一个问题？

原文链接：https://www.cnblogs.com/leefreeman/p/8315844.html

InnoDB一棵B+树可以存放多少行数据？这个问题的简单回答是：约2千万。为什么是这么多呢？因为这是可以算出来的，要搞清楚这个问题，我们先从InnoDB索引数据结构、数据组织方式说起。

我们都知道计算机在存储数据的时候，有最小存储单元，这就好比我们今天进行现金的流通最小单位是一毛。在计算机中磁盘存储数据最小单元是扇区，一个扇区的大小是512字节，而文件系统（例如XFS/EXT4）他的最小单元是块，一个块的大小是4k，而对于我们的InnoDB存储引擎也有自己的最小储存单元——页（Page），一个页的大小是16K。

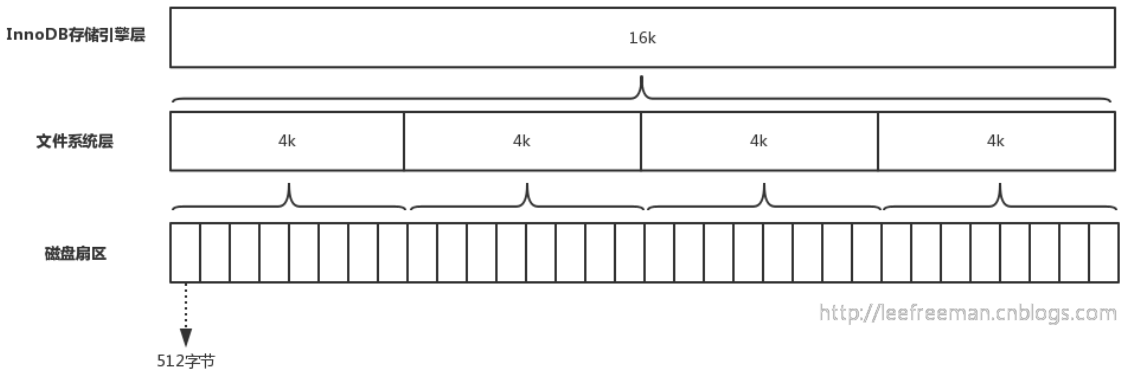
下面几张图可以帮你理解最小存储单元：

文件系统中一个文件大小只有1个字节，但不得不占磁盘上4KB的空间。

innodb的所有数据文件（后缀为ibd的文件），他的大小始终都是16384（16k）的整数倍。

```
-rw-rw--w-  1 _mysql  _mysql       8850  3  9  2017 customer.frm
-rw-rw--w-  1 _mysql  _mysql   41943040  3  9  2017 customer.ibd
-rw-rw--w-  1 _mysql  _mysql         62  3  9  2017 db.opt
-rw-rw--w-  1 _mysql  _mysql       9226  3  9  2017 lineitem.frm
-rw-rw--w-  1 _mysql  _mysql 2462056448  3  9  2017 lineitem.ibd
-rw-rw--w-  1 _mysql  _mysql       8692  3  9  2017 nation.frm
-rw-rw--w-  1 _mysql  _mysql     114688  3  9  2017 nation.ibd
-rw-rw--w-  1 _mysql  _mysql       8928  3  9  2017 orders.frm
-rw-rw--w-  1 _mysql  _mysql  293601280  3  9  2017 orders.ibd
-rw-rw--w-  1 _mysql  _mysql       8874  3  9  2017 part.frm
-rw-rw--w-  1 _mysql  _mysql   41943040  3  9  2017 part.ibd
-rw-rw--w-  1 _mysql  _mysql       8748  3  9  2017 partsupp.frm
-rw-rw--w-  1 _mysql  _mysql  184549376  3  9  2017 partsupp.ibd
-rw-rw--w-  1 _mysql  _mysql       8648  3  9  2017 region.frm
-rw-rw--w-  1 _mysql  _mysql      98304  3  9  2017 region.ibd
-rw-rw--w-  1 _mysql  _mysql       8804  3  9  2017 supplier.frm
-rw-rw--w-  1 _mysql  _mysql   10485760  3  9  2017 supplier.ibd
-rw-rw--w-  1 _mysql  _mysql       8604  3  9  2017 time_statistics.frm
-rw-rw--w-  1 _mysql  _mysql      98304  3  9  2017 time_statistics.ibd
```

磁盘扇区、文件系统、InnoDB存储引擎都有各自的最小存储单元。

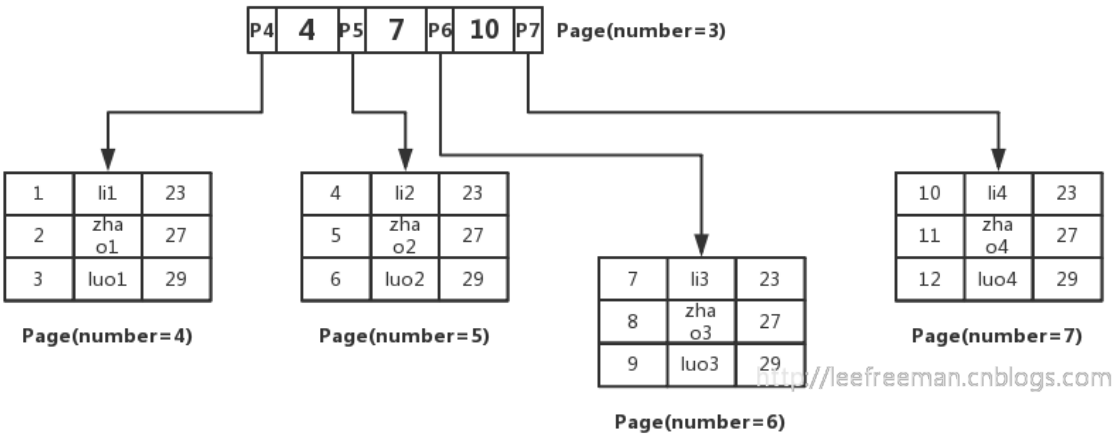


在MySQL中我们的InnoDB页的大小默认是16k，当然也可以通过参数设置：

```
mysql> show variables like 'innodb_page_size';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| innodb_page_size | 16384 |
+------------------+-------+
1 row in set (0.00 sec)
```

数据表中的数据都是存储在页中的，所以一个页中能存储多少行数据呢？假设一行数据的大小是1k，那么一个页可以存放16行这样的数据。

如果数据库只按这样的方式存储，那么如何查找数据就成为一个问题，因为我们不知道要查找的数据存在哪个页中，也不可能把所有的页遍历一遍，那样太慢了。所以人们想了一个办法，用B+树的方式组织这些数据。如图所示：



我们先将数据记录按主键进行排序，分别存放在不同的页中（为了便于理解我们这里一个页中只存放3条记录，实际情况可以存放很多），除了存放数据的页以外，还有存放键值+指针的页，如图中page number=3的页，该页存放键值和指向数据页的指针，这样的页由N个键值+指针组成。当然它也是排好序的。这样的数据组织形式，我们称为索引组织表。现在来看下，要查找一条数据，怎么查？

如select * from user where id=5;

这里id是主键,我们通过这棵B+树来查找，首先找到根页，你怎么知道user表的根页在哪呢？其实每张表的根页位置在表空间文件中是固定的，即page number=3的页（这点我们下文还会进一步证明），找到根页后通过二分查找法，定位到id=5的数据应该在指针P5指向的页中，那么进一步去page number=5的页中查找，同样通过二分查询法即可找到id=5的记录：

5	zhao2	27

现在我们清楚了InnoDB中主键索引B+树是如何组织数据、查询数据的，我们总结一下：

1、InnoDB存储引擎的最小存储单元是页，页可以用于存放数据也可以用于存放键值+指针，在B+树中叶子节点存放数据，非叶子节点存放键值+指针。

2、索引组织表通过非叶子节点的二分查找法以及指针确定数据在哪个页中，进而在去数据页中查找到需要的数据；

那么回到我们开始的问题，通常一棵B+树可以存放多少行数据？

这里我们先假设B+树高为2，即存在一个根节点和若干个叶子节点，那么这棵B+树的存放总记录数为：**根节点指针数*单个叶子节点记录行数。**

上文我们已经说明**单个叶子节点（页）中的记录数=16K/1K=16。**（这里假设一行记录的数据大小为1k，实际上现在很多互联网业务数据记录大小通常就是1K左右）。

那么现在我们需要计算出非叶子节点能存放多少指针，其实这也很好算，我们假设主键ID为bigint类型，长度为8字节，而指针大小在InnoDB源码中设置为6字节，这样一共14字节，我

们一个页中能存放多少这样的单元，其实就代表有多少指针，即**16384/14=1170。那么可以算出一棵高度为2的B+树，能存放1170*16=18720条这样的数据记录。**
根据同样的原理我们可以算出一个高度为3的B+树可以存放：1170*1170*16=21902400条这样的记录。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。

怎么得到InnoDB主键索引B+树的高度？

上面我们通过推断得出B+树的高度通常是1-3，下面我们从另外一个侧面证明这个结论。在InnoDB的表空间文件中，约定**page number为3**的代表主键索引的根页，而在根页偏移量为**64**的地方存放了该B+树的page level。如果page level为1，树高为2，page level为2，则树高为3。即B+树的高度=page level+1；下面我们将从实际环境中尝试找到这个page level。
在实际操作之前，你可以通过InnoDB元数据表确认主键索引根页的page number为3，你也可以从《InnoDB存储引擎》这本书中得到确认。

```
SELECT
b.name, a
.name,
index_id,
type,
a.space,
a.PAGE_N
O
FROM
informati
on_schem

a.INNOD
B_SYS_IN
DEXES a,
informati
on_schem
a.INNOD
B_SYS_TA
BLES b
WHERE
a.table_id
=
b.table_id
AND
a.space
<> 0;
```

执行结果：

```
+-----------------------------------+--------------------------+----------+------+-------+----------+
| name                              | name                     | index_id | type | space | PAGE_NO |
+-----------------------------------+--------------------------+----------+------+-------+----------+
| dbt3/customer                     | PRIMARY                  |       95 |    3 |    73 |        3 |
| dbt3/customer                     | i_c_nationkey            |       96 |    0 |    73 |        4 |
| dbt3/lineitem                     | PRIMARY                  |       97 |    3 |    74 |        3 |
| dbt3/lineitem                     | i_l_shipdate             |       98 |    0 |    74 |        4 |
```

可以看出数据库dbt3下的customer表、lineitem表主键索引根页的page number均为3，而其他的二级索引page number为4。关于二级索引与主键索引的区别请参考MySQL相关书籍，本文不在此介绍。

下面我们对数据库表空间文件做想相关的解析：

```
sh-3.2# ls -l *.ibd
-rw-rw--w-  1 _mysql  _mysql    41943040  3  9  2017 customer.ibd
-rw-rw--w-  1 _mysql  _mysql  2462056448  3  9  2017 lineitem.ibd
-rw-rw--w-  1 _mysql  _mysql      114688  3  9  2017 nation.ibd
-rw-rw--w-  1 _mysql  _mysql   293601280  3  9  2017 orders.ibd
-rw-rw--w-  1 _mysql  _mysql    41943040  3  9  2017 part.ibd
-rw-rw--w-  1 _mysql  _mysql   184549376  3  9  2017 partsupp.ibd
-rw-rw--w-  1 _mysql  _mysql       98304  3  9  2017 region.ibd
-rw-rw--w-  1 _mysql  _mysql    10485760  3  9  2017 supplier.ibd
-rw-rw--w-  1 _mysql  _mysql       98304  3  9  2017 time_statistics.ibd
```

因为主键索引B+树的根页在整个表空间文件中的第3个页开始，所以可以算出它在文件中的偏移量：16384*3=49152（16384为页大小）。

另外根据《InnoDB存储引擎》中描述在根页的64偏移量位置前2个字节，保存了page level的值，因此我们想要的page level的值在整个文件中的偏移量为：

16384*3+64=49152+64=49216，前2个字节中。

接下来我们用hexdump工具，查看表空间文件指定偏移量上的数据：

```
sh-3.2# hexdump -s 49216 -n 10 lineitem.ibd
000c040 00 02 00 00 00 00 00 00 05 a6
000c04a
sh-3.2# hexdump -s 49216 -n 10 region.ibd
000c040 00 00 00 00 00 00 00 00 05 b8
000c04a
sh-3.2# hexdump -s 49216 -n 10 customer.ibd
000c040 00 02 00 00 00 00 00 00 05 a4
000c04a
```

linetem表的page level为2，B+树高度为page level+1=3；

region表的page level为0，B+树高度为page level+1=1；

customer表的page level为2，B+树高度为page level+1=3；

这三张表的数据量如下：

```
mysql> select count(*) from lineitem;
+----------+
| count(*) |
+----------+
|  6001215 |
+----------+
1 row in set (3.14 sec)

mysql> select count(*) from region;
+----------+
| count(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from customer;
+----------+
| count(*) |
+----------+
|   150000 |
+----------+
1 row in set (0.08 sec)
```

总结：

lineitem表的数据行数为600多万，B+树高度为3，customer表数据行数只有15万，B+树高度也为3。可以看出尽管数据量差异较大，这两个表树的高度都是3，换句话说这两个表通过索引查询效率并没有太大差异，因为都只需要做3次IO。那么如果有一张表行数是一千万，那么他的B+树高度依旧是3，查询效率仍然不会相差太大。

region表只有5行数据，当然他的B+树高度为1。

最后回顾一道面试题

有一道MySQL的面试题，为什么MySQL的索引要使用B+树而不是其它树形结构?比如B树？现在这个问题的复杂版本可以参考本文；

他的简单版本回答是：

因为B树不管叶子节点还是非叶子节点，都会保存数据，这样导致在非叶子节点中能保存的指针数量变少（有些资料也称为扇出），指针少的情况下要保存大量数据，只能增加树的高

度，导致IO操作变多，查询性能变低；

小结

本文从一个问题出发，逐步介绍了InnoDB索引组织表的原理、查询方式，并结合已有知识，回答该问题，结合实践来证明。当然为了表述简单易懂，文中忽略了一些细枝末节，比如一个页中不可能所有空间都用于存放数据，它还会存放一些少量的其他字段比如page level，index number等等，另外还有页的填充因子也导致一个页不可能全部用于保存数据。关于二级索引数据存取方式可以参考MySQL相关书籍，他的要点是结合主键索引进行回表查询。