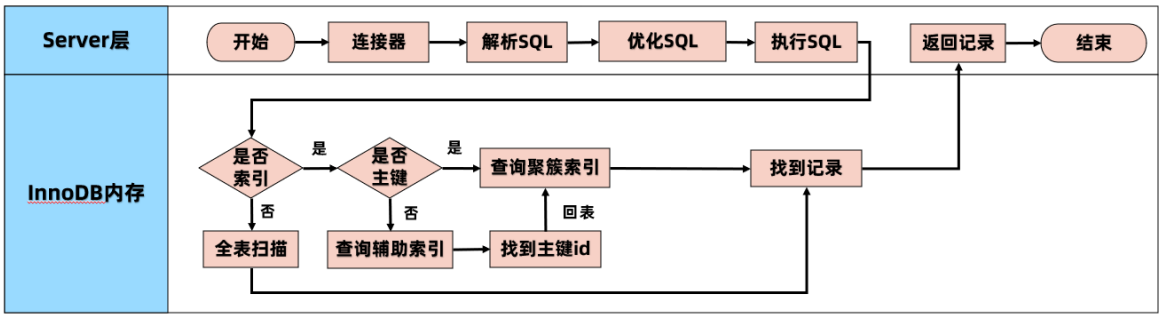
MySQL索引篇

1. 一条Select语句

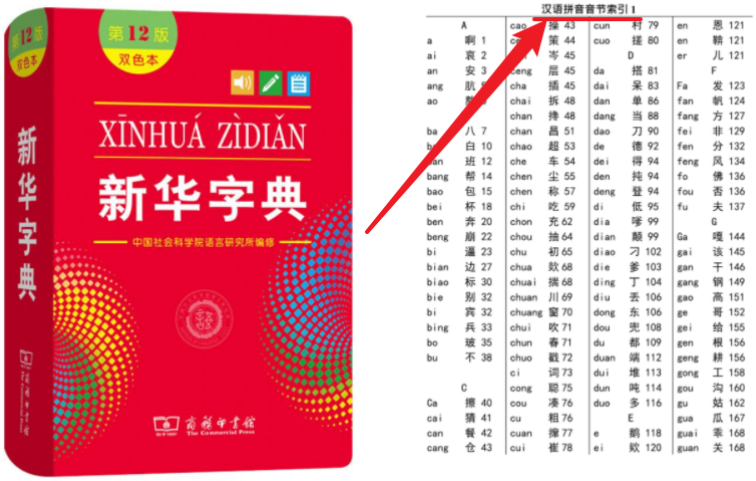
```
select * from tab_user WHERE id=1
```

执行流程：



2. MySQL索引简介

2.1 什么是索引?



官方介绍索引是帮助MySQL**高效获取数据**的**数据结构。**更通俗的说，数据库索引好比是一本书前面的目录，能**加快数据库的查询速度。**

一般来说索引本身也很大，不可能全部存储在内存中，因此**索引往往是存储在磁盘上的文件中的**（可能存储在单独的索引文件中，也可能和数据一起存储在数据文件中）。

我们通常所说的索引，包括聚簇索引、覆盖索引、组合索引、前缀索引、唯一索引等，没有特别说明，默认都是使用B+树结构组织的索引。

2.2 优势和劣势

优势：

- **可以提高数据检索的效率，降低数据库的IO成本**，类似于书的目录。
- 通过**索引列对数据进行排序**，降低数据排序的成本，降低了CPU的消耗。
    - 被索引的列会自动进行排序，包括【单列索引】和【组合索引】，只是组合索引的排序要复杂一些。
    - 如果按照索引列的顺序进行排序，对应order by语句来说，效率就会提高很多。

劣势：

- **索引会占据磁盘空间**
- **索引虽然会提高查询效率，但是会降低更新表的效率。**比如每次对表进行增删改操作，MySQL不仅要保存数据，还要维护索引文件。

2.3 不用索引行不行?

- 行不行？完全可以
- 时间复杂度O(n)

> 用不用的选择权在谁手里?

3. 索引的使用

3.1 索引的类型

按照索引列的数量分类：

- **单列索引**：索引中只有一个列。
- **组合索引**：使用2个以上的字段创建的索引。

3.1.1 单列索引

- 主键索引：索引列中的值必须是唯一的不允许有空值。

```
ALTER TABLE table_name ADD PRIMARY KEY (column_name);
```

- 普通索引：MySQL中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值。

```
ALTER TABLE table_name ADD INDEX index_name (column_name);
```

- 唯一索引：索引列中的值必须是唯一的，但是允许为空值。

```
CREATE UNIQUE INDEX index_name ON table(column_name);
```

- 全文索引：只能在文本类型CHAR，VARCHAR，TEXT类型字段上创建全文索引。字段长度比较大时，如果创建普通索引，在进行like模糊查询时效率比较低，这时可以创建全文索引。MyISAM和InnoDB中都可以使用全文索引。

- InnoDB全文索引，官网介绍：https://dev.mysql.com/doc/refman/5.7/en/innodb-fulltext-index.html
    - 全文搜索时候，全文索引一般很少使用，数据量比较少或者并发度低的时候可以用。但是数据量大或者并发度高的时候一般是用专业的工具Lucene，ES，Solr

```
#创建表时，创建全文索引
CREATE TABLE `t_fulltext` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `content` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

#创建全文索引
ALTER TABLE `t_fulltext` ADD FULLTEXT INDEX `idx_content`(`content`);

INSERT INTO `t_fulltext` (`id`,`content`) VALUES ('1','Mention');
INSERT INTO `t_fulltext` (`id`,`content`) VALUES ('2','Vincent hero man');
INSERT INTO `t_fulltext` (`id`,`content`) VALUES ('3','Benson');
INSERT INTO `t_fulltext` (`id`,`content`) VALUES ('4','Carol');
INSERT INTO `t_fulltext` (`id`,`content`) VALUES ('5','yilia');
INSERT INTO `t_fulltext` (`id`,`content`) VALUES ('6','lock and lock');
```

可以使用MATCH() ... AGAINST语法执行全文搜索。

```
SELECT * FROM t_fulltext WHERE MATCH(content) AGAINST('Vincent');
```

- 空间索引：MySQL在5.7之后的版本支持了空间索引，而且支持OpenGIS几何数据模型。MySQL在空间索引这方面遵循OpenGIS几何数据模型规则。（本课程中不做过多介绍）

  参考资料：MySQL中 OpenGIS Geometry Model

  空间数据类型
- 前缀索引：在文本类型如CHAR，VARCHAR，TEXT类列上创建索引时，可以指定索引列的长度，但是数值类型不能指定。

  ```
  ALTER TABLE table_name ADD INDEX index_name (column1(length));
  ```

3.1.2 组合索引

- 组合索引的使用，需要遵循**最左前缀原则（最左匹配原则，后面详细讲解）**。
- 一般情况下，**建议使用组合索引代替单列索引**（主键索引除外，具体原因后面讲解）。

  ```
  ALTER TABLE table_name ADD INDEX index_name (column1,column2);
  ```

3.2 删除索引

```
DROP INDEX index_name ON table
```

3.3 查看索引

```
SHOW INDEX FROM table_name
```

4. 索引的数据结构

4.1 索引基本需求

索引的数据结构，至少需要支持两种最常用的查询需求：

1. 等值查询：根据某个值查找数据，比如：`select * from t_user where age=76;`
2. 范围查询：根据某个范围区间查找数据，比如：`select * from t_user where age>=76 and age<=86;`
3. 排序
4. 分组
5. ..

同时需要考虑时间和空间因素：**性价比高**

- 在执行时间方面，我们希望通过索引，查询数据的时间尽可能小；
- 在存储空间方面，我们希望索引不要消耗太多的内存空间和磁盘空间。

4.2 索引应该使用什么数据结构?

常用的数据结构：Hash表，二叉树，平衡二叉查找树（红黑树是一个近似平衡二叉树），B树，B+树。

> 数据结构示例网站：可以通过动画看到操作过程，非常好的一个网站。https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

4.2.1 Hash表

Hash表，常见的数据结构之一。

我们使用Hash表存储表数据Key可以存储索引列，Value可以存储行记录或者行磁盘地址。Hash表在等值查询时效率很高，时间复杂度为O(1)；

- 但是不支持范围快速查找，范围查找时还是只能通过扫描全表方式。
- 数据结构比较稀疏，不适合做聚合，不适合做范围等查找。

使用场景：

- 对查询并发要求很高，**K/V内存数据库，缓存**

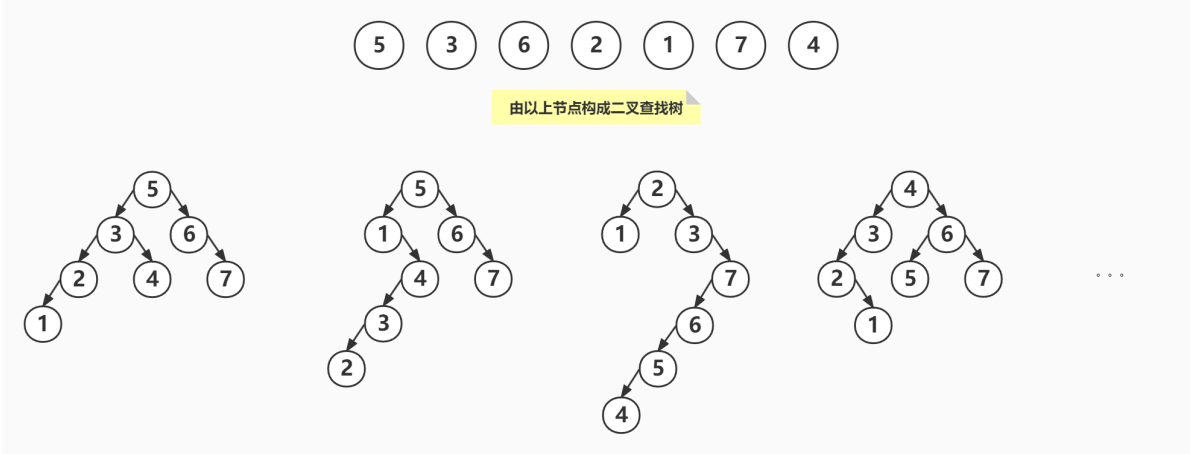
4.2.2 二叉查找树

- 二叉树特点：每个节点最多有2个分叉，左子树和右子树数据顺序左小右大。
- 二叉树的检索复杂度和树高相关：**理想状态**下效率可以达到O(logn)

是不是任何列使用二叉树效率都会提升呢？答案是否定的。

极端情况下，二叉查找树会构建成为单向链表 = 查找全表扫描。

对磁盘不友好【一旦变成了全表扫描，磁盘io将是极其沉重】



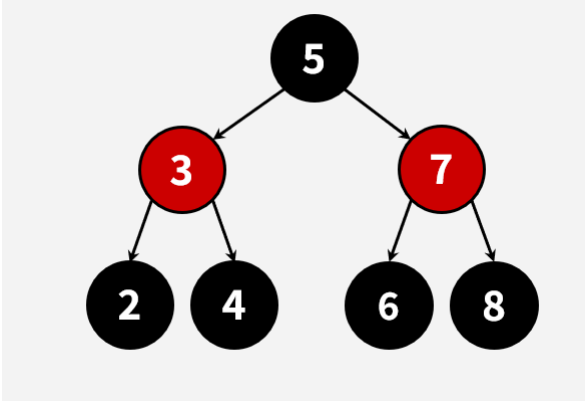
4.2.3 红黑树

> 红黑树是一个近似平衡的二叉树

平衡二叉树是采用二分法思维，平衡二叉查找树除了具备二叉树的特点，最主要的特征是树的左右两个子树的层级**最多相差1。**在插入删除数据时通过左旋/右旋操作保持二叉树的平衡，不会出现左子树很高、右子树很矮的情况。

使用平衡二叉查找树查询的性能接近于二分查找法，时间复杂度是 O(log2n)。

> unique key 为什么不用红黑树，反正只存一个主键?



平衡二叉树存在的问题

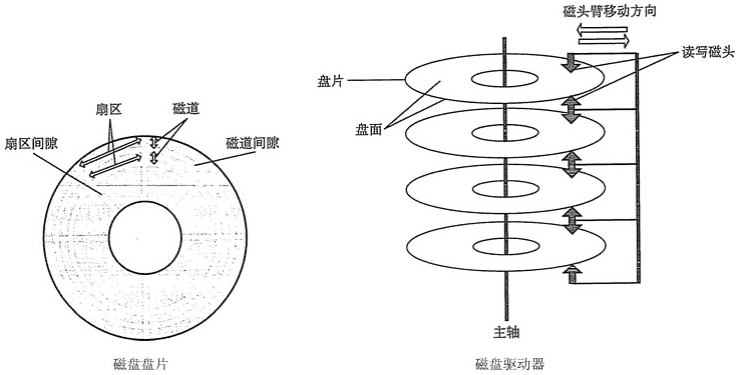
1. 时间复杂度和树高相关：树有多高就需要检索多少次，每个节点的读取，都对应一次磁盘 IO 操作【瓶颈】。
    - **磁盘每次寻道时间为10ms**，在表数据量大时，对响应时间要求高的场景下，查询性能就会出现瓶颈。
    - 举例：1百万的数据量，log2n约等于20次磁盘IO，时间20*10=0.2s
2. 平衡二叉树不支持范围查询快速查找，范围查询时需要从根节点多次遍历，查询效率极差。
3. 数据量大的情况下，索引存储空间占用巨大

举个栗子：

- 10亿行数据，时间复杂度O(logn)，最多不超过30次查到数据
- 最简单索引构成：<ID，行号，指针>
- 假如key为bigint=8字节，每个节点有两个指针，每个指针为4个字节，一个节点占用的空间16个字节（8+4*2=16）。
- 索引大小：10亿x16(bigint)=15GB

为什么磁盘IO操作就慢？我举个栗子

- 从磁盘读取数据时，系统会将逻辑地址发给磁盘，磁盘将逻辑地址转换为物理地址（哪个磁道，哪个扇区）。 磁头进行机械运动，先找到相应磁道，再找该磁道的对应扇区，扇区是磁盘的最小存储单元。



- 随机读写时，磁头需要不停的移动，时间都浪费在了磁头寻址上。 而在实际的磁盘存储里，是很少顺序存储的，因为这样的维护成本会很高。
- 性能差异：机械硬盘的连续读写性能很好，但随机读写性能很差。
    - 顺序访问：**内存访问速度**是硬盘访问速度的6~7倍
    - 随机访问：**内存访问速度**就要比硬盘访问速度快上**10万倍以上**

如何减少IO操作次数呢？如何才能降低存储空间呢？

4.2.4 B树：改进二叉树，为多叉树

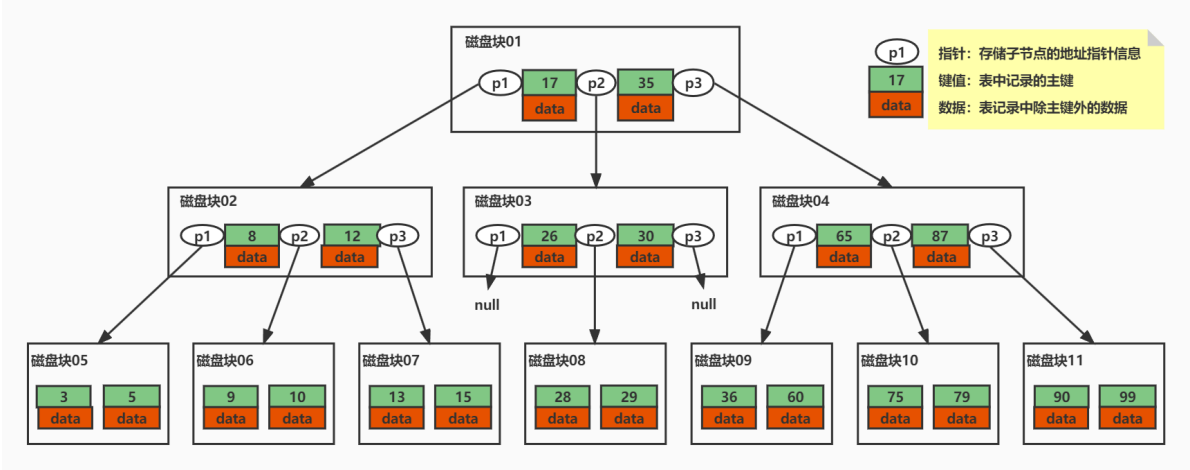
多想要减少耗时的IO操作，就要尽量降低树的高度。每个节点存储多个元素，在每个节点尽可能多的存储数据。每个节点可以存储1000个索引（16k/16=1000），这样就将二叉树改造成了多叉树，通过增加树的叉树，将树从高瘦变为矮胖。

举例：构建1百万条数据，树的高度只需要2层就可以（1000*1000=1百万），也就是说只需要2次磁盘IO就可以查询到数据。磁盘IO次数变少了，查询数据的效率也就提高了。

主要特点：

1. B树的节点中存储着多个元素，每个内节点有多个分叉。
2. 节点中的元素包含键值和数据，节点中的键值从大到小排列。也就是说，在所有的节点都储存数据。
3. 父节点当中的元素不会出现在子节点中。
4. 所有的叶子结点都位于同一层，叶节点具有相同的深度，叶节点之间没有指针连接。

以下面的B树为例，我们的键值为表主键，具备唯一性。



B树如何查询数据？：假如我们查询值等于15的数据。查询路径磁盘块1->磁盘块2->磁盘块7。

优点：

- 磁盘IO次数会大大减少。
- 比较是在内存中进行的，比较的耗时可以忽略不计。
- B树的高度一般2至3层就能满足大部分的应用场景，所以使用B树构建索引可以很好的提升查询的效率。

缺点：

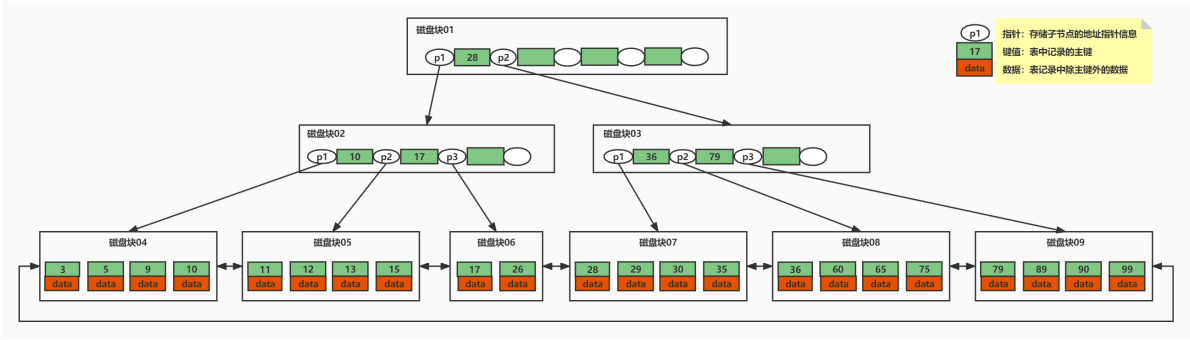
- **B树不支持范围查询的快速查找：**如果我们想要查找15和26之间的数据，查找到15之后，需要回到根节点重新遍历查找，需要从根节点进行多次遍历，查询效率有待提高。
- **空间占用较大：**如果data存储的是行记录，行的大小随着列数的增多，所占空间会变大。一个页中可存储的数据量就会变少，树相应就会变高，磁盘IO次数就会变大。

4.2.5 B+树：改进B树，非叶子节点不存储数据

在B树基础上，MySQL在B树的基础上继续改造，使用B+树构建索引。B+树和B树最主要的区别在于**非叶子节点是否存储数据**的问题

- B树：非叶子节点和叶子节点都会存储数据。
- B+树：只有叶子节点才会存储数据，非叶子节点只存储键值。叶子节点之间使用双向指针连接，最底层的叶子节点形成了一个双向有序链表。

B+树的最底层叶子节点包含所有索引项。具备中路返回特性



等值查询：假如我们查询值等于15的数据。查询路径磁盘块1->磁盘块2->磁盘块5。

范围查询：假如我们想要查找15和26之间的数据。

- 查找路径是磁盘块1->磁盘块2->磁盘块5。
- 首先查找值等于15的数据，将值等于15的数据缓存到结果集【三次磁盘IO】。
- 查找到15之后，底层的叶子节点是一个有序列表，我们从磁盘块5，键值15开始向后遍历筛选所有符合筛选条件的数据。
- 第四次磁盘IO：根据磁盘5后继指针到磁盘中寻址定位到磁盘块6，将磁盘6加载到内存中，在内存中从头遍历比较，15<17<26，15<26<=26，将data缓存到结果集。

优点：

- 继承了B树的优点【多叉树的优点】
- 保证等值和范围查询的快速查找
- MySQL的索引就采用了B+树的数据结构。

5. 存储引擎的索引案例

5.1 MyISAM索引

我们以t_user_myisam为例，来说明。t_user_myisam的id列为主键，age列为普通索引。

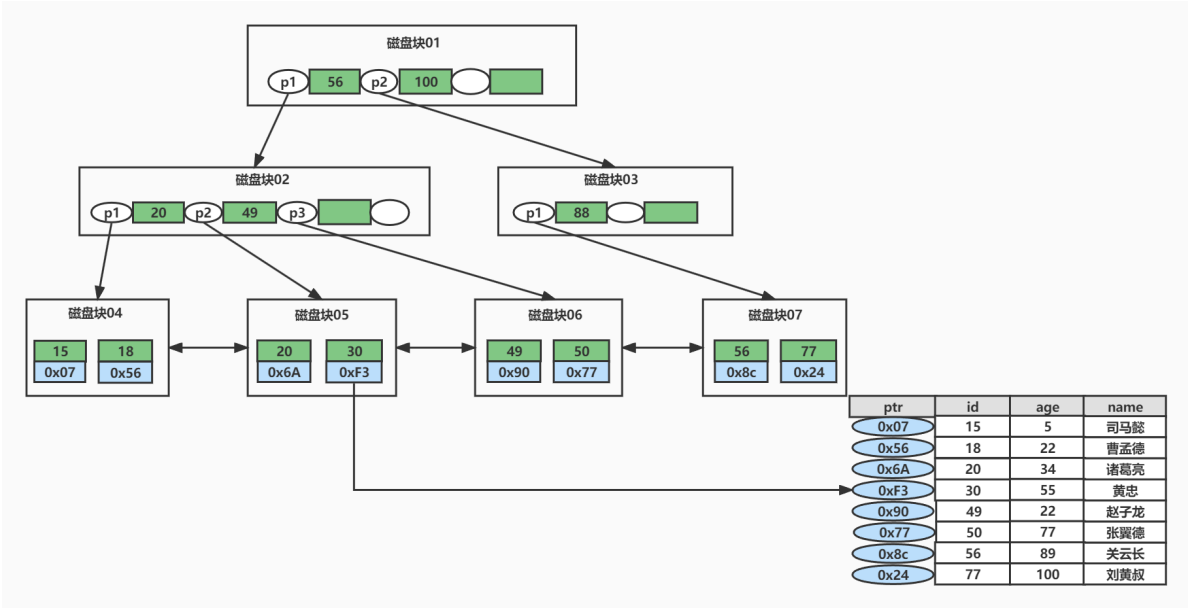
```
CREATE TABLE `t_user_myisam` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(20) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE,
  KEY `idx_age` (`age`) USING BTREE
) ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

insert into t_user_myisam values(15,'Nick',5);
insert into t_user_myisam values(18,'Zero',22);
insert into t_user_myisam values(20,'Tom',34);
```

```sql
insert into t_user_myisam values(30,'Nick',55);
insert into t_user_myisam values(49,'Mary',22);
insert into t_user_myisam values(50,'James',77);
insert into t_user_myisam values(56,'John',89);
insert into t_user_myisam values(77,'Lily',100);
```

```
mysql> select * from t_user_myisam;
+----+----------+------+
| id | username | age  |
+----+----------+------+
| 15 | Nick     |    5 |
| 18 | Zero     |   22 |
| 20 | Tom      |   34 |
| 30 | Nick     |   55 |
| 49 | Mary     |   22 |
| 50 | James    |   77 |
| 56 | John     |   89 |
| 77 | Lily     |  100 |
+----+----------+------+
8 rows in set (0.00 sec)
```

MyISAM的数据文件和索引文件是分开存储的。MyISAM使用B+树构建索引树时，叶子节点中存储的键值为索引列的值，数据为索引所在行的磁盘地址。

5.1.1 主键索引



表t_user_myisam的索引存储在索引文件t_user_myisam.MYI中，数据文件存储在数据文件t_user_myisam.MYD中。

1）等值查询数据

```
select * from t_user_myisam where id=30;
```

1. 先在主键树中从根节点开始检索，将根节点加载到内存，比较30<56，走左路。（1次磁盘IO）
2. 将左子树节点加载到内存中，比较20<30<49，向下检索。（1次磁盘IO）
3. 检索到叶节点，将节点加载到内存中遍历，比较20<30，30=30。查找到值等于30的索引项。（1次磁盘IO）
4. 从索引项中获取磁盘地址，然后到数据文件t_user_myisam.MYD中获取对应整行记录。（1次磁盘IO）
5. 将记录返给客户端。

磁盘IO次数：3+1次。

2）范围查询数据

```
select * from t_user_myisam where id between 30 and 49;
```

1. 先在主键树中从根节点开始检索，将根节点加载到内存，比较30<56，走左路。（1次磁盘IO）
2. 将左子树节点加载到内存中，比较20<30<49，向下检索。（1次磁盘IO）
3. 检索到叶节点，将节点加载到内存中遍历比较20<30，30<=30<49。查找到值等于30的索引项。
    1. 根据磁盘地址从数据文件中获取行记录缓存到结果集中。（2次磁盘IO）
    2. 我们的查询语句时范围查找，需要向后遍历底层叶子链表，直至到达最后一个不满足筛选条件。
4. 向后遍历底层叶子链表，将下一个节点加载到内存中，遍历比较，30<49<=49，根据磁盘地址从数据文件中获取行记录缓存到结果集中。（2次磁盘IO）
5. 最后得到两条符合筛选条件，将查询结果集返给客户端。

磁盘IO次数：2+检索叶子节点数量+记录数。

> MyISAM在查询时，会将索引节点缓存在MySQL缓存中，而数据缓存依赖于操作系统自身的缓存。

5.1.2 辅助索引

在 MyISAM 中，辅助索引和主键索引的结构是一样的，没有任何区别，叶子节点的数据存储的都是行记录的磁盘地址。只是主键索引的键值是唯一的，而辅助索引的键值可以重复。

查询数据时，由于辅助索引的键值不唯一，可能存在多个拥有相同的记录，所以即使是等值查询，也需要按照范围查询的方式在辅助索引树中检索数据。

5.2 InnoDB索引

5.2.1 InnoDB索引简介

每个InnoDB表都有一个**聚簇索引**，也叫聚集索引。聚簇索引使用B+树构建，叶子节点存储的数据是整行记录。一般情况下，聚簇索引等同于主键索引，当一个表没有创建主键索引时，InnoDB会自动创建一个ROWID字段来构建聚簇索引。

除聚簇索引之外的所有索引都称为辅助索引。在中InnoDB，辅助索引中的叶子节点存储的数据都是该行的主键值。 在检索时，InnoDB使用此主键值在聚簇索引中搜索行记录。

InnoDB创建索引的具体规则如下：

1. 在表上定义主键PRIMARY KEY，InnoDB将主键索引用作聚簇索引。
2. 如果表没有定义主键，InnoDB会选择第一个不为NULL的唯一索引列用作聚簇索引。
3. 如果以上两个都没有，InnoDB 会使用一个6 字节长整型的隐式字段 ROWID字段构建聚簇索引。该ROWID字段会在插入新行时自动递增。

下面我们一起来看一看这两种索引的实现。

我们以t_user_innodb为例，来说明。t_user_innodb的id列为主键，age列为普通索引。

t_user_innodb的表结构和数据与MyISAM引擎表t_user_myisam完全一致。

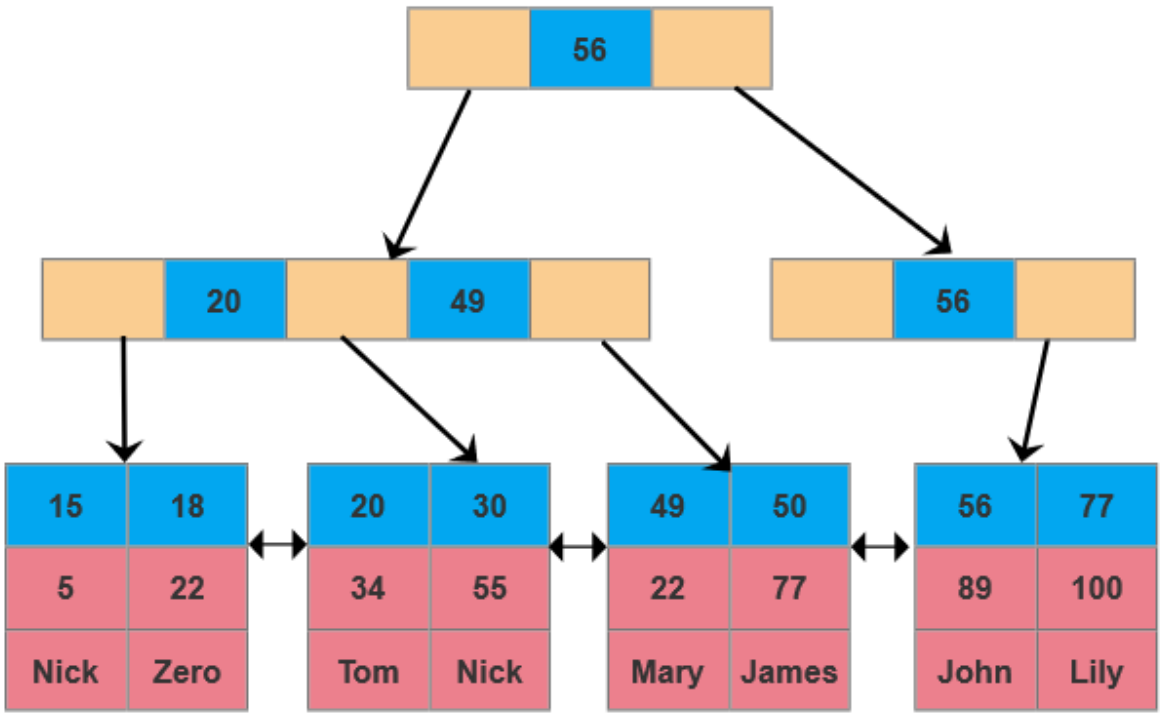
```
CREATE TABLE `t_user_innodb` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(20) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE,
  KEY `idx_age` (`age`) USING BTREE
) ENGINE=InnoDB;
insert into t_user_innodb values(15,'Nick',5);
insert into t_user_innodb values(18,'Zero',22);
insert into t_user_innodb values(20,'Tom',34);
insert into t_user_innodb values(30,'Nick',55);
insert into t_user_innodb values(49,'Mary',22);
insert into t_user_innodb values(50,'James',77);
insert into t_user_innodb values(56,'John',89);
insert into t_user_innodb values(77,'Lily',100);
```

```
mysql> select * from t_user_innodb;
+----+----------+------+
| id | username | age  |
+----+----------+------+
| 15 | Nick     |    5 |
| 18 | Zero     |   22 |
| 20 | Tom      |   34 |
| 30 | Nick     |   55 |
| 49 | Mary     |   22 |
| 50 | James    |   77 |
| 56 | John     |   89 |
| 77 | Lily     |  100 |
+----+----------+------+
8 rows in set (0.00 sec)
```

InnoDB的数据和索引存储在一个文件t_user_innodb.ibd中。InnoDB的数据组织方式是聚簇索引。

5.2.2 主键索引

- 主键索引的叶子节点会存储数据行，辅助索引只会存储主键值。
- InnoDB要求表必须有一个主键索引(MyISAM 可以没有)。



1）等值查询

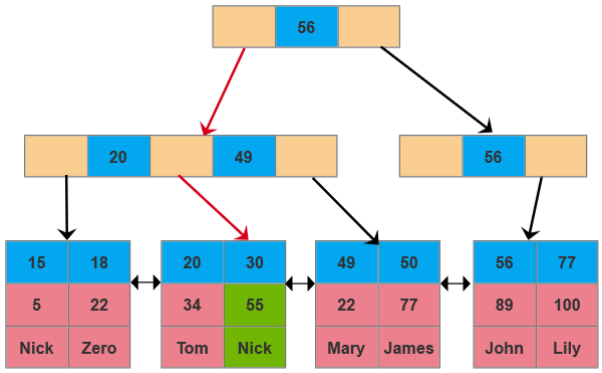
```
select * from t_user_innodb where id=30;
```

1. 先在主键树中从根节点开始检索，将根节点加载到内存，比较30<56，走左路。（1次磁盘IO）
2. 将左子树节点加载到内存中，比较20<30<49，向下检索。（1次磁盘IO）

3. 检索到叶节点，将节点加载到内存中遍历，比较20<30，30=30。查找到值等于30的索引项，直接可以获取整行数据。将改记录返回给客户端。（1次磁盘IO）

磁盘IO次数：3次。

流程分析：



2）范围查询

```
select * from t_user_innodb where id between 30 and 49;
```

1. 先在主键树中从根节点开始检索，将根节点加载到内存，比较30<56，走左路。（1次磁盘IO）
2. 将左子树节点加载到内存中，比较20<30<49，向下检索。（1次磁盘IO）
3. 检索到叶节点，将节点加载到内存中遍历比较20<30，30<=30<49。查找到值等于30的索引项。获取行数据缓存到结果集中。（1次磁盘IO）
4. 向后遍历底层叶子链表，将下一个节点加载到内存中，遍历比较，30<49<=49，获取行数据缓存到结果集中。（1次磁盘IO）
5. 最后得到2条符合筛选条件，将查询结果集返给客户端。

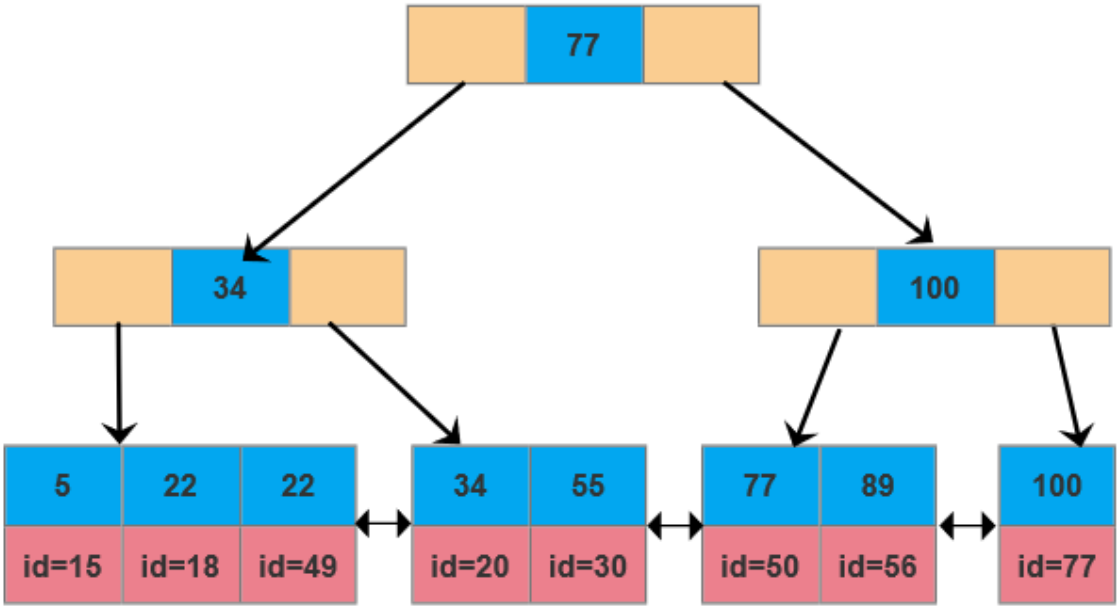
可以看到，因为在主键索引中直接存储了行数据，所以InnoDB在使用主键查询时可以快速获取行数据。当表很大时，与在索引树中存储磁盘地址的方式相比，因为不用再去磁盘中获取数据，所以聚簇索引通常可以节省磁盘IO操作。

磁盘IO次数：2次+检索叶子节点数量。

5.2.3 辅助索引

- 除聚簇索引之外的所有索引都称为辅助索引，InnoDB的辅助索引只会存储主键值而非磁盘地址。
- 使用辅助索引需要检索两遍索引：
  - 首先检索辅助索引获得主键
  - 然后使用主键到主索引中检索获得记录。

以表t_user_innodb的age列为例，age索引的索引结果如下图。底层叶子节点的按照（age，id）的顺序排序，先按照age列从小到大排序，age列相同时按照id列从小到大排序。



1）等值查询

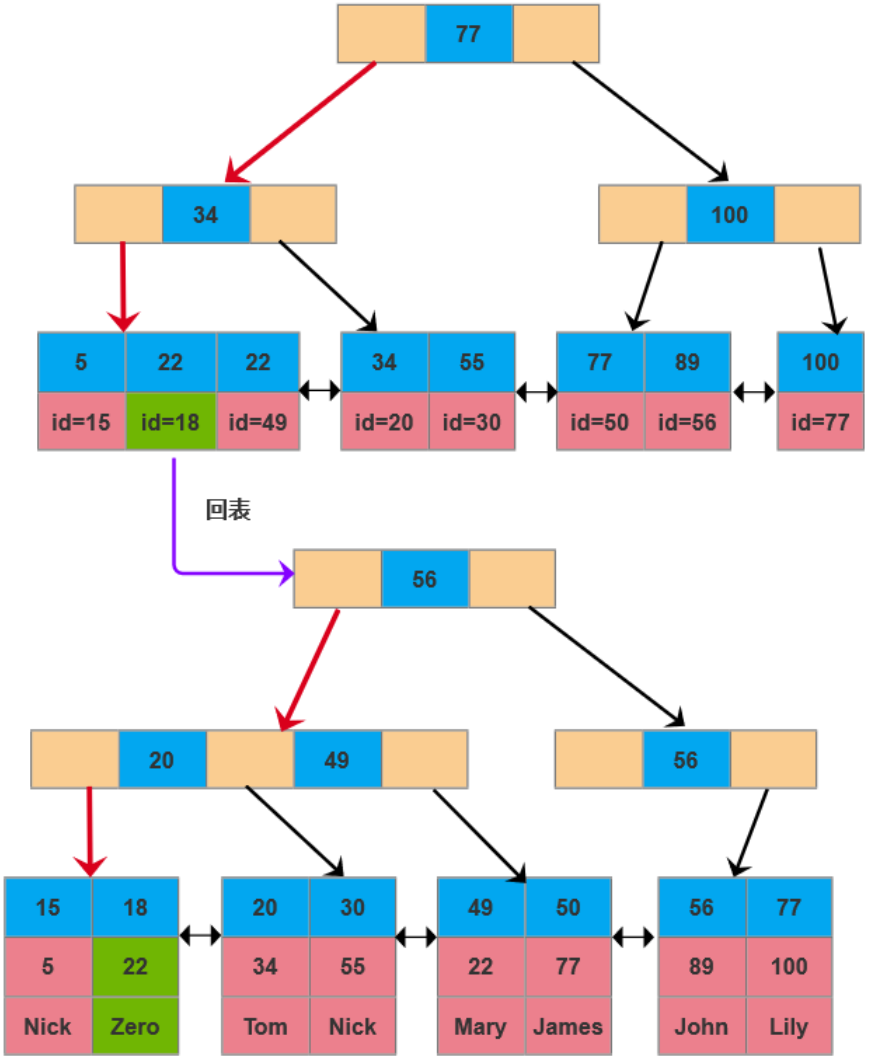
```
select * from t_user_innodb where age=22;
```

1. 先在索引树中从根节点开始检索，将根节点加载到内存，比较22<77，走左路。（1次磁盘IO）
2. 将左子树节点加载到内存中，比较22<34，向下检索。（1次磁盘IO）
3. 检索到叶节点，将节点加载到内存中从前往后遍历比较。（1次磁盘IO）
    - 第一项5：5<22不符合要求，丢弃。
    - 第二项22：等于22，符合要求，获取主键id=18，去主键索引树中检索id=18的数据放入结果集中。（回表查：3次磁盘IO）。
    - 第三项22：等于22，符合要求，获取主键id=49，去主键索引树中检索id=49的数据放入结果集中。（回表查：3次磁盘IO）
4. 向后遍历底层叶子链表，将下一个节点加载到内存中，遍历比较。（1次磁盘IO）
    - 第一项34：34>22不符合要求，丢弃。查询结束。
5. 最后得到2条符合筛选条件，将查询结果集返给客户端。

磁盘IO次数：2次+检索叶子节点数量+记录数*3。

2）什么是回表查询?

根据在辅助索引树中获取的主键id，到主键索引树检索数据的过程称为**回表**查询。



4）范围查询

```
select * from t_user_innodb where age between 30 and 49;
```

- 辅助索引的范围查询流程和等值查询基本一致，先使用辅助索引到叶子节点检索到第一个符合条件的索引项，然后向后遍历，直到遇到第一个不符合条件的索引项，终止。
- 检索过程中需要将符合筛选条件的id值，依次到主键索引检索将检索的数据放入结果集中。
- 最后将查询结果返回客户端。

5.2.4 组合索引

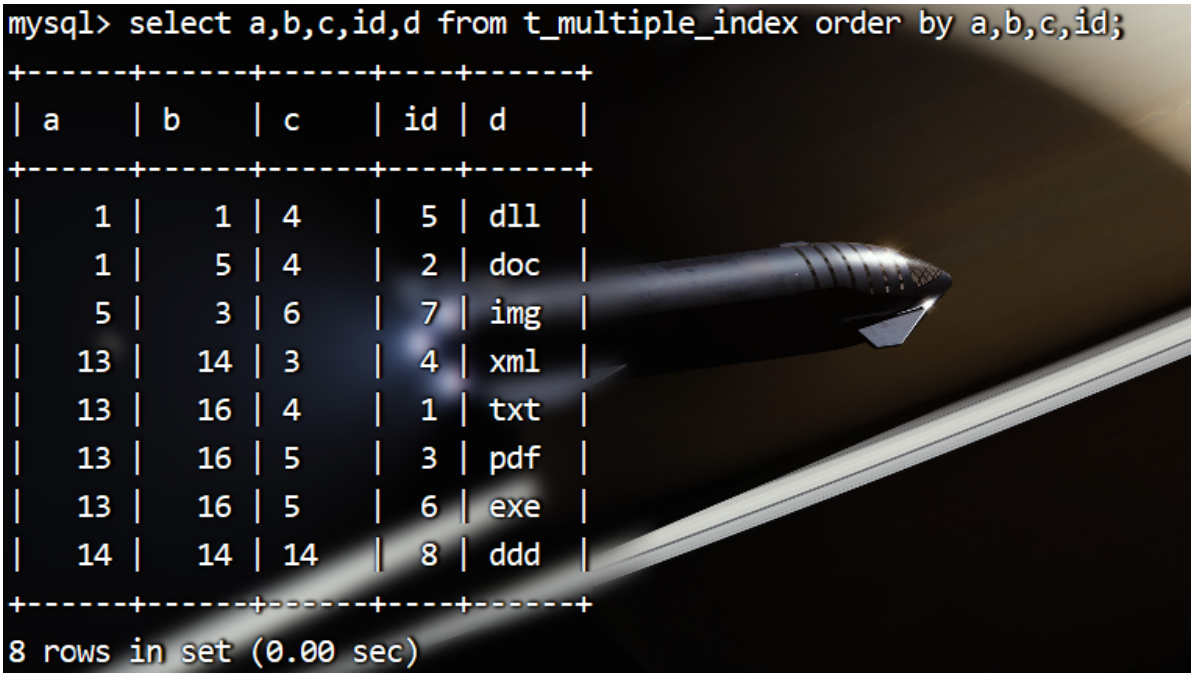
1）组合索引存储结构

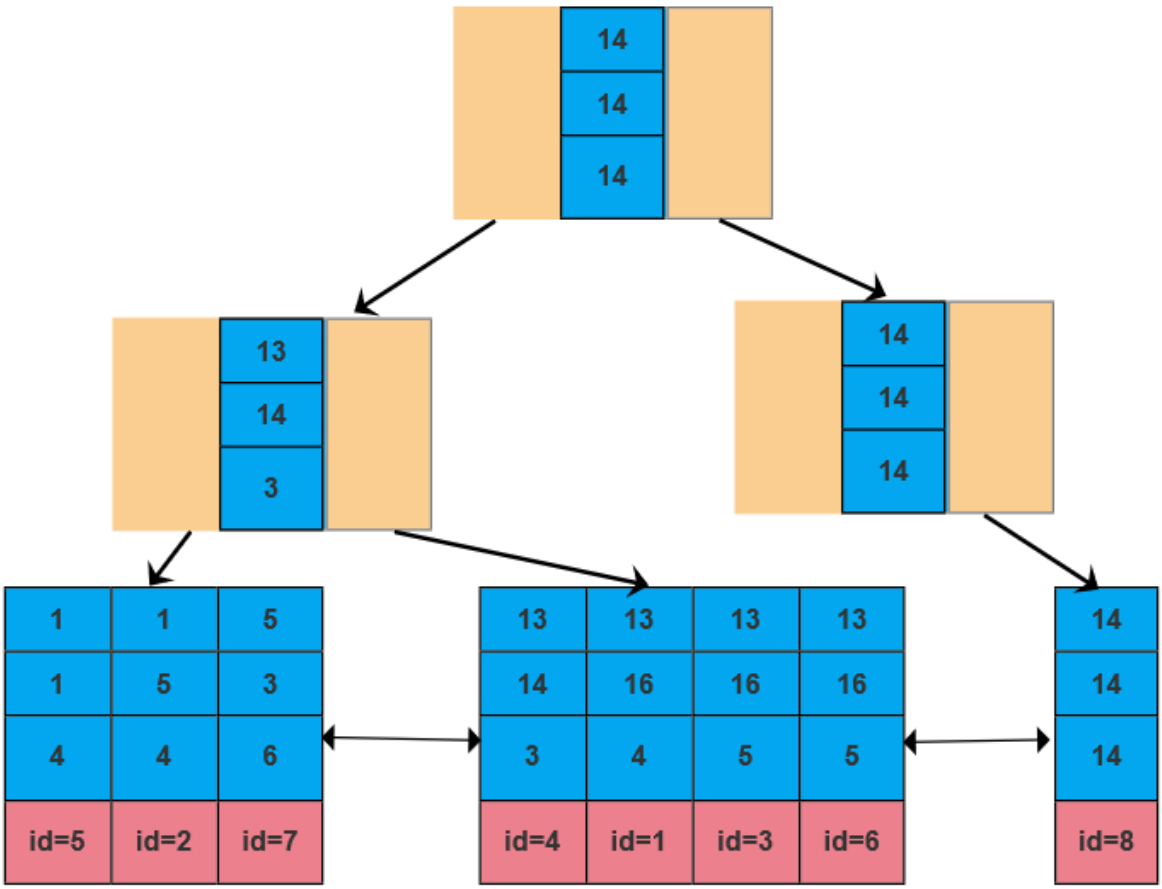
我们在使用索引时，组合索引是我们常用的索引类型。那组合索引是如何构建的，查找的时候又是如何进行查找的呢？

表t_multiple_index，id为主键列，创建了一个联合索引idx_abc(a,b,c)，构建的B+树索引结构如图所示。索引树中节点中的索引项按照（a，b，c）的顺序从大到小排列，先按照a列排序，a列相同时按照b列排序，b列相同按照c列排序。在最底层的叶子节点中，如果两个索引项的a，b，c三列都相同，索引项按照主键id排序。

所以组合索引的最底层叶子节点中不存在完全相同的索引项。

```
CREATE TABLE `t_multiple_index` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  `c` varchar(10) DEFAULT NULL,
  `d` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE,
  KEY `idx_abc` (`a`,`b`,`c`)
) ENGINE=InnoDB;
insert into t_multiple_index (a,b,c,id,d) values(1 ,1 ,4,5,'dll');
insert into t_multiple_index (a,b,c,id,d) values(1 ,5 ,4,2,'doc');
insert into t_multiple_index (a,b,c,id,d) values(5 ,3 ,6,7,'img');
insert into t_multiple_index (a,b,c,id,d) values(13,14,3,4,'xml');
insert into t_multiple_index (a,b,c,id,d) values(13,16,4,1,'txt');
insert into t_multiple_index (a,b,c,id,d) values(13,16,5,3,'pdf');
insert into t_multiple_index (a,b,c,id,d) values(13,16,5,6,'exe');
insert into t_multiple_index (a,b,c,id,d) values(14,14,14,8,'ddd');
```

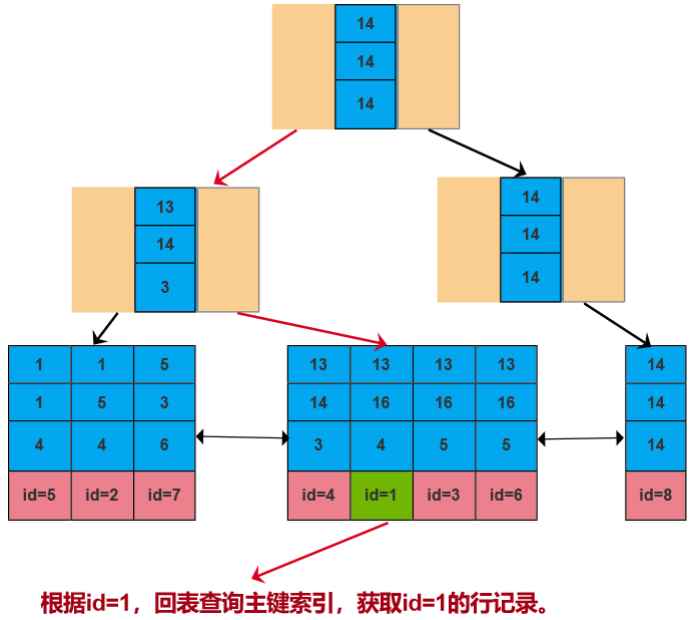




2）组合索引的查找方式

```
select * from  t_multiple_index where a=13 and b=16 and c=4;
```

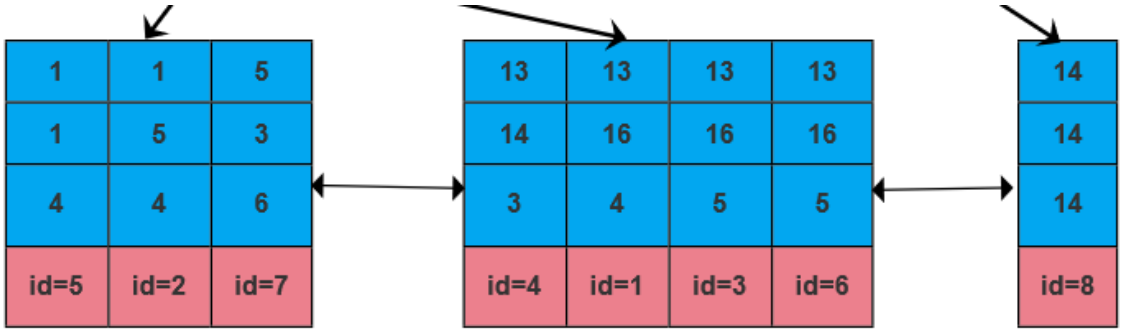
1. 先在索引树中从根节点开始检索，将根节点加载到内存，先比较a列，a=14，14>13，走左路。（1次磁盘IO）
2. 将左子树节点加载到内存中，先比较a列，a=13，比较b列b=16，14<16，走右路，向下检索。（1次磁盘IO）
3. 达到叶节点，将节点加载到内存中从前往后遍历比较。（1次磁盘IO）
    - 第一项（13,14,3,id=4）：先比较a列，a=13，比较b列b=14，b!=16不符合要求，丢弃。
    - 第二项（13,14,4,id=1）：一样的比较方式，a=13，b=16，c=4 满足筛选条件。取出索引data值即主键id=1，再去主键索引树中检索id=1的数据放入结果集中。（回表：3次磁盘IO）
    - 第三项（13,14,5,id=3）：a=13，b=16，c!=4 不符合要求，丢弃。查询结束。
4. 最后得到1条符合筛选条件，将查询结果集返给客户端。



3）最左前缀匹配原则

组合索引的最左前缀匹配原则：使用组合索引查询时，mysql会一直向右匹配直至遇到范围查询(>、<、between、like)就停止匹配。

- 最左前缀匹配原则和联合索引的**索引存储结构和检索方式**是有关系的。
- 在组合索引树中，最底层的叶子节点按照第一列a列从左到右递增排列，但是b列和c列是无序的，b列只有在a列值相等的情况下小范围内递增有序，而c列只能在a，b两列相等的情况下小范围内递增有序。
- 所以当我们使用 where a=13 and b=16 and c=4去查询数据的时候，B+树会先比较a列来确定下一步应该搜索的方向，往左还是往右。如果a列相同再比较b列。但是如果查询条件没有a列，B+树就不知道第一步应该从哪个节点查起。！



所以联合索引只能从第一列开始查找，比如以下三个查询都可以使用idx_abc索引树，检索数据。

```
select * from  t_multiple_index where a=13;
select * from  t_multiple_index where a=13 and b=16;
select * from  t_multiple_index where a=13 and b=16 and c=4;
select * from  t_multiple_index where a=13 and b>13;
select * from  t_multiple_index where a>11 and b=14;
select * from  t_multiple_index where a=16 and c=4;
```

而如果查询条件不包含a列，比如筛选条件只有(b，c)或者c列是无法使用组合索引的。下面的查询没有用到索引。

```sql
select * from  t_multiple_index where b=16 and c=4;
select * from  t_multiple_index where c=4;
```

所以创建的idx_abc(a,b,c)索引，相当于创建了(a)、（a,b）（a,b,c）三个索引。

另外，我们还需要注意的是，书写SQL条件的顺序，不一定是执行时候的where条件顺序。优化器会帮助我们优化成索引可以识别的形式。比如:

```sql
select * from  t_multiple_index where b=16 and c=4 and a=13;
#等价于下面的sql，优化器会按照索引的顺序优化
select * from  t_multiple_index where a=13 and b=16 and c=4;


explain select a,b from  t_multiple_index where  b=16;
explain select b from  t_multiple_index where b=16 and c=4;
explain select b,c from  t_multiple_index where c=4;
```

一颗索引树等价与三颗索引树，从另一方面来说，组合索引也为我们节省了磁盘空间。所以在业务中尽量选用组合索引，能使用组合索引就不要使用单列索引。

> **索引使用口诀**
>
> 全值匹配我最爱，最左前缀要遵守。
> 带头大哥不能死，中间兄弟不能断。
> 索引列上不计算，范围之后全失效。
> Like百分写最右，覆盖索引不写星。
> 不等空值还有OR，索引失效要少用。

4）组合索引创建原则

1. 频繁出现在where条件中的列，建议创建组合索引。
2. 频繁出现在order by和group by语句中的列，建议按照**顺序**去创建组合索引。
    - order by a,b 需要组合索引列顺序（a,b）。如果索引的顺序是（b,a），是用不到索引的。
3. 常出现在select语句中的列，也建议创建组合索引。

> 大家思考个问题，这个是咱们同学遇到的一个面试题。下面的SQL语句除了建a，b联合索引，还有更好的方案吗?
>
> ```sql
> select * from t where a=1 and b>2 order by c
> ```
>
> 可以考虑建立（a , c）联合索引：select * from xxx where a=1 and b>2 order by c 这样 a等值查询 c就是已经排好序的了。这种情况实际上比较的是b的区分度和c的区分度，如果b的区分度比较差，建议使用a，c。如果c的区分度比较差，建议使用a , b。

5.2.5 覆盖索引

前面我们提到，根据在辅助索引树查询数据时，首先通过辅助索引找到主键值，然后需要再根据主键值到主键索引中找到主键对应的数据。这个过程称为**回表**。

使用辅助索引查询比基于主键索引的查询多检索了一棵索引树。**那是不是所有使用辅助索引的查询都需要回表查询呢?**

表t_multiple_index，组合索引idx_abc(a,b,c)的叶子节点中包含(a,b,c,id)四列的值，对于以下查询语句

```
select a from  t_multiple_index where a=13 and b=16;
select a,b from  t_multiple_index where a=13 and b=16;
select a,b,c from  t_multiple_index where a=13 and b=16;
select a,b,c,id from  t_multiple_index where a=13 and b=16;
```

什么是覆盖索引?

select中列数据如果可以直接在辅助索引树上全部获取，也就是说索引树已经“覆盖”了我们的查询需求，这时MySQL就不会白费力气的回表查询，这中现象就是**覆盖索引**。

使用explain工具查看执行计划，可以看到extra中“Using index”，代表使用了覆盖索引。

```
explain select a,b from  t_multiple_index where a=13 and b=16;
```

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	t_multiple_index	(Null)	ref	idx_abc	idx_abc	10	const,const	3	100	Using index

大家试试将上面的语句，改为如下语句。大家猜猜这时会不会用到组合索引?

```
select a,b from  t_multiple_index where  b=16;
```

```
explain select a,b from  t_multiple_index where  b=16;
```

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	t_multiple_index	(Null)	index	(Null)	idx_abc	53	(Null)	8	12.5	Using where; Using index

上面的查询语句用到了覆盖索引进行**索引扫描**。MySQL基于成本考虑，会使用了覆盖索引进行全表扫描，使用覆盖索引可以减少了磁盘IO次数，显著提升查询性能。

覆盖索引相比与主键索引一个索引项占用的空间少，覆盖索引一个叶子节点中的就可以比主键索引存放更多的数据量，相应的存放数据用到的总叶子树很少一些。

覆盖索引是一种很常用的优化手段。

5.2.6 索引条件下推ICP

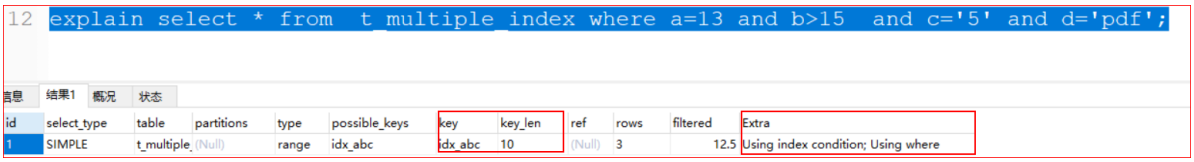
官方索引条件下推： Index Condition Pushdown，简称ICP。是MySQL5.6对使用索引从表中检索行的一种优化。ICP可以减少存储引擎必须访问基表的次数以及MySQL服务器必须访问存储引擎的次数。可用于 InnoDB 和 MyISAM 表，对于InnoDB表ICP仅用于辅助索引。

可以通过参数optimizer_switch控制ICP的开始和关闭。

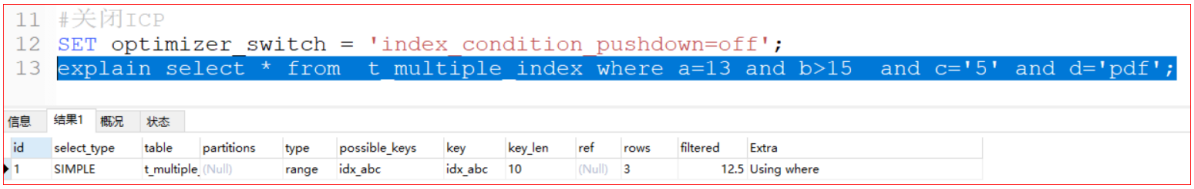
```
#optimizer_switch优化相关参数开关
mysql> show VARIABLES like 'optimizer_switch';
#关闭ICP
SET optimizer_switch = 'index_condition_pushdown=off';
#开启ICP
SET optimizer_switch = 'index_condition_pushdown=on';
```

以InnoDB的辅助索引为例，来讲解ICP的作用：MySQl在使用组合索引在检索数据时是使用最左前缀原则来定位记录，左侧前缀之后不匹配的后缀，MySQL会怎么处理?

```
select * from  t_multiple_index where a=13 and b>15  and c='5' and d='pdf';
```

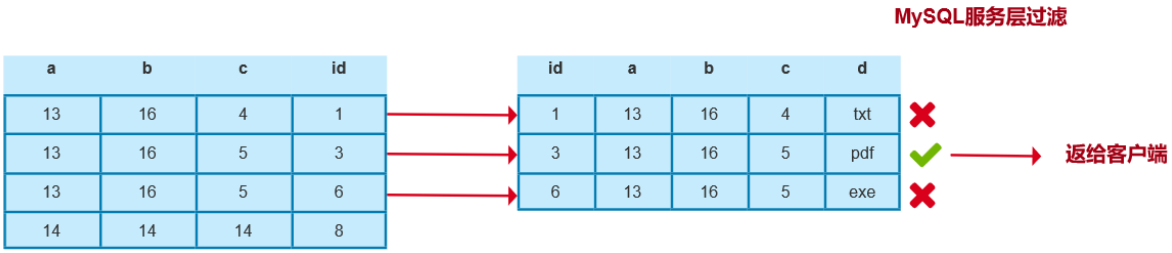


- 根据最左前缀匹配原则，这个SQL语句会使用组合索引idx_abc(a,b,c)的（a,b）两列来检索记录。
- MySQL首先会在组合索引中定位到第一个满足a=13 and b>=15的索引项，**MySQL之后会怎么处理呢?**
  - 使用explain工具，查看执行计划，extra列中的“Using index condition”执行器表示使用了索引条件下推ICP。
  - **在MySQL 5.6之前**：不使用ICP时，MySQL只能从索引项（13,16,4,1）开始，一个个回表查询找到行数据，然后再在服务层过滤后，返回给客户端。
  - **在MySQL 5.6之后**：在使用ICP和不使用ICP时MySQL的执行情况会有所不同。
- 关闭ICP，使用explain工具，查看执行计划，extra列中的“Using where”执行器表示没有使用了索引条件下推ICP。



举个栗子：

1）不使用索引ICP



具体步骤如下：

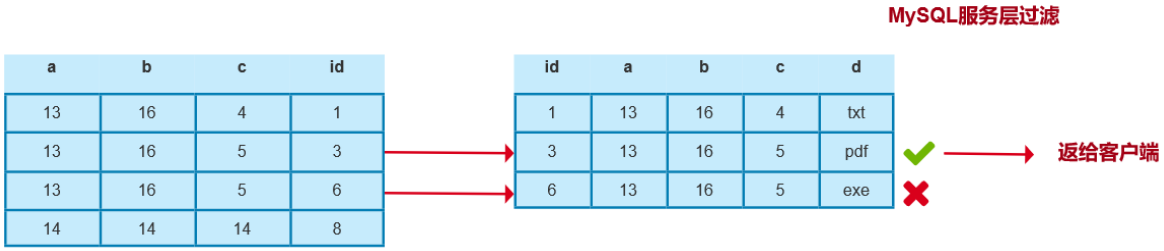
1. 执行器使用索引(a,b,c)，筛选条件a=13 and b>=15，调用存储引擎"下一行"接口。根据最左前缀原则联合索引检索定位到索引项（13,16,4,id=1），然后使用id=1回表查询，获得id=1的行记录。返回给MySQL服务层，MySQL服务层使用剩余条件c=5 and d='pdf'过滤，不符合要求，直接丢弃。
2. 执行器调用"下一行"接口，存储引擎遍历向后找到索引项（13,16,5,id=3）,使用id=3回表获得id=3的行记录。返回给MySQL服务层，MySQL服务层使用剩余条件c=5 and d='pdf'过滤，符合要求，缓存到结果集。

3. 执行器调用"下一行"接口，存储引擎遍历向后找到索引项（13,16,5,id=6）,使用id=6回表获得id=6的行记录。返回给MySQL服务层，MySQL服务层使用剩余条件c=5 and d='pdf'过滤，不符合要求，直接丢弃。
4. 执行器调用"下一行"接口，存储引擎遍历向后找到索引项（14,14,14,id=8）不满足筛选条件，执行器终止查询。
5. 最终获取一条记录，返回给客户端。

可以看到，**在不使用ICP时，回表查询了3次，然后在服务层筛选后（筛选3次）**，最后返回客户端。

在MySQL 5.6 引入了ICP，可以在索引遍历过程中，对where中包含的索引条件先做判断，只有满足条件的才会回表查询读取行数据。这么做可以减少回表查询，从而减少磁盘IO次数。

2）使用索引ICP



使用ICP时，具体步骤如下：

1. 执行器使用索引(a,b,c)，筛选条件a=13 and b>=15 and c=5，调用存储引擎"下一行"接口。根据最左前缀原则联合索引检索定位到索引项（13,16,4,id=1），然后使用ICP下推条件c=5判断，不满足条件，直接丢弃。
2. 向后遍历判断索引项（13,16,5,id=3），满足筛选条件a=13 and b>=15 and c=5，使用id=3回表获得id=3的行记录。返回给MySQL服务层，MySQL服务层使用剩余条件d='pdf'过滤，符合要求，缓存到结果集。
3. 执行器调用"下一行"接口，存储引擎遍历向后找到索引项（13,16,5,id=6）,满足筛选条件a=13 and b>=15 and c=5，使用id=6回表获得id=6的行记录。返回给MySQL服务层，MySQL服务层使用剩余条件d='pdf'过滤，不符合要求，直接丢弃。
4. 执行器调用"下一行"接口，存储引擎遍历向后找到索引项（14,14,14,id=8）不满足筛选条件，执行器终止查询。
5. 最终获取一条记录，返回给客户端。

可以看到，在使用ICP时，回表查询了2次，然后在服务层筛选后（筛选2次），最后返回客户端。

3）小结

不使用ICP，不满足最左前缀的索引条件的比较是在Server层进行的，非索引条件的比较是在Server层进行的。

使用ICP，所有的索引条件的比较是在存储引擎层进行的，非索引条件的比较是在Server层进行的。

对比使用ICP和不使用ICP，可以看到使用ICP可以有效减少回表查询次数和返回给服务层的记录数，从而减少了磁盘IO次数和服务层与存储引擎的交互次数。

6. 索引创建原则

6.1 哪些情况需要创建索引

1. 频繁出现在where 条件字段，order排序，group by分组字段
2. select 频繁查询的列，考虑是否需要创建联合索引（覆盖索引，不回表）
3. 多表join关联查询，on字段两边的字段都要创建索引

6.2 索引优化建议

1. **表记录很少不需创建索引：**索引是要有存储的开销
2. **一个表的索引个数不能过多：**

   （1）空间：浪费空间。每个索引都是一个索引树，占据大量的磁盘空间。

   （2）时间：更新（插入/Delete/Update）变慢。需要更新所有的索引树。太多的索引也会增加优化器的选择时间。

   所以索引虽然能够提高查询效率，索引并不是越多越好，应该只为需要的列创建索引。
3. **频繁更新的字段不建议作为索引：**频繁更新的字段引发频繁的页分裂和页合并，性能消耗比较高。
4. **区分度低的字段，不建议建索引：**

   比如性别，男，女；比如状态。区分度太低时，会导致扫描行数过多，再加上回表查询的消耗。如果使用索引，比全表扫描的性能还要差。这些字段一般会用在组合索引中。

   姓名，手机号就非常适合建索引。
5. **在InnoDB存储引擎中，主键索引建议使用自增的长整型，避免使用很长的字段：**

   主键索引树一个页节点是16K，主键字段越长，一个页可存储的数据量就会越少，比较臃肿，查询时尤其是区间查询时磁盘IO次数会增多。辅助索引树上叶子节点存储的数据是主键值，主键值越长，一个页可存储的数据量就会越少，查询时磁盘IO次数会增多，查询效率会降低。
6. **不建议用无序的值作为索引：**例如身份证、UUID。更新数据时会发生频繁的页分裂，页内数据不紧凑，浪费磁盘空间。
7. **尽量创建组合索引，而不是单列索引：**

   优点：

   （1）1个组合索引等同于多个索引效果，节省空间。

   （2）可以使用覆盖索引

   创建原则：组合索引应该把频繁用到的列、区分度高的值放在前面。频繁使用代表索引的利用率高，区分度高代表筛选粒度大，这样做可最大限度利用索引价值，缩小筛选范围

案例：索引失效分析

组合索引心法口诀：

- 全值匹配我最爱，最左前缀要遵守；
- 带头大哥不能死，中间兄弟不能断；
- 索引列上不计算，范围之后全失效；
- Like百分写最右，覆盖索引不写星；
- 不等空值还有OR，索引失效要少用。

1、案例环境

```
CREATE TABLE `t_user_index_analyse` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `pos` varchar(10) DEFAULT NULL,
  `pay_time` datetime DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE,
  KEY `idx_user_nameAgePos` (`name`,`age`,`pos`)
) ENGINE=InnoDB;
insert into t_user_index_analyse (id,name,age,pos,pay_time) values(1 ,'z3'
,22,'manager','2022-03-13 09:31:34');
insert into t_user_index_analyse (id,name,age,pos,pay_time) values(2 ,'July'
,23,'dev','2022-03-13 09:31:34');
insert into t_user_index_analyse (id,name,age,pos,pay_time) values(3 ,'2000'
,23,'dev','2022-03-13 09:31:34');
```

```
show  index from t_user_index_analyse;
```

```
28 show index from t_user_index_analyse;
```

信息 结果1 概况 状态

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
t_user_index_analyse	0	PRIMARY	1	id	A	3	(Null)	(Null)		BTREE		
t_user_index_analyse	1	idx_user_nameAgePos	1	name	A	3	(Null)	(Null)	YES	BTREE		
t_user_index_analyse	1	idx_user_nameAgePos	2	age	A	3	(Null)	(Null)	YES	BTREE		
t_user_index_analyse	1	idx_user_nameAgePos	3	pos	A	3	(Null)	(Null)	YES	BTREE		

2、案例演示

1. 全值匹配我最爱
2. 最左前缀匹配原则
3. 不在索引列上做任何操作【计算、函数、类型转换】，会导致索引失效，转而使用全表扫描
4. 存储引擎不能使用索引中范围条件右边的列
5. 尽量使用覆盖索引【只访问索引的查询，索引列和查询列一致】，减少使用select *
6. 不等于【!= 或 <>】，索引会失效
7. is null，is not null，索引会失效
8. like以通配符开头，索引会失效
9. 字符串不加单引号，索引会失效
10. 少用or，用它来连接时，索引会失效

1）全值匹配我最爱

```
explain select * from  t_user_index_analyse where name='July';
explain select * from  t_user_index_analyse where name='July' and age=25;
explain select * from  t_user_index_analyse where name='July' and age=25 and
pos='dev';
```

```
33 explain select * from  t_user_index_analyse where name='July';
34
```

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	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1023	const	1	100	(Null)

```
34 explain select * from  t_user_index_analyse where name='July' and age=25;
35
```

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	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1028	const,const	1	100	(Null)

```
35 explain select * from  t_user_index_analyse where name='July' and age=25 and pos='dev';
```

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	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1071	const,const,const	1	100	(Null)

2）最左前缀法则

```
1  带头索引不能死，中间索引不能断
```

如果索引了多个列，要遵守最佳左前缀法则。指的是查询从索引的最左前列开始 并且不跳过索引中的列。 正确的示例参考上图。

错误的示例：

带头索引死：

```
1  explain select * from  t_user_index_analyse where age=23 and pos='dev';
2  explain select * from  t_user_index_analyse where pos='dev';
```

```
36 explain select * from  t_user_index_analyse where age=23 and pos='dev';
37
```

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	t_user_index_analyse	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

```
37 explain select * from  t_user_index_analyse where pos='dev';
```

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	t_user_index_analyse	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

中间索引断（带头索引生效，其他索引失效）：

```
1  explain select * from  t_user_index_analyse where name='July' and pos='dev';
```

```
38 explain select * from  t_user_index_analyse where name='July' and pos='dev';
```

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	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1023	const	1	33.33	Using where

3）不要在索引上做计算

不要进行这些操作：计算、函数、自动/手动类型转换，不然会导致索引失效而转向全表扫描

```
1  explain select * from  t_user_index_analyse where name='July';
2  explain select * from  t_user_index_analyse where left(name,4)='July';
```

```
40 explain select * from  t_user_index_analyse where name='July';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1023	const	1	100	(Null)

```
43 explain select * from  t_user_index_analyse where left(name,4)='July';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_ind	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100	Using where

4）范围条件右边的列失效

```
# 不能继续使用索引中范围条件（bettween、<、>、in等）右边的列
explain select * from  t_user_index_analyse where name='July' and age=25 and pos='manager';
explain select * from  t_user_index_analyse where name='July' and age>25 and pos='manager';
```

```
43 explain select * from  t_user_index_analyse where name='July' and age=25 and pos='manager';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1071	const,const,const	1	100	(Null)

```
46 explain select * from  t_user_index_analyse where name='July' and age>25 and pos='manager';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	range	idx_user_nameAgePos	idx_user_nameAgePos	1028	(Null)	1	33.33	Using where

5）尽量使用覆盖索引

```
# 尽量使用覆盖索引（只查询索引的列），也就是索引列和查询列一致，减少select *
explain select * from  t_user_index_analyse where name='July' and age=25 and pos='manager';
explain select name,age,pos from  t_user_index_analyse where name='July' and age=25 and pos='manager';
```

```
48 explain select * from  t_user_index_analyse where name='July' and age=25 and pos='manager';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1071	const,const,const	1	100	(Null)

```
53 explain select name,age,pos from  t_user_index_analyse where name='July' and age=25 and pos='manager';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1071	const,const,const	1	100	Using index

6）索引字段上不要使用不等

```
# 索引字段上使用（！= 或者 < >）判断时，会导致索引失效而转向全表扫描
explain select * from  t_user_index_analyse where name = 'July';
explain select * from  t_user_index_analyse where name != 'July';
explain select * from  t_user_index_analyse where name <> 'July';
```

```
56 explain select * from  t_user_index_analyse where name = 'July';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1023	const	1	100	(Null)

```
60 explain select * from  t_user_index_analyse where name != 'July';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user_index_analyse	(Null)	ALL	idx_user_nameAgePos	(Null)	(Null)	(Null)	3	66.67	Using where

```
63 explain select * from  t_user_index_analyse where name <> 'July';
```

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	t_user_index_analyse	(Null)	ALL	idx_user_nameAgePos	(Null)	(Null)	(Null)	3	66.67	Using where

7）索引字段上不要判断null

```
# 索引字段上使用 is not null 判断时，会导致索引失效而转向全表扫描
explain select * from  t_user_index_analyse where name is not null;
```

```
68 explain select * from  t_user_index_analyse where name is not 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	t_user_index_analyse	(Null)	ALL	idx_user_nameAgePos	(Null)	(Null)	(Null)	3	100	Using where

8）索引字段使用like不以通配符开头

```
# 索引字段使用like以通配符开头（‘%字符串’）时，会导致索引失效而转向全表扫描
explain select * from  t_user_index_analyse where name like '%July%';
explain select * from  t_user_index_analyse where name like '%July';
explain select * from  t_user_index_analyse where name like 'July%';
```

```
71 explain select * from  t_user_index_analyse where name like '%July%';
```

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	t_user_index_analyse	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

```
75 explain select * from  t_user_index_analyse where name like '%July';
```

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	t_user_index_analyse	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

```
79 explain select * from  t_user_index_analyse where name like 'July%';
```

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	t_user_index_analyse	(Null)	range	idx_user_nameAgePos	idx_user_nameAgePos	1023	(Null)	1	100	Using where

由结果可知，like以通配符结束相当于范围查找，索引不会失效。与范围条件（bettween、<、>、in等）不同的是**不会导致右边的索引失效。**

9）索引字段字符串要加单引号

```
# 索引字段是字符串，但查询时不加单引号，会导致索引失效而转向全表扫描
explain select * from  t_user_index_analyse where name = '2000';
explain select * from  t_user_index_analyse where name = 2000;
```

```
50 explain select * from  t_user_index_analyse where name = '2000';
51
```

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	t_user_index_analyse	(Null)	ref	idx_user_nameAgePos	idx_user_nameAgePos	1023	const	1	100	(Null)

```
53 explain select * from  t_user_index_analyse where name = 2000;
```

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	t_user_index_analyse	(Null)	ALL	idx_user_nameAgePos	(Null)	(Null)	(Null)	3	33.33	Using whe

10）索引字段不要使用or

```
# 索引字段使用 or 时，会导致索引失效而转向全表扫描
explain select * from  t_user_index_analyse where name = 'July' or name='z3';
```

```
56 explain select * from  t_user_index_analyse where name = 'July' or name='z3';
```

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	t_user_index_analyse	(Null)	ALL	idx_user_nameAgePos	(Null)	(Null)	(Null)	3	66.67	Using whe