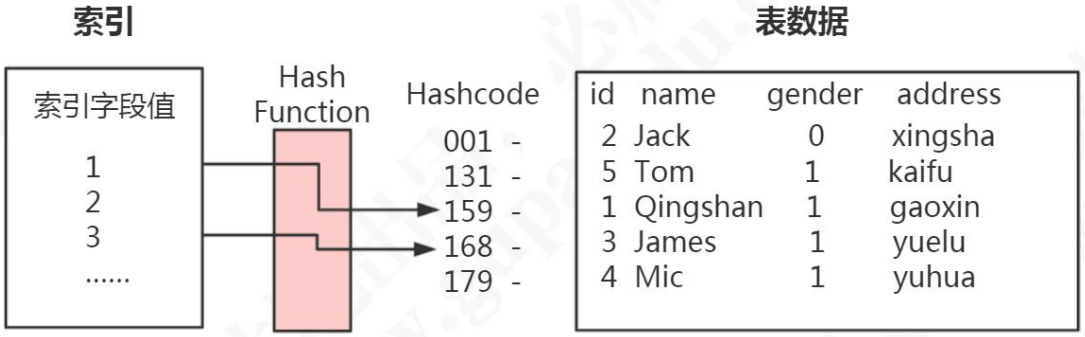
唯一（Unique）：唯一索引要求键值不能重复。另外需要注意的是，主键索引是一种特殊的唯一索引，它还多了一个限制条件，要求键值不能为空。

全文（Fulltext）：针对比较大的数据（InnoDB 在 5.6.4 以后才可以使用），主要是为了解决 like 查询效率低的问题。

索引方式：

BTREE：是 MySQL 默认的索引方式。

HASH：以 KV 的形式检索数据，它会根据索引字段生成哈希码和指针，指针指向数据。




哈希索引有什么特点呢？

第一个，它的时间复杂度是 O(1)，因为哈希索引里面的数据不是按顺序存储的，所以不能用于排序。

第二个，我们在查询数据的时候要根据键值计算哈希码，所以它只能支持等值查询，不支持范围查询。

我们说索引是一种数据结构，那么它到底应该选择一种什么数据结构，才能实现数据的高效检索呢？

## 二分查找

我们先来玩一个猜数字的游戏，给定一个 1~100 的自然数，给你 10 次机会，你能猜中这个数字吗？你会从多少开始猜？

这个就是二分查找的一种思想，也叫折半查找，每一次，我们都把候选数据缩小了一半。如果数据已经排过序的话，这种方式效率比较高。

所以第一个，我们可以考虑用**有序数组**作为索引的数据结构。

有序数组的等值查询和比较查询效率非常高，但是更新数据的时候会出现一个问题，可能要挪动大量的数据（改变 index），所以只适合存储静态的数据。

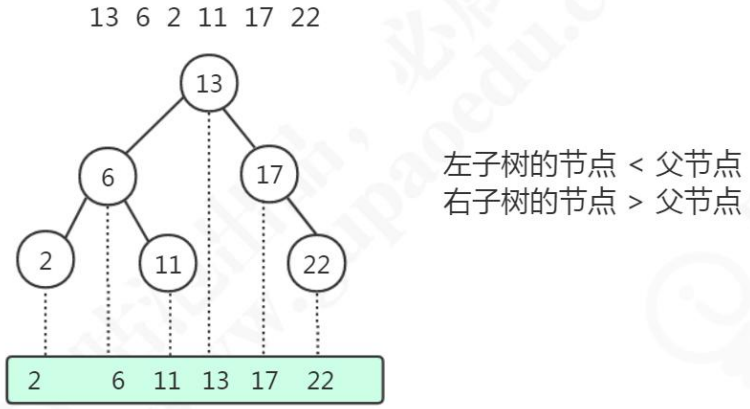
为了支持频繁的修改，我们需要采用**链表**。链表的话，如果是单链表，它的查找效率还是不够高，为了解决这个问题，BST（Binary Search Tree）也就是我们所说的二叉查找树诞生了。

# MySQL 索引模型推演

## 二叉查找树（BST Binary Search Tree）

二叉查找树的特点：

左子树所有的节点都小于父节点，右子树所有的节点都大于父节点。投影到平面以后，就是一个有序的线性表。



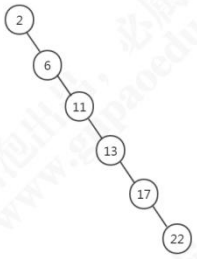

二叉查找树既能够实现快速查找，又能够实现快速插入。

但是二叉查找树有一个问题，就是它的查找耗时是和这棵树的深度相关的，在最坏的情况下时间复杂度会退化成 O(n)。

什么情况是最坏的情况呢？

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

我们依次插入 2、6、11、13、17、22。这个时候它会变成链表（“斜树”），这种情况下和顺序查找效率是没有区别的，不能达到加快检索速度的目的。




因为左右子树深度差太大——不够平衡。

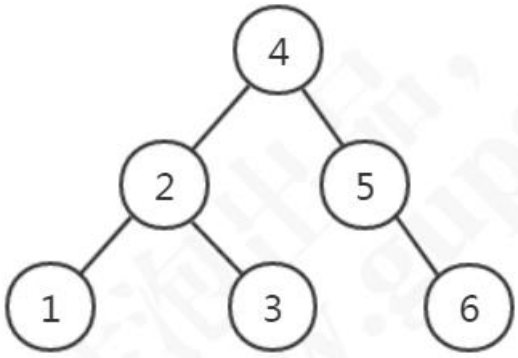
有没有更加平衡的树呢？这个就是平衡二叉树，叫做 Balanced binary search trees，或者 AVL 树。

## 平衡二叉树（AVL Tree）（左旋、右旋）

AVL Trees (Balanced binary search trees)

平衡二叉树的定义：左右子树深度差绝对值不能超过 1。比如左子树的深度是 2，右子树的深度只能是 1 或者 3。

我们按顺序插入 1、2、3、4、5、6，会变成这样：



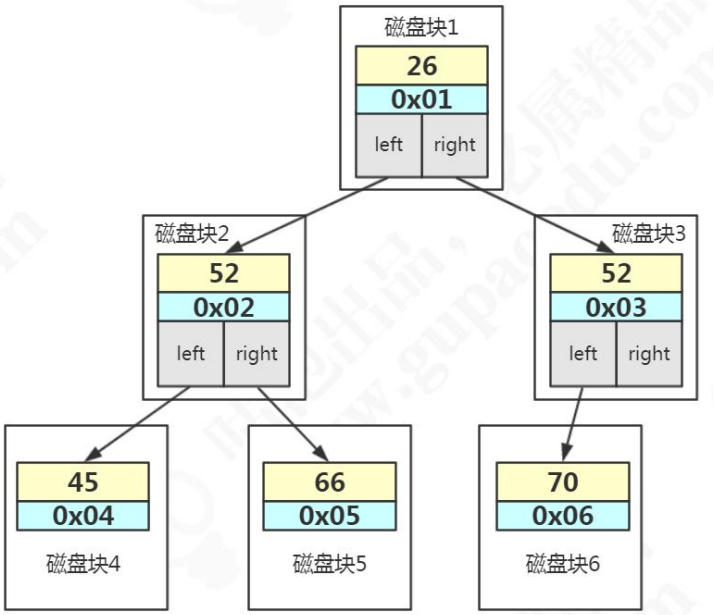

AVL 树怎么保持平衡？
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
插入 1、2、3。

当我们插入了 1、2 之后，如果按照二叉查找树的定义，3 肯定是要在 2 的右边的，这个时候根节点 1 的右节点深度会变成 2，但是左节点的深度是 0，因为它没有子节点，所以就会违反平衡二叉树的定义。

那应该怎么办呢？因为它是右节点下面接一个右节点，右-右型，所以这个时候我们要把 2 提上去，这个操作叫做**左旋**。


同样的，如果我们插入 7、6、5，这个时候会变成左左型，就会发生右旋操作，把 6 提上去。
所以为了保持平衡，AVL 树在插入和更新数据的时候执行了一系列的计算和调整的操作。

平衡的问题我们解决了，那么平衡二叉树作为索引怎么查询数据？
在平衡二叉树中，一个节点，作为索引应该存储什么内容？
它应该存储三块的内容：
第一个是索引的键值。比如我们在 id 上面创建了一个索引，我在用 where id =1 的条件查询的时候就会找到索引里面的 id 的这个键值。
第二个是数据的磁盘地址，因为索引的作用就是去查找数据的存放的地址。
第三个，因为是二叉树，它必须还要有左子节点和右子节点的引用，这样我们才能找到下一个节点。比如大于 26 的时候，走右边，到下一个树的节点，继续判断。




如果是这样存储数据的话，我们来看一下会有什么问题。

首先，我们的索引的数据，也是放在硬盘上的。操作系统的文件管理系统一次 IO，从磁盘读取的数据大小，单位是 1 页（page）（4KB = 8 个扇区*512 Bytes 字节）。

但是一个树的节点（关键字+数据区+引用）达不到 4K 的容量，我们在读取一个树的节点的时候，浪费了大量的空间。

所以如果每个节点存储的数据太少，一次检索数据跟磁盘交互次数就会过多。

比如上面这张图，我们一张表里面有 6 条数据，当我们查询 id=66 的时候，要查询两个子节点，就需要跟磁盘交互两次，如果我们有几百万的数据呢？这个时间更加难以估计。

所以我们的解决方案是什么呢？

第一个就是让每个节点存储更多的数据。

第二个，节点上的关键字的数量越多，我们的指针数也越多，也就是意味着可以有更多的分叉（我们把它叫做“路数”）。

因为分叉数越多，树的深度就会减少。

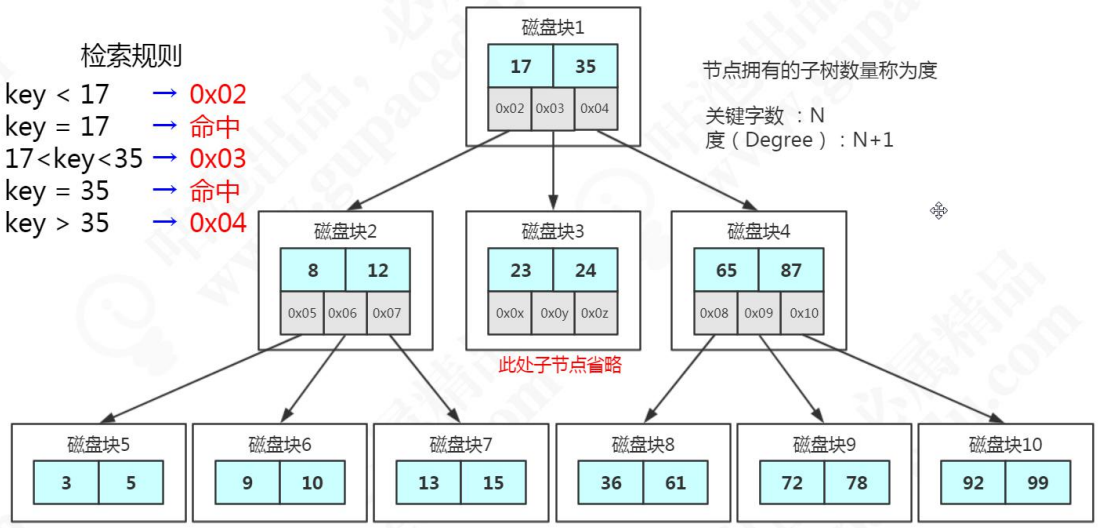
这个时候，我们的树就不再是二叉了，而是多叉，或者叫做多路。

## 多路平衡查找树（B Tree）（分裂、合并）

Balanced Tree

这个就是我们的多路平衡查找树中，叫做 B Tree（B 代表平衡）。

它有一个特点：分叉数（路数）永远比关键字数多 1。比如我们画的这棵树，每个节点存储两个关键字，那么就会有三个指针指向三个子节点。




B Tree 的查找规则是什么样的呢？

比如我们要在这张表里面查找 15。

因为 15 小于 17，走左边。

因为 15 大于 12，走右边。

在磁盘块 7 里面就找到了 15，只用了三次 IO。

这里还仅仅是 3 路的情况。InnoDB，在整形字段上建立索引，一个节点大约可以存储 1200 路。树高度为 4 的时候，可以存储的数据为 1200^3=17 亿。也就是说，一张 17 亿的表，查询数据最多需要访问三次磁盘。

这个是不是比 AVL 树效率更高呢？

那 B Tree 又是怎么实现一个节点存储多个关键字，还保持平衡的呢？跟 AVL 树有什么区别？

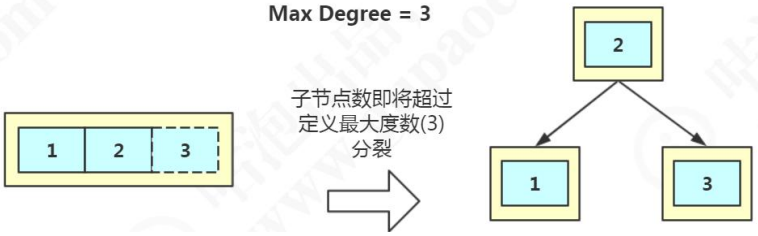
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

比如 Max Degree（路数）是 3 的时候，我们插入数据 1、2、3，在插入 3 的时候，本来应该在第一个磁盘块，但是如果一个节点有三个关键字的时候，意味着有 4 个指针，子节点会变成 4 路，所以这个时候必须进行分裂。把中间的数据 2 提上去，把 1 和 3 变成 2 的子节点。

如果删除节点，会有相反的合并的操作。

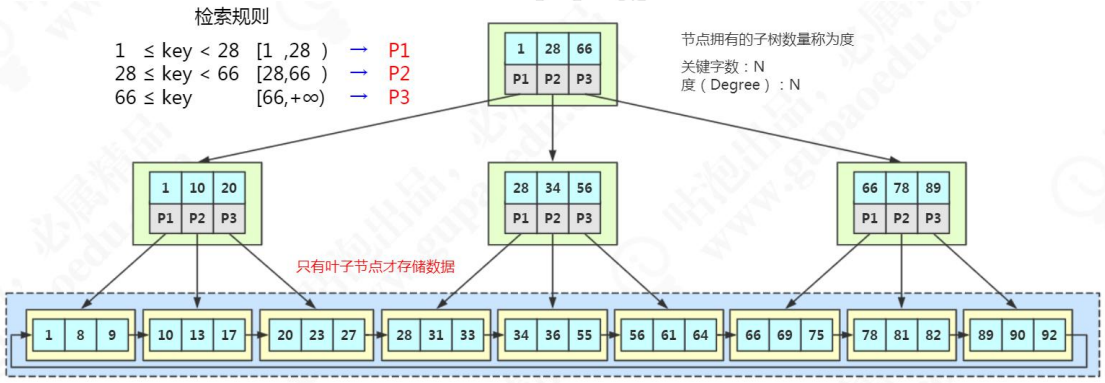
注意这里是分裂和合并，跟 AVL 树的左旋和右旋是不一样的。

我们继续插入 4 和 5，B Tree 又会出现分裂和合并的操作。



## B+树（加强版多路平衡查找树）

B Tree 的效率已经很高了，为什么 MySQL 还要对 B Tree 进行改良，最终使用了 B+Tree 呢？
总体上来说，这个 B 树的改良版本解决的问题比 B Tree 更全面。
我们来看一下 B+树的存储结构：




MySQL 中的 B+Tree 有几个特点：
1、它的关键字的数量是跟路数相等的；
2、B+Tree 的根节点和枝节点中都不会存储数据，只有叶子节点才存储数据。搜索到关键字不会直接返回，会到最后一层的叶子节点。比如我们搜索 id=28，虽然在第一层直接命中了，但是全部的数据在叶子节点上面，所以我还要继续往下搜索，一直到叶子节点。
3、B+Tree 的每个叶子节点增加了一个指向相邻叶子节点的指针，它的最后一个数据会指向下一个叶子节点的第一个数据，形成了一个有序链表的结构。
4、它是根据左闭右开的区间 [ )来检索数据。

我们来看一下 B+Tree 的数据搜寻过程：
1）比如我们要查找 28，在根节点就找到了键值，但是因为它不是页子节点，所以会继续往下搜寻，28 是[28,66)的左闭右开的区间的临界值，所以会走中间的子节点，然后继续搜索，它又是[28,34)的左闭右开的区间的临界值，所以会走左边的子节点，最后在叶子节点上找到了需要的数据。

2）第二个，如果是范围查询，比如要查询从 22 到 60 的数据，当找到 22 之后，只需要顺着节点和指针顺序遍历就可以一次性访问到所有的数据节点，这样就极大地提高了区间查询效率（不需要返回上层父节点重复遍历查找）。

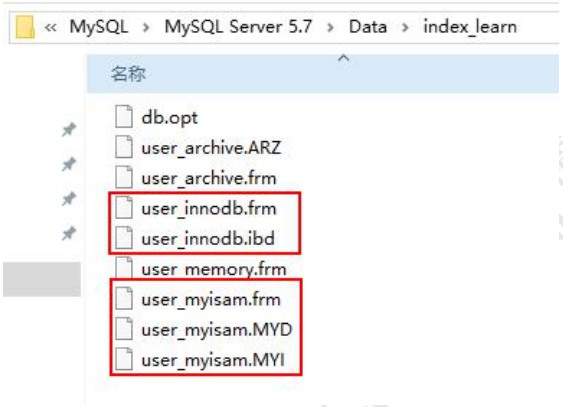
B+Tree 的特点：
1)它是 B Tree 的变种，B Tree 能解决的问题，它都能解决。
2)扫库、扫表能力更强（如果我们要对表进行全表扫描，只需要遍历叶子节点就可以了，不需要遍历整棵 B+Tree 拿到所有的数据）
3) B+Tree 的磁盘读写能力相对于 B Tree 来说更强（根节点和枝节点不保存数据区，所以一个节点可以保存更多的关键字，一次磁盘加载的关键字更多）
4)排序能力更强（因为叶子节点上有下一个数据区的指针，数据形成了链表）
5)效率更加稳定（B+Tree 永远是在叶子节点拿到数据，所以 IO 次数是稳定的）

# B+Tree 落地形式

## MySQL 架构

## MySQL 数据存储文件

InnoDB 的表有两个文件（.frm 和.ibd），MyISAM 的表有三个文件（.frm、.MYD、.MYI）。
有一个是相同的文件，.frm。 .frm 是 MySQL 里面表结构定义的文件。

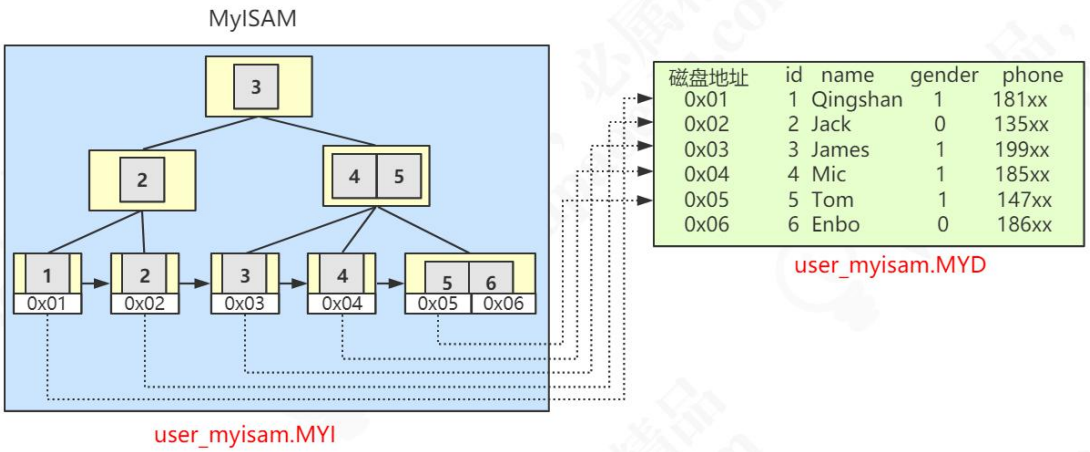



## MyISAM

在 MyISAM 里面，另外有两个文件：
一个是.MYD 文件，D 代表 Data，是 MyISAM 的数据文件，存放数据记录，比如我们的 user_myisam 表的所有的表数据。
一个是.MYI 文件，I 代表 Index，是 MyISAM 的索引文件，存放索引，比如我们在 id 字段上面创建了一个主键索引，那么主键索引就是在这个索引文件里面。

也就是说，在 MyISAM 里面，索引和数据是两个独立的文件。

MyISAM 的 B+Tree 里面，叶子节点存储的是数据文件对应的磁盘地址。所以从索引文件.MYI 中找到键值后，会到数据文件.MYD 中获取相应的数据记录。

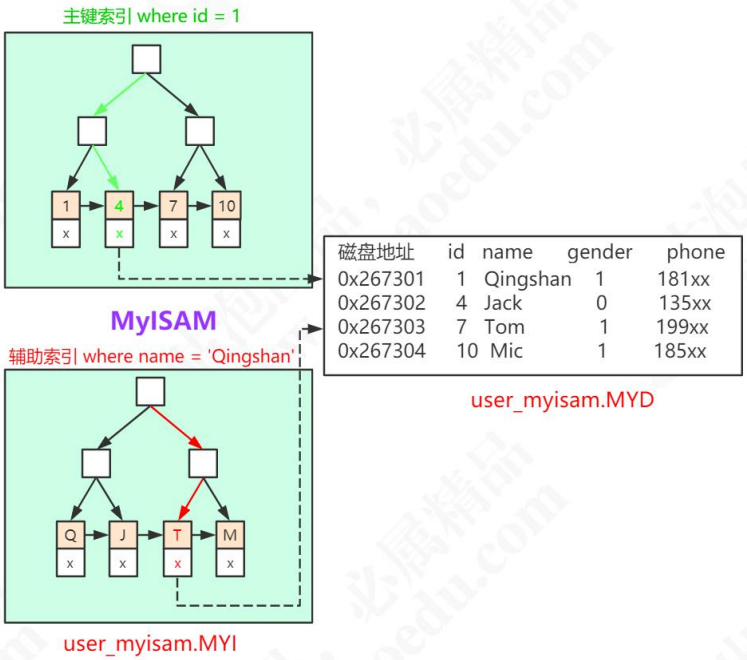



如果是辅助索引，有什么不一样呢？

ALTER TABLE user_innodb DROP INDEX index_user_name;

ALTER TABLE user_innodb ADD INDEX index_user_name (name);

在 MyISAM 里面，辅助索引也在这个.MYI 文件里面。辅助索引跟主键索引存储和检索数据的方式是没有任何区别的，一样是在索引文件里面找到磁盘地址，然后到数据文件里面获取数据。

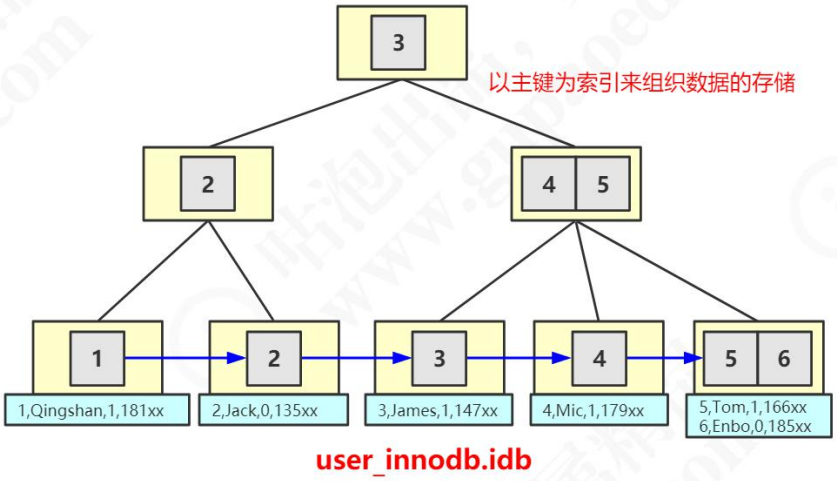



## InnoDB

InnoDB 只有一个文件（.ibd 文件）。

在 InnoDB 里面，它是以主键为索引来组织数据的存储的，所以索引文件和数据文件是同一个文件，都在.ibd 文件里面。

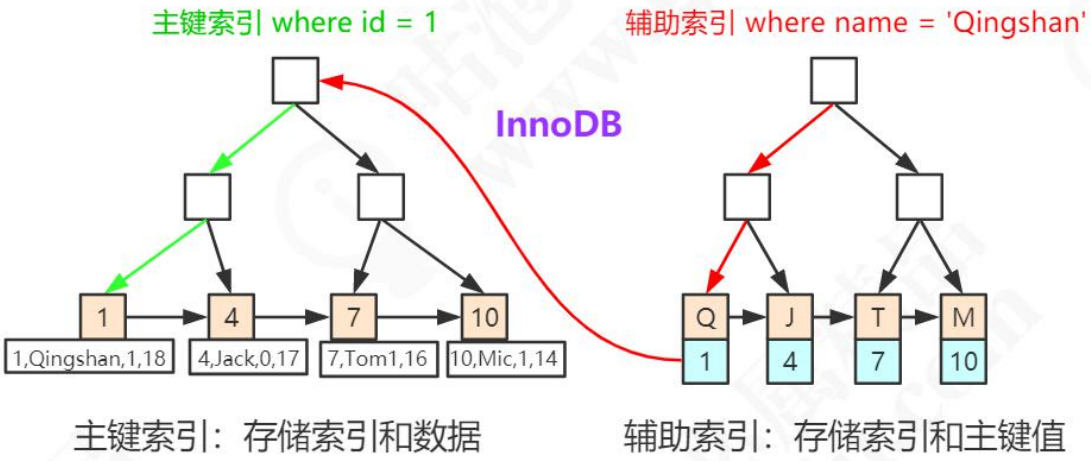
在 InnoDB 的主键索引的叶子节点上，它直接存储了我们的数据。




聚集索引就是索引中的数据物理存放地址和索引的顺序是一致的（比如字典的目录是按拼音排序的，内容也是按拼音排序的，按拼音排序的这种目录就叫聚集索引）。

所以在 InnoDB 里面，主键索引是聚集索引，非主键都是非聚集索引。

那主键之外的索引，比如我们在 name 字段上面建的普通索引，又是怎么存储和检索数据的呢？




InnoDB 中，主键索引和辅助索引是有一个主次之分的。

辅助索引存储的是辅助索引和主键值。如果使用辅助索引查询，会根据主键值在主键索引中查询，最终取得数据。

比如我们用 name 索引查询 name='青山'，它会在叶子节点找到主键值，也就是 id=1，然后再到主键索引的叶子节点拿到数据。

# 索引的几大原则

## 列的离散（sàn）度

第一个叫做列的离散度，我们先来看一下列的离散度的公式：

count(distinct(column_name)) : count(*)，列的全部不同值和所有数据行的比例。数据行数相同的情况

下，分子越大，列的离散度就越高。

| id | name | gender | phone |
|---|---|---|---|
| 1 | 青山 | 0 | 13666666666 |
| 2 | 岑粒 | 0 | 13800722654 |
| 3 | 章机轻 | 1 | 16607463532 |
| 4 | 于琴旺 | 0 | 15205286470 |
| 5 | 皮明业 | 1 | 15901557881 |
| 6 | 伊颚 | 0 | 15003812562 |
| 7 | 宁慨 | 1 | 19900275915 |
| 8 | 明鹕 | 0 | 15104237001 |
| 9 | 蒋椴 | 0 | 13206846562 |
| 10 | 魏仿巡 | 1 | 15104584161 |

简单来说，如果列的重复值越多，离散度就越低，重复值越少，离散度就越高。

当我们用在 gender 上建立的索引去检索数据的时候，由于重复值太多，需要扫描的行数就更多。例如，我们现在在 gender 列上面创建一个索引，然后看一下执行计划。

```
ALTER TABLE user_innodb DROP INDEX idx_user_gender;
ALTER TABLE user_innodb ADD INDEX idx_user_gender (gender);
EXPLAIN SELECT * FROM `user_innodb` WHERE gender = 0;
```

这里需要扫描的行是 2462168 行（rows）。

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | student_innodb | (Null) | ref | index_gender | index_gender | 2 | const | 2488028 | 100 | (Null) |

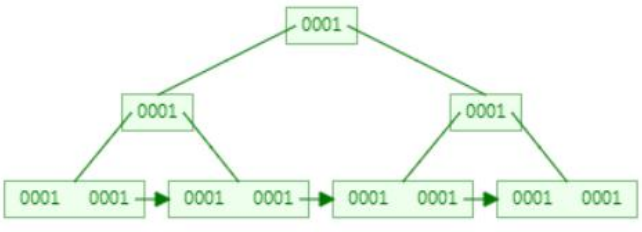
而 name 的离散度更高，比如“青山”的这名字，只需要扫描一行（rows）。

```
ALTER TABLE user_innodb DROP INDEX idx_user_name;
ALTER TABLE user_innodb ADD INDEX idx_ user_name (name);
EXPLAIN SELECT * FROM `user_innodb` WHERE name = '青山';
```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | user_inn | (Null) | ref | idx_user_name | idx_us | 1023 | const | 1 | 100 | (Null) |

如果在 B+Tree 里面的重复值太多，MySQL 的优化器发现走索引跟使用全表扫描差不了多少的时候，就算建了索引，也不一定会走索引。

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

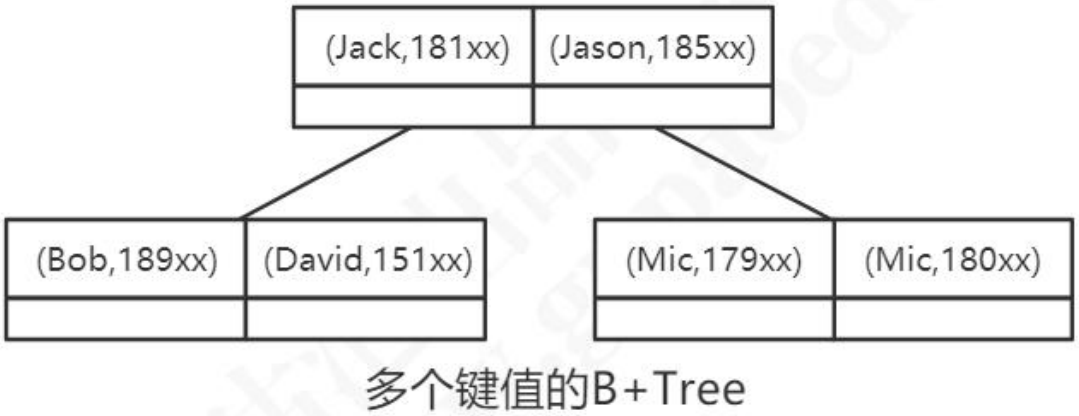



## 联合索引最左匹配

有的时候我们的多条件查询的时候，也会建立联合索引。单列索引可以看成是特殊的联合索引。

比如我们在 user 表上面，给 name 和 phone 建立了一个联合索引。

```
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX `comidx_name_phone` (`name`,`phone`);
```




联合索引在 B+Tree 中是复合的数据结构，它是按照从左到右的顺序来建立搜索树的（name 在左边，phone 在右边）。

从这张图可以看出来，第一个字段 name 是有序的，第二个字段 phone 是无序的。当 name 相等的时候，phone 才是有序的。

这个时候我们使用 where name=’青山’ and phone = ‘136xx’去查询数据的时候，B+Tree 会优先比较 name 来确定下一步应该搜索的方向，往左还是往右。如果 name 相同的时候再比较 phone。但是如果查询条件没有 name，就不知道第一步应该查哪个节点，因为建立搜索树的时候 name 是第一个比较因子，所以用不到索引。

## 什么时候用到联合索引

所以，我们在建立联合索引的时候，一定要把最常用的列放在最左边。

比如下面的三条语句，大家觉得用到联合索引了吗？

1）使用两个字段，用到联合索引：

```
EXPLAIN SELECT * FROM user_innodb WHERE name= '青山' AND phone = '15204661800'
```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | user_innodb | (Null) | ref | comixd_name_phone | comixd_name_phone | 1070 | const,const | 1 | 100 | (Null) |

2）使用左边的 name 字段，用到联合索引：

```
EXPLAIN SELECT * FROM user_innodb WHERE name= '青山'
```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | user_innodb | (Null) | ref | comixd_name_phone | comixd_name_phone | 1023 | const | 1 | 100 | (Null) |

3）使用右边的 phone 字段，无法使用索引，全表扫描：

```
EXPLAIN SELECT * FROM user_innodb WHERE phone = '15204661800'
```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | user_innodb | (Null) | ALL | (Null) | (Null) | (Null) | (Null) | 4986286 | 10 | Using where |

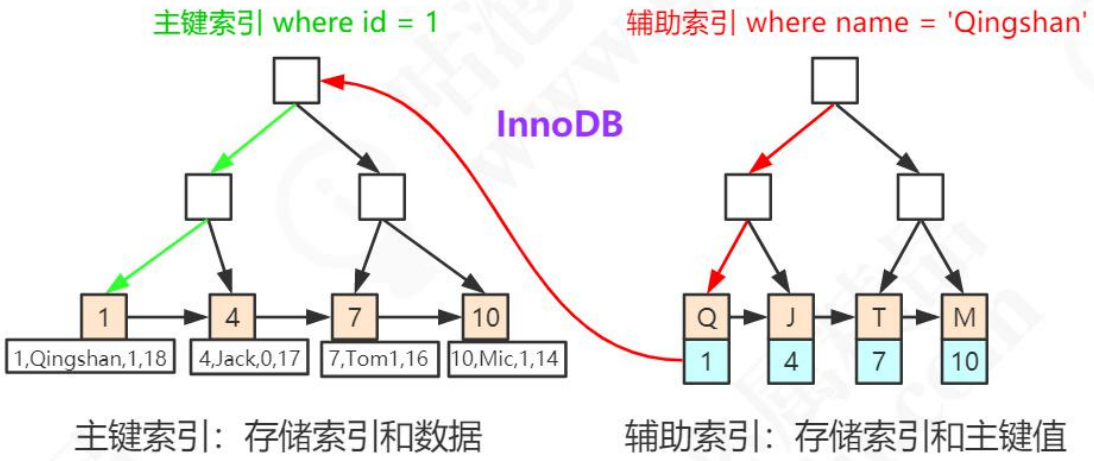
## 如何创建联合索引

创建联合索引 index(name, phone)的时候，相当于建立了两个联合索引(name),(name,phone)。
如果我们创建三个字段的索引 index(a,b,c)，相当于创建三个索引：
index(a)
index(a,b)
index(a,b,c)
用 b=? 和 b=? and c=?是不能使用到索引的。

这里就是 MySQL 里面联合索引的最左匹配原则。

## 覆盖索引

回表：
非主键索引，我们先通过索引找到主键索引的键值，再通过主键值查出索引里面没有的数据，它比基于主键索引的查询多扫描了一棵索引树，这个过程就叫回表。
例如：select * from user_innodb where name = '青山';




但是在辅助索引里面，不管是单列索引还是联合索引，如果 select 的数据列只用从索引中就能够取得，不必从数据区中读取，这时候使用的索引就叫做覆盖索引，这样就避免了回表。
我们先来创建一个联合索引：

```
## 创建联合索引
ALTER TABLE user_innodb DROP INDEX comixd_name_phone;
ALTER TABLE user_innodb add INDEX `comixd_name_phone` (`name`,`phone`);
```

这三个查询语句都用到了覆盖索引：

```
EXPLAIN SELECT name,phone FROM user_innodb WHERE name= '青山' AND phone = '13666666666';
EXPLAIN SELECT name FROM user_innodb WHERE name= '青山' AND phone = '13666666666';
EXPLAIN SELECT phone FROM user_innodb WHERE name= '青山' AND phone = '
```

```
13666666666';
```

如果 select 列再多一个 gender 字段，因为索引中没有这个字段，就无法覆盖了。

Extra 里面值为“Using index”代表使用了覆盖索引。

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | user_innodb | (Null) | ref | comixd_name_phone | comixd_name_phone | 1070 | const,const | 1 | 100 | Using index |

因为覆盖索引减少了 IO 次数，减少了数据的访问量，可以大大地提升查询效率。

当我们只用 phone 作为条件查询，因为违反了最左匹配原则，理论上用不到联合索引。但是覆盖索引跟索引的创建顺序无关。用后面的字段查询也能实现覆盖（Using index），不过要扫描 498 万行，效率也不高。

EXPLAIN SELECT name,phone FROM user_innodb WHERE phone = '13666666666';

| select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|
| SIMPLE | user_innodb | (Null) | index | (Null) | comidx_name_phone | 1070 | (Null) | 4986135 | 10 | Using where; Using index |

作者：咕泡学院-青山
QQ：3598158336
最后更新时间：2019 年 8 月 29 日 14:53:47