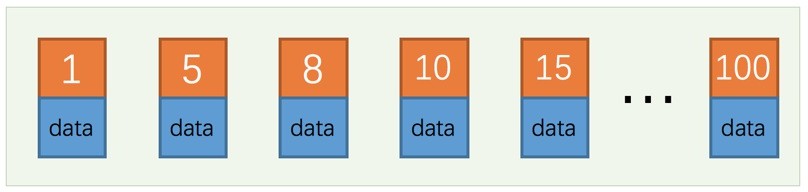
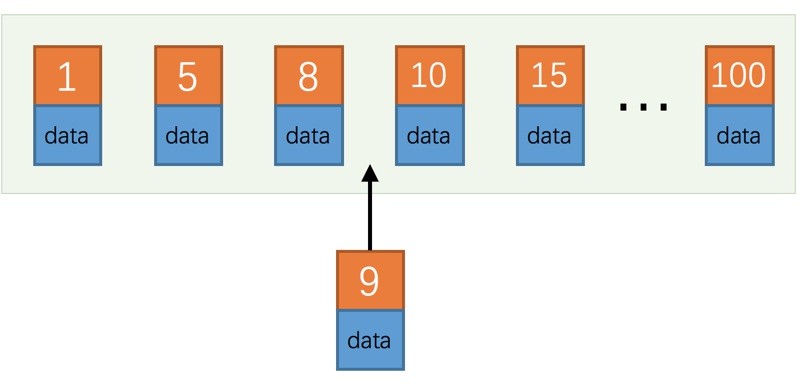
## 主键插⼊顺序

我们知道，对于⼀个使⽤InnoDB存储引擎的表来说，在我们没有显式的创建索引时，表中的数据实际上都是存储在聚簇索引的叶⼦节点的。⽽记录⼜是存储在数据⻚中的，数据⻚和记录⼜是按照记录主键值从⼩到⼤的顺序进⾏排序，所以如果我们插⼊的记录的主键值是依次增⼤的话，那我们每插满⼀个数据⻚就换到下⼀个数据⻚继续插，

⽽如果我们插⼊的主键值忽⼤忽⼩的话，这就⽐较麻烦了，假设某个数据⻚存储的记录已经满了，它存储的主键值在1~100之间：



如果此时再插⼊⼀条主键值为9的记录，那它插⼊的位置就如下图：



可这个数据⻚已经满了啊，再插进来咋办呢？我们需要把当前⻚⾯分裂成两个⻚⾯，把本⻚中的⼀些记录移动到新创建的这个⻚中。⻚⾯分裂和记录移位意味着什么？意味着：性能损耗！所以如果我们想尽量避免这样⽆谓的性能损耗，最好让插⼊的记录的主键值依次递增， 这样就不会发⽣这样的性能损耗了。所以我们建议：让主键具

有AUTO\_INCREMENT，让存储引擎⾃⼰为表⽣成主键，⽽不是我们

⼿动插⼊ ，⽐⽅说我们可以这样定义person\_info表：

CREATE TABLE person\_info(

id INT UNSIGNED NOT NULL AUTO\_INCREMENT, name VARCHAR(100) NOT NULL,

birthday DATE NOT NULL, phone\_number CHAR(11) NOT NULL, country varchar(100) NOT NULL, PRIMARY KEY (id),

KEY idx\_name\_birthday\_phone\_number (name(10), birthday, phone\_number)

);

我们⾃定义的主键列id拥有AUTO\_INCREMENT属性，在插⼊记录时存储引擎会⾃动为我们填⼊⾃增的主键值。

## 冗余和重复索引

有时候有的同学有意或者⽆意的就对同⼀个列创建了多个索引，⽐⽅说这样写建表语句：

CREATE TABLE person\_info(

id INT UNSIGNED NOT NULL AUTO\_INCREMENT, name VARCHAR(100) NOT NULL,

birthday DATE NOT NULL, phone\_number CHAR(11) NOT NULL, country varchar(100) NOT NULL, PRIMARY KEY (id),

KEY idx\_name\_birthday\_phone\_number (name(10), birthday, phone\_number),

KEY idx\_name (name(10))

);

我们知道，通过idx\_name\_birthday\_phone\_number索引就可以对name列进⾏快速搜索，再创建⼀个专⻔针对name列的索引就算是

⼀个冗余索引，维护这个索引只会增加维护的成本，并不会对搜索有什么好处。

另⼀种情况，我们可能会对某个列重复建⽴索引，⽐⽅说这样：

CREATE TABLE repeat\_index\_demo ( c1 INT PRIMARY KEY,

c2 INT,

UNIQUE uidx\_c1 (c1), INDEX idx\_c1 (c1)

);

我们看到，c1既是主键、⼜给它定义为⼀个唯⼀索引，还给它定义了⼀个普通索引，可是主键本身就会⽣成聚簇索引，所以定义的唯⼀索引和普通索引是重复的，这种情况要避免。

# 总结

上边只是我们在创建和使⽤B+树索引的过程中需要注意的⼀些点， 后边我们还会陆续介绍更多的优化⽅法和注意事项，敬请期待。本集内容总结如下：

1. B+树索引在空间和时间上都有代价，所以没事⼉别瞎建索引。
2. B+树索引适⽤于下边这些情况：

全值匹配

匹配左边的列匹配范围值

精确匹配某⼀列并范围匹配另外⼀列

⽤于排序

⽤于分组

1. 在使⽤索引时需要注意下边这些事项：

只为⽤于搜索、排序或分组的列创建索引为列的基数⼤的列创建索引

索引列的类型尽量⼩

可以只对字符串值的前缀建⽴索引

只有索引列在⽐较表达式中单独出现才可以适⽤索引

为了尽可能少的让聚簇索引发⽣⻚⾯分裂和记录移位的情况，建议让主键拥有AUTO\_INCREMENT属性。

定位并删除表中的重复和冗余索引

尽量使⽤覆盖索引进⾏查询，避免回表带来的性能损耗。