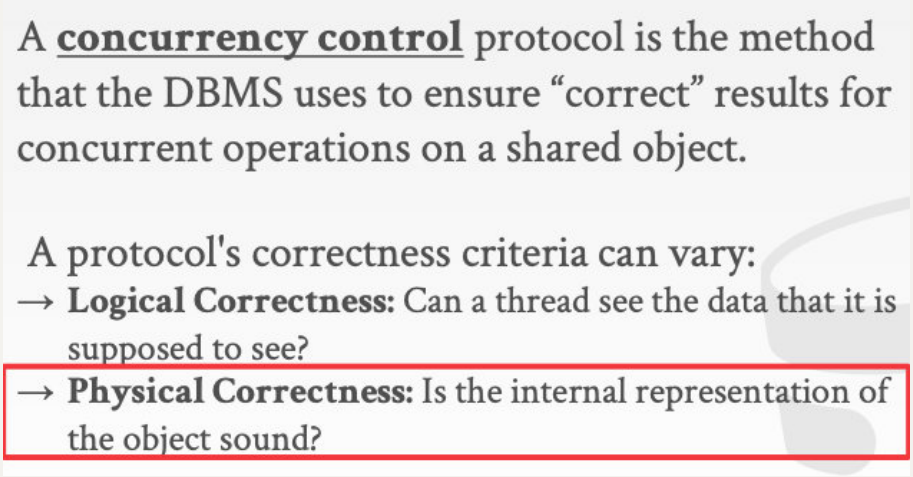
在大多数DBMS的实际使用场景中，会有多个线程在向哈希索引或B+树索引里插入数据或修改其中的数据，或者是读数据（也有少部分DBMS是只允许单线程的，比如说旧版本的Redis，诸多用户的增删改查只能串行执行），因此需要一套策略能够使得对索引的各种操作是thread-safe的

本Lecture所介绍的主要是并发控制协议中的Physical Correctness的实现策略，也就是让多线程共享的对象在被并发访问时，其内部的数据结构能够保持稳定



Logical Correctness会在后面介绍事务并发时提到

Latches Overview

再次回顾一下之前提到过的Latch/Mutex与Lock在概念上的区别，Lock经常指代逻辑上的宏观的锁，用于保护数据库里面一些逻辑内容不被其他的线程/事务所修改，比如说锁上表里的某一行数据，让别的事务不去改它，这就叫Lock，Lock一般是被事务所持有的，而且Lock还有一个特性，它在需要回滚的时候可以回滚，比如说我们先锁上某一行数据，然后修改了它，之后如果想回到修改前的状态就可以回滚

Latch一般用来保护DBMS内部的具体数据结构，比如说想锁数据库里的某一行数据，DBMS可能会给对应的tuple所在的B+树的节点上一个Latch，Latch被具体的操作过程所持有，比如说往B+树中插入一个数据，这个操作过程中间可能会持有很多的Latch，最后还会一步步地把这些Latch释放掉，Latch往往不是整个事务过程当中所持有的，可能是在一个操作，比如某次的插入操作中持有的，插入完成后便不再持有，Latch也不需要考虑事务回滚的问题

Locks

→ Protect the database's logical contents from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

Latches

→ Protect the critical sections of the DBMS's internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.

本Lecture主要探讨Latch相关的话题

Latch有读模式与写模式这两种模式，在读模式下又被称为“读锁”，在写模式下又被称为“写锁”，读锁存在的意义是，当某个线程要读某个数据时，需要读锁不让别的线程写这个数据；写锁存在的意义是，当某个线程要修改某个数据时，需要写锁确保别的线程既不能读也不能写这个数据，将这个数据完全地锁住

锁之间还有兼容的问题，很多类型的锁都有读锁和写锁两种模式，它们之间的兼容性问题如下图右侧2x2矩阵所示，只有两个读锁的话，它们两个是可以兼容的，一个线程给一个对象上了一个读锁，别的线程可以过来给这个对象再上一个读锁，即多个线程可以给一同个对象上读锁，但只要这个对象被上了读锁，别的线程只能给它再上读锁，不能再上写锁；如果一个对象被上了写锁，那么其他的线程不能对这个对象做任何的操作，加读锁/写锁都不行，因为上了写锁就表示这个对象被独占了

LATCH MODES

Read Mode

→ Multiple threads can read the same object at the same time.
→ A thread can acquire the read latch if another thread has it in read mode.

Write Mode

→ Only one thread can access the object.
→ A thread cannot acquire a write latch if another thread has it in any mode.

Compatibility Matrix

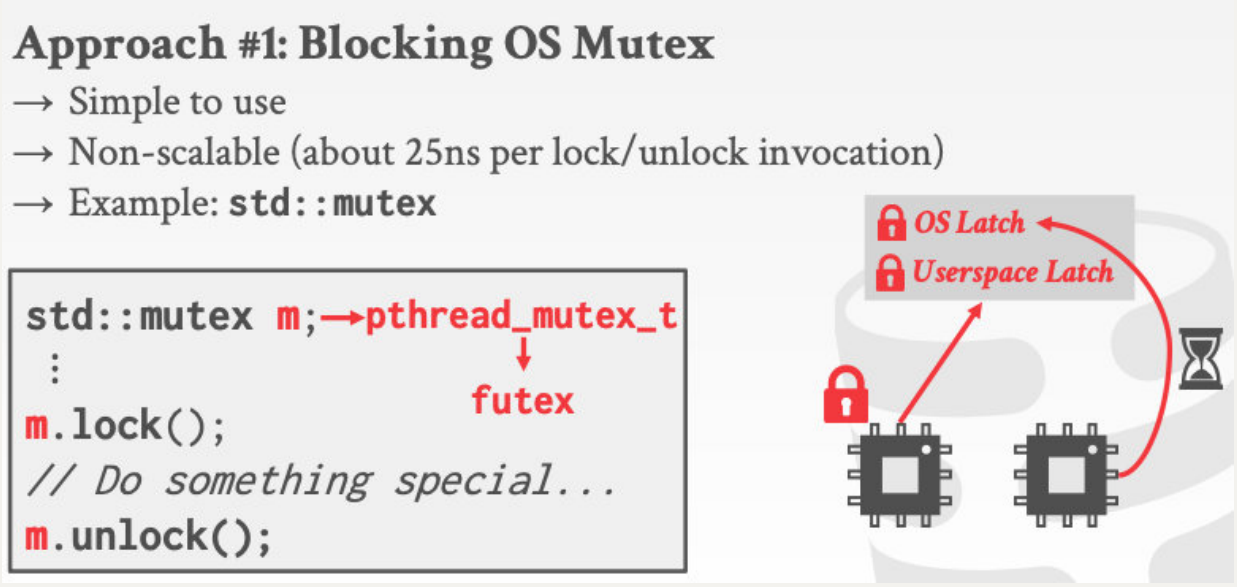
	Read	Write
Read	✔	X
Write	X	X

有的时候读锁也叫共享锁，因为加了读锁后仍然可以和其他线程共享对应的对象或变量，只是其他线程不能修改它；写锁也叫独占锁，加了写锁之后相关的对象或变量就被当前线程所独占了

Latch有如下的实现方式：

- Blocking OS Mutex

  翻译成中文就是，“阻塞式的操作系统互斥锁”，它比较容易使用，因为这是操作系统原生支持的，但不能应对大规模并发竞争的场面，比如说C++的`std::mutex`就属于这种锁，具体的实现原理和xv6的`sleeplock`差不多：用最基本的自旋锁保护一个条件变量，获取该锁时检查这个条件变量，如果满足条件则修改条件变量从而获得该互斥锁，否则进入睡眠状态，直到该互斥锁被释放时条件变量被修改回来，释放互斥锁的线程唤醒已经睡眠的线程（详情参考xv6源码）

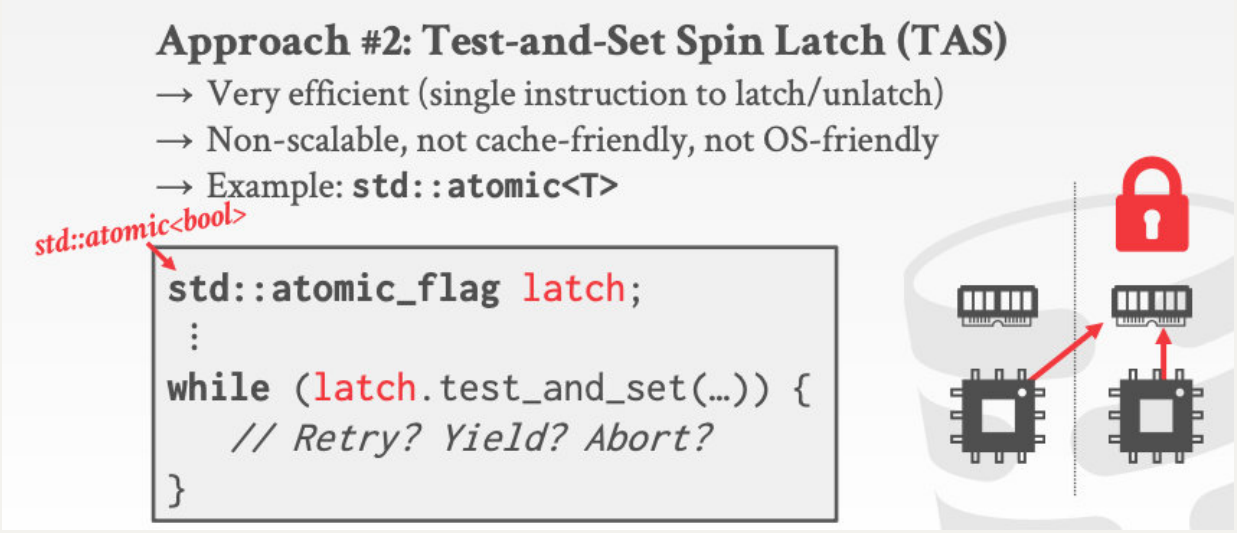
  

  互斥锁和自旋锁相比，争用锁的时候，得不到锁的线程不会一直自旋，浪费CPU资源，而是会进入睡眠，但是相应的睡眠与后续的唤醒操作由于会修改线程的状态，因此也有一定的开销

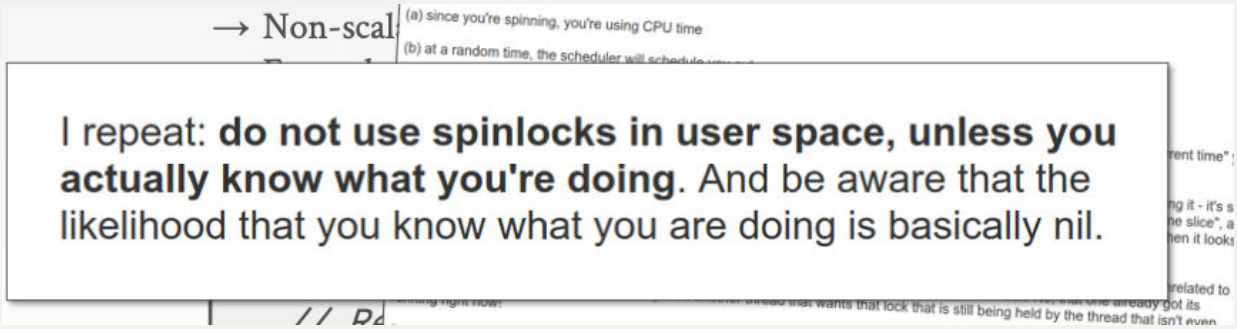
- Test-and-Set Spin Latch

  这个简单来说，就是自旋锁，好处是可以通过一个指令就能获取/释放锁，实现简单，但也不能应对大规模的并发竞争，而且对缓存和操作系统不友好

  实现方式和xv6的`spinlock`差不多

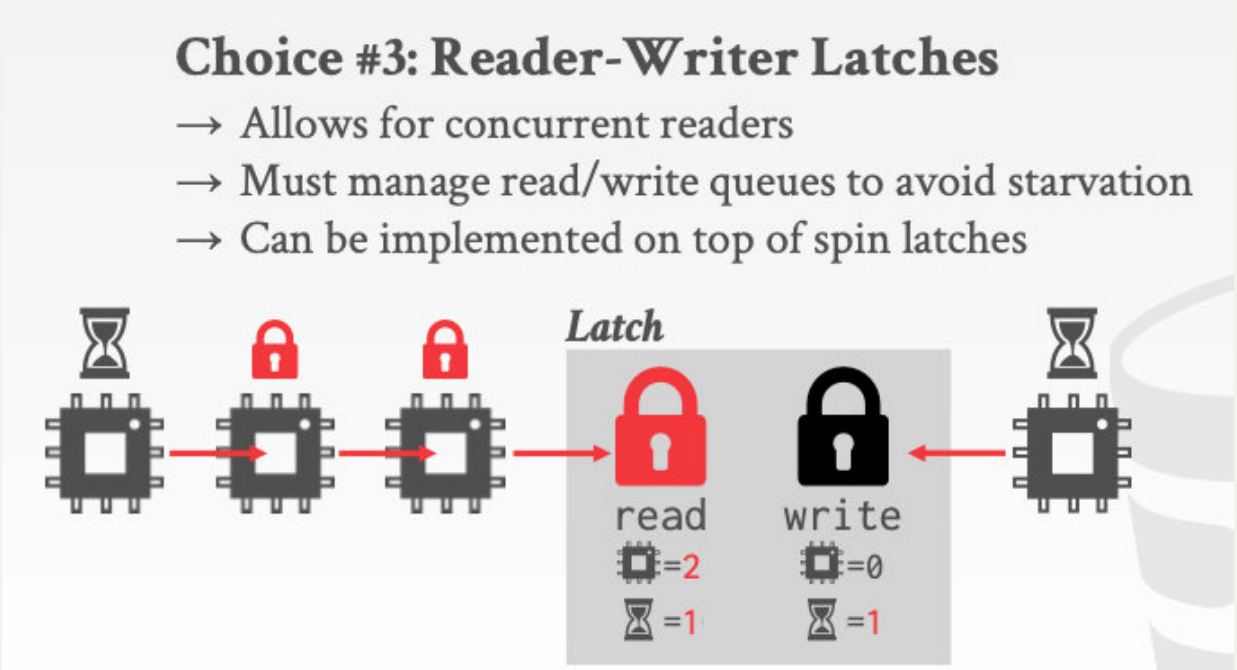


Linus曾说过，在用户态不要使用自旋锁，因为占用CPU自旋是效率极低的操作



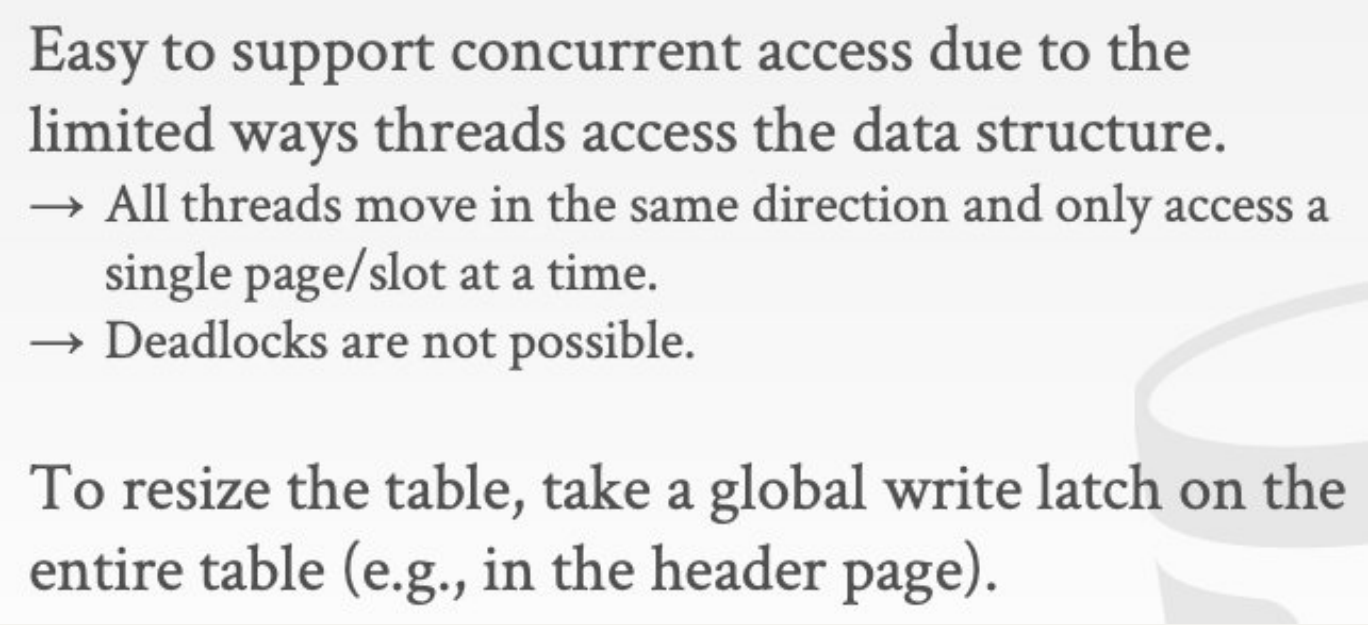
- Reader-Writer Latches

其实就是前面一开始提到的读写锁，读写锁底层是基于自旋锁来实现的，并且设有等待队列，为了避免想要获取写锁的线程等待太久（即避免starvation），会限制在同一时刻能够获取写锁的线程的数量：自从有线程开始排队等待获取写锁之后，后面想要获取读锁的线程都要开始等待，直到想获取写锁的线程成功获取写锁并完成相关操作，释放写锁为止

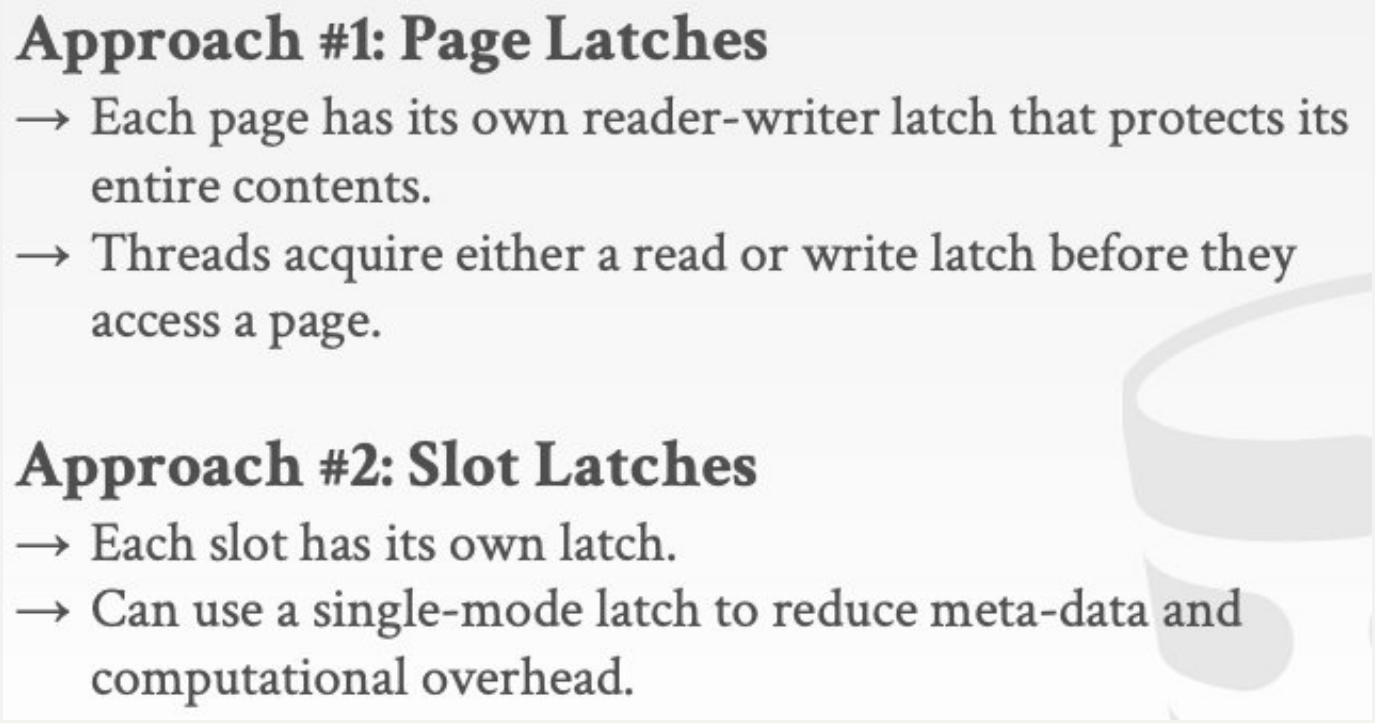


Hash Table Latching

接下来介绍哈希表的并发控制，哈希表这种数据结构比较好加锁，拿开放地址哈希策略举例，任何一个进行索引的线程都是在哈希槽数组中从上往下地查找，因此都是从上往下地获取每个哈希槽/哈希表的段的锁（这会在下面细讲），不像B+树会发生死锁，当我们对调整哈希表大小时（比如说对哈希表进行扩容操作），会对整个哈希表加一个全局的写锁，因为在扩容的过程当中里面的数据结构全是乱的，需要重新整理，因此这个过程中任何其他线程都不能读写哈希表



当我们不进行扩容操作时，可以加局部的锁，局部的锁的粒度有如下两种情况：

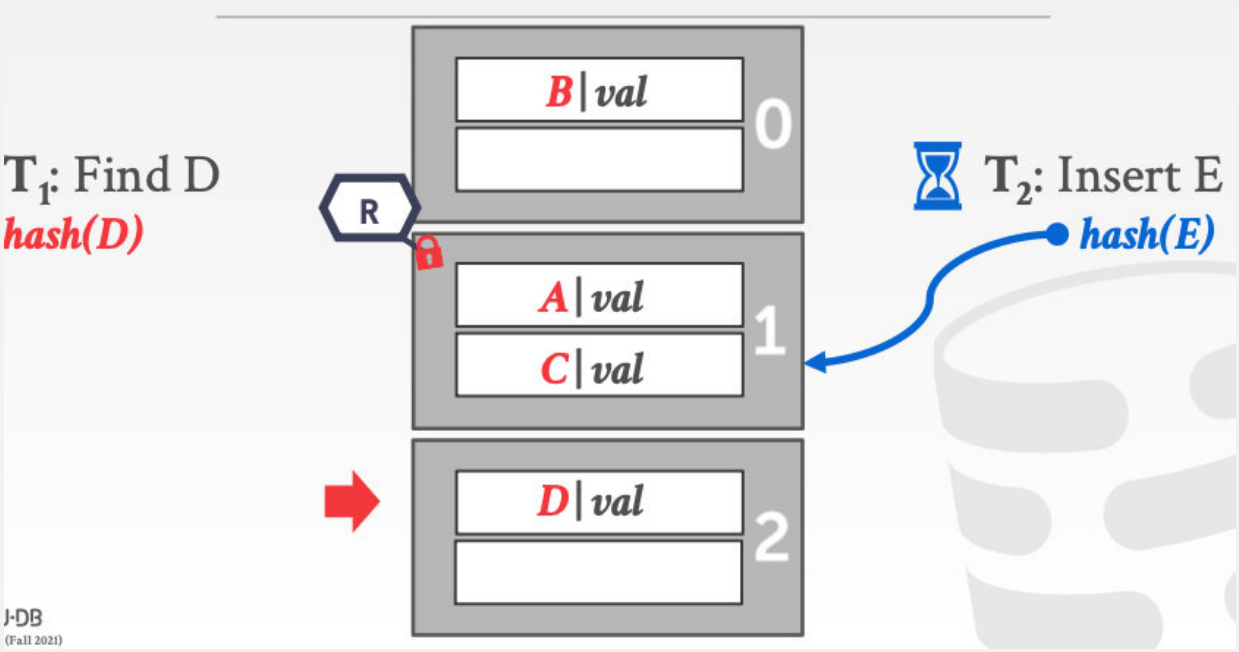


- 按Page的粒度来划分

  哈希表会被切成一个个的页被存到磁盘里，我们给每个页分配锁

  如下所示，假设还是开放地址哈希，线程T1先通过哈希函数得到理想的哈希槽号，获取这个哈希槽所在的页的读锁，之后开始查找想要的KV，此时另外一个并发的线程T2想往这个页里面插一条新的KV，它要先尝试获取这个页的写锁，因为T1持有该页的读锁，所以T2暂时无法获取该页的写锁，开始等待，此时线程T1按照前面所说，继续向下遍历查找想要的KV，结果并没有在当前页（图中的1号页）里找到，于是把1号页的读锁释放，之后获取2号页的读锁，在2号页中继续向下查找。由于线程T1释放了1号页的读锁，所以线程T2便可以获取1号页的写锁，之后在1号页里给要插入的KV找个地方插入，这时线程T1在2号页中找到了它想要找的KV，随即释放2号页的读锁；与此同时，由于哈希冲突，线程T2没能在1号页里面给即将插入的KV找到空位，于是在在离开1号页时释放1号页的写锁，并在线程T1释放2号页的读锁后获取了2号页的写锁，然后在2号页写入新的KV，最终释放写锁

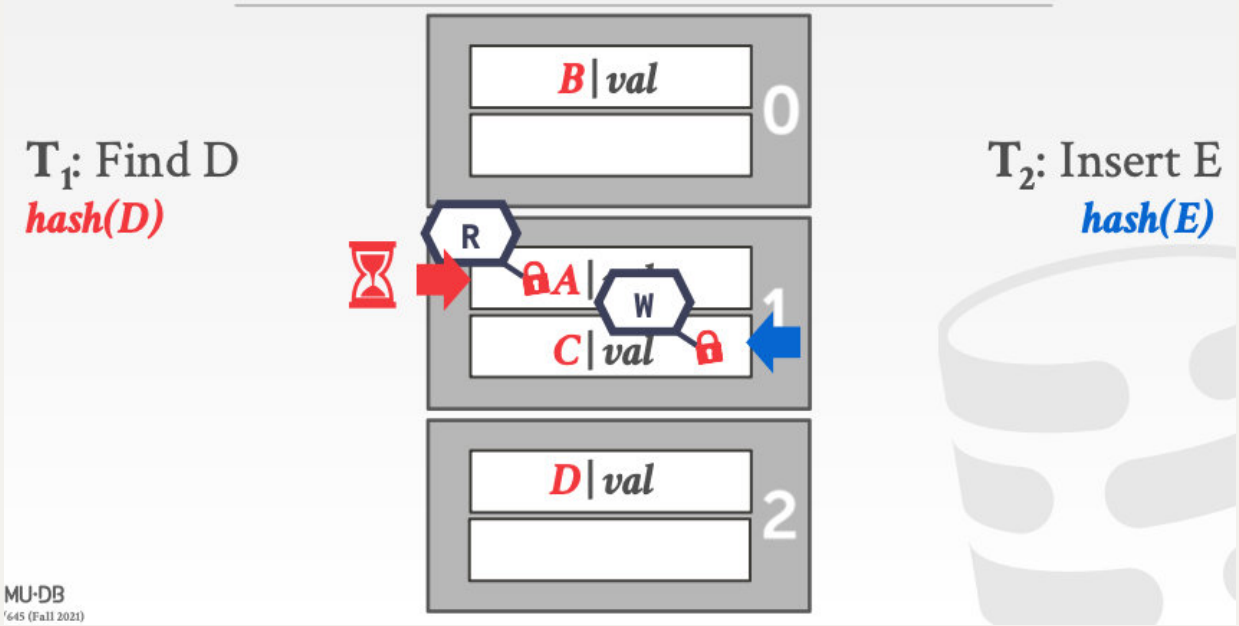
  （过程比较复杂，还需参考完整的slides）



Java中著名的concurrent hashmap也是基于上述思想，将哈希表分段，一个线程在操作某一段时，会获取这一段的锁，完成操作后再将锁释放，这时其他的线程才能获取锁并开始它们的操作。锁的粒度不是很细，一段一个锁，因此也不用维护太多的锁，并且保证了一定的并发性，每个线程任何时刻只会持有某一段的锁，其他的段在此时可以被其他线程读写

- 按Slot的粒度来划分

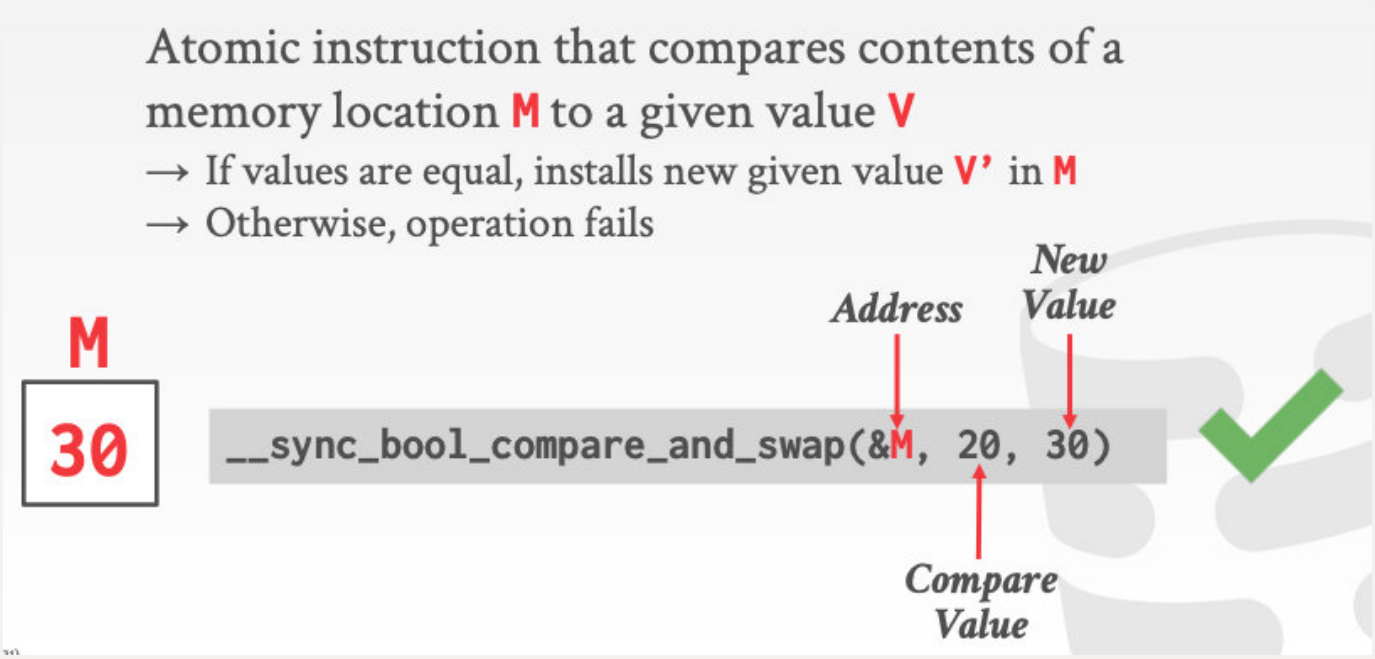
  这就是给每个哈希槽配一个锁，还是上面的例子，如下图，T1一上来想找D，直接获取哈希函数计算出的对应的槽的读锁，发现由于哈希冲突，这个槽里不是它想要的之后，便会释放读锁，往下寻找，同理，T2也会以槽为粒度获取/释放写锁，然后它们一边遵循着读写锁的规则，一边分别向下完成各自要进行的查找（具体过程复杂，还需参考slides）



以哈希槽为粒度，和以页为粒度比，粒度明显变细了，在某个线程持有某个槽的锁时，不影响其他的线程获取剩下的所有的槽的锁，因而并行性也会增加，但是也有缺点，这种策略下要维护更多的锁，尤其是每哈希个槽一个锁这么精细的粒度，大多数的应用场景承受不了，在工业界，前面的以页/段为粒度分配锁的策略更常见

除了这两种策略，还有其他的策略也可以实现哈希表的索引并发控制，比如说，读写分离，有一个只用于读的主哈希表，当线程想要读哈希表时，可以直接无锁地去读它，还有一个用于写的哈希表，在线程对哈希表进行写/读操作时需要获取/释放相应的锁，然后定期地把用于写的哈希表里面新写入的内容merge到只读的主哈希表中，Go语言的`sync.map`就是这样实现的，这样的实现的好处是读哈希表的时候几乎是无锁的

此外还有CAS（Compare and Swap）技术使得哈希表可以无锁地进行插入，很多原子操作的底层实现也都是CAS，如下所示，`&M`是我们要操作的变量的地址，我们要将M加10，把M的值从原本的20改成30，因此就有了如下对CAS语句的调用，这相当于把接下来的一系列操作整个打包给CPU和操作系统，告诉操作系统，先看看M是不是20，如果是20的话，把它锁住，然后把它改成30，也就是：先比较，再交换，这整个过程都是原子的。使用这个方法的话就可以避免两个线程同时对M进行加10而出现经典的临界区并发问题，如果两个线程都先读了M的初值为20，之后它们都会调用CAS语句对M加10，由于CAS语句是原子的，先执行CAS语句的线程会完成CAS操作，M的值变成了30，另一个线程在随后执行CAS操作时，会发现M不是20，进而返回false，表明原子操作失败

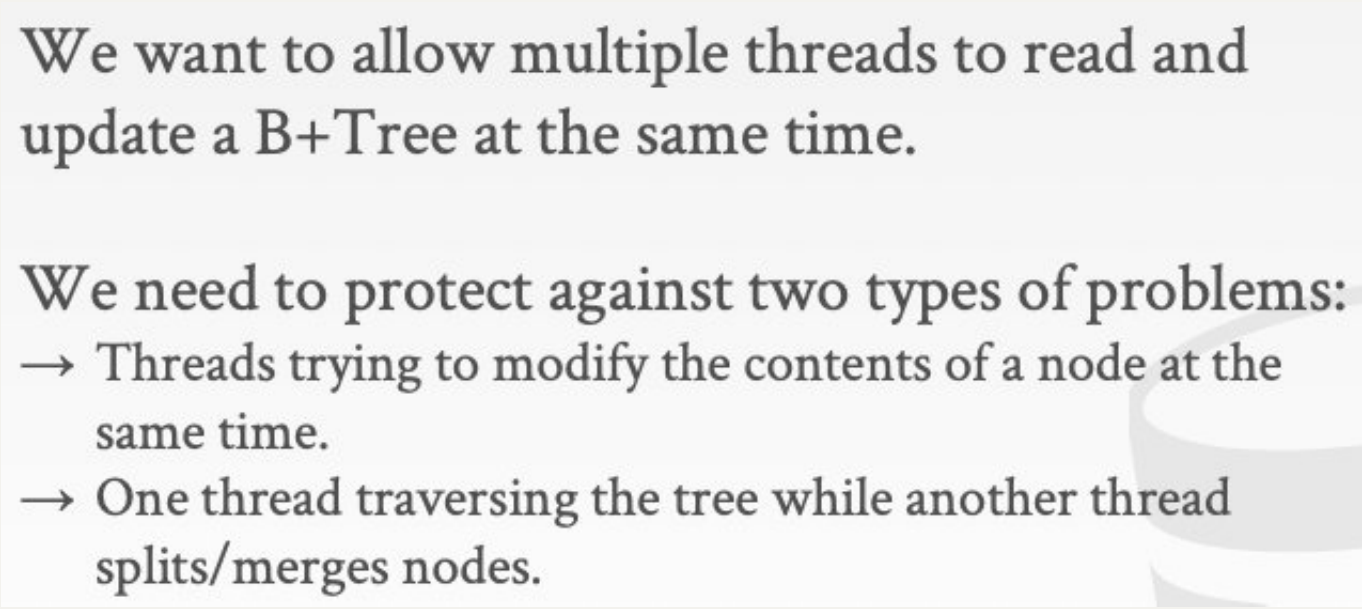


基于上述的CAS技术，当我们想向哈希表中的某个哈希槽里插入特定的值时，会先判断这个槽位是不是空的，如果是空的，那就执行插入

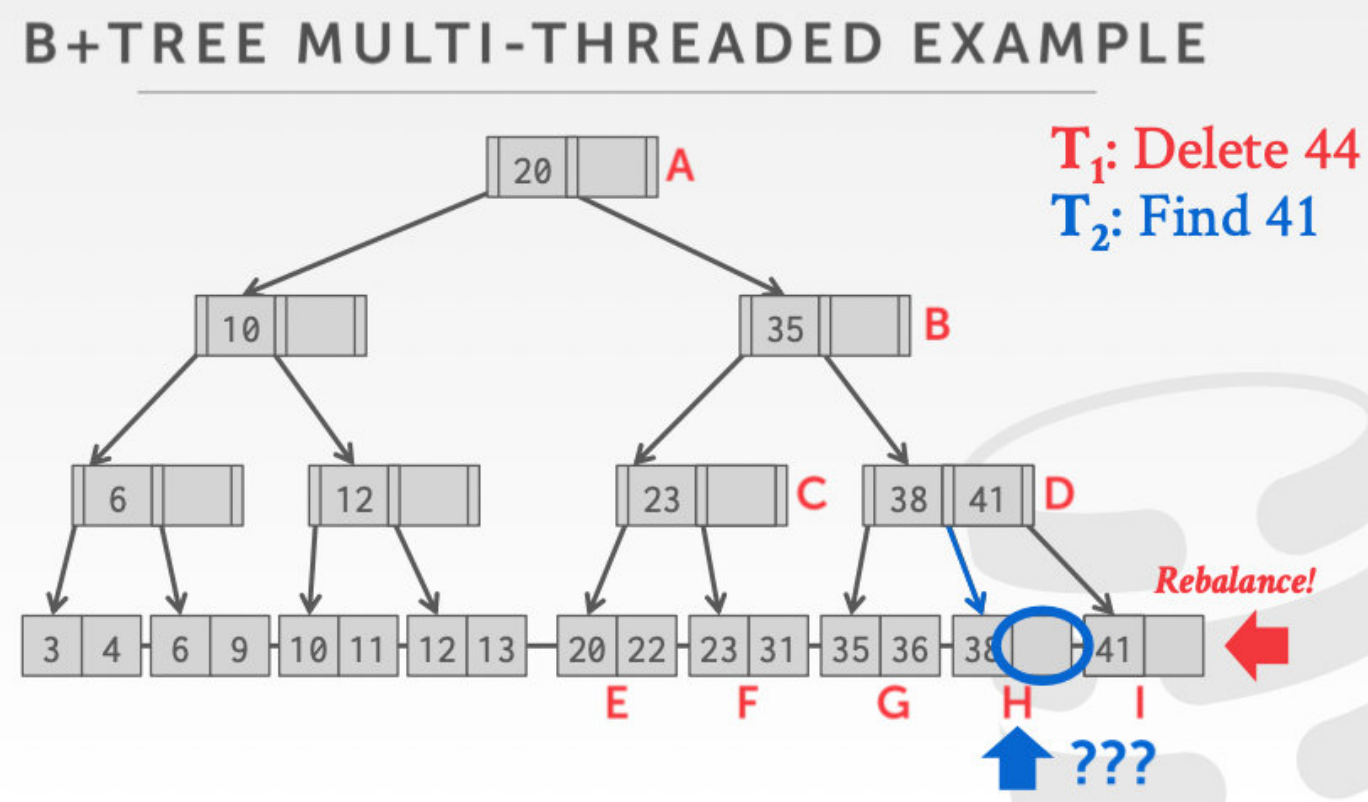
B+ Tree Latching

接下来介绍B+树的并发控制，从而允许多线程对B+树进行读写

为了实现并发控制，我们要考虑如下两方面的问题

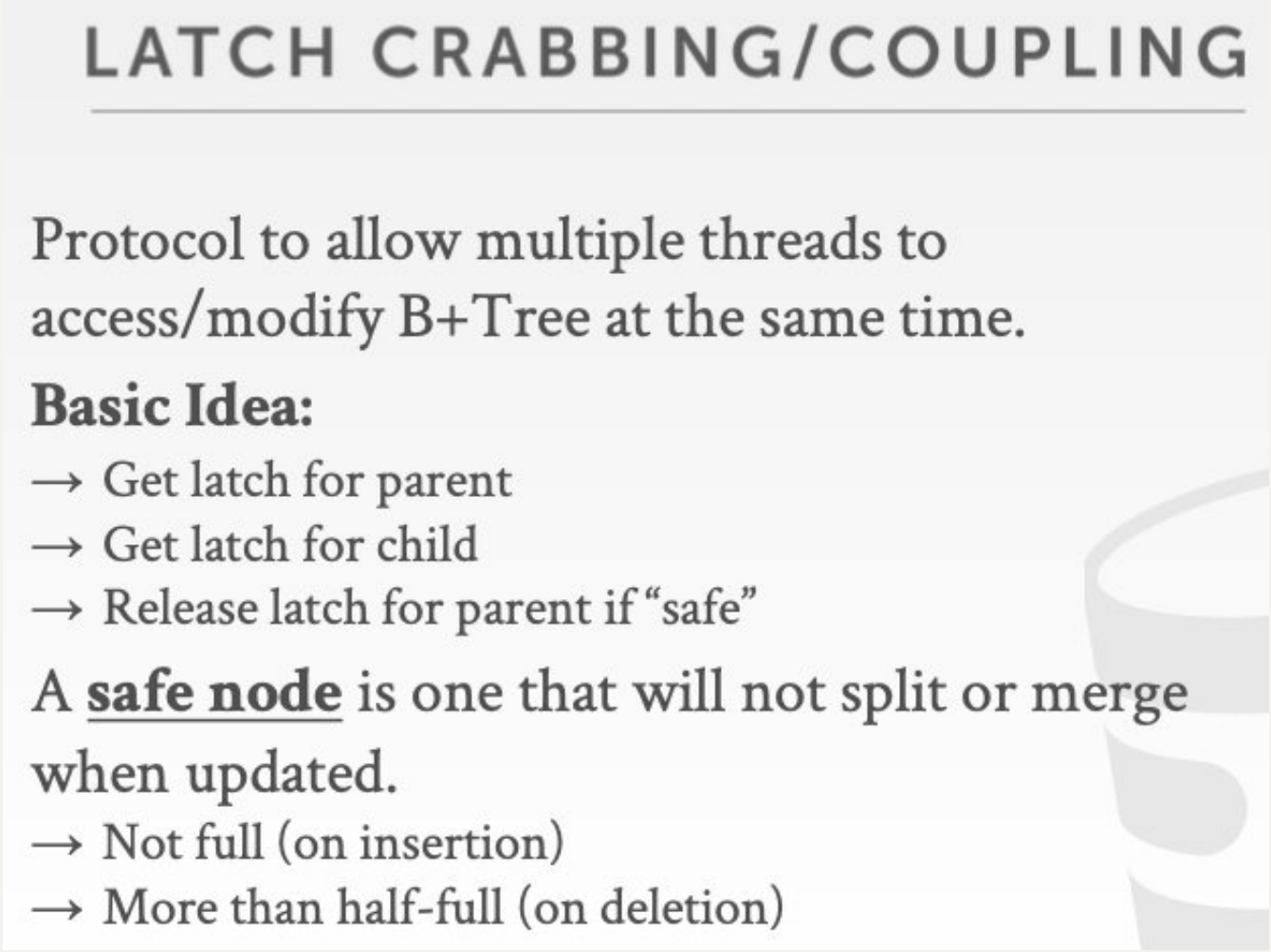


如果不进行并发控制，就会出现一些问题，比如说下面的这个例子，如下图（具体的详细过程参考slides），线程T1删去44对应的KV后，叶子节点空了，B+树变得不平衡，于是把该叶子节点和旁边的叶子节点进行再平衡：把41对应的KV转移到空的叶子节点中，但在转移还没有完成时，另外一个线程T2开始在B+树中寻找41对应的KV，在该KV的旧的位置找到了它，此时T1线程完成了转移操作，之后T2想要读KV的Value时，会发现这个地方是空的

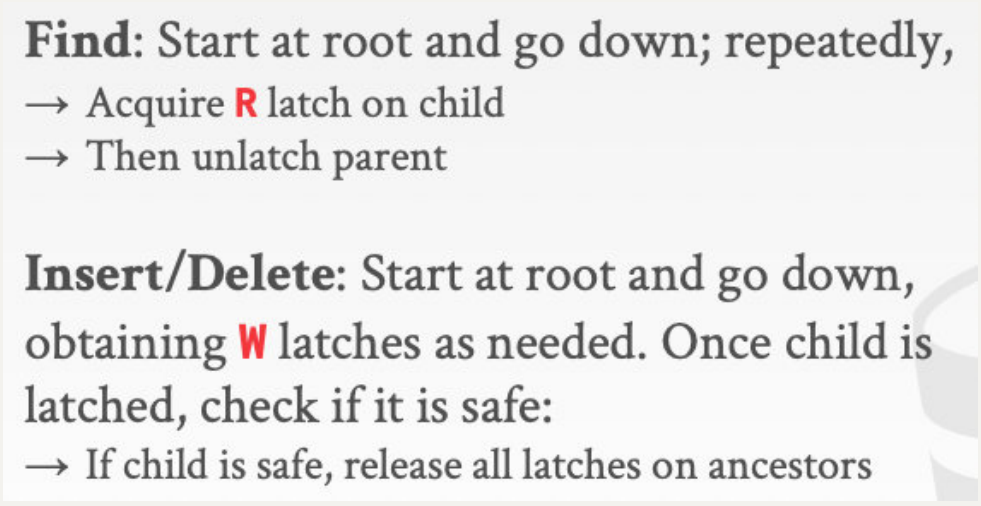


B+树的并发控制策略是"crabbing/coupling"，for example，从根节点往下搜索时，先把根节点锁上，如果发现我们想找的东西在左子树，那就再把左子树锁上，然后再判断根节点能不能解锁，如果能解锁的话就释放该锁，之后再从左子树往下走，获取它的子树的锁，再判断此时作为父节点的根节点的左子树的锁是否可以被释放...，像螃蟹一样，每次都是同时向前挪动一对儿钳子，这个策略里面也是每次处理两个节点以及它们的锁：先锁住下边的节点，再尝试解开上面的节点的锁，在如此的循环往复中前进

如何判断是否可以释放父节点的锁呢？就是去判断在这次的B+树更新中，是否会修改父节点的数据结构，也就是要去判断下面的子节点是否会发生分裂或合并：如果是的话就会影响上面的父节点的指针以及某些KV，因此不能释放上面父节点的锁；如果能够确认下面的子节点不会进行分裂或合并，那么上面的父节点在本次对B+树的操作中不会再变化了，就可以释放它的锁，让别的线程去访问它

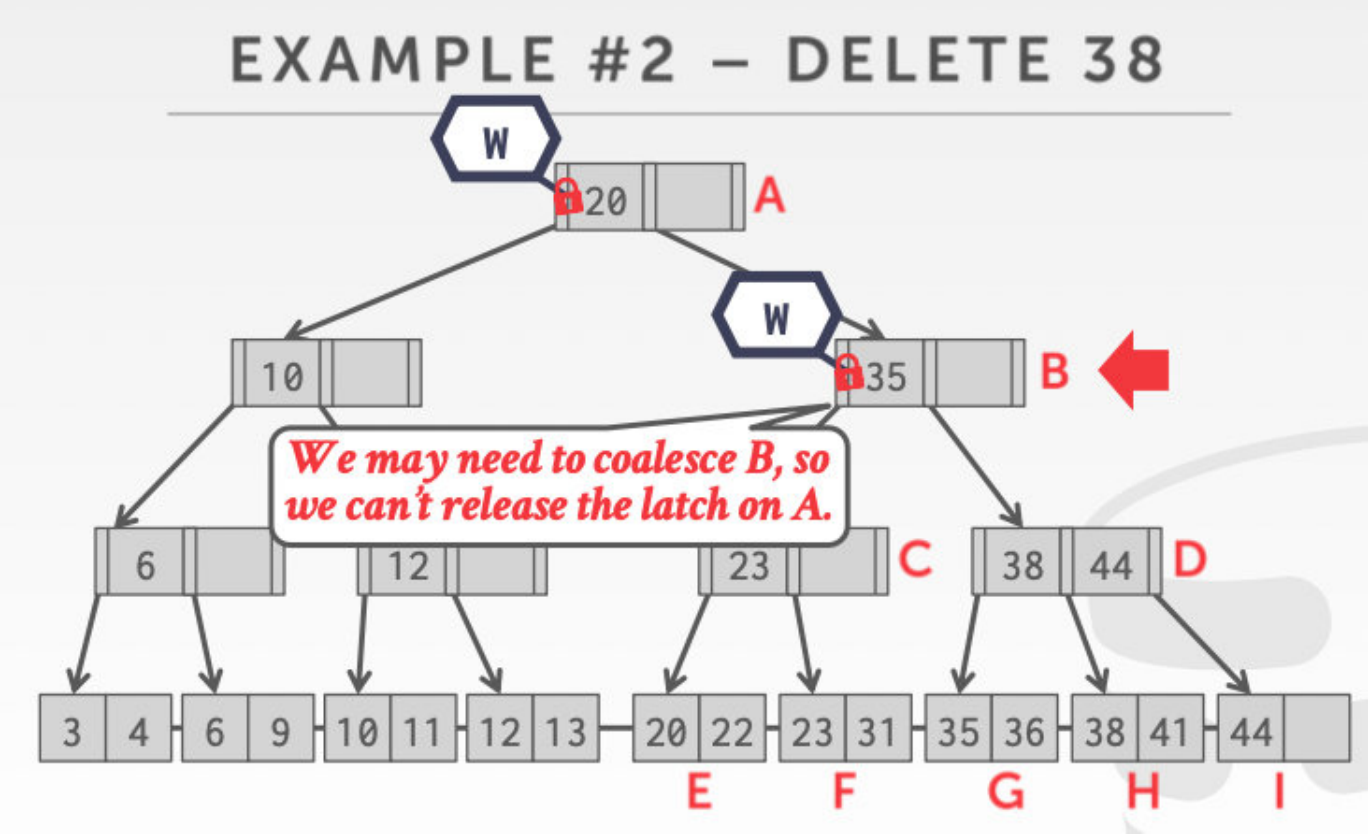


更详细的说，在对B+树进行读操作时，从根节点往下走，循环往复地进行“获取子节点的读锁，之后再释放父节点的读锁”这样的操作；对B+树进行插入/删除这种写操作时，从根节点往下走，先获取子节点的写锁，之后检查子节点是否安全（上图中有safe node的判断标准），如果安全的话释放所有祖先节点的写锁

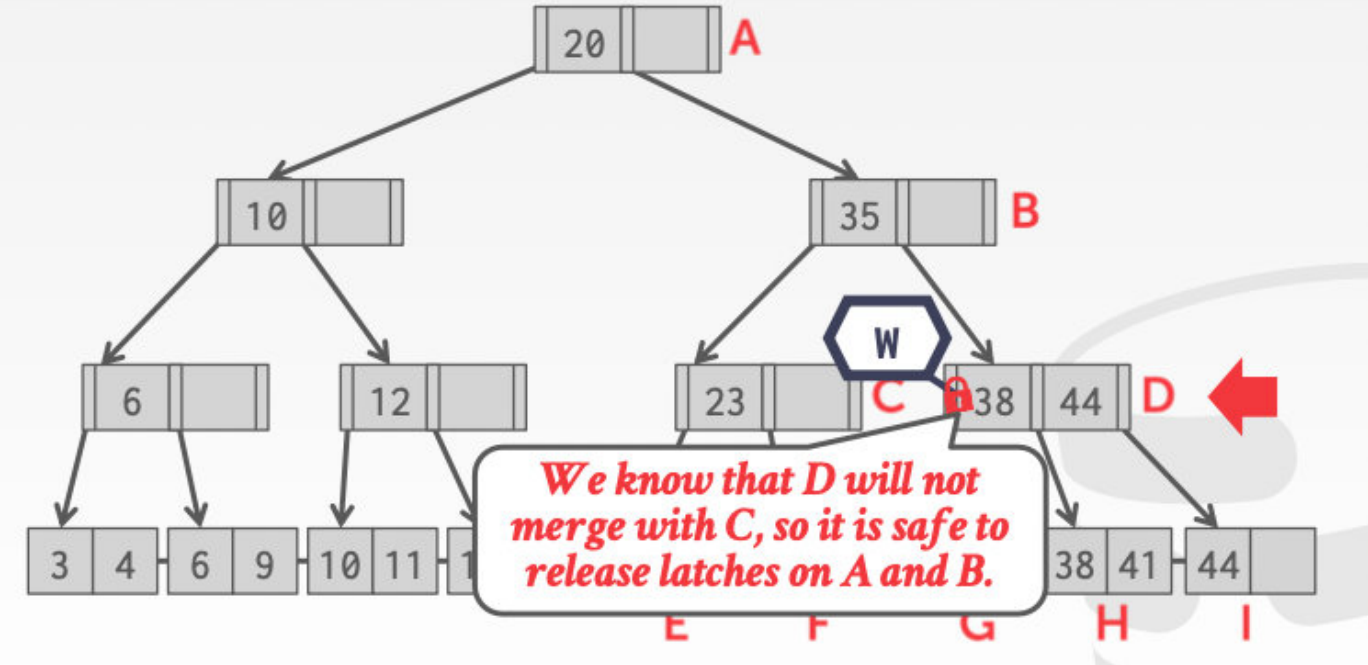


下面以比较复杂的删除操作来举例说明

如下所示，想要删除B+树里Key为38的KV，从根节点出发，先获取根节点的写锁，之后再获取相应子节点的写锁，因为B节点并不是more than half-full的，可能在未来参与合并操作，它并不安全（如果删除38对应的KV造成B下面的D或C节点的数据结构变化，进而可能导致C节点和D节点的合并，就会从而使B节点的数据结构发生变化，最后也使得根节点A的数据结构发生变化），因此还不能释放根节点A的写锁

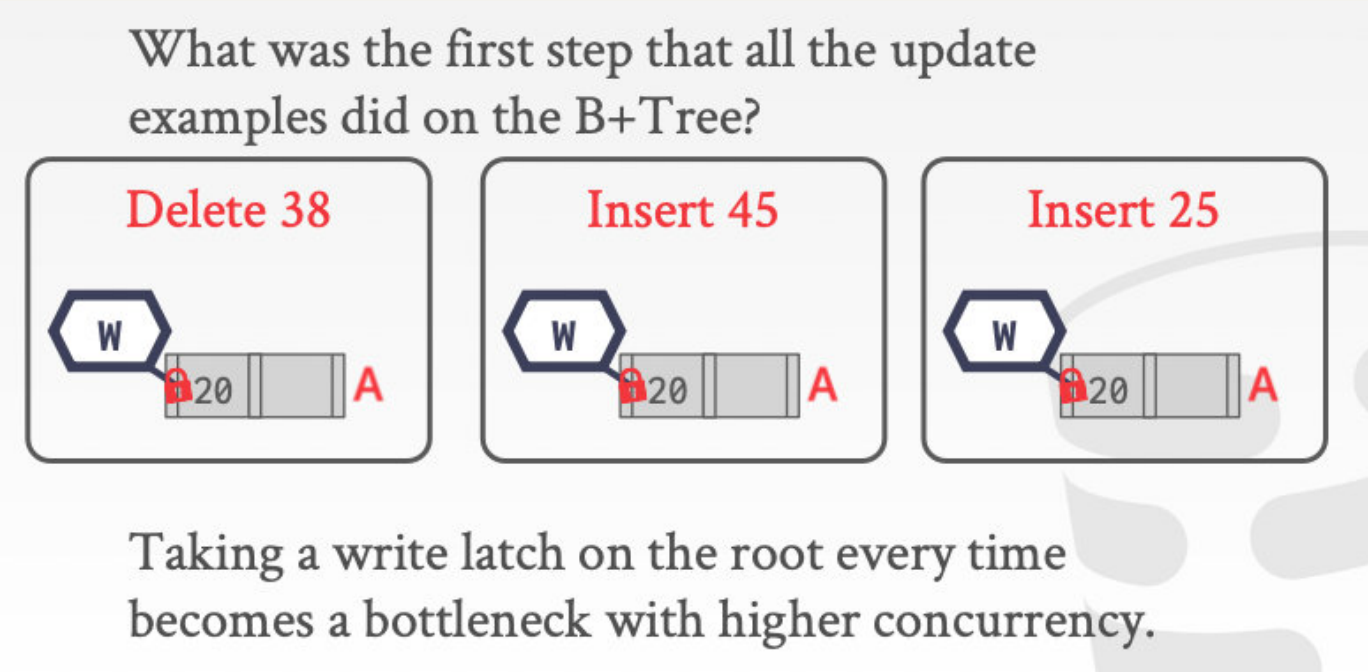


之后继续往下走，获取D节点的写锁，可以根据D节点more than half-full判断出它是安全的（就算删除38对应的KV，C节点和D节点也不会合并），于是把祖先节点A与B的锁都释放



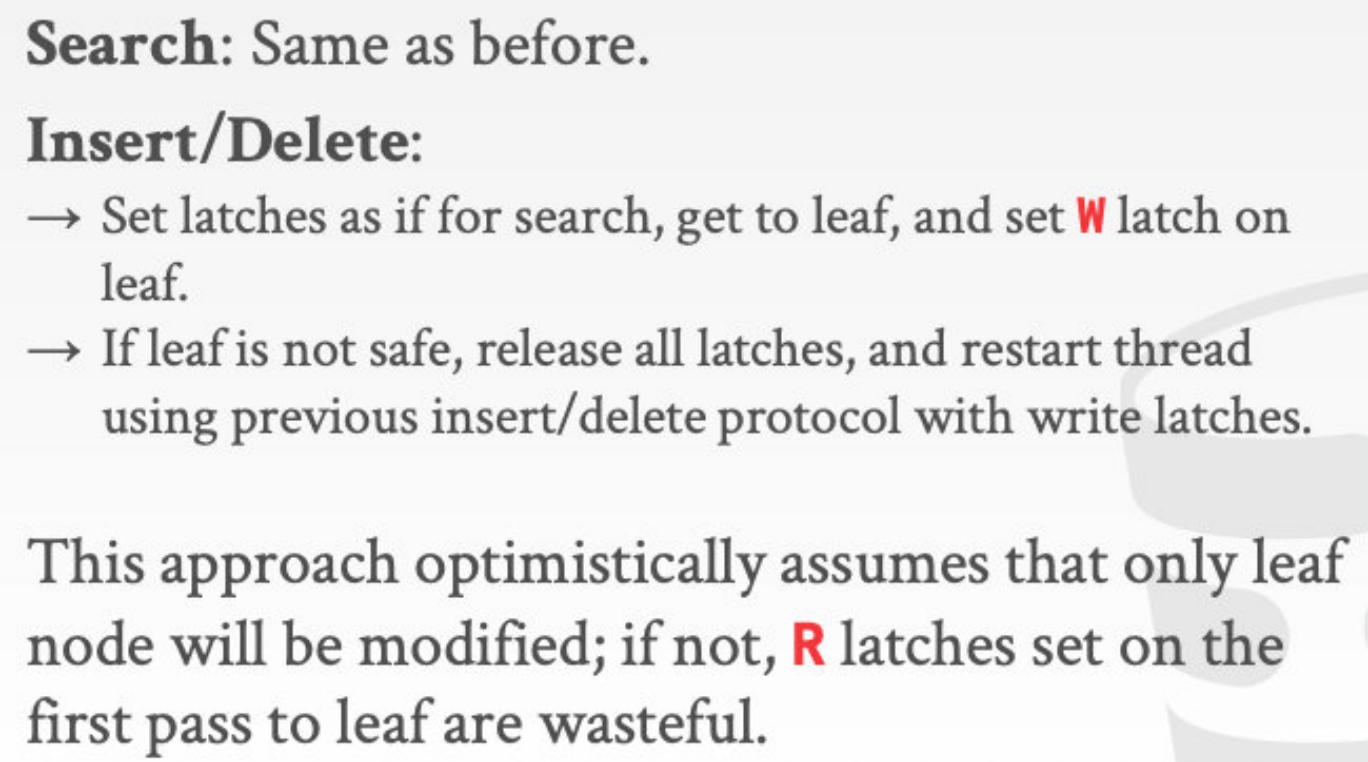
剩下的过程同理，详见slides

slides中给出的所有对B+树进行写的例子中，第一步做的事情都是获取根节点的写锁，这样一来其他的线程对根节点连读操作都进行不了，在高并发的场景下，这会导致性能瓶颈的出现



如何突破这个性能瓶颈呢？我们之所以在对B+树进行写操作时，先获取根节点的写锁，是担心有可能叶子节点的数据结构的改动一层一层地向上影响到了根节点，这是很悲观的想法，乐观的想法是去赌绝大部分操作不会对根节点与中间节点的数据结构有修改，它们只会修改叶子节点的数据结构，因此就有了如下的策略：

在读B+树时和之前一样，只获取/释放读锁，在进行插入/删除这样的写B+树的操作时，在到达叶子节点之前，只会获取/释放根节点与中间节点的读锁，如果到最后发现叶子节点不是safe node：它会分裂开或者与周围节点合并，那么就释放所有在此过程中已经获得了的锁，从根节点重新开始进行该线程想要做的操作（形象地说就是回滚），并且这次给所有的节点都上读锁。具体的流程可以参考slides里面insert 25那个例子，这种策略的开销可以被接受，因为绝大部分对B+树的操作不会引起中间节点的数据结构的修改



Leaf Node Scans

The threads in all the examples so far have acquired latches in a "top-down" manner.

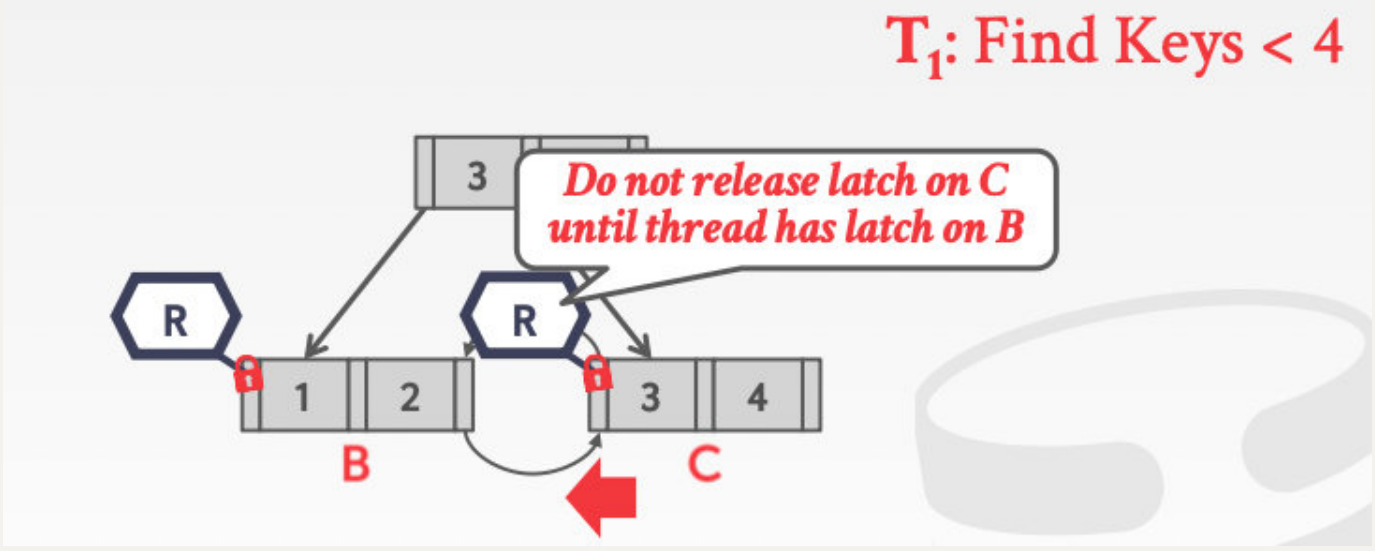
→ A thread can only acquire a latch from a node that is below its current node.

→ If the desired latch is unavailable, the thread must wait until it becomes available.

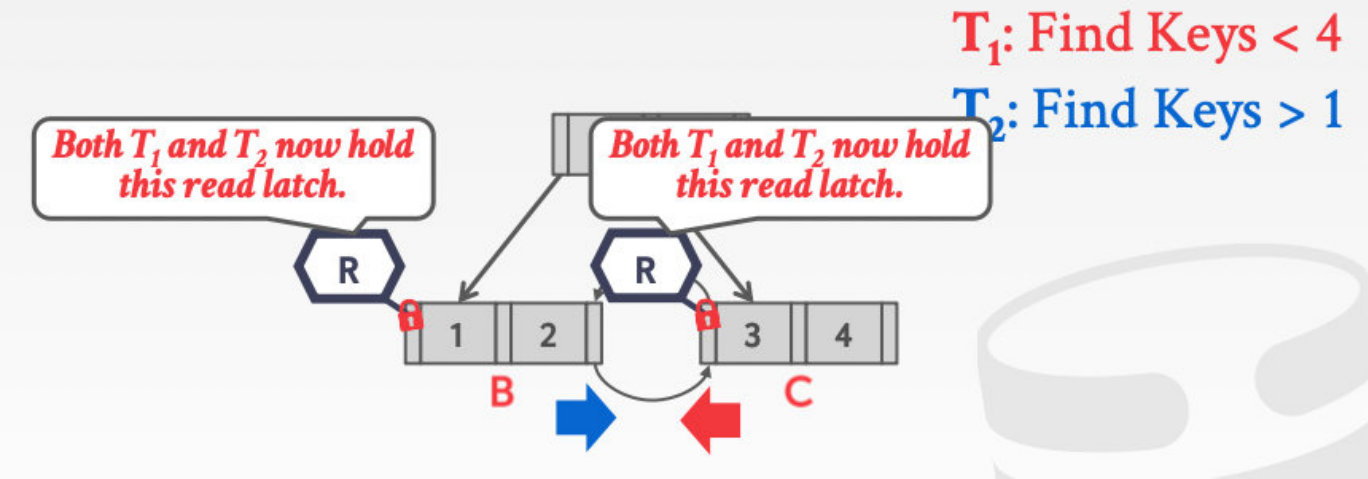
But what if we want to move from one leaf node to another leaf node?

在前面所介绍的策略与例子中，都是从根节点出发，自上而下地遍历节点，获取/释放锁，除此之外，B+树还支持对叶子节点的扫描

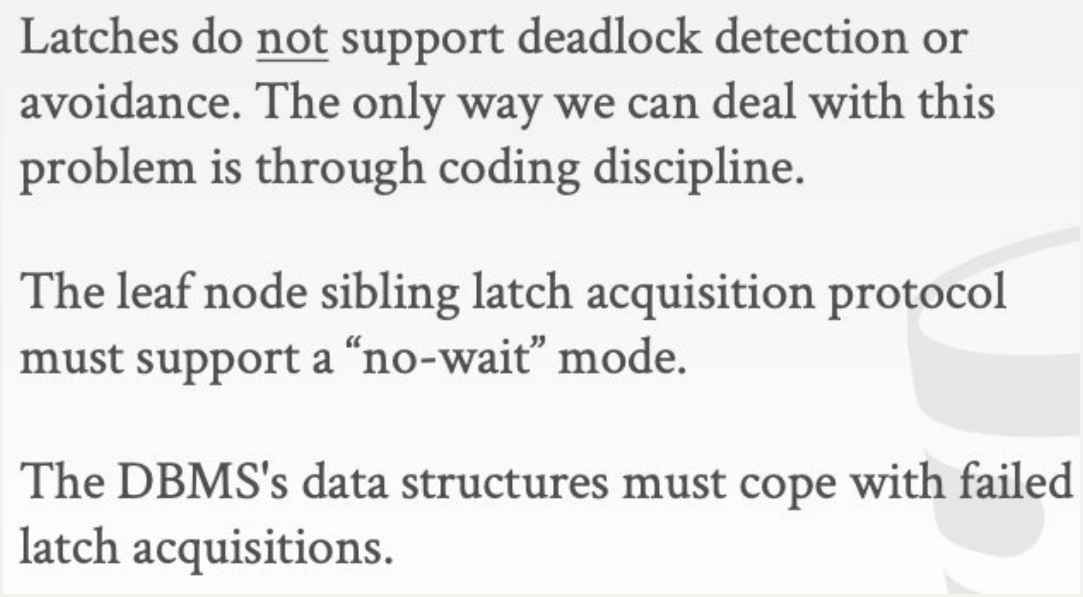
如下所示，线程T1想找所有Key<4的KV，它先获取Key等于4的KV所在的叶子节点的读锁，然后向前开始遍历这个叶子节点内部的KV，之后获取左侧兄弟节点的读锁，然后释放C节点的锁，再对兄弟节点内部的KV进行遍历



线程T2想找所有Key>1的KV，它先找到Key=1对应的叶子节点，然后从前向后开始以和上面相同的逻辑遍历叶子节点，因此，线程T1从后向前获取锁，先获取C节点的锁，再获取B节点的锁，然后释放C节点的锁；线程T2从前向后获取锁，先获取B节点的锁，再获取C节点的锁，然后释放B节点的锁，假如T1和T2并发执行，就会如下所示（详细过程见slides）



由于两个线程的操作都是find，只会获取/释放读锁，而读锁本身支持同时被两个线程持有，因此上述情况下程序可以执行，但如果涉及的是写锁（对应的场景比如说聚簇索引的情况下，全表的所有tuple都加一个字段），就会构成死锁（本质的原因就是多个线程在获取同一组锁的时候它们各自获取这组锁里的每个锁的顺序不同），B+树的并发控制的锁是不支持死锁检测的，只能通过设置规则从而避免死锁，比如说只允许朝一个方向的叶子节点遍历，很多DBMS也是这么做的（早期Mysql不支持倒序遍历也是出于这个原因，直到后来它实现了倒序索引，也就是叶子节点内部KV就是按Key倒序存放的）



这就是索引并发，后面会开始介绍SQL语句这种查询语句的执行过程

Reference:

https://www.bilibili.com/video/BV1eR4y1W7rW/?spm_id_from=pageDriver