

课程说明

- 索引的本质
- 数据结构与算法
- B+树深入剖析
- 索引的使用

1、索引的本质

在生产环境中，随着数据量不断的增长，SQL执行速度会越来越慢，常见的手段就是通过索引来提升查询速度，那么究竟为什么要添加索引？应该如何正确添加索引？本套课程将以MySQL为例，从索引的底层进行分析，一起探索索引相关的一系列问题。

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构（有序）。所以，就可以得到索引的本质：索引是有序的数据结构。

1.1、环境

本套课程以MySQL5.7为例讲解，使用docker进行部署：

```
docker run --name mysql-5.7 -e MYSQL_ROOT_PASSWORD=root -d -p 13306:3306 -v mysql-5.7-data:/var/lib/mysql -v mysql-5.7-conf:/etc/mysql/conf.d mysql:5.7.31
```

1.2、索引文件

在MySQL中，索引实际上存储在文件中的，其位置与数据库数据文件在相同的目录中。

-- 创建测试表

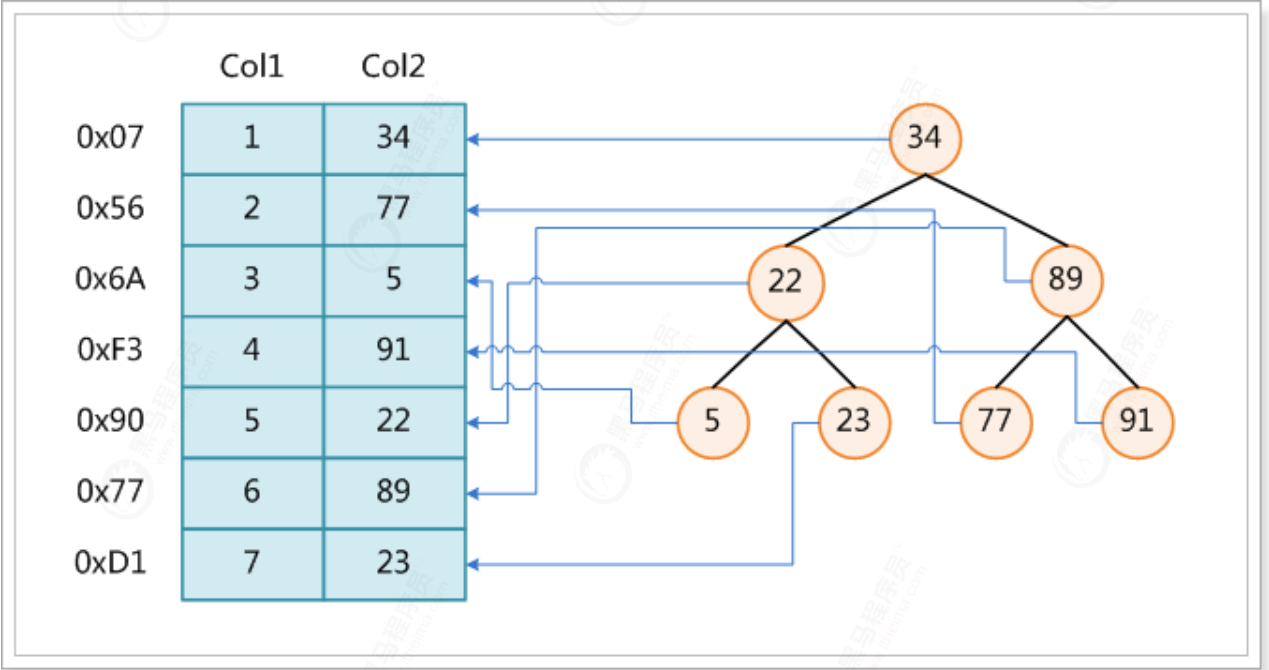
```
CREATE TABLE `tb_user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `username` varchar(20) DEFAULT NULL,  
  `password` varchar(20) DEFAULT NULL,  
  `created` datetime DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `username` (`username`) USING BTREE  
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

```
CREATE TABLE `tb_user2` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `username` varchar(20) DEFAULT NULL,  
  `password` varchar(20) DEFAULT NULL,  
  `created` datetime DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

查看数据库文件，路径：/var/lib/docker/volumes/mysql-5.7-data/_data/

```
-rw-r-----. 1 polkitd input      61 9月   2 11:31 db.opt
-rw-r-----. 1 polkitd input    8668 9月   2 11:36 tb_user2.frm  #表结构文件
-rw-r-----. 1 polkitd input       0 9月   2 11:36 tb_user2.MYD  #MyISAM引擎类型的
表数据文件
-rw-r-----. 1 polkitd input    1024 9月   2 11:36 tb_user2.MYI  #MyISAM引擎类型的
索引文件
-rw-r-----. 1 polkitd input    8668 9月   2 11:33 tb_user.frm  #表结构文件
-rw-r-----. 1 polkitd input 114688 9月   2 11:34 tb_user.ibd  #InnoDB的表空间文
件，用于存储数据以及索引文件
```

1.3、索引示例



此示例中，在左侧表示的2列数据，其中最左侧的为数据的物理地址，真实的地址可能并不是连续的地址。右侧是根据col2建立的索引结构，先把此结构看作是二叉树结构，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针。

查找数据：

- 在没有索引的情况下，会全表扫描数据，其时间复杂度为： $O(n)$ ，值为：7
- 在索引（二叉树）情况下，查找数据的时间复杂度为： $O(\log_2 n)$ ，值为：2.8073549221
- 可见，基于索引的查询要更快一些。

时间复杂度是指执行算法所需要的计算工作量，一般用大O符号表述。

由于二分查找每次查询都是从数组中间切开查询，所以每次查询，剩余的查询数为上一次的一半，从下表可以清晰的看出查询次数与剩余元素数量对应关系。

第几次查询	剩余待查询元素数量
1	$N/2$
2	$N/(2^2)$
3	$N/(2^3)$
...	...
K	$N/(2^K)$

从上表可以看出 $N/(2^K)$ 肯定是大于等于1，也就是 $N/(2^K) \geq 1$ ，我们计算时间复杂度是按照最坏的情况进行计算，也就是查到剩余最后一个数才查到我们想要的数，也就是 $N/(2^K) = 1 \Rightarrow N = 2^K \Rightarrow K = \log_2 N$

需要注意的是，数据库系统几乎没有使用二叉查找树、红黑树实现的，在MySQL中使用的是B+Tree。

2、数据结构与算法

算法的动态演示地址：<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

2.1、二分查找法

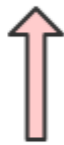
二分法查找，也称为折半法，是一种在有序数组中查找特定元素的搜索算法。

二分法查找的思路如下：

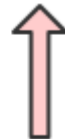
- (1) 首先，从数组的中间元素开始搜索，如果该元素正好是目标元素，则搜索过程结束，否则执行下一步。
- (2) 如果目标元素大于/小于中间元素，则在数组大于/小于中间元素的那一半区域查找，然后重复步骤(1)的操作。
- (3) 如果某一步数组为空，则表示找不到目标元素。

例如，有2、5、9、12、18、25、34、58、75、86、99这11个数，如果想要查找86这个数，其查找过程如下：

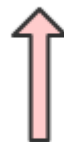
2	5	9	12	18	25	34	58	75	86	99
---	---	---	----	----	----	----	----	----	----	----



2	5	9	12	18	25	34	58	75	86	99
---	---	---	----	----	----	----	----	----	----	----



2	5	9	12	18	25	34	58	75	86	99
---	---	---	----	----	----	----	----	----	----	----



可以看到，通过二分法查找，经过了3次查找就找到了目标，如果按照顺序查找的话需要10次才能找到。显然，二分查找法要比顺序查找效率更高（平均）。

动态演示地址：<https://www.cs.usfca.edu/~galles/visualization/Search.html>

```
def binarySearch(listData, value)
    low = 0
    high = len(listData) - 1
    while (low <= high)
        mid = (low + high) / 2
        if (listData[mid] == value):
            return mid
        elif (listData[mid] < value)
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Searching For

934

Result

29

Element found

low

28

mid

29

high

31

10	54	68	73	89	167	194	223	259	293	302	362	367	380	397	410
----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

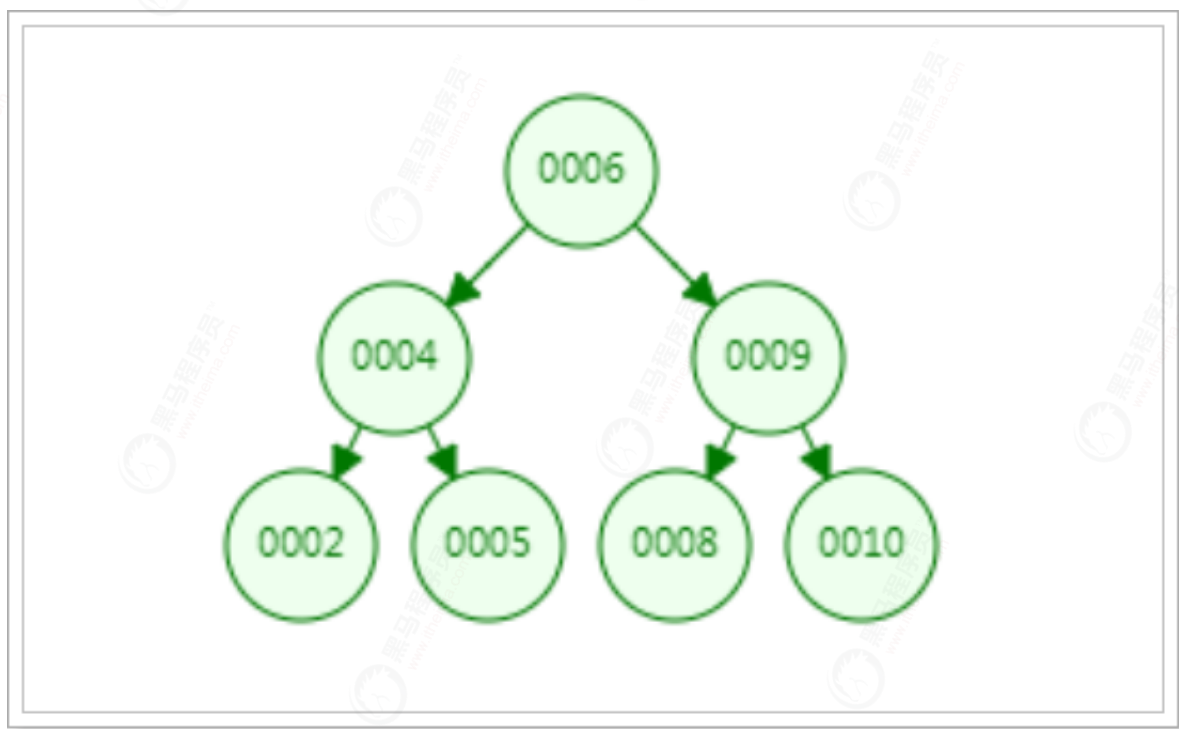
433	441	447	449	455	524	549	594	714	825	898	906	911	934	947	974
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

二分查找法一般都是在数组中操作，而数组的长度往往是不能改变的，那么，是否存在一种数据结构可以既支持二分查找，又支持动态变化大小呢？答案是肯定的，那就是二叉树。

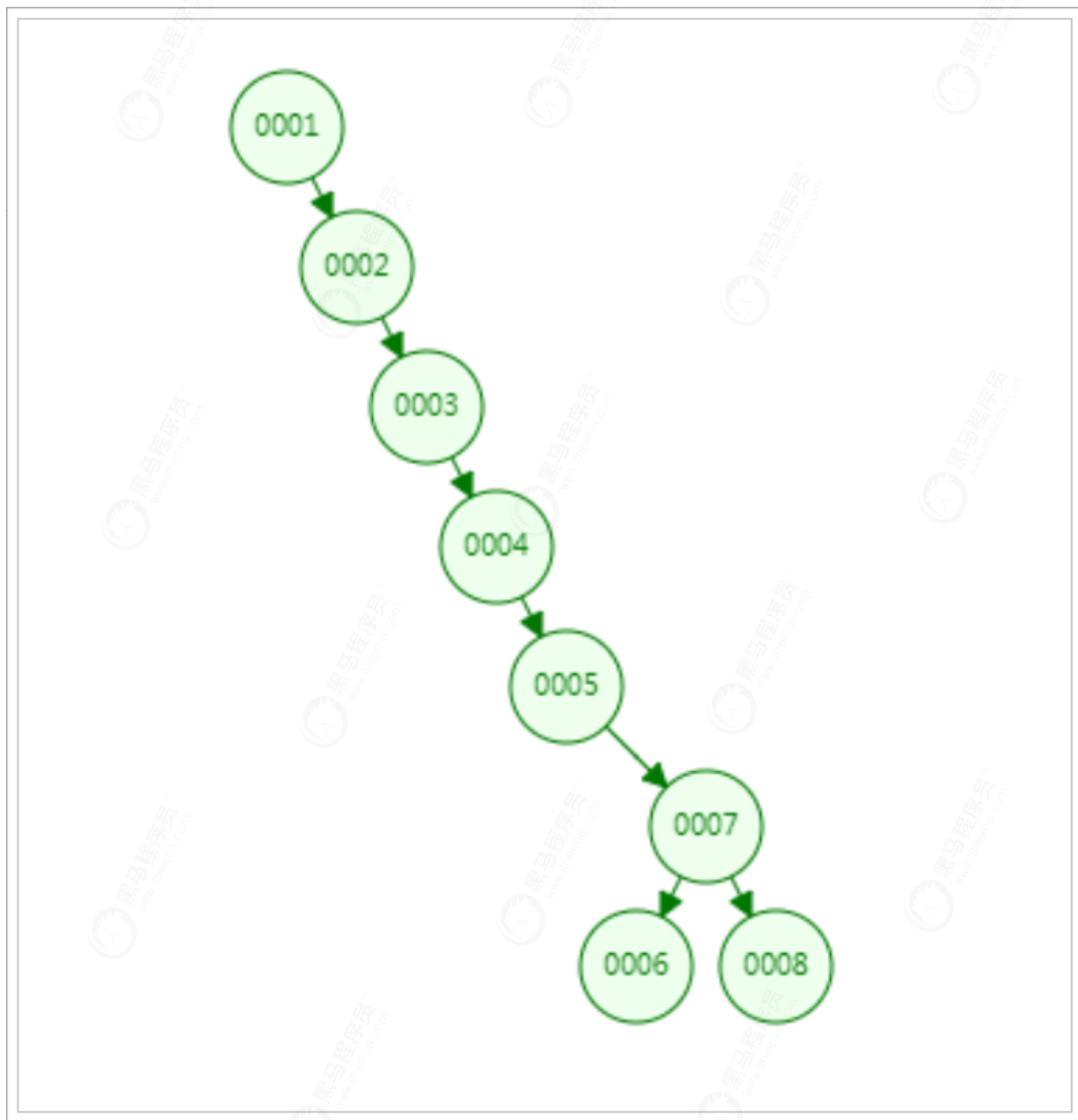
2.2、二叉树

二叉树 (binary tree) 是指树中节点的度不大于2的有序树，它是一种最简单且最重要的树。如下图：



在二叉数中，左子树的键值总数小于根的键值，右子树的键值总数大于根的键值。图中的键值排序输出为：2、4、5、6、8、9、10。

一般情况下，二叉树查找效率要高于顺序查找，但是也有特殊情况如下：

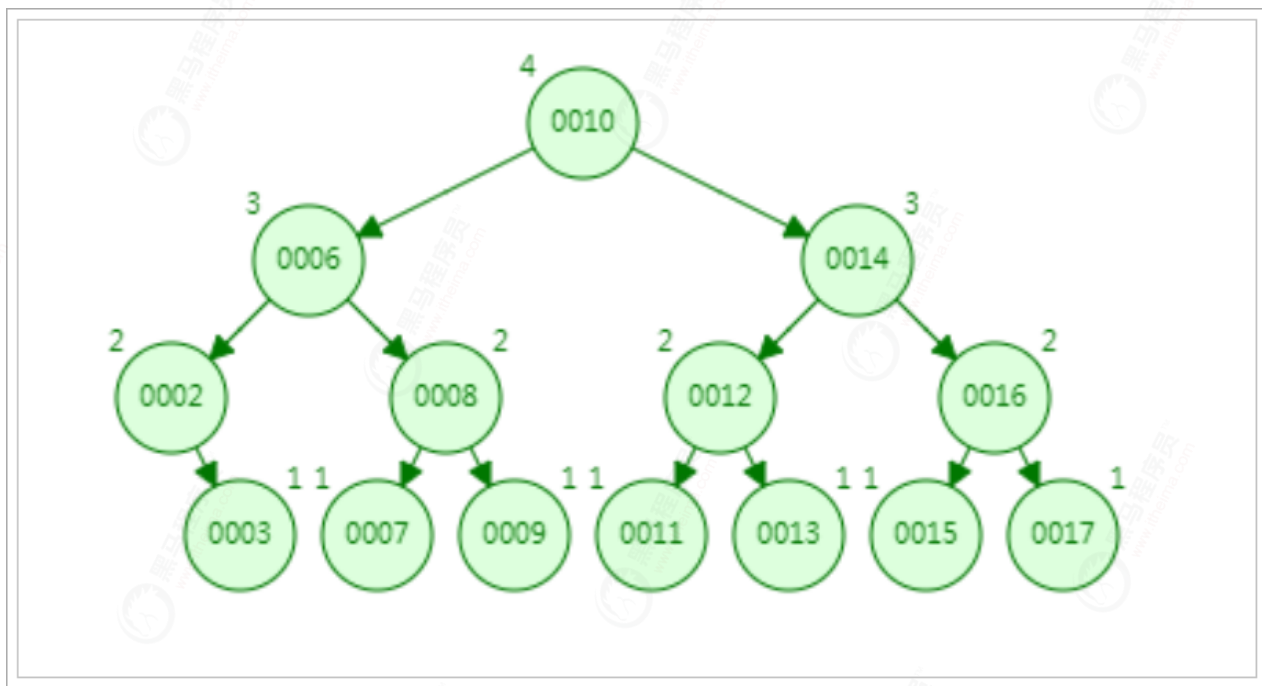


在该树中，如果查找8元素，其执行效率和顺序查找差不多了。为了解决这个问题，就需要将二叉数做平衡操作。

2.3、平衡二叉树

平衡二叉树（Balanced Binary Tree）又被称为AVL树，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在 $O(\log_2 N)$ 。但是频繁旋转会使插入和删除牺牲掉 $O(\log_2 N)$ 左右的时间，不过相对二叉查找树来说，时间上稳定了很多。



动态演示地址: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

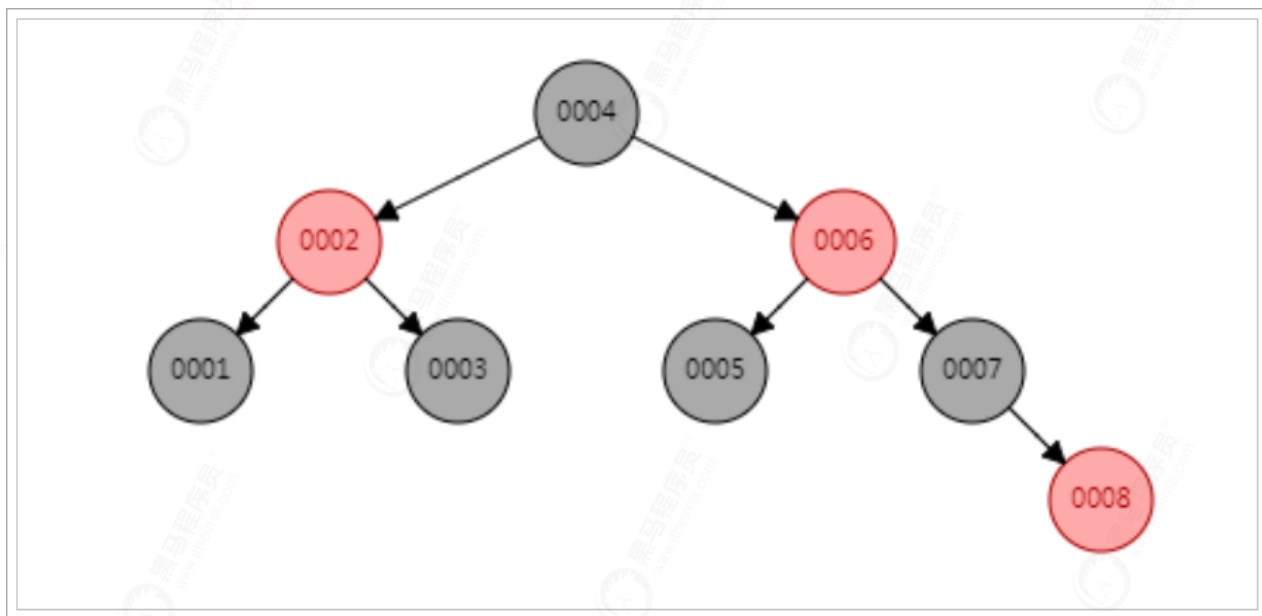
2.4、红黑树

红黑树是一种平衡二叉查找树的变体，它的左右子树高差有可能大于 1，所以红黑树不是严格意义上的平衡二叉树（AVL），但对之进行平衡的代价较低。

由于每一颗红黑树都是一颗二叉排序树，因此，在对红黑树进行查找时，可以采用运用于普通二叉排序树上的查找算法，在查找过程中不需要颜色信息。

特征：

- 节点是红色或黑色。
- 根节点是黑色。
- 所有叶子都是黑色。（叶子是NULL节点）
- 每个红色节点的两个子节点都是黑色。（从每个叶子到根的所有路径上不能有两个连续的红色节点）
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。



动态演示地址：<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

2.5、B-Tree

B-Tree是为了磁盘或其它存储设备而设计的一种多叉平衡查找树，相对于二叉树，B-Tree每个内节点有多个分支，即多叉。

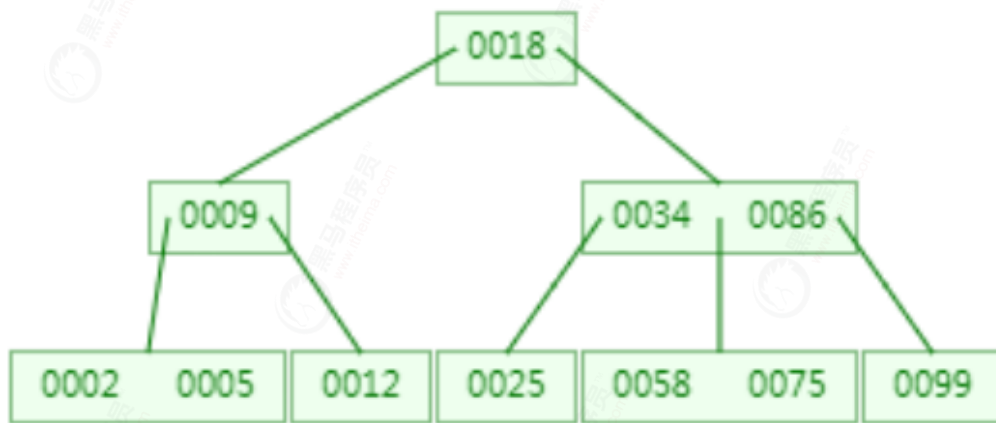
B-Tree中的2个概念：

- 度数：在树中，每个节点的子节点（子树）的个数就称为该节点的度（degree）。
- 阶数：定义为一个节点的子节点数目的最大值。

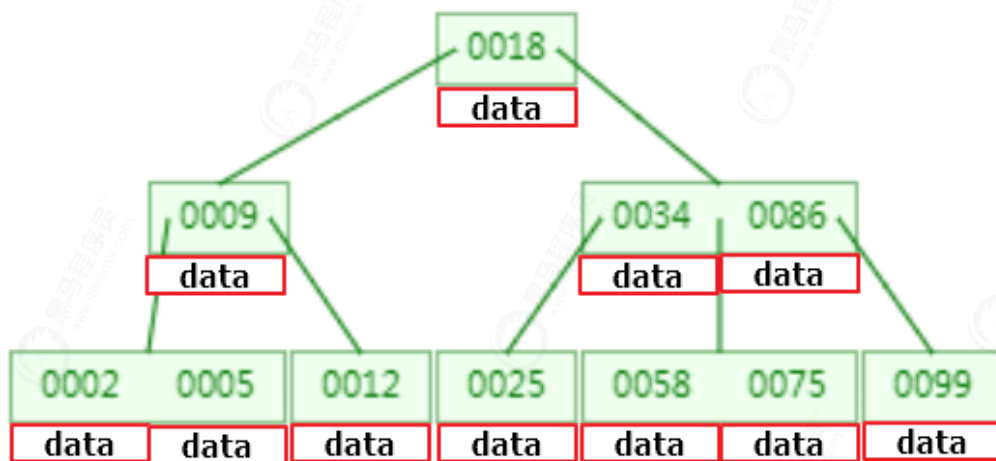
m阶B-Tree满足以下条件：

- 每个节点最多拥有m个子树
- 根节点至少有2个子树
- 分支节点至少拥有m/2颗子树（除根节点和叶子节点外都是分支节点）
- 所有叶子节点都在同一层、每个节点最多可以有m-1个key，并且以升序排列
- 每个非叶子节点由n-1个key和n个指针组成，key和指针互相间隔，节点两端是指针

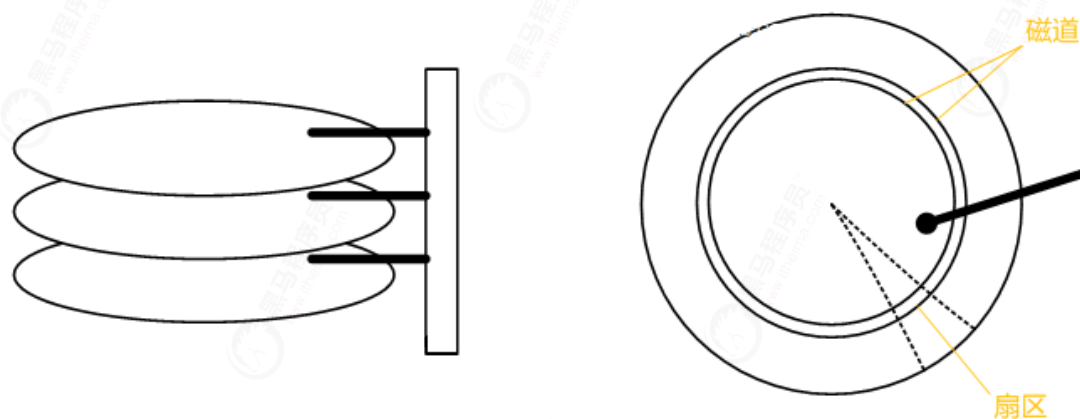
下图展现是3阶B-Tree的示意图：

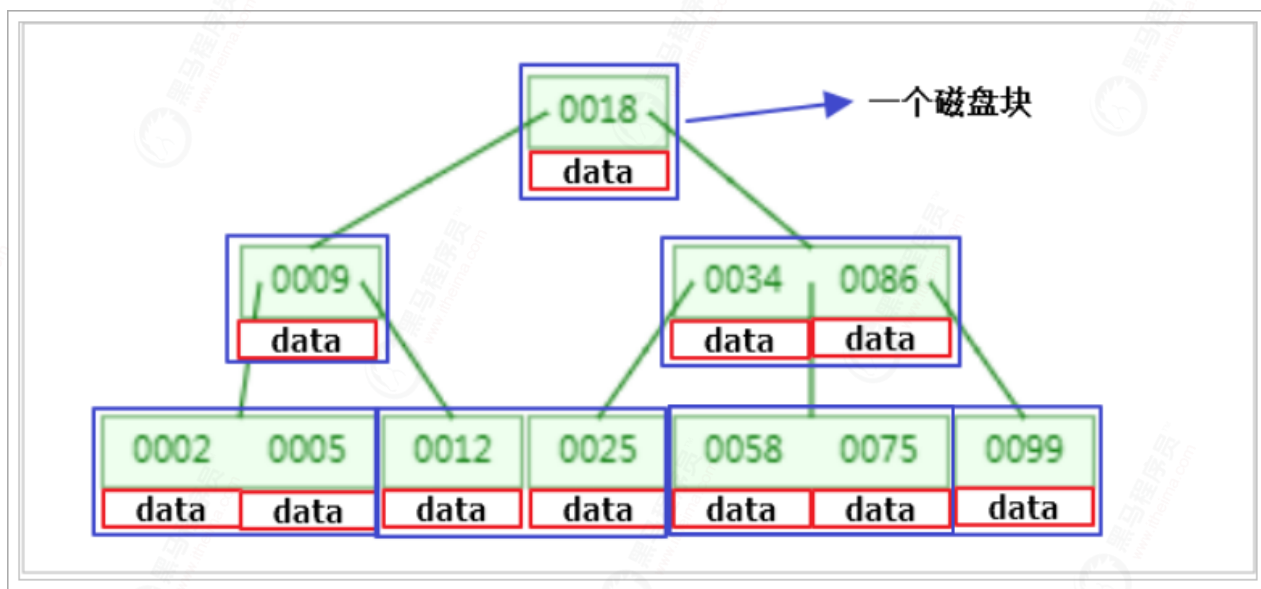


实际上，每个节点除了键值以外还包含数据，如下：



如果选用B-Tree作为索引存储的话（比如：MongoDB），每一个节点包含指针、数据都存储到一个磁盘块中（NTFS文件系统中，默认8个扇区组成一个簇，大小为4KB）：





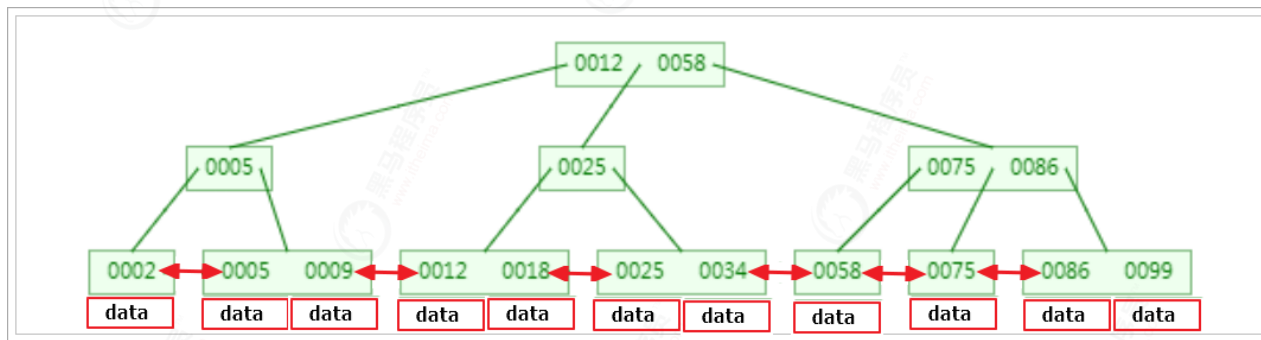
B-Tree相对较二叉树的优势在于，一次读取一个磁盘块数据中包含了多个key数据，这样就可以在内存中做比较操作，减少了磁盘IO的操作。

插入或者删除元素都会导致节点发生裂变反应，有时候会非常麻烦，但正因为如此才让B树能够始终保持多路平衡，这也是B树自身的一个优势：自平衡；

B树主要应用于文件系统以及部分数据库索引，如MongoDB，大部分关系型数据库索引则是使用B+树实现。

2.6、B+Tree

这是一个3阶的B+Tree示意图：



B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

B-Tree结构中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个块的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个块）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。

在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

- 非叶子节点只存储键值信息。
- 所有叶子节点之间都有一个链指针。

- 数据记录都存放在叶子节点中。

在B+Tree中，所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

3、B+树深入剖析

3.1、MySQL索引为什么要使用B+树

首先需要明确的是，B树或B+树要比二叉树更适合作为索引存储，因为B树中的节点可以存储多个数据，从而就可以减少树的高度，也就减少了提升了查找性能。

那么在MySQL中为什么选择B+树而不选择使用B树呢？

主要原因体现在3个方面：

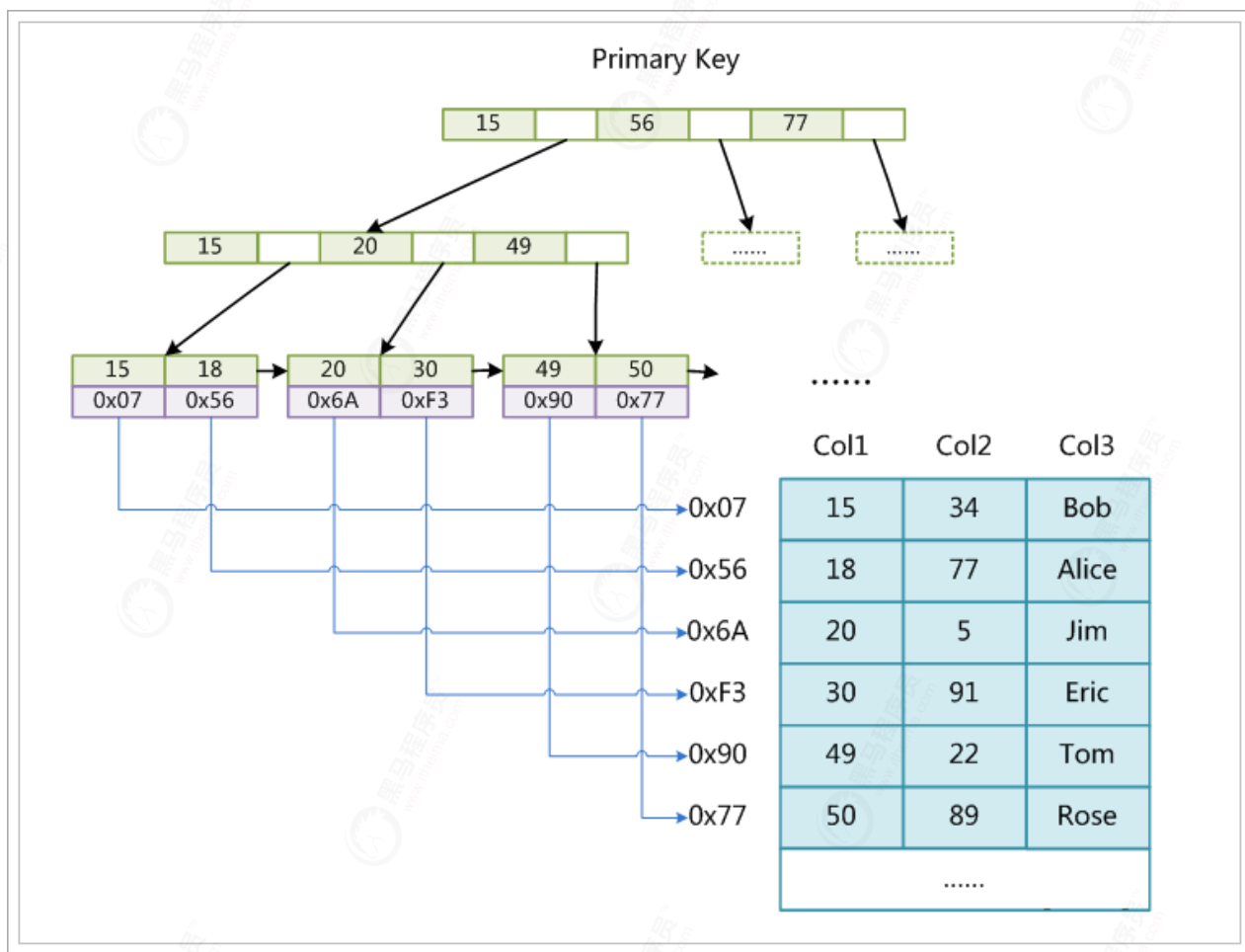
- B+树的磁盘读写代价更低
 - B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。
- B+树的查询效率更加稳定
 - 由于非终端节点并不是最终指向文件内容的节点，而只是叶子节点中关键字的索引。所以任何关键字的查找必须走一条从根节点到叶子节点的路。
 - 所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
- 由于B+树的数据都存储在叶子节点中，分支节点均为索引，方便扫库，只需要扫一遍叶子节点即可，但是B树因为其分支节点同样存储着数据，我们要找到具体的数据，需要从根节点按序开始扫描，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

3.2、MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，下面我们针对MyISAM和InnoDB两个存储引擎的索引实现方式进行讨论学习。

3.2.1、MyISAM索引实现

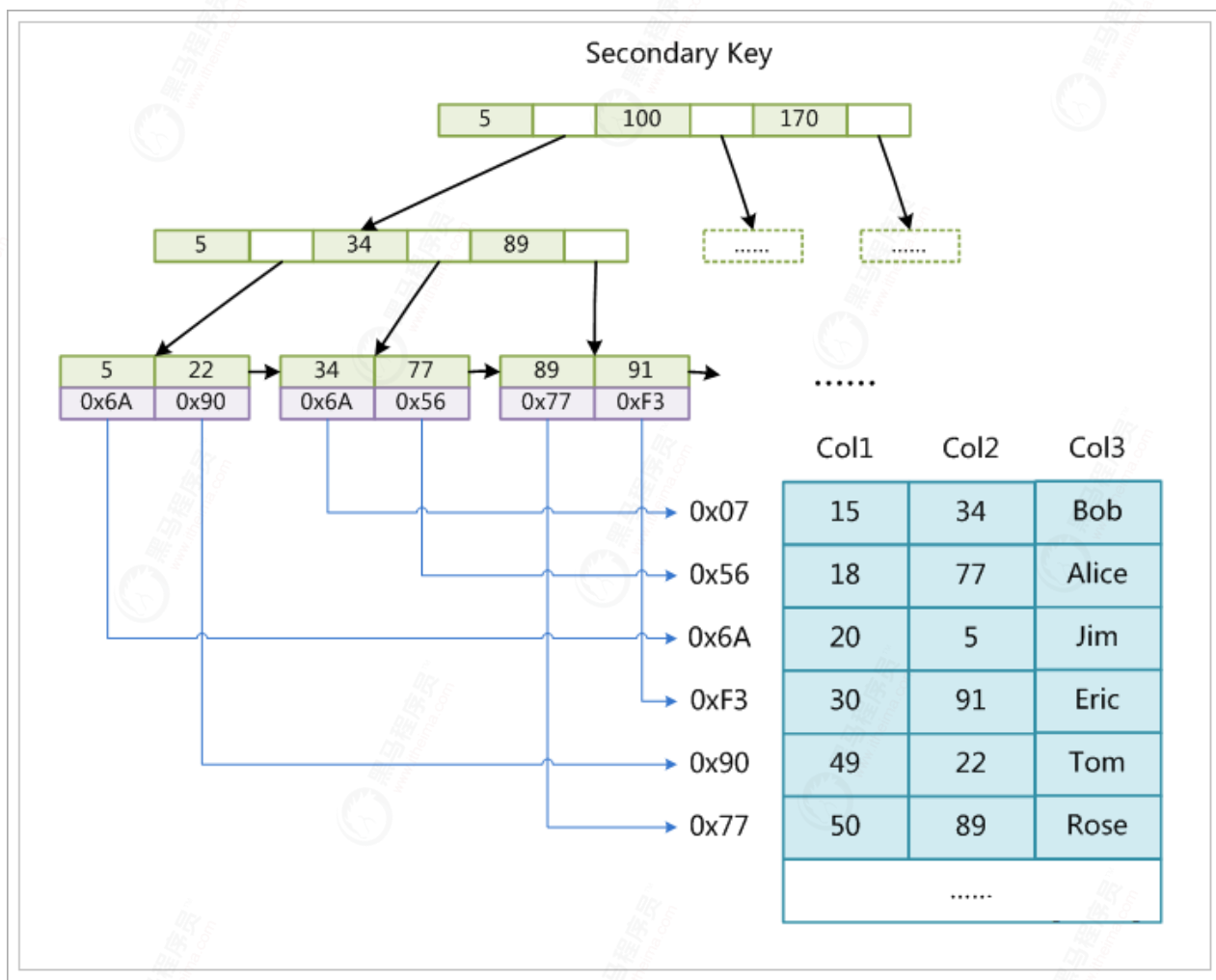
MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：



这里设表一共有三列，假设我们以Col1为主键，上图是一个MyISAM表的主索引（Primary key）示意。

可以看出MyISAM的索引文件仅仅保存数据记录的地址，在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。

如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



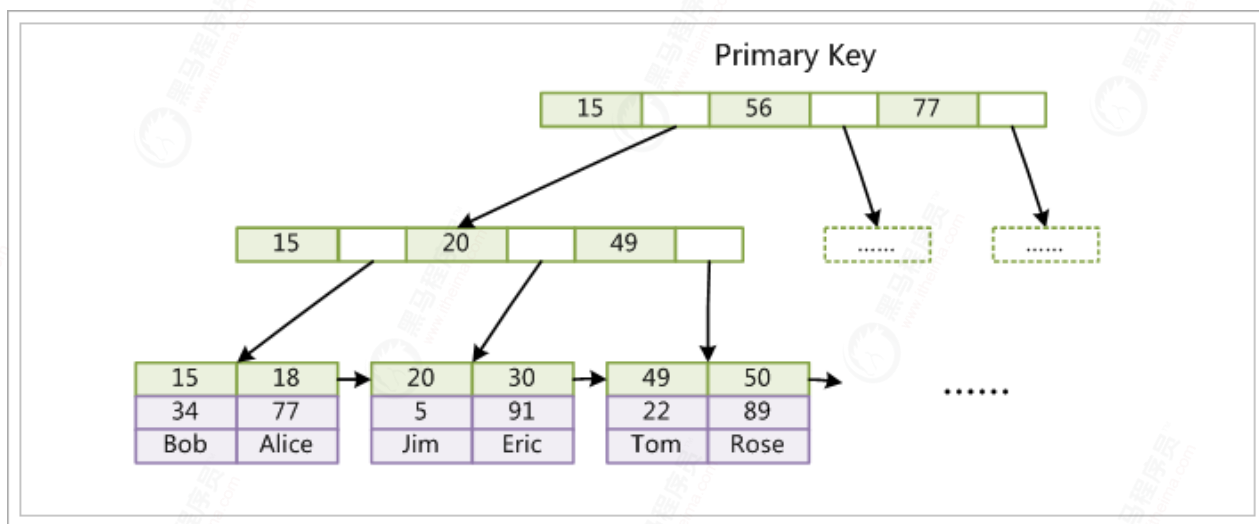
同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值作为地址，读取相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

3.2.2、InnoDB索引实现

InnoDB中的索引结构与MyISAM的索引结构有很大的不同。

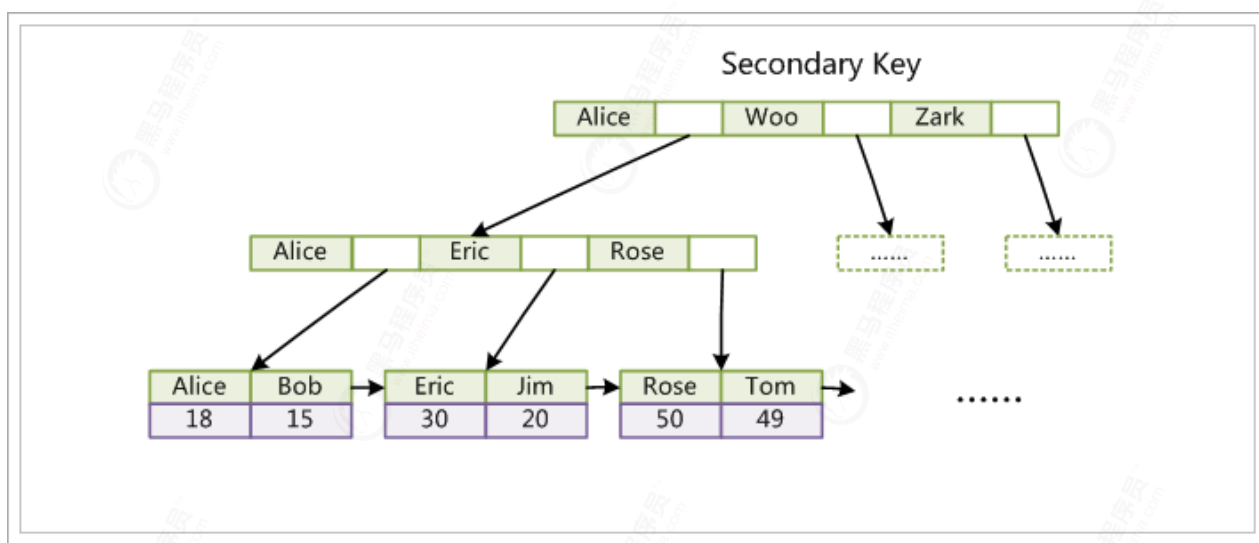
第一个重大区别是InnoDB的数据文件本身就是索引文件。在MyISAM中，索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录，这个索引的key是数据表的主键，所以InnoDB表数据文件本身就是主索引。



从图中可以看到，叶子节点包含了完整的数据记录。这种索引叫做**聚集索引**。

因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。（面试题）

第二个与MyISAM索引的不同是，InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

InnoDB的B+ 树索引的特点是高扇出性，因此一般树的高度为2~4层，这样我们在查找一条记录时只用I/O 2~4次。当前机械硬盘每秒至少100次I/O/s，因此查询时间只需0.02~0.04s。

4、索引的使用

4.1、主键索引

在InnoDB存储引擎中，每张表都会有主键，数据按照主键顺序组织存放，如果表定义时没有显式定义主键，则会按照以下方式选择或创建主键：

- 先判断表中是否有"非空的唯一索引"，如果有
 - 如果仅有一条"非空唯一索引"，则该索引为主键
 - 如果有多条"非空唯一索引"，根据索引的先后顺序，选择第一个定义的非空唯一索引为主键。
- 如果表中无"非空唯一索引"，则自动创建一个6字节大小的指针作为主键，但是该主键是不能被查询的。

测试：

--创建表

```
CREATE TABLE `tb_test1` (  
  `id` int(11) NOT NULL, -- 非空  
  `name` varchar(20) DEFAULT NULL,  
  `age` int(11) DEFAULT NULL,  
  UNIQUE KEY `id` (`id`) USING BTREE -- 唯一索引  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

--插入数据

```
INSERT INTO `tb_test1` (`id`, `name`, `age`) VALUES ('1', 'zhangsan', '20');  
INSERT INTO `tb_test1` (`id`, `name`, `age`) VALUES ('2', 'lisi', '21');
```

--查询，_rowid就是视为主键，如果表里设置了主键之后，_rowid就是对应主键。

```
SELECT *, _rowid FROM tb_test1;
```

--查询结果，可以看到_rowid的值与id值相同

id	name	age	_rowid
1	zhangsan	20	1
2	lisi	21	2

--无索引的表测试

```
CREATE TABLE `tb_test2` (  
  `id` int(11) DEFAULT NULL,  
  `name` varchar(20) DEFAULT NULL,  
  `age` int(11) DEFAULT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

--查询

```
SELECT *, _rowid FROM tb_test2;
```

--出错: [Err] 1054 - Unknown column '_rowid' in 'field list'

#解决错误

```
#[Err] 1055 - Expression #1 of ORDER BY clause is not in GROUP BY clause and
contains nonaggregated column 'information_schema.PROFILING.SEQ' which is not
functionally dependent on columns in GROUP BY clause; this is incompatible with
sql_mode=only_full_group_by
```

```
vim /var/lib/docker/volumes/mysql-5.7-conf/_data/mysql.cnf
```

#写入如下配置

```
[mysqld]
sql_mode=STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY
_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
```

#重启容器

```
docker restart mysql-5.7
```

在InnoDB中，建议使用自增id作为表的主键，这样有利于数据在底层的顺序存储，如果对于分表存储的数据，可以设置不同的步长解决或者使用第三方的代理框架解决。

4.2、联合索引

联合索引就是将表中的多个列一起进行索引，需要注意的是，联合索引是有顺序的，比如：A、B、C列的索引与A、C、B列的索引是不一样的。

4.2.1、表数据

--创建表

```
CREATE TABLE `tb_contact` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `index_code` varchar(1) DEFAULT NULL COMMENT '索引编号',
  `surname` varchar(5) DEFAULT NULL COMMENT '姓',
  `name` varchar(10) DEFAULT NULL COMMENT '名',
  `mobile_code` varchar(11) DEFAULT NULL COMMENT '手机号',
  `created` datetime DEFAULT NULL,
  `updated` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_code` (`index_code`,`surname`,`name`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8;
```

--插入测试数据

```
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('1', 'Z', 'zhang', 'san', '13911111111', '2020-
09-01 15:33:21', '2020-09-01 15:33:15');
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('2', 'L', 'li', 'si', '13922222222', '2020-09-02
15:33:29', '2020-09-02 15:33:25');
```

```

INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('3', 'W', 'wang', 'wu', '13933333333', '2020-09-
03 15:33:37', '2020-09-03 15:33:34');
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('4', 'Z', 'zhao', 'liu', '13944444444', '2020-09-
04 15:33:43', '2020-09-04 15:33:40');
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('5', 'L', 'liu', 'hulan', '13955555555', '2020-
09-05 15:33:52', '2020-09-05 15:33:47');
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('6', 'L', 'lei', 'jun', '13966666666', '2020-09-
06 15:34:02', '2020-09-06 15:33:57');
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('7', 'M', 'ma', 'yun', '13977777777', '2020-09-07
15:34:12', '2020-09-07 15:34:08');
INSERT INTO `tb_contact` (`id`, `index_code`, `surname`, `name`, `mobile_code`,
`created`, `updated`) VALUES ('8', 'Q', 'qian', 'laoda', '13988888888', '2020-
09-08 15:34:18', '2020-09-08 15:34:15');

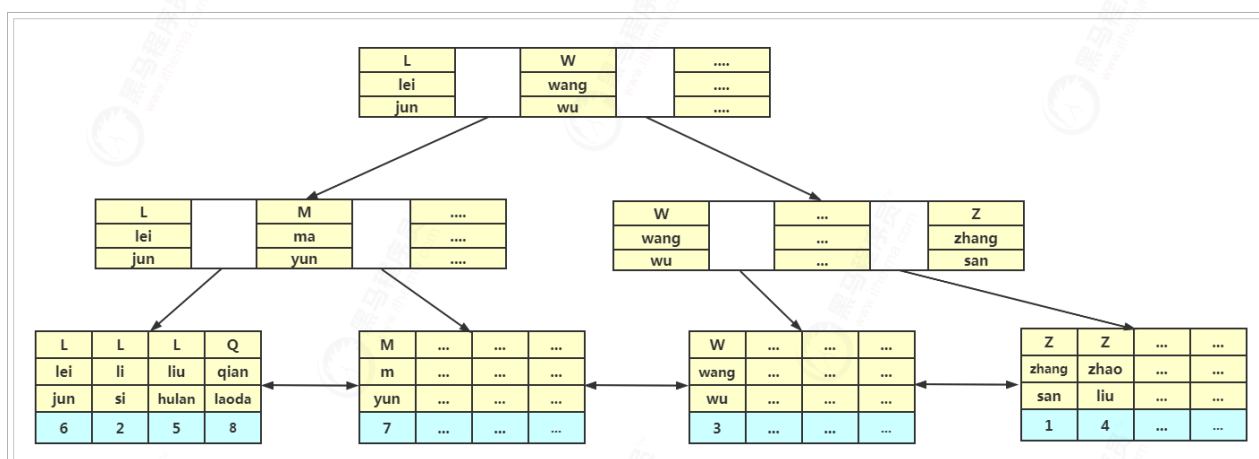
```

id	index_code	surname	name	mobile_code	updated	created
1	Z	zhang	san	13911111111	2020-09-01 15:33:15	2020-09-01 15:33:21
2	L	li	si	13922222222	2020-09-02 15:33:25	2020-09-02 15:33:29
3	W	wang	wu	13933333333	2020-09-03 15:33:34	2020-09-03 15:33:37
4	Z	zhao	liu	13944444444	2020-09-04 15:33:40	2020-09-04 15:33:43
5	L	liu	hulan	13955555555	2020-09-05 15:33:47	2020-09-05 15:33:52
6	L	lei	jun	13966666666	2020-09-06 15:33:57	2020-09-06 15:34:02
7	M	ma	yun	13977777777	2020-09-07 15:34:08	2020-09-07 15:34:12
8	Q	qian	laoda	13988888888	2020-09-08 15:34:15	2020-09-08 15:34:18

在联系人表中，id为主键，index_code,surname,name这三个字段建立了联合索引。

4.2.2、底层数据结构

联合索引的底层结构也是基于B+树的，其结构示意图如下：



InnoDB会使用主键索引在B+树维护索引和数据文件，然后我们创建了一个联合索引

(index_code,surname,name) 也会生成一个索引树，同样是B+树的结构，只不过它的data部分存储的是联合索引所在行的主键值。

对于联合索引来说只不过比单值索引多了几列，而这些索引列全都出现在索引树上。对于联合索引，存储引擎会首先根据第一个索引列排序，如上图第一个索引列，如：L、L、L、Q、M、W、Z、Z 是根据英文字母正序排序的；

如果第一列相等则再根据第二列排序，依次类推就构成了上图的索引树，如：L lei jun 、L li si、L liu hulan等。

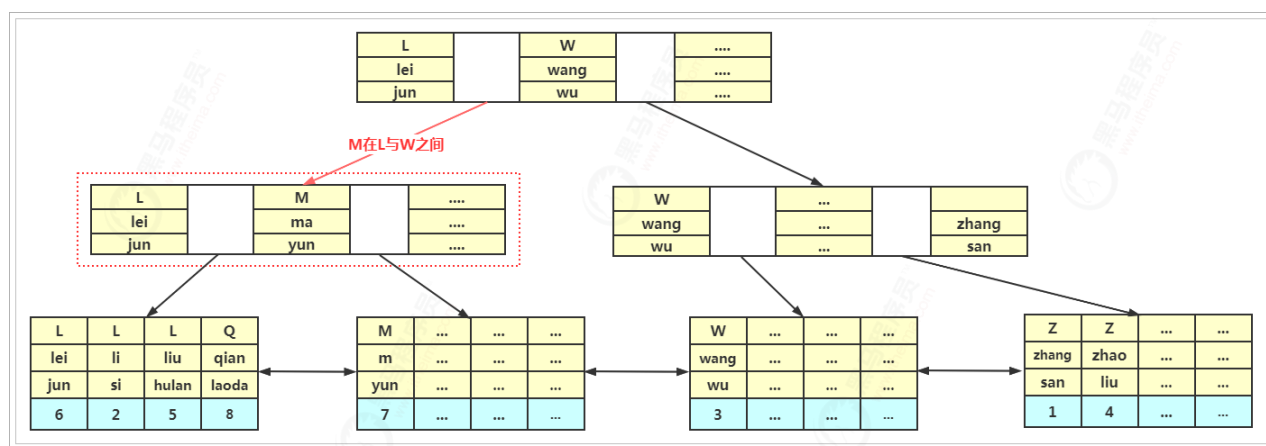
4.2.3、联合索引的查询

如果查询mayun用户的手机号码，需要执行的SQL为：

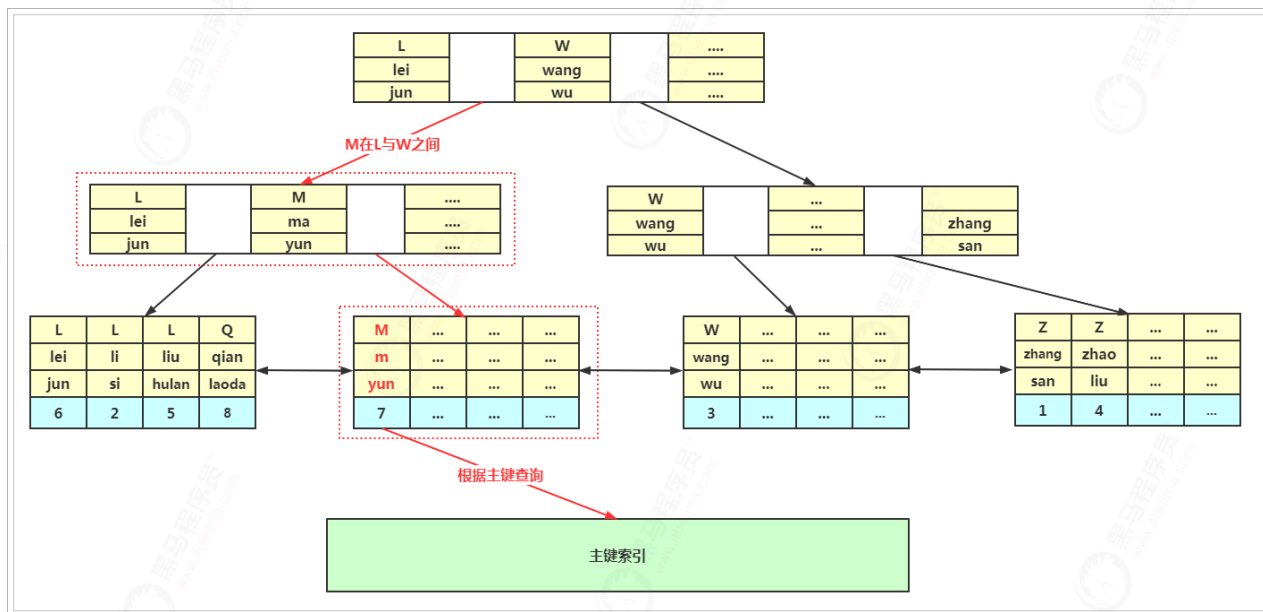
```
SELECT * FROM tb_contact WHERE index_code = 'M' AND surname = 'ma' AND name = 'yun'
```

联合索引的执行过程如下：

首先从根节点开始查找，根节点一般是常驻内存中的，第一个列为index_code，其值为：M，在L与W之间，会继续向子节点查询：



在查找到子节点后，将子节点数据从磁盘加载到内存中，采用二分法进行查找，找到M ma yun数据符合条件，再继续查找子节点，读取到子节点中的data数据，其数据就是这条记录的主键，然后再通过主键索引查询数据，最终将在主键索引中查询到数据：



4.2.4、最左前缀原则

在使用联合索引时，必须按照索引的顺序查询，例如：

```
SELECT * FROM tb_contact WHERE index_code = 'M' AND surname = 'ma' AND name = 'yun' --索引会生效
```

```
SELECT * FROM tb_contact WHERE index_code = 'M' AND surname = 'ma' --索引会生效
```

```
SELECT * FROM tb_contact WHERE index_code = 'M' --索引会生效
```

```
SELECT * FROM tb_contact WHERE surname = 'ma' AND name = 'yun' --索引不会生效
```

- 通过索引编号+姓+名就能定位到手机号，因为在索引结构中是排好序的
- 如果没有使用索引编号，那么整体来看就是混乱无序的，就无法使用排好序的索引，所以索引就不会生效

这就是最左前缀原则。

4.3、EXPLAIN

使用EXPLAIN可以查看SQL语句的执行计划，从而可以知道SQL的瓶颈在哪里，就可以有针对性的进行优化了。

用法：

```
EXPLAIN <SELECT语句>
```

--举例：

```
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'M'
```

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	6	const	1	100	(Null)

上图所展现出的就是其执行计划中的信息，根据这些信息进行SQL性能的判断。

在查询中的每个表会输出一行信息，如果有两个表通过 join 连接查询，那么会输出两行，表的含义比较广泛：可以是子查询、一个 union 结果等。

需要注意的是，在MySQL5.7.3以后的版本，EXPLAIN已经默认添加了EXTENDED参数，在结果中添加了filtered结果字段，filtered是指返回结果的行占需要读到的行(rows列的值)的百分比。紧跟着执行SHOW WARNINGS;语句查看优化后的查询语句。

```
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'M' AND surname = 'ma';
SHOW WARNINGS;
```

<

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_cc 24		const,cc 1		100	(Null)

信息 结果1 结果2 概况 状态

Level	Code	Message
Note	1003	/* select#1 */ select 'itcast','tb_contact'.'id' AS 'id','itcast','tb_contact'.'index_code' AS 'index_code','itcast','tb_contact'.'surname' AS 'surname','itcast','tb_contact'.'name' AS 'name','itcast','tb_

一般而言，优化的结果建议并不准确，仅作参考。

4.3.1、id列表

id列的编号是 select 的序列号，有几个 select 就有几个id，并且id的顺序是按 select 出现的顺序增长的。其值越大执行优先级越高，id相同则从上往下执行，id为NULL最后执行。

--子查询

```
EXPLAIN SELECT (SELECT 1 FROM tb_contact LIMIT 1) FROM tb_contact
```

结果：

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tb_contact	(Null)	index	(Null)	index_cc 57		(Null)	8	100	Using index
2	SUBQUERY	tb_contact	(Null)	index	(Null)	index_cc 57		(Null)	8	100	Using index

4.3.2、select_type

表示SELECT语句的类型。

有以下几种值：

1、**SIMPLE**：表示简单查询，其中不包含连接查询和子查询。

<

信息

结果1

结果2

概况

状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶ 1	SIMPLE	tb_contact	(Null)	ref	index_code	index_cc 24		const,cc 1		100	(Null)

2、**PRIMARY**：表示主查询，或者是最外面的查询语句。

1 EXPLAIN SELECT * FROM (SELECT id FROM tb_item WHERE price = 5288) AS tmp										
信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	PRIMARY	<derived2>	ALL	(Null)	(Null)	(Null)	(Null)	38	(Null)	
2	DERIVED	tb_item	ALL	(Null)	(Null)	(Null)	(Null)	38	Using whe	

3、**UNION**：表示连接查询的第2个或后面的查询语句。

1 EXPLAIN SELECT * FROM tb_order LIMIT 1 UNION SELECT * FROM tb_order LIMIT 2										
信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	PRIMARY	tb_order	ALL	(Null)	(Null)	(Null)	(Null)	2	(Null)	
2	UNION	tb_order	ALL	(Null)	(Null)	(Null)	(Null)	2	(Null)	
(Null)	UNION RESULT	<union1,2>	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	Using tem	

4、**DEPENDENT UNION**：UNION中的第二个或后面的SELECT语句，取决于外面的查询。

5、**UNION RESULT**：连接查询的结果。

6、**SUBQUERY**：子查询中的第1个SELECT语句。

1 EXPLAIN SELECT * FROM tb_order WHERE user_id = (SELECT id FROM tb_user WHERE name = '张三')										
信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	PRIMARY	tb_order	ref	FK_orders_1	FK_orde 8	const	1	1	Using whe	
2	SUBQUERY	tb_user	ALL	(Null)	(Null)	(Null)	(Null)	7	Using whe	

7、**DEPENDENT SUBQUERY**：子查询中的第1个SELECT语句，取决于外面的查询。

8、**DERIVED**：SELECT(FROM 子句的子查询)。

4.3.3、table

计划执行的表，当 from 子句中有子查询时，table列是<derivedN> 格式，表示当前查询依赖 id=N 的查询，于是先执行 id=N 的查询。当有 union 时，UNION RESULT 的 table 列的值为<union1,2>，1和2表示参与 union 的 select 行id。

4.3.4、type

这一列表示关联类型或访问类型，即MySQL决定如何查找表中的行，查找数据行记录的大概范围。

依次从最优到最差分别为：system > const > eq_ref > ref > range > index > ALL

- system

- 表仅有一行，这是const类型的特例，平时不会出现，这个也可以忽略不计。
- const
 - 数据表最多只有一个匹配行，因为只匹配一行数据，所以很快，常用于PRIMARY KEY或者UNIQUE索引的查询，可理解为const是最优化的。

查询创建工具 查询编辑器

```
1 EXPLAIN SELECT * FROM tb_order WHERE id = 1
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_order	const	PRIMARY	PRIMAR 4		const	1	(Null)

- eq_ref
 - primary key 或 unique key 索引的所有部分被连接使用，最多只会返回一条符合条件的记录。这可能是在 const 之外最好的联接类型了，简单的 select 查询不会出现这种 type。

1 EXPLAIN SELECT * FROM tb_order o,tb_user u WHERE u.id = o.user_id

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	o	ALL	FK_orders_1	(Null)	(Null)	(Null)	2	(Null)
1	SIMPLE	u	eq_ref	PRIMARY	PRIMAR 8		mybat 1		(Null)

- ref
 - 相比 eq_ref，不使用唯一索引，而是使用普通索引或者唯一性索引的部分前缀，索引要和某个值相比较，可能会找到多个符合条件的行。

1 EXPLAIN SELECT * FROM tb_order WHERE order_number = '20140921001'

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_order	ref	order_number	order_nt 62		const	1	Using index

- range
 - 范围扫描通常出现在 in(), between ,> ,< ,>= 等操作中。使用一个索引来检索给定范围的行。

1 EXPLAIN SELECT * FROM tb_user WHERE age < 10

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_user	range	age	age	5	(Null)	1	Using index

- index
 - 和ALL一样，不同就是mysql只需扫描索引树，这通常比ALL快一些。
- ALL
 - 即全表扫描，意味着mysql需要从头到尾去查找所需要的行。通常情况下这需要增加索引来进行优化了

4.3.5、possible_keys

这一列显示查询可能使用哪些索引来查找。

explain 时可能出现 possible_keys 有列，而 key 显示 NULL 的情况，这种情况是因为表中数据不多，mysql认为索引对此查询帮助不大，选择了全表查询。

如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查 where 子句看是否可以创建一个适当的索引来提高查询性能，然后用 explain 查看效果。

4.3.6、key

显示MySQL实际决定使用的键(索引)。如果没有选择索引,键是NULL。

可以强制使用索引或者忽略索引：

```
1 EXPLAIN SELECT * FROM tb_user IGNORE INDEX (age) WHERE age < 10
2 EXPLAIN SELECT * FROM tb_user USE INDEX (age) WHERE age < 10
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_user	range	age	age	5	(Null)	1	Using index condition

4.3.7、key_len

这一列显示了mysql在索引里使用的字节数，通过这个值可以算出具体使用了索引中的哪些列。

key_len计算规则如下：

- 字符串
 - char(n)：n字节长度
 - varchar(n)：2字节存储字符串长度，如果是utf-8，则长度 $3n + 2$
- 数值类型
 - tinyint：1字节
 - smallint：2字节
 - int：4字节
 - bigint：8字节
- 时间类型
 - date：3字节
 - timestamp：4字节
 - datetime：8字节
- 如果字段允许为 NULL，需要1字节记录是否为 NULL

```
1 EXPLAIN SELECT * FROM tb_contact WHERE id = 1
```

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	const	PRIMARY	PRIMARY	8	const	1	100	(Null)

1 EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'Z' AND surname='zhang'

计算: $(3*1+2)+(3*5+2)+1+1=24$

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	24	const,const	1	100	(Null)

索引最大长度是768字节，当字符串过长时，mysql会做一个类似左前缀索引的处理，将前半部分的字符提取出来做索引。

4.3.8、ref

这一列显示了在key列记录的索引中，表查找值所用到的列或常量，常见的有：const（常量），func，NULL，字段名（例：film.id）

4.3.9、rows

这一列是mysql估计要读取并检测的行数，注意这个不是结果集里的行数。

4.3.10、Extra

这一列展示的是额外信息。常见的重要值如下：

- Distinct：MySQL发现第1个匹配行后,停止为当前的行组合搜索更多的行。
- Not exists：MySQL能够对查询进行LEFT JOIN优化,发现1个匹配LEFT JOIN标准的行后,不再为前面的的行组合在该表内检查更多的行。
- range checked for each record (index map: #)：MySQL没有发现好的可以使用的索引,但发现如果来自前面的表的列值已知,可能部分索引可以使用。
- Using filesort（重点）：MySQL会对结果使用一个外部索引排序，而不是按索引次序从表里读取行。此时mysql会根据联接类型浏览所有符合条件的记录，并保存排序关键字和行指针，然后排序关键字并按顺序检索行信息。这种情况下一般也是要考虑使用索引来优化的。

1 EXPLAIN SELECT id,name FROM tb_contact WHERE index_code = 'Z' ORDER BY mobile_code

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	6	const	2	100	Using index condition; Using filesort

1 EXPLAIN SELECT id,name FROM tb_contact WHERE index_code = 'Z' ORDER BY surname

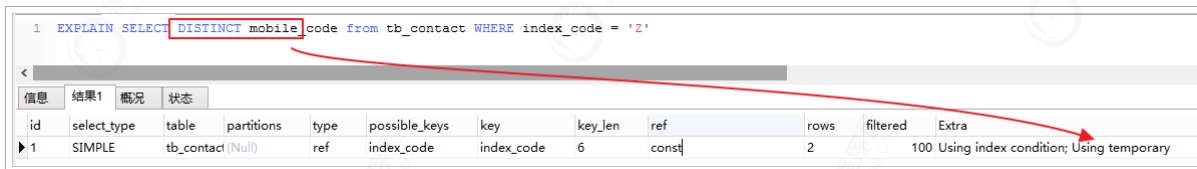
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	6	const	2	100	Using where; Using index

- Using index（重点）：从只使用索引树中的信息而不需要进一步搜索读取实际的行来检索表中的列信息。

1 EXPLAIN SELECT id,name FROM tb_contact WHERE index_code = 'Z' AND surname='zhang'

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	24	const,const	1	100	Using index

- Using temporary（重点）：MySQL需要创建一张临时表来处理查询。出现这种情况一般是要进行优化的，首先是想到用索引来优化。



```
1 EXPLAIN SELECT DISTINCT mobile code from tb_contact WHERE index_code = '2'
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	6	const	2	100	Using index condition; Using temporary

- Using where：MySQL服务器将在存储引擎检索后再进行过滤。就是先读取整行数据，再按 where 条件进行检查，符合就留下，不符合就丢弃。
- Using index condition：与Using where类似，查询的列不完全被索引覆盖，where条件中是一个前导列的范围；
- Using sort_union(...), Using union(...), Using intersect(...)：这些函数说明如何为index_merge联接类型合并索引扫描。
- Using index for group-by：类似于访问表的Using index方式,Using index for group-by表示MySQL发现了一个索引,可以用来查询GROUP BY或DISTINCT查询的所有列,而不要额外搜索硬盘访问实际的表。
- NULL：查询的列未被索引覆盖，并且where筛选条件是索引的前导列，意味着用到了索引，但是部分字段未被索引覆盖，必须通过“回表”来实现，不是纯粹地用到了索引，也不是完全没用到索引

4.4、覆盖索引与回表查询

- 什么是回表查询？
 - 先定位主键值，再定位行记录，它的性能较扫一遍索引树更低，这就是回表查询。
- 什么是覆盖索引？
 - 只需要在一棵索引树上就能获取SQL所需的所有列数据，无需回表，速度更快。

举例：

```
EXPLAIN SELECT * from tb_contact WHERE index_code = 'L'; -- 回表查询
EXPLAIN SELECT id,name from tb_contact WHERE index_code = 'L'; -- 覆盖索引
```

需要回表查询：

信息	结果1	结果2	概况	状态							
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_cc 6		const	3	100	(Null)

using index：使用覆盖索引的时候就会出现

信息	结果1	结果2	概况	状态							
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_cc 6		const	3	100	Using index

using index condition：查找使用了索引，但是需要回表查询数据

1 EXPLAIN SELECT id,name FROM tb_contact WHERE index_code = 'Z' ORDER BY mobile_code												
信息	结果1	概况	状态									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	tb_contact (Null)		ref	index_code	index_code	6	const	2	100	Using index condition; Using filesort	

using index & using where: 查找使用了索引，但是需要的数据都在索引列中能找到，所以不需要回表查询数据

1 EXPLAIN SELECT id,name FROM tb_contact WHERE index_code = 'Z' ORDER BY surname												
信息	结果1	概况	状态									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	tb_contact (Null)		ref	index_code	index_code	6	const	2	100	Using where; Using index	

4.5、实例

4.5.1、实例1

实例1：联合索引中，不能范围条件右侧的索引列

```
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'L' AND surname = 'lei' AND name = 'jun';
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'L' AND surname > 'lei' AND name = 'jun';
```

信息	结果1	结果2	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	tb_contact (Null)		ref	index_code	index_code	57	const,cc 1	100	(Null)		

从索引长度来看，3个索引都已经生效， $(3*1+2) + (3*5+2) + (3*10+2) + 1 + 1 + 1 = 57$

信息	结果1	结果2	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	tb_contact (Null)		range	index_code	index_code	24	(Null)	2	12.5	Using index condition	

从索引长度来看，只生效了前2个索引， $(3*1+2) + (3*5+2) + 1 + 1 = 24$

原因：在底层的B+tree结构中，在已确定第一个索引的情况下，第二个索引是有序存储的，那么按照第二个索引的范围查找，就不需要第三个索引的参与了。

4.5.2、实例2

实例2：索引条件使用<>时，索引会失效

```
EXPLAIN SELECT * FROM tb_contact WHERE index_code <> 'L'
```

信息	结果1	概况	状态									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	tb_contact (Null)		ALL	index_code	(Null)	(Null)	(Null)	8	75	Using where	

可以看到，查询类型为ALL，也就是进行了全表扫描，因为不等于没有办法使用索引结构进行查询。

4.5.3、实例3

实例3：Impossible WHERE

```
EXPLAIN SELECT id FROM tb_contact WHERE id IS NULL
```

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Impossible WHERE

id本身就不会为null，所以条件为IS NULL是不可能发生的条件，因此会出现Impossible WHERE，不要误以为是索引不生效。

4.5.4、实例4

实例4：like查询

```
EXPLAIN SELECT * FROM tb_contact WHERE index_code LIKE '%L'; -- 索引不生效
EXPLAIN SELECT * FROM tb_contact WHERE index_code LIKE 'L%'; -- 索引生效
EXPLAIN SELECT index_code,surname,name FROM tb_contact WHERE index_code LIKE 'L%'; -- 使用覆盖索引优化
```

信息	结果1	结果2	结果3	概况	状态						
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶ 1	SIMPLE	tb_contact	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	8	12.5	Using where

信息	结果1	结果2	结果3	概况	状态						
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	range	index_code	index_code	6	(Null)	3	100	Using index condition

信息	结果1	结果2	结果3	概况	状态						
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact (Null)		index	(Null)	index_cc57		(Null)	8	12.5	Using where; Using index

4.5.5、实例5

实例5：GROUP BY 索引使用

```
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'L' ORDER BY surname; -- 使用index_code索引查询，surname索引进行排序
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'L' ORDER BY name; -- 使用index_code索引查询，由于跳过了surname针对name排序，出现了Using filesort
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'L' ORDER BY surname,name; -- 使用index_code索引查询，surname,name索引进行排序
EXPLAIN SELECT * FROM tb_contact WHERE index_code = 'L' ORDER BY name,surname; -- 使用index_code索引查询，name,surname的顺序与索引顺序不一致，导致了Using filesort
```

信息	结果1	结果2	结果3	结果4	概况	状态					
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	6	const	3	100	Using index condition

信息	结果1	结果2	结果3	结果4	概况	状态					
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_code	6	const	3	100	Using index condition; Using filesort

信息	结果1	结果2	结果3	结果4	概况	状态					
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_cc 6		const	3	100	Using index condition

信息	结果1	结果2	结果3	结果4	概况	状态					
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_contact	(Null)	ref	index_code	index_cc 6		const	3	100	Using index condition; Using filesort

总结：

- MySQL支持两种方式的排序filesort和index，Using index是指MySQL扫描索引本身完成排序。index效率高，filesort效率低。
- order by满足两种情况会使用Using index
 - order by语句使用索引最左前列。
 - 使用where子句与order by子句条件列组合满足索引最左前列。
- 尽量在索引列上完成排序，遵循索引建立（索引创建的顺序）时的最左前缀原则。
- 如果order by的条件不在索引列上，就会产生Using filesort。