关于并发控制的乐观方法

On Optimistic Methods for Concurrency Control
1981 year

张朝威
2017.04.23

介绍：数据库和事务

- 数据库是什么?

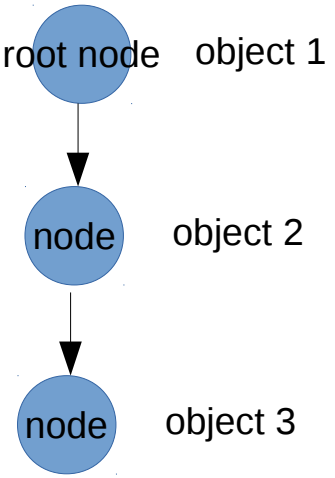
  1. 数据库是对象的集合，对数据库的共享访问就是对这些对象的访问;

- 2. 对象分为两类： roots, distinguished objects; 其他对象，必须先访问 roots ，然后通过指向这些对象的指针来访问;

- 事务是什么?

- 1. 对数据库中的对象的一系列访问的集合;

- 2. 保证访问数据的完整性约束;



介绍：锁并发控制

- 现有加锁方法来控制并发的缺点
- 1. 锁的管理负荷。 只读事务也要加锁； 死锁检测的负载；
- 2. no general-purpose deadlock-free locking protocols 。 B-Tree 索引在其他论文中有 9 种加锁协议；
- 3. 对象加锁后，等待磁盘访问的过程中，降低了访问的并发度；
- 4. 不得已回滚事务时，必须要到事务结束时才能释放所有锁，这也降低了访问的并发度；
- 5. 加锁应该是在最坏的情况下的选择，而不应该是一种常态；

介绍：无锁机制

- 前提条件
- 1. 在给定时刻，数据库中的对象要比运行中的事务所涉及的对象数目要多得多；
- 2. 运行中的事务同时修改数据库中的同一对象的概率要小得多；
- 以上 2 点主要说明事务模型的冲突是不常见的

介绍：无锁机制

- 主要想法
- 1. 读是不受限制的，主要指从一个节点读取其值或指针；但是，一个查询的结果返回认为是一个写操作，需要进行校验；
- 2. 写是严格受限的。事务主要由读阶段、校验阶段、写阶段（可选）这 2/3 个阶段构成。

读阶段：所有的写操作发生在局部的、节点副本上，只在事务内可见；
校验阶段： 用来保证数据完整性
写阶段： 校验成功后才存在，主要是让修改完成的数据全局可见；

如果事务发生失败，需要把事务进行备份，然后再次调用重新执行

读和写阶段

- 底层系统提供对象的管理
- Create 创建一个新对象并返回其名字
- delete(n) 删除对象 n
- read(n,i) 读取对象 n 的项 i 并返回其值
- write(n,i,v) 写入对象 n 的项 i 的值 v
- copy(n) 创建对象 n 的副本并返回其名字
- exchange(n1,n2) 交换两个对象 n1 和 n2 的名字
- 其中 n 是对象的名字， i 是类型管理的一个参数， v 是类型的值（可能是指针，可能是数据）

读和写阶段

```
tcreate = (
   n := create;
   create set := create set ∪ {n};
   return n)

twrite(n, i, v) = (
   if n ∈ create set
        then write(n, i, v)
   else if n ∈ write set
        then write(copies[n], i, v)
   else (
        m := copy(n);
        copies[n] := m;
        write set := write set ∪ {n};
        write(copies[n], i, v)))

tread(n, i) = (
   read set := read set ∪ {n};
   if n ∈ write set
        then return read(copies[n], i)
   else
        return read(n, i))

tdelete(n) = (
   delete set := delete set ∪ {n}).
```

对于每一个事务，它维护着自己需要访问的对象的集合。一开始， tbegin 调用会初始化该集合为空。

用户写的事务执行，就是上面所提及的读阶段；

用户写的 tend 才会触发上面所提及的校验阶段和写阶段；

copies 是对象名字映射的向量。

读阶段并不会发生全局写。对对象的第一次写，会复制一个副本，后面所有的写将会对这个对象副本进行。副本在读阶段过程中对其他事务是不可见的。

另外，我们约定：（1）所有的节点都是通过 root 节点的指针来访问的； 而所有事务都知道 root 节点的全局名字。那么， root 节点的副本是无法访问的，因为它不在全局名字集合之内。（2） root 节点是不会创建和删除的，节点删除后不会留下 dangling pointer ，创建的节点通过新建立的指针变得可访问；

读和写阶段

事务完成之后，就会使用 tend 来调用校验过程。只有校验成功之后，才会进入到写阶段：

$$\textbf{for } n \in \textit{write set } \textbf{do } \textit{exchange}(n, \textit{copies}[n]).$$

写阶段完成之后，事务创建的所有节点将全局可见，事务删除掉的节点将全局不可再见。

事务成功 / 失败完成之后，将会进行一定的清理工作：

$$\begin{aligned}&(\textbf{for } n \in \textit{delete set } \textbf{do } \textit{delete}(n);\\ &\ \textbf{for } n \in \textit{write set } \textbf{do } \textit{delete}(\textit{copies}[n])).\end{aligned}$$

校验阶段

假定 $T_1,T_2,...,T_n$ 是并发执行的。这些事务操作的共享数据结构是 d，而 D 是所有 d 的完整集合，那么事务 T_i 就是这样的一个函数：

$$T_i : D \to D.$$

假定初始化的数据结构是 d_{init}，而最终的数据结构是 d_{final}。那么事务过程就可以看成是将一个并发事务的排列的函数组合应用于数据结构的结果，即：

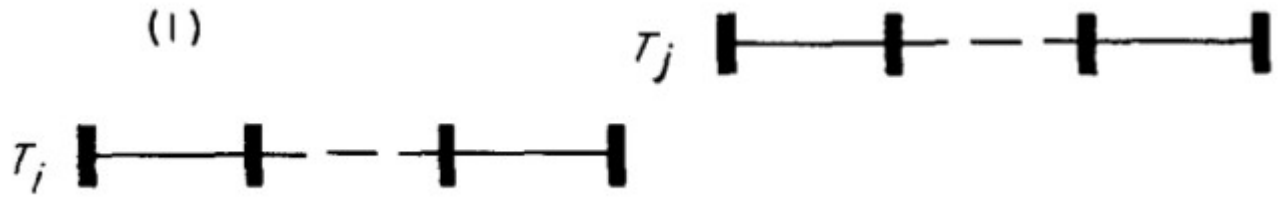
$$d_{\text{final}} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \cdots \circ T_{\pi(2)} \circ T_{\pi(1)}(d_{\text{initial}}),$$

上面这个公式可以使用归纳法证明是成立的，只要每一步满足数据完整性，那么整个操作序列就可以保证数据的完整性。那么，关键的问题就是找到这样的一个事务排列。

这个排列中的两个事务需要满足下面的三个要求之一；其中，当时间 t(i) < t(j) 成立时，事务 T_i 一定比事务 T_j 开始的早。

校验阶段：条件 1

T_i completes its write phase before T_j starts its read phase.

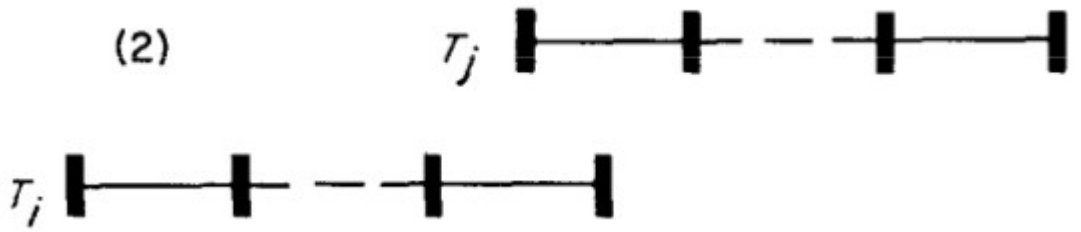


条件 1 说明，事务 T_i 先完成， 然后事务 T_j 才开始；

很明显，如果两个事务的执行时间段是没有交集的，必然可以保证数据的完整性。

校验阶段：条件 2

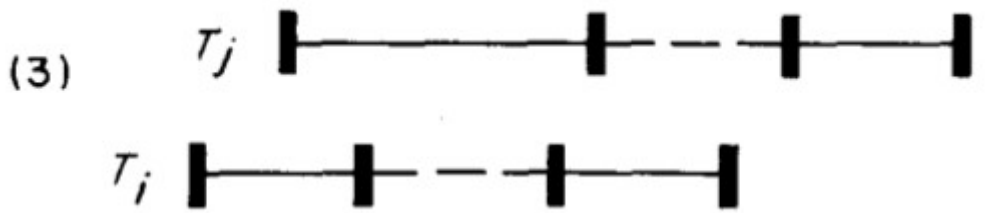
The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.



条件 2 说明，事务 Ti 的写阶段不会影响到事务 Tj 的读阶段，事务 Ti 完成了其写阶段之后事务 Tj 才开始写阶段，所以事务 Ti 不会覆盖写事务 Tj；（同样，事务 Tj 不能够影响 Ti 的写阶段）

校验阶段：条件 3

The write set of T_i does not intersect the read set or the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

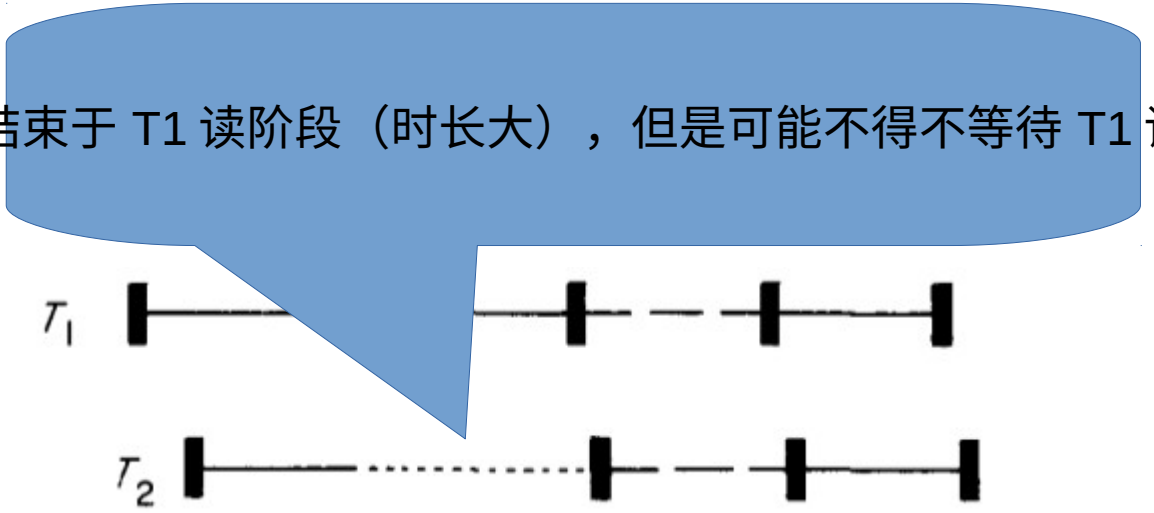


条件 3 跟条件 2 类似，只是简单要求事务 Ti 不会影响事务 Tj 的读阶段或者写阶段；同样，事务 Tj 不能够影响事务 Ti 的读阶段；

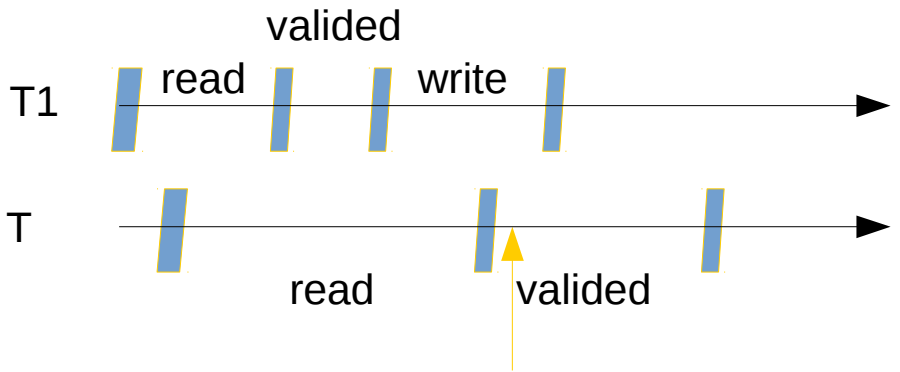
校验阶段：赋于事务号

- 事务号使用连续递增的数字
- 在何时赋于事务号?
- 方案 1： 在读阶段之前赋值（有缺陷）
- 方案 2： 在读阶段结束时赋值

T2 读阶段早结束于 T1 读阶段（时长大），但是可能不得不等待 T1 读阶段的结束



校验阶段：实践问题



当事务 T 进行校验时，必须检查 T1 的写集合。为此，并发控制协议需要维护一个有限的 most recent write sets，其大小要足够大以校验几乎所有的事务。如果写集校验失败的话，那么，事务 T 校验失败，需要对它进行备份。写集 a 比写集 b more recent 是指对应的事务号保持大于关系。

下面讨论校验阶段的优化，主要考虑的是前 2 个条件，这就要求事务的写阶段需要串行化地执行起来。

串行校验

```
tbegin = (
   create set := empty;
   read set := empty;
   write set := empty;
   delete set := empty;
   start tn := tnc)

tend = (
   ⟨finish tn := tnc;
    valid := true;
    for t from start tn + 1 to finish tn do
          if (write set of transaction with transaction number t intersects read set)
            then valid := false;
    if valid
          then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
    if valid
          then (cleanup)
          else (backup)).
```

最简单方案是：事务号赋值、校验阶段和写阶段，全部都放在临界区内。临界区在上图中使用 < > 来表示

串行校验

- 已有优化：只有在校验成功时，才会消耗 1 个事务号；
- 如果涉及到多 CPU 时，就需要将校验阶段的步骤进行并行化，下图中的优化点主要是：
- 1. 读阶段完时，读取 tnc 并将其它赋给 mid tn 。很明显，事务开始时的事务 start tn+1, start tn+2,…,mid tn 的写集合是需要校验的，并且可以在临界区外进行校验；
- 同样，还可以再进行一次优化。就是再读一次 tnc 作为 finish tn ，然后将 mid tn 到 finish tn 之间的写集进行校验。重复这个过程，我们就可以把校验阶段分为多个小步，然后这些小步就可以并行化。

将变化的部分移出临界区之外，增加并行度

串行校验

```
tend := (
    mid tn := tnc;
    valid := true;

    for t from start tn + 1 to mid tn do
    if (write set of transaction with transaction number t intersects read set)
        then valid := false;
   ⟨finish tn := tnc;
    for t from mid tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
          then valid := false;
    if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
    if valid
        then (cleanup)
        else (backup)).
```

并行校验

```
tend = (
  ⟨finish tn := tnc;
   finish active := (make a copy of active);
   active := active ∪ {id of this transaction}⟩;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
         then valid := false;
   for i ∈ finish active do
       if (write set of transaction T_i intersects read set or write set)
         then valid := false;
   if valid
       then (
         (write phase);
         ⟨tnc := tnc + 1;
          tn := tnc;
          active := active—{id of this transaction}⟩;
         (cleanup))
       else (
         ⟨active := active—{id of transaction}⟩;
         (backup))).
```