第 6 卷第 10 期
1995 年 10 月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 6, No. 10
Oct. 1995

一个基于 SQL 的并发控制机制*

曲云尧　　施伯乐

(复旦大学计算机科学系,上海 200433)

摘要　传统的读写事务模型(由 read(x)和 write(x)序列组成)不能使调度机制充分利用应用程序的语义信息对事务进行灵活调度,从而不能有效提高系统的并发度.本文根据 SQL 语言的操作语义,给出了基于 SQL 的事务模型.利用这种事务模型并结合 2PL 方法,设计了并发控制机制:Condition－locking.这个机制可以:(1)避免数据库中的幽灵(phantom)问题,(2)利用应用程序的语义信息和完整性约束提高系统的并发度,(3)减少发生死锁的机会.因此,这是一个实用的并发控制机制.

关 键 词　数据库,并发控制,幽灵问题,SQL 语言,两段锁.

1　关系数据库中传统封锁机制的问题

在多用户或分布式数据库系统中,DBMS 并发控制机制的效率对整个系统的吞吐率起着关键性的作用.特别是在事务处理速度要求较高的环境,如证券业、银行业等,就显得更为重要.因此,如何提高数据库系统并发控制机制的性能,有着很现实的意义.

并发控制方法一直是数据库研究领域的主要课题[1,2].自从 Eswaran 等发表了关于并发控制的 2PL 锁方法以来[3],并发控制技术得到了很大发展.人们在不断地试图发现更有效的方法和技术,以适应不同的应用环境.其中,主要可把它们归纳为以下几类:(1)锁方法,(2)时间戳方法,(3)多版本方法,(4)乐观方法等.但是,无论是在哪种原型中[4]或是在成熟的商品中,如 Oracle、SYBASE、INFORMIX 等,锁方法占有绝对的主导地位,其原因不仅是它简单易实现,而且在许多环境中,相对于其它方法来说,其效率是最高的[5].但锁方法也带来了一些问题.例如,如何有效确定封锁单位,如何避免幽灵进出数据库,如何减少发生死锁机会等,都是需要进一步研究的.

在研究新方法的同时,提高和改进原有并发控制机制的效率,即并发度,一直是人们极为感兴趣的问题[6,7].其主要方法是利用事务(或应用程序)的语义信息,去掉事务之间不必要的等待.即人们企图从事务的更高层次上得到更多的关于此事务的语义信息,并利用这些信息对事务进行高效调度.

---

* 本文 1994－04－23 收到,1994－11－22 定稿

本项目受到国家 863 高技术项目资助.作者**曲云尧**,1961 年生,副教授,主要研究领域为面向对象数据库,知识库的体系结构,事务管理技术.**施伯乐**,1935 年生,教授,博士导师,主要研究领域为数据库理论与实现技术,知识库,软件工程等.

本文通讯联系人:曲云尧,上海 200433,复旦大学计算机系

在关系数据库中,2PL 锁机制归结起来有下列主要问题:

(1)封锁单位问题. 关系数据库中,逻辑级上的封锁单位一般分为关系和元组. 物理级上的封锁单位一般分为页面. 用户程序对关系或元组的封锁,DBMS 最终要转换成物理级上对页面的封锁. 例如,在逻辑级上要封锁某几个元组,如这几个元组在页面 P1 中,系统则要锁住这个页面 P1. 因此,一般来讲,系统实际的封锁单位要比逻辑上要求的封锁单位大一些.

例 1:有两个事务

```
T1:  SELECT  A          T2:  UPDATE  R
     FROM    R               SET     (A=a,B=b)
     WHERE   B>3             WHERE   B<=3
```

设关系 R 中某些 B 属性值大于 3 和小于 3 的元组存放在同一页面中. 如果系统在物理级上对页面进行封锁,则这两个事务就发生冲突. 但从逻辑上我们知道 T1 封锁的 R 中 B 属性值大于 3 的元组,而 T2 则相反,所以这两个事务实际上无冲突,可以并发执行. 因此,如果使用大的封锁单位(页面),则要降低并发度. 但是,如果我们只封锁页面中(要使用)的元组,则会出现什么问题呢?

(2)幽灵问题. 幽灵问题[1,3]可以说是封锁方法中的一个较为棘手的问题. 应该说,现在许多商品化 DBMS 都没有很好地解决它. 例 2 说明了一个幽灵问题.

例 2:有一个雇员关系 R 如表 1.

表 1　雇员关系 R

EMPNAME	AGE	SALARY	DEPT
John	30	2000	SAL
Mary	40	3000	TOY
Francis	25	2500	SAL
Susan	27	2800	SAL

有两个事务 T1 和 T2,T1:统计 SAL 部门雇员的平均工资. T2:从 SAL 部门删除 John,插入一元组(Mark,25,2500,SAL)到 R 中. 一个可能的 2PL 调度如下:

(1) T2 从关系 R 中锁住并删除关于 John 的元组(John,30,2000,SAL)

(2) T1 锁住(Francis,25,2500,SAL)和(Susan,27,2800,SAL)两个元组,并统计平均值((2500+2800)/2=2650)

(3) T1 提交(COMMIT)

(4) T2 插入元组(Mark,25,2500,SAL)到 R 中,并锁住

(5) T2 提交

如果让 T1 和 T2 串行执行,则如按调度 T1 T2 执行,T1 的结果为(2000+2500+2800)/3=2433.3;如按 T2 T1 执行,T1 的结果为(2500+2800+2500)/3=2600. 因此,上面的调度是不可串行化的(尽管它们都遵守 2PL 协议). 我们称(John,30,2000,SAL)和(Mark,25,2500,SAL)为幽灵元组. 因为它们出入数据库就象幽灵一样,其它事务不能发现它们. 当然,如果加大封锁单位,例如,锁住整个关系,则可避免幽灵问题,但这对系统的效率影响极大. 可以想象关系 R 中存放了大量关于其它部门职员的有关元组.

(3)事务模型研究和实际应用程序之间脱离.通常在数据库中,我们把事务的操作类型分为:读操作(Read)和写操作(Write).可以说,这两种基本操作不能充分体现事务的操作语义,从而不能使系统利用事务的语义信息来提高并发度.在研究并发控制机制时,传统的研究方法并不考虑事务模型和应用程序之间的语义联系,因此,事务模型语义单调(只由Read(X)和Write(X)序列组成),很难进一步改善并发控制机制的性能.例如,在例1中,实际上根据T1和T2两个事务的操作条件(分别为B>3和B<=3)知它们是不冲突的,可以并发执行.确切地说,这种调度是基于应用程序语义的调度.上面这种调度实际使用了谓词锁[1,3]的概念.谓词封锁单位,或者说,基于操作条件的封锁单位是理想的,它不但可避免幽灵问题,而且这种封锁单位是准确的.但文献[1]认为,当谓词条件复杂时,实现起来较为复杂.

本文结合关系数据库标准于语言SQL和2PL锁方法,给出了基于SQL的事务模型,设计了基于这种模型的并发控制机制——Condition-Locking.这个机制可以避免数据库中的幽灵问题,利用更多的语义信息提高并发度,并在一定程度上减少了发生死锁的机会.

2 基于SQL的事务模型

在传统数据库并发控制方法的研究中,通常把事务抽象成仅由读操作(read(x))和写操作(write(x))组成的序列.在本文,我们把事务表示成由查询操作:Q(R,C)和更新操作:U(R,C),D(R,C),I(R,C)组成的序列.其中Q,U,D和I分别表示SQL程序中的选择(SELECT),修改(UPDATE),删除(DELETE),和插入(INSERT);R表示关系,C表示操作条件.Q(R,C)(U(R,C),D(R,C))表示对关系R中满足条件C的元组查询(修改,删除)I(R,C)表示插入R中的元组满足条件C.

定义1. 设事务Ti的操作Oi要对数据库中满足条件Ci的元组进行操作,则称Ci为Oi的操作条件.满足条件Ci的元组集合记为SAT(Ci).

根据SQL语法,把SQL程序转换成事务操作序列的方法可描述如下.

(1)选择语句的转换

· 在单个关系中选择

```
SELECT  A
FROM    R        转换成 Q(R,C)
WHERE   C
```

· 在多个关系中选择(连接)

```
SELECT  A
FROM    R1,R2,…,Rn        转换成 Q(R1,C1)Q(R2,C2)…Q(Rn,Cn)
WHERE   C1∧C2∧…∧Cn∧φ
```

其中Ci(i∈{1,…,n})是在Ri中做选择运算时的选择条件,φ表示n个关系之间的连接条件.

· 嵌套查询

```
SELECT  A
```

```
FROM    R1
WHERE   C1 AND α  θ  SELECT  B      转换成 Q(R1,C1)Q(R2,C2)
                     FROM    R2
                     WHERE   C2
```

当出现多层嵌套时,处理方法类似.

(2)修改语句的转换

```
UPDATE  R
SET     (A1=W1,A2=W2,…)      转换成 U(R,C)
WHERE   C
```

(3)插入语句的转换

```
INSERT
INTO    R(A1,A2,…,An)      转换成 I(R,C)
VALUE   (a1,a2,…,an)
```

C 为 A1=a AND A2=a2 AND…AND An=an

(4)删除语句的转换

```
DELETE
FROM    R       转换成 D(R,C)
WHERE   C
```

经过上述处理后,我们可把事务 Ti 写成 $Ti=(Oj(Rj,Cj))_{j=1}^{n}$

其中 $Oj\in\{Qi,Ui,Di,Ii\}$,Ti 按下列操作顺序执行 O1(R1,C1) O2(R2,C2) … On(Rn,Cn)

在执行完 On(Cn,Rn)后,再执行 COMMIT 操作表示提交,只要其中有一步无法执行,则整个事务无法执行.

例 3:设有一 SQL 程序如下

```
SELECT *
FROM R,S
WHERE R.A>1 AND R.B=S.B AND S.C >2
UPDATE R
SET A=2,B=3
WHERE B=2
```

根据前面的转换过程,其相应事务 T 的操作序列为:T: Q(R,A>1) Q(S,C>2) U(R,B=2),即事务 T 先对 R 和 S 做查询,其查询条件分别为 A>1 和 C>2,然后对 R 进行修改操作.其条件为 B=2.

3　冲突类型与相关判定算法

根据上节提出的事务模型,现给出事务之间冲突的定义.

定义 2. 设 Oi(Ri,Ci)和 Oj(Rj,Cj)分别表示 Ti 和 Tj 的操作($i\neq j$),则 Ci 和 Cj 相关当且仅当 Ri 和 Rj 为同一关系且存在一元组 $t\in DOM(Ri)$,使 $t\in SAT(Ci)$且 $t\in SAT(Cj)$.

定义 3. 操作 Oi(Ri,Ci)和 Oj(Rj,Cj)是不冲突的,当且仅当 Ci 和 Cj 不相关,或者 Oi 和

Oj 同时为查询操作,或者同时为删除操作,或者同时为插入操作.

下面介绍条件锁的概念.条件锁 Oi—LOCK(R,C)表示事务 Ti 对 R 中满足条件 C 的元组加 Oi 锁.

定义 4. 条件锁 Oi—LOCK(Ri,Ci)和 Oj—LOCK(Rj,Cj)(i≠j)相容当且仅当 Oi(Ri,Ci)和 Oj(Rj,Cj)不冲突.

如果事务 Ti 要执行操作 Oi(Ri,Ci),则需先申请条件锁 Oi—LOCK(Ri,Ci),只有条件锁 Oi—LOCK(Ri,Ci)得到允许后,才能执行 Oi(Ri,Ci).

根据条件锁的相容性定义,可得到相容性矩阵如表 2.空白表示不相容,Y 表示相容.

表 2 相容性矩阵

	Q—LOCK(Rj,Cj)		U—LOCK(Rj,Cj)		I—LOCK(Rj,Cj)		D—LOCK(Rj,Cj)	
	Ci 和 Cj		Ci 和 Cj		Ci 和 Cj		Ci 和 Cj	
	相关	不相关	相关	不相关	相关	不相关	相关	不相关
Q—LOCK(Ri,Ci)	Y	Y		Y		Y		Y
U—LOCK(Ri,Ci)		Y		Y		Y		Y
I—LOCK(Ri,Ci)		Y		Y	Y	Y		Y
D—LOCK(Ri,Ci)		Y		Y		Y	Y	Y

从表 2 可看出,尽管更新操作 I(Ri,Ci)和 I(Rj,Cj)的操作条件是相关的,但 I—LOCK(Ri,Ci)和 I—LOCK(Rj,Cj)是相容的,因此它们可并发执行. 同样 D(Ri,Ci)和 D(Rj,Cj)也可并发执行.

为充分利用数据库中的语义信息进行调度,在相关算法中,我们利用属性值之间的完整性约束条件改善系统的性能.

定义 5. 如果属性 B 的取值受限于属性 A 的取值,即有完整性断言:$A\theta_1$ a⇒$B\theta_2$ b(如 $A\theta_1$ a 为真,则 $B\theta_2$ b 也为真),则称 B 受限于 A,记为 dependent(B,A).

显然,如果数据库中有完整性断言:$A\theta_1$ a⇒$B\theta_2$ b,且 $A\theta_1$ a 和~($B\theta_2$ b)分别为两个操作条件,则 $A\theta_1$ a 和~($B\theta_2$ b)是不相关的.因为,对任一元组 t∈SAT($A\theta_1$ a)由完整性断言:$A\theta_1$ a⇒$B\theta_2$ b 知 t∈SAT($B\theta_2$ b),所以 t 不属于 SAT(~($B\theta_2$ b)).

例 4:设 A,B 分别为一个关系的两个属性,在数据库中有完整性断言:A>3⇒B>4,两个操作条件 P1 和 P2 分别为 P1:A>5,P2:B≤1.对任一元组 t∈SAT(A>5),知 t∈SAT(A>3),由完整性断言 A>3⇒B>4,知 t∈SAT(B>4),所以 t 不属于 SAT(B≤1),即 P1 和 P2 是不相关的.

根据 SQL 条件表达式的构成规则,我们把操作条件定义成下列形式*:

P1 AND P2 AND … AND Pn

并设 Pi(i∈{1,…,n})的形式如下:

Pi: A θ_i a

A 为属性名,a∈DOM(A),θ_i∈{=,≠,>,≥,<,≤}.

设 Pi 和 Pj 为两个操作条件,Pi:A θ_i a,Pj:B θ_j b,我们给出判断 Pi 和 Pj 是否相关的

* 当操作条件为更复杂的条件表达式时,可以把它化成析取范式,从而对每一个析取式分别考虑,见文献[3].

算法.

算法:判断 Pi 和 Pj 是否相关

输入:Pi 和 Pj

输出:Pi 和 Pj 是否相关的判定

方法:IF A 和 B 为同一属性

```
THEN  DO CASE
      CASE a=b
      IF (θi ∈{=}∧θj∈{≠,>,<})
            ∨(θi∈{≠}∧θj∈{=})
            ∨(θi∈{<}∧ θj∈{≥})
            ∨(θi∈{≤}∧θj∈{>})
            ∨(θi∈{>}∧θj∈{=,≤})
            ∨(θi∈{≥}∧θj∈{<})
         THEN 返回“不相关”
         ENDIF
      CASE a<b
         IF (θj∈{=,≥,>}∧θi∈{=, ≤,<})
         THEN 返回“不相关”
         ENDIF
      CASE a>b
         IF (θi∈{=,≥,>}∧θj∈{=, ≤,<})
         THEN 返回“不相关”
         ENDIF
      ENDCASE
ENDIF
IF A 和 B 为不同的属性名
THEN IF dependent(B,A) /* 有完整性断言:Aθ1 a1 ⇒ B θ2 b1 */
     THEN IF (P1 ⇒ Aθ1 a1 AND P2⇒ ~(Bθ2 b1))
     /* P1隐含 Aθ1 a1,P2隐含~(Bθ2 b1) */
        THEN 返回“不相关”
        ENDIF
     ENDIF
ENDIF
返回“相关”
```

设两个操作条件 Ci 和 Cj 如下:

Ci: Pi1 AND Pi2 AND … AND Pim　　Cj: Pj1 AND pj2 AND … AND Pjn

我们给出判定 Cj 和 Ci 是否相关的算法.

算法:判定操作条件 Ci 和 Cj 是否相关

输入:Ci 和 Cj

输出:Ci 和 Cj 是否相关的判定

方法:

```
FOR K=1 TO m
    FOR L=1 TO n
        IF Pik 和 Pjl 不相关
        THEN 返回“不相关”
        ENDIF
    NEXT L
    NEXT K
    返回“相关”
```

例5:设有两个 SQL 程序如下

程序1:SELECT *　　程序2:SELECT *

```
FROM R                          FROM R
WHERE A>2 AND A<5               WHERE A>3
                                DELETE
                                FROM R
                                WHERE A>=5 AND A<=8
```

其相应的SQL事务模型为

T1:Q1(R,A>2 AND A<5)

T2:Q2(R,A>3) D2(R,A>=5 AND A<=8)

根据算法知Q1和Q2的操作条件是相关的,Q1的操作条件和D2的操作条件是不相关的.由定义知T1和T2的操作之间是不冲突的.

4 并发控制机制Condition-Locking

在本节中,根据我们提出的SQL事务模型和相关算法,结合2PL并发控制方法,设计并发控制机制Condition-Locking.

对每个关系R,定义一个条件锁集合:

LOCK-SET(R)={Oi-LOCK(R,Ci)|操作Oi(R,Ci)被允许对关系R操作}

LOCK-SET(R)记录了所有事务对关系R加的条件锁.

对每个事务T,定义一个关系集合.

R-USED(T)={R|T对关系R加了条件锁}

R-USED(T)记录了T使用的所有关系.

Condition-locking方法和2PL方法是类似的,在第一阶段事务只得锁,在第二阶段事务只释放锁.

算法:Condition-Locking

输入:Oi(R,Ci)

方法:

```
DO CASE
    CASE Oi≠Cmommit /* 上升段 */
        IF 存在Oj-LOCK(R,Cj)∈LOCK-SET(R)
          且Oj-LOCK(R,Cj)和Oi-LOCK(R,Ci)不相容
        THEN 挂起Ti,直到LOCK-SET(R)中不存和Oi-LOCK(R,Ci)不相容的锁
        ENDIF
        LOCK-SET(R)=LOCK-SET(R)∪Oi-LOCK(R,Ci)
        返回并执行
      CASE Oi=Commit /* 下降段 */
        对每一个Rm∈R-USED(Ti) DO
        LOCK-SET(Rm)=LOCK-SET(Rm)-Oi-LOCK(Rm,Ci) /* 释放锁 */
      ENDCASE
```

例6:在例1中,T1和T2可由SQL描述如下:

```
T1  SELECT AVG(SALARY)
    FROM EMP
    WHERE DEPT="SAL"
T2  DELETE
```

FROM EMP
WHERE EMPNAME="John" AND DEPT="SAL"
INSERT
INTO EMP(EMPNAME,AGE,SALARY,DEPT)
VALUE (Mark,25,2500,SAL)

令 P1为 DEPT="SAL"

令 P2为 EMPNAME="John"

令 P3为 EMPNAME="Mark" AND AGE=25 AND SALARY=2500

用 SQL 事务模型表示 T1和 T2:

T1: Q1(EMP,P1)

T2: D2(EMP,P1 AND P2) I2(EMP,P1 AND P3)

表3是使用 Condition—Locking 对 T1和 T2进行调度的过程,此调度是可串行化的.因此,执行结果是正确的.

表3　T1和 T2的执行过程

步骤	操作	LOCK —SET(EMP)	R—USED(T1)	R—USED(T2)
1	D2(EMP,P1 AND P2)	{D2—LOCK(EMP,P1 AND P2)}	{}	{EMP}
2	Q1(EMP,P1)	由于 D2—LOCK(EMP,P1 AND P2)和 Q1—LOCK(EMP,P1)不相容,挂起 T1	{}	{EMP}
3	I2 (EMP,P1 AND P3)	{D2—LOCK(EMP,P1 AND P2), I2—LOCK(EMP,P1 AND P3)}	{}	{EMP}
4	T2 COMMIT	{}	{}	{}
5	Q1(EMP,P1)	{Q1—LCK(EMP,P1)} {EMP}	{EMP}	
6	T1 COMMIT	{}	{}	

5　Condition —locking 分析与正确性证明

我们分析和讨论 Condition—locking 的一些特点,并给出一些结论的正确性证明.

5.1　Condition—locking 分析

(1)关于封锁单位

Condition—locking 使用 SQL 程序的操作条件作为封锁单位,理论上是最准确的.因为这正是事务要操作的元组(包括当前在数据库中的或将来要进入数据库的).但是当操作条件不是简单条件表达式:P1 AND P2 AND…AND Pn 时,其相关算法还需改进.在许多应用环境中,如证券业、银行、定票系统等,其应用程序的操作条件往往都具有上述这种简洁形式.因此,Condition—locking 在这样的环境中可充分发挥其特点提高系统的事务处理效率.另一方面,Condition—locking 对每个操作只加锁一次,这和封锁元组的方式相比,明显减少了加锁次数.

(2)关于幽灵问题

Condition—locking 彻底解决了幽灵问题,并且和使用传统读写事务模型相比不降低系统的并发度(见证明).这是传统方法所不能达到的.

在有些情况下,该机制和传统方法相比,似乎"降低"了并发度.例如,有一关系 R 如表4,两个事务 T1、T2,如下:

表4 关系R

	X	Y
T1:Q1(R,X<=2)	1	2
T2:U2(R,Y>4)	2	3
	4	5
	6	7

设X和Y之间无值之间的约束联系,即X和Y互不受限.如果使用封锁元组的方式,则T1封锁(1,2)和(2,3)两个元组,T2封锁(4,5)和(6,7)两个元组.因此它们之间不发生冲突,从而可以并发执行.而Condition—Locking判定Q1(R,X<=2,R)和U2(R,Y>4)之间冲突,所以不能并行执行.但我们需记住的是Condition—Locking避免了由幽灵引起的潜在的非串行化调度.

(3)关于死锁问题

Condition—Locking在一定程度上减少了发生死锁的机会(见证明).

例7:有两个事务对表4的关系R操作

T1: U1(R,X<=2)

T2: U2(R,X<=2)

T1和T2要封锁的元组有(1,2)和(2,3),下面的调度过程是使用传统方法对元组进行封锁.

(1) T1锁住元组(1,2).

(2) T2锁住元组(2,3).

(3) T1申请对元组(2,3)加锁,等待T2.

(4) T2申请对元组(1,2)加锁,等待T1.

从而T1和T2相互等待,发生了死锁.而Condition—Locking将判定U1(R,X<=2)和U2(R,X<=2)是冲突的,所以T1和T2不能同时对R中的元组持有锁,即不会发生死锁.

(4)关于冲突类型

Condition—Locking使用了SQL的操作类型.O(R,C)可表示选择、更新、删除、插入中的任何一个操作.这和传统的Read(x)和Write(x)相比,能给DBMS以更灵活的方式进行调度.

5.2 正确性证明

现给出Condition—Locking的几个性质和正确性证明.

一个数据库应用程序可用多种事务模型描述.传统的读写事务模型和本文给出的SQL事务模型中的操作之间的对应关系如表5.

表5 传统事务模型和SQL事务模型中操作之间的对应关系

传统读写事务模型	基于SQL的事务模型
READ(x)	Q(x)
WRITE(x)	U(x),D(x),I(x)

即在SQL事务模型中,传统读写事务模型中的READ(x)被表示成Q(x),而WRITE(x)将根据其在应用程序的类型可分别表示成U(x),D(x)或I(x).

我们知道在传统读写事务模型中,两个操作是冲突的当且仅当它们对同一个数据项操作且至少有一个为写操作.从表2和表5中知,如在传统读写事务模型中两个操作是冲突的,但它们相应的操作在SQL事务模型中并不一定冲突.如两个删除操作便是这种情况.

定理1. 对任何一个应用程序所产生的事务 T,如果使用传统事务模型对 T 调度是无冲突的(T 可执行),则使用 SQL 事务模型对 T 进行调度也是无冲突的(T 也可执行),反之不然.

证明:在传统读写事务模型中,如果两个操作 O1和 O2是不冲突的,则有两种情况:

(1) O1和 O2对不同的数据项操作.

(2) O1和 O2都为读操作.

在 SQL 事务模型中(1)(2)两种情况都是不冲突的.

设在 SQL 事务模型中,有两个删除操作 D1(x)和 D2(x),由表2知,D1(x)和 D2(x)是不冲突,可并发执行.而在传统读写事务模型中,D1(x)和 D2(x)将被分别表示成 WRITE1(x)和 WRITE2(x),因此,WRITE1(x)和 WRITE2(x)是冲突的.□

定理2. 对任一组事务 TA={T1,T2,T3,…,Tn},如果使用 SQL 事务模型对 TA 调度发生了死锁,则使用传统读写模型对 TA 调度也发生死锁.反之不然.

证明:如系统用 SQL 事务模型对 TA 调度发生了死锁,则 TA 中某些事务的操作之间发生冲突,从而使这些事务相互等待.由定理1知,如果使用传统读写事务模型进行调度,则这些操作之间也是冲突的,从而也互相等待,即也发生死锁.

从例7知,使用传统事务模型对 T1,T2调度发生了死锁,而使用 SQL 事务模型时,不发生死锁.□

定理3. 设在数据库中有完整性断言:P1⇒P2.C1和 C2为两个操作条件.如果 C1⇒P1,C2⇒～P2(C1隐含 P1,C2隐含非 P2),则 C1和 C2是不相关的.

证明:设 t∈SAT(C1),因 C1⇒P1,则 t∈SAT(P1),由 P1⇒P2,知 t∈SAT(P2).由 C2⇒～P2,知 P2⇒～C2.再由 t∈SAT(P2)和 P2⇒～C2,知 t∈SAT(～C2),所以 t 不属于 SAT(C2).即 C1和 C2不相关.□

定理4. Condition－Locking 所产生的调度是可串行化的.

证明:因为2PL 调度是可串行化的,所以只证明 Condition－Locking 为二段锁调度方式即可.证明分两步:

(1) 当事务要执行 O(x)操作时,先对 x 加适当的锁.由 Condition－Locking 算法知,当系统收到一个事务 T 对数据库的操作 O(x)时,系统将根据其操作类型,对封锁单位加适当的锁,只有加锁过程完毕后,才能执行 O(x),否则 T 等待.

(2) 事务一旦放锁,不再申请锁.由 Condition－Locking 知,系统只有在接到事务 T 的 commit 操作后,才把 T 所加的锁放掉,因此,T 一旦放锁,不会再申请锁.□

6　总结与展望

在本文中,我们给出了基于 SQL 的事务模型,该模型和传统的读写事务模型相比有更丰富的语义,从而使 DBMS 更灵活地对事务进行调度.

Condition－Locking 可以避免幽灵问题,利用语义信息提高并发度,并减少发生死锁的机会,因此我们相信该方法可以提高 RDBMS 的效率.

Condition－Locking 的速度主要取决于相关算法.如果操作条件很复杂,如何提高相关算法的速度是主要问题.

本文仅仅是根据SQL的操作语义进行调度，实际上在具体的应用环境中，大量具体的语义信息还是可以用来提高系统的并发度.所以，作者认为未来的并发控制机制应该允许用户方便的输入各种语义信息.这一点在面向对象的数据库或知识库中显得更有意义.

参考文献

1 Bernstein P, Hadilacos V, Goodman N. Concurrency control and recovery in database system. Addison Wesley, Reading MA, 1987.

2 Barghouti N S, Kaiser G E. Concurrency control in advanced database application. ACM Computing Surveys, Sept., 1991, 23(3).

3 Eswaran K, Gray J, Lorie R *et al*. The notions of consistency and predicate locks in a database system. Communication of the ACM 19, 11, NOV. 1976. 624—633.

4 Won Kim, Frederick H Lochovsky. Object—oriented concepts, databases and applications. Addison Wesley Publishing Company, 1989.

5 Wang Y, Rowe L A. Cache consistency and concurrency control in a client/server DBMS architecture. ACM SIGMOD, Denver, Colorado, May 1991.

6 Garcia—Molina H. Using semantic knowledge for transaction processing in a distributed database. ACM TODS, June 1983. 186—213.

7 Badrinath B, Ramamritham K. Semantic—based concurrency control: beyond commutativity. ACM TODS, Mar. 1992.

A CONCURRENCY CONTROL MECHANISM BASED ON SQL

Qu Yunyao Shi Baile

(*Department of Computer Science, Fudan University, Shanghai* 200433)

Abstract By the standard transaction model (composed of read and write operations), the scheduler can not exploit the semantics of application programs, thus not enhancing concurrency effectively. In this paper, a new transaction model based on SQL language are presented, which can exploit the semantics of high—level operations. Depending on the the transaction model and Two—Phase Locking method, the authors design a concurrency control mechanism—— Condition — Locking, which can, (1) prevent phantoms from database, (2) exploit semantic information of application programs and the knowledge of the integrity constraints of the database system to enhance concurrency, (3) reduce the chances of producing deadlock. Hence, it is a practical concurrency control mechanism.

Key words Database, concurrency control, phantom, SQL language, two—phase locking.