数据库系统的优化并行控制算法

赵若韵　黄国兴　肖　晶
（华东师范大学计算机科学技术系　上海200062）

On Optimized Methods for Concurrency Control Algorithm of Database Systems

ZHAO Ruo-Yun　HUANG Guo-Xing　XIAO Jing
(Department of Computer Science and Technology, ECNU, Shanghai 200062)

Abstract　The concurrency control problem in database systems has been examined by many people and several concurrency control algorithms have been proposed. The most popular algorithms are two-phase locking and timestamp ordering. Due to its inefficiency in some situations, we propose the optimized method which is combined of timestamp ordering and optimistic method. With this method, the efficiency can be improved in those systems where the conflicts between transactions rarely happen.

Keywords　Transaction, Concurrency controls, Databases, Timestamp ordering, Optimistic method, Optimized method

1. 引言

所谓事务就是可以保持数据库一致性的任何原子操作序列。操作可以是对数据库实体的读或写。为了一组事务可以同时执行，要引入并行控制机制[3]，以保证多个事务对数据库操作后，数据库仍保持一致性。

很多并行控制算法是对数据对象锁定来作为控制机制，两阶段封锁法[1]就是用这一机制对数据库的一致性进行控制。在那些事务之间的冲突基本不会发生的系统中，如查询占主要地位的系统、大型树型结构的索引系统，若用两阶段封锁法，对象的上锁解锁开销会影响整个系统的效率[4]，而且在多个CPU的系统中，不能充分发挥每个CPU的功效，不能很好地利用资源。所以引入优化方法，它是时间戳方法和乐观方法的结合，在事务基本上不会发生冲突的系统中可以明显提高效率。

2. 优化并行控制算法

数据库优化并行控制方法是时间戳方法和乐观方法的结合。两个事务的全局时间戳不同，按照时间戳方法控制；两个事务的全局时间戳相同，用乐观方法根据局部时间戳控制，确保数据库的一致性。

2.1　时间戳方法[1]

在时间戳方法中，给每个事务 T_i 分配一个唯一的时间戳 $TS(T_i)$，并要求所有的事务按照时间戳的大小来运行。每个实体 x 有两个相关记录 TSR(x)、TSW(x)，分别记录对 x 进行读和写的最大时间戳号。事务 T_i 对实体 x 的读操作 $R_i(x)$ 被接收后，系统把 T_i 的时间戳和 TSW(x) 比较，如果小于 TSW(x)，则 $R_i(x)$ 被拒绝，事务 T_i 返回，获得更大的时间戳再次进行尝试；反之则接收。对于 T_i 对 x 的写操作 $W_i(x)$，系统把 T_i 的时间戳和 max[TSR(x), TSW(x)] 比较，若小于 max 值，则拒绝，事务 T_i 返回，获得更大的时间戳再次进行尝试；否则接收。

2.2　乐观方法[2]

(1)乐观方法整体思想

(i) 从一个节点中读取一个值或是一个指针，不会破坏数据完整性，对读操作不加以限制。

(ii) 对于写操作严格限制。把每个事务划分成三阶段：读阶段，验证阶段，写阶段（见图1）。在读阶段，只对本地节点进行写操作，只有验证通过，才在写阶段把本地数据全局化，写到全局数据库中。验证阶段保证事务的操作不会导致数据库的不完整性。

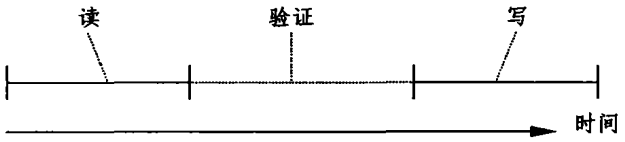

图1　事务的三阶段

乐观方法在验证阶段失败时，会从头开始执行。失败的事务返回，作为一个新的事务重新开始执行。

(2)读写阶段　实际系统允许对不同类型的对象进行操作，假设所有对象有相同的类型。在读写阶段对对象的操作有如下过程函数，其中 n 为对象名，v 是任意一种类型的值（指针、对象的名称或数据等）。

Create	创建一个新的对象，返回它的名字
Delete(n)	删除对象 n
Read(n, i)	读取对象 n 的 i 项，并返回 i 的值
Write(n, i, v)	把 v 值写入 n 的 i 项中去

为支持事务的读写，引入两个过程函数：

copy(n)	创建一个 n 的副本，返回该对象的名字
exchange(n1,n2)	换 n1和 n2的名字

并行控制对用户来说是不可见的，在用户看来，事务的读写都是直接使用以上所定义函数直接操作的，其实事务都是使用下面这些具有相同语法的过程，对于每个事务，并行控制模块都为其维护一组对象用于访问。被访问的对象在 tbegin 中被初始化成空，在乐观方法的读阶段，对这组对象进行写操作（对本地备份对象的写操作），最后调用 tend 进行验证和写操作（本地数据全局化），如果验证成功，把读阶段中写入的本地数据全局化。tbegin 和 tend 在以后的并行算法中给出，其他相关函数描述如下：

```
tcreate=(
  n:=create;
  create set:=create set ∪ {n};
  return n
tread (n,i)=(
  read set:=read set ∪ {n};
```

```
  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}
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)))
```

当一个事务结束后，调用 tend 进行验证，如果验证成功，事务进入写阶段，即局部数据全局化：

for n∈write set do exchange (n, copies[n]).

写操作成功执行后，所有被创建的结点变得可以访问，被删除的结点变得不可访问，要做清理工作：

for n∈delete set do delete (n);

for n∈write set do delete (copies[n]));

如果事务非正常中止，清理工作也是必需的。

(3) 验证阶段　两个事务间的关系有以下三种情况(图2)：

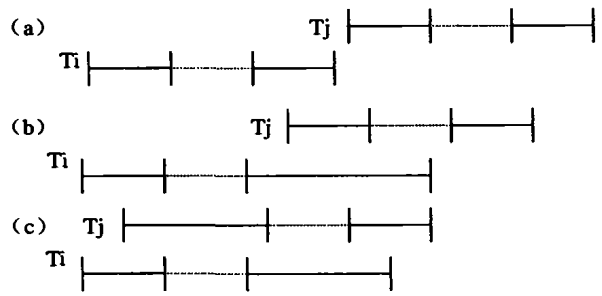

图2　个事务可能的交叉情况

具体描述如下：

(a) 第一种情况，T_i 在 T_j 读阶段开始前结束了写操作，显然满足一致性要求。

(b) 第二种情况，T_i 在 T_j 写阶段开始前结束了写操作，T_i 的写操作和 T_j 的读操作相并行，只要 T_i 的写集交 T_j 的读集为空即可满足一致性要求。

(c) 第三种情况，T_i 的读操作结束在 T_j 的读操作前，T_i 的写集与 T_j 的读写集都并行，T_i 的写集交 T_j 的读写集均为空，才满足一致性要求。

根据实际 CPU 数量和事务读写时间的长短，并行控制算法有所不同。

A. 单 CPU 简单法

只有一个 CPU 的情况，写阶段是依次进行的，不可能是并行的，第三种情形不用考虑，保证(1)(2)满足就可以了，最简单的方法是把整个验证和写阶段放在临界区(算法中〈〉之间的为临界区)，并行算法如下：

```
    tbegin = (
    create set := empty;
    read set := empty;
    write set := empty;
    delete set := empty;
    start tn := tnc )
tend = (
 〈finish tn := tnc; //进入临界区
 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))
          //在写阶段结束后分配事务号
     if valid              //并退出临界区
     then (cleanup)
     else (backup))
```

在每个事务结束的时候为它分配事务号，而不是在开始的时候。在 tbegin 中给一个事务分配事务号 start tn，事务号小于 start tn 的事务肯定都已经完成了写阶段，只要对事务号在 start tn+1和 finish tn 之间的事务进行验证就可以了。

B. 多 CPU 改进法

有多个 CPU 的系统，并且 cpu 大部分时间不是忙于读阶段(即事务的验证阶段和读阶段相比不是非常短)，用下面的并行控制算法提高并行性(tbegin 同前)。

```
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; //进入临界区
  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)) //退出临界
        区
  if valid
      then (cleanup)
      else (backup))
```

在上面的算法中，在读阶段完毕进入验证阶段之前，读当前的 tnc 到 mid tn 中，在临界区外对事务号在 start tn +1到 mid tn 之间的事务进行验证，查看当前事务的读阶段是否和写阶段有冲突。那么一个 CPU 在对一个事务进行验证的同时，其他 CPU 可以对新的事务分配事务号和验证，提高资源利用率。最后对 mid tn +1到 finish tn 的事务进行验证，验证通过，则进入写阶段，并分配事务号。

C. 单 CPU 改进法

单 CPU 简单法，适用于写操作阶段大多在主存中完成的情况，如果大多数写操作不能在主存中完成，并且写操作和读操作相比不是非常短，则写操作可以并行，算法改进如下(tbegin 同上)：

```
tend = (
 〈finish tn := tnc;
 finish active := (make a copy of active ); //active 存放那些在该事
       务写阶段开始前已结束读阶段，但还没完成写阶段的事务
 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 //验证写阶段与本事务写阶段相重合的事务
     if ( write set of transaction Ti 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)))
```

在这个算法中，增加了一个 active 变量，用于保存已经完成读操作且正在进行写操作的事务号，因为写操作可能并行进行，所以要增加验证 finish active 中事务的写集和本事务写集和读集是否相交，写阶段也放在临界区外，可以更好地实施并行。

但是这个算法还有个小小的问题，即便 finish active 中已有的事务是验证失败的，新加入的事务也有可能因为那个验证失败的事务的存在而验证失败。为解决此问题，可以像前面多 cpu 的并行控制算法一样，分阶段进行验证，在把一个事务加到 active 集合之前，尽可能多地验证它和其他事务是否冲突，使得 active 集合中的元素尽可能地少，以减少事务验证失败的可能性，提高效率。

2.3 时间戳方法和乐观方法相结合

分析了时间戳方法和乐观方法之后，我们考虑如何使两种方法结合起来，保证数据库的一致性。总体思想是：如果全局时间戳不同，那么按照时间戳方法进行相应的操作；如果全局时间戳相同，那么就要根据乐观方法来进行并行控制。

(1)事务时间戳　每个事务在开始时都被分配唯一的时间戳 ts(T_i)。ts(T_i)是一个数对(tg(T_i),tl(T_i))。tg(T_i)和 tl(T_i)分别代表 T_i 的全局时间戳和局部时间戳。引入变量 L，称为 strictness level，表示可以拥有相同全局时间戳的正在执行的事务总数。假设 L 的值在执行过程中不会改变。

给每个事务分配时间戳的方法如下。定义四个变量 C1，C2，C3，C4；C1，C2分别用来产生全局时间戳和局部时间戳，C3跟踪目前有相同全局时间戳 V(C1)的事务数，V(C3)不能超过 L。C4跟踪正在执行的所有事务数，V(C4)不能超过 M，M 为系统的最大任务数。若 V(C4)=M，则其他事务都不能启动，直到有一个当前事务退出。算法如下：

(i)tg(T_i)，tl(T_i)赋值如下：

a) 若 V(C4)=M，则 T_i 必须要等到当前在执行的事务中的一个退出，才能继续；否则，V(C4)=V(C4)+1

b) 若 V(C3)<L，则 V(C3)=V(C3)+1，V(C2)=V(C2)+1；否则，V(C3)=V(C2)=1，V(C1)=V(C1)+1

c) tg(T_i)=V(C1)，tl(T_i)=V(C2)

(ii)事务正常终止或异常中止时的操作：

a) 若在中止时 tg(Ti)=V(C1)，则 V(C3)=V(C3)−1

b) V(C4)=V(C4)−1

(2)与实体相关的一些数据

(i)GTSW(x)和 LTSW(x)

GTSW(x)存放对 x 进行过写操作的最大的全局时间戳号，当某个事务的写操作被接收时，该事务的全局时间戳被记录在其中，同时该事务的局部时间戳放在 LTSW(x)列表中，在某个事务正常或非正常退出并且它的全局和局部时间戳都记录在以上两个变量中时，它的局部时间戳会从 LTSW(x)中删除。

(ii)GTSR(x) 和 LTSR(x)

GTSR(x)存放对 x 进行过读操作的最大的全局时间戳号，一个事务的读操作被接收时，该事务的全局时间戳被记录在内，同时该事务的局部时间戳放在 LTSR(x)列表中，在某个事务正常或非正常退出并且它的全局和局部时间戳都记录在以上两个变量中时，它的局部时间戳会从 LTSR(x)中删除。

(3)读和写操作的处理

(i)读操作的处理：当并行控制程序收到一个读操作 R_i(x)时，有以下几种情况：

(a)tg(T_i)<GRSW(x)

这种情况表明 x 被一个有更大全局时间戳的事务写过，则 R_i(x)被拒绝。T_i 返回获得一个新的全局时间戳，重新进行读操作。

(b)tg(T_i) > GTSW(x)

这种情形表明 x 没有被比 T_i 有更大全局时间戳或和 T_i 有相同全局时间戳的事务写过，则 R_i(x)被接收。

(c)tg(T_i) = GTSW(x)

这种情况下，表明 x 被一个或多个与 T_i 有相同全局时间戳的事务写过，则要考虑以下几种情况：

1) LTSW(x)为空

在这种情况下，所有和 T_i 有相同全局时间戳的事务已经完成了对 x 的写操作，那么 R_i(x)被接收。

2) LTSW(x)不为空

这种情况下，和 T_i 有相同全局时间戳的事务仍然在进行对 x 的写操作，那么用乐观方法，在本地完成对对象的写操作，然后验证，判断当前事务是否和之前的事务的写集相交，如果满足一致性的要求，则把当前事务的处理结果全局化。

(ii)写操作的处理：当并行控制程序收到一个写操作 W_i(x)时，会有以下几种情况：

(a)tg(T_i) < Maximum[GTSR(x),GTSW(x)]

这种情况表明 x 被一个有更大全局时间戳的事务读写过，则 R_i(x)被拒绝。T_i 返回获得一个新的全局时间戳，重新进行写操作。

(b)tg(T_i) > Maximum[GTSR(x),GTSW(x)]

这种情形表明 x 没有被比 T_i 有更大全局时间戳或和 T_i 有相同全局时间戳的事务读写过，则 W_i(x)被接收。

(c)tg(T_i) = Maximum[GTSR(x),GTSW(x)]

这种情况下，表明 x 被一个或多个与 T_i 有相同全局时间戳的事务读写过，考虑以下几种情况：

1) GTSW(x) > GTSR(x)

在这种情况下，x 目前不被所有和 T_i 有相同全局时间戳的事务进行读操作，那么检查 LTSW(x)，如果这个集合为空，那么其他和 T_i 有相同全局时间戳的事务已经完成了写操作，那么 W_i(x)被接收；否则用乐观方法进行验证，根据实际情况进行相应的处理。

2) GTSW(x) < GTSR(x)

在这种情况下，x 目前不被所有和 T_i 有相同全局时间戳的事务在进行写操作，那么检查 LTSR(x)，如果这个集合为空或者只包含和 T_i 有相同局部时间戳的事务，那么其他对 x 进行读操作的事务已经结束，W_i(x)被接收；否则用乐观方法对 LTSR(x)中的元素进行验证，根据实际情况进行相应的处理。

3) GTSW(x) = GTSR(x)

在这种情况下，检查 LTSW(x)和 LTSR(x)，如果 LTSW(x)为空并且 LTSR(x)为空或者只包含和 T_i 有相同局部时间戳的事务，那么 W_i(x)被接收；否则用乐观方法验证当前事务和 LTSW(x)和 LTSR(x)之中的事务冲突与否，根据实际情况进行相应的处理。

小结　关于数据库系统并行控制的研究，很多方法都是基于锁机制，但其实有两种机制：锁机制和备份机制，优化方法便是基于备份机制的。优化方法在那些事务之间的冲突很少发生的系统中，明显优于锁机制方法。但是优化方法可能导致饥饿，就像锁机制可能导致死锁一样，当然这可以通过引入锁机制来解决饥饿问题。

参考文献

1 Farrag A A, Ozsu M. Towards a General Concurrency Control Algorithm for Database Systems. IEEE Transactions on Software Engineering, 1987, SE-13(10)

2 Kung H T, Robinson J T. On Optimistic Methods for Concurrency Control. Readings in database systems (3rd ed.), July 1998

3 Bhargava B. Concurrency Control in Database Systems. IEEE Transactions on Knowledge and Data Engineering, 1999, 11(1)

4 Thomasian A. Concurrency control : methods, performance, and analysis. ACM Computing Surveys (CSUR), 1998, 30(1)