第 08 讲:树索引

15-445/645 数据库系统(秋季 2022)

https://15445.courses.cs.cmu.edu/fall2022/

卡内基梅隆大学安迪·帕夫洛

1 表索引

在数据库系统内部可以使用许多不同的数据结构，例如内部元数据、核心数据存储、临时数据结构或表索引。对于可能涉及范围扫描查询的表索引，

*表索引*是表列子集的副本，它被组织和/或排序，以便使用这些属性的子集进行有效访问。因此，DBMS 可以查找表索引的辅助数据结构来更快地找到元组，而不是执行顺序扫描。DBMS 确保表和索引的内容在逻辑上始终保持同步。

在每个数据库创建索引的数量之间存在权衡。虽然更多的索引使查找查询更快，但索引也使用存储并且需要维护。找出用于执行查询的最佳索引是 DBMS 的工作。

2 B+树

B+Tree 是一种自平衡树数据结构，它保持数据排序，并允许在 O(log(n))内进行搜索、顺序访问、插入和删除。它针对读/写大数据块的面向磁盘的 DBMS 进行了优化。

几乎所有支持保序索引的现代 DBMS 都使用 B+树。有一种特定的数据结构称为 *B-树*，但人们也用这个术语泛指一类数据结构。原始 *B-树和 B+树*的主要区别在于 B-树在*所有节点*中存储键和值，而 B+树 *仅在叶节点中*存储值。现代 B+Tree 实现结合了其他 B-树变体的特性，例如 Blink-Tree 中使用的兄弟指针。

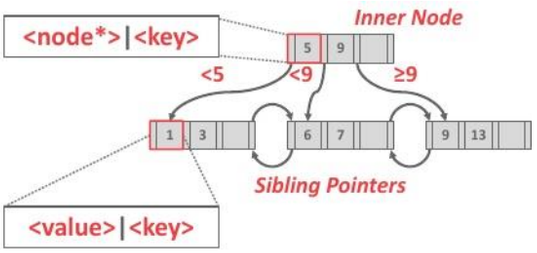

图 1:B+ Tree 图

正式地说，B+树是一个 M-way 搜索树(其中 M 表示一个节点可以拥有的子节点的最大数量)，具有以下属性:

- 它是完美平衡的(即，每个叶节点都在相同的深度)。
- 除根节点外的每个内部节点至少为半满(M/2−1 <= num of keys <= M−1)。
- 每个有 k 个键的内部节点有 k+1 个非空子节点。

B+Tree 中的每个节点都包含一个键/值对数组。这些键/值对中的键来自索引所基于的属性。这些值将根据节点是内部节点还是叶节点而有所不同。对于内部节点，值数组将包含指向其他节点的指针。获取叶节点值的两种方法是*记录 id* 和*元组数据*。记录 id 引用指向元组位置的指针。拥有元组数据的叶节点将元组的实际内容存储在每个节点中。

虽然根据 B+树的定义，这不是必须的，但每个节点上的数组几乎总是按键排序的。

从概念上讲，内部节点上的键可以被认为是导柱。它们引导树遍历，但不表示叶节点上的键(以及它们的值)。这意味着你可能在一个内部节点(作为一个引导柱)中有一个键，而这个键在叶节点上是找不到的。尽管必须注意的是，传统上，内部节点只拥有那些存在于叶节点中的键。

插入

要在 B+树中插入一个新条目，必须遍历树并使用内部节点来找出将键插入到哪个叶节点中。

1. 找到正确的叶子 L。
2. 按排序顺序向 L 中添加新条目:
    - 如果 L 有足够的空间，则操作完成。
    - 否则将 L 拆分为两个节点 L 和 L2。均匀地重新分配条目并向上复制中键。将指向 L2 的索引项插入到 L 的父节点中。
3. 要拆分内部节点，要均匀地重新分配条目，但要向上推中间键。

删除

然而在插入中，当树变得太满时，我们偶尔不得不拆分叶子，如果删除导致树少于一半满，我们必须合并以重新平衡树。

1. 找到正确的叶子 L。
2. 删除条目:
    - 如果 L 至少为半满，则操作结束。
    - 否则，你可以尝试重新分配，借用兄弟姐妹。
    - 如果再分配失败，合并 L 和 sibling。
3. 如果发生合并，必须删除 parent 中指向 L 的条目。

选择条件

因为 B+树是有序的，查找有快速的遍历，也不需要整个键。如果查询提供了搜索键的任何属性，DBMS 可以使用 B+Tree 索引。这与哈希索引不同，哈希索引需要搜索键中的所有属性。

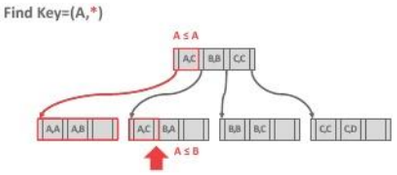

图 2:要在 B+Tree 上执行前缀搜索，查看键上的第一个属性，沿着路径向下，并在叶子上执行顺序扫描，以找到所需的所有键。

非唯一索引

与哈希表一样，B+树可以通过复制键或存储值列表来处理非唯一索引。在重复键方法中，使用相同的叶节点布局，但重复键被存储多次。在值列表方法中，每个键只存储一次，并维护一个唯一值的链表。

复制键

在 B+Tree 中有两种复制键的方法。

第一种方法是*附加记录 id* 作为键的一部分。由于每个元组的记录 ID 是唯一的，这将确保所有的键都是可识别的。

第二种方法是允许叶子节点*溢出到包含重复键的溢出节点*中。虽然不存储冗余信息，但这种方法维护和修改起来比较复杂。

聚集索引

表按照主键指定的排序顺序存储，作为堆组织存储或索引组织存储。由于一些 dbms 总是使用聚集索引，如果一个表没有显式索引，它们将自动创建一个隐藏的行 id 主键，但其他的根本不能使用它们。

堆集群

元组使用聚类索引指定的顺序在堆的页面中排序。如果使用集群索引的属性访问元组，DBMS 可以直接跳转到页面。

索引扫描页面排序

由于直接从未聚类索引中检索元组的效率很低，因此 DBMS 可以首先找出它需要的所有元组，然后根据它们的页面 id 对它们进行排序。

3 B+树的设计选择

3.1 节点大小

根据存储介质的不同，我们可能喜欢更大或更小的节点大小。例如，存储在硬盘上的节点通常以兆字节为数量级，以减少查找数据所需的寻道次数，并将昂贵的磁盘读取摊销到一大块数据上，而内存数据库可能会使用小至 512 字节的页面大小，以便将整个页面放入 CPU 缓存中，同时减少数据碎片。这种选择也可能取决于工作负载的类型，因为点查询更喜欢尽可能小的页面，以减少不必要的额外信息加载量，而大型顺序扫描可能更喜欢大页面，以减少它需要做的读取次数。

3.2 合并阈值

虽然 B+树有一个关于在删除后合并下溢节点的规则，但有时暂时违反该规则以减少删除操作的数量可能是有益的。例如，急切合并可能导致震荡，其中大量连续的删除和插入操作导致不断的拆分和合并。它还允许批处理合并，其中多个合并操作同时发生，减少了必须在树上采取昂贵的写锁存器的时间。

3.3 可变长度键

目前我们只讨论了具有固定长度键的 B+树。然而，我们可能也希望支持可变长度键，比如大键的小子集导致大量空间浪费的情况。有几种方法可以解决这个问题:

1. **指针**

   而不是直接存储键，我们可以只是存储一个指向键的指针。由于每个键都要追踪一个指针的效率低下，所以在生产中唯一使用这种方法的地方是嵌入式设备，其微小的寄存器和缓存可能会受益于这种节省空间的方式

2. **变长节点**

   我们也可以像平常一样存储键，并允许可变长度的节点。这是不可行的，很大程度上没有使用，因为处理可变长度节点的内存管理开销很大。

3. **填充**

   我们可以将每个键的大小设置为最大键的大小，而不是改变键的大小，并填充所有较短的键。在大多数情况下，这是对内存的巨大浪费，所以你也不会看到有人使用这种方法。

4. **键映射表/间接**

   几乎每个人都使用的方法是在单独的字典中用键值对的索引替换键。这提供了显著的空间节省和潜在的快捷点查询(因为索引指向的键值对与叶节点指向的键值对完全相同)。由于字典索引值的大小很小，有足够的空间在索引旁边放置每个键的前缀，潜在地允许一些索引搜索和叶子扫描甚至不必追踪指针(如果前缀与搜索键完全不同)。

3.4 节点内搜索

到达节点后，我们仍然需要在节点内进行搜索(从内部节点查找下一个节点，或者在叶节点中查找键值)。虽然这相对简单，但仍然需要考虑一些权衡:

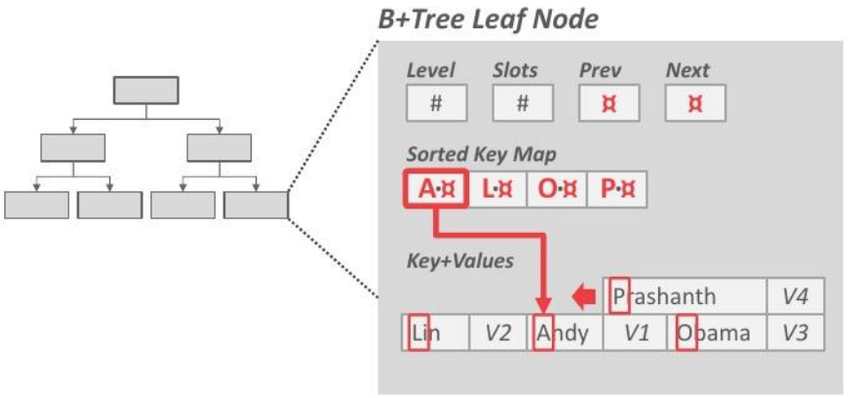

图 3: Key Map/Indirection 的一个例子。map 存储了键的一个小前缀，以及一个指向键值对的指针。

1.线性

最简单的解决方案是扫描节点中的每个键，直到找到我们的键。一方面，我们不必担心对键进行排序，使得插入和删除要快得多。另一方面，这相对效率较低，每次搜索的复杂度为 O(n)。这可以使用 SIMD(或等效)指令进行矢量化。

2.二进制

更有效的搜索解决方案是保持每个节点排序，并使用二分搜索来查找键。这就像跳到一个节点的中间，根据键之间的比较向左或向右旋转一样简单。这种方式的搜索效率要高得多，因为这种方法每次搜索的复杂度只有 O(ln(n))。然而，插入变得更加昂贵，因为我们必须维护每个节点的排序。

3.插值

最后，在某些情况下，我们可能能够利用插值来找到键。这种方法利用存储在节点上的任何元数据(如 max 元素、min 元素、平均值等)，并使用它来生成键的近似位置。例如，如果我们在一个节点中寻找 8，我们知道 10 是最大键，10 - (n + 1)是最小键(其中 n 是每个节点中的键数)，那么我们知道从最大键开始搜索 2 个槽，因为在这种情况下，距离最大键一个槽的键必须是 9。尽管是我们给出的最快的方法，但由于其对具有某些属性(如整数)和复杂性的键的有限适用性，该方法仅在学术数据库中可见。

4 优化

4.1 指针切换

因为 B+Tree 的每个节点都存储在缓冲池中的一个页面中，所以每次我们加载一个新页面时都需要从缓冲池中取出它，这需要锁存和查找。为了完全跳过这一步，我们可以将实际的原始指针存储在页 id 的位置(称为“swizzling”)，从而完全防止缓冲池的抓取。而不是手动获取整个树并手动放置指针，我们可以在正常遍历索引时简单地存储页面查找的结果指针。注意，我们必须跟踪哪些指针被“swizzle”，并在它们指向的页面被解钉和受害时，将它们解糊弄回页面 id。

4.2 批量插入

当最初构建 B+Tree 时，必须以通常的方式插入每个键将导致不断的拆分操作。由于我们已经给了叶子兄弟指针，如果我们构造一个叶节点的排序链表，然后使用每个叶节点的第一个键从下往上轻松地构建索引，那么初始数据插入的效率要高得多。请注意，根据我们的上下文，我们可能希望尽可能紧密地打包叶节点以节省空间，或者在每个叶节点中留出空间，以便在必要的拆分之前允许更多的插入。

4.3 前缀压缩

大多数情况下，当我们在同一个节点上有键时，每个键的某些前缀会有部分重叠(因为在排序的 B+树中，相似的键最终会紧挨着彼此)。而不是将这个前缀作为每个键的一部分存储多次，我们可以简单地在节点的开头存储一次前缀，然后只包括每个槽中每个键的唯一部分。

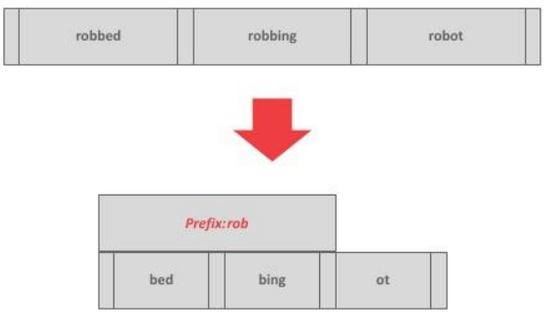

图 4:前缀压缩的一个例子。由于键是按字典顺序排列的，所以它们很可能共享一些前缀。

4.4 重复数据删除

在索引允许非唯一键的情况下，我们最终可能会得到叶节点一遍又一遍地包含相同的键，只是附加了不同的值。一种优化方法是只写一次键，然后用它的所有相关值跟随它。

4.5 后缀截断

在大多数情况下，内部节点中的键项只是用作路标，而不是用于它们实际的键值(因为即使索引中存在键，我们仍然需要搜索到底部以确保它没有被删除)。

我们可以利用这一点，只存储将探测正确路由到正确节点所需的最小前缀。