OLTP Index (B+ Tree)

Whole-key data structure

Andy自创的一个概念

- preserve order

节点内key顺序排序

- It means the data structure stores all te digits of a key together in nodes.

一个key是完整存储在节点上，与之相对应的**partial-key** data structure(tries),key可能会离散存储在不同的结点

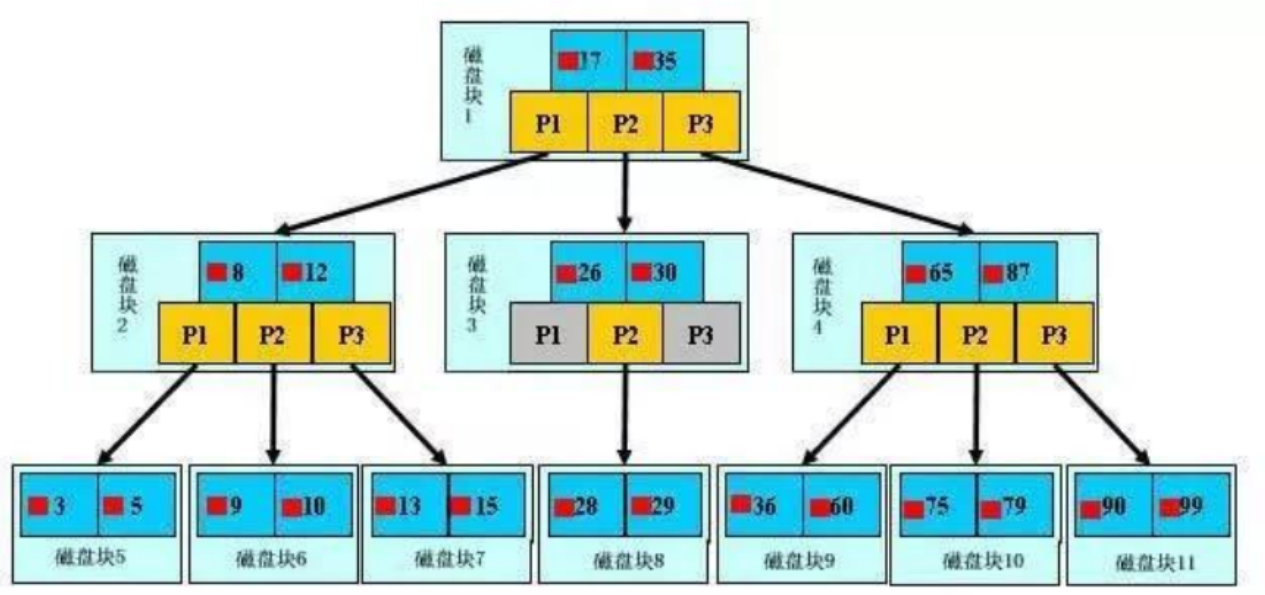
例如某个key是ABC三个字段，那么这三个字段是存储在同一个node的，那么就叫whole-key data structure.

B+ Tree

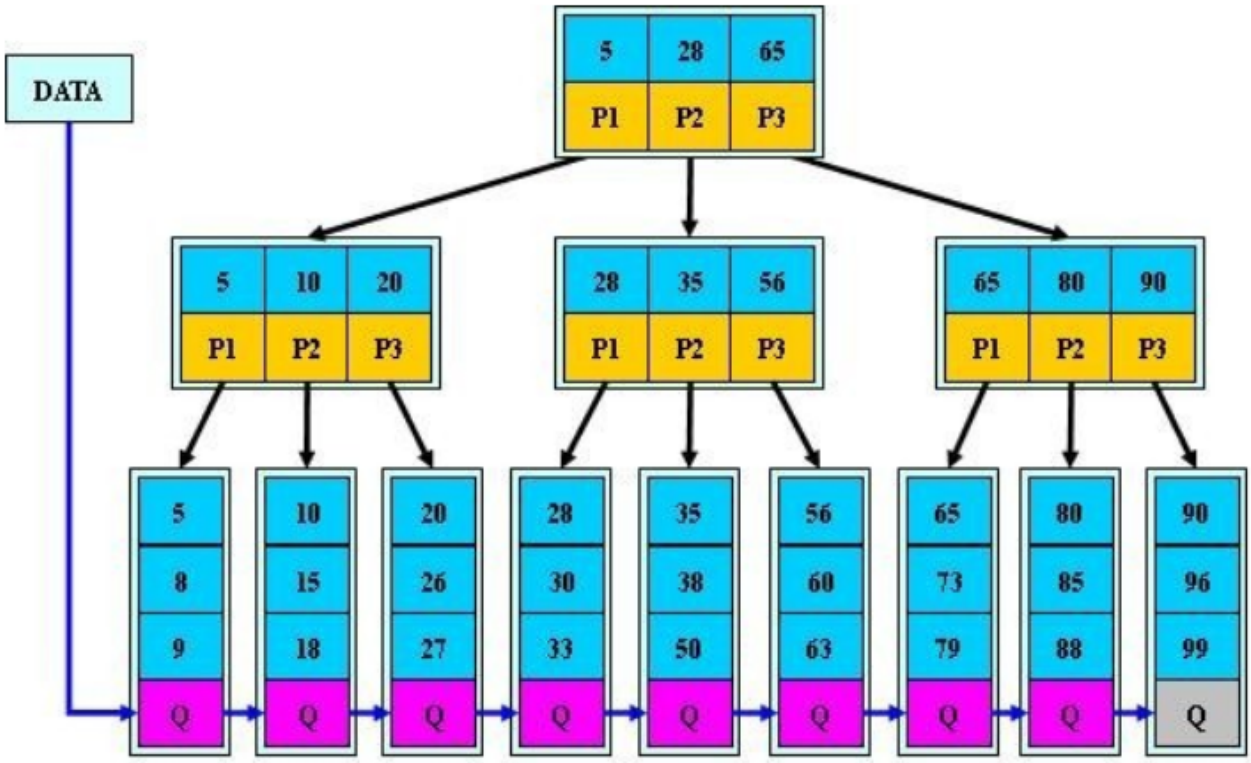
80年代提出的B+Tree是用来解决获取存储在慢速磁盘上顺序排列的数据的索引数据结构。

首先我们要搞清楚B树和B+树的区别

B树



B+树



不同之处：

1.Key存储的位置不同。

B+树的数据都存储在叶子节点上，而B树的数据会存储在每一个节点上，不仅仅是在叶子节点上。

这也就是说，B+树的内部节点仅仅包含索引的信息(子节点磁盘偏移量)，不存储数据本身。

2.叶子节点构造不同

B+树的叶子节点会形成一个有序的单向链表，加快了范围查询和顺序查询的效率。

并且B树的任何key只会出现一次，而B+树key必须出现在叶节点中，也可能在非叶子结点。

3.**性能上不同**

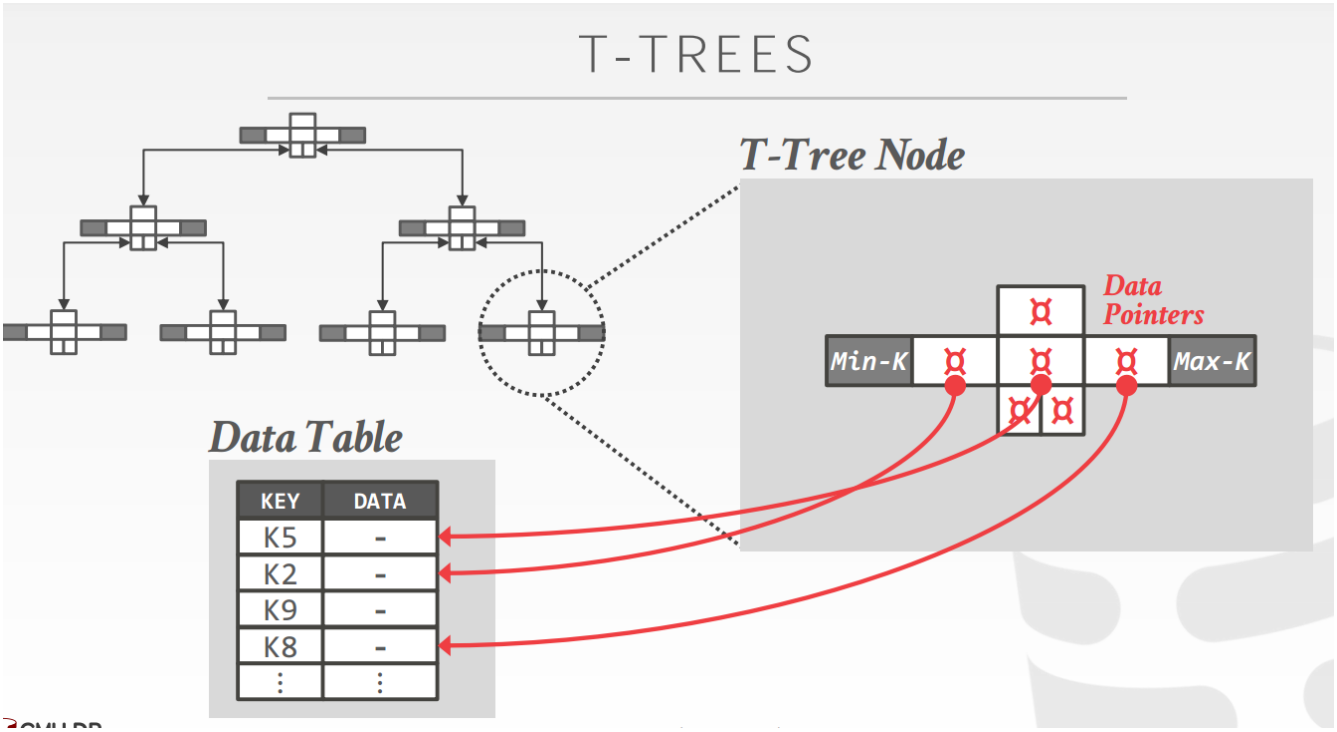
- B树的遍历效率是比B+树要低的，因为B+树的所有叶子节点都用指针连接在一起。只需要遍历叶子节点就可以实现**整棵树的遍历**，而且B+树的查询必须走过从根节点到叶子节点的一条完整路径。不像B树可能要进行一个跨层的遍历。因此B+树的查询效率是比B树要稳定的.因此B树只适合随机检索，而B+树适合随机和顺序检索
- 我们知道索引本身通常就很大，不可能完全存储在内存中。因此索引常常以所以文件存储在磁盘上，B+/-树上的每个节点都对应磁盘一个frame对应OS的一个Page。查找的时候涉及到操作系统的一次换页，涉及到磁盘IO。 我们上面说过，B+树内部节点不需要存储执行某个key的指针（通常64bit），因单个结点可以存储的数据量变多 -> 结点数目减少 ->树的高度减少 -> 查询效率提高。

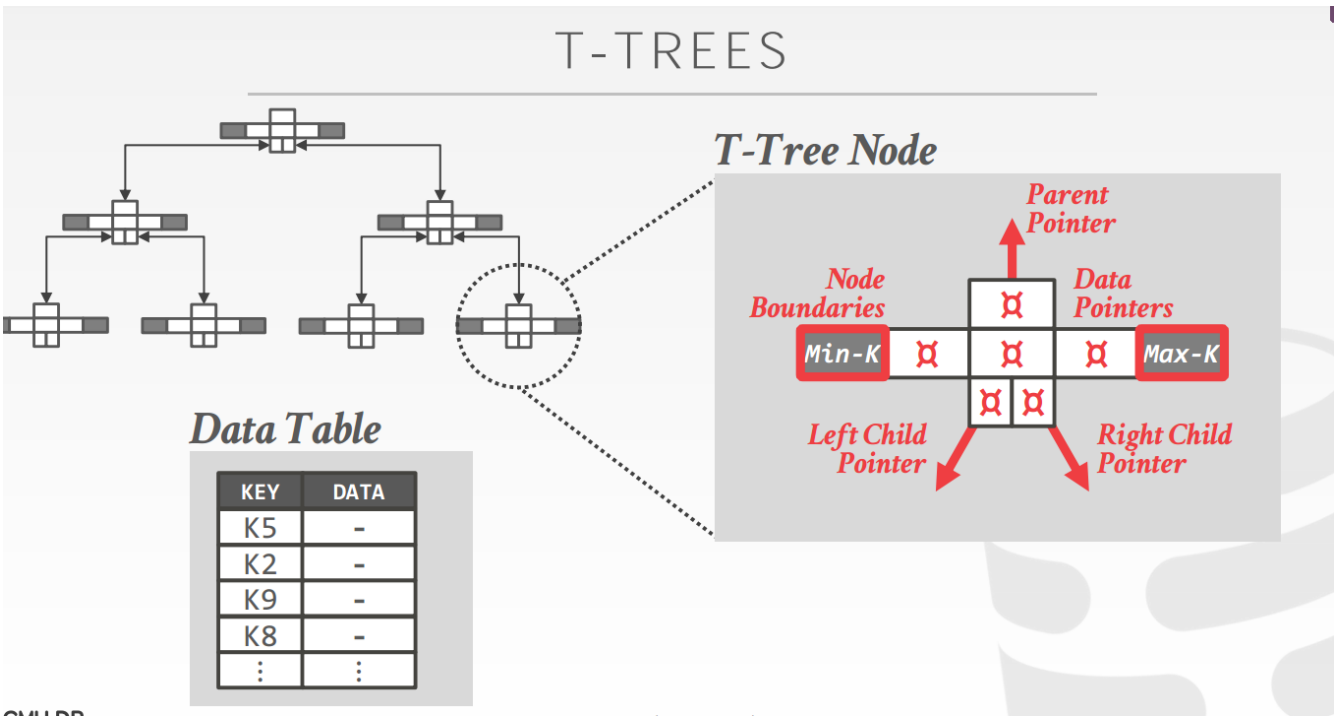
但是对于内存型的数据库来说,不存在磁盘IO作为性能瓶颈问题。Random IO和顺序IO并没有性能上的区别。那么有没有替代B+树的并且能够在多线程环境下性能更好的数据结构？.

T-Tree

和B+树不同，B+树所有key都是存储在叶子节点上，所有非叶子结点都不会存储数据。

而T树会在所有节点上存储指向真实数据的**指针**





Pros and cons:

- Pros:

1.无需在节点存储数据本身，只需要存储数据的指针，节省内存。

- Cons:

1.指针多了->慢遍历,效率低，还会破坏cache局部性

2.难以实现安全并发操作

3.树的结构删除后难以rebalane，因为是基于AVL所以要涉及大量旋转操作，而B+树只涉及分裂，分裂的话只需要调整parent和child node指针的指向即可。

所以T-Tree基本没啥人用。

无法保证树结构调整线程安全的原因?

CAS操作单次只能更新单个地址的值，而如果要实现一个latch-free的B+Tree, 在split/merge要求更新多个指针地址。

解决方法：提供一个`indirect layer that allowed us up update multiple addresses atomically`

所以引入了BW-TREE!

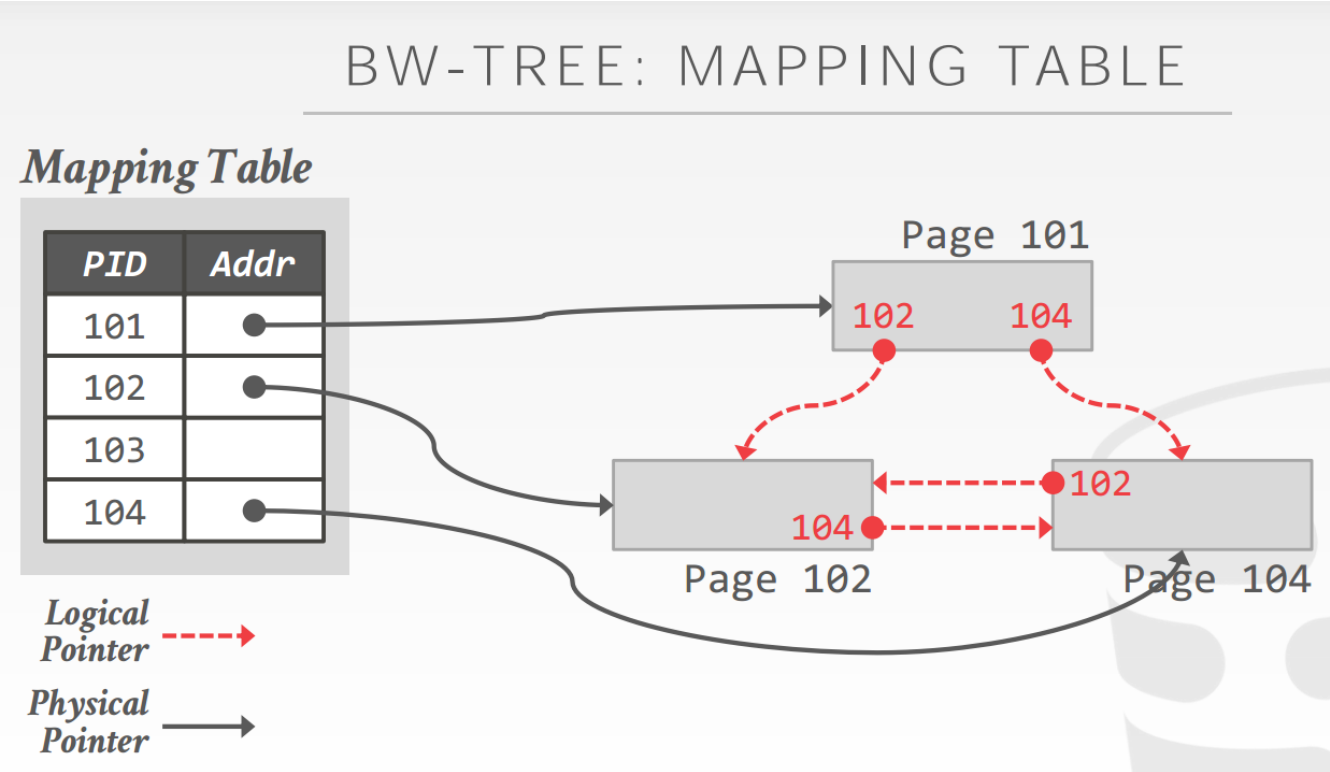
Bw-Tree是一种无锁的索引数据结构,线程使用CAS而不是latch(你可以理解为工程上latch是一个信号量)来对临界区资源进行操作。

前面也提到，如果要实现一个无锁的B+ Tree，叶子节点是不能够用指针连接在一起的。因为随着B+树的分裂，需要同时修改多个结点的指针指向。而CAS单次操作只能改变单个地址的值(通常32/64bit，我不知道现在能不能支持128bit)。

Bw-Tree 引入了一层隐含层叫**Mapping Table**，用于实现逻辑页地址转换为物理地址。所以CAS操作改变的是页的物理地址。

另外一个Trick是引入**Delta Records**,用于记录对单个结点的变更记录。

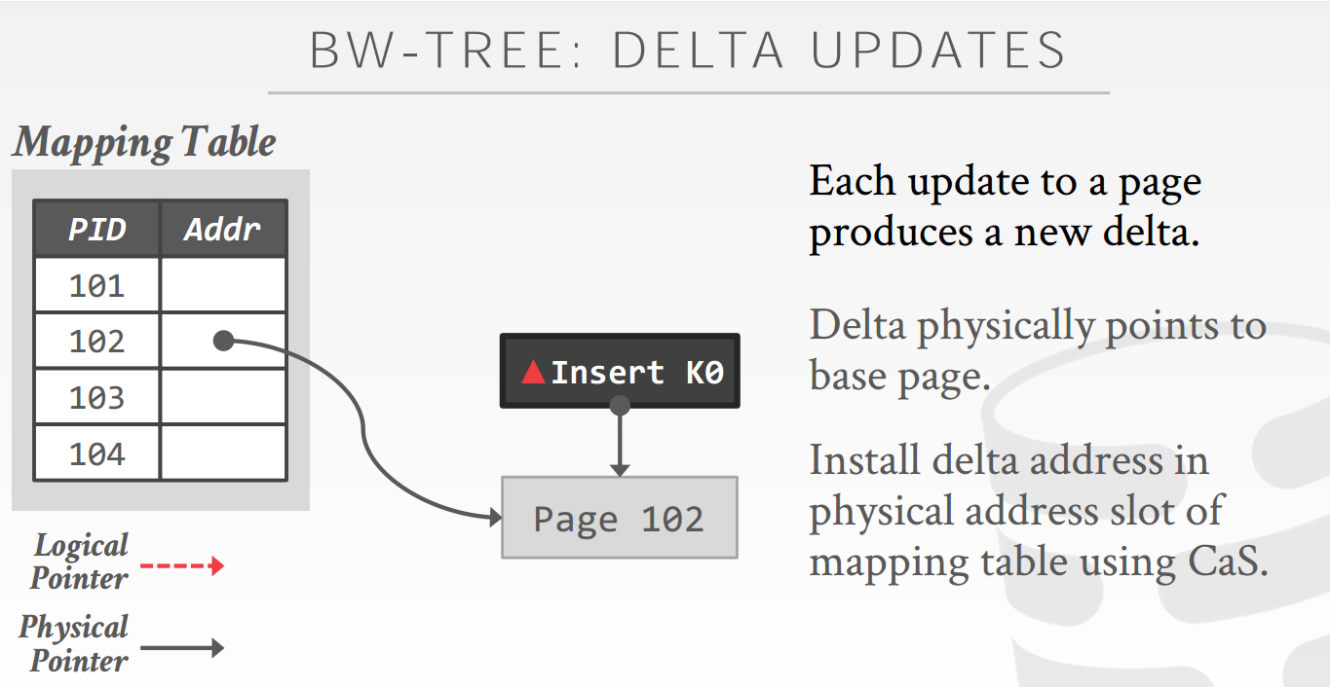
听起来有点复杂，上图;



Mapping Table存储着Page ID到Page physical address的映射

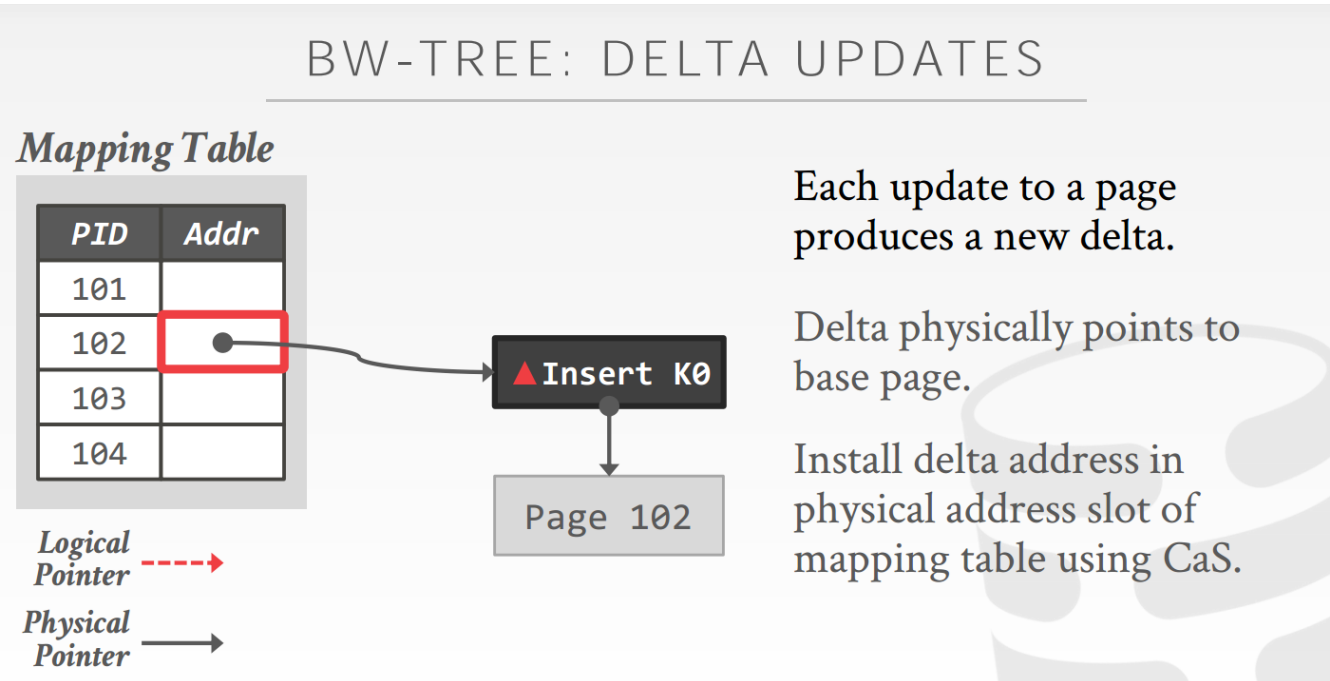
假设现在要遍历这颗树,Page 101要访问Page 102, 那么它实际上不是跟着102这个红色的指针来找到Page 102的物理地址，而是通过查Mapping Table表来找到其真实的物理地址。那么更新树的结构转化为更新Mapping Table的目录项，就可以用CAS操作来保证其线程安全，不需要同时改变多个指针。

Delta Updates的原理：



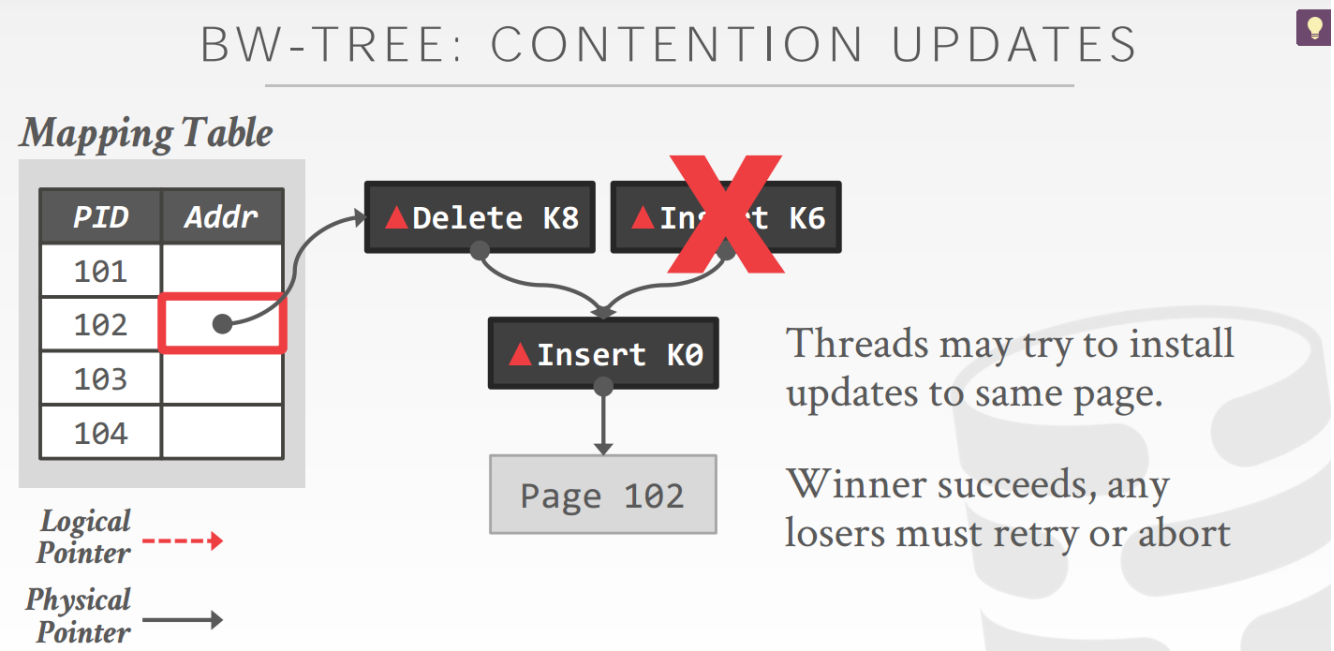
假如对102这个页Insert K0数据项，这个数据项就是**delta record**，其会指向base page即Page 102的地址。

然后CAS更新Addr为Insert K0这个数据项的物理地址。



那假如现在地址指向的是Insert K0这个数据项的地址，但你要找的Key不是K0，是Page 102的其他Key，咋处理？这时候你就忽略这个数据项，沿着链表继续遍历就好了。

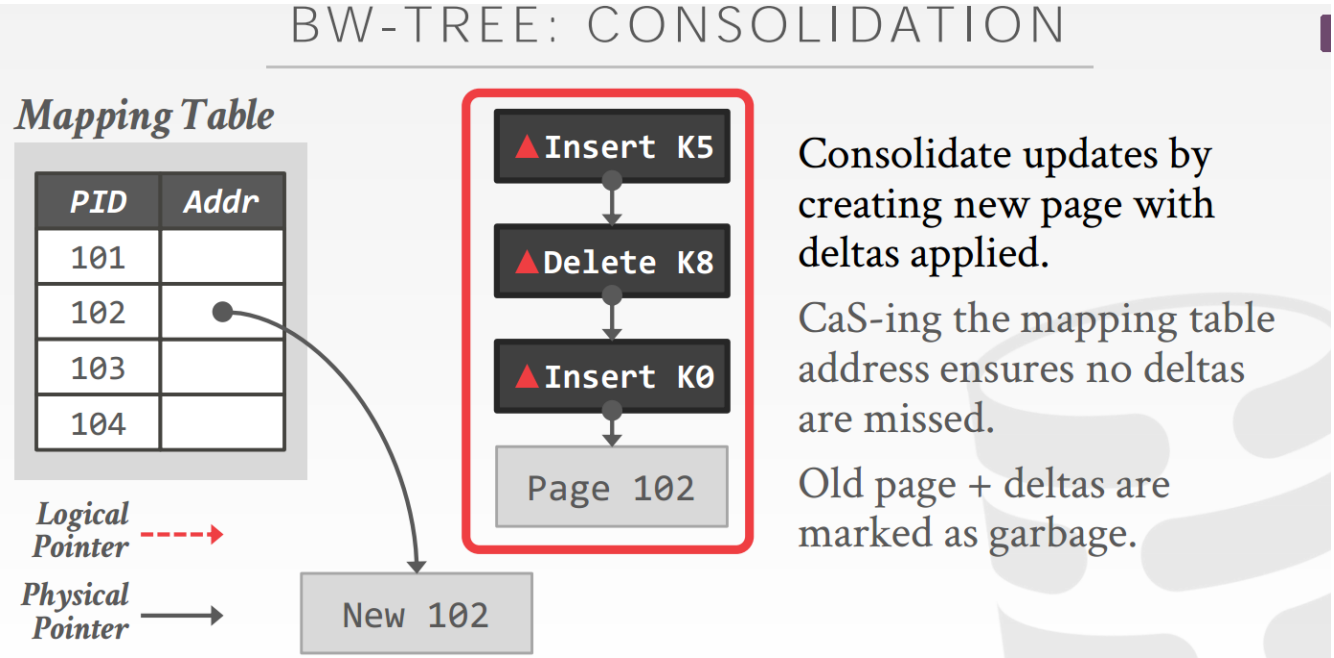
那如果同时有多个delta record更新呢?



CAS保证了只有一个记录能够更新成功，更新失败的就会abort。

垃圾回收

delta record数目一多，链表就会越长，遍历时间无疑会变长，因此要定期进行合并。



开辟一个新的physical page 102，对其依次从INSERT K0， DELETE K8， INSERT K5顺序进行对新页上的数据进行操作，然后红框内的所有目录项都会被标记为垃圾。New 102相当于Old 102的一次最新的状态机转换。最后CAS把Addr更新为新的102 Page Address.

垃圾回收策略：

引用计数法

每一个结点用一个counter来记录正在访问它的线程数量。

对于多核CPU来说性能差：**因为缓存一致性会导致性能受到影响**

个人猜测：为了实现线程可见，这个counter应该会用volatile来修饰，然后引起总线风暴...

因此一般不采取这种方法。

BW Tree选择的垃圾回收策略:

epoch-based garbage collection.

- 引入epoch概念，是一个全局变量，定期更新例如per 10ms
- epoch是一种为了保护对象避免在使用前被提前释放的机制:

通常一个线程要进行一个操作如get,delete,add的时候，会把这个线程和将来可以回收的对象加入到epoch

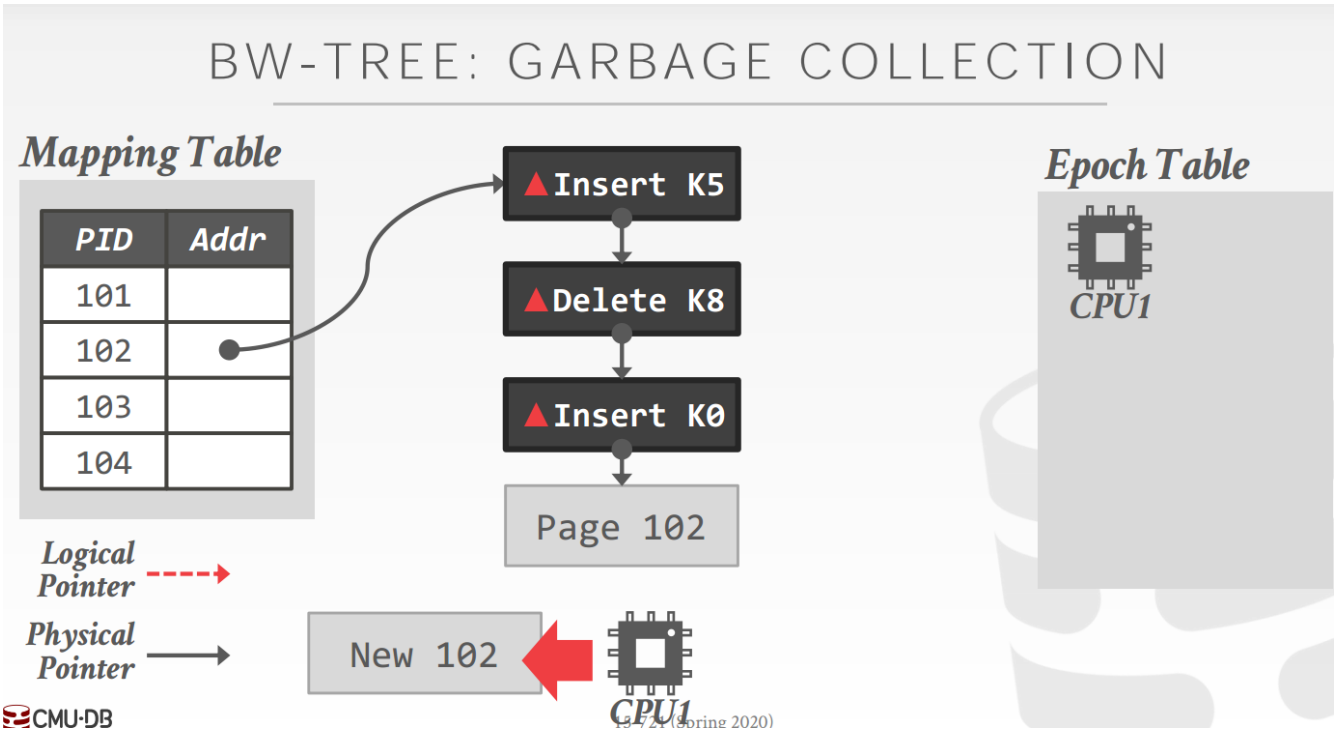
操作完成后，线程会退出这个epoch

一旦所有的线程注册到epoch E并完成然后退出这个epoch，回收epoch中的所有对象是安全的

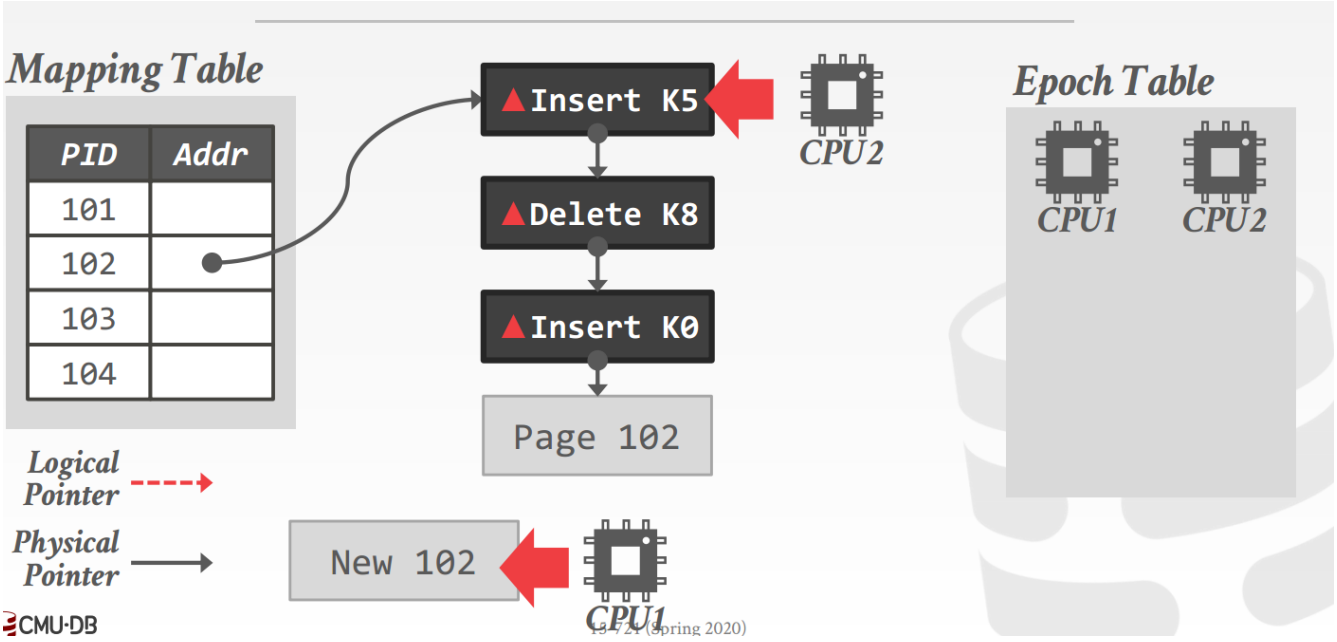
- Linux中这种trick叫做[Read-Copy-Update]

例子：

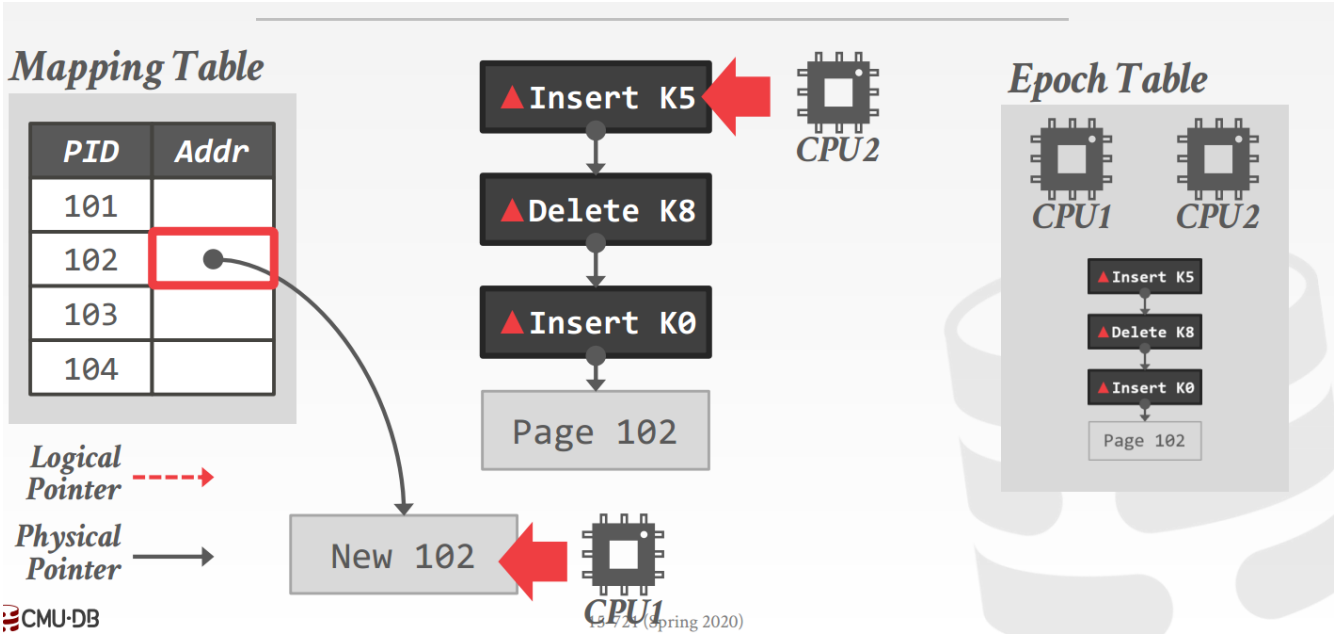
1.线程1对Page 102和所有delta record作consolidation,这个操作加入到epoch



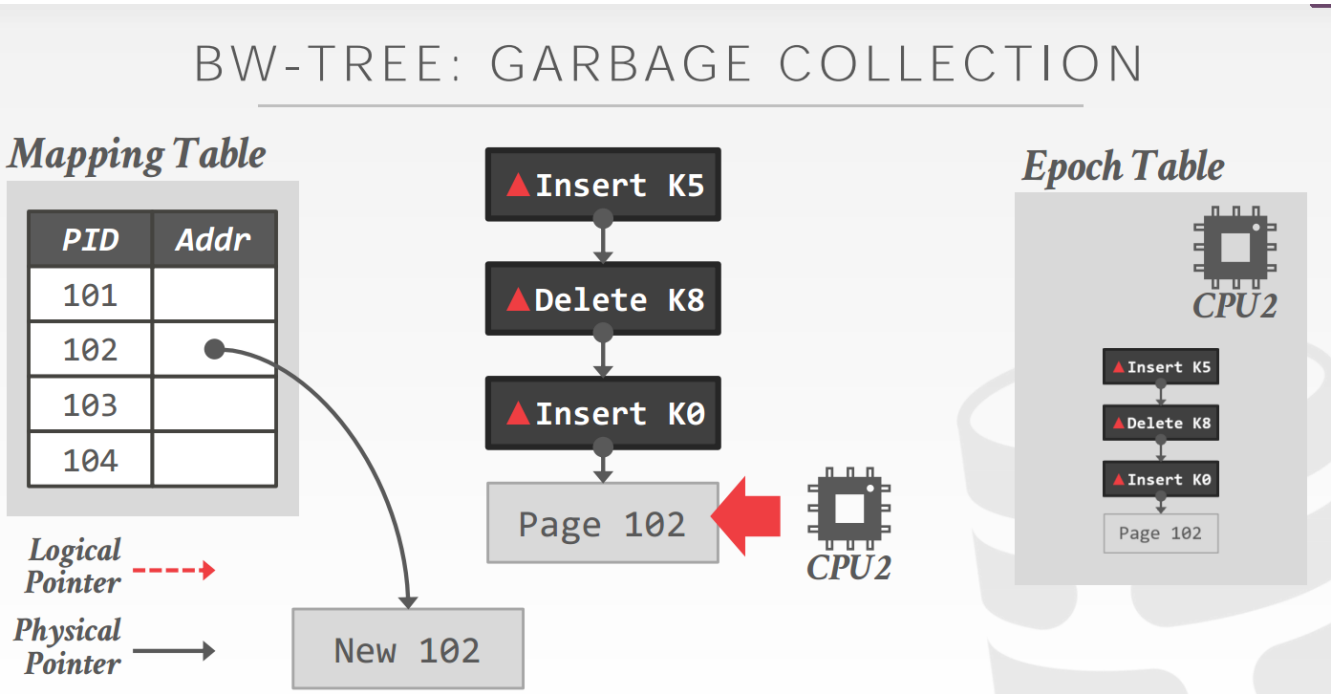
2.线程2开始访问P102,这时候线程1尚未进行CAS更新Addr



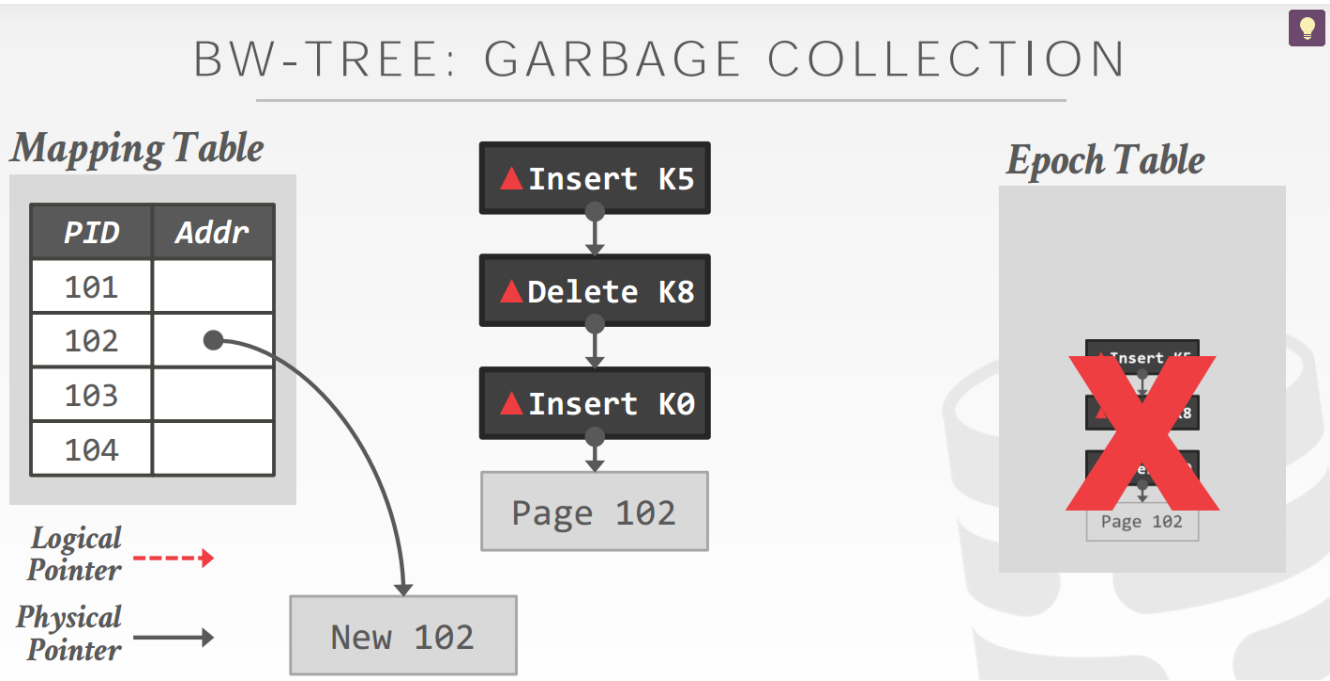
3.CAS更新完成，把consolidation的旧历史记录加入到epoch



4.由于线程2还在遍历旧的对象，这个时候不能回收掉。所以可以看出**epoch**是用来保证将要被回收可是还在使用的对象不被回收的一种机制。



5.线程2也结束，epoch中所有线程都已经结束。这时候可以安全地回收在epoch中的对象



ref

Comparison between B and B+ Tree

Bw-Tree阅读笔记