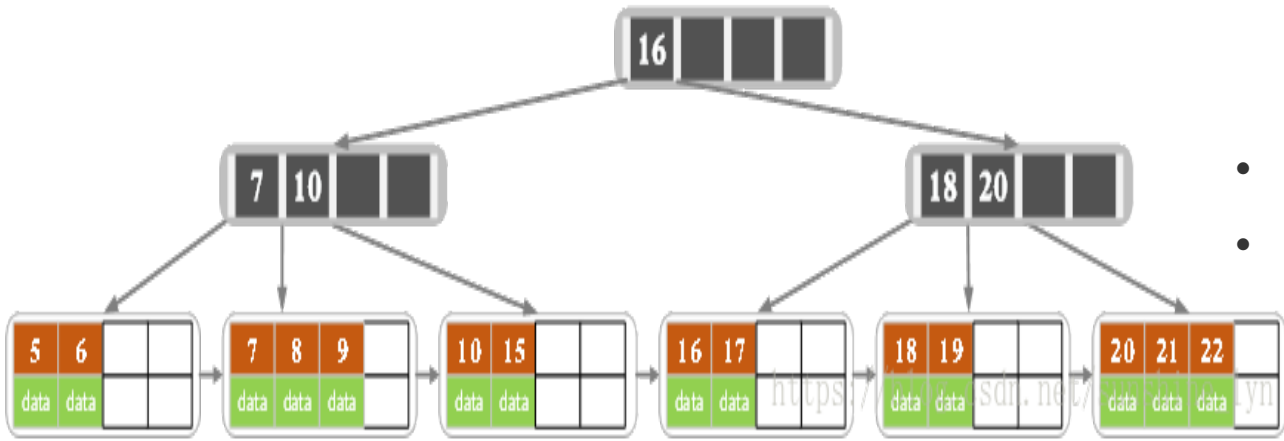
Concurrency Control of Index

胡卉芪
华东师范大学
数据科学与工程学院
hqhu@dase.ecnu.edu.cn

索引上的并发控制问题
(B+树为例)

B+树的数据结构



- 从根节点到叶节点的所有路径都具有相同的长度
- 所有数据信息都存储在叶节点上，非叶节点仅作为叶节点的索引存在
- 每个树节点最多拥有M个键值对
- 每个树节点（除了根节点）拥有至少M/2个键值对

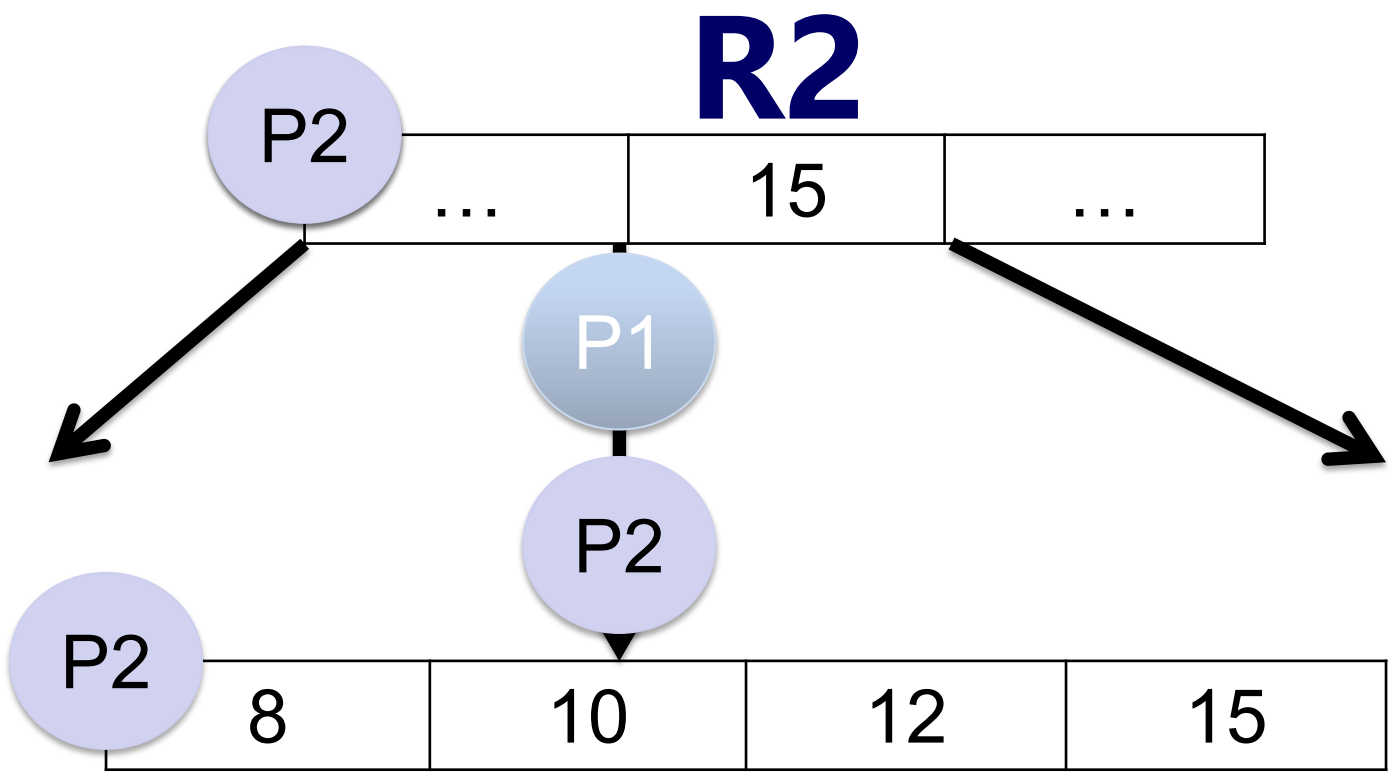
- B+树需支持以下操作：
- 单键值操作：Search/Insert/Update/Delete（以Search/Insert操作为例)
- 范围操作：Range Search

B+树并发控制的要求

- 正确的读操作：
  - R.1 不会读到一个处于中间状态的键值对：读操作访问中的键值对正在被另一个写操作修改
  - R.2 不会找不到一个存在的键值对：读操作正在访问某个树节点，这个树节点上的键值对同时被另一个写操作（分裂/合并操作）移动到另一个树节点，导致读操作没有找到目标键值对
- 正确的写操作：
  - W.1 两个写操作不会同时修改同一个键值对

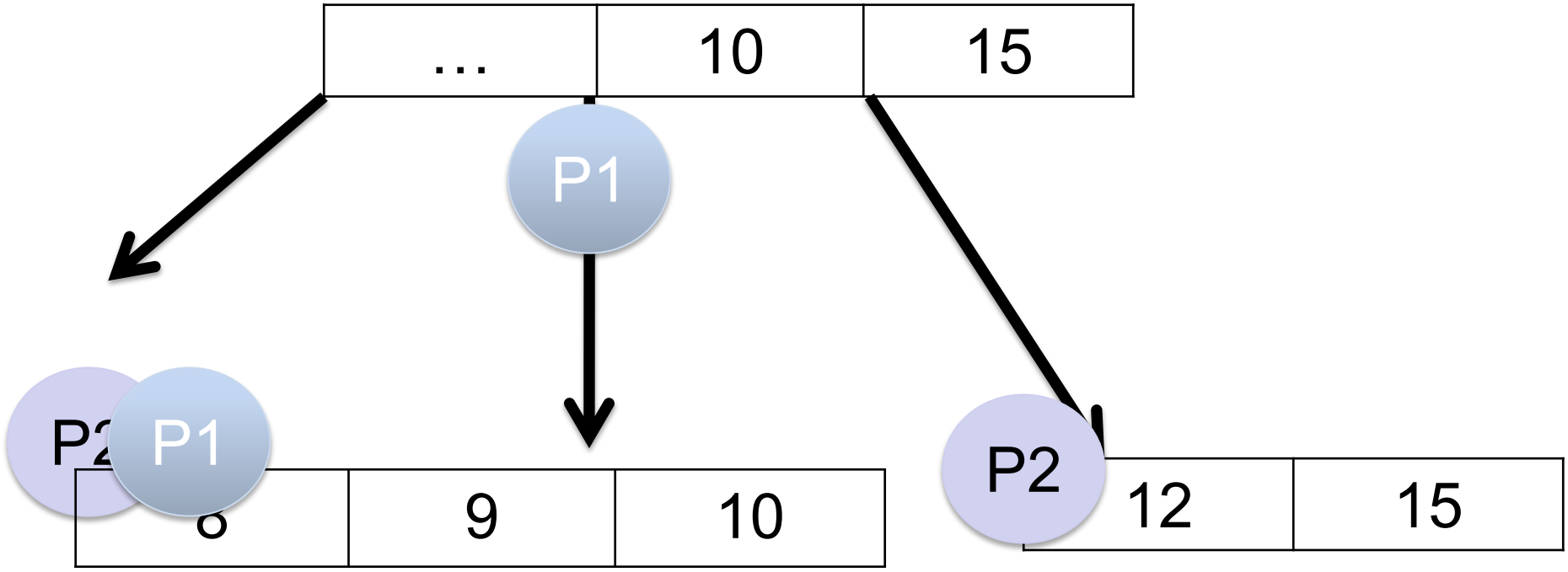
R2的并发控制需求

- R1, W1 是常见的写写冲突与读写冲突，如果是单版本存储，几乎任何时候都存在
- R2是B+树特殊的并发控制需求
  - 场景： 考虑一个读操作刚拿到一个叶子节点的指针，打算访问这个节点后半部分的数据，此时一个写操作发生使得节点分裂，那么后半部分数据将分裂到另一个节点上，因此读操作的指针无法获得目标数据。
  - 简单的对数据加锁无法保证R2的需求。



- P1 searches for 15
- P2 inserts 9

After the Insertion



- P1 searches for 15
- P2 inserts 9

P1 Finds no 15!

How could we fix this?

基础并发控制(MySQL5.6)

- 加锁的方式解决
  - 对整个索引上锁

SL (Shared Lock): 共享锁 — 加锁 （又称读锁）
SU (Shared Unlock) : 共享锁 — 解锁 （又称写锁）
XL (Exclusive Lock) : 互斥锁 — 加锁
XU (Exclusive Unlock): 互斥锁 — 解锁

	S	X
S	y	n
X	n	n

Table 1: Compatibility Matrix for S and X Locks

基础并发控制(读操作)

/* Algorithm1. 读操作 */

1. SL(index) //对整个索引加读锁
2. Travel down to the leaf node
3. SL(leaf) //先对叶节点加读锁
4. SU(index) //释放索引的读锁
5. Read the leaf node
6. SU(leaf) //释放叶节点读锁

基础并发控制(写操作-悲观)

/* Algorithm2. 悲观写操作 */

1. XL(index)
2. Travel down to the leaf node
3. XL(leaf) /* lock prev/curr/next leaves */
4. Modify the tree structure
5. XU(index)
6. Modify the leaf node
7. XU(leaf)

为何要先锁住叶子节点？

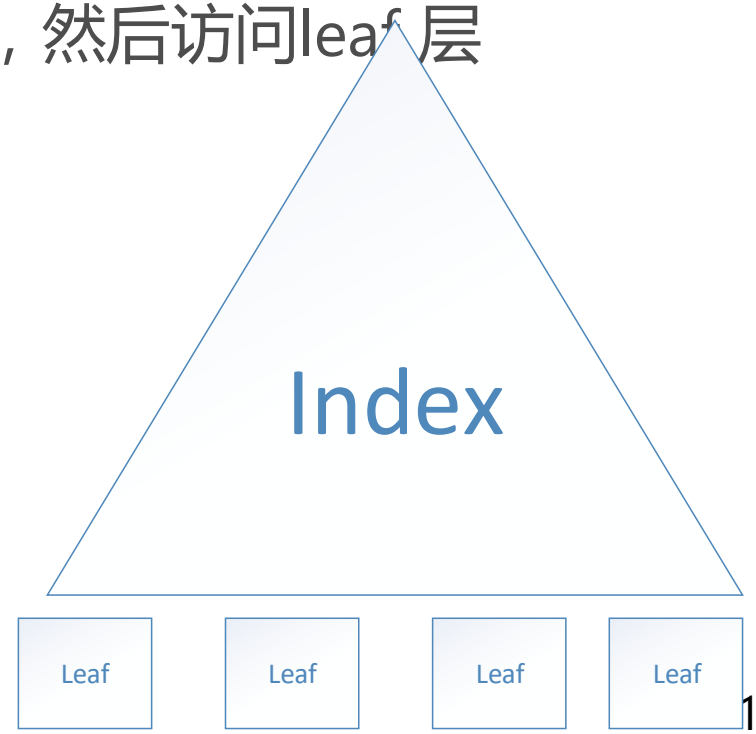
防止索引修改完成后，叶子节点已经被其他写操作修改掉

分析

- 我们可以把整个B+树看作两层
  - 一个写操作首先访问到叶子节点，修改Index层，然后修改leaf 层
  - 一个读操作首先访问Index层，然后访问leaf 层

为何写操作要先锁住叶子节点？

防止索引修改完成后，叶子节点已经被其他写操作修改掉



只锁修改分支

- 不再对整个索引加锁，使用节点粒度的S/X锁

只锁修改分支

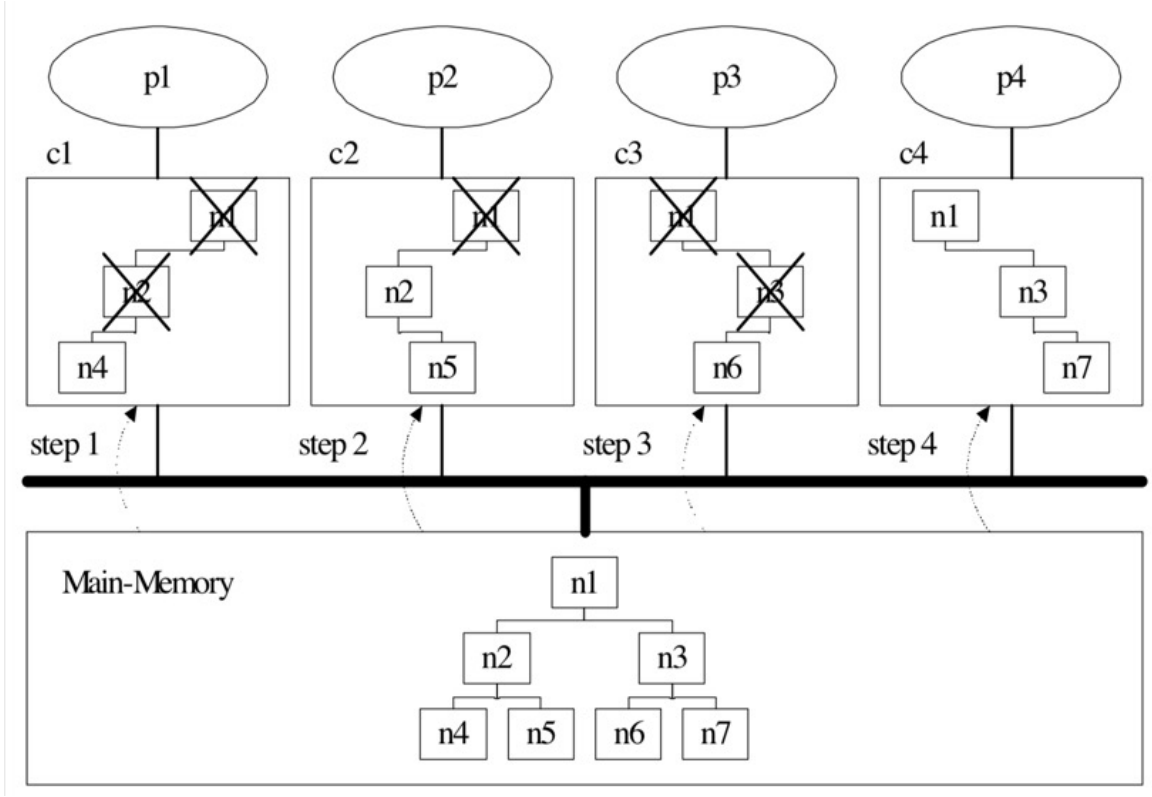
```
/* Algorithm5. 写操作 */
1.  current <= root
2.  XL(current)
3.  While current is not leaf do {
4.     XL(current->son)
5.     current <= current->son
6.     If current is safe do {  //如何判断？
7.        /* Unlocked ancestors on stack. */
8.        XU(locked ancestors)
9.     }
10. }
11. /* Already lock the modified branch. */
12. Modify the leaf and upper nodes
13. XU(current) and XU(locked ancestors)
```

只锁修改分支

- /* Algorithm4. 读操作 */
- 1. current <= root
- 2. SL(current)
- 3. While current is not leaf do {
  - 4. SL(current->son)
  - 5. SU(current)
  - 6. current <= current->son
  - 7. }
- 8. Read the leaf node
- 9. SU(current)

在多核系统下加锁方式效率低

- 锁的cache coherence问题导致CPU效率低下
  - 《Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems》



Cache Coherence问题

- 繁加锁操作在多核处理器上会产生Coherence Cache Miss过高的问题。以上图为例，假设有4个处理器（p1/p2/p3/p4），每个处理器分别有自己的private cache（c1/c2/c3/c4）。假设有4个线程（p1/p2/p3/p4），与处理器一一绑定。下文中的n1/n2/n3/n4/n5/n6/n7可以指的是树节点的锁，也可以指代树节点。为什么频繁加锁会引入较高的Coherence Cache Miss开销？

- a. p1访问树节点n1/n2/n4，然后将它们放在缓存c1；
- b. p2访问树节点n1/n2/n5，然后将它们放在缓存c2；
- c. p2修改的S锁会导致缓存c1中的n1/n2失效；
- d. 注意即使缓存c1中有足够大的空间，这些缓存缺失操作依然会发生；
- e. p3访问树节点n1/n3/n6，然后导致缓存c2中的n1失效；
- f. p4访问树节点n1/n3/n7，然后导致缓存c3中的n1/n3失效；

消除R2所带来的加锁-Blink Tree

- 在每个节点添加一个右指针，指向右兄弟节点
- 一旦search的key比节点最高值打，则使用右指针访问右兄弟而不是孩子
- 假定节点写操作是原子的，那么search操作无需加锁

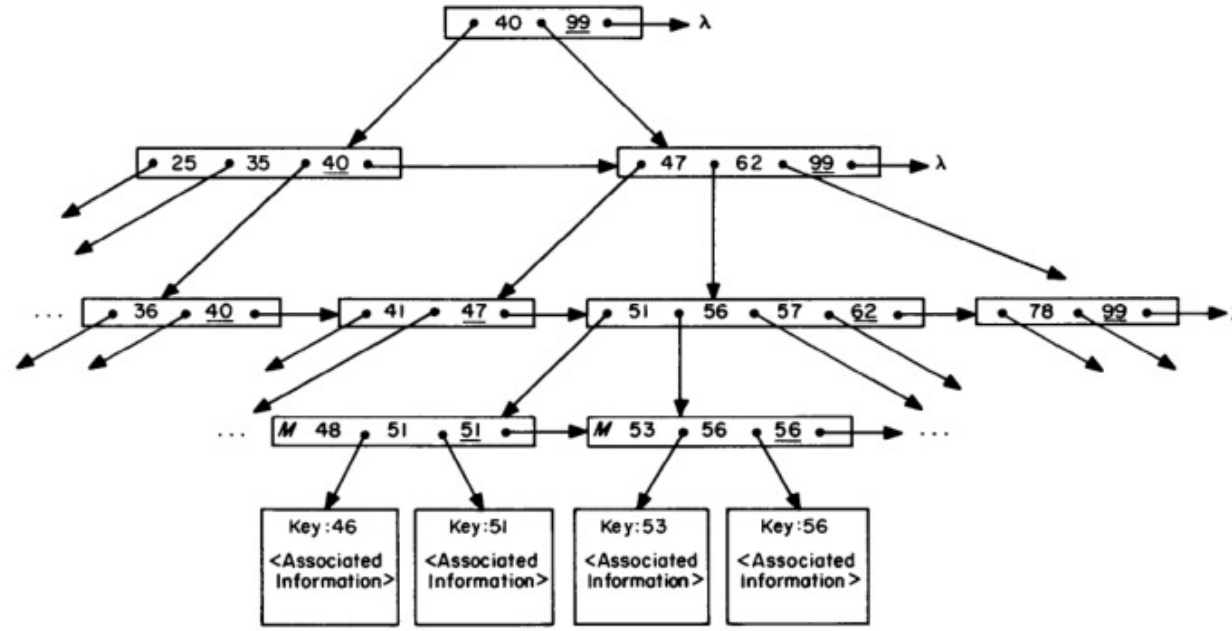
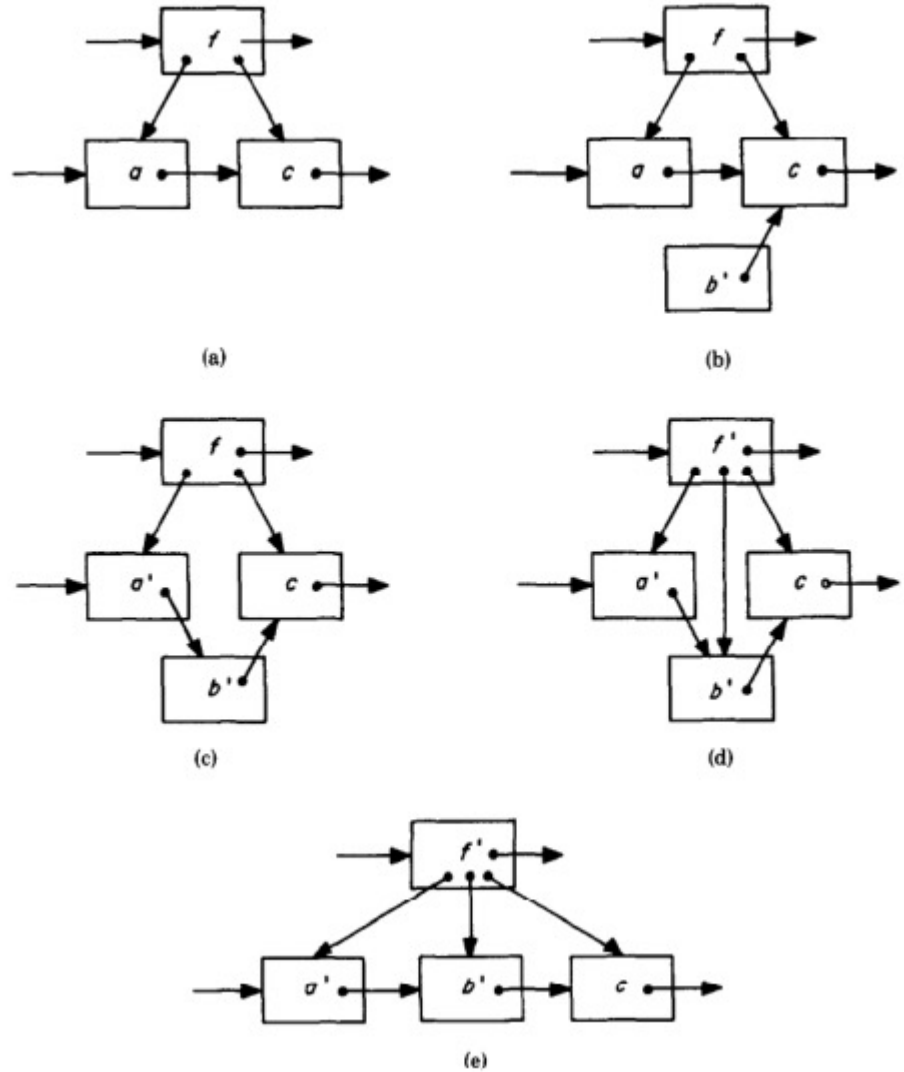


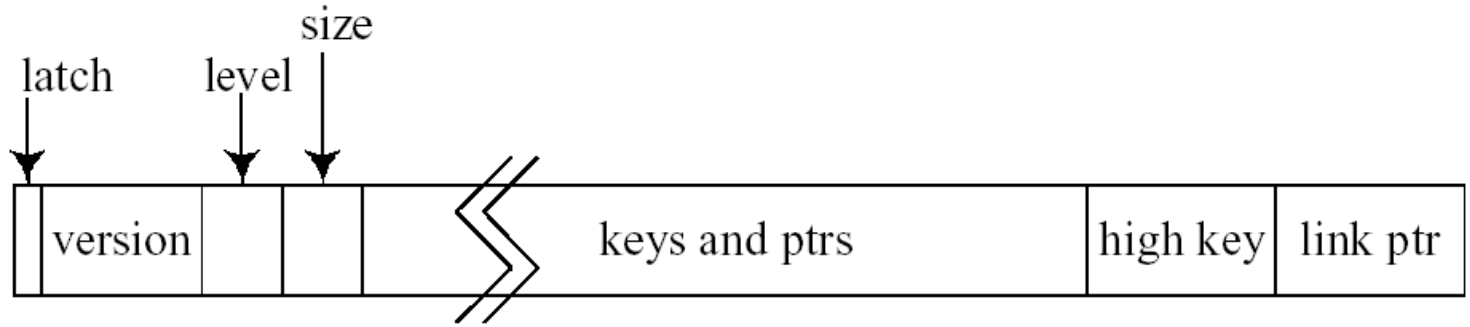
Fig. 7. A B^{link}-tree.

Blink-tree的插入



引入乐观读-Olift

- 使用一个节点的版本标记判断节点是否更新
  - 减少R1带来的读操作带来的锁或latch开销




```
procedure traverse(root, key){
1.    node:= root;
2.    while (node is not a leaf) {
3.       t:= turn-off-LSB(node.ccinfo);
4.       next:= find_next(node, key);
5.       if (node.ccinfo = t) node:= next;
6.    }
7.    return node;
}
```

判断节点是否被修改过

还有很多后续

- Masstree
  - 《Cache craftiness for fast multicore key-value storage》
- Bw-tree
  - The Bw-Tree: A B-tree for New Hardware Platforms

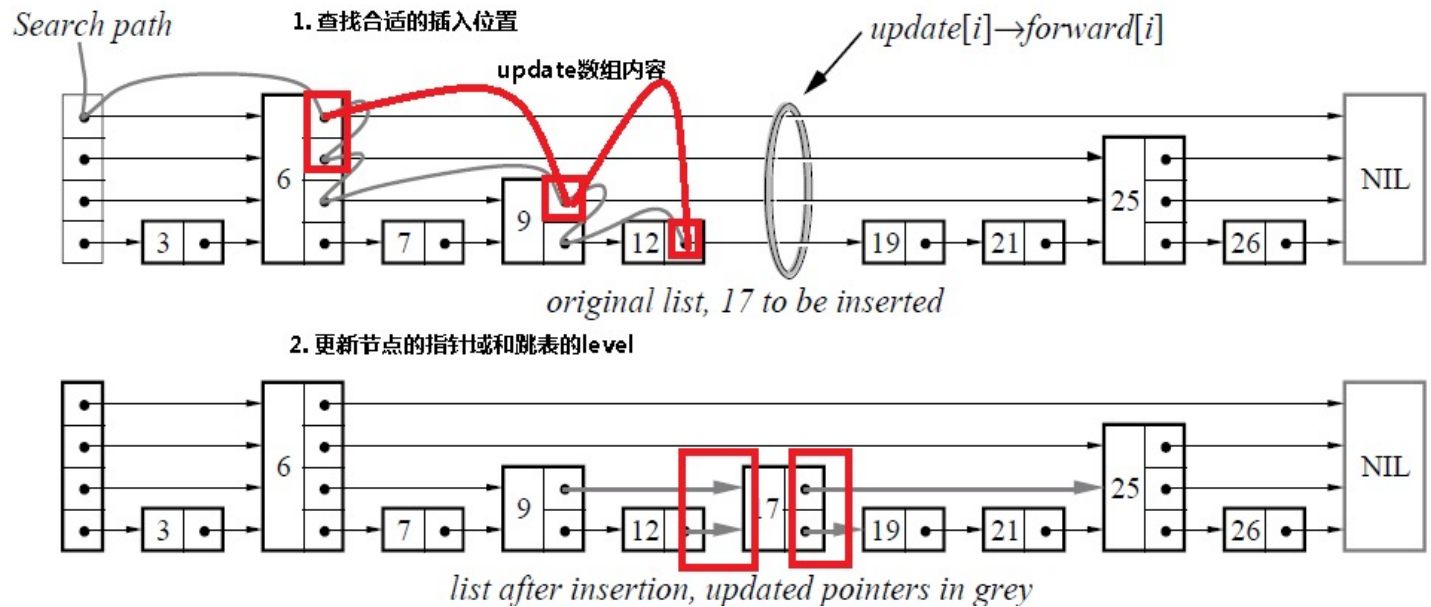
SkipList上的并发控制问题

SkipList的并发维护-仅讨论插入和查找

- 首先我们讨论并发操作对数据结构本身的正确性是否会产生破坏
  - 插入/插入操作
    - 丢失更新的问题
  - 查找/插入操作
    - 插入一个数据，有没有可能让另一个已经存在数据读取失败
- 其次、查找/插入操作的一致性

数据结构正确性

- 插入17，有无可能让19或者12的查找失败？
  - 如果向前指针自下而上更新？
  - ~~如果自上而下更新？~~



插入/查找操作的一致性

- 考虑插入17和查找17两者并发？
  - 理论上，线性一致性可规约到操作时间内的任意一个点
  - 实际上，会为每个操作“定“一个时间点，两种形式
    - timestamp
    - 版本(一个增长的数字)

插入/查找操作的一致性

- 这样我们把一致性问题变成一个很具体的问题
  - 假设插入17的timestamp是A, 查找是B
    - 如果A>B , 那么查找17将返回找不到
    - 如果A<B, 那么查找17能找到
- 所以，假定我们定插入/查找的timestamp分配时间是他们操作开始的时候，是否能做到线性一致？
  - 如果A>B, 一定能做到，因为可通过版本号直接判断
  - 如果A<B（插入在前），有可能无法保证，为什么？
    - 实际插入操作可能迟迟没有发生
  - 上述定timestamp的方式无法做到线性一致