第 07 讲:哈希表

15-445/645 数据库系统(2022 秋季)

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

卡内基梅隆大学安迪·帕夫洛

1 数据结构

DBMS 对系统内部的许多不同部分使用不同的数据结构。一些例子包括:

- **内部元数据:**这是跟踪有关数据库和系统状态的信息的数据。

  例如:页表，页目录

- **核心数据存储:**数据结构被用作数据库中元组的基础存储。
- **临时数据结构:**DBMS 可以在处理查询时动态构建数据结构以加快执行速度(例如，连接的哈希表)。
- **表索引:**辅助数据结构可以用来更容易地找到特定的元组。

在为DBMS 实现数据结构时，有两个主要的设计决策需要考虑:

1.数据组织:为了支持高效的访问，我们需要弄清楚如何布局内存，以及在数据结构中存储什么信息。

2.并发性:我们还需要考虑如何在不产生问题的情况下使多个线程能够访问数据结构。

2 哈希表

哈希表实现了将键映射到值的关联数组抽象数据类型。它提供了平均 O(1)的操作复杂度(最坏情况下为 O(n))和 O (n)的存储复杂度。请注意，即使平均为 O(1)的操作复杂度，在现实世界中也有需要考虑的常数因子优化。

哈希表的实现由两部分组成:

- **哈希函数:**这告诉我们如何将一个大的键空间映射到一个较小的域。它用于计算桶或槽数组的索引。我们需要考虑快速执行和碰撞率之间的权衡。在一个极端情况下，我们有一个总是返回常量的哈希函数(非常快，但一切都是碰撞)。在另一个极端，我们有一个“完美”的哈希函数，没有碰撞，但计算时间会非常长。理想的设计是介于两者之间。
- **哈希方案:**这告诉我们如何处理哈希后的键冲突。在这里，我们需要考虑分配一个大哈希表来减少冲突和在发生冲突时必须执行额外指令之间的权衡。

3 哈希函数

*哈希函数*接受任意键作为输入。然后它返回该键的整数表示(即“哈希”)。该函数的输出是确定性的(即，相同的键应该总是产生相同的哈希输出)。

DBMS 不需要使用加密安全哈希函数(例如 SHA-256)，因为我们不需要担心保护密钥的内容。这些散列函数主要在 DBMS 内部使用，因此信息不会泄露到系统外部。一般来说，我们只关心哈希函数的速度和碰撞率。

目前最先进的哈希函数是 Facebook 的 XXHash3。

4 静态哈希方案

静态哈希方案是指哈希表的大小是固定的。这意味着，如果 DBMS 耗尽了哈希表中的存储空间，那么它必须从头开始重建一个更大的哈希表，这是非常昂贵的。通常，新哈希表的大小是原哈希表的两倍。

为了减少浪费的比较次数，避免哈希键的碰撞是很重要的。通常，我们使用两倍的槽数作为期望的元素数。

下面的假设在现实中通常是不成立的:

1.元素的数量是提前知道的。

2.钥匙是独一无二的。

3.存在一个完美的哈希函数。

因此，我们需要适当地选择哈希函数和哈希模式。

4.1 线性探测哈希

这是最基本的哈希方案。它通常也是最快的。它使用数组槽的圆形缓冲区。哈希函数将键映射到槽。当发生碰撞时，我们线性搜索相邻的槽，直到找到一个开放的槽。对于查找，我们可以检查键哈希到的槽，并线性搜索，直到找到所需的条目。如果我们找到了一个空槽，或者我们遍历了哈希表中的每个槽，那么这个键就不在表中。请注意，这意味着我们必须将键和值都存储在槽中，以便我们可以检查条目是否是所需的条目。删除就比较棘手了。我们必须小心地从槽中删除条目，因为这可能会阻止以后的查找找到已经放在现在空槽下面的条目。这个问题有两种解决方案:

- 最常见的方法是使用“墓碑”。而不是删除条目，我们用一个“墓碑”条目来代替它，它告诉以后的查找继续扫描。
- 另一种选择是在删除条目后移动相邻的数据来填充现在的空槽。然而，我们必须小心，只移动那些最初被移动的数据。这在实践中很少实现。

非唯一键:在同一个键可能与多个不同的值或元组相关联的情况下，有两种方法。

- 独立链表:我们不是用键来存储值，而是存储一个指向单独存储区域的指针，该存储区域包含所有值的链表。
- 冗余键:更常见的方法是简单地在表中多次存储相同的键。即使我们这样做，所有带有线性探测的东西仍然可以工作。

4.2 罗宾汉哈希

这是线性探测哈希的扩展，旨在减少每个键在哈希表中的最佳位置(即它们被哈希到的原始槽)的最大距离。这种策略从“富”键中窃取槽，并将其提供给“穷”键。

在这种变体中，每个条目还记录了它们与最优位置的“距离”。然后，在每次插入时，如果被插入的键离它们在当前槽的最佳位置的距离比当前表项的距离更远，我们就替换当前表项，并继续尝试将旧表项插入表中更远的位置。

4.3 布谷鸟哈希法

这种方法不使用单个哈希表，而是用不同的哈希函数维护多个哈希表。哈希函数是相同的算法(例如，XXHash, CityHash);它们通过使用不同的种子值，为同一个键生成不同的哈希值。

当我们插入时，我们检查每个表并选择一个有空闲槽的表(如果多个有一个，我们可以比较诸如负载因子之类的东西，或者更常见的是，只是随机选择一个表)。如果没有表有空闲槽，我们选择(通常是随机的)并驱逐旧的表项。然后我们将旧的表项重新散列到另一个表中。在极少数情况下，我们可能会进入一个循环。如果发生这种情况，我们可以用新的哈希函数种子(不太常见)重建所有的哈希表，或者使用更大的表(更常见)重建哈希表。

布谷鸟哈希保证了 O(1)次查找和删除，但插入可能会更昂贵。

5 动态哈希方案

静态散列方案要求 DBMS 知道它想要存储的元素的数量。否则，如果需要增加/缩小大小，它必须重新构建表。

动态哈希方案能够根据需要调整哈希表的大小，而不需要重建整个表。这些方案以不同的方式执行这种大小调整，既可以最大化读，也可以最大化写。

5.1 链式哈希

这是最常见的动态哈希方案。DBMS 为哈希表中的每个槽维护一个桶的链表。散列到同一槽的键被简单地插入到该槽的链表中。

5.2 可扩展哈希

链式哈希的改进变体，可以拆分桶而不是让链永远增长。这种方法允许哈希表中的多个槽位置指向同一个桶链。

重新平衡哈希表背后的核心思想是在 split 上移动桶条目，并增加要检查的位数以查找哈希表中的条目。这意味着 DBMS 只需要在分割链的桶内移动数据;其他所有的桶都保持不变。

- DBMS 维护全局和本地深度位计数，确定在槽数组中查找桶所需的位数。
- 当一个桶已满时，DBMS 将桶拆分并重新洗牌其元素。如果分割桶的局部深度小于全局深度，那么新桶只是添加到现有的槽数组中。否则，DBMS 将槽数组的大小加倍以容纳新桶，并增加全局深度计数器。

5.3 线性哈希

这种方案不是在桶溢出时立即拆分桶，而是维护一个*拆分指针*，跟踪下一个要拆分的桶。无论这个指针是否指向溢出的存储桶，DBMS 总是分裂。溢出标准留给实现。

- 当任何桶溢出时，通过添加一个新的槽项在指针位置拆分桶，并创建一个新的哈希函数。
- 如果哈希函数映射到先前被指针指向的槽位，则应用新的哈希函数。
- 当指针到达最后一个槽时，删除原来的哈希函数，用新的哈希函数代替。