MySQL 单表不要超过 2000W 行，靠谱吗?

作为在后端圈开车的多年老司机，是不是经常听到过：

- “MySQL 单表最好不要超过 2000W”
- “单表超过 2000W 就要考虑数据迁移了”
- “你这个表数据都马上要到 2000W 了，难怪查询速度慢”

这些名言民语就和 “群里只讨论技术，不开车，开车速度不要超过 120 码，否则自动踢群”，只听过，没试过，哈哈。

下面我们就把车速踩到底，干到 180 码试试.....

> 原文链接：https://my.oschina.net/u/4090830/blog/5559454

实验

实验一把看看... 建一张表

```
CREATE TABLE person(
    id int NOT NULL AUTO_INCREMENT PRIMARY KEY comment '主键',
    person_id tinyint not null comment '用户id',
    person_name VARCHAR(200) comment '用户名称',
    gmt_create datetime comment '创建时间',
    gmt_modified datetime comment '修改时间'
) comment '人员信息表';
```

插入一条数据

```
insert into person values(1, 1,'user_1', NOW(), now());
```

利用 MySQL 伪列 rownum 设置伪列起始点为 1

```
select (@i:=@i+1) as rownum, person_name from person, (select @i:=100) as init
set @i=1;
```

运行下面的 sql，连续执行 20 次，就是 2 的 20 次方约等于 100w 的数据；执行 23 次就是 2 的 23 次方约等于 800w , 如此下去即可实现千万测试数据的插入。

如果不想翻倍翻倍的增加数据，而是想少量，少量的增加，有个技巧，就是在 SQL 的后面增加 where 条件，如 id > 某一个值去控制增加的数据量即可。

```
insert into person(id, person_id, person_name, gmt_create, gmt_modified)
select @i:=@i+1,
left(rand()*10,10) as person_id,
concat('user_',@i%2048),
date_add(gmt_create,interval + @i*cast(rand()*100 as signed) SECOND),
date_add(date_add(gmt_modified,interval +@i*cast(rand()*100 as signed) SECOND)
from person;
```

此处需要注意的是，也许你在执行到近 800w 或者 1000w 数据的时候，会报错：The total number of locks exceeds the lock table size。

这是由于你的临时表内存设置的不够大，只需要扩大一下设置参数即可。

```
SET GLOBAL tmp_table_size =512*1024*1024;  (512M)
SET global innodb_buffer_pool_size= 1*1024*1024*1024 (1G);
```

先来看一组测试数据，这组数据是在 MySQL 8.0 的版本，并且是在我本机上，由于本机还跑着 idea , 浏览器等各种工具，所以并不是机器配置就是用于数据库配置，所以测试数据只限于参考。

数据量：100w，count 查询耗时：0.046s

```
SELECT COUNT(1) FROM  person;
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 0.046 秒. */
```

数据量：100w,条件查询耗时：0.219s

```
SELECT COUNT(*)  FROM  person   WHERE person_id='9';
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 0.219 秒. */
```

数据量：500w，count 查询耗时：0.234s

```
SELECT COUNT(1) FROM  person;
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 0.234 秒. */
```

数据量：500w，count 查询耗时：1s

```
SELECT COUNT(*)  FROM  person   WHERE person_id='9';
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 1.031 秒. */
```

数据量：1000w，count(1) 耗时：0.46s

```
SELECT COUNT(1) FROM  person;
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 0.469 秒. */
```

数据量：1000w，条件查询耗时：2.1s

```
SELECT COUNT(*)  FROM  person   WHERE person_id='9';
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 2.125 秒. */
```

数据量：2000w，count(1)查询耗时：18s

```
SELECT COUNT(1) FROM  person;
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 18.750 秒. */
```

数据量：2000w，条件查询耗时：5.57s

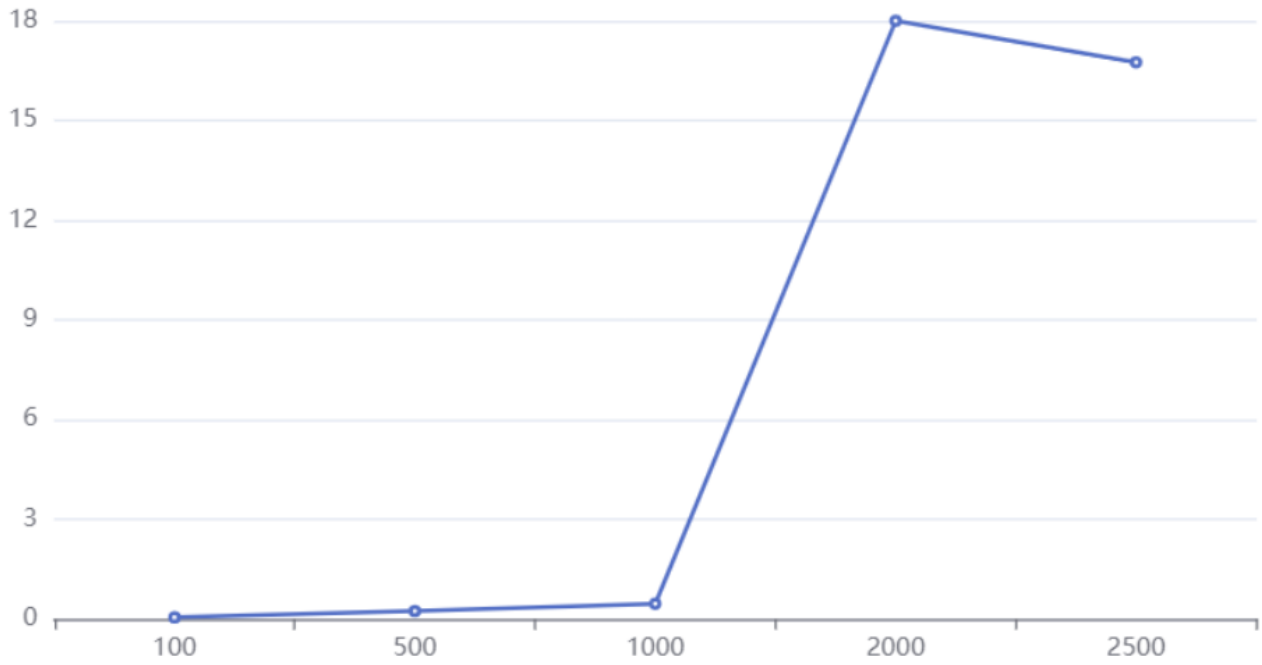
```
SELECT COUNT(*)  FROM  person   WHERE person_id='9';
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 5.578 秒. */
```

数据量：2400w，count 查询：16.75s

```
SELECT COUNT(1) FROM  person;
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 16.750 秒. */
```

数据量：2400w，条件查询耗时：5.98s

```
SELECT COUNT(*)  FROM  person   WHERE person_id='9';
/* 受影响记录行数: 0  已找到记录行: 1  警告: 0  持续时间 1 查询: 5.985 秒. */
```



看到这组数据似乎好像真的和标题对应，当数据达到 2000W 以后，查询时长急剧上升，难道这就是铁律吗?

那下面我们就来看看这个建议值 2000W 是怎么来的?

单表数量限制

首先我们先想想数据库单表行数最大多大?

```
CREATE TABLE person(
    id int(10) NOT NULL AUTO_INCREMENT PRIMARY KEY comment '主键',
    person_id tinyint not null comment '用户id',
    person_name VARCHAR(200) comment '用户名称',
    gmt_create datetime comment '创建时间',
    gmt_modified datetime comment '修改时间'
) comment '人员信息表';
```

看看上面的建表 sql。id 是主键，本身就是唯一的，也就是说主键的大小可以限制表的上限：

- 如果主键声明 `int` 类型，也就是 32 位，那么支持 2^32-1 ~~21 亿；

- 如果主键声明 `bigint` 类型，那就是 2^62-1 （36893488147419103232），难以想象这个的多大了，一般还没有到这个限制之前，可能数据库已经爆满了！！

有人统计过，如果建表的时候，自增字段选择无符号的 bigint , 那么自增长最大值是 18446744073709551615，按照一秒新增一条记录的速度，大约什么时候能用完?

一秒增加的记录数	大约多少年用完
1/1秒	584942417355 年
1w/秒	58494241 年
100w/秒	584942年
1亿/秒	5849年

表空间

下面我们再来看看索引的结构，我们下面讲内容都是基于 Innodb 引擎的，大家都知道 Innodb 的索引内部用的是 B+ 树。

id	person_id	person_name	gmt_create	gmt_modified
1	1	user_1	2022-04-12 16:26:14	2022-04-12 16:26:14
2	6	user_2	2022-04-12 16:26:54	2022-04-23 16:37:15
3	8	user_3	2022-04-12 16:27:41	2022-04-23 05:17:29
4	7	user_4	2022-04-12 16:31:34	2022-04-24 19:14:02
5	2	user_5	2022-04-12 16:30:34	2022-04-13 18:05:40
6	1	user_6	2022-04-12 16:27:18	2022-05-04 06:16:49
7	5	user_7	2022-04-12 16:39:21	2022-05-01 19:00:09
8	7	user_8	2022-04-12 16:32:46	2022-05-05 22:23:49
9	5	user_9	2022-04-12 16:33:44	2022-04-22 00:31:47

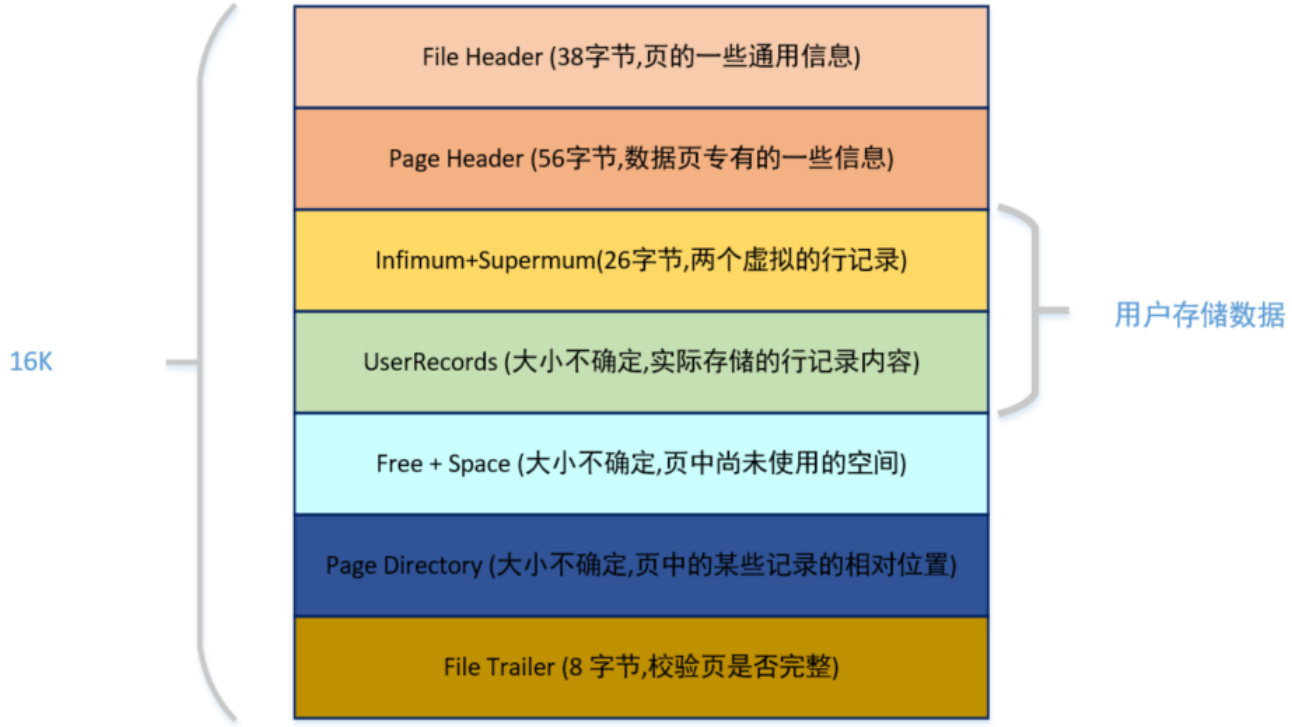
这张表数据，在硬盘上存储也是类似如此的，它实际是放在一个叫 person.ibd （innodb data）的文件中，也叫做表空间；虽然数据表中，他们看起来是一条连着一条，但是实际上在文件中它被分成很多小份的数据页，而且每一份都是 16K。

大概就像下面这样，当然这只是我们抽象出来的，在表空间中还有段、区、组等很多概念，但是我们需要跳出来看。

ibd文件内部有大量的页

页的数据结构

实际页的内部结构像是下面这样的：



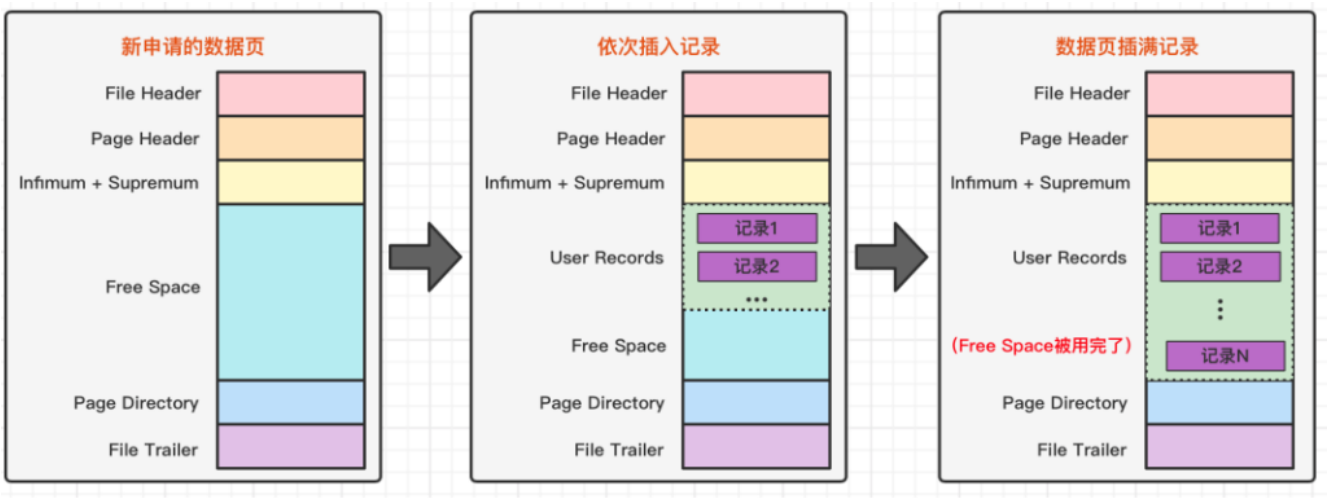
从图中可以看出，一个 InnoDB 数据页的存储空间大致被划分成了 7 个部分，有的部分占用的字节数是确定的，有的部分占用的字节数是不确定的。

在页的 7 个组成部分中，我们自己存储的记录会按照我们指定的行格式存储到 `User Records` 部分。

但是在一开始生成页的时候，其实并没有 User Records 这个部分，每当我们插入一条记录，都会从 Free Space 部分，也就是尚未使用的存储空间中申请一个记录大小的空间划分到 User Records 部分。

当 Free Space 部分的空间全部被 User Records 部分替代掉之后，也就意味着这个页使用完了，如果还有新的记录插入的话，就需要去申请新的页了。

这个过程的图示如下：



刚刚上面说到了数据的新增的过程。

那下面就来说说，数据的查找过程，假如我们需要查找一条记录，我们可以把表空间中的每一页都加载到内存中，然后对记录挨个判断是不是我们想要的。

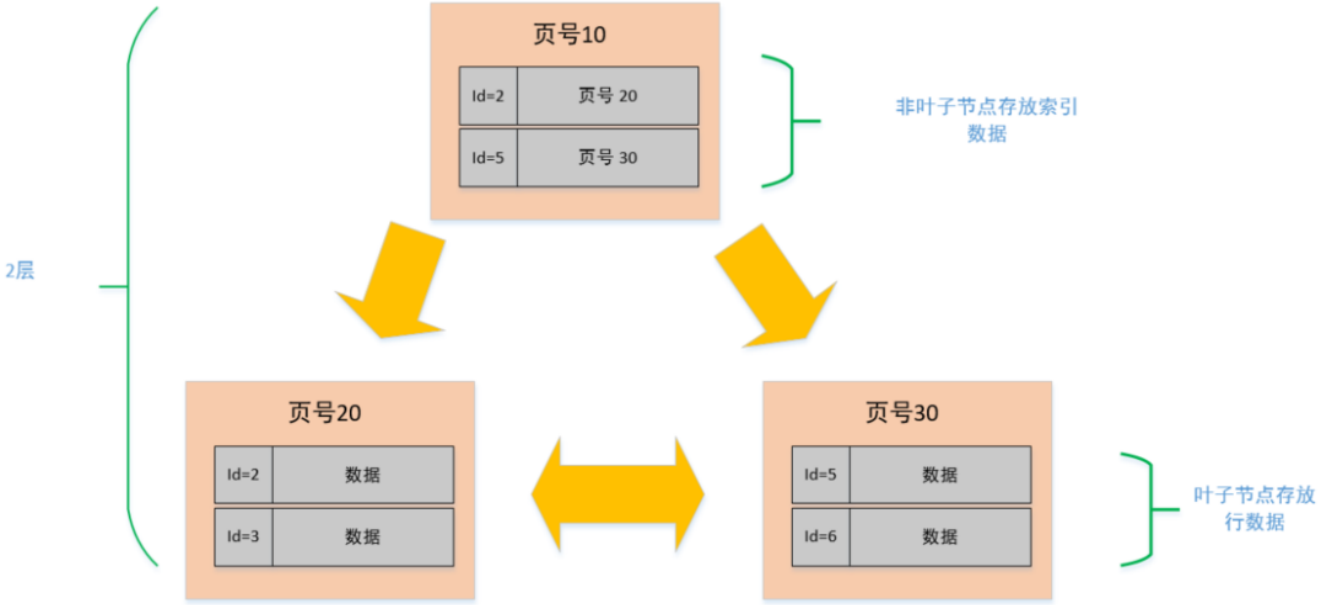
在数据量小的时候，没啥问题，内存也可以撑。但是现实就是这么残酷，不会给你这个局面。

为了解决这问题，MySQL 中就有了索引的概念，大家都知道索引能够加快数据的查询，那到底是怎么个回事呢？下面我就来看看。

索引的数据结构

在 MySQL 中索引的数据结构和刚刚描述的页几乎是一模一样的，而且大小也是 16K,。

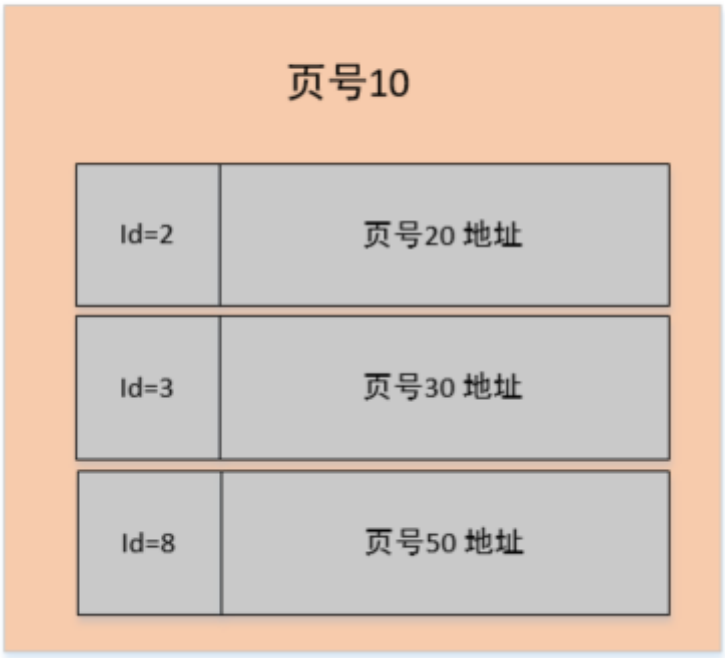
但是在索引页中记录的是页 (数据页，索引页) 的最小主键 id 和页号，以及在索引页中增加了层级的信息，从 0 开始往上算，所以页与页之间就有了上下层级的概念。



看到这个图之后，是不是有点似曾相似的感觉，是不是像一棵二叉树啊，对，没错！它就是一棵树。

只不过我们在这里只是简单画了三个节点，2 层结构的而已，如果数据多了，可能就会扩展到 3 层的树，这个就是我们常说的 B+ 树，最下面那一层的 page level =0, 也就是叶子节点，其余都是非叶子节点。

索引页



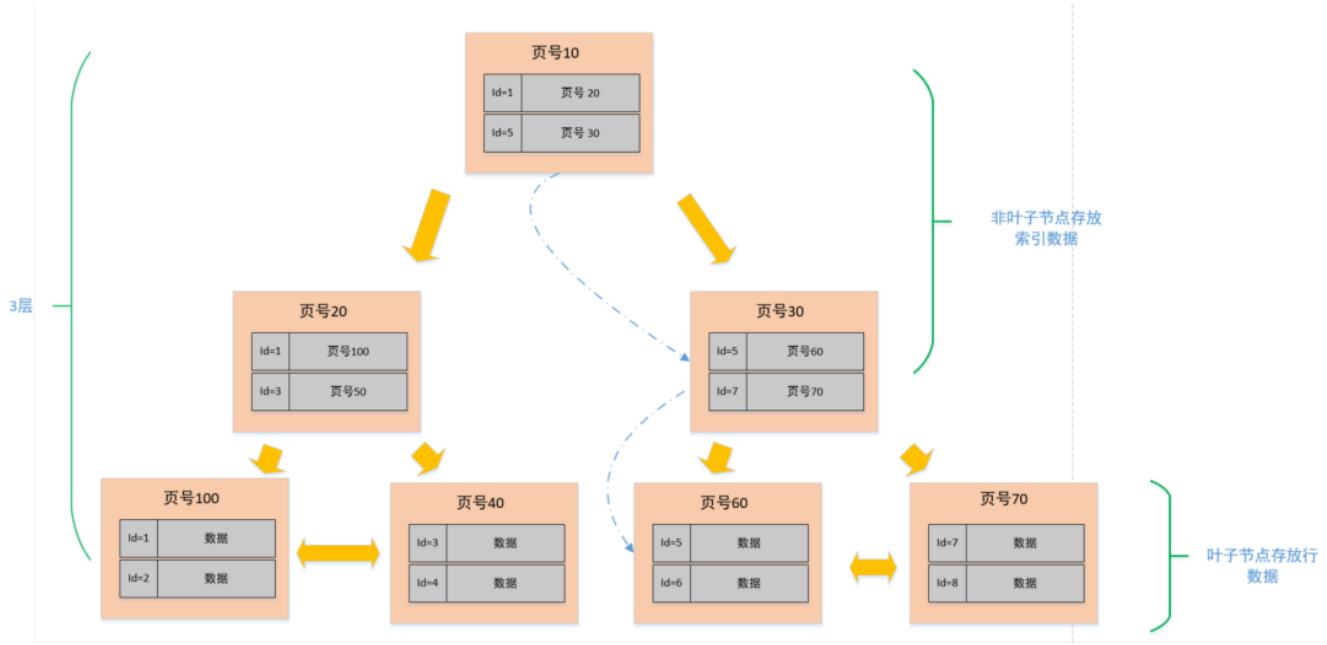
看上图中，我们是单拿一个节点来看，首先它是一个非叶子节点（索引页），在它的内容区中有 id 和 页号地址两部分：

- id ： 对应页中记录的最小记录 id 值；
- 页号：地址是指向对应页的指针；

而数据页与此几乎大同小异，区别在于数据页记录的是真实的行数据而不是页地址，而且 id 的也是顺序的。

单表建议值

下面我们就以 3 层，2 分叉（实际中是 M 分叉）的图例来说明一下查找一个行数据的过程。



比如说我们需要查找一个 id=6 的行数据：

- 因为在非叶子节点中存放的是页号和该页最小的 id，所以我们从顶层开始对比，首先看页号 10 中的目录，有 [id=1, 页号 = 20],[id=5, 页号 = 30], 说明左侧节点最小 id 为 1，右侧节点最小 id 是 5。6>5, 那按照二分法查找的规则，肯定就往右侧节点继续查找；
- 找到页号 30 的节点后，发现这个节点还有子节点（非叶子节点），那就继续比对，同理，6>5 && 6<7, 所以找到了页号 60；
- 找到页号 60 之后，发现此节点为叶子节点（数据节点），于是将此页数据加载至内存进行一一对比，结果找到了 id=6 的数据行。

从上述的过程中发现，我们为了查找 id=6 的数据，总共查询了三个页，如果三个页都在磁盘中（未提前加载至内存），那么最多需要经历三次的磁盘 IO。

需要注意的是，图中的页号只是个示例，实际情况下并不是连续的，在磁盘中存储也不一定是顺序的。

至此，我们大概已经了解了表的数据是怎么个结构了，也大概知道查询数据是个怎么的过程了，这样我们也就能大概估算这样的结构能存放多少数据了。

从上面的图解我们知道 B+ 数的叶子节点才是存在数据的，而非叶子节点是用来存放索引数据的。

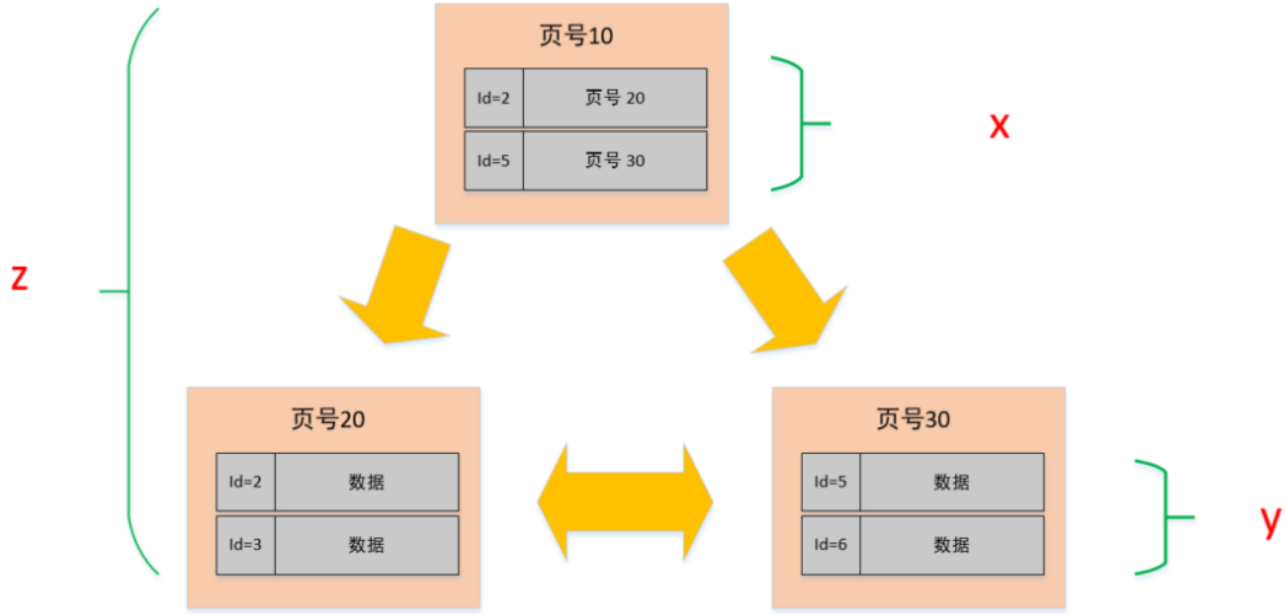
所以，同样一个 16K 的页，非叶子节点里的每条数据都指向新的页，而新的页有两种可能

- 如果是叶子节点，那么里面就是一行行的数据
- 如果是非叶子节点的话，那么就会继续指向新的页

假设

- 非叶子节点内指向其他页的数量为 x
- 叶子节点内能容纳的数据行数为 y
- B+ 数的层数为 z

如下图中所示，**Total =x^(z-1) *y 也就是说总数会等于 x 的 z-1 次方 与 Y 的乘积**。



> X =?

在文章的开头已经介绍了页的结构，索引也也不例外，都会有 File Header (38 byte)、Page Header (56 Byte)、Infimum + Supermum（26 byte）、File Trailer（8byte），再加上页目录，大概 1k 左右。

我们就当做它就是 1K, 那整个页的大小是 16K, 剩下 15k 用于存数据，在索引页中主要记录的是主键与页号，主键我们假设是 Bigint (8 byte), 而页号也是固定的（4Byte），那么索引页中的一条数据也就是 12byte。

所以 x=15*1024/12≈1280 行。

> Y=?

叶子节点和非叶子节点的结构是一样的，同理，能放数据的空间也是 15k。

但是叶子节点中存放的是真正的行数据，这个影响的因素就会多很多，比如，字段的类型，字段的数量。每行数据占用空间越大，页中所放的行数量就会越少。

这边我们暂时按一条行数据 1k 来算，那一页就能存下 15 条，Y = 15*1024/1000 ≈15。

算到这边了，是不是心里已经有谱了啊。

根据上述的公式，Total =x^(z-1) *y，已知 x=1280，y=15：

- 假设 B+ 树是两层，那就是 z = 2， Total = （1280 ^1 ）*15 = 19200
- 假设 B+ 树是三层，那就是 z = 3， Total = （1280 ^2） *15 = 24576000 （约 2.45kw）

哎呀，妈呀！这不是正好就是文章开头说的最大行数建议值 2000W 嘛！对的，一般 B+ 数的层级最多也就是 3 层。

你试想一下，如果是 4 层，除了查询的时候磁盘 IO 次数会增加，而且这个 Total 值会是多少，大概应该是 3 百多亿吧，也不太合理，所以，3 层应该是比较合理的一个值。

> 到这里难道就完了？

不。

我们刚刚在说 Y 的值时候假设的是 1K ，那比如我实际当行的数据占用空间不是 1K , 而是 5K, 那么单个数据页最多只能放下 3 条数据。

同样，还是按照 z = 3 的值来计算，那 Total = （1280 ^2） *3 = 4915200 （近 500w）

所以，在保持相同的层级（相似查询性能）的情况下，在行数据大小不同的情况下，其实这个最大建议值也是不同的，而且影响查询性能的还有很多其他因素，比如，数据库版本，服务器配置，sql 的编写等等。

MySQL 为了提高性能，会将表的索引装载到内存中，在 InnoDB buffer size 足够的情况下，其能完成全加载进内存，查询不会有问题。

但是，当单表数据库到达某个量级的上限时，导致内存无法存储其索引，使得之后的 SQL 查询会产生磁盘 IO，从而导致性能下降，所以增加硬件配置（比如把内存当磁盘使），可能会带来立竿见影的性能提升哈。

总结

- MySQL 的表数据是以页的形式存放的，页在磁盘中不一定是连续的。
- 页的空间是 16K, 并不是所有的空间都是用来存放数据的，会有一些固定的信息，如，页头，页尾，页码，校验码等等。
- 在 B+ 树中，叶子节点和非叶子节点的数据结构是一样的，区别在于，叶子节点存放的是实际的行数据，而非叶子节点存放的是主键和页号。
- 索引结构不会影响单表最大行数，2000W 也只是推荐值，超过了这个值可能会导致 B + 树层级更高，影响查询性能。
