什么是索引？为什么要用索引？

1.1、索引的含义

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询，更新数据库中表的数据。索引的实现通常使用B树和变种的B+树（MySQL常用的索引就是B+树）。除了数据之外，数据库系统还维护为满足特定查找算法的数据结构，这些数据结构以某种方式引用数据，这种数据结构就是索引。简言之，索引就类似于书本，字典的目录。

1.2、为什么用索引？

打个比方，如果正确合理设计使用索引的MySQL是一辆兰博基尼的话，那么没有设计和使用索引的MySQL就是一个人力三轮车。对于没有索引的表，单表查询可能几十万数据就是瓶颈，二通常大型网站单日就可能产生几十万甚至几百万的数据，没有索引查询会变得非常缓慢。一言以蔽之，合理使用索引，可以加快数据库的查询效率和提升程序性能

索引的作用与缺点

2.1、作用

通过创建索引，可以再查询的过程中，提高系统的性能
通过创建唯一性索引，可以保持数据库表中每一行数据的唯一性
在使用分组和排序子句进行数据检索时，可以减少查询中分组和排序的时间

2.2、缺点

创建索引和维护索引要耗费时间，而且时间随着数据量的增加而增大
索引需要占用物理空间，如果要建立聚簇索引，所需要的空间会更大
在对表中的数据进行增删改时需要耗费较多的时间，因为索引也要动态地维护

3、索引的使用场景

3.1、应创建索引的场景
主键，unique 字段自动创建索引
经常需要搜索的列上
经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度
经常需要根据范围进行搜索的列上
经常需要排序的列上
经常使用在where子句上面的列上

3.2、不应该创建索引的场景
查询中很少用到的列
对于那些具有很少数据值的列，比如数据表中的性别列，bit数据类型的列
对于那些定义为text，image的列，因为这些列的数据量相当大
当对修改性能的要求远远大于搜索性能时，因为当增加索引时，会提高搜索性能，但是会降低修改性能

索引的分类与说明

4.1、主键索引

设定为主键后数据库会自动建立索引，innodb为聚簇索引

#随表一起建索引：

CREATE TABLE customer (id INT(10) UNSIGNED AUTO_INCREMENT ,customer_no VARCHAR(200),customer_name VARCHAR(200),

PRIMARY KEY(id)

);

#使用AUTO_INCREMENT关键字的列必须有索引(只要有索引就行)。

CREATE TABLE customer2 (id INT(10) UNSIGNED,customer_no VARCHAR(200),customer_name VARCHAR(200),

PRIMARY KEY(id)

);

#单独建主键索引：

ALTER TABLE customer add PRIMARY KEY customer(customer_no);

#删除建主键索引：

ALTER TABLE customer drop PRIMARY KEY ;

#修改建主键索引：

#必须先删除掉(drop)原索引，再新建(add)索引

4.2、单列索引

一个索引只包含单个列，一个表可以有多个单列索引

#随表一起建索引：

CREATE TABLE customer (id INT(10) UNSIGNED AUTO_INCREMENT ,customer_no VARCHAR(200),customer_name VARCHAR(200),

PRIMARY KEY(id),

KEY (customer_name)

);

#随表一起建立的索引 索引名同 列名(customer_name)

#单独建单值索引：

CREATE INDEX idx_customer_name ON customer(customer_name);

#删除索引：

DROP INDEX idx_customer_name ;

4.3、唯一索引

索引列的值必须唯一，但允许有空值

#随表一起建索引：

```
CREATE TABLE customer (id INT(10) UNSIGNED AUTO_INCREMENT ,customer_no
VARCHAR(200),customer_name VARCHAR(200),
PRIMARY KEY(id),
KEY (customer_name),
UNIQUE (customer_no)
);
#建立 唯一索引时必须保证所有的值是唯一的（除了null），若有重复数据，会报错。
#单独建唯一索引：
CREATE UNIQUE INDEX idx_customer_no ON customer(customer_no);
#删除索引：
DROP INDEX idx_customer_no on customer ;
```

4.4、复合索引

一个索引包含多个列，在数据库操作期间，复合索引比单值索引所需要的开销更小（对于相同的多个列建索引）

如果一个表中的数据在查询时有多个字段总是同时出现则这些字段就可以作为复合索引，形成索引覆盖可以提高查询的效率！

```
#随表一起建索引：
CREATE TABLE customer (id INT(10) UNSIGNED AUTO_INCREMENT ,customer_no
VARCHAR(200),customer_name VARCHAR(200),
PRIMARY KEY(id),
KEY (customer_name),
UNIQUE (customer_name),
KEY (customer_no,customer_name)
);
#单独建索引：
CREATE INDEX idx_no_name ON customer(customer_no,customer_name);
#删除索引：
DROP INDEX idx_no_name on customer ;
```

4.5、聚集索引与非聚集索引

4.5.1、聚集索引

聚集索引：指索引项的排序方式和表中数据记录排序方式一致的索引。它会根据聚集索引键的顺序来存储表中的数据，即对表的数据按索引键的顺序进行排序，然后重新存储到磁盘上。因为数据在物理存放只能有一种排列方式，所以一个表只能有一个聚集索引。比如字典中，用拼音查汉字，就是聚集索引。因为正文中字都是按照拼音排序的。而用偏旁部首查汉字，就是非聚集索引，因为正文中的字

并不是按照偏旁部首排序的，我们通过检字表得到正文中的字在索引中的映射，然后通过映射找到所需要的字。
聚集索引的使用场合为：
查询命令的回传结果是以该字段为排序依据的；
查询的结果返回一个区间的值；
查询的结果返回某值相同的大量结果集
聚集索引会降低insert，和update操作的性能，所以，是否使用聚集索引要全面衡量。
4.5.2、非聚集索引
非聚集索引：与聚集索引相反，索引顺序与物理存储顺序不一致
非聚集索引的使用场合为：
查询所获数据量较少时；
某字段中的数据的唯一性比较高时；
非聚集索引必须是稠密

4.6聚簇索引与非聚簇索引
4.6.1、聚簇索引
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。将数据存储于索引放到了一块，索引结构的叶子节点保存了行数据。
聚簇索引的特点：
1、聚簇索引具有唯一性，由于聚簇索引是将数据跟索引结构放到一块，因此一个表仅有一个聚簇索引。
2、表中行的物理顺序和索引中行的物理顺序是相同的，在创建任何非聚簇索引之前创建聚簇索引，这是因为聚簇索引改变了行的物理顺序，数据行，按照一定的顺序排列，并且自动维护这个顺序；
3、聚簇索引默认是主键，如果表中没有定义主键，InnoDB会选择一个唯一且非空的索引代替。如果没有这样的索引，InnoDB会隐式定义一个主键（类似oracle中的Rowld）来作为聚簇索引。如果已经设置了主键为聚簇索引又希望再单独设置聚簇索引，必须先删除主键，然后再添加我们想要的聚簇索引，随后恢复设置主键即可。
4.6.2、非聚簇索引
不是聚簇索引的二级索引，也叫作辅助索引，都称为非聚簇索引。将数据与索引分开存储，索引结构的叶子节点指向了数据对应的位置。

索引的底层原理
抛开其他的数据库索引实现，主讲MySQL的索引底层实现，其底层是通过B+树来实现的数据结构存储。
数据结构存储，决定了数据查找和操作时的效率，包括时间复杂度和空间复杂度，而在取舍的时候，也无非就是时间换空间，空间换时间的权衡罢了，所以，这就很好的解释了，为什么MySQL在索引的

底层设计上，选用了B+树，而没有选用B-树，或是红黑树，AVL树等等其他数据结构。总之，就是使用B+树作为索引的结构存储，能在I/O性能上得到一个较大的优势。

那么具体优势在哪里呢？以B-树与B+树的对比，来阐述具体差异和B+树的优势。

5.1、B-Tree

B-树是一种多路自平衡的搜索树，它类似普通的平衡二叉树，不同的一点是B-树允许每个节点有更多的子节点。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

注：B-Tree就是我们常说的B树

那么m阶B-Tree是满足下列条件的数据结构：

所有键值分布在整棵树中

搜索有可能在非叶子节点结束，在关键字全集内做一次查找，性能逼近二分查找

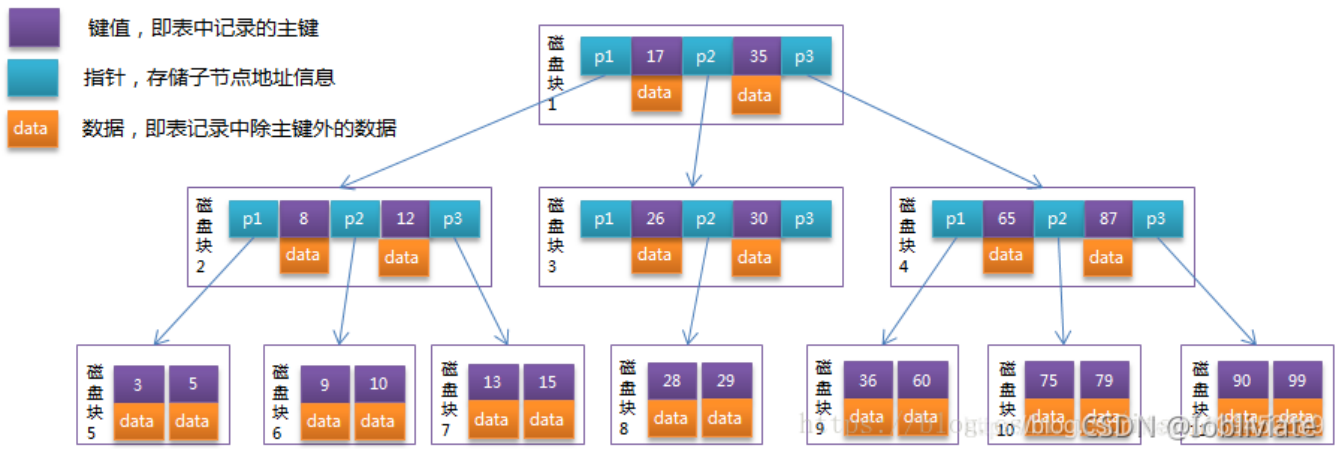
每个节点最多拥有m个子树

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

每个节点占用一个磁盘块，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围大于35.



模拟查找关键字29的过程：

1、根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】

2、比较关键字29在区间（17,35），找到磁盘块1的指针P2。

3、根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】

4、比较关键字29在区间（26,30），找到磁盘块3的指针P2。

5、根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】

6、在磁盘块8中的关键字列表中找到关键字29。

7、分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。

但同时B-Tree也存在问题：

每个节点中有key，也有data，而每一个页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（即一个页）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

5.2、B+Tree

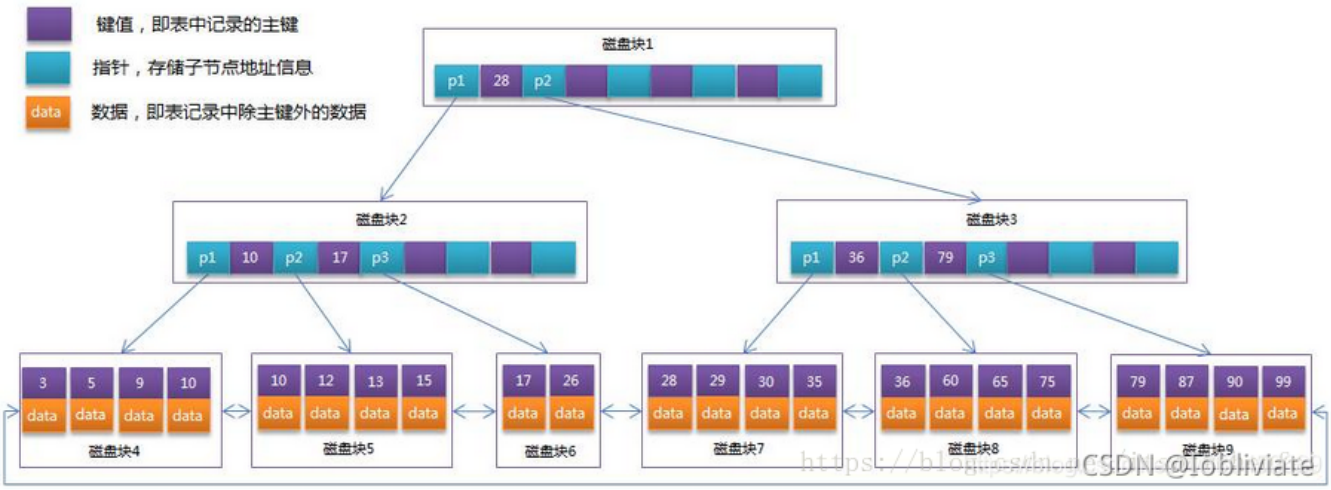
B+Tree是在B-Tree基础上的一种优化，InnoDB存储引擎就是用B+Tree实现其索引结构。它带来的变化点：

B+树每个节点可以包含更多的节点，这样做有两个原因，一个是降低树的高度。另外一个是将数据范围变为多个区间，区间越多，数据检索越快

非叶子节点存储key，叶子节点存储key和数据

叶子节点两两指针相互链接（符合磁盘的预读特性），顺序查询性能更高

注：MySQL的InnoDB存储引擎在设计时是将根节点常驻内存，因此力求达到树的深度不超过3，也就是说I/O不需要超过3次。



通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（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：一种是对于主键的范围查找的分页查找，另一种是从根节点开始，进行随机查找。

5.3、B-树和B+树的区别

B+树内节点不存储数据，所有数据存储在叶节点导致查询时间复杂度固定为log n

B-树查询时间复杂度不固定，与Key在树中的位置有关，最好为O(1)

B+树叶节点两两相连可大大增加区间访问性，可使用在范围查询等

B+树更适合外部存储（存储磁盘数据）。由于内节点无data域，每个节点能索引的范围更大更精确。

原文链接：https://blog.csdn.net/qq_44483424/article/details/121385545

索引失效的情况

https://www.jianshu.com/p/ff3470407b19

1、where 条件后以通配符开头（优化：尽量避免用通配符开始）

2.列类型是字符串，查询条件未加引号。

3.未使用该列作为查询条件

4. 在查询条件中使用OR，要想是索引生效，需要将or中的每个列都加上索引

5.对索引列进行函数运算/数值运算（优化建议，尽量在应用程序中进行计算和转换。）

6、使用复合索引，没有使用最左侧的列查找