为什么大家说 MySQL 数据库单表最大两千万？依据是啥？

转载 程序员乔戈里 于 2022-05-03 23:59:55 发布 761 收藏 4 版权

文章标签： mysql java 数据库 数据结构 python

故事从好多年前说起。

想必大家也听说过数据库单表建议最大两千万条数据这个说法。如果超过了，性能就会下降得比较厉害。

巧了。我也听说过。

但我不接受它的建议，硬是单表装了 1 亿条数据。

这时候，我们组里新来的实习生看到了之后，天真无邪地问我："单表不是建议最大两千万吗？为什么这个表都放了 1 个亿还不 分库分表 "?

我能说我是因为懒吗？我当初设计时哪里想到这表竟然能涨这么快……

我不能。

说了等于承认自己是开发组里的毒瘤，虽然我确实是，但我不能承认。

我如坐针毡，如芒刺背，如鲠在喉。

开始了一波骚操作。

"我这么做是有道理的。"

"虽然这个表很大，但你有没有发现它查询其实还是很快。"

"这个两千万是个建议值，我们要来看下这个两千万是怎么来的。"

数据库单表行数最大多大?

我们先看下单表行数理论最大值是多少。

建表的 SQL 是这么写的，

```
CREATE TABLE `user` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `name` varchar(100) NOT NULL DEFAULT '' COMMENT '名字',
  `age` int(11) NOT NULL DEFAULT '0' COMMENT '年龄',
  PRIMARY KEY (`id`),
  KEY `idx_age` (`age`)
```

```
) ENGINE=InnoDB AUTO_INCREMENT=10003 DEFAULT CHARSET=utf8;
```

其中 id 就是主键。主键本身唯一，也就是说主键的大小可以限制表的上限。

如果主键声明为 int 大小，也就是 32 位。那么能支持 2^32-1，也就是 21 个亿左右。

如果是 bigint，那就是 2^64-1。但这个数字太大，一般还没到这个限制之前，磁盘先受不了。

搞离谱点。

如果我把主键声明为 tinyint 一个字节， 8位。最大 2^8-1，也就是 255。

```
CREATE TABLE `user` (
  `id` tinyint(2) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `name` varchar(100) NOT NULL DEFAULT '' COMMENT '名字',
  `age` int(11) NOT NULL DEFAULT '0' COMMENT '年龄',
  PRIMARY KEY (`id`),
  KEY `idx_age` (`age`)
) ENGINE=InnoDB AUTO_INCREMENT=0 DEFAULT CHARSET=utf8;
```

如果我想插入一个 id=256 的数据，那就会报错。

```
mysql> INSERT INTO `tmp` (`id`, `name`, `age`) VALUES (256, '', 60);
ERROR 1264 (22003): Out of range value for column 'id' at row 1
```

也就是说，tinyint 主键限制表内最多 255 条数据。

除了主键，还有哪些因素会影响行数?

索引的结构

索引内部是用的 B+ 树，这个也是八股文 老股了，大家估计也背得很熟了。

为了不让大家有过于强烈的审丑疲劳，今天我尝试从另外一个角度给大家讲讲这玩意。

页的结构

假设我们有这么一张 user 数据表。

id	name	age
1	小白	1
2	小黑	2
3	小红	3
4	小绿	4
5	小蓝	5

user 表

其中 id 是唯一主键。

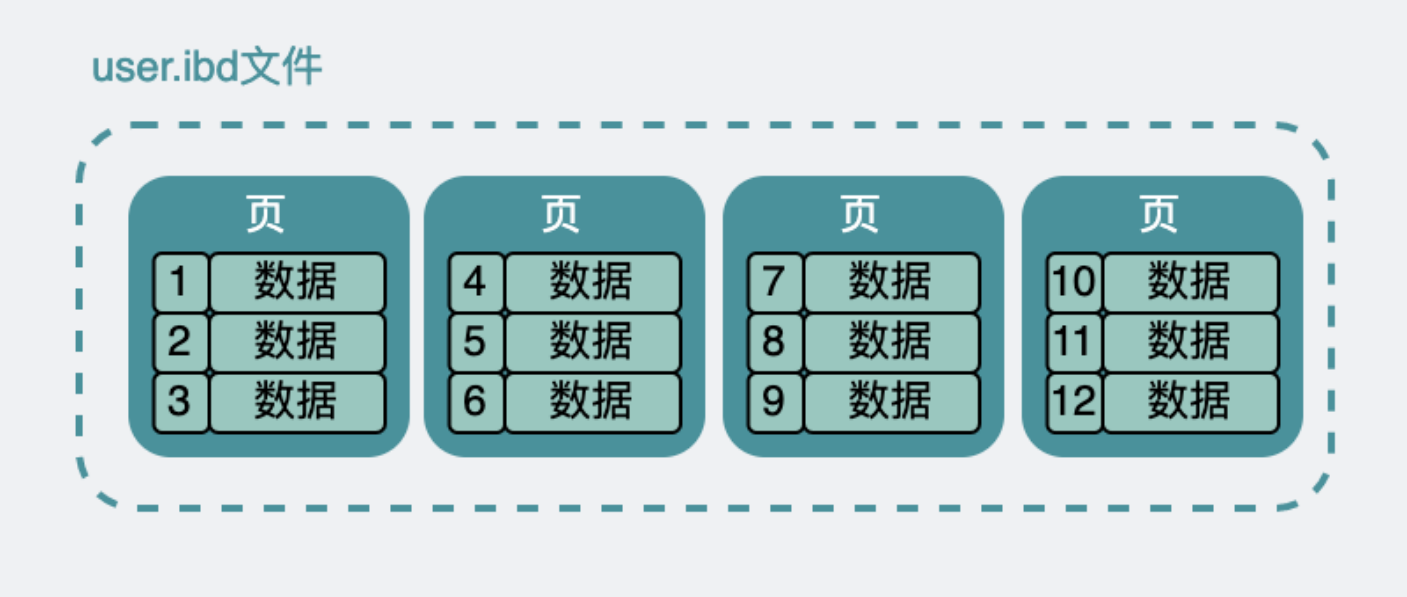
这看起来的一行行数据，为了方便，我们后面就叫它们 record 吧。

这张表看起来就跟个 Excel 表格一样。Excel 的数据在硬盘上是一个 xx.xlsx 文件。

而上面 user 表数据，在硬盘上其实也是类似，放在了 user.ibd 文件下。含义是 user 表的 innodb data 文件，专业说法又叫表空间。

虽然在数据表里，它们看起来是挨在一起的。但实际上在 user.ibd 里他们被分成很多小数据页，每份大小16K。

类似于下面这样：



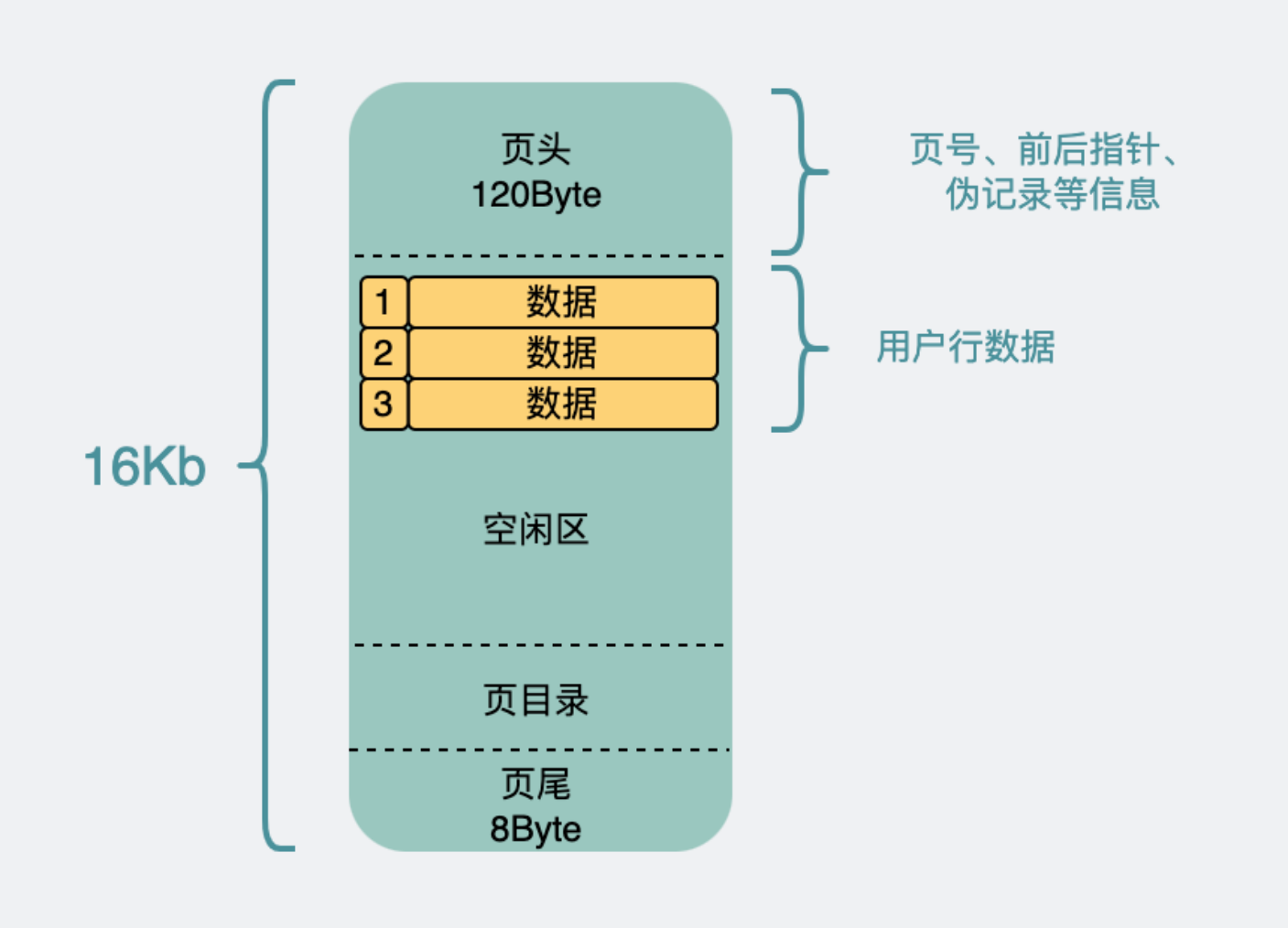
ibd 文件内部有大量的页
我们把视角聚焦到页上面。

整个页大小为 16K，不大。但 record 这么多，一页肯定放不下，所以会分开放到很多页里。并且这 16K 也不可能全用来放 record，对吧。

因为，这些 record 被分成好多份，放到各个页里了。为了唯一标识具体是哪一页，那就需要引入页号（其实是一个表空间的地址偏移量）。同时为了把这些数据页给关联起来，于是引入了前后指针，用于指向前后的页。这些都被加到了页头里。

页需要支持读写，16K 说小也不小，写一半电源线被拔了也是有可能发生的。所以，为了保证数据页的正确性，还引入了校验码。这个被加到了页尾。

那剩下的空间才被用来放 record。如果 record 行数特别多，进入到页内时会挨个遍历效率也不太行。所以，为这些数据生成了一个页目录。具体实现细节不重要，只需要知道，它可以通过二分查找的方式将查找效率从 O(n) 变成 O(logn)。



页结构

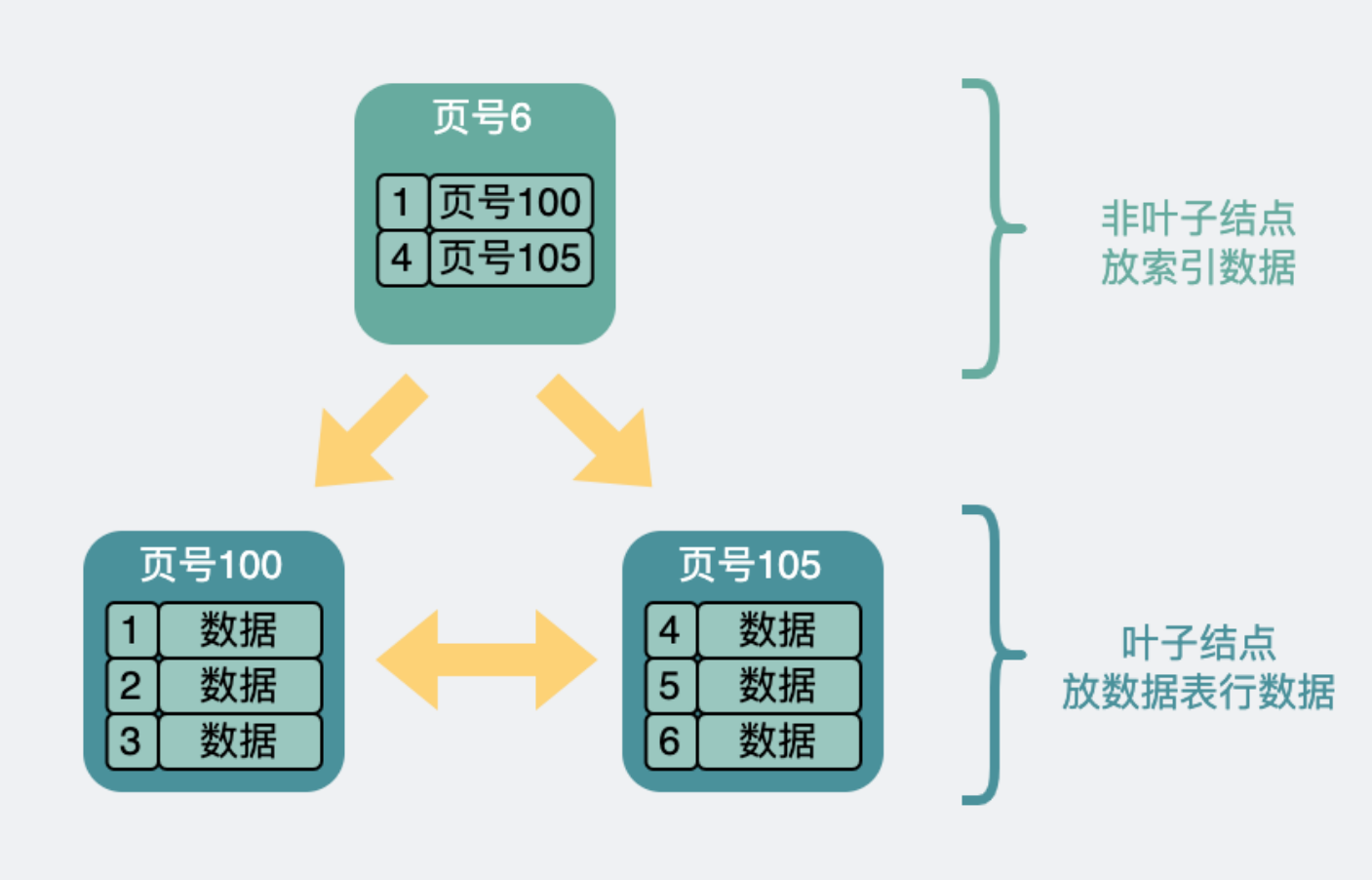
从页到索引

如果想查一条 record，可以把表空间里每一页查出来，再把里面的 record 挨个判断是不是我们要找的。

行数小的时候，这么操作也没啥问题。行数多了，性能就慢了。

于是为了加快搜索，可以在每个数据页里选出主键 id 最小的 record，而且只需要它们的主键 id 和所在页的页号。将它们组成新的 record，放入到一个新生成的一个数据页中。这个新数据页跟之前的页结构没啥区别，大小还是 16K。

但为了跟之前的数据页进行区分，数据页里加入了页层级（Page Level）信息，从 0 开始往上算。于是页与页之间就有了上下层级的概念，就像下面这样。

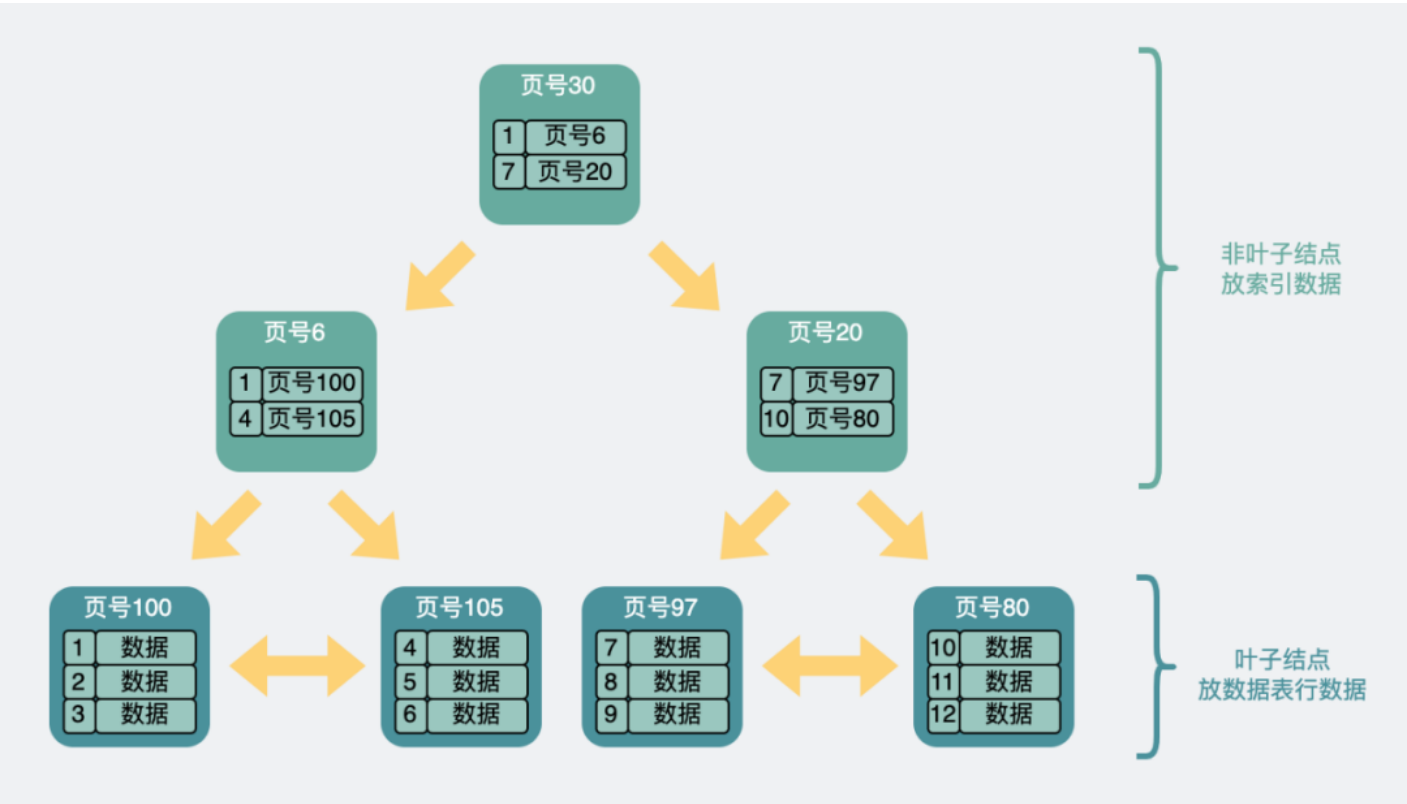


两层 B+ 树结构

页跟页之间看起来就像是一棵倒过来的树，也就是我们常说的 B+ 树索引。

最下面一层 Page Level 为 0，也就是所谓的叶子结点。其余都叫非叶子结点。

上面展示的是两层的树。如果数据变多了，还可以再通过类似的方法，往上构建一层，成了三层树。



三层 B+ 树结构

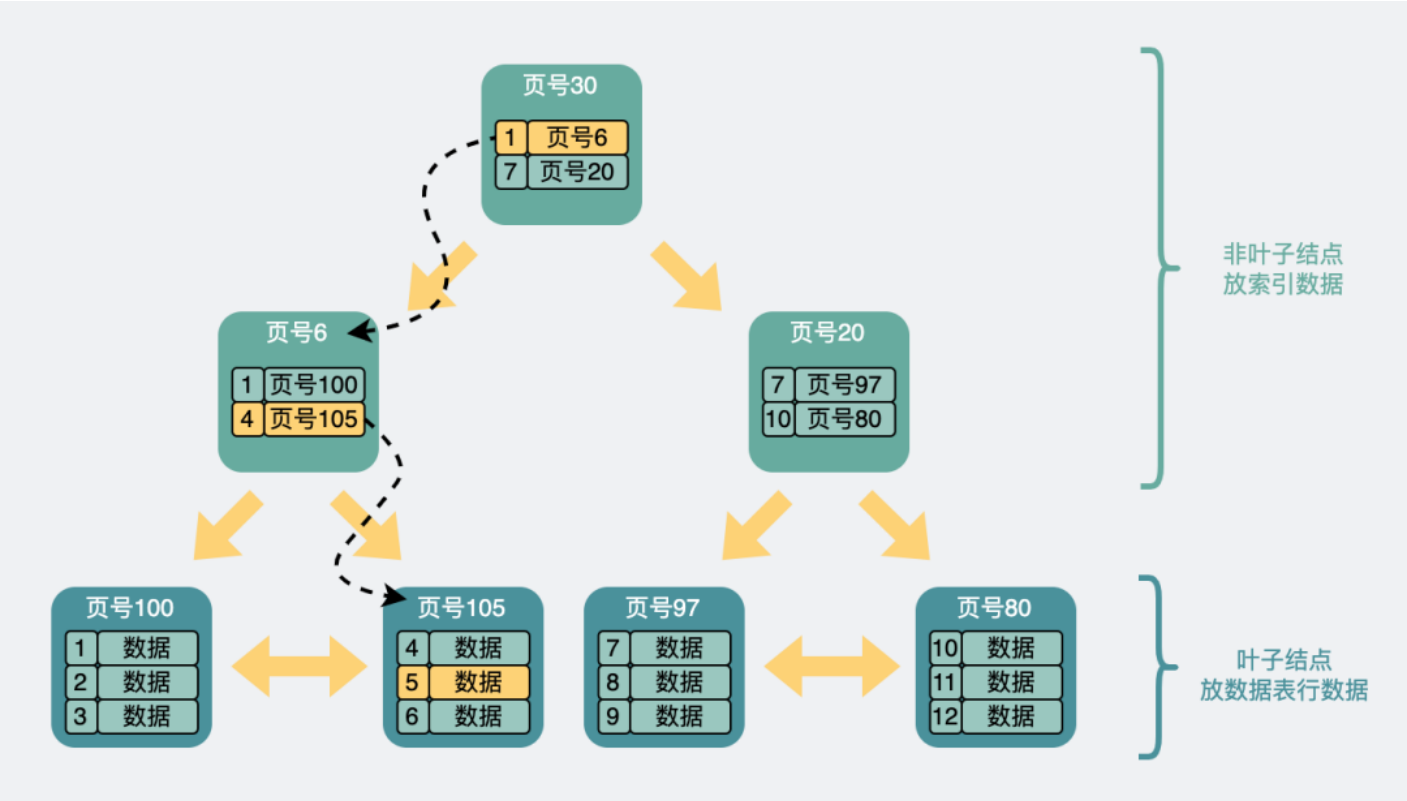
现在，可以通过这样一棵 B+ 树加速查询。

举个例子，比方说我们想要查找数据行 5。

先从顶层页的 record 入手。record 里包含了主键 id 和页号（页地址）。

下图中黄色的箭头：向左最小 id 是 1，向右最小 id 是 7。

1. 如果 id=5 的数据存在，那必定在左边箭头；
2. 于是顺着的 record 的页地址就到了 6 号数据页里；
3. 再判断 id=5>4，所以肯定在右边的数据页里；
4. 于是加载 105 号数据页；
5. 在数据页里找到 id=5 的数据行，完成查询。



B+ 树的查询过程

另外需要注意，上面的页的页号并不是连续的，它们在磁盘里也不一定是挨在一起的。

这个过程中查询了三个页，如果这三个页都在磁盘中（没有被提前加载到内存中），那么最多需要经历三次磁盘 IO 查询，它们才能被加载到内存中。

B+ 树承载的记录数量

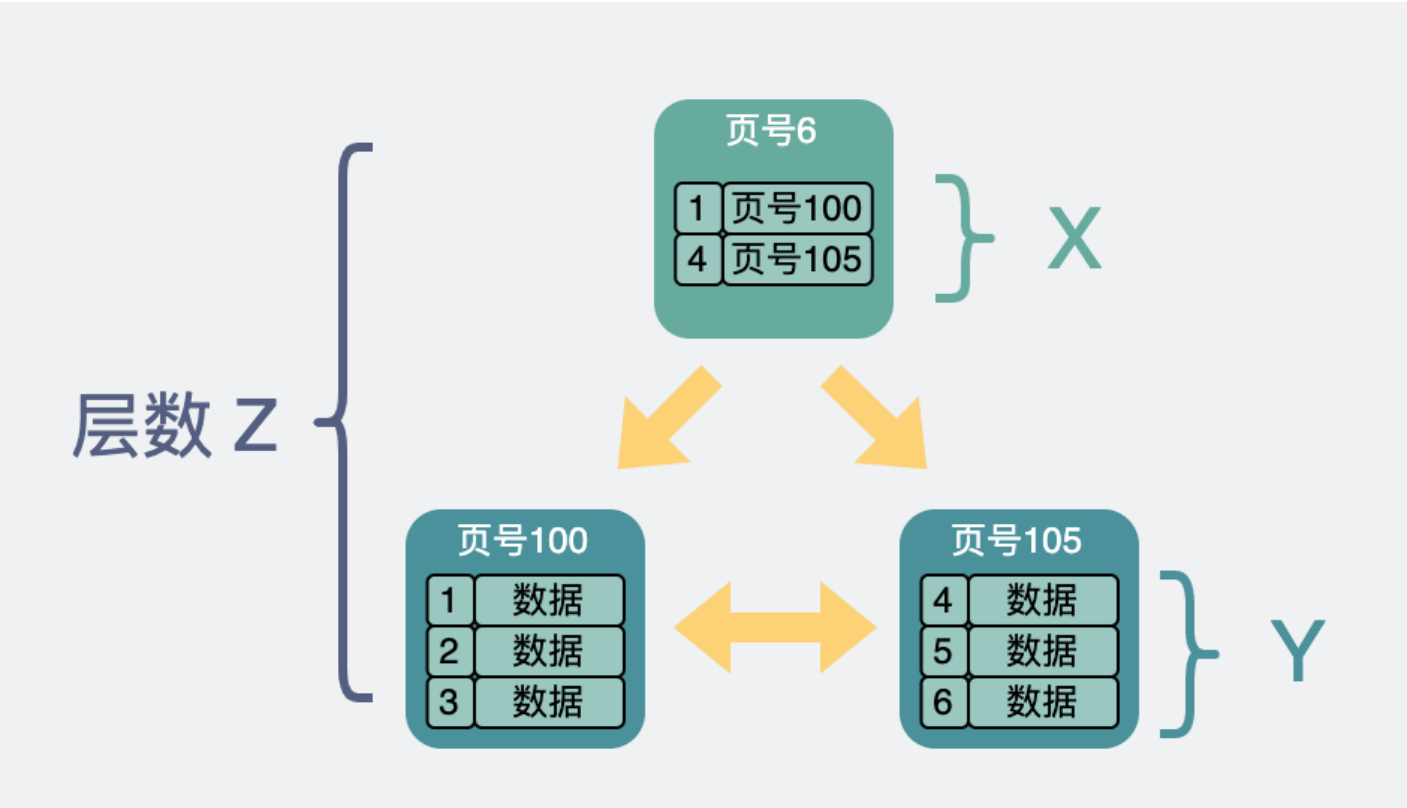
从上面的结构里可以看出，B+ 树的最末级叶子结点里放了 record 数据。而非叶子结点里则放了用来加速查询的索引数据。

也就是说，同样一个 16K 的页，非叶子节点里每一条数据都指向一个新的页。而新的页有两种可能。

- 如果是末级叶子节点的话，那么里面放的就是 record 数据；
- 如果是非叶子节点，那么就会循环继续指向新的数据页。

假设：

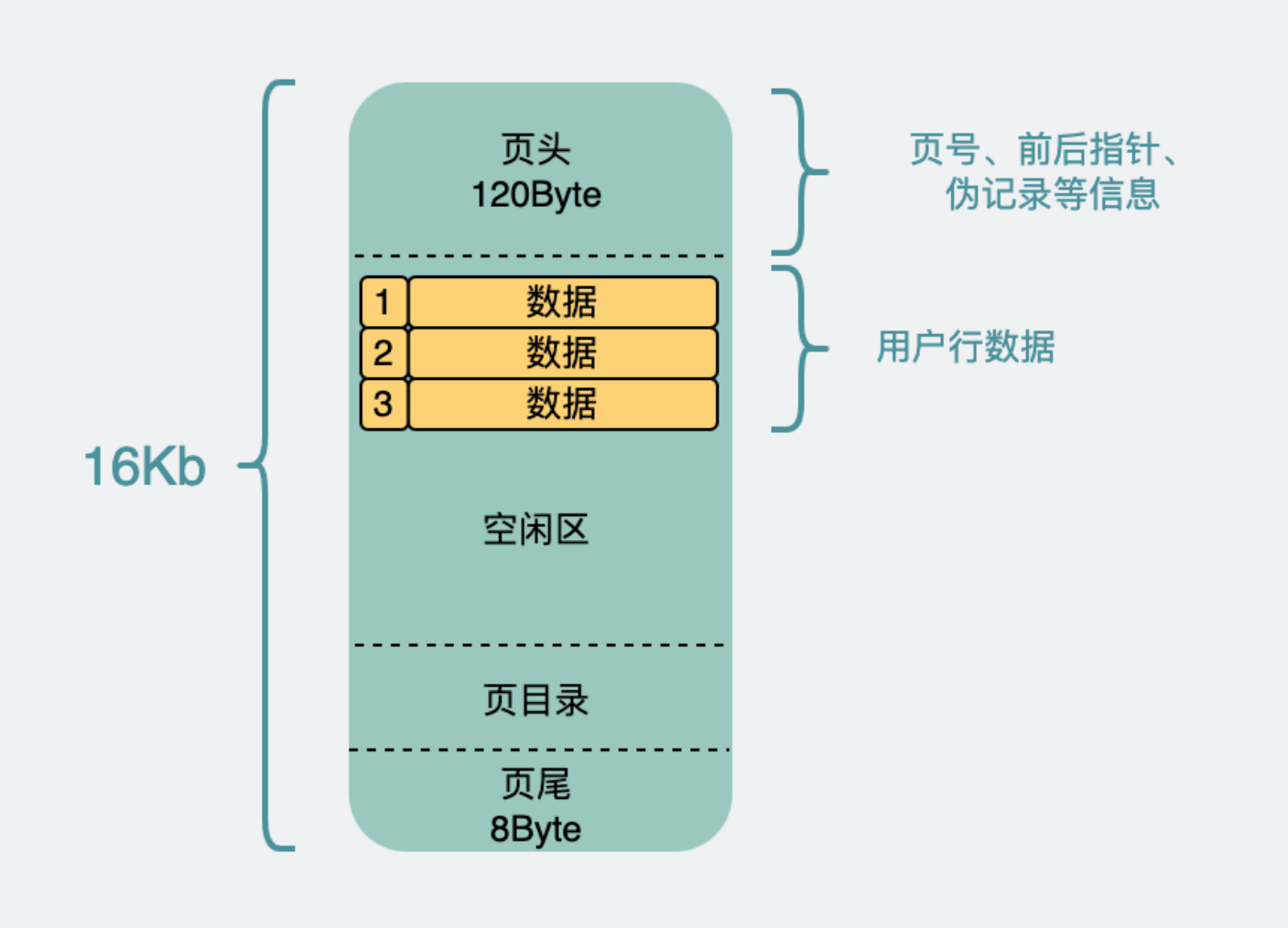
- 非叶子结点内指向其他内存页的指针数量为 X；
- 叶子节点内能容纳的 record 数量为 Y；
- B+ 树的层数为 Z。



总行数的计算方法
那这棵 B+ 树放的行数据总量等于 (X ^ (Z-1)) * Y。

怎么计算 X

我们回去看数据页的结构。



页结构

非叶子节点里主要放索引查询相关的数据，放的是主键和指向页号。

主键假设是 bigint（8Byte），而页号在源码里叫 FIL_PAGE_OFFSET（4 Byte），那么非叶子节点里的一条数据是 12 Byte 左右。

整个数据页 16K， 页头页尾那部分数据全加起来大概 128 Byte，加上页目录毛估占 1K 吧。那剩下的 15K 除以 12 Byte 等于 1280，也就是可以指向 X=1280 页。

我们常说的二叉树指的是一个结点可以发散出两个新的结点。m 叉树一个节点能指向 m 个新的节点。这个指向新节点的操作就叫扇出（Fanout）。

而上面的 B+ 树能指向 1280 个新的节点。恐怖如斯，可以说扇出非常高了。

如何计算 Y

叶子节点和非叶子节点的数据结构是一样的，所以也假设剩下 15KB 可以利用。

叶子节点里放的是真正的行数据。假设一条行数据 1KB，所以一页里能放 Y=15 行。

行总数计算

回到 (X ^ (Z-1)) * Y 这个公式，已知 X=1280，Y=15。

- 假设 B+ 树是两层，那 Z=2。总行数 (1280 ^ (2-1)) * 15 ≈ 2万

- 假设 B+ 树是三层，那 Z=3。总行数 (1280 ^ (3-1)) * 15 ≈ 2.5千万

这个 2.5千万，就是我们常说的单表建议最大行数两千万的由来。毕竟再加一层，数据就大得有点离谱了。三层数据页对应最多三次磁盘 IO，也比较合理。

行数超 1 亿就慢了吗?

上面假设单行数据用了 1KB，所以一个数据页能放个 15 行数据。

如果我单行数据用不了这么多，比如只用了 250 Byte。那么单个数据页能放 60 行数据。

那同样是三层 B+ 树，单表支持的行数就是 (1280 ^ (3-1)) * 60 ≈ 1亿。

你看我 1 亿数据，其实也就三层 B+ 树。在这个 B+ 树里要查到某行数据，最多也是三次磁盘 IO，所以并不慢。

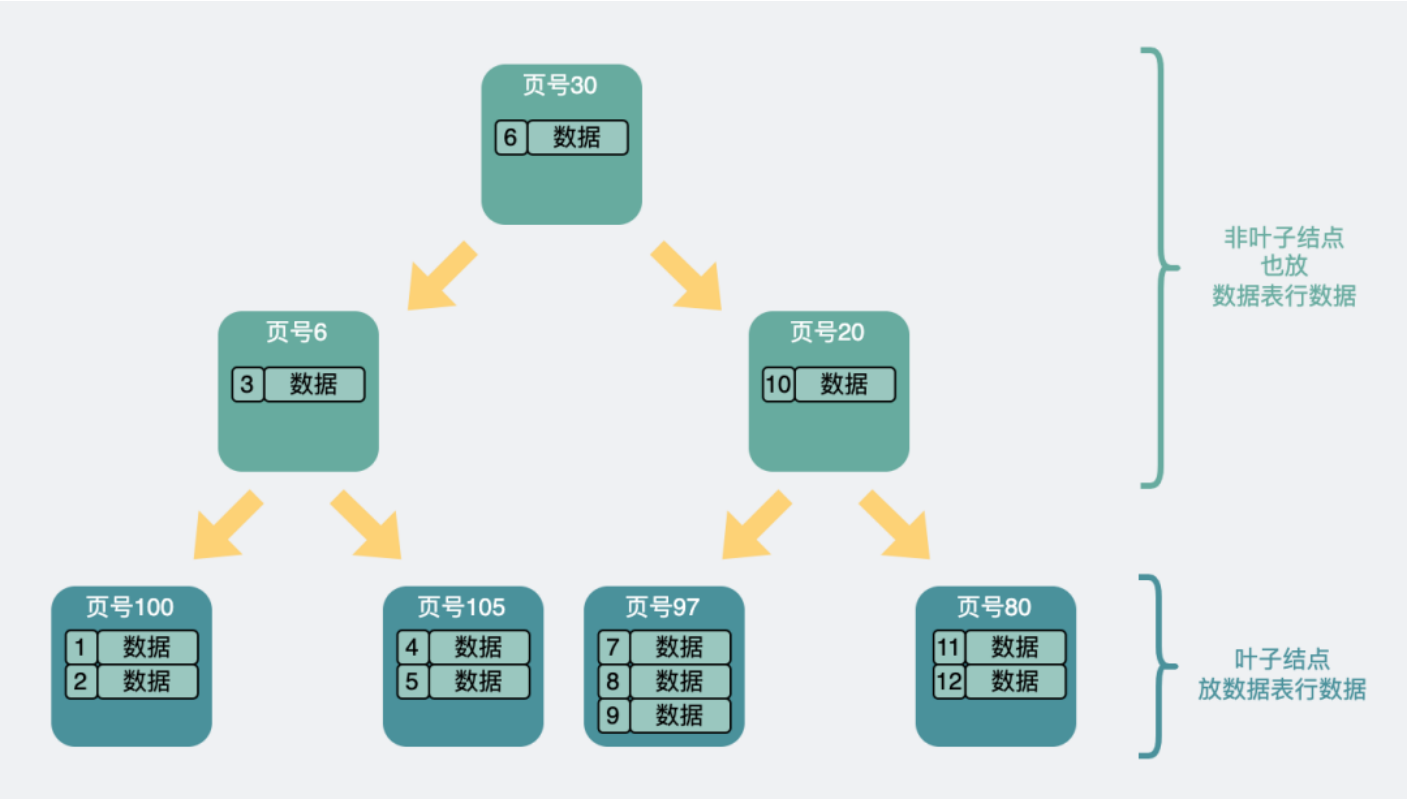
这就很好的解释了文章开头，为什么我单表 1 亿条数据，但查询性能没啥大毛病。

B 树承载的记录数量

既然都聊到这里了，我们就顺着这个话题多聊一些吧。

我们都知道，现在 MySQL 的索引都是 B+ 树。而有一种树，跟 B+ 树很像叫 B 树，也叫 B- 树。它跟 B+ 树最大的区别在于，B+ 树只在末级叶子结点处放数据表行数据，而 B 树则会在叶子和非叶子结点上都放。

B 树的结构类似这样：



B 树结构

B 树将行数据都存在非叶子节点上。假设每个数据页还是 16KB，掐头去尾每页剩 15KB，并且一条数据表行数据还是占 1KB。就算不考虑各种页指针的情况下，也只能放个 15 条数据，数据页的扇出明显变小了。

计算可承载的总行数的公式也变成了一个等比数列。

```
15 + 15^2 +15^3 + ... + 15^Z
```

其中 Z 还是层数的意思。

为了能放两千万左右的数据需要 Z>=6，也就是树需要有 6 层。查一次要访问 6 个页。假设这 6 个页并不连续，为了查询其中一条数据，最坏情况需要进行 6 次磁盘 IO。

而 B+ 树同样情况下放两千万数据左右，查一次最多是 3 次磁盘 IO。

磁盘 IO 越多则越慢，这两者在性能上差距略大。因此，B+ 树比 B 树更适合成为 MySQL 索引。

总结

- B+ 树叶子和非叶子结点的数据页都是 16KB，并且数据结构一致。区别在于叶子节点放的是真实的行数据，而非叶子节点放的是主键和下一个页的地址；
- B+ 树一般有两到三层。由于其高扇出，三层就能支持两千万以上的数据。并且一次查询最多 1~3 次磁盘 IO，性能也还行；
- 存储同样量级的数据，B 树比 B+ 树层级更高，因此磁盘 IO 也更多。所以，B+ 树更适合成为 MySQL 索引。
- 索引结构不会影响单表最大行数，两千万也只是推荐值。超过了这个值可能会导致 B+ 树层级更高，影响查询性能；
- 单表最大值还受主键大小和磁盘大小限制。

参考资料

《MYSQL内核：INNODB存储引擎 卷1》

- EOF -