4.1选择性持久化

选择性持久化可以描述为在SCM中保留最小的主数据集，所有实现一致性的工作将集中在这些数据集上，并在恢复时重建放置在DRAM中的所有非主数据。如图3所示，应用于B+-Tree时，叶节点使用持久链表放在SCM中，而内部节点则放在DRAM中，只要叶处于一致状态，就可以重建内部节点。因此，与短暂的B+-树相比，只访问树叶的成本更高。此外，内部节点仅代表B+-树总大小的一小部分。因此，选择性持久化应该使我们的持久化树具有与瞬态B+-树类似的性能，而只使用最小一部分RAM。我们的方法与NV-Tree[28]的基本硬件假设不同：我们假设混合SCM-DRAM配置，而NV-Tree假设仅配置SCM。

简单地说，内部节点将保持经典结构并完全驻留在DRAM中，而叶节点将完全驻留在SCM中，并需要特别注意以确保它们的一致性。SCM和DRAM之间的这种相互作用对于我们在第4.4节中介绍的并发机制是至关重要的。

4.2指纹

未排序的叶子需要昂贵的线性扫描在SCM。为了实现更好的性能，我们提出了一种称为指纹的技术。指纹是叶键的单字节散列，连续存储在叶首，如图2所示。通过在搜索过程中首先扫描指纹，指纹起到了过滤器的作用，以避免探测指纹与搜索键的指纹不匹配的键。在下面的文章中，我们证明了，从理论上讲，指纹比wBTree和NV-Tree有更好的性能。我们只考虑唯一密钥的情况，这在实践中通常是一个可以接受的假设[15]。我们证明，使用指纹图，在一个成功的搜索过程中，叶内键探针的预期数量等于一个。在下面我们计算这个预期的数字。我们假设一个生成均匀分布的散列函数指纹。设m为叶中的条目数，n为可能的散列值(单字节指纹n=256)。

首先，我们计算指纹在叶中出现的预期次数(表示E[K])，这相当于指纹数组中的散列冲突数加1(因为我们假设存在搜索键)：

其中P[K=I]是指搜索指纹在知道至少出现一次的情况下出现的概率。设A和B为以下两个事件：

提名符反映的是二项分布，分母反映的是至少存在一个匹配指纹的概率，表示为不存在匹配指纹的互补概率。知道预期的指纹命中次数，我们可以确定叶内键探针的预期数(表示EFPTree[T])，这是在指纹命中指示的键上进行长度为E[K]的线性搜索时的键探针的预期数：

wBTree能够使用二进制搜索，这要归功于它排序的间接槽阵列，因此，它的叶内键探测的预期数量是：EwBTree[T]=log2(M)。NV-Tree通过执行从叶的最后一个条目开始的反向线性搜索来扫描该叶，以便在找到匹配的密钥时，确保它是最新的版本。然后，在对NV-Tree的搜索操作中，期望的叶内键探测数是线性搜索的数：Env-Tree[T]=1/2(m+1)。图4显示了FPTree、wBTree和NV-Tree的叶内键探针的预期数量。我们观察到FPTree在理论上比wBTree和NV-Tree有更好的性能。例如，对于m=32，FPTree需要单个叶内键探针，而wBTree和NV-Tree分别需要5和16。基本上，指纹识别所需的关键探针不到两个，平均可达m~400。wBTree的性能优于仅从m~4096开始的FPTree。需要注意的是，对于可变大小的键，因为只有键引用保存在叶节点中，所以每个键探测都会导致缓存丢失。因此，每个保存的叶内键探针对SCM来说都是一个已保存的缓存未命中。这一理论结果在我们的实验评估中得到了验证，如第6.2节所示。使用叶组可以减少昂贵的持久内存分配数量，从而提高插入性能，如第6.2节所示。

4.4选择性并发

事务内存是一种工具，它通过使用事务使一批写操作原子可见来实现并发。HTM是硬件支持的事务性内存。一些硬件供应商提供HTM，例如IBM on Blue Gene/Q和POWER8处理器，Intel提供事务同步扩展(TSX)。尽管我们在这项工作中使用了IntelTSX，但我们的算法对于任何当前可用的HTM实现都是有效的。从程序员的角度来看，HTM被用作围绕关键部分的粗粒度锁，但从硬件角度看，它的行为就像细粒度锁：在高速缓存线的粒度处检测到事务之间的冲突。对于TSX，关键代码放在使用XBEGIN和XEND指令的事务中。当线程到达XBEGIN指令时，首先读取相应的锁，如果可用，则启动事务而不获取锁。如果事务成功，也就是说，如果事务到达XEND指令而没有检测到任何冲突，则事务内的所有更改都通过原子提交以原子方式对其他线程可见。如果事务中止，则丢弃所有缓冲的更改，并按照程序员定义的回退机制重新执行操作。在我们的实现中，我们使用IntelThreadingBuildingBlock1推测旋转互斥，它使用全局锁作为后退机制。为了检测冲突，每个事务在L1缓存中保留读和写集。读集由事务读取的所有内存缓存行组成，写集由事务写入的所有内存缓存行组成。如果一个事务从另一个事务的写入集读取或写入另一个事务的读或写集，则硬件会检测到冲突。发生这种情况时，将中止两个事务中的一个。在这种情况下，中止的事务将退回到程序员定义的并发机制。为了提高性能，允许事务在使用回退机制之前重试几次。这是一个乐观的并发方案，因为它的工作假设是，只会发生很少的冲突，并且事务将以很高的概率执行无锁的事务。HTM是通过使用L1缓存监视更改在当前体系结构中实现的。因此，影响缓存的CPU指令(例如CLFLUSH)被检测为冲突，如果事务在其读或写集中执行，则会触发事务的终止。标准B+-Tree基于HTM的锁省略的一个简单实现就是在HTM事务[17]中执行它的基本操作。但是，INSERT和DELETE操作需要刷新叶中修改过的记录，从而中止事务并获取全局锁。这意味着实际上所有的插入和删除操作都将被序列化。因此，在使用HTM和我们需要将数据刷新到持久内存以确保一致性之间存在明显的不兼容性。为了解决这一问题，我们建议对数据的瞬态部分和持久性部分使用不同的并发方案，同样地，对存储在DRAM和SCM中的数据采用不同的一致性方案。我们将我们的方法命名为选择性并发。

5.

在本工作中，我们实现了三种不同的持久树：

1。FPTREE。它是实现选择性持久性、指纹、摊销分配和未排序叶子的单线程版本。

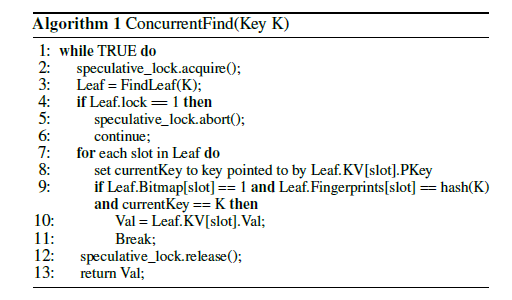
2。并发FPTree。这个版本实现了选择性持久化、选择性并发性、指纹和未排序的叶子。至于摊销分配，因为它们构成了一个中央同步。

在这一点上，我们发现它们阻碍了可伸缩性。因此，在此版本中不使用它们。

3。PTree。它反映了仅实现选择性持久性和未排序树叶的FPTree的轻型版本。与FPTree和wBTree相反，它在线性扫描键时将键和值保存在单独的数组中，以获得更好的数据局部性。在接下来的文章中，我们将讨论具有固定大小密钥的FPTree的基本操作，并解释它如何在一致状态下从任何软件崩溃或断电情况中恢复。附录C中对可变大小的键也有类似的讨论。此外，在附录B中解释了管理FPTree单线程版本中的叶组的算法。在下面的示例中，投机锁表示启用TSX的锁。由于叶锁仅在TSX事务中使用，因此不需要使用Atomics对其进行修改。实际上，如果许多线程试图写入相同的锁-从而写入相同的高速缓存行-则只有一个线程会成功，而其他线程将被中止。

FIND

由于搜索操作不会修改持久数据，因此可以将它们完全包装在TSX事务中，以防止它们与其他线程的写入操作发生冲突。如果另一个线程写入执行查找的线程读取的位置，则事务将中止并重试，如果超过重试阈值，事务将最终接受全局锁定。搜索操作仍然需要检查其他线程在TSX事务之外执行更改时可能采取的任何非TSX锁(即叶锁)。算法1显示查找操作的伪代码。并发查找将一直执行到成功为止，只要中止，就会重试，这要么是因为目标叶被锁定，要么是因为检测到冲突。投机锁可以通过两种方式执行：只要TSX重试阈值未达到，就使用XBEGIN指令启动常规TSX事务，或者采用全局锁。



事务中止的方式取决于当前情况：如果启动了常规TSX事务，XABORT指令将撤消所有更改(此处不相关，因为未进行任何更改)，然后倒带到XBEGIN指令。如果采用全局锁，则XABORT指令无效。在这种情况下，WHILE循环和Continue指令起到中止和重试的作用。在到达叶时，如果它的锁尚未被获取，则执行密钥查找，而不使用叶锁。如果另一个线程在搜索密钥时获取了叶锁，则检测到冲突并中止两个事务中的一个。

INSERT

算法2给出了并发插入操作的伪代码。它分为三个步骤：(1)在TSX事务中，遍历树以到达叶并将其锁定，并传播是否需要拆分的信息；(2)在TSX事务之外，应用对插入叶的更改：如果需要，将进行叶拆分，并在叶中插入密钥值对。这些操作使用持久性原语，如果在TSX事务中使用，将触发中止；(3)如果在步骤2中发生叶拆分，则在第二个TSX事务中更新内部节点。这可以通过两种方式完成：一种是直接更新叶的父节点(如果它在修改叶时没有拆分)，另一种是通过重新遍历树。然后通过将叶锁设置为0来释放该叶锁。可以注意到，在算法2中，由于内部节点没有锁，所以第二个事务不使用While循环。因此，无需手动中止事务。当不需要叶分割时，密钥值对和指纹将被写入各自的插槽并保持不变。这些写入的顺序并不重要，因为只要位图未更新，它们就不可见。因此，在写入键和值时发生崩溃的情况下，不需要执行任何操作，操作将被视为未完成。然后，以p原子方式更新并持久化该位图。简而言之，如果在持久化位图之前发生故障，则不会插入键值对，否则，操作将成功执行。在这两种情况下，都不需要采取任何行动。算法3表示叶分割的伪代码。为了确保树在拆分过程中出现故障时的一致性，我们需要使用一个微型日志，该日志由两个持久性指针组成：一个指向要拆分的叶，表示PCurrent叶子，另一个指向新分配的叶，表示PNew叶子。基本上，并发FPTree包含由瞬态无锁队列索引的拆分和删除微日志阵列。微日志由无锁队列提供，并在操作结束时返回。分裂恢复函数如算法4所示。拆分操作首先将要拆分的叶的持久地址写入PCurrentLeave。如果此时发生故障，我们只需要重置微日志，因为PNewLeave仍然为NULL。然后，我们分配一个新的叶，并向分配器提供一个对PNewLeave的引用，分配器在返回之前将分配到的内存的地址保存在该分配器中。如果此时发生崩溃，则Split Recovery函数将检查PNewLeave是否为NULL。如果是，则重置微日志并返回。否则，它将检测到分配已完成，从而继续拆分操作。此后，裂叶的内容持续复制到新叶中。然后确定新的密钥鉴别器(分割密钥)，并相应地更新新叶的位图。如果在后两个步骤中发生崩溃，恢复函数只需在检测到PNewLeave不为NULL时重新执行它们。随后，将更新分割叶的位图，并将分割叶的下一个指针设置为指向新叶。后一种写入不需要是p原子的，因为在此时间点出现故障时，恢复函数将从复制阶段开始重做拆分。最后，重置微日志。

