数据库索引数据结构总结

摘要

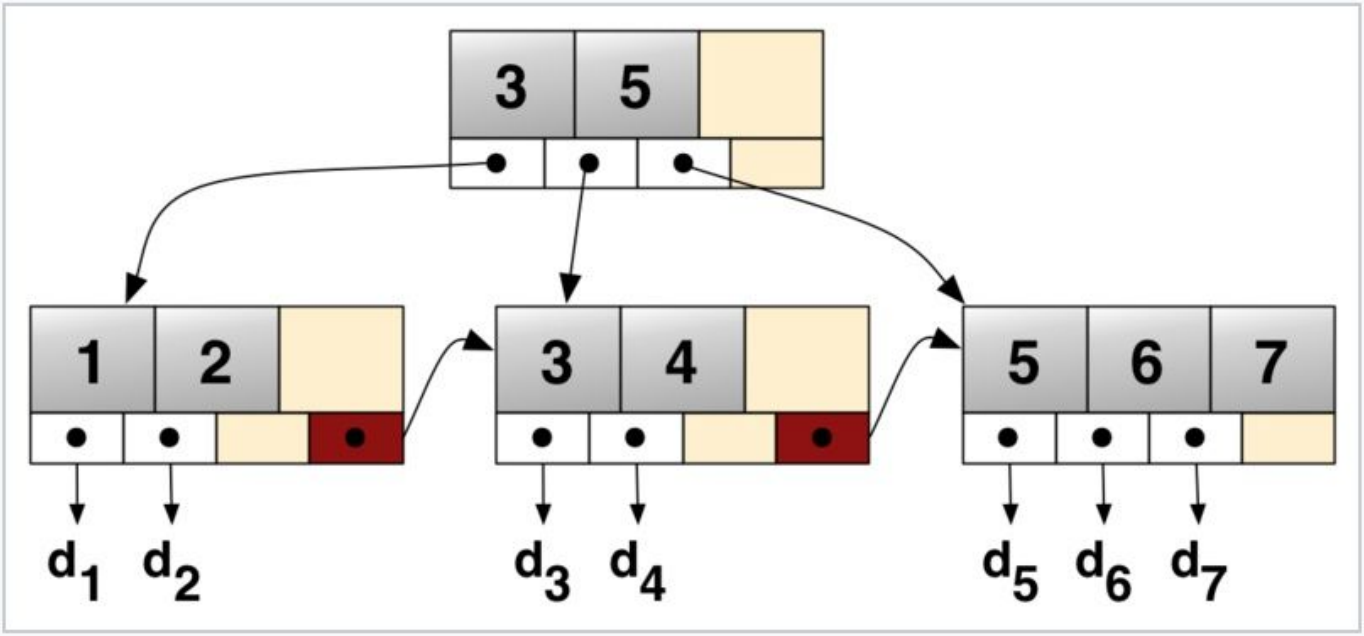
数据库索引是数据库中最重要的组成部分，而索引的数据结构设计对数据库的性能有重要的影响。本文尝试选取几种典型的索引数据结构，总结分析，以窥数据库索引之全貌。

B+Tree

B+Tree 是一种树数据结构，是一个n叉排序树，每个节点通常有多个孩子，一棵B+Tree包含根节点、内部节点和叶子节点。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。

B+Tree 几乎是数据库默认的索引实现，其细节如下：

> 维基百科在 B+ 树中的节点通常被表示为一组有序的元素和子指针。如果此B+树的序数（order）是m ，则除了根之外的每个节点都包含最少$ {\displaystyle \lfloor m/2\rfloor } \lfloor m/2\rfloor$ 个元素最多 m-1 个元素，对于任意的节点有最多 m 个子指针。对于所有内部节点，子指针的数目总是比元素的数目多一个。因为所有叶子都在相同的高度上，节点通常不包含确定它们是叶子还是内部节点的方式。 每个内部节点的元素充当分开它的子树的分离值。例如，如果内部节点有三个子节点（或子树）则它必须有两个分离值或元素 a1 和 a2。在最左子树中所有的值都小于等于 a1，在中间子树中所有的值都在 a1 和 a2 之间((a1，a2])，而在最右子树中所有的值都大于 a2。



把键1–7连接到值 d_1–d_7 的B+树。链表（红色）用于快速顺序遍历叶子节点。树的分叉因子 b=4。

B+Tree 有如下性质：

1. 查询时间复杂度为 $O(\log_m n)$
2. 插入时间复杂度 $O(\log_m n)$
3. 删除时间复杂度 $O(\log_m n)$
4. 搜索一个范围的键（k 个键）时间复杂度为 $O(\log_m n + k)$

B+ Tree 的多线程同步

- **搜索：**从根节点开始，获取子节点的读闩，然后释放父节点的读闩；重复这个过程，直到找到目标节点位置。
- **插入/删除：**从根节点开始，获取子节点的写闩；重复这个过程，直到找到目标节点位置；如果子节点是安全的，插入/删除不会引起树结构的变化即父节点不需要调整，可释放所有祖先写闩；乐观的插入/删除是先走搜索获得目标节点的读闩，如果目标节点并不安全，则回归上述从根节点获得写闩的过程。

Skip List（跳表）

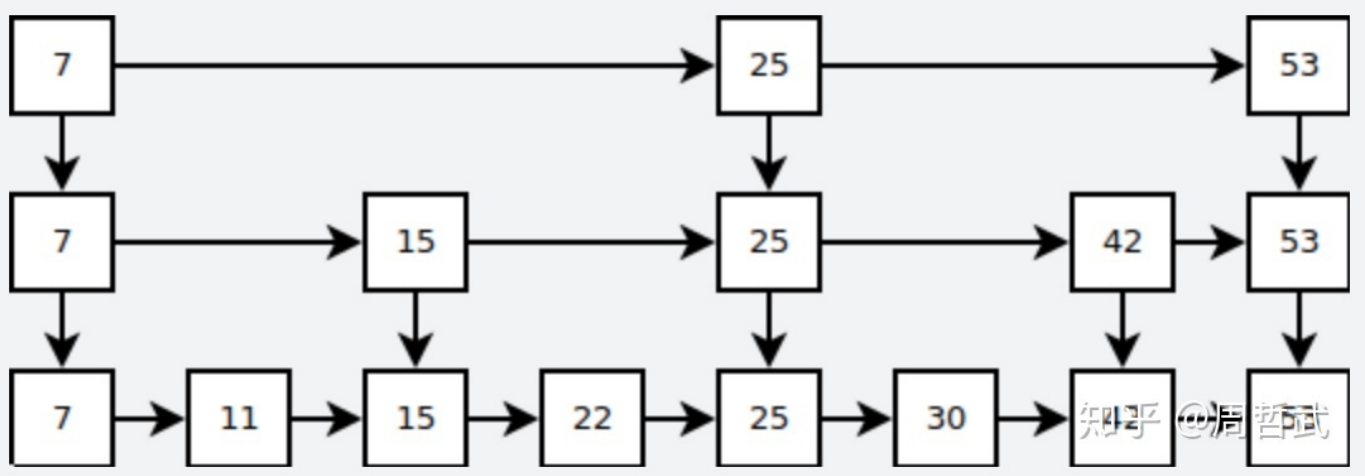
Skip List是一种随机化的数据结构，基于并联的链表，其效率可比拟于二叉查找树（对于大多数操作需要O(log n)平均时间）。基本上，跳跃列表是对有序的链表增加上附加的前进链接，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分列表(因此得名)。所有操作都以对数随机化的时间进行。Skip List可以很好解决有序链表查找特定值的困难。

一个跳表，应该具有以下特征：

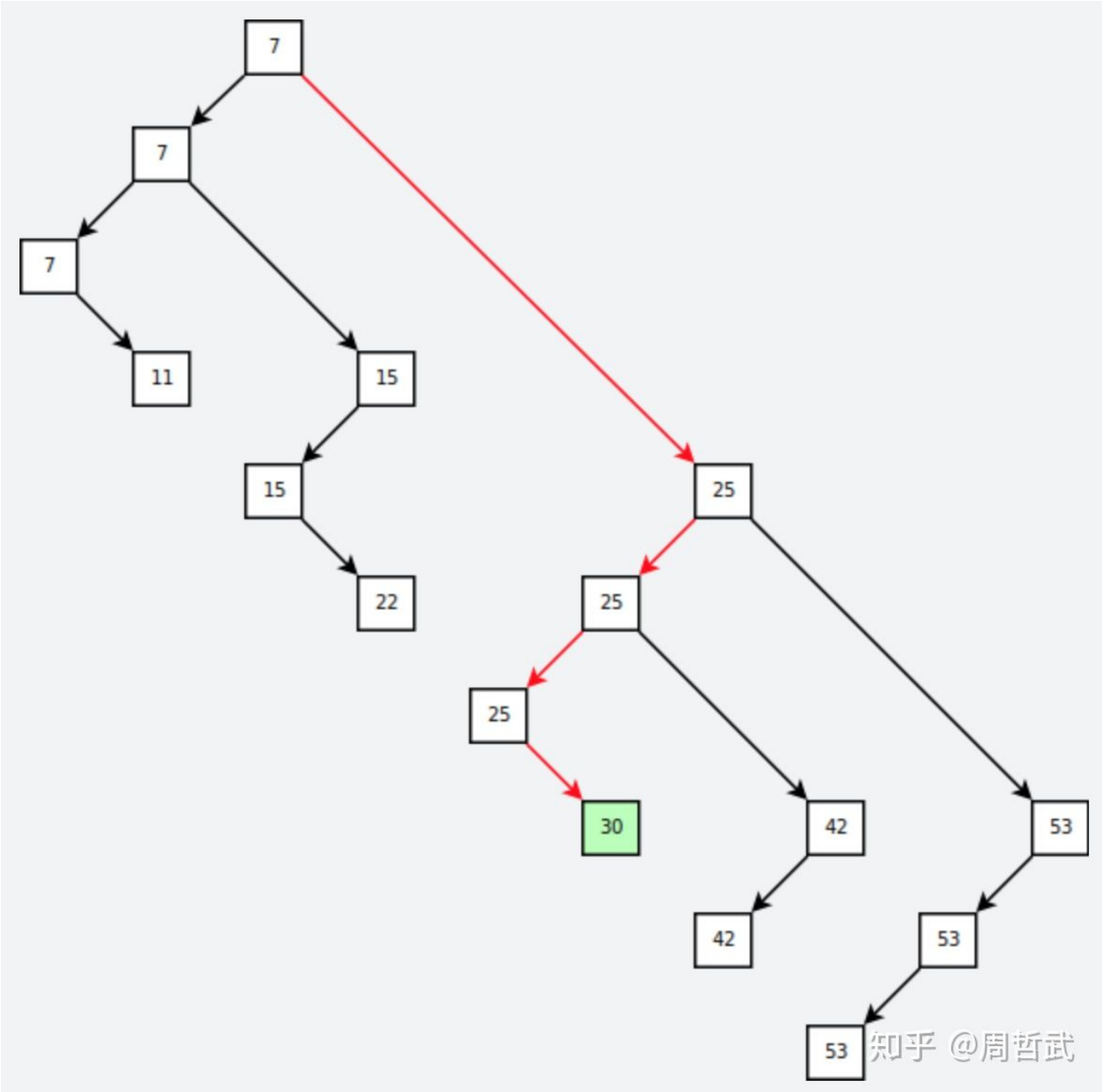
1. 一个跳表应该有几个层（level）组成；
2. 跳表的第一层包含所有的元素；
3. 每一层都是一个有序的链表；
4. 如果元素x出现在第i层，则所有比i小的层都包含x；
5. 第i层的元素通过一个down指针指向下一层拥有相同值的元素；
6. 在每一层中，-1和1两个元素都出现(分别表示INT_MIN和INT_MAX)；
7. Top指针指向最高层的第一个元素。

相对于 B+Tree，Skip List 有如下优势：

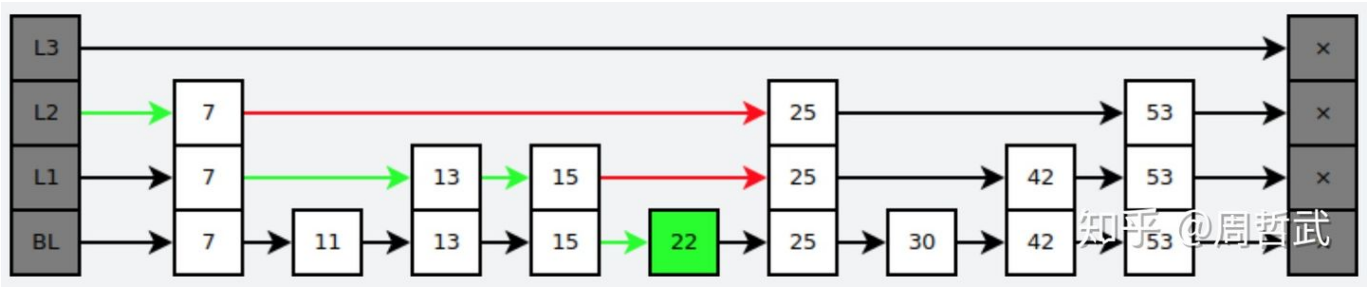
- B+ Tree 的插入删除操作有可能会引起树结构的变化，需要从新平衡；与之相对的，跳表插入要简单的多，更加简单高效。
- B+ Tree 的实现诸如保持树平衡非常复杂；与之相对的，跳表并没有非常复杂的逻辑，实现相对更加简单。
- 取下一个元素可以再常数时间内，相对于 B+ Tree 的对数时间。
- 因为链表非常简单，可以很容易的修改跳表结构，以更好地支持诸如范围索引之类的操作。
- 链表结构使得多线程修改可以仅用 CAS 保证原子性，从而避免重量级的同步机制。
- 链表的持久化更加简单。



跳表看起来非常像树，比如说检索



跳表横向看来是有很多链表组成，然而指针跳转对于 CPU 缓存 来讲非常不友好，可以用纵向数组来实现跳表以增加 CPU 缓存。



Bw-Tree

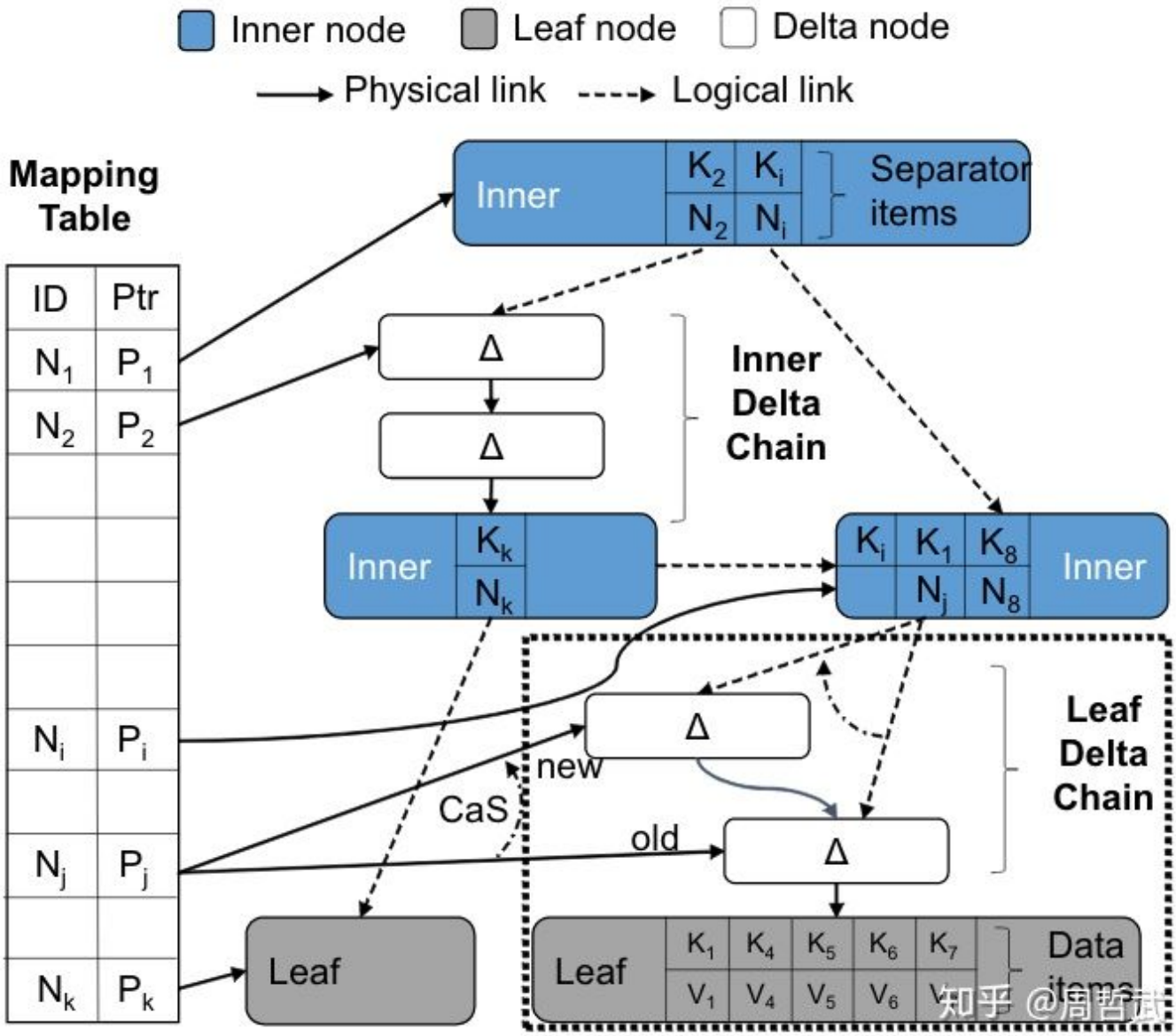
Hekaton 是微软 SQLServer 专门针对 OLTP 应用场景进行优化的数据库引擎，其索引实现基于 Bw-Tree。Bw-Tree 是一种无需使用任何闩同步的 B+Tree，其主要设计思想如下：

1. **Mapping Table（映射表）** 映射表存储内存页的ID与其对应的物理内存地址，使得线程可以通过访问映射表找到需要方位的内存地址，映射表的更新通过CAS操作。
2. **不直接修改节点**，任何的更新操作都会生成新的数据并通过指针指向被更新节点；新生成的数据所导致的元数据的修改，比如修改映射表都通过 CAS 完成。
3. **垃圾回收**，Bw-Tree 通过不断新增数据的方式避免直接修改树节点，在树不断更新的过程中，不可避免的会产生很多垃圾，因此 Bw-Tree 实现了基于 Epoch 的垃圾回收机制：当一个线程想保护一个它正在使用但是将会被回收的对象，例如检索的时候，访问了一个内存页，就把当前线程加入 Epoch，当这个线程完成检索页面的操作后，就会退出 Epoch。通常一个线程在一个epoch的时间间隔内完成一次操作，例如检索。在线程成功加入 Epoch 的时候，可能会看到将要被释放的老版本的对象，但不可能看到已经在前一个 Epoch 中释放的对象，因为其在当前 Epoch 中的操作并不依赖上一个 Epoch 中的数据。因此，一旦所有的线程成功加入Epoch 并完成操作然后退出这个Epoch，回收该 Epoch 中的所有对象是安全的。

由于维护了映射表，和新增数据链，因此树结构调整相对复杂，不仅仅要调整树，切要保证树结构和映射表之间的关系。具体操作可参考此篇文章。

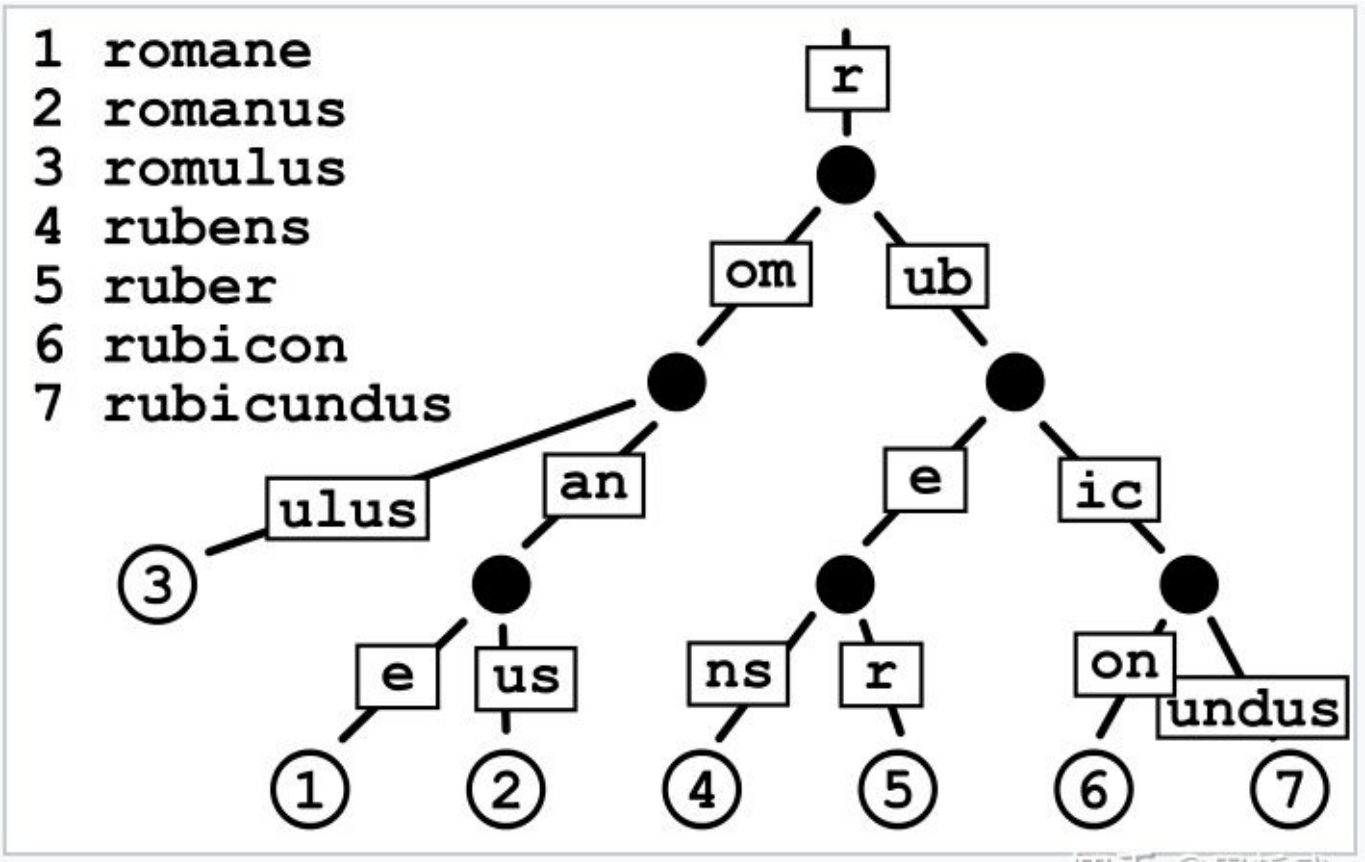
尽管实现非常复杂，Bw-Tree 作为无锁的数据库索引树，有如下优势：

- **无闩**: 实现无锁数据结构十分困难，Bw-Tree 在多线程场景下没有引入任何的闩，只使用 CAS 指令保证线程同步，因此多核的扩展性优于普通用闩同步的B+Tree。
- **CPU 缓存**： 由于不直接修改节点而是追加修改补丁，因此 CPU 缓存不会应为更新数据而失效，因此可以显著提高 CPU 缓存命中率。微软论文中的数据表明，90% 的读操作数据来自 CPU L1/L2 缓存。



Adaptive Radix Tree（自适应基数/前缀树，ART）

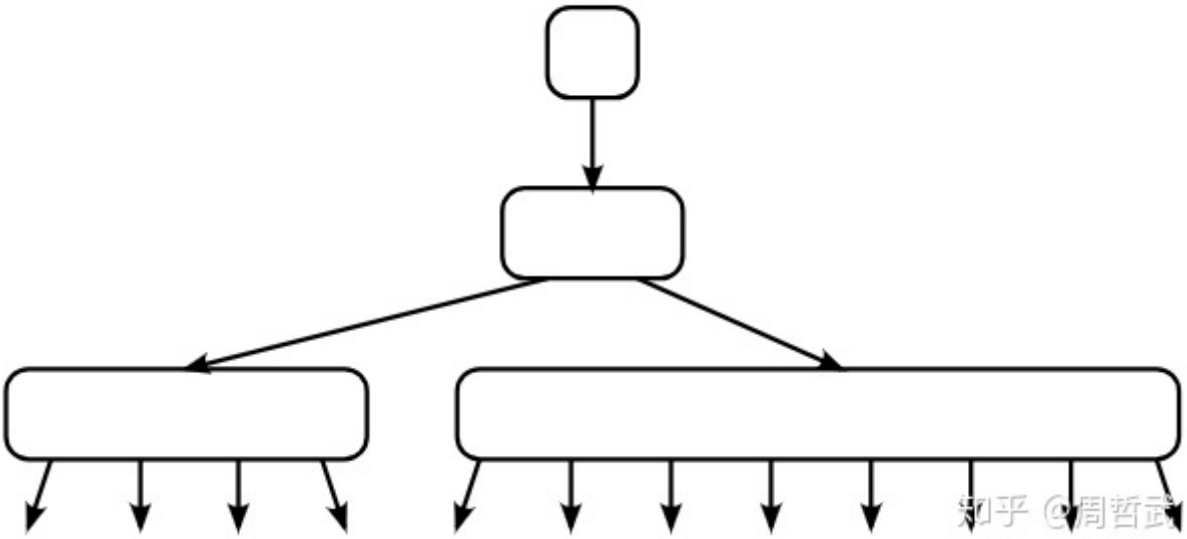
Radix Tree （基数树)是一种常见的前缀树，Linux Kernel 文件系统就用到了该数据结构：



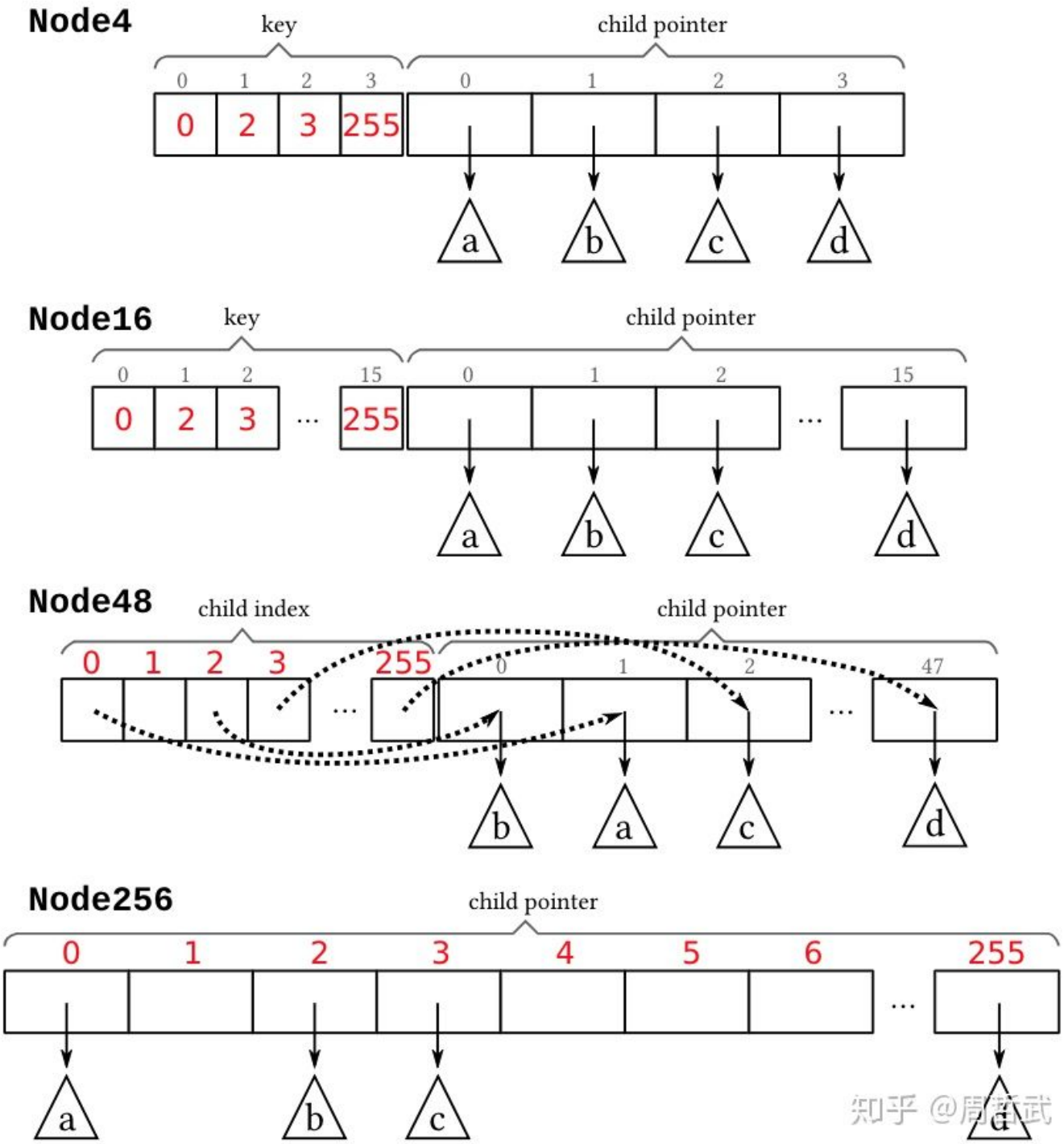
An example of a radix tree

[]Hyper 数据库]3中实现了 Adaptive Radix Tree （自适应基数/前缀树，ART）作为其索引。基数树的每个节点可以存储任意长度的键切片，比如 Linux Kernel 中的基数树每个节点存储 6位的键切片；然而数据库索引很多场景下会被频繁修改，每个节点固定长度的键切片会造成时间（切片过长）和空间上（切片过短）的浪费，因此，Hyper 实现了自适应的基数树，也就是节点根据长度的不同分成若干种，随着数据的变化而自行调整。

ART 结构：



ART 数据节点类型：



其主要特点有：

- 树的高度仅取决于键长度。
- 更新和删除不涉及到树结构的调整，不需要平衡操作。
- 到达叶子节点的路径就是键。
- 时间复杂度取决于键的长度，而跟数据量无关，如果数据的增加远远超过键长度的增加，那么使用 ART 将会在性能上带来非常大的收益。

讲述ART同步的论文中提供了描述了两种ART的同步机制：

1. 乐观锁：

- 读不阻塞写
- 写操作在获得对应的节点闩之后，更新版本信息
- 读操作在读下一个结点前，检查版本信息是否发生改变

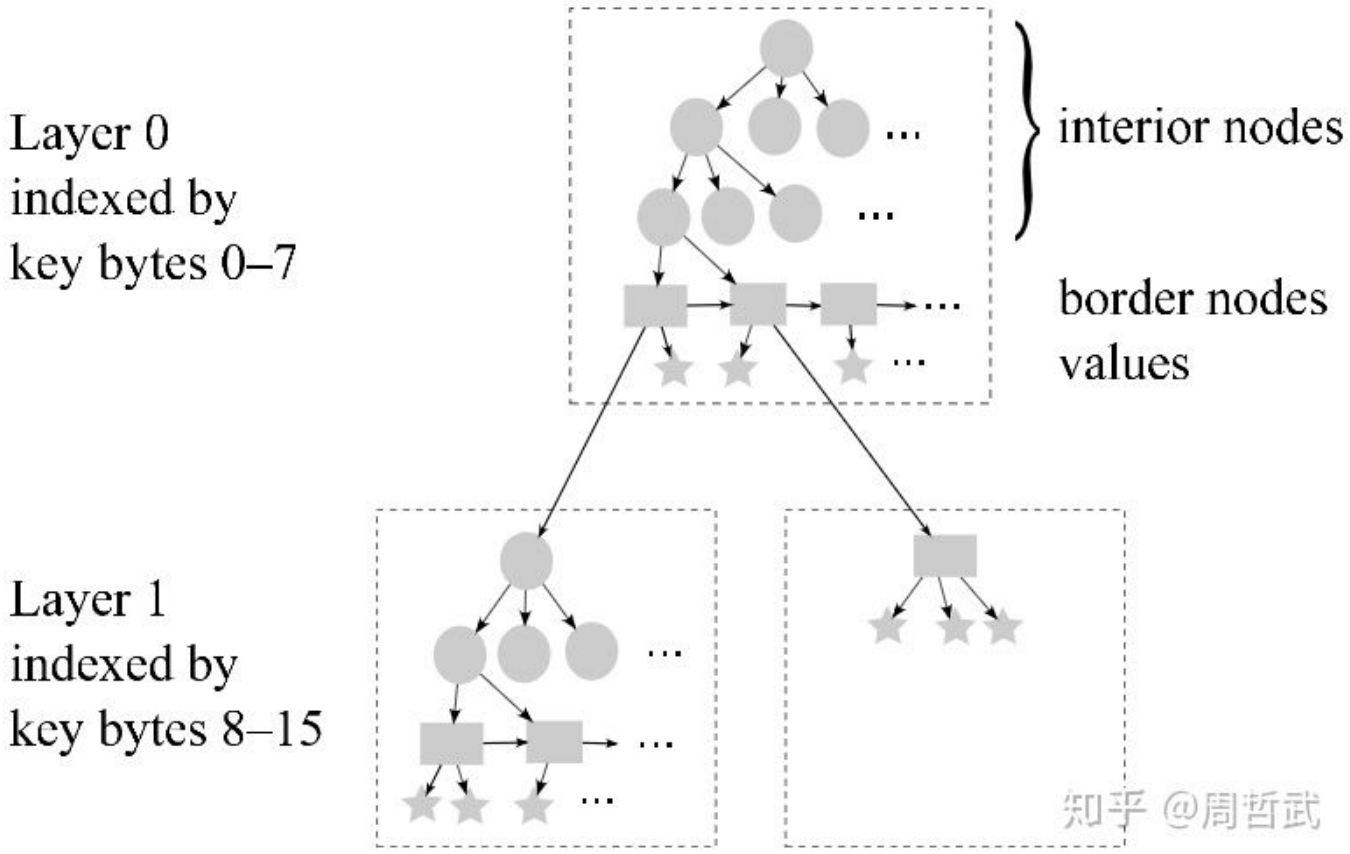
1. 乐观读悲观写

- 所有的节点都包含一个互斥锁，当某一个读操作获得此互斥锁之后，阻塞其他写操作
- 读操作不用获取任何的锁或者闩，也不用检查版本信息
- 写操作保证同一个节点读操作的数据一致性，即写操作使用原子指令进行写入

Masstree

2012年发表的论文 Cache craftiness for fast multicore key-value storage 提出了 Masstree，其特点如下：

- **可以理解为B+ Tree 和 Radix Tree 的混合体**，即将键切分成多个部分，每个部分为一个节点；每个节点内部又是一个 B+ Tree，兼顾空间和性能。



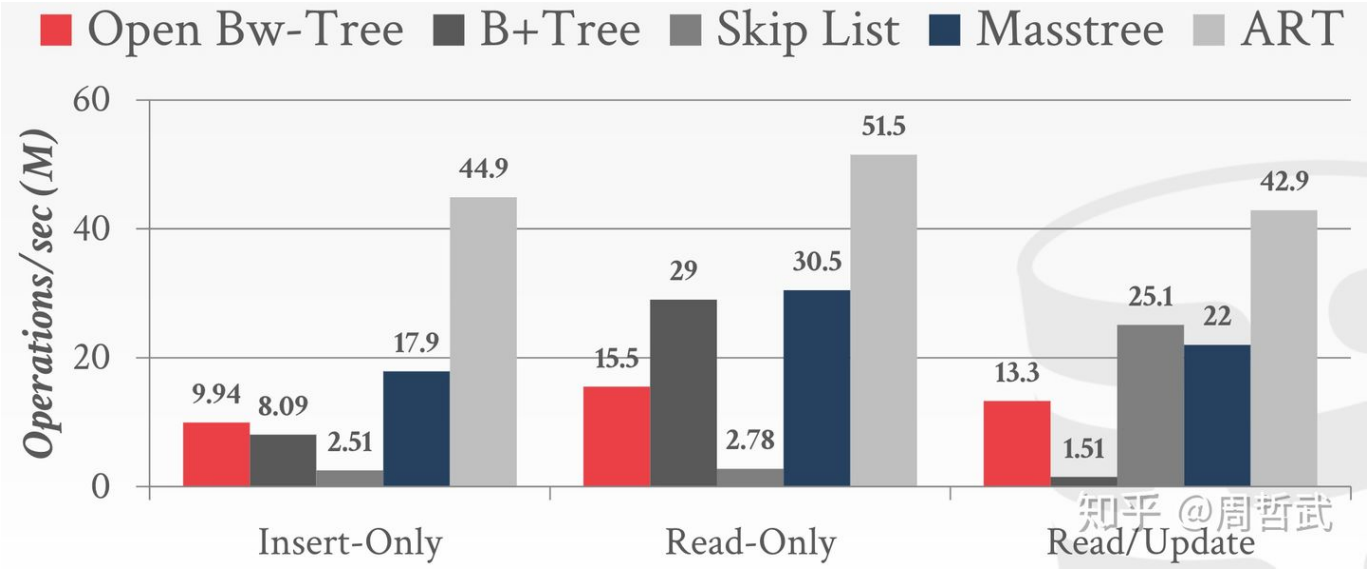
- Masstree将变长键划分成多个固长部分，每个固长部分可以通过int类型表示，而不是char类型。由于处理器处理int类型比较操作的速度远远快于char数组的比较，因此Masstree通

过int类型的比较进一步加速了查找过程。固定长度可以设置为 CPU 缓存行长度，以增加 CPU 缓存效率。

- 每个节点是一个 B+ Tree，因此 CPU 在查询的时候可以将节点所代表的B+ Tree 加载到 CPU 缓存中，以增加 CPU 缓存命中率。
- 其并发控制用到了Read-Copy-Update(RCU)。读不因任何数据更新而阻塞，但更新数据的时候，需要先复制一份副本，在副本上完成修改，再一次性地替换旧数据。因此读不会造成 CPU 缓存无效。

性能对比

上述几种索引数据结构性能对比如下：



作者的其他数据库文章链接

1. SQL：数据世界的通用语
2. 数据库性能之翼：SQL 语句运行时编译
3. 每周一论文：A Survey of B-Tree Locking Techniques
4. 每周一论文：An Empirical Evaluation of In-Memory Multi-Version Concurrency Control
5. 数据库索引数据结构总结