前言

说到索引，很多人都知道“索引是一个排序的列表，在这个列表中存储着索引的值和包含这个值的数据所在行的物理地址，在数据十分庞大的时候，索引可

以大大加快查询的速度，这是因为使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。”

但是索引是怎么实现的呢？因为索引并不是关系模型的组成部分，因此不同的DBMS有不同的实现，我们针对MySQL数据库的实现进行说明。

一、MySQL中索引的语法

创建索引

在创建表的时候添加索引

```
CREATE TABLE mytable(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL,
    INDEX [indexName] (username(length))
);
```

在创建表以后添加索引

```
ALTER TABLE my_table ADD [UNIQUE] INDEX index_name(column_name);
```

或者

```
CREATE INDEX index_name ON my_table(column_name);
```

注意：

1、索引需要占用磁盘空间，因此在创建索引时要考虑到磁盘空间是否足够

2、创建索引时需要对表加锁，因此实际操作中需要在业务空闲期间进行

> 原文链接：https://blog.csdn.net/tongdanping/article/details/79878302

删除索引

```
DROP INDEX my_index ON tablename;
```

或者

```
ALTER TABLE table_name DROP INDEX index_name;
```

查看表中的索引

```
SHOW INDEX FROM tablename
```

查看查询语句使用索引的情况
//explain 加查询语句
explain SELECT * FROM table_name WHERE column_1='123';
————————————————

> 原文链接：https://blog.csdn.net/tongdanping/article/details/79878302

二、索引的优缺点

优势：可以快速检索，减少I/O次数，加快检索速度；根据索引分组和排序，可以加快分组和排序；
劣势：索引本身也是表，因此会占用存储空间，一般来说，索引表占用的空间的数据表的1.5倍；

索引表的维护和创建需要时间成本，这个成本随着数据量增大而增大；

构建索引会降低数据表的修改操作（删除，添加，修改）的效率，因为在修改数据表的同时还需要修改索引表；

> 原文链接：https://blog.csdn.net/tongdanping/article/details/79878302

三、索引的分类

常见的索引类型有：**主键索引、唯一索引、普通索引、全文索引、组合索引**

1、**主键索引**：即主索引，根据主键pk_clolum（length）建立索引，不允许重复，不允许空值；
ALTER TABLE 'table_name' ADD PRIMARY KEY pk_index('col');

2、**唯一索引**：用来建立索引的列的值必须是唯一的，允许空值
ALTER TABLE 'table_name' ADD UNIQUE index_name('col');

3、**普通索引**：用表中的普通列构建的索引，没有任何限制
ALTER TABLE 'table_name' ADD INDEX index_name('col');

4、**全文索引**：用大文本对象的列构建的索引（下一部分会讲解）
ALTER TABLE 'table_name' ADD FULLTEXT INDEX ft_index('col');

5、组合索引：用多个列组合构建的索引，这多个列中的值不允许有空值
ALTER TABLE 'table_name' ADD INDEX index_name('col1','col2','col3');
*遵循“最左前缀”原则，把最常用作为检索或排序的列放在最左，依次递减，组合索引相当于建立了col1,col1col2,col1col2col3三个索引，而col2或者col3是不能使用索引的。
*在使用组合索引的时候可能因为列名长度过长而导致索引的key太大，导致效率降低，在允许的情况下，可以只取col1和col2的前几个字符作为索引
ALTER TABLE 'table_name' ADD INDEX index_name(col1(4),col2（3));
表示使用col1的前4个字符和col2的前3个字符作为索引

6、前缀索引：当索引是很长的字符序列时，这个索引将会很占内存，而且会很慢，这时候就会用到前缀索引了。所谓的前缀索引就是去索引的前面几个字母作为索引，但是要降低索引的重复率，索引我们还必须要判断前缀索引的重复率
alter table test add key(name(4));

> select 1.0*count(distinct name)/count(*) from test这是比较整个name的重复率,选择较好的索引长度
> https://blog.csdn.net/ma2595162349/article/details/79449493

> 原文链接：https://blog.csdn.net/tongdanping/article/details/79878302

四、索引的实现原理

MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，B+Tree索引，哈希索引，全文索引等等，

4.1、哈希索引

只有memory（内存）存储引擎支持哈希索引，哈希索引用索引列的值计算该值的hashCode，然后在hashCode相应的位置存执该值所在行数据的物理位置，因为使用散列算法，因此访问速度非常快，但是一个值只能对应一个hashCode，而且是散列的分布方式，因此哈希索引不支持范围查找和排序的功能。

> 缺点:
> 0、因为Hash索引比较的是经过Hash计算的值，所以只能进行等式比较，不能用于范围查询
> 1、每次都要全表扫描
> 2、由于哈希值是按照顺序排列的，但是哈希值映射的真正数据在哈希表中就不一定按照顺序排列，所以无法利用Hash索引来加速任何排序操作
> 3、不能用部分索引键来搜索，因为组合索引在计算哈希值的时候是一起计算的。

> 4、当哈希值大量重复且数据量非常大时，其检索效率并没有Btree索引高的。

https://www.cnblogs.com/zhidongjian/p/10414129.html

4.2、全文索引

FULLTEXT（全文）索引，仅可用于MyISAM和InnoDB，针对较大的数据，生成全文索引非常的消耗时间和空间。对于文本的大对象，或者较大的CHAR类型的数据，如果使用普通索引，那么匹配文本前几个字符还是可行的，但是想要匹配文本中间的几个单词，那么就要使用LIKE %word%来匹配，这样需要很长的时间来处理，响应时间会大大增加，这种情况，就可使用时FULLTEXT索引了，在生成FULLTEXT索引时，会为文本生成一份单词的清单，在索引时及根据这个单词的清单来索引。FULLTEXT可以在创建表的时候创建，也可以在需要的时候用ALTER或者CREATE INDEX来添加：

全文索引的查询也有自己特殊的语法，而不能使用LIKE %查询字符串%的模糊查询语法

SELECT * FROM table_name MATCH(ft_index) AGAINST('查询字符串');

> 注意:
> *对于较大的数据集，把数据添加到一个没有FULLTEXT索引的表，然后添加FULLTEXT索引的速度比把数据添加到一个已经有FULLTEXT索引的表快。
> *5.6版本前的MySQL自带的全文索引只能用于MyISAM存储引擎，如果是其它数据引擎，那么全文索引不会生效。**5.6版本之后InnoDB存储引擎开始支持全文索引**
> *在MySQL中，全文索引支队英文有用，目前对中文还不支持。**5.7版本之后通过使用ngram插件开始支持中文。**
> *在MySQL中，如果检索的字符串太短则无法检索得到预期的结果，检索的字符串长度至少为4字节，此外，如果检索的字符包括停止词，那么停止词会被忽略。

五、索引数据结构

为什么用B/B+树这种结构来实现索引呢?

答：红黑树等结构也可以用来实现索引，但是文件系统及数据库系统普遍使用B/B+树结构来实现索引。mysql是基于磁盘的数据库，索引是以索引文件的形式存在于磁盘中的，索引的查找过程就会涉及到磁盘IO(为什么涉及到磁盘IO请看文章后面的附加理解部分)消耗，磁盘IO的消耗相比较于内存IO的消耗要高好几个数量级，所以索引的组织结构要设计得在查找关键字时要尽量减少磁盘IO的次数。为什么要使用B/B+树，跟磁盘的存储原理有关。

局部性原理与磁盘预读

为了提升效率，要尽量减少磁盘IO的次数。实际过程中，磁盘并不是每次严格按需读取，而是每次都会预读。磁盘读取完需要的数据后，会按顺序再多读一部分数据到内存中，这样做的理论依据是计算机科学中注明的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用;程序运行期间所需要的数据通常比较集中

（1）由于磁盘顺序读取的效率很高(不需要寻道时间，只需很少的旋转时间)，
因此对于具有局部性的程序来说，预读可以提高I/O效率.预读的长度一般为页(page)的整倍数。
（2）MySQL(默认使用InnoDB引擎),将记录按照页的方式进行管理,每页大小默认为16K(这个值可以修改)。linux 默认页大小为4K。

为什么不用红黑树?
B-Tree可以借助计算机磁盘预读的机制，并使用如下技巧：
每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个结点只需一次I/O。
假设 B-Tree 的高度为 h,B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3，也即索引的B+树层次一般不超过三层，所以查找效率很高）。
而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，**无法利用局部性**，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

> 链接：https://www.jianshu.com/p/0371c9569736
> https://blog.csdn.net/kongmin_123/article/details/82055901

每次插入和删除都会更新树,树需要重新达到平衡(或者说需要保持定义),btree和b+tree的操作有点不同,

> 具体图解:
> https://blog.csdn.net/disiwei1012/article/details/78632859
> https://www.cnblogs.com/nullzx/p/8729425.html

5.1 BTree索引

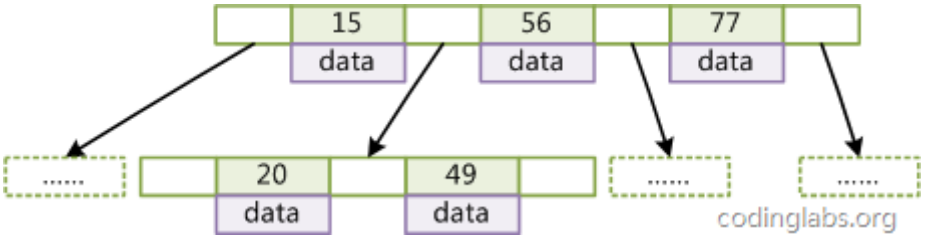
> 也就是B-tree https://ac.nowcoder.com/discuss/299888?type=1&order=0&pos=11&page=1

BTree是平衡多路搜索树，设树的度为2d（d>1），高度为h，那么BTree要满足以下条件：

- 每个叶子结点的高度一样，等于h；
- 每个非叶子结点由n-1个key和n个指针point组成，其中d<=n<=2d,key和point相互间隔，结点两端一定是key；
- 叶子结点指针都为null；
- 非叶子结点的key都是[key,data]二元组，其中key表示作为索引的键，data为键值所在行的数据；

> **多路搜索树是指每个节点的孩子数可以有多个,每个的节点存储的数据也可以有多个**,
> 普通的二叉树只能一个节点数据+两个孩子,
> 一个节点不像图中画的只有几个数值,为了方便理解嘛,可能是成千上万个数值, **每个索引节点一般都是操作系统页的整数倍**,InnoDB默认是16K,到时候查出来的数据,在内存中进行筛选(索引数据都是在磁盘上的)
> **一个千万量级，且存储引擎是MyISAM或者InnoDB的表，其索引树的高度在3~5之间**
> **来自: https://cloud.tencent.com/developer/news/373193**

BTree的结构如下：



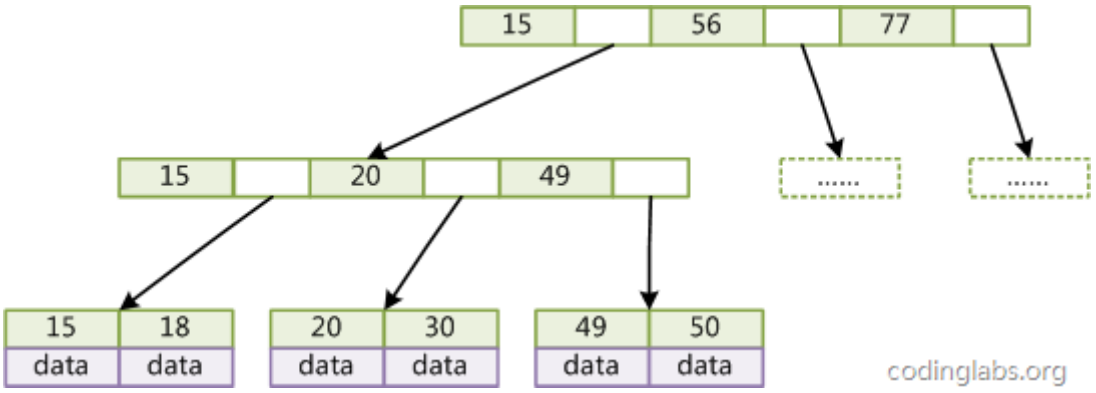
在BTree的机构下，就可以使用二分查找的查找方式，时间复杂度为h*log(n)，一般来说树的高度是很小的，h一般为3左右，因此BTree是一个非常高效的查找结构。

5.2 B+Tree索引

B+Tree是BTree的一个变种，设d为树的度数，h为树的高度，B+Tree和BTree的不同主要在于：

- B+Tree中的非叶子结点不存储数据，只存储键值；
- B+Tree的叶子结点没有指针，所有键值都会出现在叶子结点上，且key存储的键值对应data数据的物理地址；
- B+Tree的每个非叶子节点由n个键值key和n个指针point组成；

B+Tree的结构如下：



> 上面的15等等数据都是虚拟的,真正的数据节点在叶子节点上,上图中一个节点保持了三个数据,这样就确定了范围,整棵树也将变得小了,(也不一定是三个数,比如数据少的时候)

B+Tree对比BTree的优点：

1、磁盘读写代价更低

一般来说B+Tree比BTree更适合实现外存的索引结构，因为存储引擎的设计专家巧妙的利用了外存（磁盘）的存储结构，即磁盘的最小存储单位是扇区（sector），而操作系统的块（block）通常是整数倍的sector，操作系统以页（page）为单位管理内存，一页（page）通常默认为4K，**数据库的页通常设置为操作系统页的整数倍，因此索引结构的节点被设计为一个页的大小**，然后利用外存的“预读取”原则，每次读取的时候，把整个节点的数据读取到内存中，然后在内存中查找，已知内存的读取速度是外存读取I/O速度的几百倍，那么**提升查找速度的关键就在于尽可能少的磁盘I/O，那么可以知道，每个节点中的key个数越多，那么树的高度越小，需要I/O的次数越少，因此一般来说B+Tree比BTree更快，因为B+Tree的非叶节点中不存储data，就可以存储更多的key**。

2、查询速度更稳定

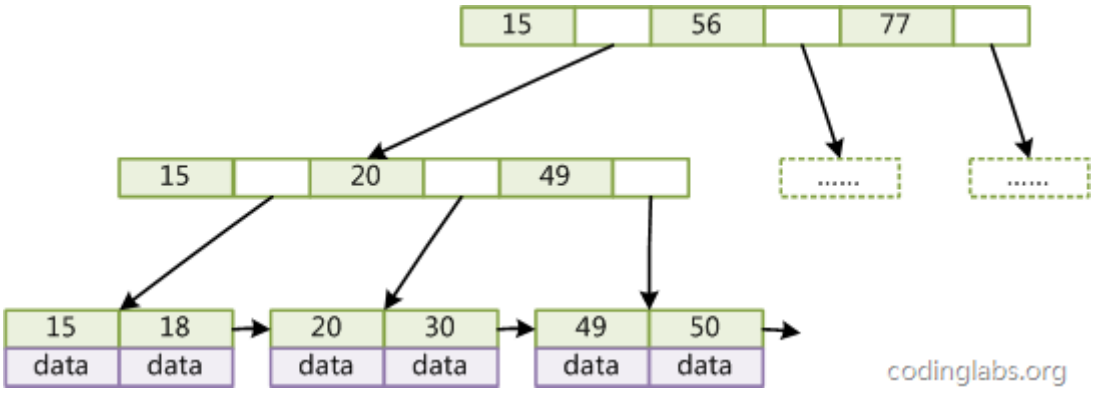
由于B+Tree非叶子节点不存储数据（data），因此所有的数据都要查询至叶子节点，而叶子节点的高度都是相同的，因此所有数据的查询速度都是一样的。

> 原文链接：https://blog.csdn.net/tongdanping/article/details/79878302

5.3 带顺序索引的B+TREE

很多存储引擎在B+Tree的基础上进行了优化，添加了指向相邻叶节点的指针，形成了带有顺序访问指针的B+Tree，这样做是为了提高区间查找的效率，只要找到第一个值那么就可以顺序的查找后面的值。

B+Tree的结构如下：



六、聚簇索引和非聚簇索引

分析了MySQL的索引结构的实现原理，然后我们来看看具体的存储引擎怎么实现索引结构的，MySQL中最常见的两种存储引擎分别是MyISAM和InnoDB，分别实现了非聚簇索引和聚簇索引。

首先要介绍几个概念，在索引的分类中，我们可以按照索引的键是否为主键来分为“主索引”和“辅助索引”，使用主键键值建立的索引称为“主索引”，其它的称为“辅助索引”。因此主索引只能有一个，辅助索引可以有很多个。

InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，聚簇索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。这个特性决定了索引组织表中数据也是索引的一部分；

聚簇索引并不是一种单独的索引类型，而**是一种数据存储方式**。具体细节依赖于其实现方式。

> https://www.cnblogs.com/jiawen010/p/11805241.html

6.1 MyISAM——非聚簇索引

- MyISAM存储引擎采用的是非聚簇索引，非聚簇索引的**主索引和辅助索引几乎是一样的**，只是主索引不允许重复，不允许空值，他们的**叶子结点的key都存储指向键值对应的数据的物理地址**。
- 非聚簇索引的**数据表和索引表是分开存储**的。
- 非聚簇索引中的数据是根据数据的插入顺序保存。因此非聚簇索引更适合单个数据的查询。插入顺序不受键值影响。
- 只有在MyISAM中才能使用FULLTEXT索引。(mysql5.6以后innoDB也支持全文索引)

6.2 InnoDB——聚簇索引

- 聚簇索引的**主索引的叶子结点存储的是键值对应的数据本身，辅助索引的叶子结点存储的是键值对应的数据的主键键值**。因此主键的值长度越小越好，类型越简单越好。
- 聚簇索引的**数据和主键索引存储在一起**。
- 聚簇索引的数据是根据主键的顺序保存。因此适合按主键索引的区间查找，可以有更少的磁盘I/O，加快查询速度。但是也是因为这个原因，聚簇索引的插入顺序最好按照主键单调的顺序插入，否则会频繁的引起页分裂，严重影响性能。
- 在InnoDB中，如果只需要查找索引的列，就尽量不要加入其它的列，这样会提高查询效率。

> * 使用主索引的时候，更适合使用聚簇索引，因为聚簇索引只需要查找一次，而非聚簇索引在查到数据的地址后，还要进行一次I/O查找数据。
> * 因为聚簇辅助索引存储的是主键的键值，因此可以在数据行移动或者页分裂的时候降低成本，因为这时不用维护辅助索引。但是由于主索引存储的是数据本身，因此聚簇索引会占用更多的空间。
> * 聚簇索引在插入新数据的时候比非聚簇索引慢很多，因为插入新数据时需要检测主键是否重复，这需要遍历主索引的所有叶节点，而非聚簇索引的叶节点保存的是数据地址，占用空间少，因此分布集中，查询的时候I/O更少，但聚簇索引的主索引中存储的是数据本身，数据占用空间大，分布范围更大，可能占用好多的扇区，因此需要更多次I/O才能遍历完毕。

> 聚簇索引:https://www.cnblogs.com/jiawen010/p/11805241.html

七、索引使用场景

八、索引优化

8.1 使用explain(执行计划)

作用：

1、表的读取顺序；

2、数据读取操作的操作类型；

3、哪些索引可以使用；

4、那些索引被实际使用；

5、表之间的引用；

6、每张表有多少行被优化器查询；

执行后会有固定几列，如下：

id select_type `table` `type` possible_keys `key` key_len `ref` `rows` Extra

id

说明：**SQL执行的顺序的标识,SQL从大到小的执行**

1. id相同时，执行顺序由上至下

2. 如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行

3.id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行

select_type

说明：**查询中每个select子句的类型**

(1) **SIMPLE**: 简单SELECT,不使用UNION或子查询； 有连接查询时，外层的查询为simple，且只有一个

(2) **PRIMARY:** 一个需要union操作或者含有子查询的select，位于最外层的单位查询的select_type即为primary。且只有一个

(3) **UNION :** UNION中的第二个或后面的SELECT语句 union连接的两个select查询，第一个查询是dervied派生表，除了第一个表外，第二个以后的表select_type都是union 若第二个SELECT出现在UNION之后，则被标记为UNION：若UNION包含在FROM子句的子查询中，外层SELECT将被标记为：DERIVED (4)**DEPENDENT UNION** :与union一样，出现在union 或union all语句中，但是这个查询要受到外部查询的影响

(5) **UNION RESULT** :包含union的结果集，在union和union all语句中,因为它不需要参与查询，所以id字段为null

(6) **SUBQUERY** :除了from字句中包含的子查询外，其他地方出现的子查询都可能是subquery

(7) **DEPENDENT SUBQUERY** :子查询中的第一个SELECT，取决于外面的查询 与dependent union类似，表示这个subquery的查询要受到外部表查询的影响

(8) **DERIVED** :from字句中出现的子查询，也叫做派生表，其他数据库中可能叫做内联视图或嵌套select

(9) **UNCACHEABLE SUBQUERY** :一个子查询的结果不能被缓存，必须重新评估外链接的第一行

table:

说明：显示这一行的数据是关于哪张表的，如果查询使用了别名，那么这里显示的是别名 ,如果不涉及对数据表的操作，那么这显示为null,如果显示为尖括号括起来的<derived N>就表示这个是临时表，后边的N就是执行计划中的id，表示结果来自于这个查询产生。如果是尖括号括起来的<union M,N>，与<derived N>类似，也是一个临时表，表示这个结果来自于union查询的id为M,N的结果集

```
mysql> explain select * from (select * from ( select * from t3 where id=3952602) a) b;
+----+-------------+------------+--------+-----------------+---------+---------+------+------+-------+
| id | select_type | table      | type   | possible_keys   | key     | key_len | ref  | rows | Extra |
+----+-------------+------------+--------+-----------------+---------+---------+------+------+-------+
|  1 | PRIMARY     | <derived2> | system | NULL            | NULL    | NULL    | NULL |    1 |       |
|  2 | DERIVED     | <derived3> | system | NULL            | NULL    | NULL    | NULL |    1 |       |
|  3 | DERIVED     | t3         | const  | PRIMARY,idx_t3_id | PRIMARY | 4       |      |    1 |       |
+----+-------------+------------+--------+-----------------+---------+---------+------+------+-------+
```

type

说明：不同连接类型的解释（**按照效率由高->低的顺序排序**）,除了all之外，其他的type都可以使用到索引，除了index_merge之外，其他的type只可以用到一个索引，**一般来说，得保证查询至少达到range级别，最好能达到ref**

NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

explain select 1 from dual where 1\G #dual是一个虚拟的表，可以直接忽略
select 1+1 from dual;

（system和const 应该是并列的）

system：表中只有一行数据或者是空表，且只能用于myisam和memory表。如果是Innodb引擎表，type列在这个情况通常都是all或者index

const：使用唯一索引或者主键，返回记录一定是1行记录的等值where条件时，通常type是const。其他数据库也叫做唯一索引扫描(where primeryIdField=?或者uniqueIdField=？)
查找主键索引，返回的数据至多一条（0或者1条）。 属于精确查找

explain select * from student where id = 1\G
#注释：如果上表中film表中只有一行数据，那么type就是system。

eq_ref：出现在要连接多个表的查询计划中 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，且这行数据是第二个表的主键或者唯一索引，且必须为not null，唯一索引和主键是多列时，只有所有的列都用作比较时才会出现eq_ref；简单来说，就是多表连接中使用primary key或者 unique key作为关联条件

查找唯一性索引，返回的数据至多一条。属于精确查找
相对于ref来说就是使用的是唯一索引，对于每个索引键值，只有唯一的一条匹配记录（在联表查询中使用primary key或者unique key作为关联条件）
（在film和film_text中film_id都是主键，即都是唯一索引）

mysql> explain select * from film a ,film_text b where a.film_id = b.film_id\G
mysql> ex[lain select * from student as s ,user as u where s.s_mobile=u.u_mobile;

ref：表示表的连接匹配条件，即哪些列或常量被用于查找索引列上的值 这个连接类型只有在查询使用了不是惟一或主键的键或者是这些类型的部分（比如，利用最左边前缀）时发生。对于之前的表的每一个行联合，全部记录都将从表中读出。这个类型严重依赖于根据索引匹配的记录多少—越少越好。（不像eq_ref那样要求连接顺序，也没有主键和唯一索引的要求，只要使用相等条件检索时就可能出现，常见与辅助索引的等值查找。或者多列主键、唯一索引中，使用第一个列之外的

列作为等值查找也会出现，总之，返回数据不唯一的等值查找就可能出现。）　使用非唯一性索引或者唯一索引的前缀扫描，返回匹配某个单独值的记录行
查找非唯一性索引，返回匹配某一条件的多条数据。属于精确查找、数据返回可能是多条
（1）使用非唯一性索引customer_id单表查询
mysql> explain select * from payment where customer_id = 350;

（2）使用非唯一性索引联表查询（由于customer_id在a表中不是主键，是普通索引（非唯一），所以是ref)
mysql> explain select b.*, a.* from payment a ,customer b where a.customer_id = b.customer_id;
fulltext：*全文索引检索，要注意，全文索引的优先级很高，若全文索引和普通索引同时存在时，mysql不管代价，优先选择使用全文索引*
ref_or_null
index_merge:*表示查询使用了两个以上的索引，最后取交集或者并集，常见and ，or的条件使用了不同的索引，官方排序这个在ref_or_null之后，但是实际上由于要读取所个索引，性能可能大部分时间都不如range*
range *索引范围扫描，一个有限制的索引扫描。key 列显示使用了哪个索引。当使用=、 <>、>、>=、<、<=、IS NULL、<=>、BETWEEN 或者 IN 操作符,用常量比较关键字列时,可以使用 range*
只检索给定范围的行，索引范围扫描，即查找某个索引的部分索引；常见于使用=,<>,>,<,>=,<=,<==>,is null,between ,in ,like等运算符的查询中。

```
mysql> explain select * from t3 where id=3952602 or id=3952603 ;
+----+-------------+-------+-------+-----------------+-----------+---------+------+------+-------------+
| id | select_type | table | type  | possible_keys   | key       | key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+-----------------+-----------+---------+------+------+-------------+
|  1 | SIMPLE      | t3    | range | PRIMARY,idx_t3_id | idx_t3_id | 4       | NULL |    2 | Using wher
+----+-------------+-------+-------+-----------------+-----------+---------+------+------+-------------+
```

unique_subquery：
index_subquery：
index：*全索引扫描，index与ALL区别为index类型只遍历索引树 这个连接类型对前面的表中的每一个记录联合进行完全扫描（比ALL更好，因为索引一般小于表数据）。 索引全表扫描，把索引从头到尾扫一遍，常见于使用索引列就可以处理不需要读取数据文件的查询、可以使用索引排序或者分组的查询。*
#如：虽然where条件中没有用到索引，但是要取出的列title是索引包含的列，所以只要全表扫描索引即可，直接使用索引树查找数据
explain select title from film
ALL：*全表扫描， MySQL将遍历全表以找到匹配的行 这个连接类型对于前面的每一个记录联合进行完全扫描，这一般比较糟糕，应该尽量避免。这个就是全表扫描数据文件，然后再在server层进行过滤返回符合要求的记录。*
#如：当我们的where字段无索引时会全表扫描
explain extended sruelect * from film where rating > 9

possible_keys

指出MySQL能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用

该列完全独立于EXPLAIN输出所示的表的次序。这意味着在possible_keys中的某些键实际上不能按生成的表次序使用。
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子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用EXPLAIN检查查询

Key

key列显示MySQL实际决定使用的键（索引） select_type为index_merge时，这里可能出现两个以上的索引，其他的select_type这里只会出现一个
如果没有选择索引，键是NULL。要想强制MySQL使用或忽视possible_keys列中的索引，在查询中使用FORCE INDEX、USE INDEX或者IGNORE INDEX。

> 某个索引出现在 possible_keys , 却没有在key中,则想办法让其生效,或者寻找更好索引

key_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（key_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的）
用于处理查询的索引长度，如果是单列索引，那就整个索引长度算进去，如果是多列索引，那么查询不一定都能使用到所有的列，具体使用到了多少个列的索引，这里就会计算进去，没有使用到的列，这里不会计算进去。留意下这个列的值，算一下你的多列索引总长度就知道有没有使用到所有的列了。要注意，mysql的ICP特性使用到的索引不会计入其中。另外，key_len只计算where条件用到的索引长度，而排序和分组就算用到了索引，也不会计算到key_len中。
不损失精确性的情况下，长度越短越好

ref

表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值 如果是使用的常数等值查询，这里会显示const，如果是连接查询，被驱动表的执行计划这里会显示驱动表的关联字段，如果是条件使用了表达式或者函数，或者条件列发生了内部隐式转换，这里可能显示为func

rows

表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数 这里是执行计划中估算的扫描行数，不是精确值

extra:

说明：extra列返回的描述的意义 **该列包含MySQL解决查询的详细信息,有以下几种情况:**
Distinct：在select部分使用了distinc关键字 一旦mysql找到了与行相联合匹配的行，就不再搜索了。
no tables used：不带from字句的查询或者From dual查询
Not exists ?：mysql优化了LEFT JOIN，一旦它找到了匹配LEFT JOIN标准的行，就不再搜索了。
使用not in()形式子查询或not exists运算符的连接查询，这种叫做反连接。即，一般连接查询是先查询内表，再查询外表，反连接就是先查询外表，再查询内表

Range checked for each Record（index map:#）：没有找到理想的索引，因此对从前面

表中来的每一个行组合，mysql检查使用哪个索引，并用它来从表中返回行。这是使用索引的最慢的连接之一。

Using filesort：看到这个的时候，查询就需要优化了。mysql需要进行额外的步骤来查询如何对返回row(行)排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行。排序时无法使用到索引时，就会出现这个。常见于order by和group by语句中(这是可能是ordery by，group by语句的结果，这可能是一个CPU密集型的过程，可以通过选择合适的索引来改进性能，用索引来为查询结果排序)

Using index：查询时不需要回表查询，直接通过索引就可以获取查询的数据。，这发生在对表的全部的请求列都是同一个索引的部分的时候。 说明查询是覆盖了索引的，不需要读取数据文件，从索引树（索引文件）中即可获得信息。如果同时出现using where，表明索引被用来执行索引键值的查找，没有using where，表明索引用来读取数据而非执行查找动作。这是MySQL服务层完成的，但无需再回表查询记录。

Using temporary：看到这个的时候，查询需要优化了。表示使用了临时表存储中间结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上(常用于GROUP BY 和 ORDER BY操作中??)。 临时表可以是内存临时表和磁盘临时表，执行计划中看不出来，需要查看status变量，used_tmp_table，used_tmp_disk_table才能看出来
Using where:列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤
Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。
using join buffer（block nested loop），using join buffer（batched key accss）：5.6.x之后的版本优化关联查询的BNL，BKA特性。主要是减少内表的循环数量以及比较顺序地扫描查询。
using sort_union，using_union，using intersect，using sort_intersection：
    using intersect：表示使用and的各个索引的条件时，该信息表示是从处理结果获取交集
    using union：表示使用or连接各个使用索引的条件时，该信息表示从处理结果获取并集
    using sort_union和using sort_intersection：与前面两个对应的类似，只是他们是出现在用and和or查询信息量大时，先查询主键，然后进行排序合并后，才能读取记录并返回。

Impossible where：这个值强调了where语句会导致没有符合条件的行。

Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行

Where used：使用了WHERE从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行，并且连接类型ALL或index，这就会发生，或者是查询有问题。
using where：表示存储引擎返回的记录并不是所有的都满足查询条件，需要在server层进行过滤。查询条件中分为限制条件和检查条件，5.6之前，存储引擎只能根据限制条件扫描数据并返回，然后server层根据检查条件进行过滤再返回真正符合查询的数据。5.6.x之后支持ICP特性，可以把检查条件也下推到存储引擎层，不符合检查条件和限制条件的数据，直

接不读取，这样就大大减少了存储引擎扫描的记录数量。extra列显示注意：Extra列出现Using where表示MySQL服务器将存储引擎返回服务层以后再应用WHERE条件过滤
using index condition:这是MySQL 5.6出来的新特性，叫做“索引条件推送”。简单说一点就是MySQL原来在索引上是不能执行如like这样的操作的，但是现在可以了，这样减少了不必要的IO操作，但是只能用在二级索引上。
firstmatch(tb_name)：5.6.x开始引入的优化子查询的新特性之一，常见于where字句含有in()类型的子查询。如果内表的数据量比较大，就可能出现这个
loosescan(m..n)：5.6.x之后引入的优化子查询的新特性之一，在in()类型的子查询中，子查询返回的可能有重复记录时，就可能出现这个
除了这些之外，还有很多查询数据字典库，执行计划过程中就发现不可能存在结果的一些提示信息
select tables optimized away:在没有GROUP BY子句的情况下，基于索引优化MIN/MAX操作，或者对于MyISAM存储引擎优化COUNT(*)操作，不必等到执行阶段再进行计算，查询执行计划生成的阶段即完成优化。

filtered

使用explain extended时会出现这个列，5.7之后的版本默认就有这个字段，不需要使用explain extended了。这个字段表示存储引擎返回的数据在server层过滤后，剩下多少满足查询的记录数量的比例，注意是百分比，不是具体记录数。

partitions

explain **partitions**：相比 explain 多了个 partitions 字段，如果查询是基于分区表的话，会显示查询将访问的分区