从一个实例演示了解索引

我们来看**user_innodb**这张表，里面有四个字段：

```
CREATE TABLE `user_innodb` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `gender` tinyint(1) DEFAULT NULL,
  `phone` varchar(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1000001 DEFAULT
CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

通过**batch-insert**这个脚本，往这个表中插入300W条数据，然后通过下面的sql进行检索

```
select * from user_innodb where name='杨戚榛';
```

300W的数据查询耗时是2.14s左右。

```
select * from user_innodb where name='杨戚榛'
> OK
> 时间: 2.142s
```

接下来我们在**name**字段上创建一个索引

```
ALTER TABLE user_innodb DROP INDEX idx_name;
ALTER TABLE user_innodb ADD INDEX idx_name (name);
```

```
ALTER TABLE user_innodb ADD INDEX idx_name (name)
> OK
> 时间: 15.411s
```

添加索引的耗时较长，在一张300W数据的表中，创建一个普通索引耗时15S。

添加索引之后，再来执行:

```
select * from user_innodb where name='杨戚榛'
> OK
> 时间: 0.002s
```

可以很明显的看到效率提升非常大，通过这个案例大家应该可以非常直观地感受到，索引对于数据检索的性能改善是非常大的。

所以为什么能够为查询提供这么高的性能呢?

索引的原理揭秘

在揭秘索引的原理之前，我们先来了解什么是索引:

官方定义是： 索引（Index）是帮助MySQL高效获取数据的数据结构，简单来说，索引是一种数据结构。

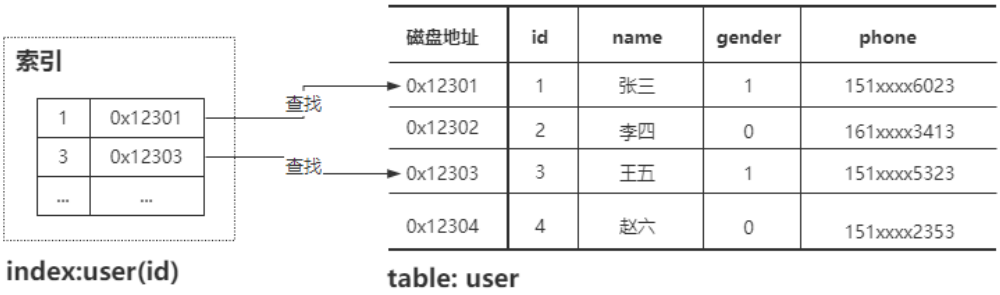
按照生活场景来理解，我们以一本中华字典为例，这本字典非常厚，如果我们要从字典里面找一个汉字是非常耗时的，所以在字典的首页，提供了根据拼音和偏旁来查找，这个拼音或者偏旁就是这本字典的索引。

除了词典，生活中随处可见索引的例子，如火车站的车次表、图书的目录等。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方式来锁定数据。

那么数据库中的索引应该如何实现呢?

索引设计思考

在了解索引的实现之前，我们不妨先来看一下下面这个图



这个图是一个抽象结构图，并不是真正的物理存储结构。

以这个图为例，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。

> 可是这里还有一个问题，就是索引应该使用什么样的数据结构来存储， 才能减少磁盘IO次数呢?

索引数据结构设计

其实，数据结构的选择，无非就是时间复杂度的选择。

简单来说，就是通过通过最少的IO次数，从磁盘中检索到目标数据！所以索引的实现本质上就是一个**查找算法**。

那么，我们需要找到一个最合适的数据结构来实现快速查找功能，最基本的查询算法当然是顺序查找（linear search），这种复杂度为O(n)的算法在数据量很大时显然是不合适的，除了顺序查找，在数据结构中还有很多选择，比如:

1. 二叉查找树

2. 平衡二叉查找树
3. 红黑树

多路平衡查找树（B Tree）

基于上述分析的问题，其实有一个核心的点在于树的深度。

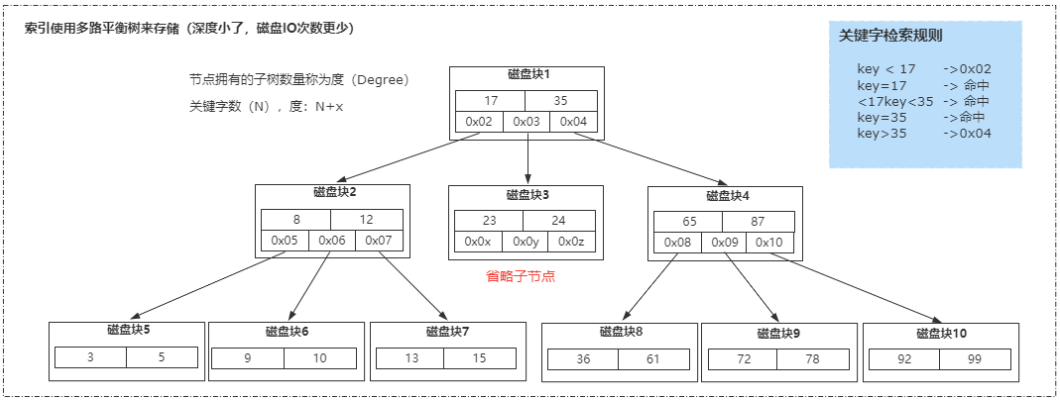
有没有其他的数据结构能够减少树的深度？从而降低和磁盘IO的次数呢？

于是，我们找到另外一种**多路平衡查找树**，也就是我们熟知的B Tree（B代表平衡）。

> 1970 年，R.Bayer 和 E.Mccreight 提出了一种适用于外查找的平衡多叉树——B-树，磁盘管理系统中的目录管理，以及数据库系统中的索引组织多数采用 B-Tree 这种数据结构。

B Tree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（当然肯定不只存3个这么少）！

分叉数越多，树的深度就会更少，这样一来，原本高高瘦瘦的二叉树就变成了矮矮胖胖的样子，如下图所示。



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。

1. 15小于17，走左边。
2. 15大于12，走右边。

3. 在磁盘块7里面就找到了15。

20条数据，由于多路数减少了树的深度，所以只有了3次磁盘IO就找到了，极大的减少了磁盘交互次数。

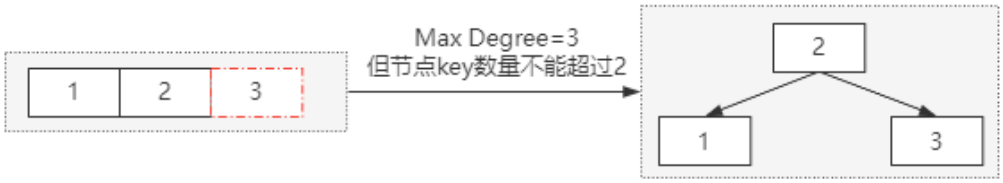
B Tree的工作原理

B Tree的整体结构很好理解，那么它是怎么实现一个阶段存储多个关键字，还能继续保持平衡的呢?

我们先来看一下插入数据时的场景。

假设在Max Degree(路数)是3的的B Tree中，意味着一个节点只能有3个指针，所以一个节点只能存储2个key。

假设我们插入1、2、3这三个key，那么这个时候需要通过分裂的方式来保持B Tree的平衡，那么它的演变过程如下。



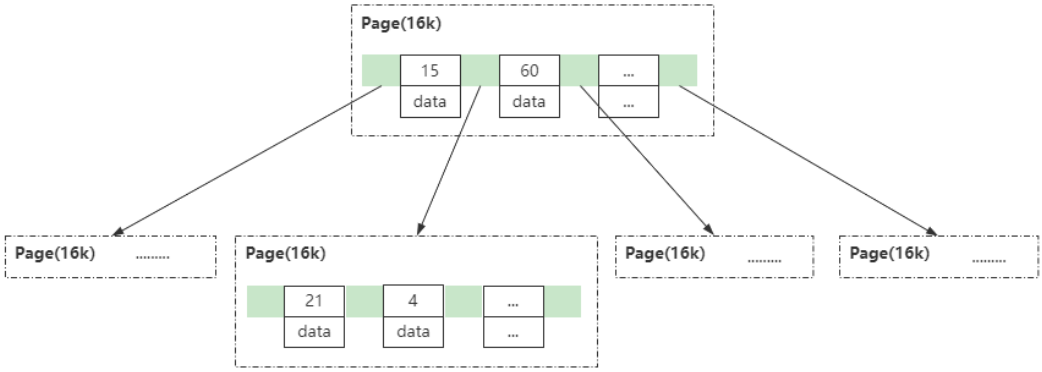
节点的排列规则需要满足平衡二叉树的基本规则（左子节点小于父节点，右子节点大于父节点）！

B Tree的优势和弊端

我们上面描述的B Tree中的节点，在Mysql中代表的是Page页。

前面我们说过，Mysql为了更好的利用磁盘的预读能力，把一个Page页的大小设置为16k，也就是一个节点（磁盘块）的大小是16K。也就是说在进行一次磁盘IO时，会把一个节点（16K）的索引加载到内存中。

假设我们设置的一个表的索引字段类型是int，也就是4个字节，如果每个关键字对应的数据区(data)也是4个字节。



在不考虑子节点引用的情况下，那么在B Tree中，每个阶段大概能存储的关键字数量是：

(16 * 1024) / 4+4 = 2048 个关键字

> 16*1024 把K单位转化成字节单位， 4+4表示字段长度以及内容的长度!

在B Tree中，一共有2049路。 对于二叉树来说，三层高度最多可以保存7个关键字，而对于这种2001路的B树，三层高度能够保存的关键字数远远大于二叉树，因此B Tree用来存储索引非常合适。

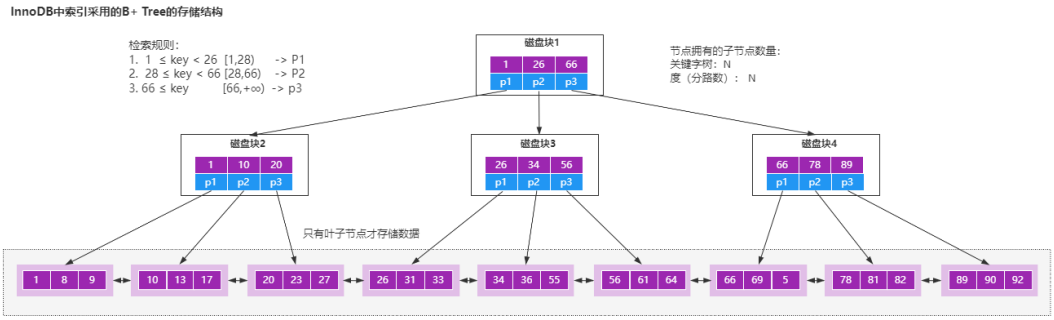
> B Tree的弊端

B Tree在做检索时，检索效率非常高，但是在做数据插入和删除时，会破坏B Three本身的行知，所以为了保持B Tree的平衡，需要对节点进行分裂、合并、转移等操作，而这个操作在节点数量较多的情况下性能影响较大。

所以这也是为什么我们在做索引的创建和索引更改时，性能较慢的原因!

B+Tree(加强版多路平衡查找树)

在Mysql的InnoDB引擎中，并没有使用B Tree作为索引的存储结构，而是使用B+Tree！它的存储结构如下图所示。



B+ Tree相比B Tree，做了以下几个方面的优化：

1. B树的路数和关键字的个数的关系不再成立了，数据检索规则采用的是左闭合区间，路数和关键个数关系为1比1
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，并且每个叶子节点都会增加一个指针指向响铃的叶子节点，形成一个有序链表结构。

在这样的几个B+ Tree结构下，假设我们要检索x=1的数据，那么检索的规则是：

1. 取出跟磁盘块， 加载1/26/66三个关键字。
2. x<=1， 取出磁盘块2，加载1/10/20三个关键字。
3. x<=1， 取出叶子节点，加载1/8/9三个关键字。此时已经达到了叶子节点，命中1， 所以只需要加载对应1的数据内容（或者内容地址）即可！

B TREE和B+TREE区别是什么？

1. B+Tree 关键字的搜索采用的是左闭合区间，之所以采用左闭合区间是因为他要最好的去支持自增id，这也是mysql的设计初衷。即，如果id = 1命中，会继续往下查找，直到找到叶子节点中的1。
2. B+Tree 根节点和支节点没有数据区，关键字对应的数据只保存在叶子节点中。即只有叶子节点中的关键字数据区才会保存真正的数据内容或者是内容的地址。而在B树种，如果根节点命中，则会直接返回数据。
3. 在B+Tree中，叶子节点不会去保存子节点的引用。
4. B+Tree叶子节点是顺序排列的，并且相邻的节点具有顺序引用的关系，如上图中叶子节点之间有指针相连接。

MySQL为什么最终要去选择B+Tree?

总结一下，InnoDB中的B+Tree特性带来的优势：

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（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）

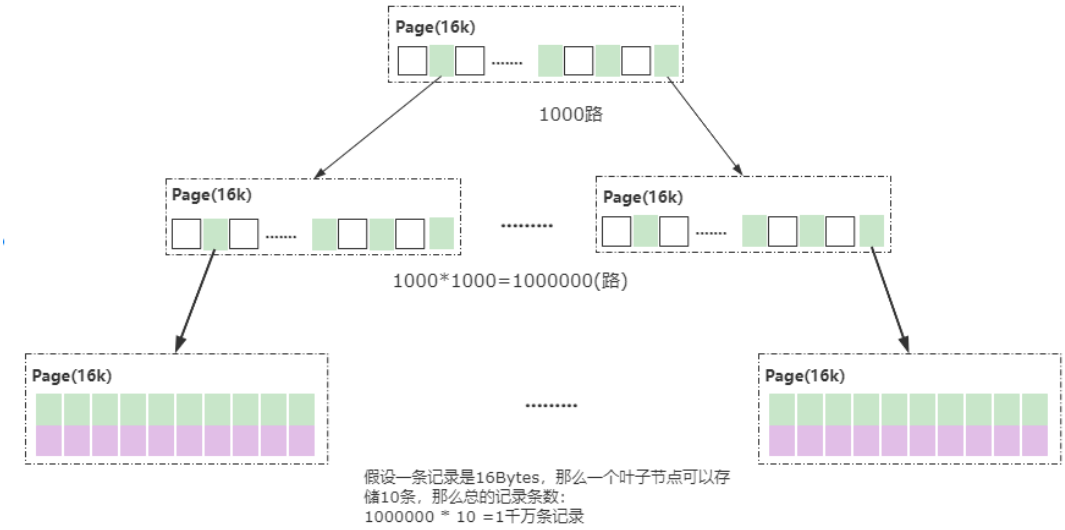
B+Tree作为索引的数据结构带来的好处

由于在B+ Tree中，每个节点不存存储数据区，只需要存储键值+指针，使得B+ Tree在每个节点存储的路数更多。

假设索引字段+指针大小一共是16个字节，那么一个Page页（一个节点）能存储1000个这样的单元（键值+指针）。

假设一条记录是16bytes，一个叶子节点（一页）可以存储10条记录！

当数的深度是2的时候，就有1000^2个叶子节点，可存储的数据为1000 x 1000 x 10 =10000000（千万）



也就是说，在InnoDB中B+树的深度在3层左右，就能满足千万级别的数据存储。

MyISAM引擎的索引实现

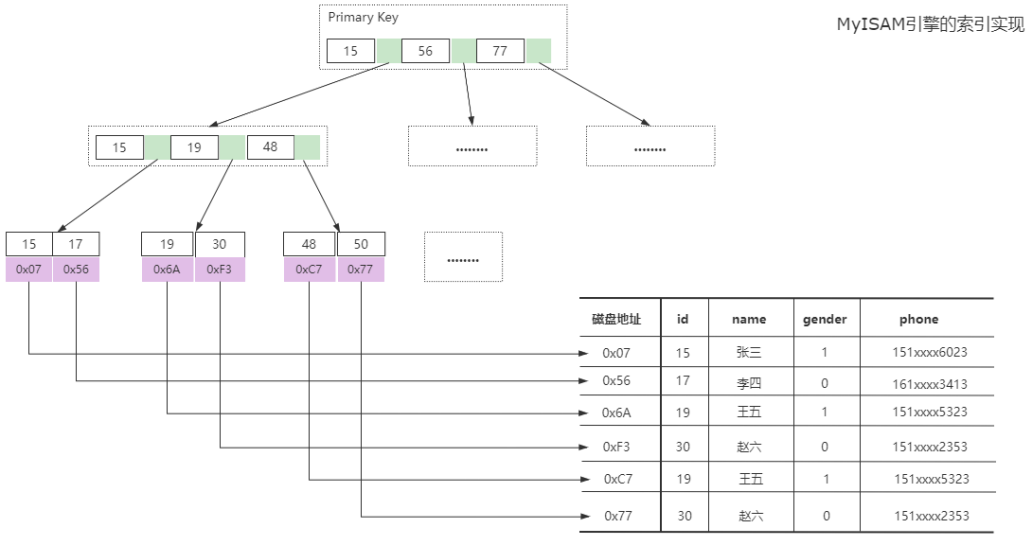
在MyISAM里面，另外有两个文件：

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。一个索引就会有一棵B+Tree，所有的B+Tree都在这个MYI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？实现机制如下图所示。



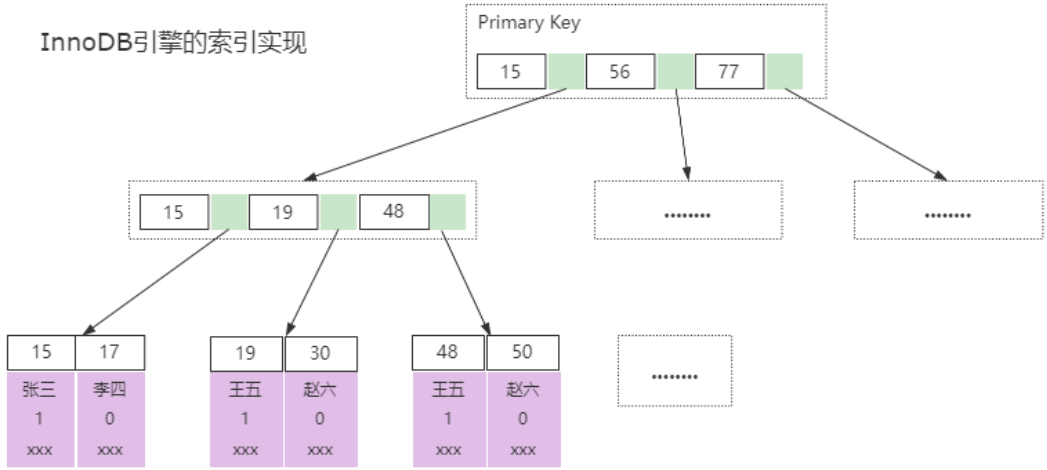
从MyISAM引擎中索引的实现来看，由于索引文件和数据文件是分离的，叶子节点存储的是数据文件对应的磁盘地址，从索引文件.MYI中找到键值后，会到数据文件.MYD中获取相应的数据记录。

在MyISAM引擎中，主键索引和辅助索引在结构上没有任何区别，只是主键索引要求key是唯一的，而辅助索引的key允许重复！

InnoDB的索引实现

在InnoDB中，只有一个ibd文件，里面包含索引和数据。

同时，在B + Tree中的叶子节点存储了索引对应的数据行，所以我们称InnoDB中索引即数据、数据即索引，它的整体结构如下图所示。



如上图中，叶子节点的数据区保存的就是真实的数据，在通过索引进行检索的时候，命中叶子节点，就可以直接从叶子节点中取出行数据。

聚簇索引和非聚簇索引

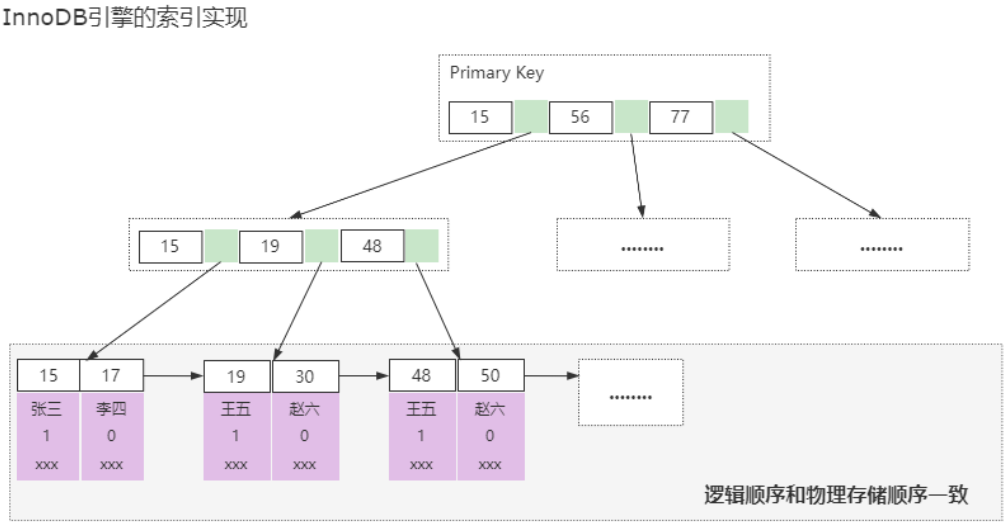
在一个表中，我们可以建立很多中索引，如唯一索引、主键索引、辅助索引等等，那如果是一个表中存在多个索引的情况下？

数据表应该保存到哪个索引的叶子节点呢？ 还是说每个索引的叶子节点都保存一份？

很显然，即便是让我们自己设计，也不可能在每种索引上都冗余一份数据，因为这会带来额外的存储空间浪费和计算消耗，所以在InnDB中，引入了聚集索引（聚簇索引）和非聚集索引的概念。

聚簇索引

所谓的聚簇索引，就是只索引键值的逻辑顺序和表数据行的物理存储顺序一致。只有聚簇索引才会在叶子节点缓存表中的数据。

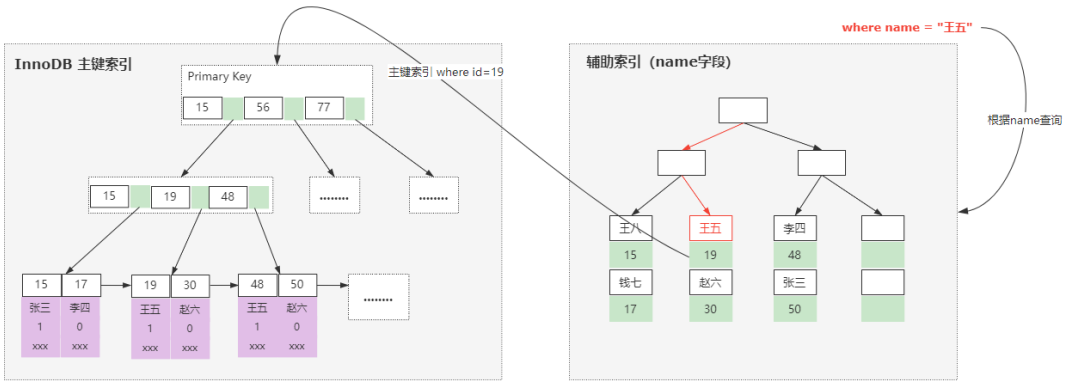


在InnoDB中，组织数据的方式就是用聚簇索引组织表（Clustered Index Organize Table），所以一张表创建了主键索引，那么这个主键索引就是聚集索引。

非聚簇索引

除了主键索引以外，其他索引均属于非聚簇索引，非聚簇索引的叶子节点不会存储表数据，那么在这种情况下。

如果要查询一个非聚簇索引的字段，怎么去获取到数据的值呢？我们来看一下非聚簇索引的存储结构。



从上面这个图可以看到，真正的数据仍然是保存到主键索引的叶子节点（这也就是为什么InnoDB表必须要有主键的原因），而辅助索引的叶子节点的数据区保存的是主键索引的关键字的值（**非主键索引叶子节点的逻辑顺序和磁盘顺序不一致**）。

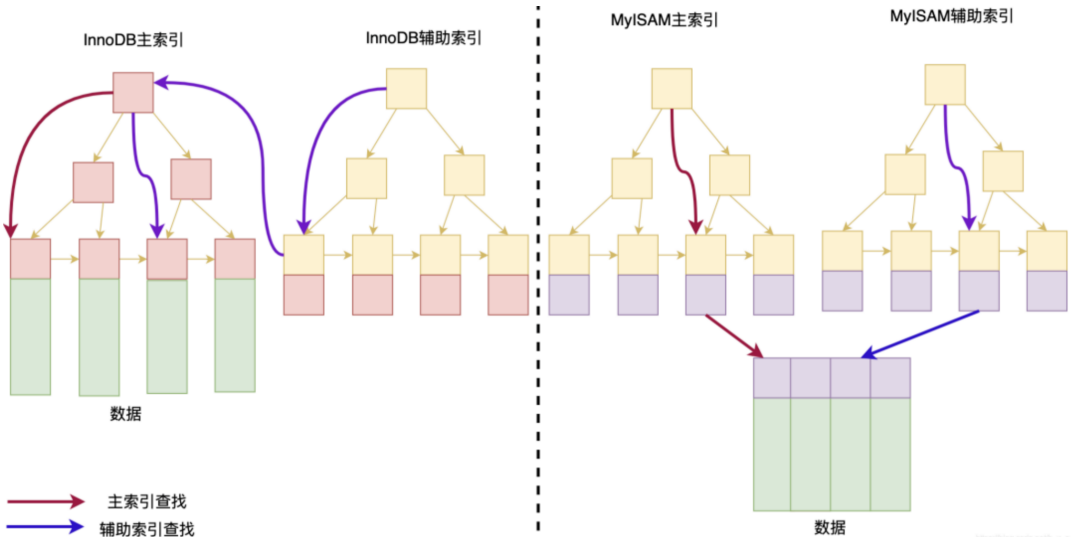
当我们要查询where name = '王五'时，先通过二级索引去B + Tree中找到王五的叶子节点，拿到对应的主键值，也就是id=15。

接着在根据这个条件去主键索引中查找到叶子节点拿到数据（这里不能存储磁盘地址，因为在数据insert和delete时，B+Tree的结构会发生变化）。

所以，从这个角度来说，因为主键索引比二级索引少扫描了一棵B+Tree（这个动作叫做回表），它的速度相对会快一些。

MyISAM和InnoDB两种引擎索引区别

下面我们通过这个图来给大家综合对比一下两种引擎的索引区别！



索引的实际使用建议

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。

索引创建建议

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

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。http://blog.itpub.net/30317998/viewspace-2654308/

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。

——B+树的平衡导致 页分裂 ，影响效率！

6、随机无序的值，不建议作为索引，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！

什么时候索引会失效?

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

2. 字符串不加引号，出现隐式的类型转化

```
--创建索引
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX comidx_name_phone
(name,phone);

--针对字符串列的where条件，分别添加引号和不添加引号
explain SELECT * FROM `user_innodb` where name = 136;
explain SELECT * FROM `user_innodb` where name = '136';
```

不带引号存在类型转化，所以会导致全表扫描!

3. like条件中前面带%

where条件中like %wang%，like %wang都用不到索引，

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

那为什么like wang%这种写法会执行索引呢?

首先，字符串的匹配是基于最左匹配规则来执行的，为什么是最左匹配，其实是和 B+Tree 有些关系，索引树从左到右都是有顺序的。对于索引中的关键字进行对比的时候，一定是从左往右以此对比，且不可跳过。

为什么是最左匹配原则？这个其实很好理解。比如，我们要比较一个字符串。michael 与 mic，我们肯定是先从第一个字符开始比较吧，第一个相同后，再比较第二个字符，以此类推。所以要从左边开始，并且是不能跳过的。SQL 索引也是这样的。

然后我们再来分析%，% 在前，就代表，我前面的内容不确定，对于不确定的数据，索引无法比较，所以只能一个个的去比，这就相当于是全表扫描了，全表扫描就没必要走索引，因此%写在前面是一定不会走索引的。

如果%写在后面，由于前面的字符是确定的，所以可以通过索引来进行匹配！

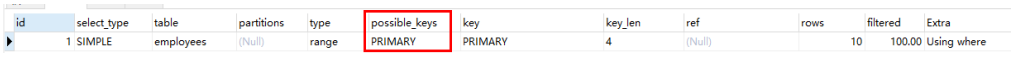
4. 负向查询

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;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	(Null)	range	PRIMARY	PRIMARY	4	(Null)	10	100.00	Using where

从执行计划中可以看到用到了主键索引，这事因为索引是有序的，因此只要从1之后开始顺序读取就行，所以会走主键索引！

随堂问题

like %%这个要怎么优化才能走索引?	
MySQL使用正则表达会使用索引吗	
联合索引最左匹配原则那个例子，把name放到phone后面，查看下，会不会走索引?	

like %%这个要怎么优化才能走索引?	
老师索引下推怎么理解？索引下推底层原理和执行流程是什么？怎么理解	
索引离散度的那个例子，为什么那个优化器，怎么优化不走索引了，	
索引的离散度多少为合适呢?	
索引列离散度太低不会优化成全表扫描嘛	
创建索引期间，频繁查询，修改数据会有什么影响，为什么	
在使用MySQL主建是否可以自定义有序的字符串 若每个表主键都是bigint 感觉业务区分id很混淆	
自增主键索引和uuid主键索引有区别吗？uuid主键索引会比自增主键索引慢吗?	
老师官方手册定义可设置varchar最大长度为65535，是不是可以理解为所有varchar列总和为65535。这样不就超过页的大小?	
B+空间复杂度时间复杂度怎么算的?	
硬盘存储索引的空间有大小限制吗？存放在什么地方？如果没有限制及时超大数据量是不是都可以建索引呢？如果有限制的话是多大呢	
MySQL表结构设计时，列数一般有没有限制？业务上要求的列很多，至少几百列，是一个大宽表好，还是多个小表好?	
InnoDB的页大小为16k，总共多少个字节？那实际可以存储记录长度为多少时，会发生溢出?	
一页16kb，一个数据区是16b，那么第一路有1000个数据的话，第二路就应该有1001个，第三路就有应该有1003个数据区啊	
锁引的使用，跟数据的ID有关系吗？比如自增id和随机的ID.有影响吗	

like %%这个要怎么优化才能走索引?	
B+树如果根节点和枝节点都无法存储完所有的索引，那怎么办？看图是枝节点只有一层。 枝节点是不是也可以有多层？B+树不是一般最多三次io就可以找到数据么？	
MySQL大宽表情况下，查询的列没有加索引的话，InnerDb怎么查找呢？比如查找个人简介中某个关键词——架构师 之类的，个人简介数据类型是长字符串	
同一个表建立一个锁引和多个锁引，所占的空间一样吗	
刚才的b+树为什么存储1000万的数据，怎么计算的，第二层是10001000，第三层应该是10001000*1000个节点吧？	
一千万数据，索引真的就无效了吗？	
16字节，怎么B+树叶子节点存储10条记录	
b+树根节点的路数数据怎么判断产生的	
B树的节点存数据吗？	
红黑树什么是红节点什么事黑节点，怎么看，定义是什么	