由MongDB中的B+树引发的随想

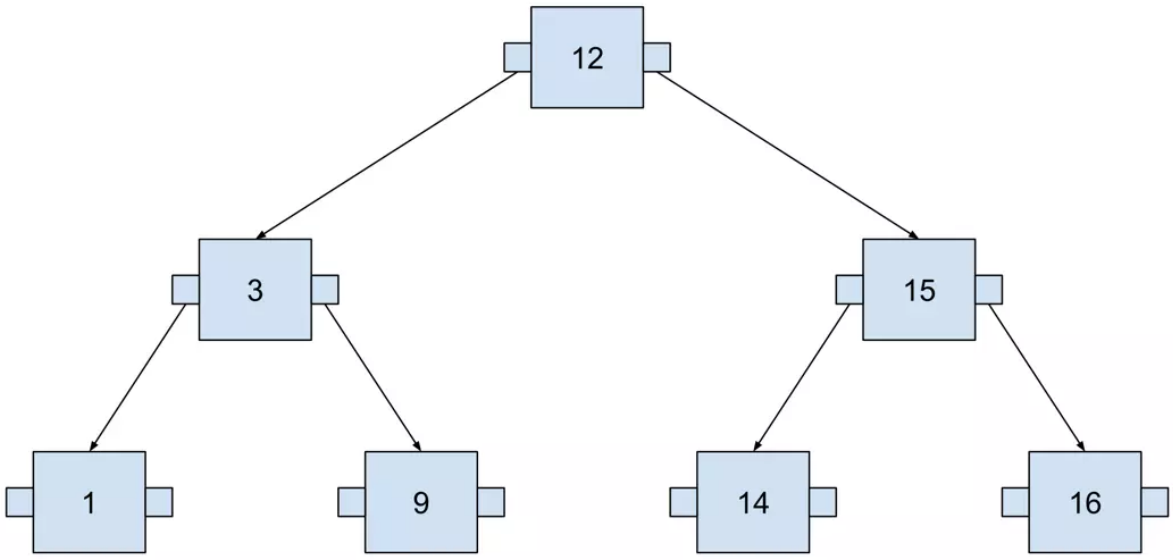
> 随想的意思，就是可能暂时还没有拿到正确答案，但是我认为这是一个有趣的问题。

缘起

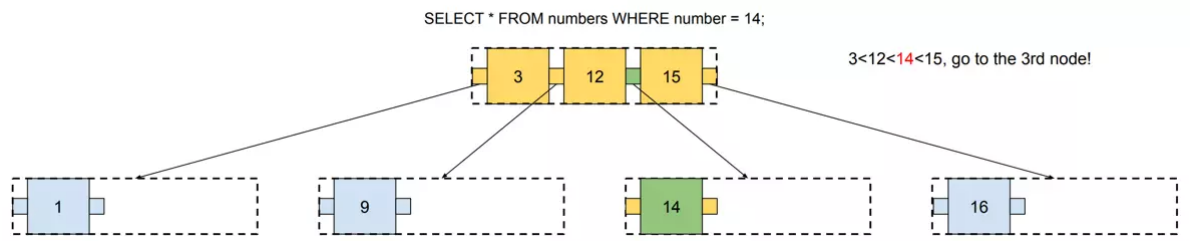
偶然在腾讯技术工程的一篇技术文章《MongoDB 索引使用总结》：

> MongoDB 底层是如何存储数据的，一个 collection 一个文件吗?索引在底层是如何组织的？ 一个 collection 对应到底层存储引擎就是一个文件，另外每个索引也是单独的文件，每个数据和索引文件的默认结构是 b 树，用户建表的时候也可以指定 lsm 结构，不过绝大多数用户基本都是使用 b 树结构

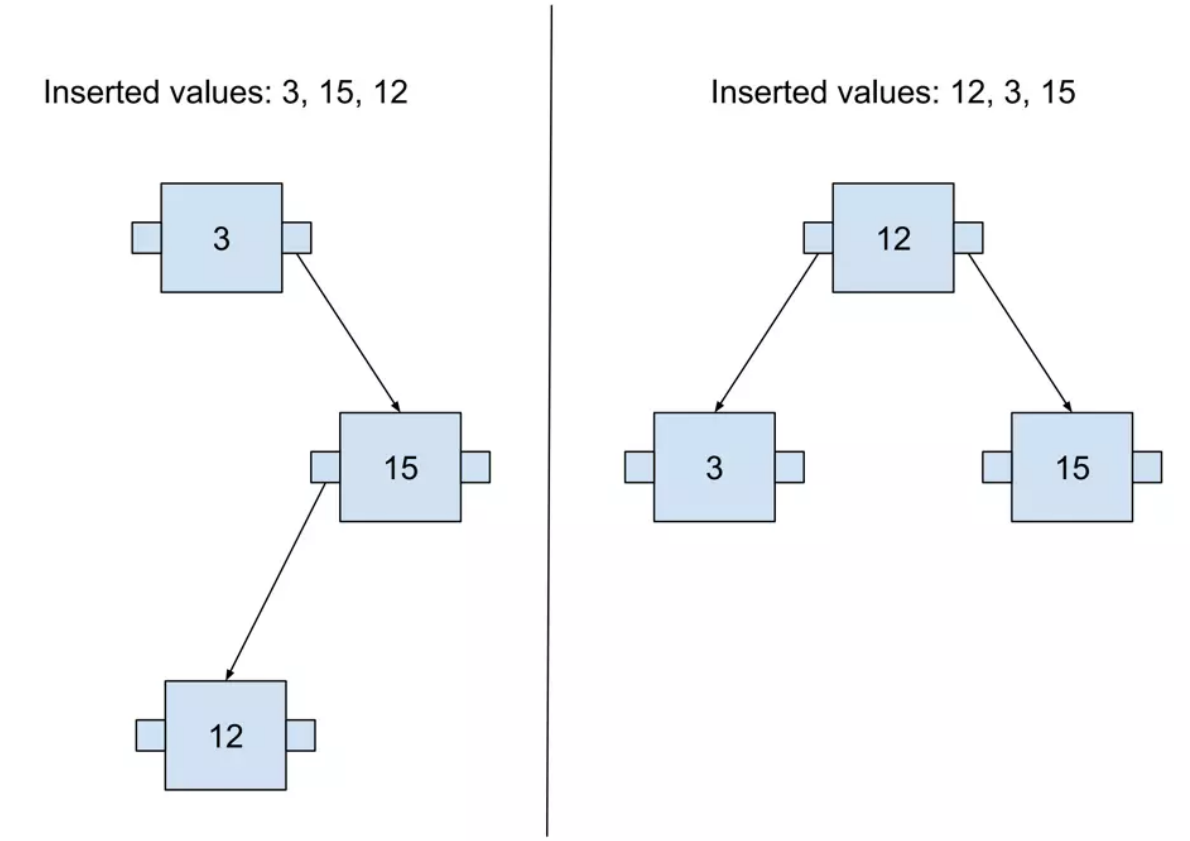
嗯哼？ B树，跟MySQL 不一样？ 说起B树，我想起了什么呢，我想到了二叉搜索树，二叉搜索树中，右边节点的数字总是比双亲节点大，左节点的数据总是比双亲节点要小:



但是我们不能使用这种数据结构来当存储数据，原因在于树太高会增大I/O次数，这会让查询变得很慢，那么我们就希望将树变得矮一点，将树变矮的一个思路是让单个节点承载更多的数据，将多个值打包到一个节点上像下面这样:



但是这还不够，在一些情况下树会进行退化，像下面这样:



在插入的值基本有序的情况，树会退化为链表，这无疑是我们不想看到的，于是我们需要一种自平衡算法。所以一般来说B树具备两个特点:

- 插入新值的时候会有一个自平衡算法来防止树的退化
- 每个节点包含超过一个值

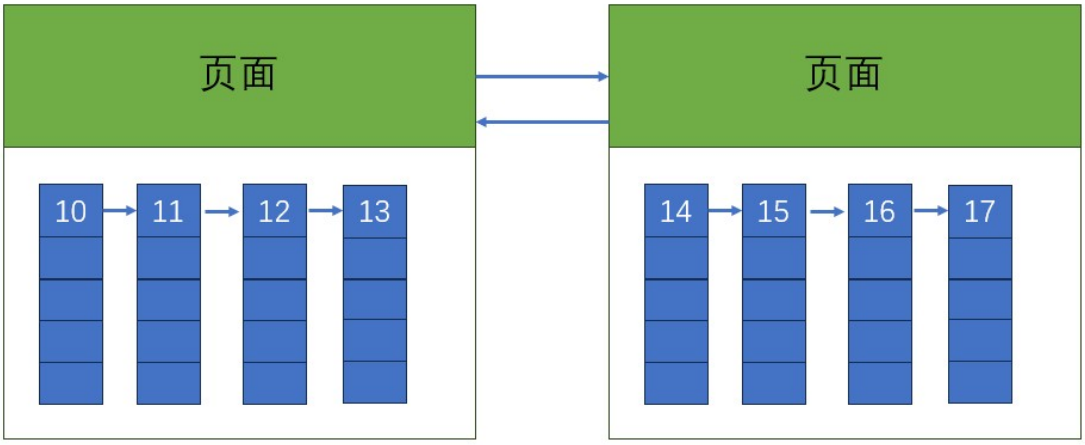
而在MySQL中，MySQL以页为单位管理数据，数据页里面存储了行数据，在InnoDB中页的大小一般为16KB，通过show global status like 'innodb_page_size'可以查看这一页的大小:

```
1  show global status like 'innodb_page_size';
```

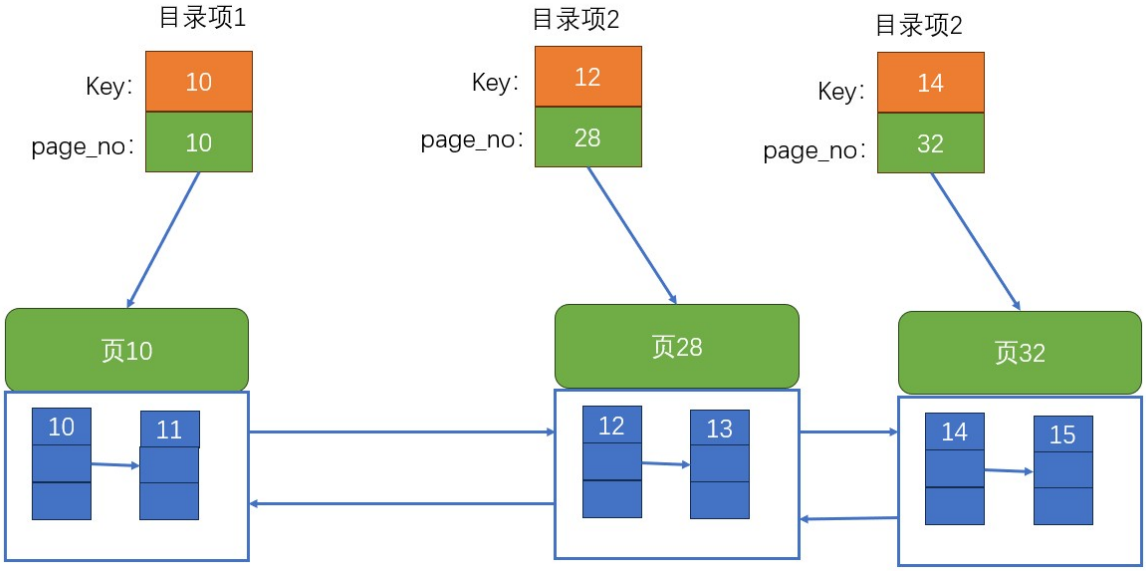
信息 | 结果 1 | 剖析 | 状态

Variable_name	Value
Innodb_page_size	16384

16kb * 1024 = 16384。在这个数据页里面存储了我们日常使用的行数据，每个行记录里面存储了下一行的地址，所有的记录按照主键从小到大组成了一个单链表。数据页之间是一个双向链表。

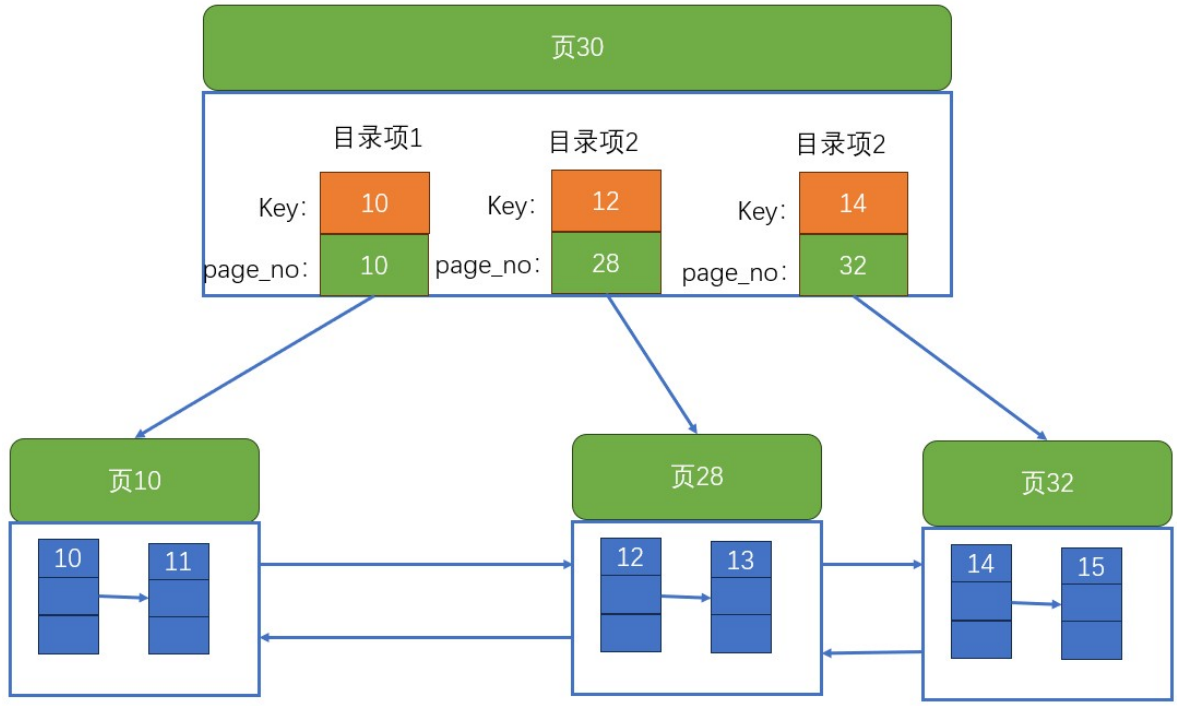


那怎么构成B+树呢？如果用双向链表来去查找速度，虽然数据之间基本有序，我们可以用二分法来定位记录，如果用非主键列来遍历恐怕速度会慢的惊人，从头遍历到尾。那该怎么优化呢，还是转成B+树，让我们先看有一个简单的方案， 现在让我们为页编码，建立目录项:

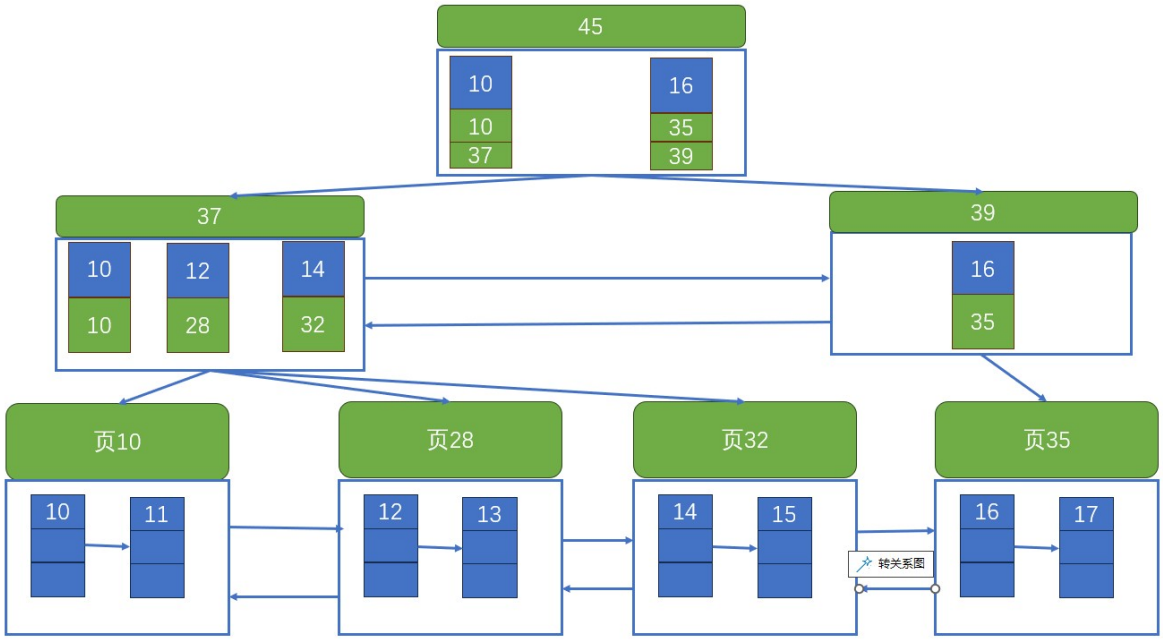


以页10为例，它对应目录项2，这个目录项中包含着该页的页号10以及该页中用户记录的最小主键值10。我们只需要将这个目录项放在一个数组里面，就可以实现根据主键值快速查找某条记录的功能了。比方说，我们想找主键值为13的记录，因为13大于12，所以我们就去页28中寻找。

在MySQL中复用了普通数据记录页做目录页，只不过目录页里面放的是主键值和页号而已，那InnoDB如何区分数据页和数据页呢，一个普通得到数据记录页分为记录头和记录体，在记录头里面有个record_type字段，这个字段的不同取值代表了不同的语义。也就是说数据页存储普通数据项记录，目录页存储目录项记录。上面的索引方案就变成了下面这样:



在目录页里面存储了最大记录和最小记录的地址，就能根据这些记录来定位当前处于哪些目录页里面，然后在目录页里面在进行遍历定位到目录项，在定位到具体在哪一页。那么问题来了，如果我们的表里面的数据非常多会产生很多存储目录项记录的页，那么我们我们怎么根据主键值快速定位一个存储目录项记录的页呢? 其实也很简单，为这些目录页再生成一个更高级的目录，就像一个多级目录一样，大目录里嵌套小目录，小目录里才是实际的数据。



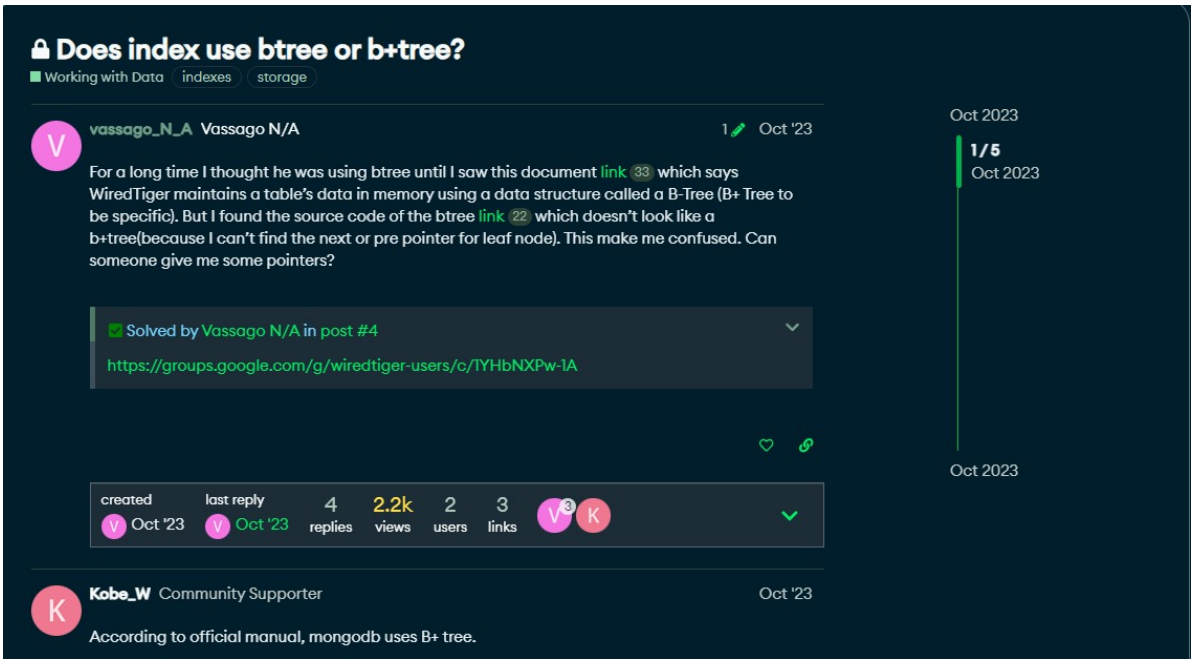
在MySQL中B+树按叶子节点是否存储完整的用户记录可以分为两类: 聚簇索引和非聚簇索引。聚簇索引中叶子结点存储了所有列的值，非聚簇索引只存储索引列、主键值。聚簇索引的特点是:

1. 使用记录主键值的大小进行记录和页的排序，这包括三个方面的含义：
    - 页内的记录是按照主键的大小顺序排成一个单向链表。
    - 各个存放用户记录的页也是根据页中用户记录的主键大小顺序排成一个双向链表。
    - 存放目录项记录的页分为不同的层次，在同一层次中的页也是根据页中目录项记录的主键大小顺序排成一个双向链表。
2. `B+`树的叶子节点存储的是完整的用户记录。

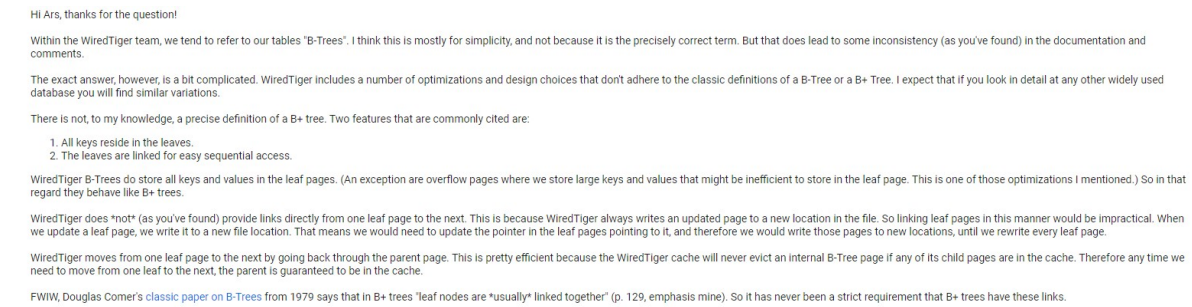
    所谓完整的用户记录，就是指这个记录中存储了所有列的值（包括隐藏列）。

回到MongDB中来

回顾了MySQL中的B+树，我就想到了MongDB中的B树，带着这个问题我打开了搜索引擎，开始搜索: “MongDB B tree”. 于是心满意足看到有人在MongDB的论坛上看到有人这么问“Does index use btree or b+tree?”



顺着这个链接点过去看到Keith Smith的回复:



Hi Ars, thanks for the question!

Within the WiredTiger team, we tend to refer to our tables "B-Trees". I think this is mostly for simplicity, and not because it is the precisely correct term. But that does lead to some inconsistency (as you've found) in the documentation and comments.

The exact answer, however, is a bit complicated. WiredTiger includes a number of optimizations and design choices that don't adhere to the classic definitions of a B-Tree or a B+ Tree. I expect that if you look in detail at any other widely used database you will find similar variations.

There is not, to my knowledge, a precise definition of a B+ tree. Two features that are commonly cited are:

1. All keys reside in the leaves.
2. The leaves are linked for easy sequential access.

WiredTiger B-Trees do store all keys and values in the leaf pages. (An exception are overflow pages where we store large keys and values that might be inefficient to store in the leaf page. This is one of those optimizations I mentioned.) So in that regard they behave like B+ trees.

WiredTiger does *not* (as you've found) provide links directly from one leaf page to the next. This is because WiredTiger always writes an updated page to a new location in the file. So linking leaf pages in this manner would be impractical. When we update a leaf page, we write it to a new file location. That means we would need to update the pointer in the leaf pages pointing to it, and therefore we would write those pages to new locations, until we rewrite every leaf page.

WiredTiger moves from one leaf page to the next by going back through the parent page. This is pretty efficient because the WiredTiger cache will never evict an internal B-Tree page if any of its child pages are in the cache. Therefore any time we need to move from one leaf to the next, the parent is guaranteed to be in the cache.

FWIW, Douglas Comer's classic paper on B-Trees from 1979 says that in B+ trees "leaf nodes are *usually* linked together" (p. 129, emphasis mine). So it has never been a strict requirement that B+ trees have these links.

> Within the WiredTiger team, we tend to refer to our tables "B-Trees". I think this is mostly for simplicity, and not because it is the precisely correct term. But that does lead to some inconsistency (as you've found) in the documentation and comments.

在WiredTiger团队，我们倾向于表述为B- tree。 我认为这主要是为了简单起见，而不是因为它是一个非常精确的术语(正如你发现的那样)。

> The exact answer, however, is a bit complicated. WiredTiger includes a number of optimizations and design choices that don't adhere to the classic definitions of a B-Tree or a B+ Tree. I expect that if you look in detail at any other widely used database you will find similar variations.
>
> 然后真实的答案会稍微有些复杂，WiredTiger 包含许多优化和设计选择，这些优化和设计并没有完全遵循B树或B+树的经典定义。我想你研究任何其他广泛使用的数据库会发现类似的变化。
>
> There is not, to my knowledge, a precise definition of a B+ tree. Two features that are commonly cited are:
>
> 据我所知，没有一个关于B+ Tree的精确的定义，通常提到的两个特点是:
>
> 1. All keys reside in the leaves.
>
>   所有的键都存储在叶子节点中。

> 2. The leaves are linked for easy sequential access.
>
>   叶子节点之间有联系，便于顺序访问。

> WiredTiger B-Trees do store all keys and values in the leaf pages. (An exception are overflow pages where we store large keys and values that might be inefficient to store in the leaf page. This is one of those optimizations I mentioned.) So in that regard they behave like B+ trees.
>
> WiredTiger 中的B-Tree确实将所有键和值都存储在叶子页面中(一个例外是溢出页面，,我们在其中存储可能效率低下的大键和大值,不适合存储在叶子页面中。这是我提到的优化之一)。 所以在这个方面，它的行为类似B+ 树。

> WiredTiger does *not* (as you've found) provide links directly from one leaf page to the next. This is because WiredTiger always writes an updated page to a new location in the file.
>
> WiredTiger 不直接提供从一个页面到下一个页面的链接。这是因为WiredTiger 总是将更新后的页面写入到文件中的新位置。

> So linking leaf pages in this manner would be impractical. When we update a leaf page, we write it to a new file location.
>
> 所以，以这种方式链接是不切实际的，当更新一个页面的时候，我们会将其写入到一个新的位置。
>
> That means we would need to update the pointer in the leaf pages pointing to it, and therefore we would write those pages to new locations, until we rewrite every leaf page.
>
> 这意味着我们需要更新指向它的叶子页面中的指针,因此我们将这些页面写入新的位置,直到我们重写每个叶子页面。

于是想到了CopyOnWriteArrayList和ConcurrentHashMap

注意不是在原页面上进行更新，而是将更新之后的页面上写入到一个新的位置，这看起来有点像是CopyOnWriteArrayList的操作:

```
public E set(int index, E element) {
    // 获取锁
    final ReentrantLock lock = this.lock;
    // 开始锁定
    lock.lock();
    try {
        // 获取数组
        Object[] elements = getArray();
        // 获取旧值
        E oldValue = get(elements, index);
        if (oldValue != element) {
            // 获取数组长度
            int len = elements.length;
            // 复制一份
            Object[] newElements = Arrays.copyOf(elements, len);
            // 在新的数组上赋值
            newElements[index] = element;
            // 将新数组写回去
            setArray(newElements);
        } else {
            // 相等将原来的数组放入
            setArray(elements);
        }
        return oldValue;
```

```
    } finally {
        lock.unlock();
    }
}
public E get(int index) {
  return get(getArray(), index);
}
```

这是解决线程安全集合的一个思路，读写并发会导致问题，那就在写的时候排队执行，适合读多写少的场景，因为每次写都是在原先的基础上复制一份，避免在一个数组上操作带来问题，频繁写入会消耗大量内存。我时常对这个并发安全集合不满意，某些情况下我可能是读少写多，这回消耗大量内存，于是想到了ConcurrentHashMap，对于ConcurrentHashMap来说读操作同样没上锁，而put操作则是能上锁就尽量不上锁:

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
        // 首先判断是否为空
        if (key == null || value == null) throw new NullPointerException();
        // 然后计算哈希值
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            // 如果数组为空,初始化数组
            if (tab == null || (n = tab.length) == 0)
                //
                tab = initTable();
            // 如果对应的位置上没有值,就CAS尝试写入
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
                if (casTabAt(tab, i, null,
                             new Node<K,V>(hash, key, value, null)))
                    break;
            }
            // 走到这里说明有值了，然后判断是否正在扩容，则帮助扩容并返回最新table[]
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
                // 部分代码省略
                // 进入同步代码块,挂链表，树化
                synchronized (f) {}
        }
        addCount(1L, binCount);
}
```

不允许key和value为空的原因是，ConcurrentHashMap的作者Doug Lea认为:

> The main reason that nulls aren't allowed in ConcurrentMaps (ConcurrentHashMaps, ConcurrentSkipListMaps) is that ambiguities that may be just barely tolerable in non-concurrent maps can't be accommodated.
>
> ConcurrentHashMap不允许为空是因为非并发的Map可以容忍这种歧义，并发Map中是不可以接受的。
>
> The main one is that if `map.get(key)` returns `null`, you can't detect whether the key explicitly maps to `null` vs the key isn't mapped.

> 在非并发Map中如果map.get(key) 返回为null，有另种可能性，一种不存在，一种value就是null。
>
> In a non-concurrent map, you can check this via `map.contains(key)`, but in a concurrent one, the map might have changed between calls.
>
> 如果 `map.contains(key)` 返回 true,说明key存在,只是它的值为 null;如果返回 false,说明键不存在。 但是在并发中调用可能，问题在于,在你调用 `map.get(key)` 和 `map.contains(key)` 之间,Map可能已经被其他线程修改了

我们举个例子来说明，假设我们允许空值进入ConcurrentHashMap：

1. 线程 A 调用 `map.get(key)`,返回 null。
2. 在线程 A 调用 `map.contains(key)` 之前,线程 B 将 `<key, null>` 这个键值对插入到映射中。
3. 线程 A 调用 `map.contains(key)`,返回 true。

这样就出现了歧义，返回为null，究竟是不存在还是存在这个ConcurrentHashMap中只是value为null。

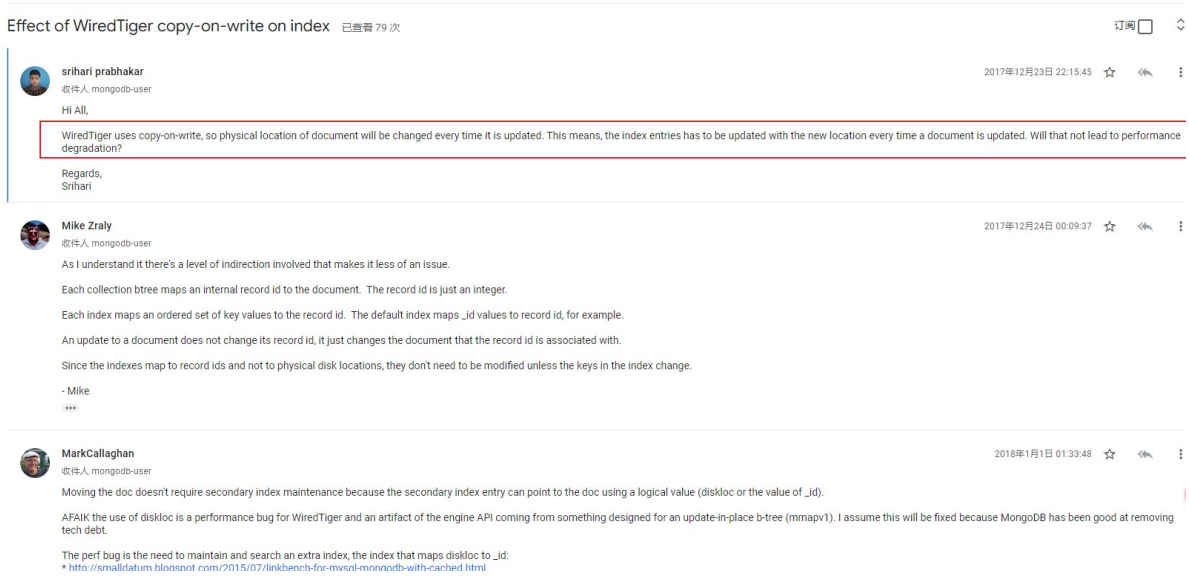
对比这个实现我就想要一个ConcurrentArrayList。略过这个不谈，通过ConcurrentHashMap是如何实现线程安全的，我们就可以的得出HashMap为什么是线程不安全的:

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        // 多个线程可能会同时执行初始化
        n = (tab = resize()).length;
    // 丢值 两个key hash值形同,都走到这一行,然后没有形成链表
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
```

```
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

MongDB 也是COW?

上面只是我们的推测，我还想验证一下我的猜想，于是我在搜索引擎中搜索 MongDB copy-on-write 关键词，于是搜到了一个人这样问:



WiredTiger(MongDB存储引擎的名字) 使用COW，每次更新的时候文档的位置都会被改变，因此文档的物理位置在每次更新时都会发生变化。这意味着，每次更新文档时，都必须用新的位置更新索引条目。这会不会导致性能下降。看这个回复，早期更新策略和MySQL一样，都是就地更新，后面改成COW机制了。我们不做过多的深究。

接着回到MongDB

> WiredTiger moves from one leaf page to the next by going back through the parent page. This is pretty efficient because the WiredTiger cache will never evict an internal B-Tree page if any of its child pages are in the cache. Therefore any time we need to move from one leaf to the next, the parent is guaranteed to be in the cache.FWIW, Douglas Comer's classic paper on B-Trees from 1979 says that in B+ trees "leaf nodes are *usually* linked together" (p. 129, emphasis mine). So it has never been a strict requirement that B+ trees have these links.
>
> WiredTiger 通过回溯父页面来从一个叶页面移动到下一个叶页面。这样做非常有效，因为如果 B-Tree 的任何子页面都在缓存中，WiredTiger 缓存就不会驱逐该内部页面。因此，当我们需要从一个叶子页移动到下一个叶子页时，父页面一定会在缓存中。
>
> 顺便提一下, Douglas Comer 在 1979 年发表的关于 B 树的经典论文中提到，在 B+ 树中，"叶节点通常**连接在一起"（第 129 页，着重号为笔者所加）。因此，B+ 树必须有这些链接从来都不是一个严格的要求

到这里MongDB团队已经给出答案了，也就是B+树没有严格定义，B+树的作者也只是提到叶子结点通常链接在一起，但这不是一个严格的要求，所以《MongoDB 索引使用总结》中说索引的结构是B树页没问题，因为B+树没有严格定义，MongDB用的是B树的变体，只是将数据存储在叶子结点上，但叶子结点上没有维持双向链表。但如果用一样的名称，大家也许会认为MySQL的索引结构B+树和MongDB用的索引结构B+树是一样的，由于精确定义所以B树会有许多变体，下次如果别人跟你讨论B树，要多问问你说的B树做了哪些变体。

对比一下MongDB 和 MySQL

写到这里就想起几年前写到一篇文章《MongDB学习笔记(一) 初遇篇》基本介绍了一下MongDB，关系型数据库以行为单位，MongDB以json为单位，在MongDB中称之为文档。行存储在表里面，文档存储在集合里面。然后我们提到了为什么引入了MongDB，我们给出了几个理由: 自然、性能、灵活、扩展性。以我现在的观点来看这些观点都是没有直逼要害。

1. 自然

   文档(JSON)模型与面向对象的数据表达方式更相似更自然。与关系型数据库中的表结构不同，文档(JSON)中可以嵌入数组和子文档(JSON)，就像程序中的数组和成员变量一样。这是关系型数据库三范式不允许。但实际中我们经常看到一对多和一对一的的数据。比如，一篇博客文章的Tag列表作为文章的一部分非常直观，而把Tag与文章的从属关系单独放一张表里就不那么自然。SQL语音可以很精确地形式化。然而三大范式强调的数据没有任何冗余，并不是今天程序员们最关心的问题，他们用着方便不方便才是更重要的问题。

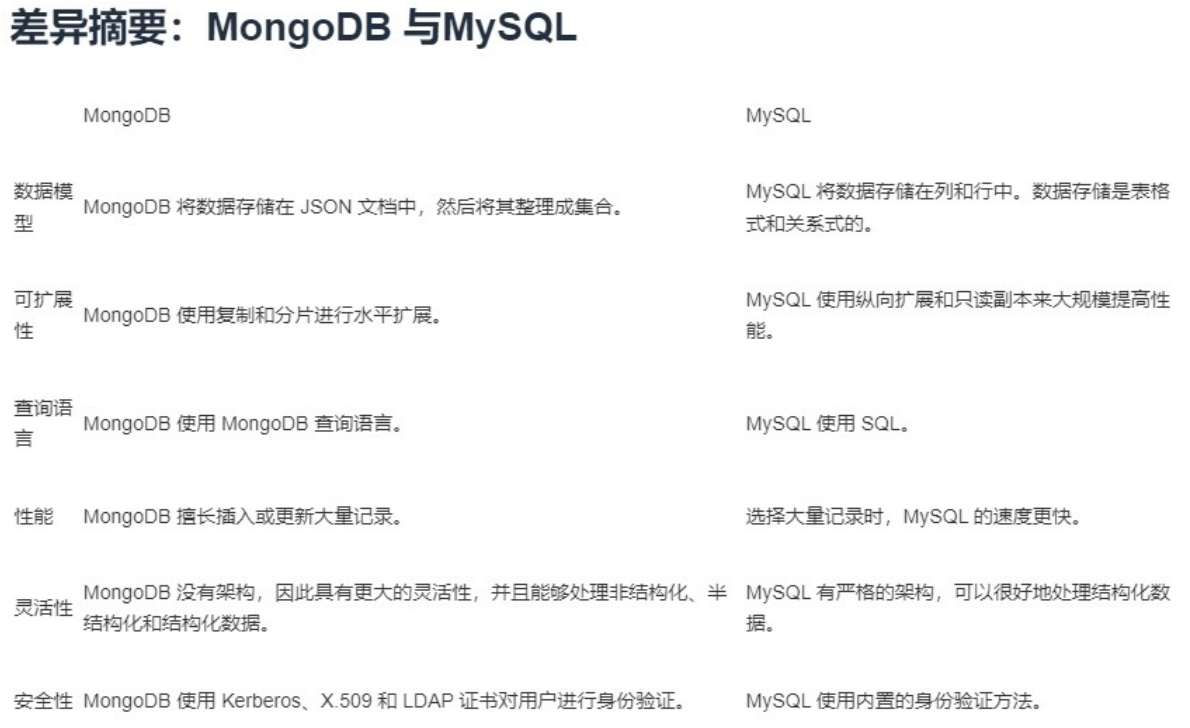
2. 性能

   三大范式带来的jojn，有的时候为了满足一个查询，不得不join多表，但join多表的代价是随着数据的提升，查询速度会变得很慢，更简单的访问模式可以让开发者更容易理解数据库的性能表现，举一个典型的场景就是，我join了十张表，在这样的情况下，我该如何加索引才能获得最优对的查询速度。

3. 灵活

如果需要加字段，从数据库到应用层可能都需要改一遍，尽管有些工具可以把它自动化，但这仍然是一个复杂的工作。 MongDB没有Schema，就不需要改动数据库，只需要在应用层做必要的改动。(这句话可以这么理解，仔细看上面画的MongDB存取数据的图，集合中现在存取的学生都有三个字段，事实上MongDB允许文档拥有不同的字段)

我希望有更简洁的概括，有一天我在瞎翻文档的时候翻到了亚马逊官网的一张图，我觉得简洁而有力:



灵活性上一言以蔽之，处理非结构化、半结构化和结构化数据，那怎么理解这个结构化，数据都具备相同的组成，就是关系型数据库所追求的结构化，结构化意味着可预测，而半结构化的数据，在这个语境下指的就是MongDB的文档可以有不同的字段，有些文档三个字段，有些文档四个字段，有些文档里面可以放一个数组和子文档。而关系型数据库我们要为一张表字段，那么这个表的所有数组都会有这个字段。半结构化意味着某些数据可以引用了一些文档集合，一些文档可以没有，限制没有那么强。但是在MySQL之间有引用关系要么通过加表来保存，要么通过在一个字段里面放这种一对多的这种关系，用逗号分割。但是现在关系型数据库也在引入对json的支持，NoSQL和SQL之间的分界线在变得模糊起来。

总结一下

- B+树没有严格定义，MongDB的B+树就和MySQL的B+树实现上有所区别，当别人跟你讨论B+树的时候，可以先对齐一下概念。
- 思想是共通的，比如COW机制，在Java中有体现，在NoSQL中也有体现。
- 所谓的结构化是一种约束，意味着表里面的所有数据都拥有相同的字段， 半结构化和非结构化放松了约束，集合里面的文档可以有不同数量的字段。

参考资料

[1] What are the differences between B trees and B+ trees?

[2] Why does ConcurrentHashMap prevent null keys and values? https://stackoverflow.com/questions/698638/why-does-concurrenthashmap-prevent-null-keys-and-values

[3] Effect of WiredTiger copy-on-write on index https://groups.google.com/g/mongodb-user/c/m3MDwCjT5hY/m/5Bs-ucjPDQAJ