有道翻译 支持免费文档翻译240万字符/年 立即下载→ https://fanyi.youdao.com/download

讲座#09:索引并发控制

15-445/645 数据库系统(2022 年秋季)

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

卡内基梅隆大学安迪·帕夫洛

1 指标并发控制

到目前为止，我们假设我们讨论的数据结构是单线程的。然而，大多数 dbms 需要允许多个线程安全地访问数据结构，以利用额外的 CPU 内核并隐藏磁盘 I/O 停顿。

*并发控制*协议是 DBMS 用来确保共享对象上并发操作的“正确”结果的方法。

一个协议的正确性标准可以不同:

- **逻辑正确性:**这意味着线程能够读它应该读的值，例如，线程应该读回它之前写的值。
- **物理正确性:**这意味着对象的内部表示是可靠的，例如，在数据结构中没有指针会导致线程读取无效的内存位置。

出于这堂课的目的，我们只关心强制执行物理正确性。我们将在后面的讲座中重新讨论逻辑正确性。

2 锁与闩锁

在讨论 DBMS 如何保护其内部元素时，锁和闩锁之间有一个重要的区别。

锁

锁是一种高级的逻辑原语，它保护数据库的内容(例如，元组、表、数据库)不受其他事务的影响。事务将在其整个持续时间内保持锁。数据库系统可以向用户公开查询运行时所持有的锁。锁需要能够回滚更改。

闩锁

锁存器是低级保护原语，用于保护 DBMS 内部数据结构(例如，数据结构，内存区域)的关键部分免受其他线程的影响。锁存器仅在操作发生时保持。锁存器不需要能够回滚更改。锁存器有两种模式:

- **READ:**允许多个线程同时读取同一项。一个线程可以获得读取锁存器，即使另一个线程也获得了它。
- **WRITE:**只允许一个线程访问该项目。一个线程不能获得一个写锁存器，如果另一个线程在任何模式下持有这个锁存器。持有写锁存器的线程也阻止其他线程获得读锁存器。

3 闩锁实现

用于实现锁存的底层原语是通过现代 cpu 提供的*原子比较与交换*(CAS)指令实现的。有了这个，一个线程可以检查内存位置的内容，看看它是否有某个值。如果有，那么 CPU 会用一个新的值来交换旧的值。否则，内存位置保持不变。

有几种方法可以在 DBMS 中实现闩锁。每种方法在工程复杂性和运行时性能方面都有不同的权衡。这些测试和设置步骤是原子执行的(即，没有其他线程可以更新测试和设置步骤之间的值。

阻塞 OS 互斥锁

锁存器的一种可能实现是 OS 内置互斥锁基础设施。Linux 提供了 futex(快速用户空间互斥锁)，它由(1)用户空间中的自旋锁存器和(2)操作系统级互斥锁组成。如果 DBMS 可以获得用户空间锁存，则设置锁存。它显示为 DBMS 的单个锁存器，尽管它包含两个内部锁存器。如果 DBMS 无法获取用户空间锁存，那么它就会进入内核，并尝试获取一个更昂贵的互斥锁。如果 DBMS 未能获得第二个互斥锁，那么线程通知 OS 它在互斥锁上被阻塞，然后它被重新调度。

OS 互斥锁在 dbms 中通常不是一个好主意，因为它是由 OS 管理的，并且开销很大。

- **例子:**std::互斥锁

  优点:使用简单，不需要额外的 DBMS 编码。

  缺点:由于 OS 调度，成本高且不可扩展(每次锁/解锁调用约 25 ns)。

Test-and-Set Spin Latch (TAS)

自旋锁存器是 OS 互斥锁的一种更有效的替代方案，因为它由 dbms 控制。自旋锁存器本质上是线程试图更新的内存位置(例如，将布尔值设置为 true)。线程执行 CAS 以尝试更新内存位置。DBMS 可以控制如果无法获得锁存会发生什么。它可以选择再次尝试(例如，使用 while 循环)或允许 OS 重新调度它。因此，与 OS 互斥锁相比，这种方法为 DBMS 提供了更多的控制权，在 OS 中，无法获得锁存将控制权交给操作系统。

- **例子:**std::原子< T >
- **优点:**Latch/unlatch 操作高效(单个指令锁定/解锁)。
- **缺点:**不可扩展，也不支持缓存，因为使用多线程时，CAS 指令将在不同的线程中多次执行。这些浪费的指令会在高竞争环境中堆积起来;这些线程在操作系统看来很忙，即使它们没有做有用的工作。这会导致缓存一致性问题，因为线程正在轮询其他 cpu 上的缓存行。

读写器闩锁

互斥锁和自旋闩锁不区分读/写(也就是说，它们不支持不同的模式)。DBMS 需要一种允许并发读取的方法，所以如果应用程序有大量的读取，它将有更好的性能，因为读取器可以共享资源而不是等待。

读写器锁存器允许锁存器保持在读或写模式。它跟踪在每种模式下持有锁存器并等待获得锁存器的线程数。读写器锁存器使用前面两个锁存器实现中的一个作为基元，并有额外的逻辑来处理读写器队列，

它们是在每种模式下对锁存器的队列请求。不同的 dbms 对于如何处理队列可以有不同的策略。

示例:std::共享互斥量

优点:允许并发读取。

- **缺点:**DBMS 必须管理读/写队列以避免饥饿。由于额外的元数据，比 spin 闩锁有更大的存储开销。

4 .哈希表锁存

由于线程访问数据结构的方式有限，在静态哈希表中很容易支持并发访问。例如，当从槽移动到下一个槽时，所有线程都在同一个方向移动(即自顶向下)。线程在同一时间也只访问一个页面/槽。因此，死锁在这种情况下是不可能的，因为没有两个线程可以竞争另一个线程持有的锁存器。当我们需要调整表的大小时，我们可以只在整个表上取一个全局闩锁来执行操作。

动态哈希方案(例如，可扩展)中的闩锁是一个更复杂的方案，因为有更多的共享状态需要更新，但一般方法是相同的。

在哈希表中支持闩锁的方法有两种，它们在闩锁的粒度上不同:

- **页面锁存:**每个页面都有自己的读写锁存，保护其整个内容。线程在访问页之前获得读锁存或写锁存。这降低了并行性，因为可能一次只有一个线程可以访问一页，但访问一页中的多个槽对单个线程来说会很快，因为它只需要获得一个锁存器。
- **槽位锁存:**每个槽位都有自己的闩锁。这增加了并行性，因为两个线程可以访问同一页面上的不同插槽。但它增加了访问表的存储和计算开销，因为线程必须为它们访问的每个插槽获取一个闩锁，每个插槽必须为闩锁存储数据。DBMS 可以使用单模式锁存器(即 Spin 锁存器)以一些并行性为代价来减少元数据和计算开销。

也可以直接使用 compare-and-swap (CAS)指令创建一个无锁存器的线性探测哈希表。通过尝试将一个特殊的“null”值与我们希望插入的元组进行比较和交换，可以实现在一个插槽上插入。如果这失败了，我们可以探测下一个槽，一直到它成功为止。

5 B+树锁存

B+树锁存的挑战在于防止以下两个问题:

- 试图同时修改节点内容的线程。
- 一个线程遍历树，而另一个线程拆分/合并节点。

闩锁抓取/耦合是一个协议，允许多个线程同时访问/修改 B+树。其基本思想如下。

1. 给父母留个门闩。
2. 给孩子拿个门闩。
3. 如果孩子被认为是“安全的”，为父母提供释放锁闩。“安全”的节点是指在更新时不会分裂、合并或重新分发的节点。

注意，“安全”的概念取决于操作是插入还是删除。一个完整的节点对于删除来说是“安全的”，因为合并将不需要，但对于插入来说是“安全的”，因为我们可能需要分裂节点。注意，读锁存器不需要担心“安全”的情况。

基本闩锁捕获协议:

- **搜索:**从根开始向下，反复获取 child 上的 latch，然后打开 parent 上的 latch。
- **插入/删除:**从根开始，向下，根据需要获取 X 个锁存器。一旦孩子被锁住，检查它是否安全。如果孩子是安全的，释放锁住它所有的祖先。

从正确性的角度来看，锁存器释放的顺序并不重要。但是，从性能的角度来看，释放树中较高位置的锁存器是更好的，因为它们阻碍了对更大一部分叶子节点的访问。

改进的锁存器捕获协议:基本锁存器捕获算法的问题是，事务**总是**为每个插入/删除操作获取一个根上的独占锁存器。这限制了并行性。相反，人们可以假设必须调整大小(即，分裂/合并节点)是罕见的，因此交易可以获取到叶子节点的共享锁存。每个事务都假定到达目标叶节点的路径是安全的，并使用 READ 锁存器和抓取来到达它并进行验证。如果叶节点不安全，那么我们中止并执行前面的算法，在那里我们获得 WRITE 锁存。

- **搜索:**和之前一样的算法。
- **插入/删除:**将 READ 锁存设置为搜索，转到叶子，并将 WRITE 锁存设置为叶子。如果叶子不安全，释放之前所有的锁存器，并使用之前的插入/删除协议重新启动交易。

叶子节点扫描

这些协议中的线程以“自上而下”的方式获得锁存器。这意味着一个线程只能从低于其当前节点的节点获得锁存器。如果所需的锁存器不可用，则线程必须等待，直到它可用为止。考虑到这一点，就永远不会出现死锁。

然而，叶节点扫描很容易发生死锁，因为现在我们有线程试图同时在两个不同的方向上获取独占锁(例如，线程 1 试图删除，线程 2 进行叶节点扫描)。索引锁存器不支持死锁检测或避免。

因此，程序员处理这个问题的唯一方法就是通过编码规范。叶子节点兄弟闩锁获取协议必须支持“no-wait”模式。也就是说，B+树代码必须应对失败的闩锁捕获。由于闩锁的目的是被(相对地)短暂地持有，如果一个线程试图获得叶节点上的闩锁，但该闩锁不可用，那么它应该迅速中止其操作(释放它持有的任何闩锁)并重新启动操作。