

Lec-07

Lock & Latch

DB中的Lock的作用对象是table,tuple等DB中的entities实现事务的隔离

Latch更像是低级的同步原语如信号量,spinLock,用于保证对临界区访问的线程安全,实现的是线程的隔离,在OS中并没有Latch的概念, DB中的Latch更偏向于OS中的Lock

Latch Implementation Goals

- 不要去造Latch的轮子
- Latch是没有线程的等待队列的
- Don't use spinLock in userspace, you can never perform better than what the kernel can do.

Latch Implementations

- Test-and-Set SpinLock

1.硬件支持,单条指令提供上锁解锁的原子操作

```
TSL  RX, LOCK
```

读取LOCK值->读取的值存入到寄存器RX->给LOCK设置非0的值

上面的三个步骤是不可拆分的原子操作,执行该指令的CPU会锁住内存总线,导致其他CPU不能访问内存。

顺便说说关中断和TSL的区别:

中断屏蔽只会影响当前的CPU,其他CPU依然能够访问内存,因此中断屏蔽只适用于单处理器

而想要别的处理器访问不到内存只能使用TSL

Java里面的自旋锁应该可以用Unsafe或者VarHandler的CAS实现

但一般不采用这种方法实现Latch,一是不能进入临界区的资源会不断自旋损耗CPU,二是可能会有cache invalidation就是缓存一致性的一些issue

- Blocking OS Mutex

类似于Linux内核中的Futex

右下角大概讲述了工作原理:

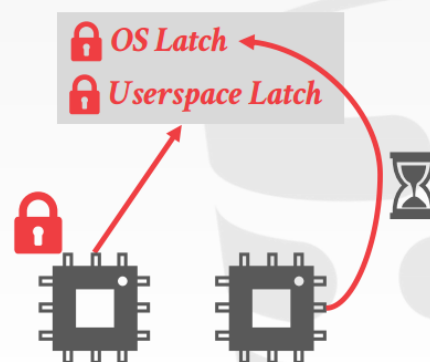
假设有两个CPU竞争,第一个会竞争到Userspace的Latch,如果失败了就会占有OS的Latch,然后一直等待等到另一个CPU释放Userspace的Latch。总所周知涉及到OS肯定会有syscall,这样性能也会很差好吧

LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: `std::mutex`

```
std::mutex m; → pthread_mutex_t
:
m.lock();      futex
// Do something special...
m.unlock();
```



- Adaptive SpinLock (Right thing to choose)

如上面不同, 如果线程竞争Userspace的latch失败了, 他们会暂时阻塞并且存储到一个全局变量"parking lot", 就好像到停车场泊车一样。然后如果又有别的线程进来了, 看到停车场"有车了", 即有别的线程已经在等待userspace的latch, 那么它就不会去那么傻再自旋而是直接也进入parking lot, 减少CPU损耗对吧。

- Queue-based SpinLock

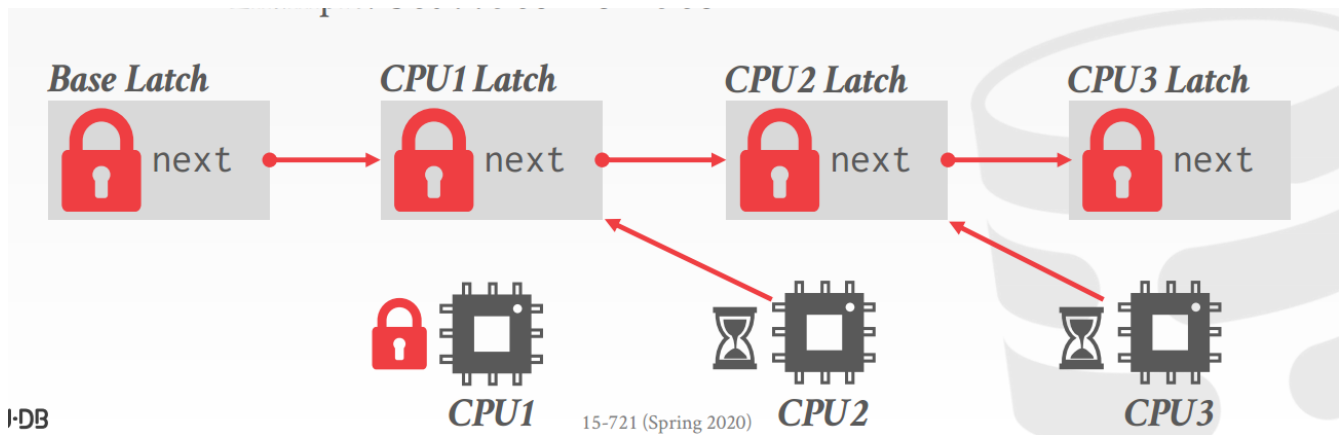
MCS spinLock in Linux Kernel这里简单介绍下

你想, 如果spinLock变量发生变化, 所有尝试获取这个spinLock的CPU都需要从内存读取然后刷新到自己对应的缓存行, 但最终只能有一个CPU可以获得锁, 只有它的刷新是有意义的, 锁的竞争越激烈, 这种无效的开销就会越多。

大概意思是说假设CPU1的线程占有Base Latch, 现在CPU2的线程进来了, 发现Base Latch已经被别的线程占有了。咋办? 如果CPU2也是在Base Latch自旋, 那么就会有上一段我讲的问题。这里的做法是CPU2会在Base Latch的分身CPU1 Latch自旋, 具体是维护一个Latch队列, 依次把新的线程对应的Latch加入到这个队列中。

当CPU1释放Base Latch之后, 接下来队首就是CPU1 Latch了对吧, 这个时候CPU2就可以占有这个Base Latch了... 以此类推。

说白了就是防止多个线程同时在一个memory space的latch做spinning, 避免因此而产生的缓存不一致所造成的性能损耗。



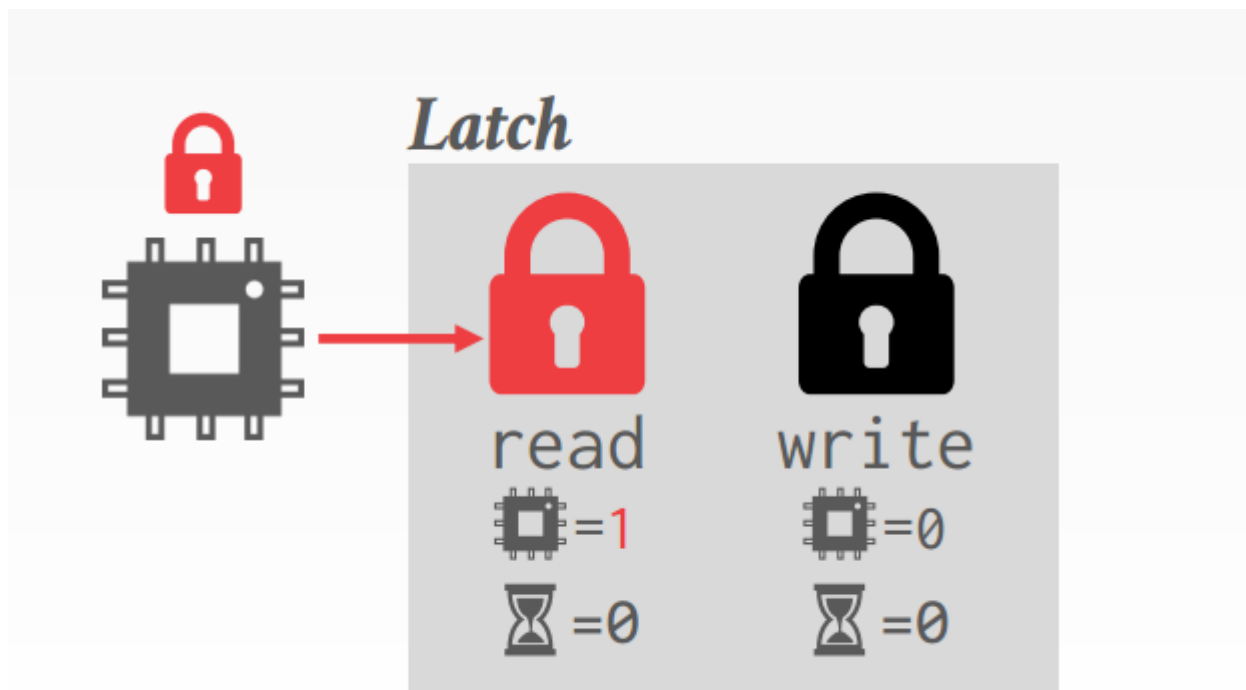
- Reader-Writer Lock

啊我记得阿里云面试就问过我自旋锁和读写锁性能的对比...分别在啥场景下适用，可惜当时答得不怎么样

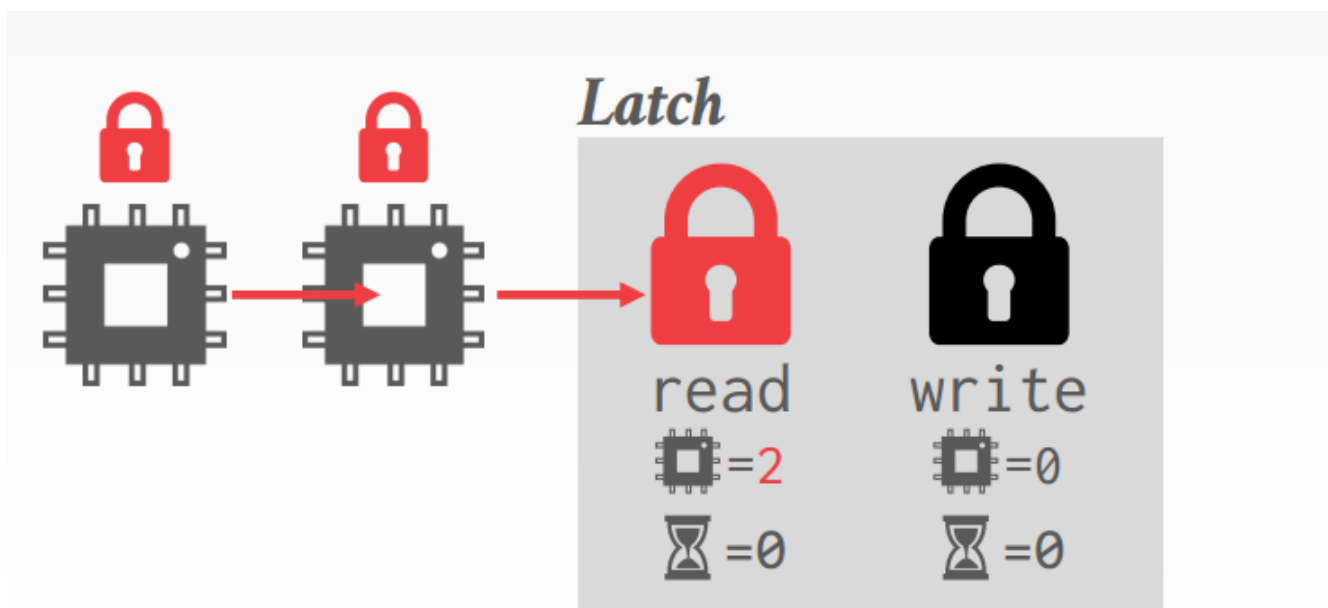
- 1.允许并发读
- 2.需要对读写线程分别维护线程队列以避免饥饿
- 3.可以在spinLock基础上实现

下面给出例子:

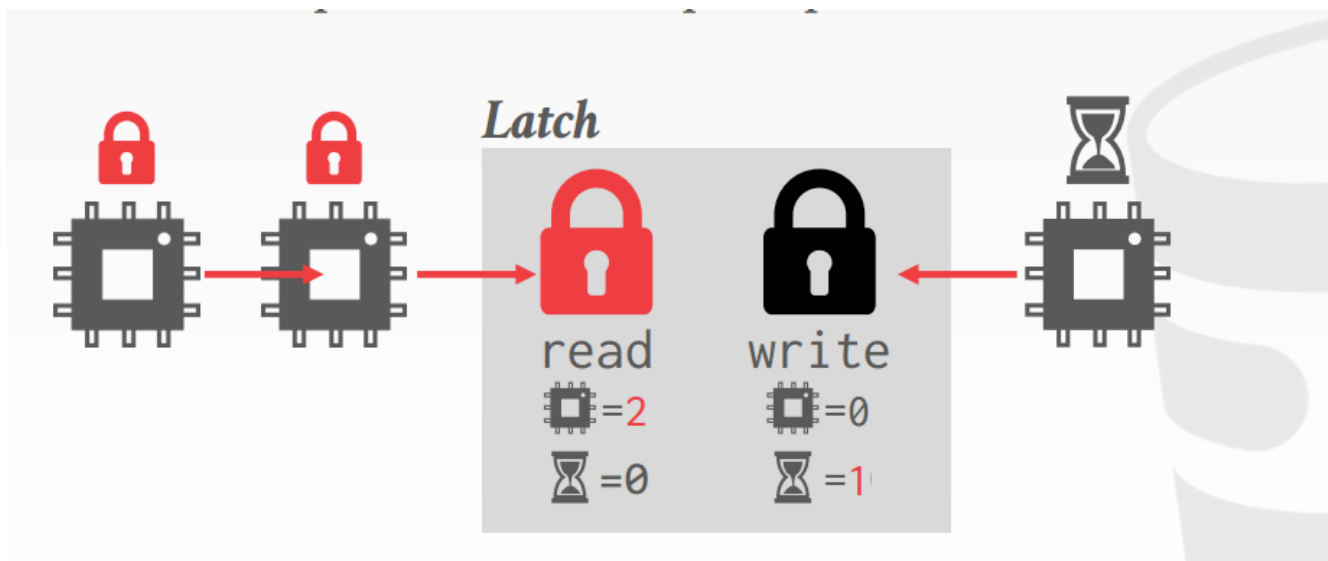
读写锁分别维护现在占有锁的线程数量和等待锁的线程数量



那么现在来了两个读线程，他们并不是互斥的，可以共享读锁



这时候呢来了个写线程，由于读锁现在有两个线程占有，写线程只能入队等待了



这时候呢有一条新的读线程进来了，因为这时候有写锁等待着，所以新的读线程只能入队等待

Latch Crabbing & Latch Coupling

- Acquire and release latches on B+ Tree nodes.

那啥时候可以释放掉结点的latch呢？

如果你的子节点可以被视为安全的，那么可以释放其父节点的latch。

那什么是安全啊？（战术后仰

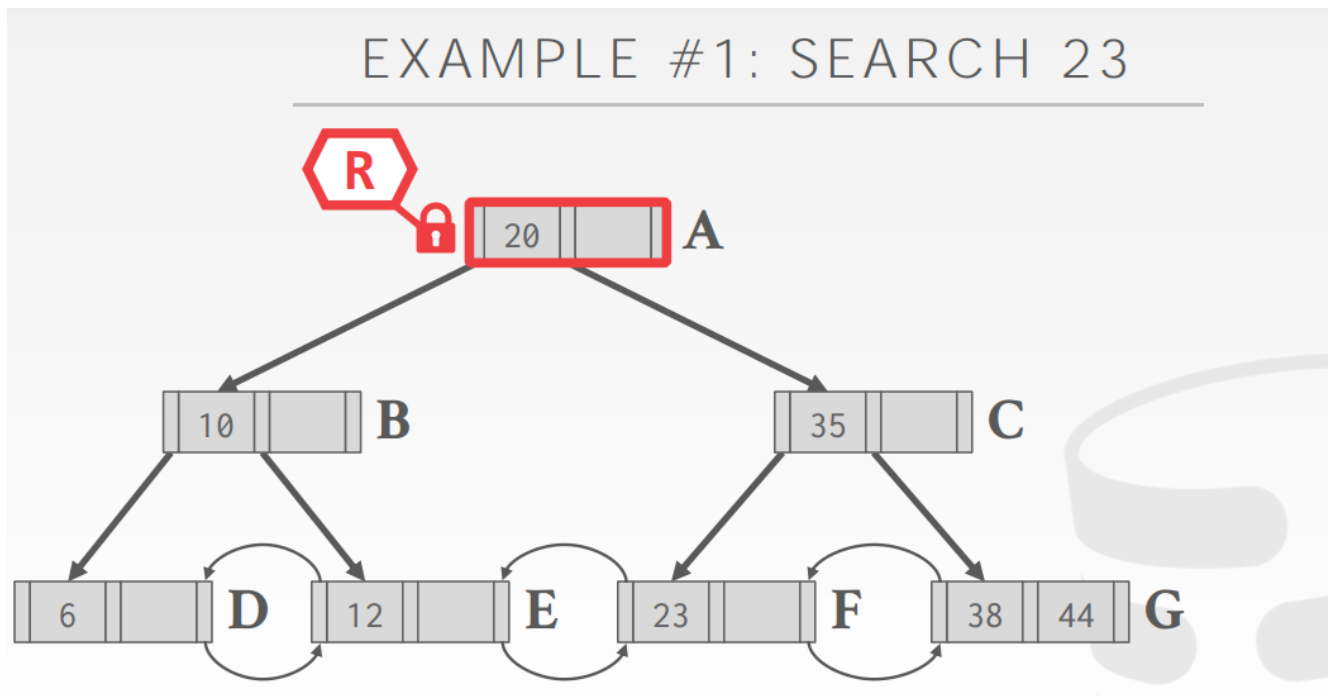
具体来说如果一个节点不会split或者merge就可以说是安全的

就例如insert时候如果结点不是满的，就是安全的。满了的话就会split

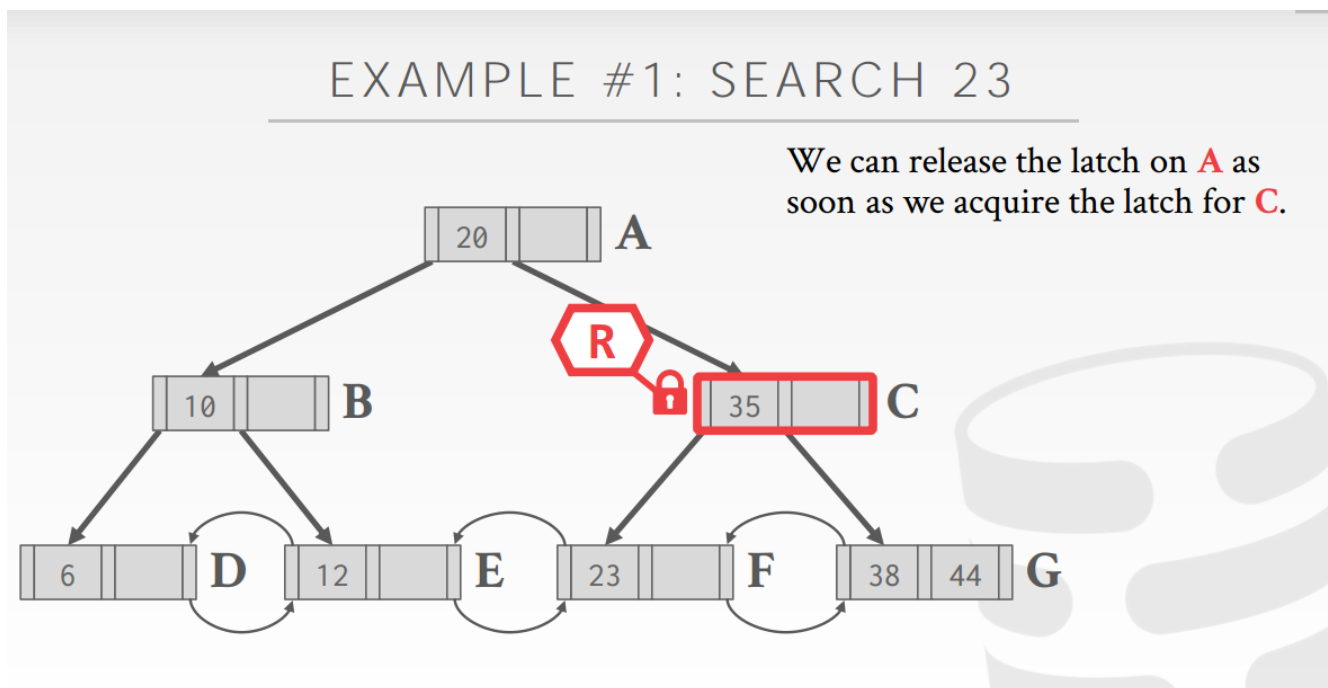
或者如果删除操作的时候，节点数量已经超过一半了，那也是安全的。否则如果没有一半的话，就会merge到其他结点

对于search操作，从根节点开始往下遍历

- 1.对子节点获取读latch
- 2.然后释放父节点的latch



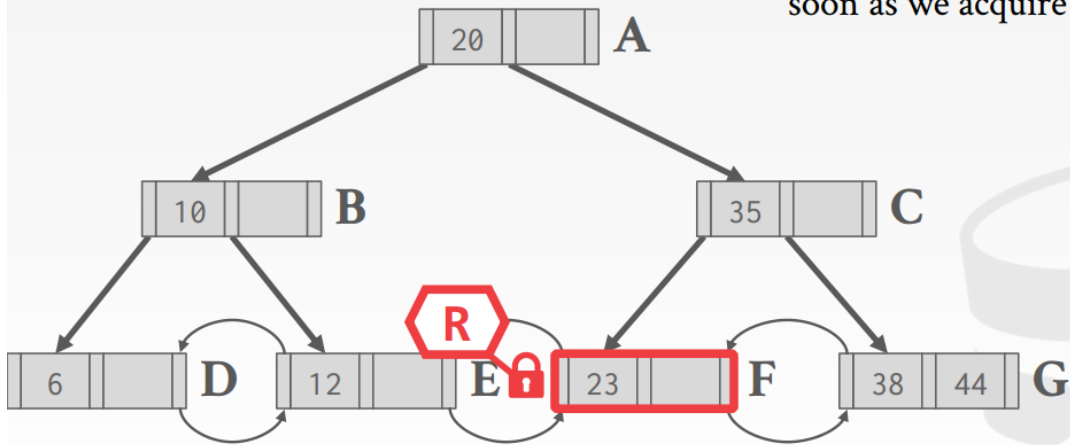
如下图，一旦获得了C的锁，就可以释放其父节点A的锁



以此类推最后找到F

EXAMPLE #1: SEARCH 23

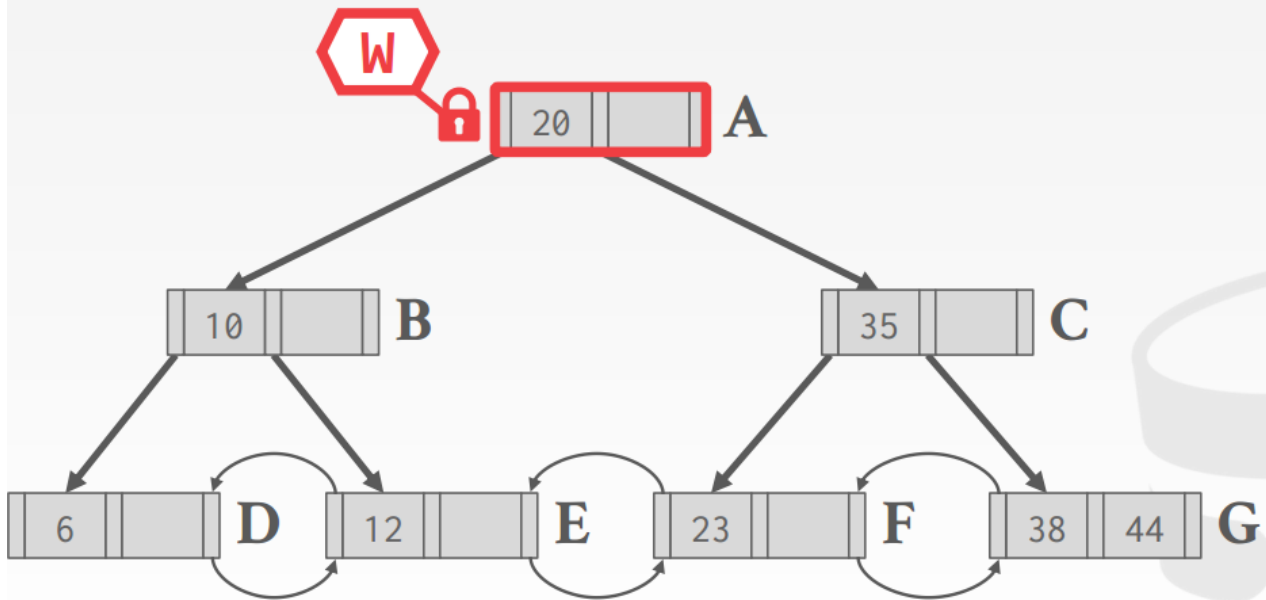
We can release the latch on **A** as soon as we acquire the latch for **C**.



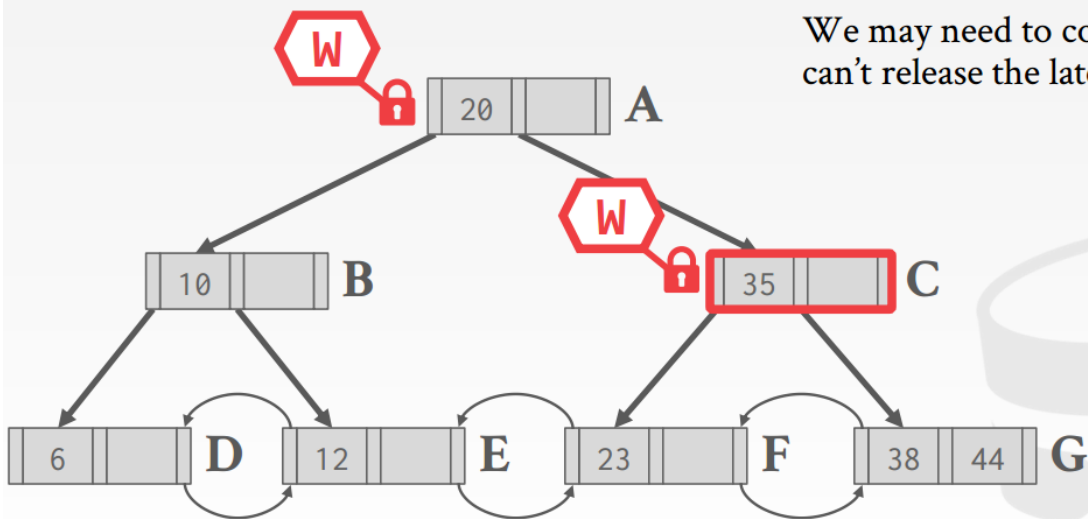
对于insert, delete操作, 从根节点往下遍历, 然后获取写latch, 一旦子节点上锁了, 就检查其是否安全。如果子节点安全, 释放其所有父节点的latch。

delete操作:

EXAMPLE #2: DELETE 44



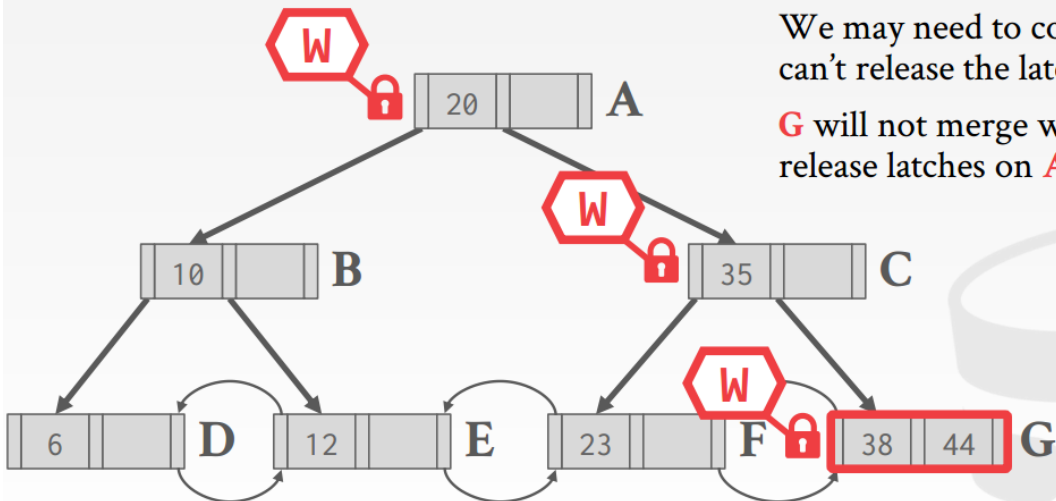
EXAMPLE #2: DELETE 44



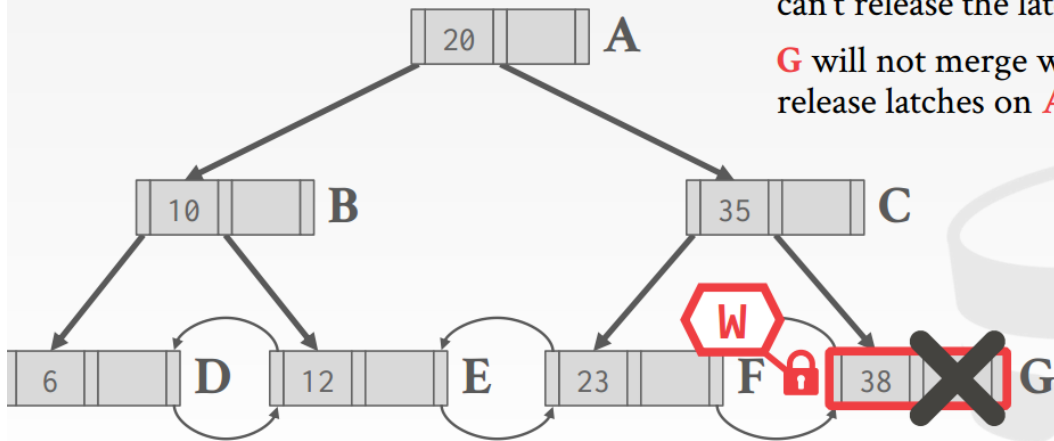
一直找到44之前A,C都不能解锁，因为不能判断C是否安全，C可能会合并到其他结点

然后判断G结点删除之后是否会与F结点merge，如果不会数目G是安全的，这时候才可以释放其所有父节点上的锁。

EXAMPLE #2: DELETE 44

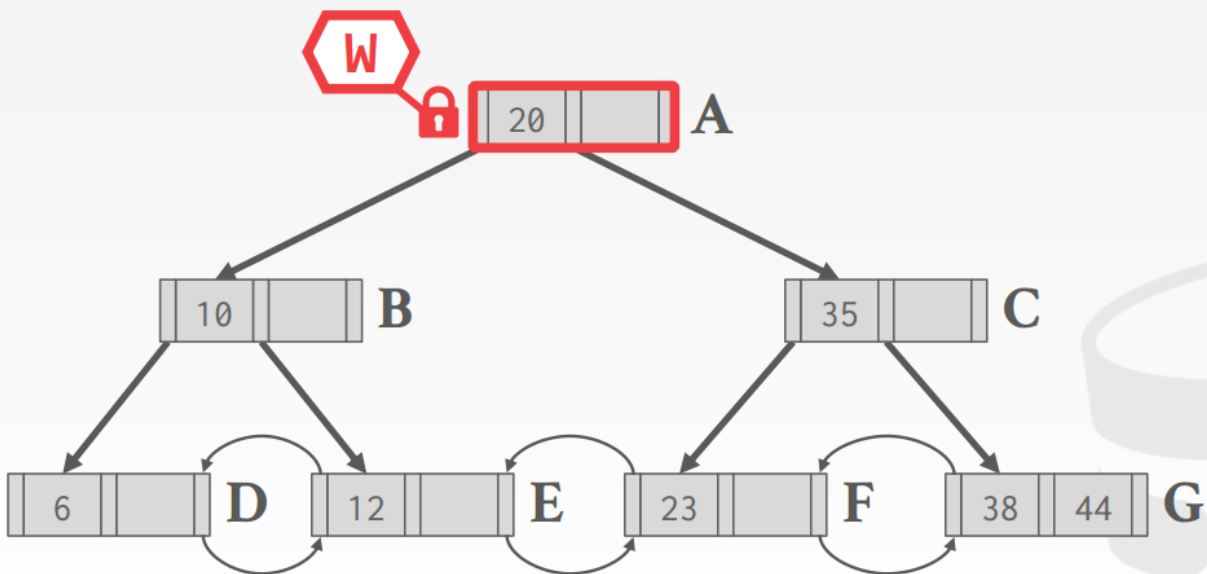


EXAMPLE #2: DELETE 44



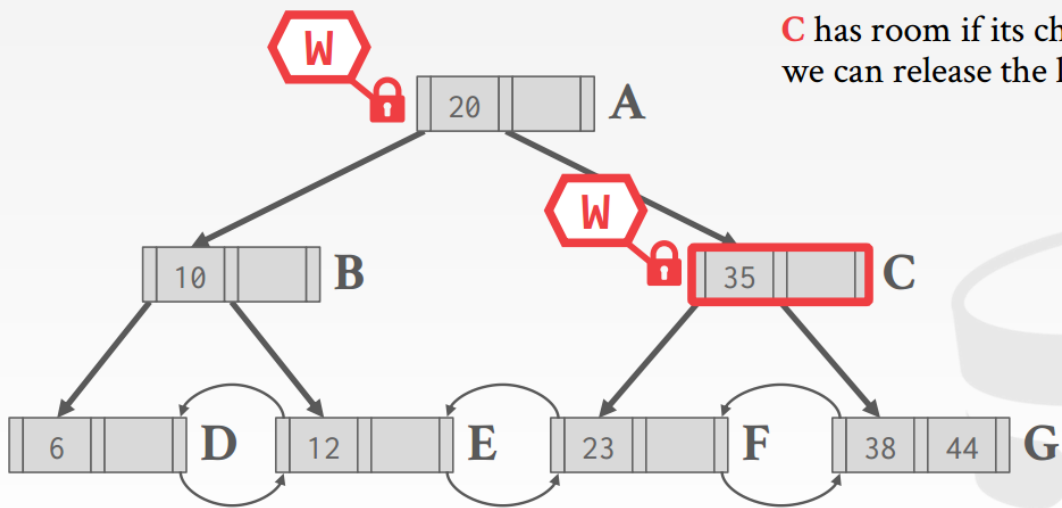
insert操作

EXAMPLE #3: INSERT 40



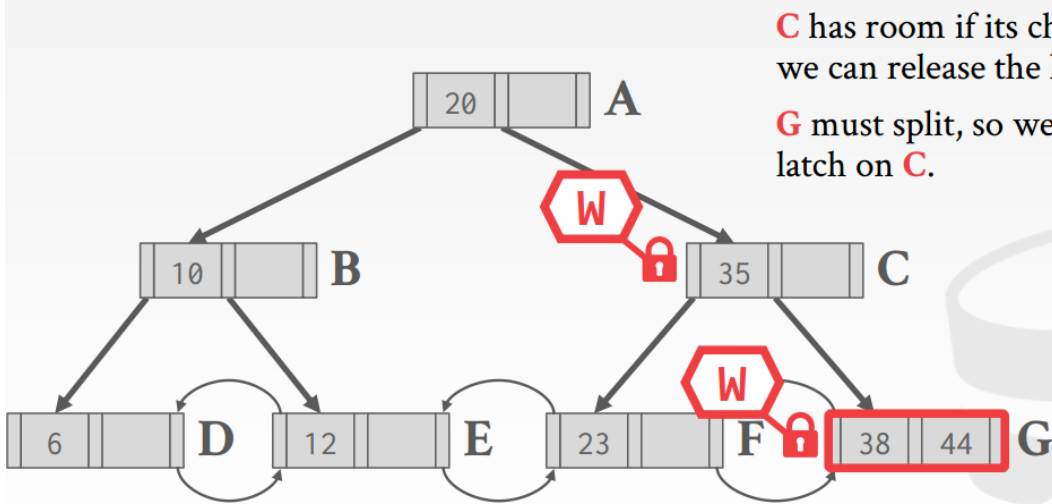
对于插入操作而言，重要的是判断C结点的子节点分裂的时候，C结点还有没有空间容纳。这里是有的，因此它是安全的，可以释放掉A的锁。

EXAMPLE #3: INSERT 40

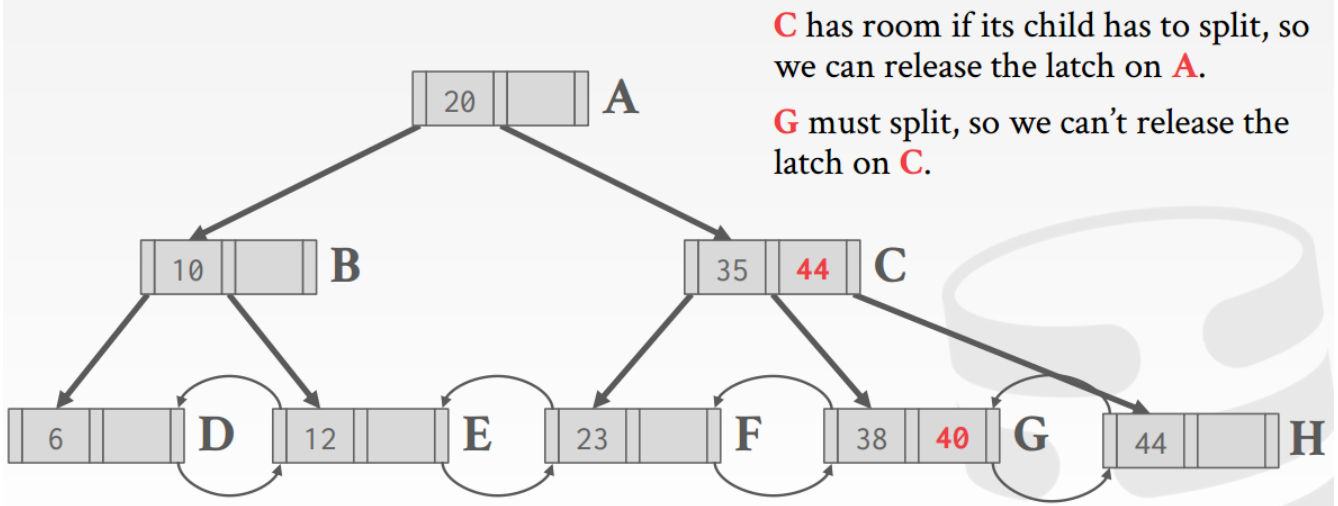


这时候插入40，G一定要split，G不安全，因此不能释放C的锁

EXAMPLE #3: INSERT 40



EXAMPLE #3: INSERT 40



但是上面这种方法效率有点低，因为ancestor node随时要上一个Write Latch,可能会block掉其他的读请求。所以我们可以采取以下方法：

乐观认为子节点是Safe的，上的是读锁，那么一旦子节点安全就可以释放掉父节点的读锁，避免block掉其他请求。如果不安全，就只好按照上面那种思路来，上写锁了。

BETTER LATCH CRABBING

The basic latch crabbing algorithm always takes a write latch on the root for any update.

→ This makes the index essentially single threaded.

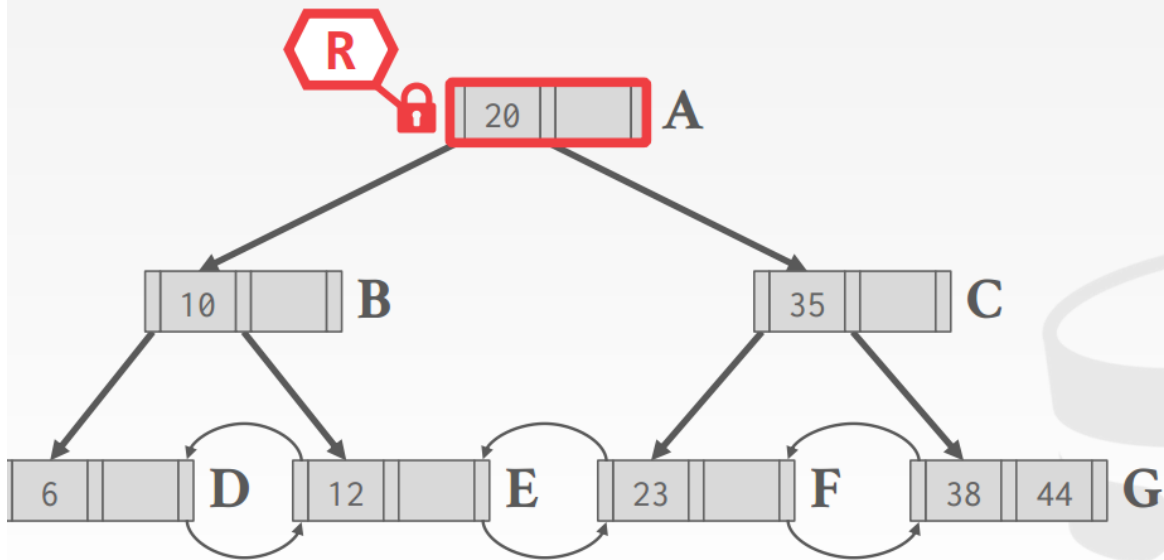
A better approach is to optimistically assume that the target leaf node is safe.

→ Take **R** latches as you traverse the tree to reach it and verify.

→ If leaf is not safe, then do previous algorithm.

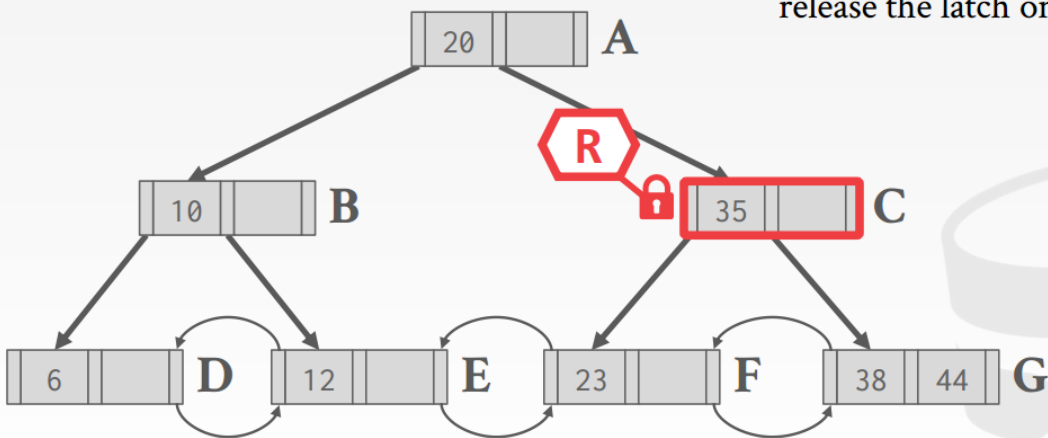
还是以delete 44为例子

EXAMPLE #4: DELETE 44



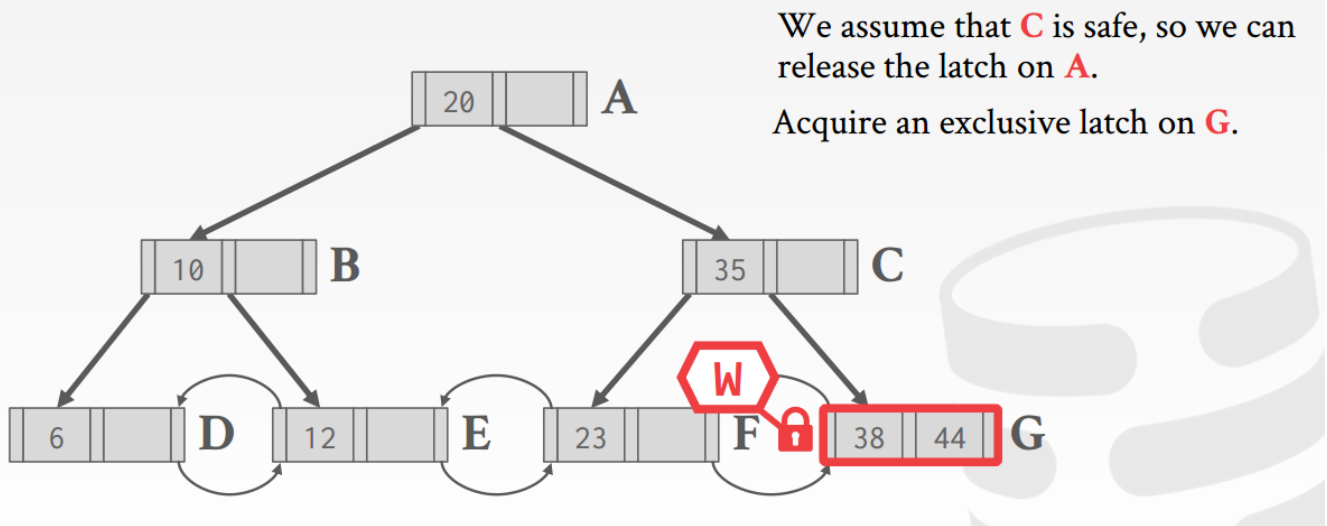
EXAMPLE #4: DELETE 44

We assume that **C** is safe, so we can release the latch on **A**.



因为delete44不会merge掉G，因此可以释放掉C的latch

EXAMPLE #4: DELETE 44



更好的方法：基于版本号机制：

VERSIONED LATCH COUPLING

Optimistic crabbng scheme where writers are not blocked on readers.

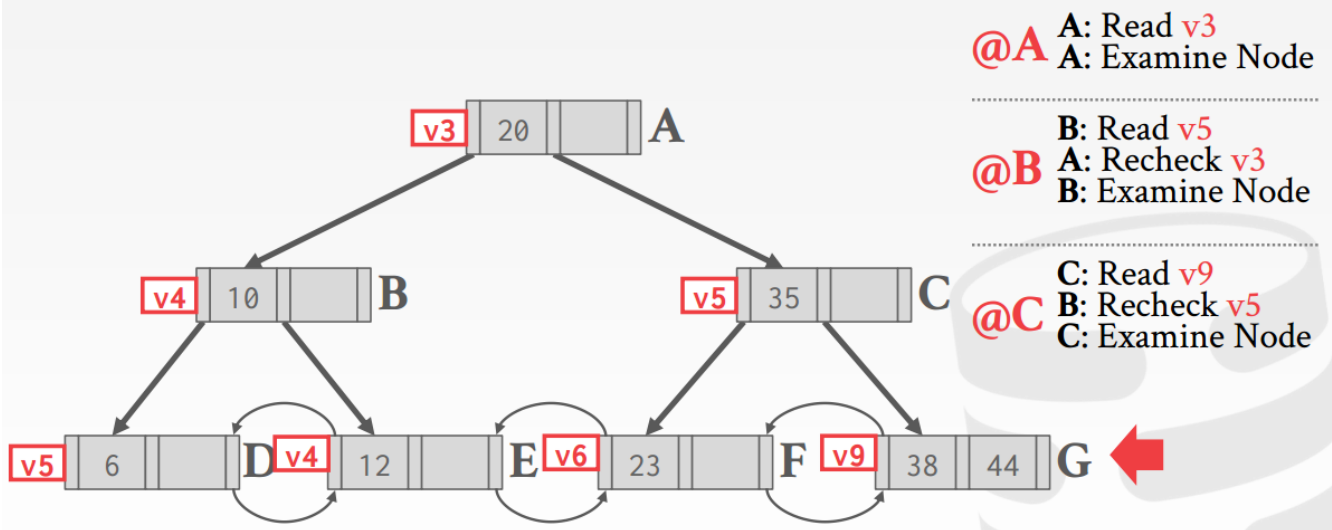
Every node now has a version number (counter).

- Writers increment counter when they acquire latch.
- Readers proceed if a node's latch is available but then do not acquire it.
- It then checks whether the latch's counter has changed from when it checked the latch.

Relies on epoch GC to ensure pointers are valid.

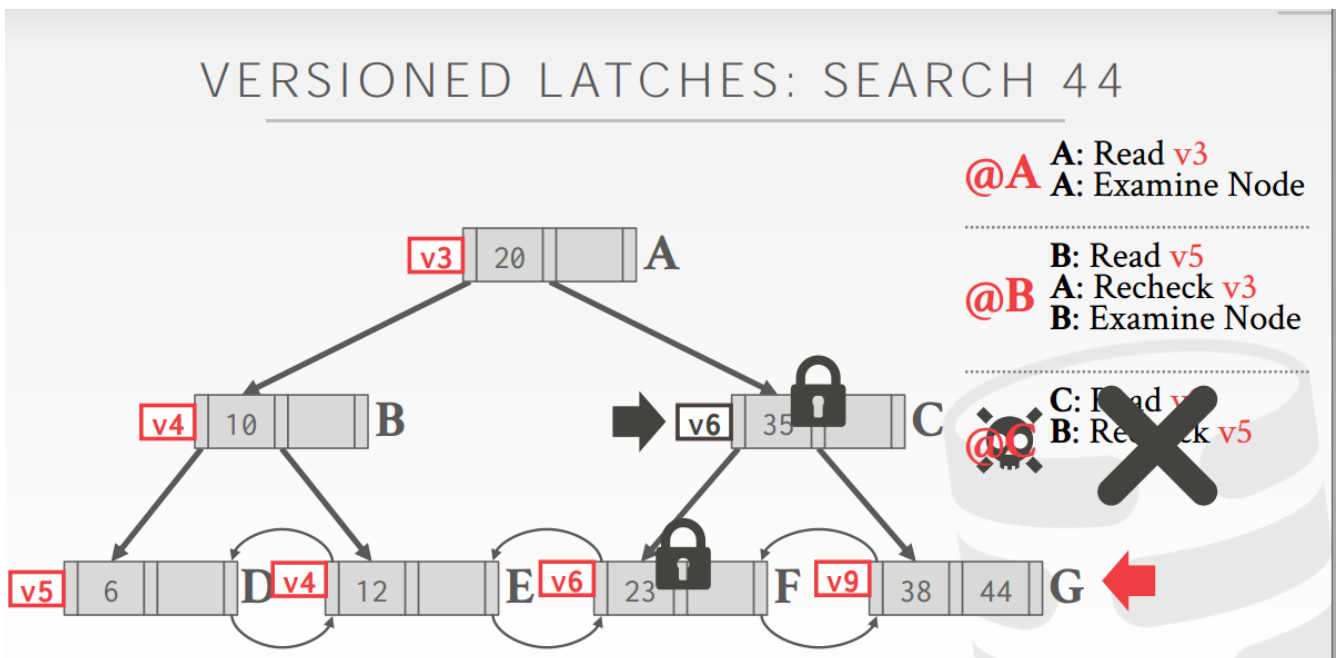
注意PPT中的@B应该是@C,@C应该是@G

VERSIONED LATCHES: SEARCH 44



每读一个Node，都会检查其父节点的版本号是否改变。如果改变了，有可能涉及到结点的split/merge导致结点路径改变。那么走原来的老路径可能就会search不到。

如果在读G结点的时候，有线程修改了C，C的版本号·发生了改变那么G的recheck就会失败，这时候就只能从头开始来了。



Trie

The inner node keys in a B+tree cannot tell you whether a key exists in the index. You always must traverse to the leaf node.

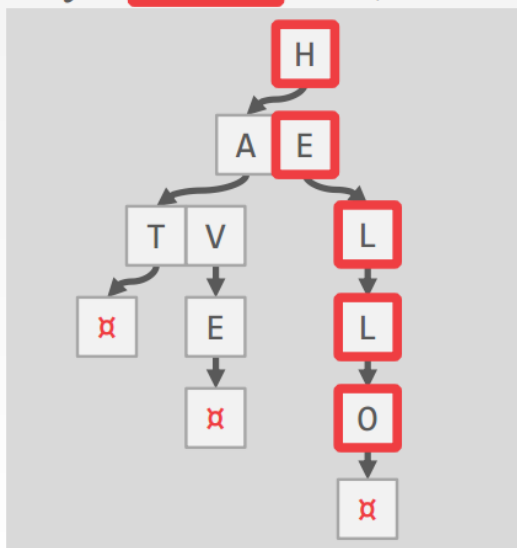
This means that you could have (at least) one cache miss per level in the tree.

B+树的数据都是在叶子节点，内部节点不能告诉你某个key是否存在。为了寻找某个数据，你可能需要从头遍历到叶子节点。那么每一层都其实会对应一个缓存行的失效。

于是我们引入字典树，方便做前缀匹配加速查找。

TRIE INDEX

Keys: **HELLO** HAT, HAVE



Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as *Digital Search Tree*, *Prefix Tree*.

那么search操作就没有必要都遍历到叶子节点才能找到某个key是否存在，在字典树中所有操作的时间复杂度为 $O(\text{length})$

TRIE INDEX PROPERTIES

Shape only depends on key space and lengths.

- Does not depend on existing keys or insertion order.
- Does not require rebalancing operations.

All operations have $O(k)$ complexity where k is the length of the key.

- The path to a leaf node represents the key of the leaf
- Keys are stored implicitly and can be reconstructed from paths.

TRIE KEY SPAN

The span of a trie level is the number of bits that each partial key / digit represents.

- If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

This determines the fan-out of each node and the physical height of the tree.

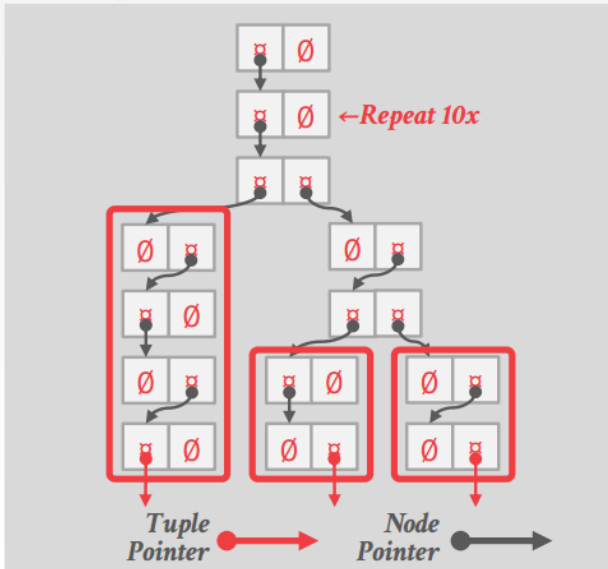
- n -way Trie = Fan-Out of n

一种表示方式如下图所示：

每个结点2项,如果该位是0则第一项指向下一个结点，如果是1就第二项指向下一个结点

TRIE KEY SPAN

1-bit Span Trie



Keys: K10, K25, K31

K10 → 00000000 00001010

K25 → 00000000 00011001

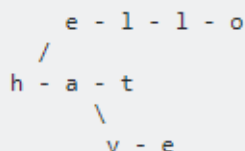
K31 → 00000000 00011111

Radix Tree

压缩版本的Trie

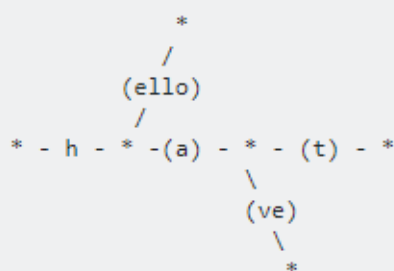
A radix tree is a compressed version of a trie. In a trie, on each edge you write a single letter, while in a PATRICIA tree (or radix tree) you store whole words.

Now, assume you have the words `hello`, `hat` and `have`. To store them in a *trie*, it would look like:



And you need nine nodes. I have placed the letters in the nodes, but in fact they label the edges.

In a radix tree, you will have:



and you need only five nodes. In the picture above nodes are the asterisks.

So, overall, a radix tree takes *less memory*, but it is harder to implement. Otherwise the use case of both is pretty much the same.

ref

<https://zhuanlan.zhihu.com/p/89058726>