千万数据量下，掌握Mysql索引底层原理将会拯救世界

- Mysql索引的本质
- Mysql索引的底层原理
- Mysql索引的实战经验

> 学海无涯，我们一起勉力前行
>
> 课程讲师：**周瑜老师QQ：3413298904**
>
> 往期课程资料：**安其拉老师QQ: 3164703201**
>
> VIP课程咨询：**木兰老师QQ: 2746251334**

面试

问：数据库中最常见的慢查询优化方式是什么?

答：加索引。

问：为什么加索引能优化慢查询?

答1：...不知道

答2：因为索引其实就是一种优化查询的数据结构，比如Mysql中的索引是用B+树实现的，而B+树就是一种数据结构，可以优化查询速度，可以利用索引快速查找数据，所以能优化查询。

问：你知道哪些数据结构可以提高查询速度？（听到这个问题就感觉此处有坑...）

答：哈希表、完全平衡二叉树、B树、B+树等等。

问：那这些数据结构既然都能优化查询速度，那Mysql种为何选择使用B+树?

答：...不知道

提问

```
SHOW INDEX FROM employees.titles;
```

Table	Non_uni...	Key_name	Seq_in_ind...	Column_name	Collation	Cardinality	Sub_p...	Pack...	Null	Index_ty...	Comm...	Index_com...
titles	0	PRIMARY	1	emp_no	A	296714	NULL	NULL		BTREE		
titles	0	PRIMARY	2	title	A	442308	NULL	NULL		BTREE		
titles	0	PRIMARY	3	from_date	A	442308	NULL	NULL		BTREE		

有一个titles表，主键由emp_no，title，from_date三个字段组成。

那么以下几个语句会用到索引吗：

1. `select * from employees.titles where emp_no = 1`
2. `select * from employees.titles where title = '1'`
3. `select * from employees.titles where title = '1' and emp_no = 1;`

索引(Index)

到底什么是索引(Index)?

大学老师是这么定义的：**索引就像书的目录**

Mysql官网是这么定义的：**Indexes are used to find rows with specific column values quickly**

我是这么定义的：**索引是一种优化查询的数据结构**

为什么哈希表、完全平衡二叉树、B树、B+树都可以优化查询，为何Mysql独独喜欢B+树?

哈希表是什么?

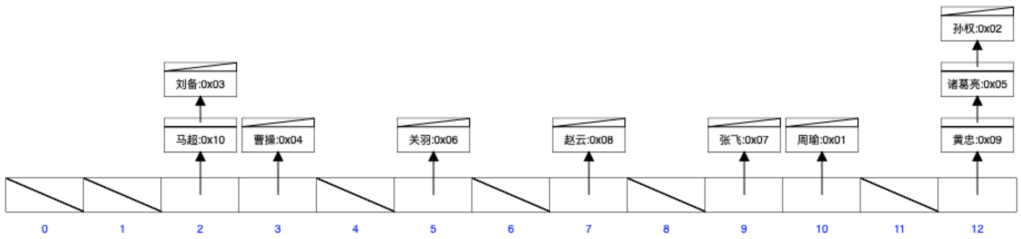
> 哈希表（Hash table，也叫散列表），是根据键值(Key value)而直接进行访问的数据结构。也就是说，它通过把键值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。
>
> 哈希表的做法其实很简单，就是把Key通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的数组空间里。而当使用哈希表进行查询的时候，就是再次使用哈希函数将key转换为对应的数组下标，并定位到该空间获取value，如此一来，就可以充分利用到数组的定位性能进行数据定位。

哈希表的特点是什么?

假如有这么一张表(表名：sanguo)：

	id	name	role
▶	1	周瑜	吴国大都督
	2	孙权	吴国国王
	3	刘备	蜀国国王
	4	曹操	魏国国王
	5	诸葛亮	蜀国军师
	6	关羽	五虎上将一
	7	张飞	五虎上将二
	8	赵云	五虎上将三
	9	黄忠	五虎上将四
	10	马超	五虎上将五
	NULL	NULL	NULL

现在对name字段建立哈希索引：



注意字段值所对应的数组下标是哈希算法随机算出来的，所以可能出现**哈希冲突**。

那么对于这样一个索引结构，现在来执行下面的sql语句：

```
select * from sanguo where name = '周瑜'
```

可以直接对‘周瑜’按哈希算法算出来一个数组下标，然后可以直接从数据中取出数据并拿到锁对应那一行数据的地址，进而查询那一行数据。

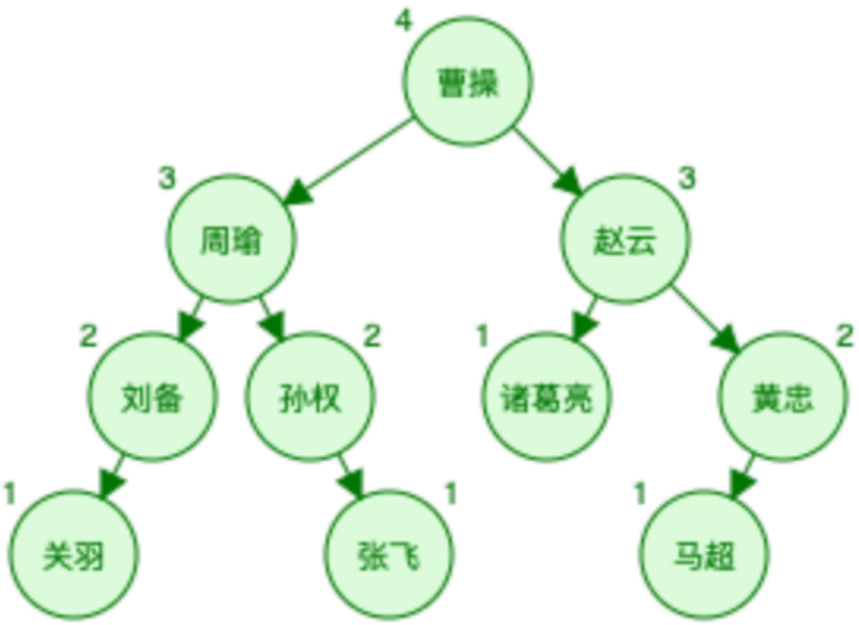
那么如果现在执行下面的sql语句：

```
select * from sanguo where name > '周瑜'
```

则无能为力，因为哈希表的特点就是**可以快速的精确查询，但是不支持范围查询**。

如果用完全平衡二叉树呢?

还是上面的表数据用完全平衡二叉树表示如下图（为了简单，数据对应的地址就不画在图中了。）：



图中的每一个节点实际上应该有四部分：

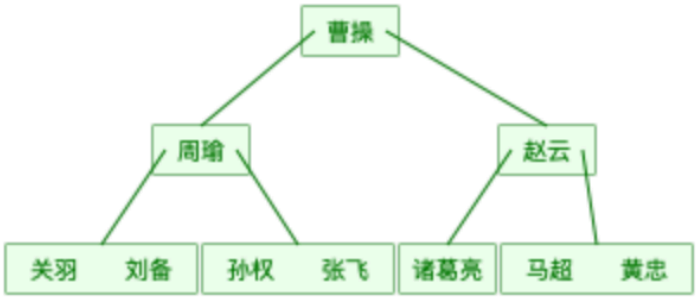
1. 左指针，指向左子树
2. 键值
3. 键值所对应的数据的存储地址
4. 右指针，指向右子树

另外需要提醒的是，二叉树是有顺序的，简单的说就是“左边的小于右边的”

假如我们现在来查找‘周瑜’，需要找2次（第一次曹操，第二次周瑜），比哈希表要多一次。而且由于完全平衡二叉树是有序的，所以也是支持范围查找的。

如果用B树呢?

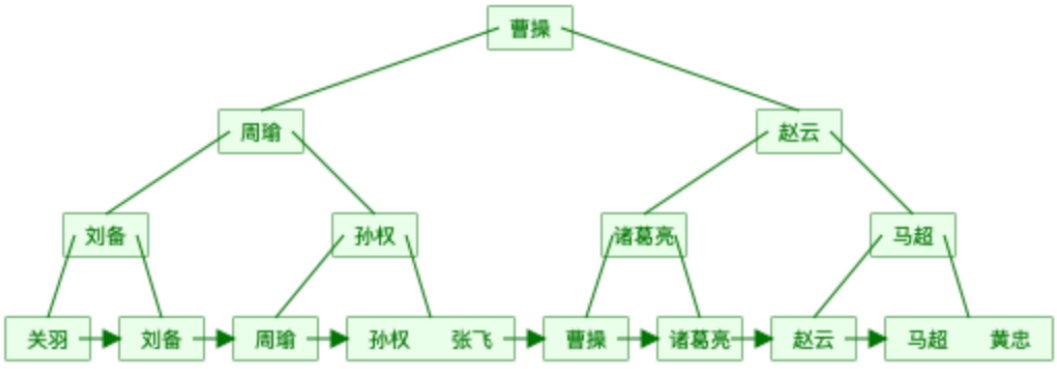
还是上面的表数据用B树表示如下图（为了简单，数据对应的地址就不画在图中了。）：



我们可以发现同样的元素，B树的表示要比完全平衡二叉树要“矮”，原因在于B树中的一个节点可以存储多个元素。

如果用B+树呢?

还是上面的表数据用B+树表示如下图（为了简单，数据对应的地址就不画在图中了。）：



我们可以发现同样的元素，B+树的表示要比B树要“胖”，原因在于B+树中的非叶子节点会冗余一份在叶子节点中，并且叶子节点之间用指针相连。

那么B+树到底有什么优势呢?

这里我们用“反证法”，假如我们现在就用完全平衡二叉树作为索引的数据结构，我们来看一下有什么不妥的地方。

实际上，索引也是很“大”的，因为索引也是存储元素的，我们的一个表的数据行数越多，那么对应的索引文件其实也是会很大的，实际上**也是需要存储在磁盘中的，而不能全部都放在内存中**，所以我们在考虑选用哪种数据结构时，我们可以换一个角度思考，**哪个数据结构更适合从磁盘中读取数据**，或者**哪个数据结构能够提高磁盘的IO效率**。

回头看一下完全平衡二叉树，当我们需要查询“张飞”时，需要以下步骤

1. 从磁盘中取出“曹操”到内存，CPU从内存取出数据进行笔记，“张飞”<“曹操”，取左子树（产生了一次磁盘IO）
2. 从磁盘中取出“周瑜”到内存，CPU从内存取出数据进行笔记，“张飞”>“周瑜”，取右子树（产生了一次磁盘IO）
3. 从磁盘中取出“孙权”到内存，CPU从内存取出数据进行笔记，“张飞”>“孙权”，取右子树（产生了一次磁盘IO）
4. 从磁盘中取出“黄忠”到内存，CPU从内存取出数据进行笔记，“张飞”=“张飞”，找到结果（产生了一次磁盘IO）

同理，回头看一下B树，我们发现只发送三次磁盘IO就可以找到“张飞”了，这就是B树的优点：**一个节点可以存储多个元素，相对于完全平衡二叉树所以整棵树的高度就降低了，磁盘IO效率提高了**。

而，B+树是B树的升级版，只是把非叶子节点冗余一下，这么做的好处是**为了提高范围查找的效率**。

> 所以，到这里，我们可以总结出来，Mysql选用B+树这种数据结构作为索引，可以提高查询索引时的磁盘IO效率，并且可以提高范围查询的效率，并且B+树里的元素也是有序的。

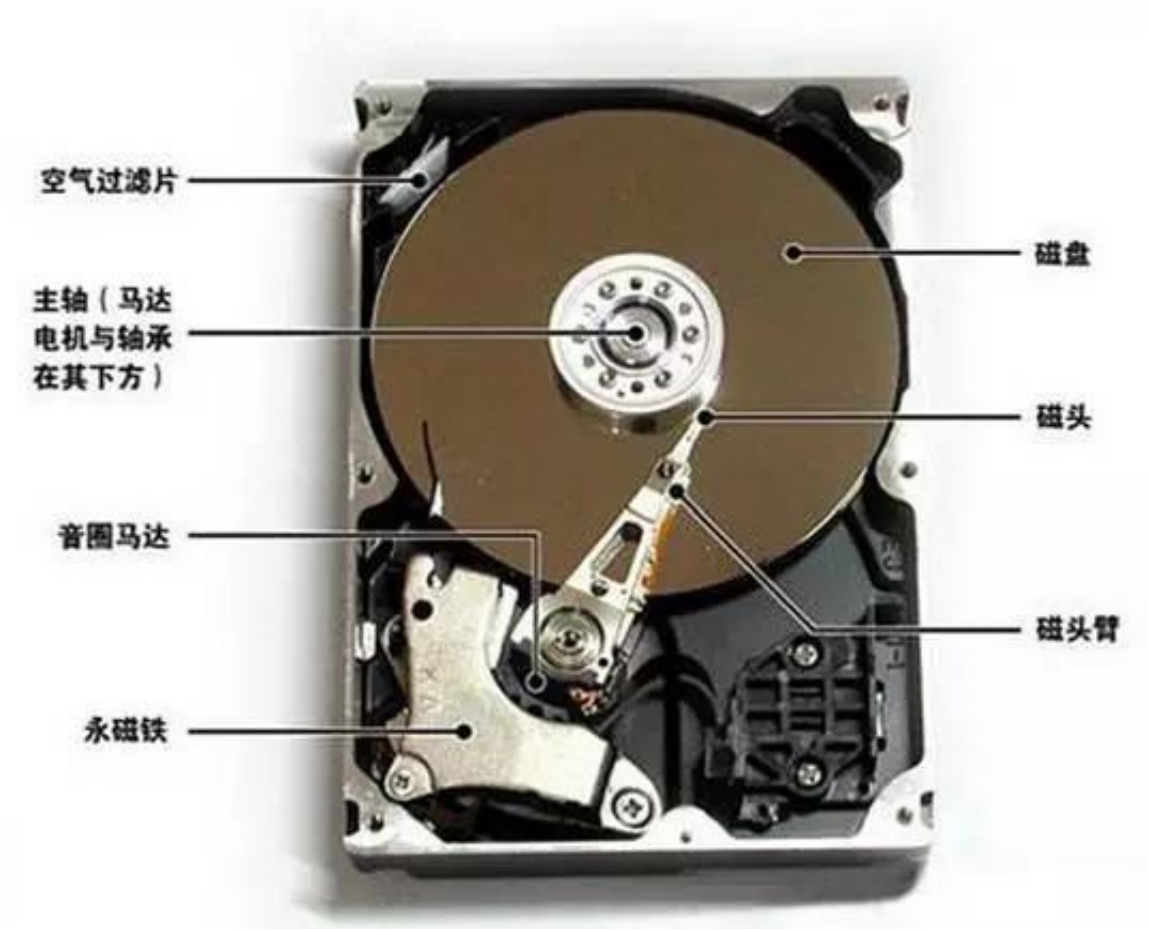
那么，一个B+树的节点中到底存多少个元素合适呢?

这里有必要先来了解一下磁盘IO的原理。

磁盘I/O的本质

磁盘分类

机械硬盘



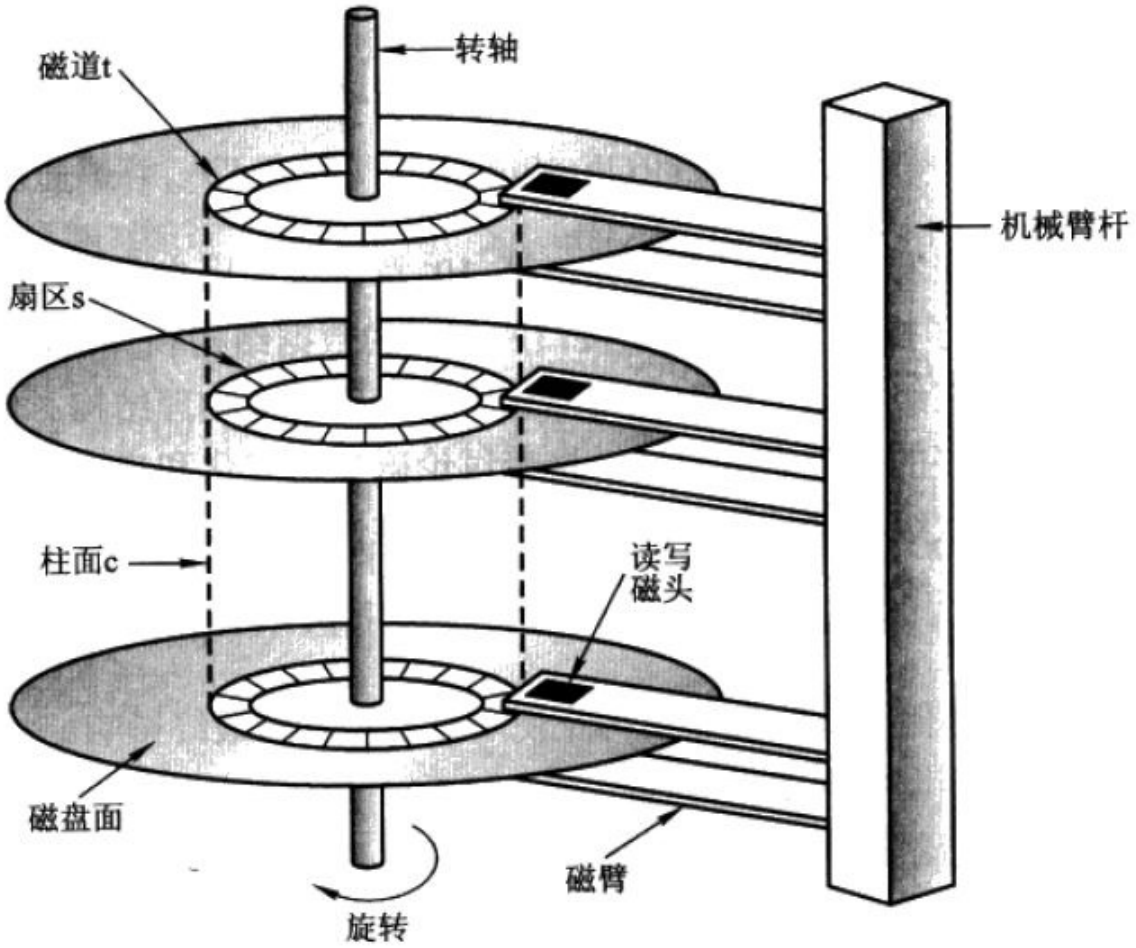
固态硬盘

从上面的原理我们也能知道，固态硬盘比机械硬盘快的最根本最简单的原因就是：**固态硬盘使用的电路进行读写，而机械硬盘使用的机械运动。**

其实不管是机械硬盘还是固态硬盘都是存储介质，真正控制读写的是操作系统。

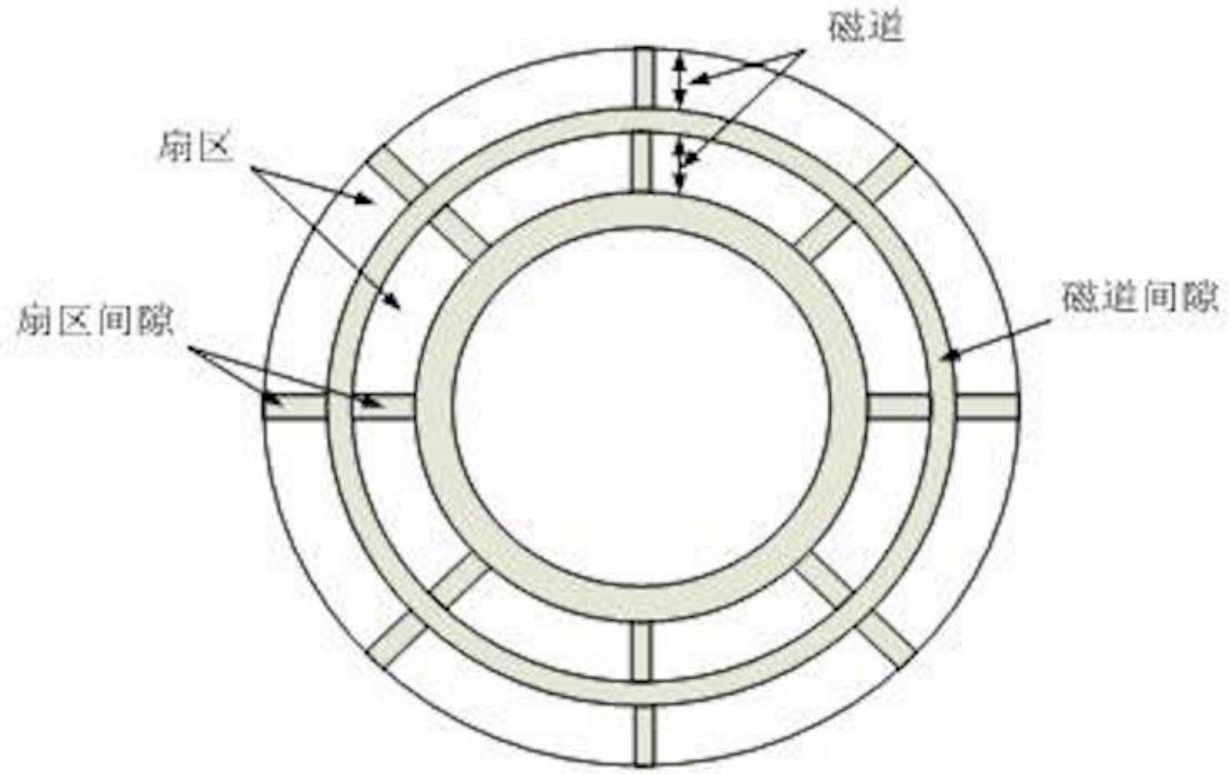
机械硬盘存储原理

磁盘立体结构图



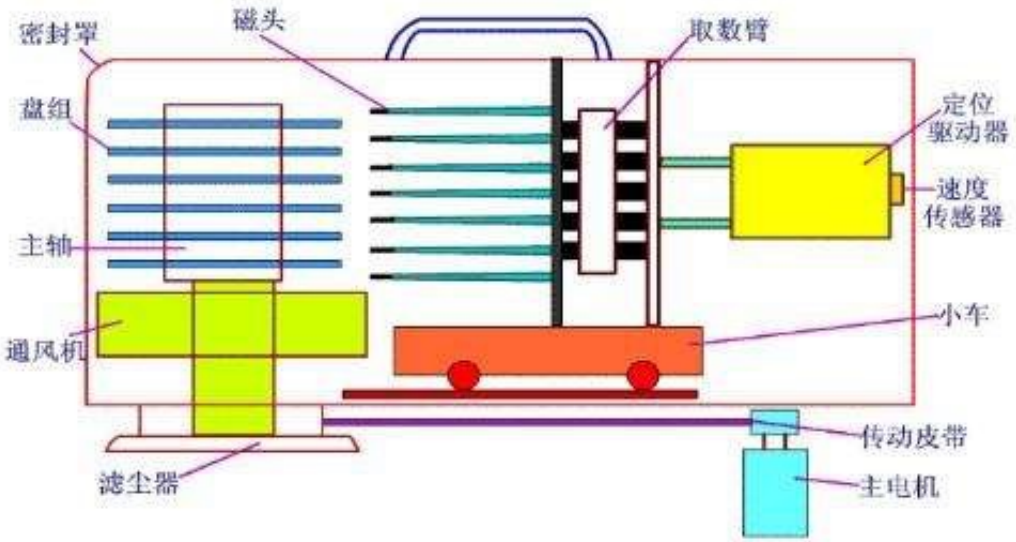
一个磁盘由大小相同且同轴的圆形盘片组成，磁盘可以转动（各个磁盘必须同步转动）。在磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不能转动，但是可以沿磁盘半径方向运动（实际是斜切向运动），每个磁头同一时刻也必须是同轴的，即从正上方向下看，所有磁头任何时候都是重叠的（不过目前已经有多磁头独立技术，可不受此限制）。

磁盘片结构图



盘片被划分成一系列同心环，圆心是盘片中心，每个同心环叫做一个磁道，所有半径相同的磁道组成一个柱面。磁道被沿半径线划分成一个个小的段，每个段叫做一个扇区，每个扇区是磁盘的最小存储单元，大小一般为521字节。

磁盘读取数据逻辑



当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间。

固态硬盘存储原理

固态硬盘（Solid State Drives），用固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元（FLASH芯片、DRAM芯片）组成。固态硬盘在接口的规范和定义、功能及使用方法上与普通硬盘的完全相同，在产品外形和尺寸上也完全与普通硬盘一致。

控制单元

每个SSD都有一个控制器(controller)将存储单元连接到电脑，主控器可以通过若干个通道（channel）并行操作多块存储单元

存储单元

一个Flash Page由两个或者多个Die(又称chips组成)，这些Dies可以共享I/0数据总线和一些控制信号线。一个Die又可以分为多个Plane,而每个Plane又包含多个多个Block,每个Block又分为多个Page。以Samsung 4GB Flash为例，一个4GB的Flash Page由两个2GB的Die组成，共享8位I/0数据总线和一些控制信号线。每个Die由4个Plane组成，每个Plane包含2048个Block，每个Block又包含64个4KB大小的Page

访问SSD的原理

Host是通过LBA（Logical BlockAddress，逻辑地址块）访问SSD的，每个LBA代表着一个Sector（一般为512B大小），操作系统一般以4K为单位访问SSD，我们把Host访问SSD的基本单元叫用户页（Host Page）。而在SSD内部，SSD主控与Flash之间是Flash Page为基本单元访问Flash的，我们称Flash Page为物理页（Physical Page）。Host每写入一个Host Page, SSD主控会找一个Physical Page把Host数据写入，SSD内部同时记录了这样一条映射（Map）。有了这样一个映射关系后，下次Host需要读某个Host Page 时，SSD就知道从Flash的哪个位置把数据读取上来。

局部性原理与磁盘预读

计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。

所以操作系统为了提高效率，读取数据时往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，操作系统也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这里的一定长度叫做**页**，也就是操作系统操作磁盘时的基本单位。一般操作系统中一页的大小是4Kb。

总结

所以，回到我们的问题，**B+树中一个节点到底存多少个元素合适？**，其实也可以换个角度来思考**B+树中一个节点到底多大合适？**

答案是：**B+树中一个节点为一页或页的倍数最为合适**。因为如果一个节点的大小小于1页，那么读取这个节点的时候其实也会读出1页，造成资源的浪费；如果一个节点的大小大于1页，比如1.2页，那么读取这个节点的时候会读出2页，也会造成资源的浪费；所以为了不造成浪费，所以最后把一个节点的大小控制在1页、2页、3页、4页等倍数页大小最为合适。

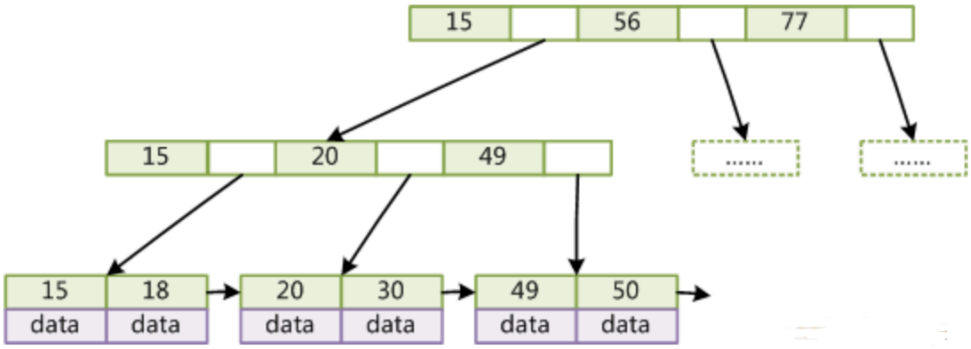
那么，Mysql中B+树的一个节点大小为多大呢?

这个问题的答案是“1页”，这里说的“页”是Mysql自定义的单位（其实和操作系统类似），Mysql的Innodb引擎中一页的默认大小是16k（如果操作系统中一页大小是4k，那么Mysql中1页=操作系统中4页），可以使用命令**SHOW GLOBAL STATUS like 'Innodb_page_size';**查看。并且还可以告诉你的是，一个节点为1页就够了。

为什么一个节点为1页（16k）就够了?

解决这个问题，我们先来看一下Mysql中利用B+树的具体实现。

Mysql中MyISAM和innodb使用B+树



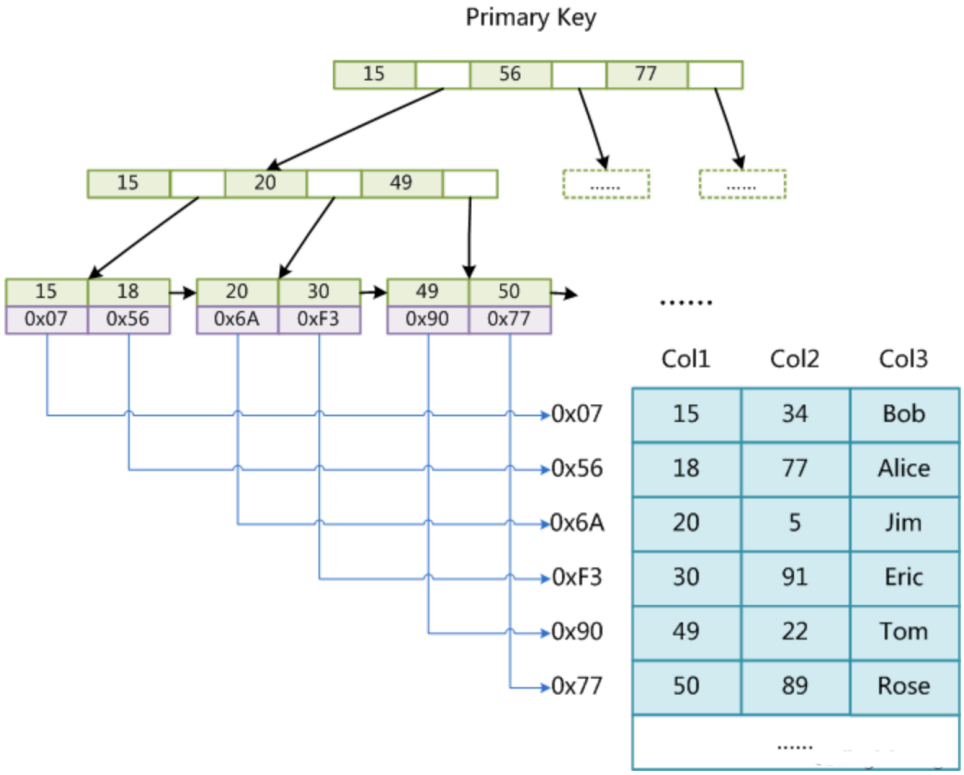
通常我们认为B+树的非叶子节点不存储数据，只有叶子节点才存储数据；而B树的非叶子和叶子节点都会存储数据，会导致非叶子节点存储的索引值会更少，树的高度相对会比B+树高，平均的I/O效率会比较低，所以使用B+树作为索引的数据结构，再加上B+树的叶子节点之间会有指针相连，也方便进行范围查找。

上图的data区域两个存储引擎会有不同。

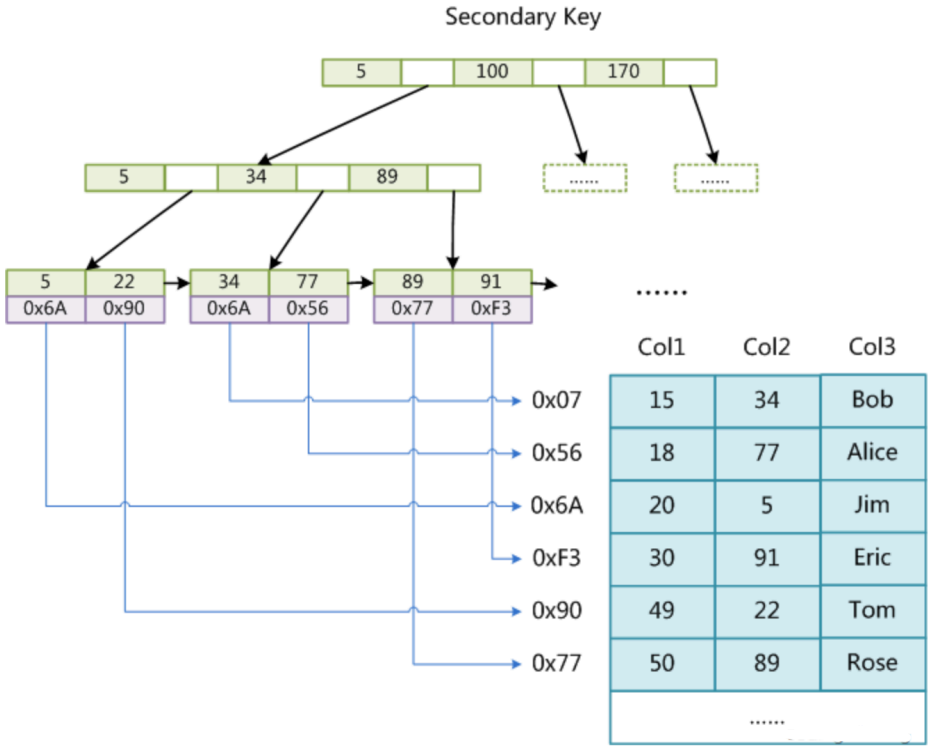
MyISAM中的B+树

MYISAM中叶子节点的数据区域存储的是数据记录的地址

主键索引


Primary Key
15
56
77
15
20
49
.....
.....
15
18
20
30
49
50
0x07
0x56
0x6A
0xF3
0x90
0x77
.....
Col1
Col2
Col3
15
34
Bob
18
77
Alice
20
5
Jim
30
91
Eric
49
22
Tom
50
89
Rose
.....
0x07
0x56
0x6A
0xF3
0x90
0x77

辅助索引


Secondary Key
5
100
170
5
34
89
.....
.....
5
22
34
77
89
91
0x6A
0x90
0x6A
0x56
0x77
0xF3
.....
Col1
Col2
Col3
15
34
Bob
18
77
Alice
20
5
Jim
30
91
Eric
49
22
Tom
50
89
Rose
.....
0x07
0x56
0x6A
0xF3
0x90
0x77

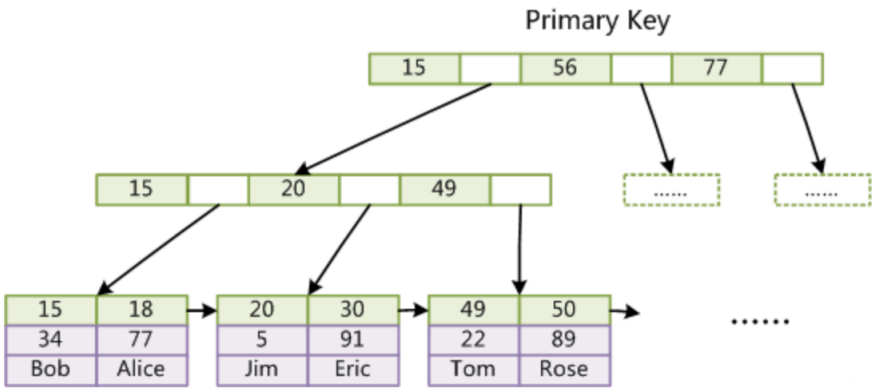
总结

MyISAM存储引擎在使用索引查询数据时，会先根据索引查找到数据地址，再根据地址查询到具体的数据。并且主键索引和辅助索引没有太多区别。

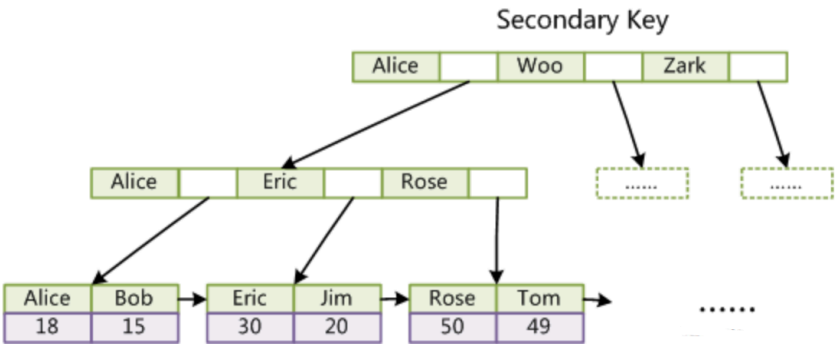
InnoDB中的B+树

InnoDB中主键索引的叶子节点的数据区域存储的是数据记录，辅助索引存储的是主键值

主键索引



辅助索引



总结

Innodb中的主键索引和实际数据时绑定在一起的，也就是说Innodb的一个表一定要有主键索引，如果一个表没有手动建立主键索引，Innodb会查看有没有唯一索引，如果有则选用唯一索引作为主键索引，如果连唯一索引也没有，则会默认建立一个隐藏的主键索引（用户不可见）。

另外，Innodb的主键索引要比MyISAM的主键索引查询效率要高（少一次磁盘IO），并且比辅助索引也要高很多。

所以，我们在使用Innodb作为存储引擎时，我们最好：

1. 手动建立主键索引

2. 尽量利用主键索引查询

回到我们的问题：为什么一个节点为1页（16k）就够了?

对着上面Mysql中Innodb中对B+树的实际应用（主要看主键索引），我们可以发现，B+树中的一个节点存储的内容是：

- 非叶子节点：主键+指针
- 叶子节点：数据

那么，假设我们一行数据大小为1K，那么一页就能存16条数据，也就是一个叶子节点能存16条数据；

再看非叶子节点，假设主键ID为bigint类型，那么长度为8B，指针大小在Innodb源码中为6B，一共就是14B，那么一页里就可以存储16K/14=1170个(主键+指针)，那么一颗高度为2的B+树能存储的数据为：1170*16=18720条，一颗高度为3的B+树能存储的数据为：1170*1170*16=21902400（千万级条）。所以在InnoDB中B+树高度一般为1-3层，它就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次IO操作即可查找到数据。所以也就回答了我们的问题，1页=16k这么设置是比较合适的，是适用大多数的企业的，当然这个值是可以修改的，所以也能根据业务的时间情况进行调整。

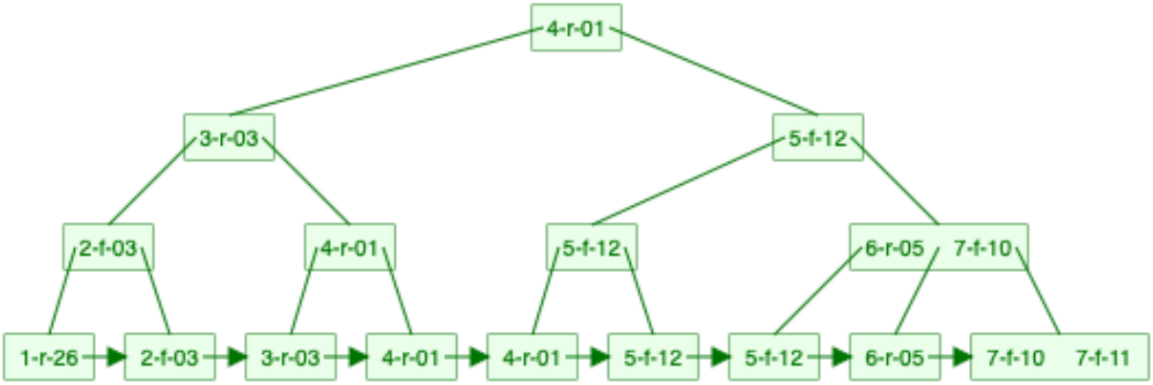
最左前缀原则

我们模拟数据建立一个联合索引

```
select *, concat(right(emp_no,1), "-", right(title,1), "-", right(from_date,2))
from employees.titles limit 10;
```

emp_no	title	from_date	to_date	concat(right(emp_no,1), "-", right(title,1), "-",...
10001	Senior Engineer	1986-06-26	9999-01-01	1-r-26
10002	Staff	1996-08-03	9999-01-01	2-f-03
10003	Senior Engineer	1995-12-03	9999-01-01	3-r-03
10004	Engineer	1986-12-01	1995-12-01	4-r-01
10004	Senior Engineer	1995-12-01	9999-01-01	4-r-01
10005	Senior Staff	1996-09-12	9999-01-01	5-f-12
10005	Staff	1989-09-12	1996-09-12	5-f-12
10006	Senior Engineer	1990-08-05	9999-01-01	6-r-05
10007	Senior Staff	1996-02-11	9999-01-01	7-f-11
10007	Staff	1989-02-10	1996-02-11	7-f-10

那么对应的B+树为



> 我们判断一个查询条件能不能用到索引，我们要分析这个查询条件能不能利用某个索引缩小查询范围

对于 `select * from employees.titles where emp_no = 1` 是能用到索引的，因为它能利用上面的索引所有查询范围，首先和第一个节点“4-r-01”比较，1<4，所以可以直接确定结果在左子树，同理，依次按顺序进行比较，逐步可以缩小查询范围。

对于 `select * from employees.titles where title = '1'` 是不能用到索引的，因为它不能用到上面的所以，和第一节点进行比较时，没有emp_no这个字段的值，不能确定到底该去左子树还是右子树继续进行查询。

对于 `select * from employees.titles where title = '1' and emp_no = 1` 是能用到索引，按照我们的上面的分析，先用title='1'这个条件和第一个节点进行比较，是没有结果的，但是mysql会对这个sql进行优化，优化之后会将emp_no=1这个条件放到第一位，从而可以利用索引。

Mysql总结

1. B+树可以更好的结合磁盘IO原理提高查询效率
2. Innodb一定要有主键，没有主键会以唯一索引为主键，否则会建立一个隐藏主键
3. Innodb的数据是和主键索引存在一起的（数据在叶子节点中，MyISAM中的叶子节点存储的数据地址）
4. 建立索引时要考虑已有索引，一个SQL语句只会选择花费最低的一个索引执行
5. 索引是一种有序的数据结构（B+树），一个节点可以存多个有序的元素，所以要利用好最左前缀原则
6. 真实场景中一颗B+树的高度通常为3