基于混合粒度冲突检测的事务工作流调度算法*

丁 柯+, 魏 峻, 冯玉琳

(中国科学院 软件研究所 计算机科学重点实验室,北京 100080)
(中国科学院 软件研究所 软件工程技术中心,北京 100080)

A Scheduling Protocol for Transactional Workflows Based on Mix-Grained Conflict Detection

DING Ke+, WEI Jun, FENG Yu-Lin

(Key Laboratory for Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)
(Software Engineering Technology Center, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)
+ Corresponding author: Phn: 86-10-62630989 ext 203, E-mail: dingke@otcaix.iscas.ac.cn; dingke@cellix.com.cn
http://www.iscas.ac.cn
Received 2002-01-15; Accepted 2002-05-13

Ding K, Wei J, Feng YL. A scheduling protocol for transactional workflows based on mix-grained conflict detection. *Journal of Software*, 2003,14(3):369~375.

Abstract: A transactional workflow is composed of traditional flat transactions, and its execution has relaxed transactional atomicity. Due to different termination characteristics of transactions, only one workflow is allowed to execute non-compensatable transactions with current scheduling protocols. In this paper, two granularities of conflict based on transaction classes and transaction instances are defined, and a scheduling protocol by using both granularities of conflict detection is proposed. Besides generating serializable and recoverable schedules, this method provides a higher degree of concurrency in following two ways. On the one hand, the fine-grained locking mechanism based on transaction instances is used to reduce conflict possibility among concurrent workflows. On the other hand, the coarse-grained conflict mechanism based on transaction classes is used to predict future conflict among workflows, multiple workflows are therefore allowed to execute non-compensatable transactions if they will not conflict in predicated future execution.
Key words: transactional workflow; concurrency control; recovery; scheduling protocol; locking granularity

摘 要: 事务工作流由若干个平面事务组成,其执行满足松弛原子性.由于组成事务工作流的平面事务具有不同的完成特性,为了防止不可串行化的执行,现有的调度算法通常只允许一个活动工作流执行不可补偿事务,这大大限制了并发度.定义了基于事务类型和事务实例两种粒度的冲突关系,并提出了一种基于这两种粒度冲突检测的调度算法,保证了并发事务工作流的可串行化和可恢复执行.该算法从两个方面提高了并发度:一方面通

* Supported by the National Natural Science Foundation of China under Grant No.69833030 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant Nos.2001AA113010, 2001AA414020 (国家高技术研究发展计划); the National High Technology Development 973 Program of China under Grant No.G1999035807 (国家重点基础研究发展规划(973))

第一作者简介: 丁柯(1975－),男,江西南昌人,博士生,主要研究领域为分布式计算,分布事务处理技术.

过事务实例之间(细粒度)的冲突检测减少了工作流冲突的概率;另一方面通过事务类型之间(粗粒度)的冲突预测,允许多个将来不冲突的工作流执行不可补偿事务.

关键词: 事务工作流;并发控制;恢复;调度算法;锁粒度

中图法分类号: TP311 **文献标识码**: A

工作流是由若干个相互关联的任务组成的过程,它是通过计算机软件进行定义、执行并监控的业务过程[1].为了使计算机能够支持业务过程,就要分析业务过程,抽象出业务过程的本质特征,并使用一种计算机可处理的方式表示,其结果称为过程模型.某些应用领域的业务过程计算机自动化往往要求其中的某一个工作流作为一个整体要么正确提交,要么回滚结束,也就是说,工作流具有事务特性.于是提出了事务工作流(transactional workflow)的概念[2],即一个事务工作流的执行将系统从一个一致性状态转换到另一个一致性状态.

事务工作流由平面事务组合形成,其执行满足松弛原子性(relaxed atomicity)[3].由于某些平面事务不具有可补偿性(compensatability),因此一旦执行了这种平面事务,事务工作流将不能全局回滚.如果两个并发的事务工作流均执行了不可补偿的事务,并在随后的执行中产生循环冲突(详见本文第 2.1 节),那么系统就不能保证可串行化的调度,并且也不能回滚这两个事务.为了避免这种情况的发生,现有的调度算法只允许最多一个事务工作流执行不可补偿事务,因而极大地影响了并发度[4].

本文利用两种不同粒度的冲突检测机制来解决循环冲突问题,并进一步提高工作流执行的并发度.一方面,我们利用事务实例之间细粒度的冲突检测来减少工作流之间的冲突;另一方面,通过预测事务工作流将要执行的事务类型集合,并利用事务类型之间粗粒度的冲突检测来判断两个工作流是否会在将来发生冲突.本文第 1 节给出事务工作流模型,定义了在事务类型和事务实例上两种粒度的冲突关系,并形式化地定义了工作流的结构和执行.第 2 节首先分析循环冲突问题,然后提出基于混合粒度冲突检测的事务工作流调度算法.第 3 节是本文工作和其他相关工作的比较.最后是小结.

1 事务工作流模型

我们考虑的事务工作流采用两层系统模型,这两个层次分别解决业务过程需求中的不同问题.系统的底层提供具有严格 ACID 特性的平面事务,这些平面事务之间相互隔离.系统的高层提供具有松弛事务特性的事务工作流.对这些事务工作流而言,平面事务是黑盒方式的基本构造块[3].

事务工作流具有松弛事务特性,这是因为它放松了 ACID 平面事务的隔离性和原子性.由于业务过程之间需要协作,因此每个平面事务的执行效果在它提交后就对其他事务工作流可见,而无须等到工作流结束以后.工作流的回滚则根据应用语义,通过补偿事务(compensation transaction)来消除已提交事务的更新.

1.1 事务类型和事务实例

事务工作流的基本活动是事务.事务通常由一组操作组成,这些操作的组合模式实际代表了某种事务类型,而事务类型的某次具体执行则被称为一个事务实例.事务实例由事务类型和具体的输入参数组成.设 T 是一个事务类型,$t(a_1,...,a_r)$是 T 的一个实例,其中 $a_1,\ldots,a_r$ 是输入参数.在不产生歧义的情况下,事务实例 $t(a_1,...,a_r)$可简写成 t.

设$\mathcal{T}=\{t_1,t_2,t_3,\ldots\}$是所有事务实例组成的集合.下面我们给出事务的可补偿性和可重复性的严格定义.

定义 1. 设 $\sigma=\langle t_1t_2t_3...t_k\rangle$是一个事务序列,其中 $t_i\in\mathcal{T}$.如果对$\mathcal{T}$上的所有事务序列 α 和 β,串连的事务序列$\langle\alpha\ \sigma\ \beta\rangle$和$\langle\alpha\ \beta\rangle$的执行效果相同,那么称事务序列 σ 是无影响的(effect-free).

定义 2. 对事务类型 T的任何实例 $t(a_1,...,a_r)\in\mathcal{T}$,如果存在另一个事务类型 T^{-1}和它的实例 $t^{-1}(a_1,\ldots,a_r)\in\mathcal{T}$,使得事务序列 $\sigma=\langle t\ t^{-1}\rangle$是无影响的,则称 T^{-1}是 T 的补偿事务类型,并称 T 是可补偿的(compensatable);否则称 T 是不可补偿的.

为了能够形式化地定义可重复事务,我们用 $t^{(n)}$来表示事务 t 的第 n 次执行.

定义 3. 对事务类型 T的任何实例 $t\in\mathcal{T}$,如果存在一个正整数 $m\in$N,即使对所有的 $1\le j<m$,$t^{(j)}$都被放弃,$t^{(m)}$仍能确保提交,则 T 是可重复的(retriable).

为了保证事务的可补偿性,任何补偿事务类型 T^{-1} 必须是可重复的.如果事务类型 T 是可补偿的/可重复的,我们称其事务实例也是可补偿的/可重复的.

1.2 事务工作流的结构

基本活动通过若干控制结构可组合成复合活动.事务工作流本身就是一个复合活动.事务工作流使用的控制结构包括顺序、并行、条件、优先和循环结构:

$$A=T|(A_1\rightarrow A_2)|(A_1 \| A_2)|(A_1 \triangleright A_2)|(cond?A_1:A_2)|(cond[A_1]),$$

其中 T 是基本活动,即事务类型,A,A_1 和 A_2 是(复合)活动,$cond$ 是条件谓词.$A_1\rightarrow A_2$ 表示顺序结构,首先执行 A_1 然后再执行 A_2.$A_1 \| A_2$ 表示并行结构,两个并行分支分别执行 A_1 和 A_2.$A_1 \triangleright A_2$ 表示优先结构,它提供了一种向前恢复机制,在执行过程中总是试图先执行 A_1,若 A_1 执行失败,则执行 A_2,进行向前恢复.条件结构 $(cond?A_1:A_2)$ 由一个条件谓词 $cond$ 和两个互斥执行分支组成,当 $cond$ 为真时执行 A_1,否则执行 A_2.循环结构 $(cond[A_1])$ 表示当条件谓词 $cond$ 为真时,反复执行循环体 A_1.下面是一个事务工作流结构表达式的例子,其对应的图示如图 1 所示.

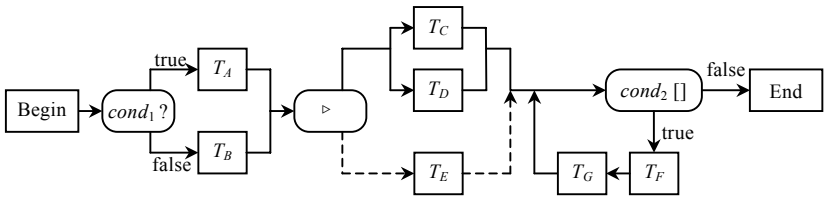

Fig.1 An example of transactional workflow

图 1 一个事务工作流的例子

例 1:$(cond_1?T_A:T_B)\rightarrow((T_C \| T_D)\triangleright T_E)\rightarrow(cond_2[T_F\rightarrow T_G])$.

1.3 事务工作流的执行和调度

事务工作流的执行是按照事务工作流的结构定义,依次执行其中的事务直至结束.事务工作流的执行也称为事务工作流的一个实例.

定义 4. 事务工作流的执行是一个二元组 $P_i=(\mathcal{T}_i,\varphi_i)$,其中 $\mathcal{T}_i\subseteq\mathcal{T}^*$ 是一个事务实例集合,$\ll_i$ 是定义在 $\mathcal{T}_i$ 上的偏序关系 $\ll_i\subseteq\mathcal{T}_i\times\mathcal{T}_i$,表示事务的执行次序:$t_1\ll_i t_2$ 当且仅当 t_1 在 t_2 之前执行.

在不引起歧义的情况下,事务工作流和事务工作流的执行在下文代表同一含义.我们分别用 a_i 和 c_i 来标记事务工作流 P_i 的放弃和提交.

定义 5. 两个事务实例 $t_i,t_j\in\mathcal{T}^*$,如果对 $\mathcal{T}^*$ 上的任意事务序列 α 和 β,都有事务序列 $\langle\alpha\ t_i\ t_j\ \beta\rangle$ 的执行效果和 $\langle\alpha\ t_j\ t_i\ \beta\rangle$ 的执行效果相同,那么事务实例 t_i 和 t_j 是可交换的;否则称这两个事务实例是不可交换的或者冲突的,记作 $t_i\ conf\ t_j$.

若事务 t_i 和 t_j 是冲突的,那么对任何 $\alpha,\beta\in\{-1,1\}$ 的组合,均有 $t_i^{\alpha}\ conf\ t_j^{\beta}$.类似地,如果事务 t_i 和 t_j 是可交换的,那么对任何 $\alpha,\beta\in\{-1,1\}$ 的组合,t_i^{α} 和 t_j^{β} 均可交换.

定义 6. 两个事务类型 T_i 和 T_j,如果存在 T_i 的实例 t_i 和 T_j 的实例 t_j,有 $t_i\ conf\ t_j$,那么事务类型 T_i 和 T_j 是冲突的,记作 $T_i\ conf\ T_j$.如果两个事务类型 T_i 和 T_j 的任何实例都不冲突,那么称 T_i 和 T_j 不冲突.

至此,我们分别基于事务类型和事务实例定义了两种粒度的冲突关系.

定义 7. 事务工作流的调度 S 是一个三元组 $(\mathcal{P}_S,\mathcal{T}_S,\ll_S)$,其中 $\mathcal{P}_S=\{P_1,\ldots,P_n\}$ 是一个事务工作流执行的集合,其中 $\mathcal{T}_S\subseteq\mathcal{T}^*$ 是 $\mathcal{P}_S$ 中工作流执行的事务实例集合,$\ll_S$ 是定义在 $\mathcal{T}_S$ 上的偏序关系,有 $\ll_S\subseteq\mathcal{T}_S\times\mathcal{T}_S$.偏序 $\ll_S$ 满足下面两个条件:

(1) $\forall P_i:\ll_i\subseteq\ll_S$,并且

(2) $\forall(t_i\in P_i,t_j\in P_j),i\neq j$,且 $t_i\ conf\ t_j$,那么或者 $t_i\ll_S t_j$ 或者 $t_j\ll_S t_i$.

1.4 可串行化和可恢复调度

定义 8. 如果下面条件成立,则事务工作流的调度 S 和 S' 是等价的:

(1) S 和 S' 定义在同样的事务工作流集合上,即 $\mathcal{P}_S=\mathcal{P}_{S'}$ 且 $\mathcal{T}_S=\mathcal{T}_{S'}$;

(2) S 和 S'对冲突事务的偏序是一致的,即如果 t_i 和 t_j 是调度 S 中的冲突事务,且 $t_i \ll_S t_j$,则有 $t_i \ll_{S'} t_j$.

定义 9. 如果事务工作流的调度 S 等价于某个串行的调度 S',则称 S 是可串行化的.

定义 10. 设 t_i 是事务工作流的执行 $P_i=(\Sigma_i,\ll_i)$的一个可补偿事务,即 $t_i \in \Sigma_i$,不可补偿事务 t_i^*被称为 t_i 的下一个不可返回点(next-point-of-no-return),如果 $t_i \ll_i t_i^*$,并且不存在不可补偿事务 t_i'使得 $t_i \ll_i t_i' \ll_i t_i^*$成立;若不存在这样的事务,则 c_i 是 t_i 的下一个不可返回点.

定义 11. 如果对调度 S 中的任何两个冲突事务 $t_i \in P_i$ 和 $t_j \in P_j$,t_i 是可补偿事务且 $t_i \ll_S t_j$,若下面条件成立,则 S 是可恢复的:

(1) $t_i^{-1} \ll_S t_j$ 或者 $t_i^* \ll_S t_j$,其中 t_i^*是 t_i 的下一个不可返回点事务;

(2) 如果 t_j 是可补偿的并且 $t_j^* \in S$,那么必然有 $t_i^* \ll_S t_j^*$,其中 t_j^*是 t_j 的下一个不可返回点事务;

(3) 如果 t_j 是不可补偿的,那么必然有 $t_i^* \ll_S t_j$.

直观上来说,如果 t_i $conf$ t_j 且 $t_i \ll_S t_j$,那么 P_j 的执行效果受 P_i 影响.可恢复调度要求一个事务工作流必须在影响它的其他事务工作流提交之后才能提交.一旦某个事务工作流执行了不可补偿事务,那么它就进入完成状态,它之前的补偿操作 t 不能再被补偿,因此与 t 冲突的事务也不能再被补偿.

2 事务工作流的调度算法

2.1 循环冲突问题

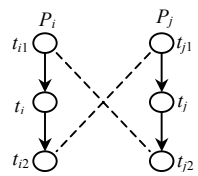

Fig.2 The cyclic conflict problem

图 2 循环冲突问题

如果允许两个不同事务工作流都执行不可补偿事务,那么可能产生不可串行化的调度.假设 $t_{i1},t_i \in P_i$ 和 $t_{j1},t_j \in P_j$,t_i 和 t_j 是不可补偿事务,在调度 S 中有 $t_{i1} \ll_S t_i$ 和 $t_{j1} \ll_S t_j$.如果 t_{i2} 和 t_{j2} 分别是 P_i 和 P_j 接下去要执行的事务(即 $t_i \ll_i t_{i2}$ 和 $t_j \ll_j t_{j2}$),并且 t_{i1} $conf$ t_{j2} 和 t_{j1} $conf$ t_{i2},那么就出现了一个循环冲突(如图 2 所示,其中箭头表示事务的执行次序,虚线表示事务冲突),S将不能满足可串行化,并且也不能恢复.正是由于这个问题,为了避免最坏情况的发生,目前的解决方法是一次只让一个事务工作流进入完成状态[4].

2.2 基于混合粒度冲突检测的调度

循环冲突问题出现的主要原因是不能预测工作流将来的执行情况,因此只能采用最保守的方法来避免循环冲突的发生.解决该问题的有效方法是预测事务工作流将来的执行情况.如果两个工作流将来不会发生冲突,那么它们可以安全地执行不可补偿事务,这也是我们开发的调度算法的基本出发点.

2.2.1 事务实例锁和后继事务类型集

每当执行某个事务实例时,工作流必须首先获取一个实例锁.事务实例 t 的实例锁用 lt 表示.如果事务实例 t_i 和 t_j 冲突,那么我们称实例锁 lt_i 和 lt_j 冲突,同样记作 lt_i $conf$ lt_j.

在事务工作流 P_i 的执行过程中,始终维护两个集合:事务实例锁集 LS_i 和后继事务类型集 FS_i,其中 LS_i 用来记录目前 P_i 获取的所有事务实例锁,FS_i 表示 P_i 将要执行的事务类型集.随着事务工作流 P_i 的执行,LS_i 不断变大,而 FS_i 不断变小.当 P_i 准备提交时,$FS_i=\varnothing$.根据事务实例锁集和后继事务类型集,我们可以判断两个工作流是否会在将来产生冲突.工作流 P_i 和 P_j 将来不冲突的充分条件是:(1) P_i 已执行的事务类型与 P_j 将要执行的事务类型不冲突;(2) P_j 已执行的事务类型与 P_i 将要执行的事务类型不冲突;(3) P_i 将要执行的事务类型和 P_j 将要执行的事务类型不冲突.因此有下面的引理 1.

引理 1. 事务工作流 P_i 和 P_j 在将来不产生冲突的充分条件是:

(1) $\forall lt_i \in LS_i, T_j \in FS_j: \neg(T_i\ conf\ T_j)$,其中 T_i 是 t_i 的事务类型;并且

(2) $\forall lt_j \in LS_j, T_i \in FS_i: \neg(T_i\ conf\ T_j)$,其中 T_j 是 t_j 的事务类型;并且

(3) $\forall T_i \in FS_i, T_j \in FS_j: \neg(T_i\ conf\ T_j)$.

为了实现事务实例之间的冲突检测,系统按照事务类型来组织冲突矩阵 CON,其中每个 $CON(i,j)$存放了一

个返回值为布尔值的冲突检测函数 $conflict_{ij}(a_1,...,a_r,b_1,...,b_s)$,用来判断事务类型 T_i 的实例 $t_i(a_1,...,a_r)$和事务类型 T_j 的实例 $t_j(b_1,...,b_s)$是否冲突.例如对银行帐户的存款事务 $deposit(account_1,amount_1)$和取款事务 $withdraw(account_2,amount_2)$而言,它们的冲突函数 $conflict(account_1,amount_1,account_2,amount_2)$判断两者访问的帐户是否相同,如果相同,则返回 TRUE 表示实例冲突,否则返回 FALSE 表示不冲突.

另外,如果两个事务类型 T_i 和 T_j 相互不冲突,那么 $CON(i,j)$存放常量 FALSE;如果 T_i 和 T_j 的所有实例都相互冲突,那么 $CON(i,j)$存放常量 TRUE.对引理 1 而言,当 $CON(i,j)$存放常量 FALSE 时,$\neg(T_i\ conf\ T_j)$取真值.

2.2.2 后继事务类型集的构造

在每执行一个事务实例之后,事务工作流的后继事务类型集就需要更新.本节给出后继事务类型集的计算方法.后继事务类型集的计算通过对事务工作流的结构进行分析而完成.

事务工作流的结构可以用一个树来表示,例如第 1.2 节中的例 1 可以用图 3 来表示.树的叶结点是事务类型,其他结点代表各种控制结构.每个结点有两个属性:*CS* 和 *FS*.*CS* 表示以该结点为根的子树中包含的所有事务类型集,*FS* 表示执行完该结点为根的子树后还要执行的事务类型集.那么,叶结点的 *FS* 就是我们关心的后继事务类型集.

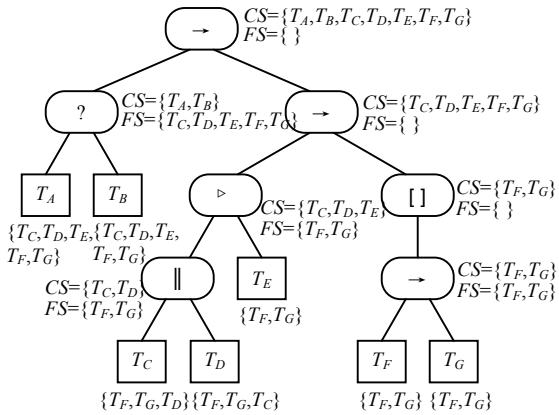

Fig.3 Construction of *FS*

图 3 *FS* 的构造

CS 集合的构造通过后序遍历树即可获得:任何一个结点的 *CS* 是其子结点 *CS* 的并集;而叶结点的 *CS* 是它本身.而 *FS* 通过先序遍历树获取.树根结点的 *FS* 为空集,其他结点的 *FS* 通过表 1 中的计算规则得出(其中 parent 表示父结点,left 和 right 分别表示它的左边子结点和右边子结点.另外,循环结构只有一个子结点).图 3 表示了每个结点的 *CS* 和 *FS* 集,其中叶结点只列出 *FS* 集.例如,当工作流执行完事务 T_D 后,它的 $FS=\{T_F,T_G,T_C\}$.

Table 1 The computation rule of *FS*

表 1 *FS* 的计算规则

Flow control construct	Construction of *FS*
Sequential construct	*left.FS=right.CS*∪*parent.FS*; *right.FS=parent.FS*
Conditional construct	*left.FS=right.FS=parent.FS*
Contingency construct	*left.FS=right.FS=parent.FS*
Parallel construct	*left.FS=right.CS*∪*parent.FS*; *right.FS=left.CS*∪*parent.FS*
Iterative construct	*left.FS=parent.CS*∪*parent.FS*

2.2.3 调度算法

每个新运行的工作流 P_i 都被附上一个单调递升的时间戳 $ts(P_i)$.系统为每个工作流 P_i 维护事务实例锁集 LS_i 和后继事务类型集 FS_i,并将所有活动的事务工作流动态地分成两个集合:可回滚工作流集合 RC 和不可回滚工作流集合 NR.如果某个工作流已执行的事务都是可补偿的,那么它被归于 RC 集合.如果某个工作流执行了至少一个不可补偿事务,那么系统将把它移至 NR 集合.我们开发的基于混合粒度冲突检测的调度算法遵循下面几个规则:

(R_1) 集合 NR 中的工作流不会等待其他工作流;

(R_2) 时间戳较小的工作流不会等待 RC 中时间戳较大的工作流;

(R_3) 集合 NR 中的工作流相互不会发生冲突.

由于循环冲突产生的前提条件是两个 NR 中的工作流相互冲突,规则 R_3 防止了循环冲突的出现.

图 4 表示了调度算法执行时的一个快照,其中事务工作流 P_1 和 P_4 正在等待 P_3,而 P_3 则在等待 P_6.注意,NR 中的集合不会冲突,也不会相互等待.

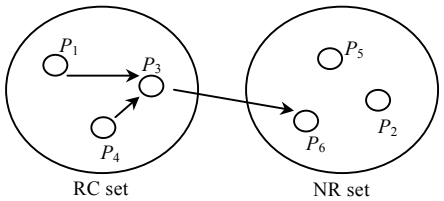

Fig.4 Scheduling of transactional workflows

图 4 事务工作流的调度

基于混合粒度冲突检测的调度算法详细描述如下:

(1) 工作流开始执行:当系统准备执行工作流 P_i 并且尚未执行它的任何事务时,系统将 P_i 放入 RC 集合.

(2) 事务实例执行:当事务工作流 P_i 执行事务实例 t_i 时,首先试图获取实例锁 lt_i,根据实例锁冲突检测,若存在下面两种冲突情况,则放弃相应的工作流 P_j:

(2.1.1) 如果 P_i∈NR,lt_i 与 RC 中的工作流 P_j 的 LS_j 冲突,那么放弃 P_j;

(2.1.2) 如果 P_i∈RC,lt_i 与 RC 中的工作流 P_j 的 LS_j 冲突,并且 $ts(P_i)<ts(P_j)$,那么放弃 P_j.

然后判断是否存在下面 3 种情况,如果存在,则 P_i 必须等待:

(2.2.1) lt_i 和一个时间戳较小的工作流或者 NR 中的工作流 P_j 的 LS_j 冲突,P_i 必须等待 P_j 结束.在 P_j 结束之后,P_i 再重新判断(2.2.1)~(2.2.3)的冲突情况.

(2.2.2) t_i 是一个不可补偿事务,并且 P_i 在将来会和 NR 中工作流 P_j 冲突(利用引理 1),那么 P_i 必须等待 P_j 结束.在 P_j 结束之后,P_i 再重新判断(2.2.1)~(2.2.3)的冲突情况.

(2.2.3) t_i 是一个不可补偿事务,并且 P_i 在将来会和某个处于(2.2.2)等待状态的工作流 P_j 冲突,并且 $ts(P_i)>ts(P_j)$,那么 P_i 必须等待 P_j 先进入 NR.P_j 进入 NR 后,P_i 再重新判断(2.2.1)~(2.2.3)的冲突情况.

P_i 获取了实例锁 lt_i 之后,系统即可执行 t_i.如果 t_i 是不可补偿事务,并且 P_i 在 RC 集合中,系统将 P_i 移至 NR 集合.

(3) 工作流提交:只有当事务工作流 P_i 和任何时间戳较小的事务工作流 P_j 都不冲突时,系统才能提交 P_i.系统提交工作流 P_i 时,将释放 LS_i 中的所有锁.如果 P_i 和时间戳较小的某个工作流 P_k 冲突,即 $ts(P_k)<ts(P_i)$,那么 P_i 必须等待 P_k 提交后才能提交.P_i 一旦提交完成,那么从相应的 RC 集合或 NR 集合中移出.

(4) 工作流放弃:当事务工作流 P_i 放弃时,对每个已执行的可补偿事务 t_i,按其执行的相反次序执行补偿事务 t_i^{-1}.当全部事务都被补偿后,工作流释放 LS_i 中的所有锁.如果工作流是由于冲突原因而被放弃,那么系统将重新运行该工作流,并且使用原来的时间戳.注意,只有 RC 集合中的工作流才会被放弃.P_i 一旦放弃完成,那么从相应的 RC 集合或 NR 集合中移出.

随着 NR 集合中工作流的执行和完成,这些工作流的 FS_i 不断变小,越来越多的 RC 集合中的工作流可允许进入 NR 集合.由于算法始终遵循 R_1 和 R_2 规则,因此工作流之间不存在循环等待,因而可以防止调度死锁现象的出现.虽然 RC 中的某个工作流 P_i 可能等待时间戳较大的 NR 中的工作流,但算法中的(2.2.3)能确保 P_i 最终进入 NR 集合,因此不会出现饿死现象.但是该算法不能避免连锁放弃,即如果某个事务被放弃,那么所有等待它完成的事务也被放弃.

3 相关工作和讨论

数据库系统中的并发控制和恢复是一个经典问题,根据数据读写操作之间的冲突规则,人们提出了若干解决方法来保证可串行化调度和可恢复调度[5].当具有丰富语义的操作引入数据库之后,它们之间的冲突关系由可交换性决定,R.Vingralek 等人为该情况下的并发控制和恢复的正确性提出了统一的理论框架[6].

有关事务工作流的研究来源于扩展事务模型[7].事务工作流由若干平面事务组成,这些平面事务之间的冲突关系由可交换性决定.文献[8]研究了事务工作流的并发控制问题,但是他们假定工作流完全由可补偿事务组成,因而不能很好地反映现实应用环境.A.Zhang 等人提出的 Flexible 事务是为实现异构多数据库事务管理的一种扩展事务模型[9].一个 Flexible 事务包含多个子事务,这些子事务通过 precedence 和 preference 两种约束关系组合在一起.子事务按其完成保证(termination guarantee)分成可补偿事务、可重复事务和 pivot 事务 3 种类型.Flexible 事务的子事务之间的冲突根据它们的读写数据集来判断.由于组成事务工作流的事务是黑盒,其读写数据集不可知,因此文献[9]中提出的调度方法不适用于事务工作流.H.Schuldt 等人研究了事务性过程

(transactional process)的并发控制和恢复处理问题[10].在文献[4]中,H.Schuldt 提出一种基于有序共享锁(ordered shared lock)的调度算法.为了避免本文第 2.1 节中提出的循环冲突问题,该算法在全局范围内最多只允许一个事务工作流能够执行不可补偿事务,大大限制了工作流执行的并发度.

以上文献中事务间的冲突规则都是基于事务类型来判断的,是一种粗粒度的冲突检测方法.本文给出的事务工作流模型区分事务类型冲突和事务实例冲突,通过运行时刻进行基于事务实例的冲突检测,可以显著地减少工作流之间的冲突.另外,本文的调度算法在工作流执行过程中,通过预测将来执行的事务类型集合,允许多个不冲突的工作流执行不可补偿事务.因此本文提出的算法从两方面提高了事务工作流执行的并发度.

4 结束语

事务工作流是满足松弛事务特性的工作流.在事务工作流引擎中,调度算法被用来控制并发事务工作流的执行,产生可串行化和可恢复调度.由于组成事务工作流的平面事务具有不同的完成特性,为了防止不可串行化的执行,在没有预测时,系统只能允许最多一个事务工作流执行不可补偿事务,限制了事务工作流执行的并发度.

本文首先给出事务工作流模型,在模型中区分事务类型和事务实例,定义了两种粒度的冲突关系,并给出了事务工作流结构和执行的形式化定义.在事务工作流的执行过程中,调度器为每个活动事务工作流维护两个集合:事务实例锁集 LS_i 和后继事务类型集 FS_i.事务实例锁集 LS_i 包含所有已经获取的事务实例锁,用于细粒度的事务实例冲突检测;而后继事务类型集 FS_i 包含所有将要执行的事务类型,用于粗粒度的事务类型冲突检测.本文给出了后继事务类型集 FS_i 的构造方法.通过结合两种粒度的冲突检测,本文提出的调度算法在两方面提高了并发度:一方面,细粒度冲突检测减少了事务工作流之间的冲突概率;另一方面,粗粒度的冲突检测能够预测事务工作流是否会在将来发生冲突,而不会发生冲突的工作流可以安全地并发执行,从而解决一次只能执行一个不可补偿事务的问题.本文提出的事务工作流模型考虑了平面事务不同的可补偿性和可重复性,能够较好地适应现实的应用环境.

References:

[1] Luo HB, Fan YS, Wu C. Workflow technology survey. Journal of Software, 2000,11(7):899~907 (in Chinese with English Abstract).

[2] Alonso G, Agrawal D, Abbadi AE, Kamath M, Gunthoer R, Mohan C. Advanced transaction models in workflow contexts. In: Proceedings of the International Conference on Data Engineering. 1996. 574~581.

[3] Grefen P, Vonk J, Boertjes E, Apers P. Semantics and architecture of global transaction support in workflow environments. In: Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems. 1999. 348~359.

[4] Schuldt H. Process locking: a protocol based on ordered shared locks for the execution of transactional processes. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS 2001). Santa Barbara: ACM Press, 2001. 289~300.

[5] Bernstein P, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[6] Vingralek R, Hasse-Ye H, Breitbart Y, Schek H-J. Unifying concurrency control and recovery of transactions with semantically rich operations. Theoretical Computer Science, 1998,(190):363~396.

[7] Jajodia S, Kerschberg L. Advanced Transaction Models and Architectures. Kluwer, 1997.

[8] Li HC, Shi ML, Chen XX. Concurrency control algorithm for transactional workflows. Journal of Software, 2001,12:1~9 (in Chinese with English Abstract).

[9] Zhang A, Nodine M, Bhargava B. Global scheduling for flexible transactions in heterogeneous distributed database systems. IEEE Transactions on Knowledge and Data Engineering, 2001,13(3):439~450.

[10] Schuldt H, Alonso G, Schek H. Concurrency control and recovery in transactional process management. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99). Philadelphia: ACM Press, 1999. 316~326.

附中文参考文献:

[1] 罗海滨,范玉顺,吴澄.工作流技术综述.软件学报,2000,11(7):899~907.

[8] 李红臣,史美林,陈信祥.事务工作流的并发控制算法.软件学报,2001,12:1~9.