

MySQL 索引深入剖析——青山

1. 索引是什么？

1.1. 索引是什么

1.1.1. 索引图解

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。

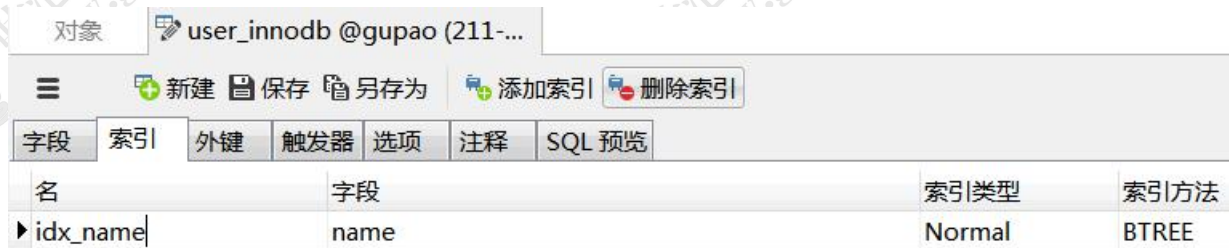


数据是以文件的形式存放在磁盘上面的，每一行数据都有它的磁盘地址。如果没有索引的话，我们要从 500 万行数据里面检索一条数据，只能依次遍历这张表的全部数据（循环调用存储引擎的读取下一行数据的接口），直到找到这条数据。

但是我们有了索引之后，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。

1.1.2. 索引类型

创建索引：



索引类型：

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。主键索引用 primary key 创建。

全文 (Fulltext)：针对比较大的数据，比如我们存放的是消息内容，有几 KB 的数据的这种情况，如果要解决 like 查询效率低的问题，可以创建全文索引。只有文本类型的字段才可以创建全文索引，比如 char、varchar、text。

```
create table m3 (
  name varchar(50),
  fulltext index(name)
);
```

```
select * from fulltext_test where match(content) against('咕泡学院' IN NATURAL LANGUAGE MODE);
```

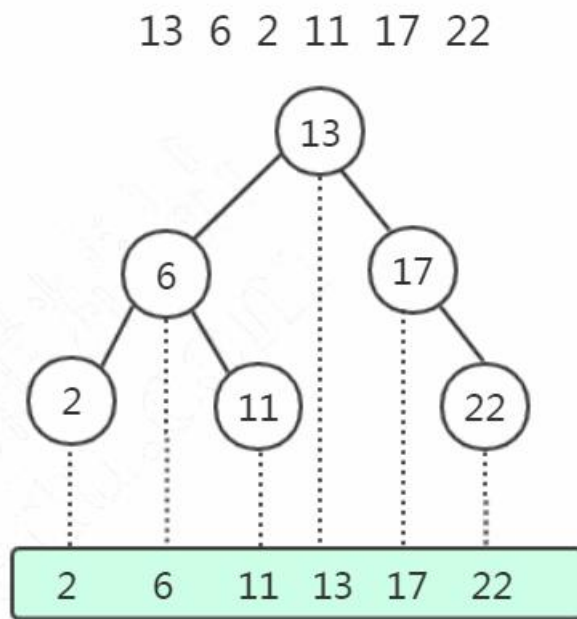
MyISAM 和 InnoDB 支持全文索引。

2. 索引存储模型推演

1. 2. 二叉查找树 (BST Binary Search Tree)

二叉查找树的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。



左子树的节点 < 父节点
右子树的节点 > 父节点

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 $O(n)$ 。

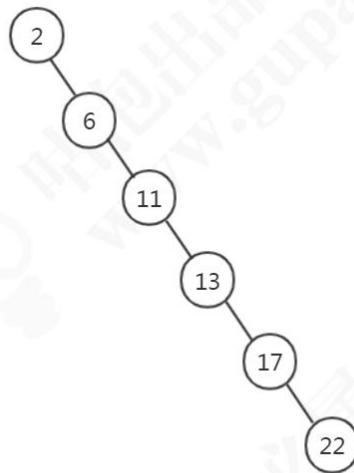
什么情况是最坏的情况呢？

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

如果我们插入的数据刚好是有序的，2、6、11、13、17、22。

它会变成链表（我们把这种树叫做“斜树”），这种情况下不能达到加快检索速度

的目的，和顺序查找效率是没有区别的。



造成它倾斜的原因是什么呢？

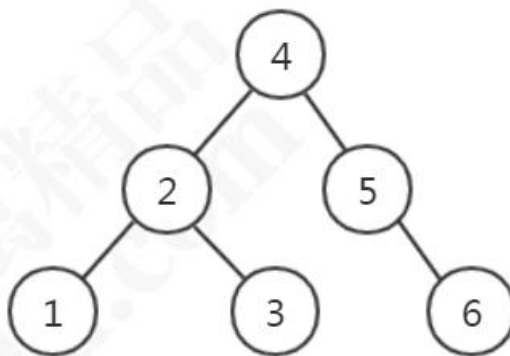
因为左右子树深度差太大，这棵树的左子树根本没有节点——也就是它不够平衡。

1.3. 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。

按顺序插入 1、2、3、4、5、6，一定是这样：



平衡是怎么做到的呢？怎么保证左右子树的深度差不能超过 1 呢？

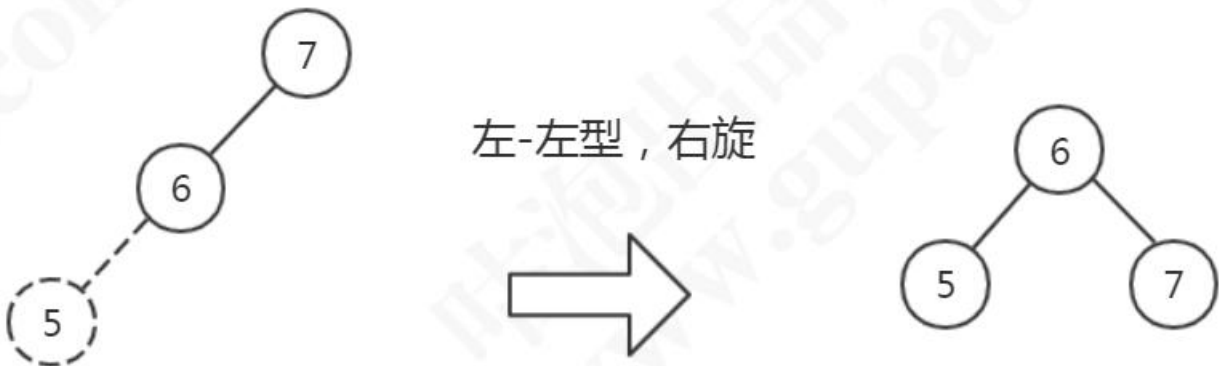
<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

插入 1、2、3。

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？

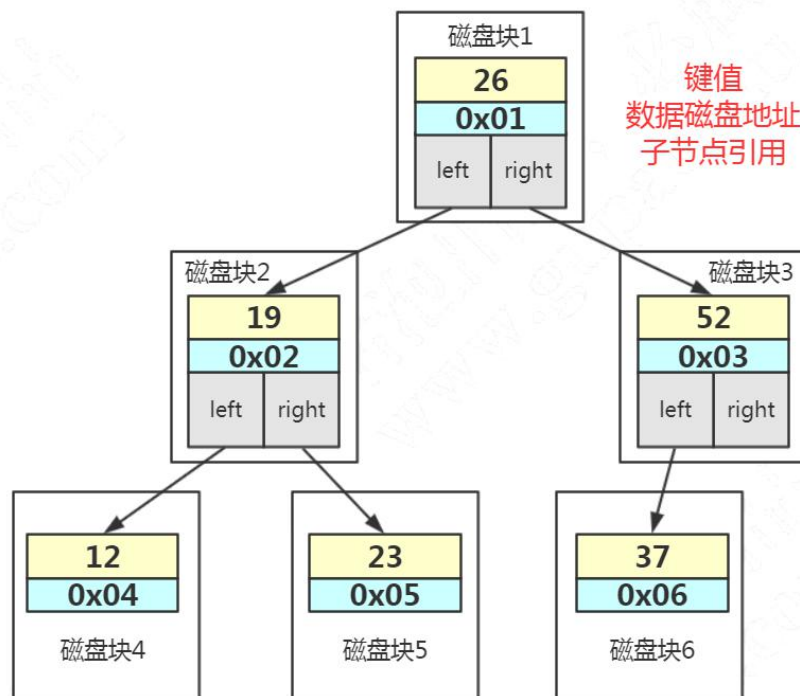
在平衡二叉树中，一个节点，它的大小是一个固定的单位，作为索引应该存储什么内容？

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，索引的数据，是放在硬盘上的。查看数据和索引的大小：

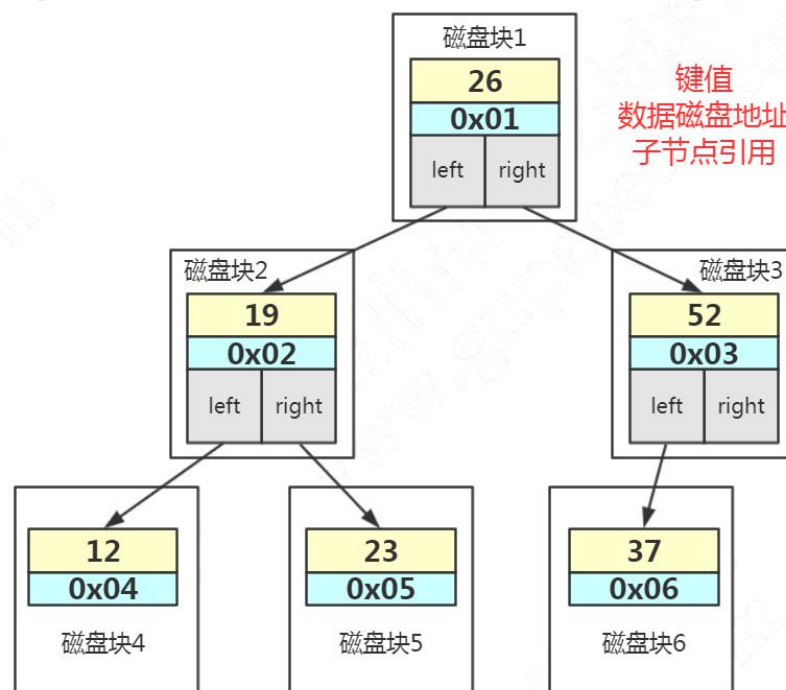
当我们用树的结构来存储索引的时候，因为拿到一块数据就要在 Server 层比较是不是需要的数据，如果不是的话就要再读一次磁盘。访问一个节点就要跟磁盘之间发生一次 IO。InnoDB 操作磁盘的最小的单位是一页（或者叫一个磁盘块），大小是 16K(16384 字节)。

那么，一个树的节点就是 16K 的大小。

如果我们一个节点只存一个键值+数据+引用，例如整形的字段，可能只用了十几个或者几十个字节，它远远达不到 16K 的容量，所以访问一个树节点，进行一次 IO 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，从索引中找到我们需要的数据，就要访问更多的节点，意味着跟磁盘交互次数就会过多。

每次从磁盘读取数据需要寻址时间，交互次数越多，消耗的时间就越多。



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 3 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（根节点是 0）。

这样，我们的树是不是从原来的高瘦高瘦的样子，变成了矮胖矮胖的样子？

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。

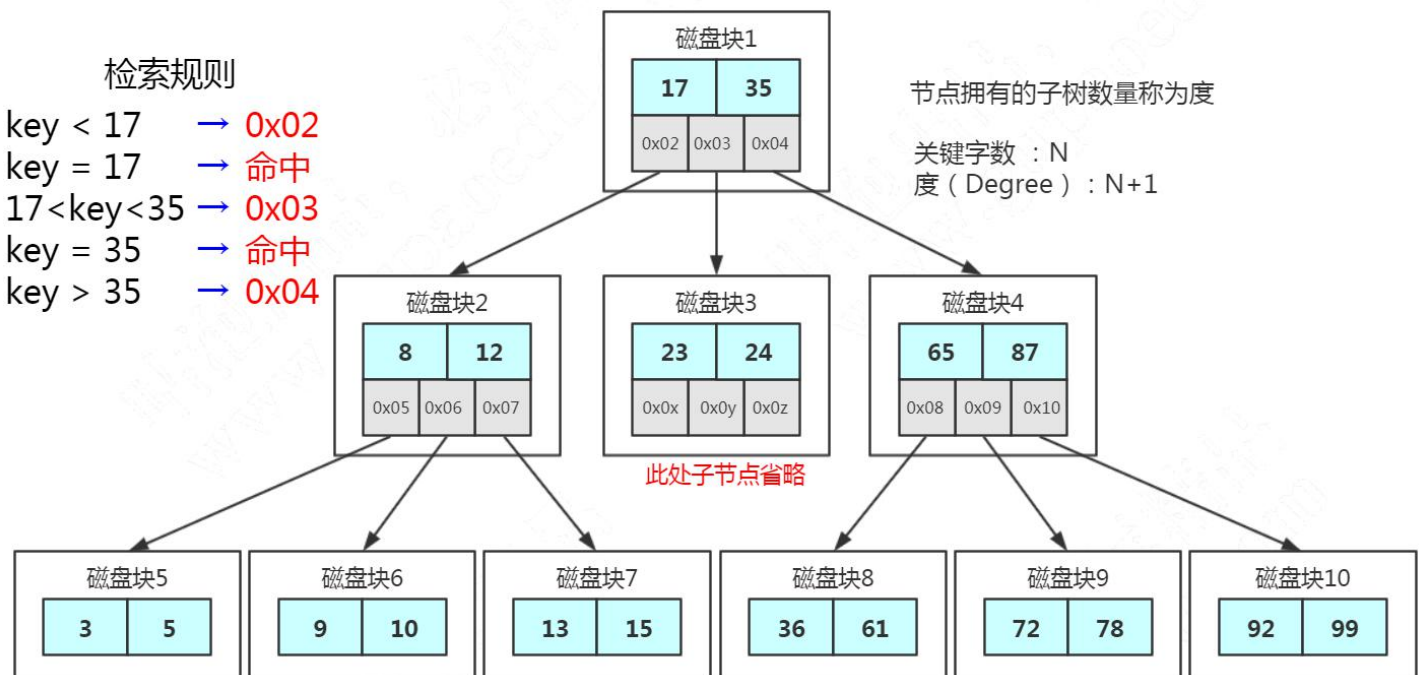
1.4. 多路平衡查找树（B Tree）（分裂、合并）

Balanced Tree

这个就是我们的多路平衡查找树，叫做 B Tree（B 代表平衡）。

跟 AVL 树一样，B 树在枝节点和叶子节点存储键值、数据地址、节点引用。

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 3 次 IO。

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？

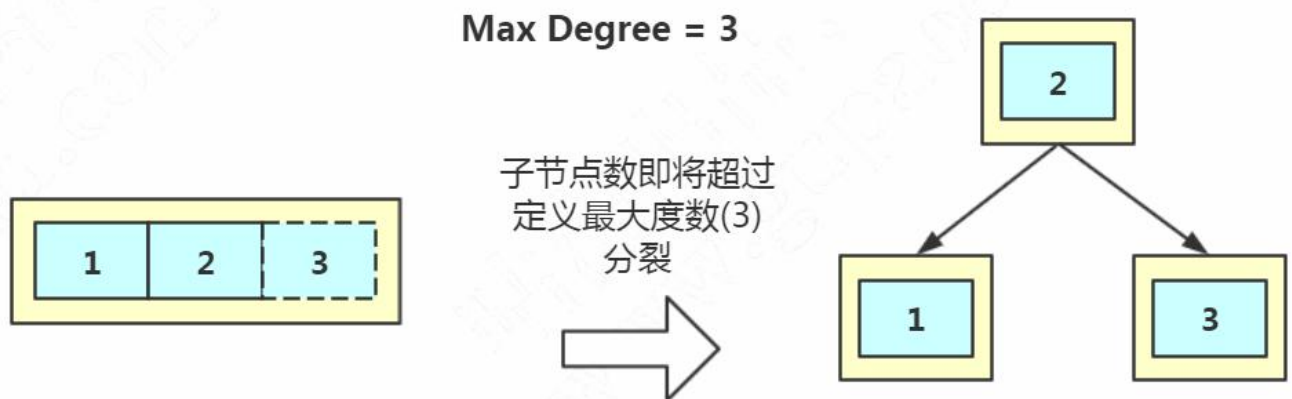
<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

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（其实就是 B+Tree）。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。



节点的分裂和合并，其实就是 InnoDB 页（page）的分裂和合并。

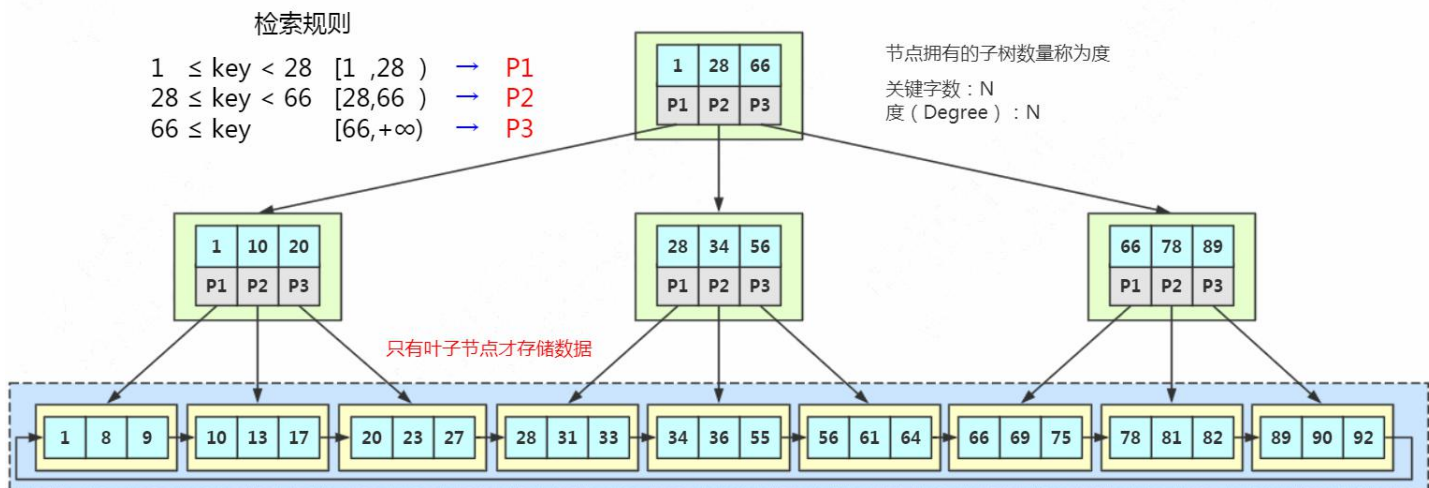
1.5. B+树（加强版多路平衡查找树）

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 InnoDB 里面的 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。

目前的认知：我们这要存放的数据是什么？是不是真实数据的地址？

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，但是数据地址在叶子节点上面，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。

InnoDB 中的 B+Tree 这种特点带来的优势：

- 1)它是 B Tree 的变种，B Tree 能解决的问题，它都能解决。B Tree 解决的两大问题

是什么？（每个节点存储更多关键字；路数更多）

2) 扫库、扫表能力更强（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）

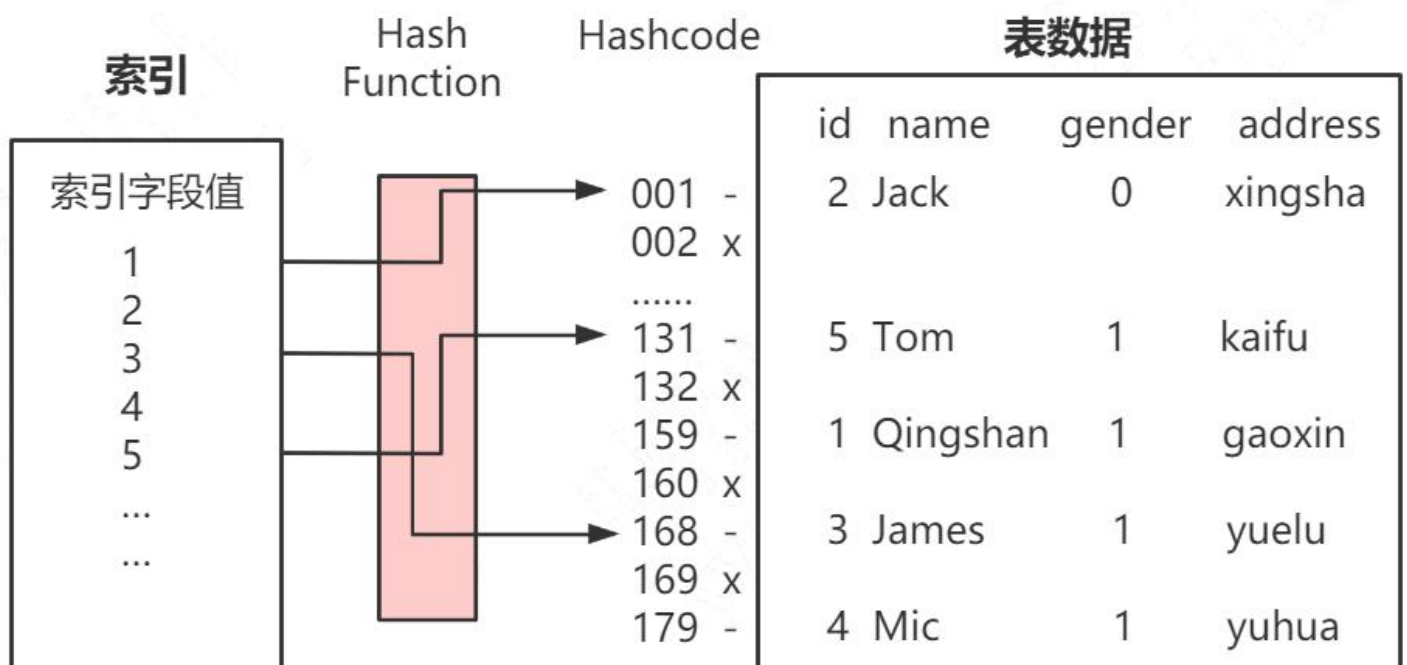
4) 排序能力更强（因为叶子节点上有下一个数据区的指针，数据形成了链表）

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）

1.6. 索引方式：真的是用的 B+Tree 吗？

在 Navicat 的工具中，创建索引，索引方式有两种。

HASH：以 KV 的形式检索数据，也就是说，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 $O(1)$ ，查询速度比较快。但是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（= IN），不支持范围查询（> < >= <= between and）。

第三：如果字段重复值很多的时候，会出现大量的哈希冲突（采用拉链法解决），效率会降低。

需要注意的是，InnoDB，不能显式地创建一个哈希索引（所谓的支持哈希索引指的是 AHI）。

memory 存储引擎可以使用 Hash 索引。

```
CREATE TABLE `user_memory` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `gender` tinyint(1) DEFAULT NULL,
  `phone` varchar(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_name` (`name`) USING HASH
) ENGINE=MEMORY AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;
```

因为 B Tree 和 B+Tree 的特性，它们广泛地用在文件系统和数据库中，例如 Windows 的 HPFS 文件系统，Oracle、MySQL、SQLServer 数据库。

3. B+Tree 落地形式

2. 1. MySQL 数据存储文件

上一节课我们知道了不同的存储引擎文件不一样。

```
show VARIABLES LIKE 'datadir';
```

每张 InnoDB 的表有两个文件（.frm 和 .ibd），MyISAM 的表有三个文件

(.frm、.MYD、.MYI)。

```
user_innodb.frm  
user_innodb.ibd  
user_memory.frm  
user_myisam.frm  
user_myisam.MYD  
user_myisam.MYI
```

3.2.1. MyISAM

在 MyISAM 里面，另外有两个文件：

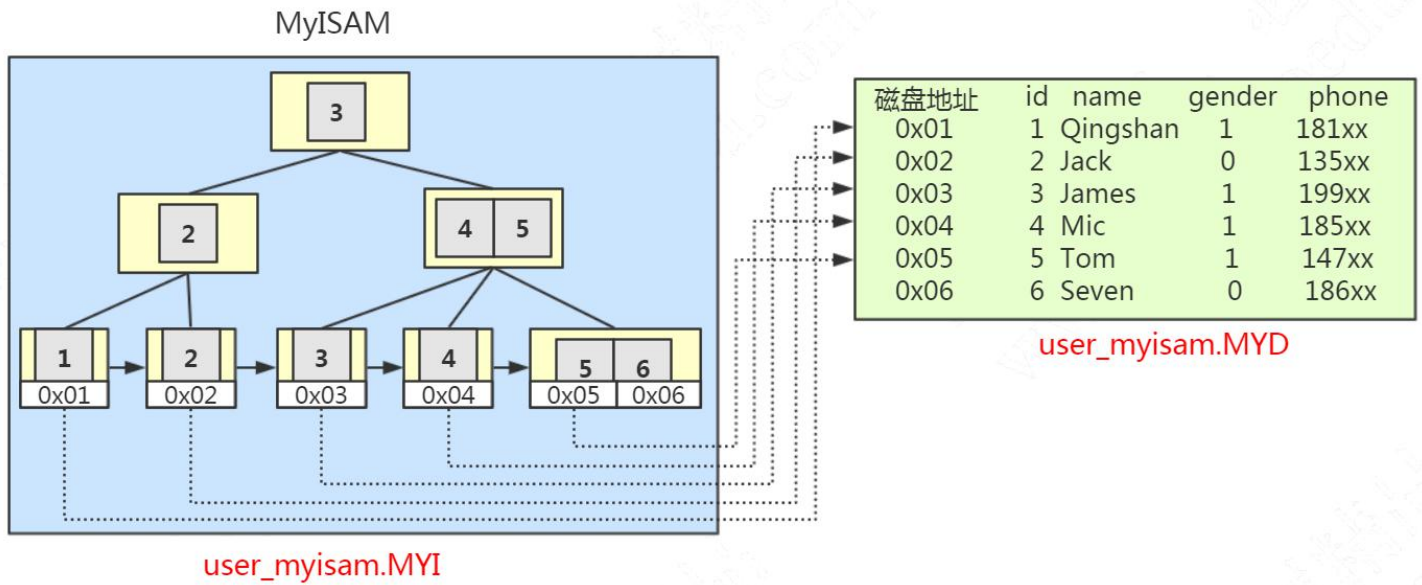
一个是.MYD 文件，D 代表 Data，是 MyISAM 的数据文件，存放数据记录，比如我们的 user_myisam 表的所有的表数据。

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。

那我们怎么根据索引找到数据呢？

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。

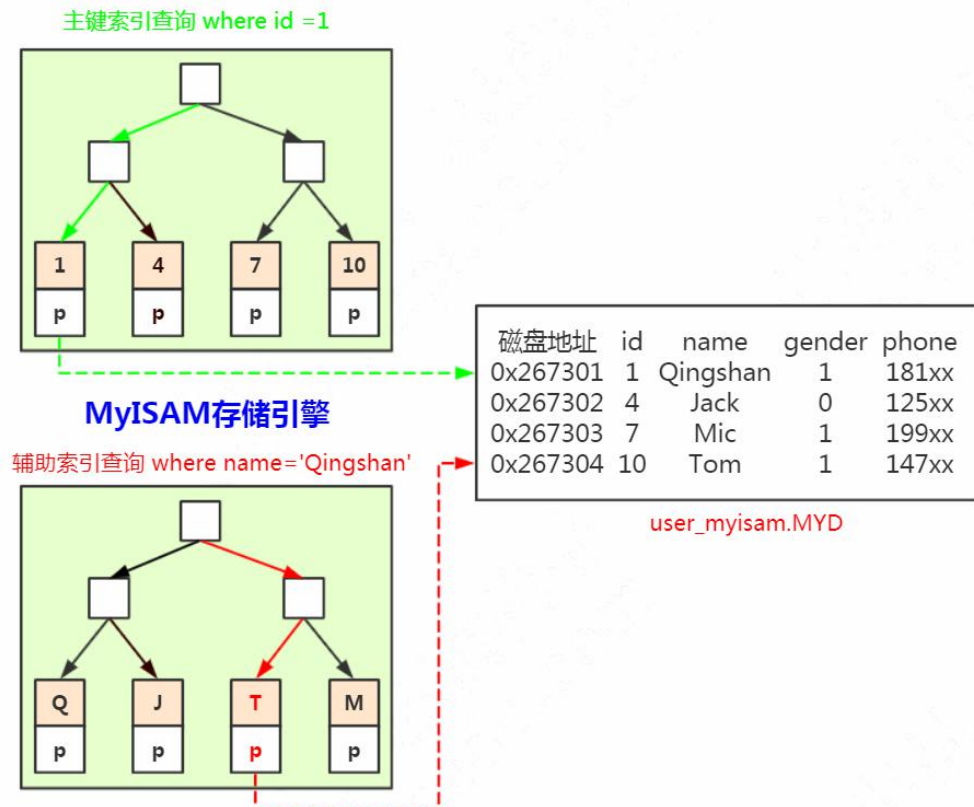


如果是辅助索引，有什么不一样呢？

```
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。

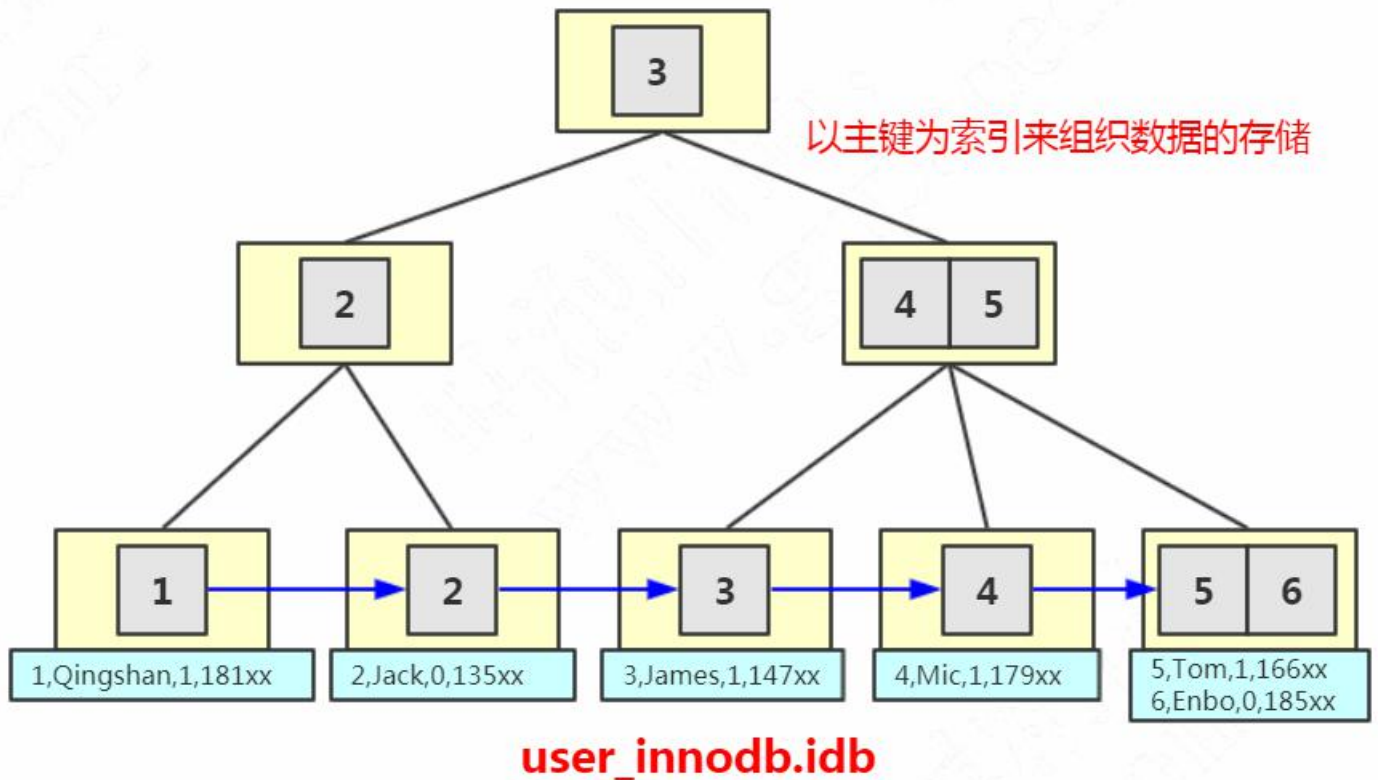


这个就是 MyISAM 里面的索引落地的形式。但是在我们的 InnoDB 里面是不一样的。我们来看一下。

3. 2. 2. InnoDB

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。

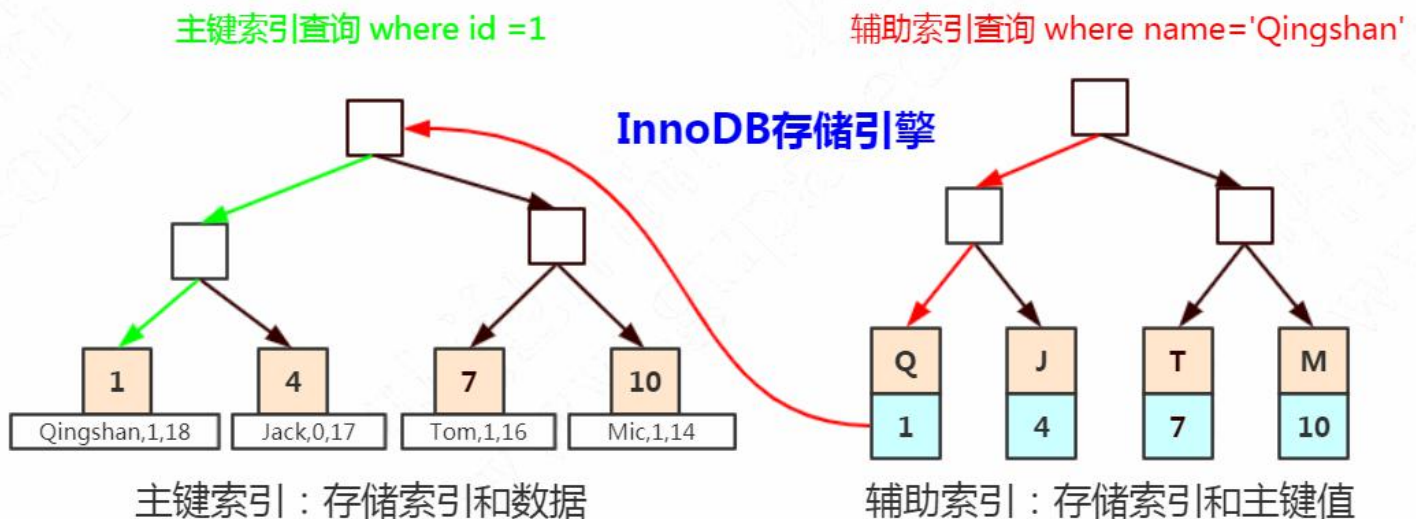
在 InnoDB 的主键索引的叶子节点上，它直接存储了我们的数据。



聚集索引（聚簇索引）：就是索引键值的逻辑顺序跟表数据行的物理存储顺序是一致的。（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）。

在 InnoDB 里面，它组织数据的方式叫做叫做（聚集）索引组织表（clustered index organize table），所以主键索引是聚集索引，非主键都是非聚集索引。

主键之外的索引，比如在 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。

如果一张表没有主键怎么办？

<https://dev.mysql.com/doc/refman/5.7/en/innodb-index-types.html>

- 1、如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引。
- 2、如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引作为主键索引。
- 3、如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐藏的聚集索引，它会随着行记录的写入而主键递增。

```
select _rowid name from t2;
```

4. 索引使用原则

3.1. 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：

$\text{count}(\text{distinct}(\text{column_name})) : \text{count}(*)$ ，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	13101880079
2	郑摘	1	15501862216
3	王致葱	0	18504734367
4	秦柜羴	1	15106797784
5	王涩鑫	0	15000770789
6	王錡	0	15900528227
7	朱瘰	1	13806617196
8	陈怀密	0	13707077795
9	冯混钩	0	15604604290
10	蒋迥	0	13702963295

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。

这里需要扫描的行是 X 行。

```
ALTER TABLE user_innodb DROP INDEX idx_user_gender;
ALTER TABLE user_innodb ADD INDEX idx_user_gender (gender); -- 耗时比较久
EXPLAIN SELECT * FROM `user_innodb` WHERE gender = 0;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	user_innodb	(Null)	ref	idx_user_gender	idx_user_gender	2	const	498385

```
show indexes from user_innodb;
```

而 name 的离散度更高，比如“青山”的这名字，只需要扫描一行。

```
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
1	SIMPLE	user_innodb	(Null)	ref	idx_user_name	idx_user_name	1023	const	1

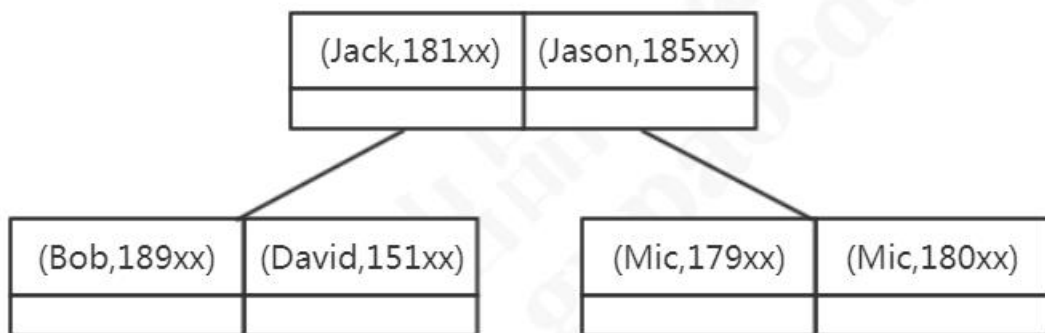
3.2. 联合索引最左匹配

前面我们说的都是针对单列创建的索引，但有的时候我们的多条件查询的时候，也会建立联合索引，举例：查询成绩的时候必须同时输入身份证和考号。

单列索引可以看成是特殊的联合索引。

比如 user 表给 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);
```



多个键值的B+Tree

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，name 是有序的，phone 是无序的。当 name 相等的时候，

phone 才是有序的。

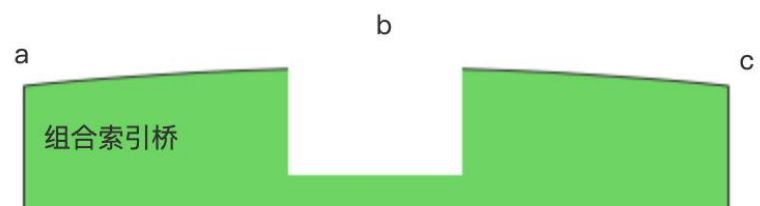
这个时候我们使用 `where name= '青山' and phone = '136xx'` 去查询数据的时候，B+Tree 会优先比较 name 来确定下一步应该搜索的方向，往左还是往右。如果 name 相同的时候再比较 phone。但是如果查询条件没有 name，就不知道第一步应该查哪个节点，因为建立搜索树的时候 name 是第一个比较因子，所以用不到索引。

4.2.1. 什么时候用到联合索引

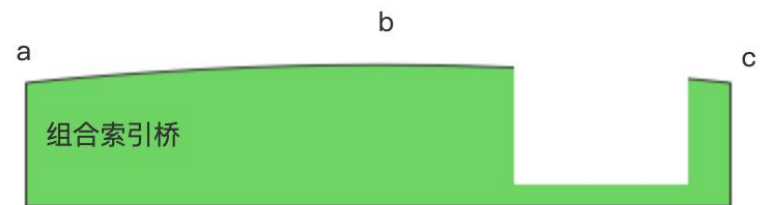
- where a and c and b: 索引字段全部命中



- where a and c: 索引命中a



- where a and b: 索引命中a、b



- where b and c: 索引未命中



- where a and b order by c: 索引字段全部命中



1) 使用两个字段，用到联合索引：

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

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	user_innodb	(Null)	ref	comidx_name_phone	comidx_name_phone	1070

2) 使用左边的 name 字段，用到联合索引：

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	user_innodb	(Null)	ref	comidx_name_phone	comidx_name_phone	1023	const

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
1	SIMPLE	user_innodb	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	996770

4.2.2. 如何创建联合索引

```
CREATE INDEX idx_name ON user_innodb(name);
CREATE INDEX idx_name_phone ON user_innodb(name,phone);
```

当我们创建一个联合索引的时候，按照最左匹配原则，用左边的字段 name 去查询的时候，也能用到索引，所以第一个索引完全没必要。

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)

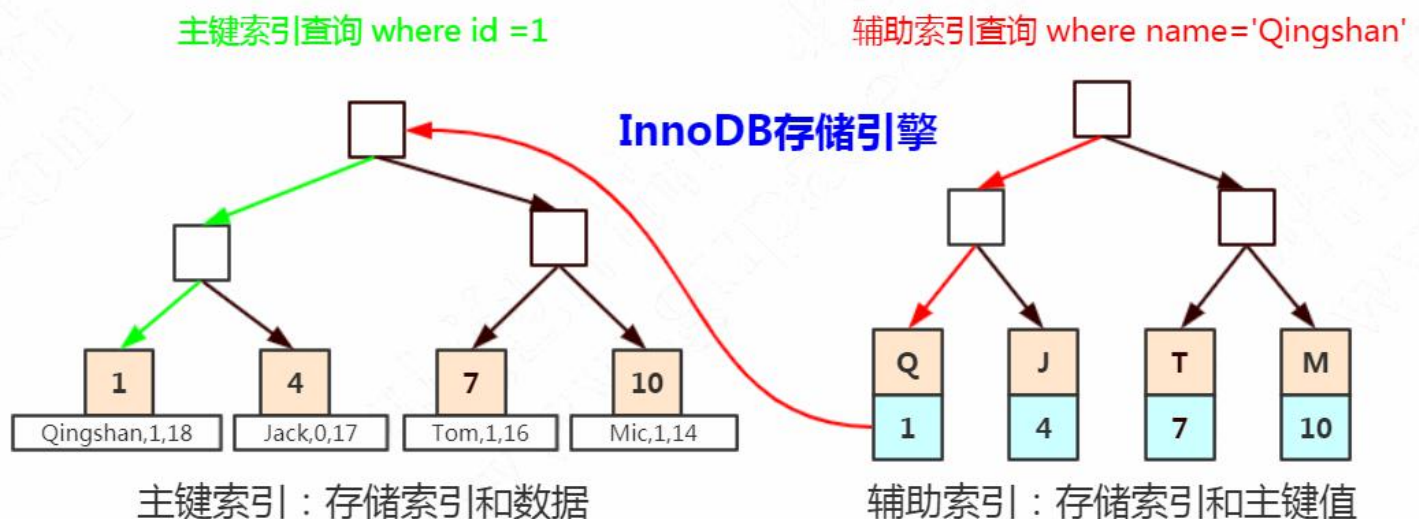
用 where b=? 和 where b=? and c=? 是不能使用到索引的。

3.3. 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';



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';
```

select *，此处用不到覆盖索引。

假如一个主键索引，其他 3 个字段建立辅助索引，select * 也属于覆盖索引。

Extra 里面值为 “Using index” 代表使用了覆盖索引。

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	user_innodb	ref	comidx_name_phone	comidx_name_phone	1070	const,const	1	100	Using where; Using index

5. 索引的创建与使用

因为索引对于改善查询性能的作用是巨大的，所以我们的目标是尽量使用索引。

4. 1. 索引的创建

1、在用于 where 判断 order 排序和 join 的 (on) 字段上创建索引

2、索引的个数不要过多。

——浪费空间，更新变慢。

3、过长的字段，建立前缀索引。

4、区分度低的字段，例如性别，不要建索引。

——离散度太低，导致扫描行数过多。

5、频繁更新的值，不要作为主键或者索引。

——页分裂

6、随机无序的值，不建议作为主键索引，例如身份证、UUID。

——无序，分裂

7、组合索引把散列性高（区分度高）的值放在前面

8、创建复合索引，而不是修改单列索引

4.2. 什么时候用不到索引？

1、索引列上使用函数（replace\SUBSTR\CONCAT\sum count avg）、表达式计算（+ - * /）：<https://www.runoob.com/mysql/mysql-functions.html>

```
explain SELECT * FROM `t2` where id+1 = 4;
```

2、字符串不加引号，出现隐式转换

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

```
explain SELECT * FROM `user_innodb` where name = 136;  
explain SELECT * FROM `user_innodb` where name = '136';
```

3、like 条件中前面带%

where 条件中 like abc%, like %2673%, like %888 都用不到索引吗？为什么？

```
explain select * from user_innodb where name like 'wang%';  
explain select * from user_innodb where name like '%wang';
```

过滤的开销太大。这个时候可以用全文索引。

4、负向查询

NOT LIKE 不能：

```
explain select *from employees where last_name not like 'wang'
```

!= (<>) 和 NOT IN 在某些情况下可以:

```
explain select *from employees where emp_no not in (1)
explain select *from employees where emp_no <> 1
```

注意跟数据库版本、数据量、数据选择度都有关系。

其实，用不用索引，最终都是**优化器**说了算。

优化器是基于什么的优化器？

基于 cost 开销(Cost Base Optimizer), 它不是基于规则(Rule-Based Optimizer), 也不是基于语义。怎么样开销小就怎么来。

使用索引有基本原则，但是没有具体细则，没有什么情况一定用索引，什么情况一定不用索引的规则。