공학 연구논문집 제22권 제2호 pp. 107~125, 1991. 울산대학교
Journal of Engineering Research Vol. 22, No. 2, pp. 107~125, 1991.
University of Ulsan.

순차 트랜잭션 처리를 위한 새로운 그래프 모델에 의한 교착상태의 주기적 검출 및 회복 기법*

박 영 철
전자계산학과

〈요 약〉

이 논문은 엄격한 두 단계 로킹과 다단계 로킹규약이 다섯 가지 로크 모드(IS, IX, S, SIX 그리고 X)를 적용하는 환경에서의 순차 트랜잭션 처리에 있어서의 교착상태 문제를 다룬다. 우리의 스케쥴링 정책은 로크의 변경을 제외하고는 먼저온 순서대로 로크 요구를 취급한다. 기본 도구로, 교착상태에 대한 시스템의 정확한 상태를 파악하기 위하여, H/W-TWBG라는 새로운 지향그래프를 소개한다. 그 그래프는 여러가지 바람직한 특성을 가지고 있어서 대기그래프를 대신하여 사용할 수 있다. H/W-TWBG의 특성들이 제시되고 이론적인 측면들이 해석되었다. H/W-TWBG에 기초를 두고, 교착상태 원에서 희생자 후보들의 판별 원리를 설정하고, 그리고 타당한 시간 및 면적 복잡성을 가진 새로운 주기적인 교착상태의 검출 및 회복 알고리즘을 제시한다. 우리의 교착상태 검출기법의 한 중요한 특징은 어떤 교착상태들은 어느 트랜잭션의 포기 없이도 해결될 수 있다.

A Periodic Deadlock Detection and Resolution Algorithm with a New Graph Model for Sequential Transaction Processing

Young-Chul Park
University of Ulsan

〈Abstract〉

This paper addresses the deadlock problem in the sequential transaction

* This paper is supported in part by Asan Foundation.

processing where the strict two-phase locking and the multiple granularity locking protocol with five lock modes (IS, IX, S, SIX and X)are employed. Our scheduling ploicy honors lock requsts in a first-in-first-out basis except for lock conversions. As a basic tool, we introduce a new directed graph model called Holder/Waiter-Transaction Waited-By Graph (H/W-TWBG) to capture precise status of systems in terms of deadlock. The graph has a number of desirabe properties which allow us to use instead of the wait-for graph. The properties of H/W-TWBG are presented and theoretical aspects of it are also analyzed. Based on H/W-TWBG, we establish the identification principles of victim candidates in a deadlock cycle, and then present a new periodic deadlock detection and resolution algorithm which has a reasonable time and storage complexity. One important feature of our deadlock resolution scheme is that some deadlocks can be resolved without aborting any transaction.

1. Introduction

A transaction in database management systems is a sequence of database operations which is atomic with respect to the recovery, thereby transforms the database from one consistent state to another consistent state. Each transaction requests at most one lock at a time, and when a request cannot be granted immediately, the transaction is blocked until either the request is granted or the transaction itself is aborted by some reasons (deadlock or site failure). Deadlocks are usually characterized in terms of a *transaction wait-for graph*(TWFG). The TWFG is a directed graph where each vertex represents a transaction; each edge of the form $T_i \rightarrow T_j$ means that transaction T_i is waiting for the completion of T_j. It has been shown that there exists a deadlock if and only if there is a cycle in TWFG [11].

Agrawal et al. [1] present a periodic deadlock detection algorithm which has $O(n)$ storage and time complexity (Chin[6] modified this algorithm for the correct detection of deadlocks) where n is the number of transactions in the system considering only exclusive lock modes (X locks). They extend the algorithm to allow shared lock modes(S locks) as well as X locks. In order to maintain the same time and space complexity however, only one of the readers, say T_j, is selected to represent a wait-for relationship $T_i \rightarrow T_j$ in TWFG even though a writer Ti is blocked by multiple readers. Because of that, detection of some deadlocks can be delayed and some transactions may hold resources or wait for other

transactions unnecessarily even they might be aborted later. In [2,3] they permit each transaction to have multiple wait-for edges but do not show the cycle detedtion scheme and the victim selection strategy.

Jiang [14] proposes a continuous algorithm to cure those problems of Agrawal et al. [1] while supporting the same lock modes. The TWFG in his algorithm is represented by a $(n+1)^* n$-matrix, and $O(e)$ time is required for finding a cycle and listing all participators in the cycle, where n is the number of transactions and e is the number of wait-for relationships in the grph. In general, his algorithm is restricted to the continuous case. Furthermore when a deadlock is involved in multiple cycles and each cycle has a different cycle length, to list all participators in each cycle he shows that it works in an exponential function of the number of transcations $O(3^{n/3})$ in the worst case.

Elmagarmid [8] develops a continuous algorithm for multiple outstanding lock requests with the same lock modes as the above two algorithms. Because a special case of that can be the conventional transaction mode, a slight modification of his algorithm can make his to be compared to the others. Instead of TWFG, two tables are maintained in his algorithm, namely T-table and R-Table. R-table keeps all the blocked transactions with their requesting resources and lock modes. R-table keeps all the resources which are currently held by recording holding transactions with their requesting lock modes. His algorithm works in $O(n+e)$ space and time complexity where n is the number of blocked transactions and e is the number of edges in the tables. However, his resolution scheme which always aborts the current blocker whenever there is a deadlock is simple but far from being optimal [2]. Moreover, the whole T-table has to be searched to schedule some waiting requests whenever all the holders of a resource are completed, because each resource being locked does not contain its own queue of blocked requests. The scheduling policy might be unfair and indicates the possiblity of live-lock [5].

The motivation of our research is due to the fact that a work which demonstrates the graph construction schemes when multiple lock modes as well as lock conversions are supported, has not been found in the literature to our best knowledge. In contrast to the aforementioned schemes, the periodic algorithm proposed here as a companion of the continuous one [17] permits lock conversions, allows multiple lock modes and provides victim selection strategies while maintaining a reasonable time and space complexity. The rest of this paper is organized as follows. In Section 2, the transaction model and the lock modes under consideration are explained. Our systematic scheduling policy based on one important observation is discussed in Section 3. Section 4 introduces a new deadlock

detection graph model and provides the identification principles of the viction selection strategies as well as deadlock detection and resolution scheme are presented in Section 5. Section 6 contains some concluding remarks.

2. Preliminaries

A transaction in this paper is a sequence of database operations which has four well known ACID properties:atomicity, consistency, isolation and durability [13]. For ensuring serializability, our model takes *strict two-phase locking* [5, 7] which requires that a transaction has to lock a resourece before it accesses the resource and all locks of a transaction are held until the transaction terminates. We support *multiple lock modes*(IS, IX, S, SIX and X locks) [5, 10, 11] and our model is upward compatible with the *multiple granularity locking* (MGL) protocol [10, 11] in a sense that it integrates without changes into a system that supports a resource hierarchy.

Two lock requests for the same resource by two different transactions are said to be *compatible* if they can be granted concurrently. The *compatiblity matrix*, say *Comp*, is shown in Table 1 where NL means No Lock. Comp [lock1, lock2] is true (false) if lock 1 and lock 2 are compatible (incompatible, respectively). For example, Comp[S, IS] is true but Comp[IX, SIX] is false.

	NL	IS	IX	SIX	S	X
NL	t	t	t	t	t	t
IS	t	t	t	t	t	f
IX	t	t	t	f	f	f
SIX	t	t	f	f	f	f
S	t	t	f	f	f	f
X	t	f	f	f	f	f

Table. 1 Compatibility Matrix

	NL	IS	IX	SIX	S	X
NL	NL	IS	IX	SIX	S	X
IS	IS	IS	IX	SIX	S	X
IX	IX	IX	IX	SIX	SIX	X
SIX	SIX	SIX	SIX	SIX	SIX	X
S	S	S	SIX	SIX	S	X
X	X	X	X	X	X	X

Table. 2 Conversion Matrix

A transaction which holds a resource might re-request the same resource to convert a lock from the granted mode to a more exclusive mode. We call such re-requests *lock conversions*. When a request turns out to be a conversion, the granted mode of the requestor and the newly requested mode are used to compute the new mode by use of a *conversion matrix*, say *Conv*. It is represented by Table 2 with the granted mode as row and the requested mode as column. For example, when a transaction holds an IX lock on a resource an re-requests an S lock for the resource, the transaction eventually wants to hold an SIX lock (conv[IX,

S]) for the resource.

In order to implement our scheduling policy in which lock requests are satisfied in a first-in-first-out basis and to keep track of the requests of each transaction, *the lock* manager maintains a lock table which holds the following information for each resource being locked:a holder list, a queue and a *total mode* (*tm*) of the holders. Each holder in a holder list takes three attributes:a transaction identifier (*tid*), a granted mode (*gm*) and a blocked mode (*bm*), i.e. (*tid*, *gm*, *bm*);each request in a queue takes two attributes:a transaction identifier (*tid*) and a blocked lock mode (*bm*), i.e. (*tid*, *bm*);and the total mode of the holders is definde as Conv [Conv[Conv[gm_1, bm_1], gm_2],..., bm_n], assuming that n requests are in the holder list and each one takes (T_i, gm_i, bm_i) where $1 \leq i \leq n$.

The notion of the total mode is introduced to check the grantability of lock requests. The total mode is different from the *group mode* [11] in that the latter is defined as conv[... Conv[Conv[gm_1, gm_2], gm_3], ..., gm_n] in our context. We hope that the reader shall understand why the total mode is more efficient than the group mode after reading Section 3.

3. Scheduling Policy Revisited

A transaction can request resources while running. If a requested resource is held by other transactions with conflicting lock modes or the corresponding queue is not empty, then the request-ing transaction is suspended(i.e. blocked) until the request is satisfied. More specifically, when a request from a transaction T_i asking a lock L_i for a resource R_x arrives, the holder list of R_x is checked first to see if the request is a lock conversion (i.e. T_i already holds a lock). In the case of a new requestor (i.e. the request is not a lock conversion), the queue status of R_x is checked. If the queue is not empty, then the request is not granted by appending (T_i, L_i) to the end of the queue. If the queue is empty and the requested mode L_i is compatible with the total mode *tm* of R_x, then T_i is notified that the request is granted after placing (T_i, L_i, NL) to the holder list and updating tm of R_x by Conv[*tm*, L_i]. Otherwise, (T_i, L_i) is appended to the queue. When a request turns out a lock conversion (i.e. there is an entry (T_i, gm_i, bm_i) in the holder list where bm_i is NL), a new mode is computed as Conv[gm_i, Li]. If the new mode is compatible with the granted mode of all the other holders, then T_i is notified that the request is granted after substituting the new mode for gm_i and updating *tm* of R_x by Conv[*tm*, L_i]. Otherwise, transaction T_i is blocked by substituting the new mode for bm_i and *tm* of R_x is updated by Conv[*tm*, L_i].

Example 3.1 Assume that a resource R_1 is held by two transactions (T_1 with IS lock and T_2 with IX lock)

which render the total mode of R_1 IX lock and at the same time is waited by two other transactions (T_3 with S lock and T_4 with X lock) to be released in the queue. That situation can be visualized as follows.

R1[IX]: Holder((T_1, IS, S) (T_2, IX, NL) Queue((T_3, S) (T_4, X))

Suppose that transaction T_1 re-requests S lock for R_1. That request cannot be granted because S lock (Conv[IS, S]) is incompatible with IX lock (the granted mode of transaction T_2). The situation becomes as follows.

R1[IX]: Holder((T_1, IS, S) (T_2, IX, NL) Queue((T_3, S) (T_4, X))

A transaction cannot request another resource when being blocked. That is, a transaction whose request is not satisfied cannot proceed. However, there are two cases where we need to check whether there are some blocked requests which can be granted and, if grantable we need to grant those requests at this point. The first case is that a member of the holder list is forced out either due to the commit or the abort. The second case is that the first member of the queue leaves the system due to the abort. When any one of the two cases takes place, we want to check the blocked requests in a systematic and efficient way. In order to present the detailed scheme some preliminary materials need to be presented first.

Observation 3.1 Assume that there are two blocked requests (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) in a holder list due to lock conversions.

(1) if Comp[bm_i, bm_j], one of them can be scheduled in the presence of another.

(2) If Comp[bm_i, gm_j] and not Comp [gm_i, bm_j], T_i can be scheduled before T_j but the reverse is impossible.

(3) If not Comp[bm_i, gm_j] and not Comp[bm_j, gm_i], none of them can be scheduled in the presence of another (this is a kind of deadlock).

Based on Observation 3.1, when a conversion is blocked, the position of the requestor (T_i, gm_i, bm_i) in the holder list, after substituting Conv [gm_i, requesting lock mode] for bm_i, is rearranged according to the following *upgrader positioning rule* (UPR):

(1) if there are some requests whose *bm* are not NL and are compatible with bm_i, put (T_i, gm_i, bm_i) right before the first request among them.

(2) If UPR(1) cannot be applicable and there are some requests whose *gm* are compatible with bm_i and whose bm are not compatible with gm_i, put (T_i, gm_i, bm_i) right before the first request among them.

(3) If UPR(1) and (2) cannot be applicable, put (T_i, gm_i, bm_i) after all requests whose bm are not NL and before all requests whose bm are NL.

Theorem 3.1 Let (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) be two blocked requests in the holder list of a resource where (T_i, gm_i, bm_i) precedes (T_j, gm_j, bm_j)

according to UPR. If the request of T_i cannot be granted, then that of T_j cannot be either.

proof: Let us consider each conditions of UPR one by one.

(1) Assume UPR-1 is applied. Because a blocked lock mode can take one of {IX, S, SIX, X} and bm_i and bm_j are compatible accordng to the rule, either bm_i and bm_j are IX or bm_i and bm_j are S. The theorem follows.

(2) Assume UPR-2 is applied. That means that the request of T_j cannot be granted in the presence of T_i.

(3) Assume UPR-3 is applied. There are two cases to satisfy UPR-3. The first case takes following conditions: not Comp[gm_i, bm_j] and not Comp[bm_i, gm_j]. That implies none of them can be granted in the presence of the other (i.e. a deadlock). So, trivial. The second case takes following conditions: not Comp [bm_i, bm_j], Comp[gm_i, bm_j] and Comp [bm_i, gm_j]. From these conditions we can simply derive that bm_i and bm_j can take one of {IX, S, SIX}. Because bm_i and bm_j are incompatible, the combination (bm_i, bm_j) or (bm_j, bm_i) can have one of {(IX, S), (IX, SIX), (S, SIX)}. Considering that T_i and T_j are blocked at the holder list of a resource, there should be at least one holder (T_k, gm_k, bm_k) such that gm_k is incompatible with both or bm_i and bm_j. Therefore, gm_k is an SIX lock for (IX, X) combination, an S or an SIX for (IX, SIX) and and IX or an SIX for (S, SIX). This gives us that both gm_i and gm_j can have IS locks only. In that environment, when the last holder which block both of the requests of T_i and T_j leaves the holder list, any one of them can be granted. Otherwise, none of them can be granted.

QED

When a holder is deleted from the holder list of a resource R_x, the total mode (tm) of R_x is recomputed and the process (checking whether some blocked request can be granted) can start at the front of the holder list all the way down to the end as well as can stop immediately when either one cannot be granted anymore or a non-blocked one is found. All the newly granted ones are put after the blocked holders after substituting the blocked mode (NL) for the granted mode (the blocked mode) respectively. In addition to that, if the queue of R_x is not empty, the queue is checked to grant some requests from the first waiter, say (T_i, bm_i). Once the tm of R_x is compatible with the bm_i, the request is granted by placing (T_i, bm_i, NL) to the holder list and the tm of Rx is updated by Conv[tm, bm_i]. That process continues until either the queue becomes empty or the total mode of R_x is not compatible with the waiter's blocked mode. When the first member of the queue leaves the system, after deleting the request (i.e. the second member becomes a new first member), the same process is done for the queue.

There is another important point of UPR which should be precisely

stated. Our UPR greatly influences our graph(H/W-TWBG) construction shceme which is presented in the next section so that H/W-TWBG has a number of desirable properties.

4. H/W-TWBG: A New Graph Model

In this section, we introduce a new directed graph called a Holder/Waiter-Transaction Waited-By Graph(H/W-TWBG). Each vertex of H/W-TWBG represents a transaction identifier and each edge $T_i \rightarrow T_j$ is labeled with H/W, where the completion of transaction T_i is waited by transaction T_j and either T_i is a holder of the resource which T_j is waiting(H-label) or another waiter in the queue of the resource(W-label). Edges in H/W-TWBG are constructed by the following *edge construction rules* (ECR):

(1) Let two requests in the holder list of a resource be (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) such that (T_i, gm_i, bm_i) precedes (T_j, gm_j, bm_j). If not Comp(gm_i, bm_j) or not Comp(bm_i, bm_j), then add an edge $T_i \rightarrow T_j$ with H-label. If not Comp(gm_j, bm_i), then add an edge $T_j \rightarrow T_i$ with H-label.

(2) For each request (T_i, gm_i, bm_i) in the holder list of a resource, let the first request, if any, in the queue of the resource whose blocked mode is not compatible with either gm_i or bm_i be (T_j, bm_j). Add an edge $T_i \rightarrow T_j$ with H-label.

(3) Let two adjacent requests in the queue of a resource be (T_i, bm_i) and (T_j, bm_j) such that (T_i, bm_i) precedes (T_j, bm_j). Add an edge $T_i \rightarrow T_j$ with W-label.

In H/W-TWBG, each H-labeled edge is followed by a sequence (possibly zero) of W-labeled edges. We call a path which consists of one H-labeled edge and all of its following W-labeled edges, possibly empty, a Transaction Resource Request Path (TRRP). A TRRP shows a partial status of the holder list and the queue of a resource. It is worthwhile at this point to note a number of properties of H/W-TWBG. Because the theoretical analysis on H/W-TWBG is found in Appendix, we only list the properties here.

(1) No cycle can be made without any H-labeled edge.

(2) No cycle can be made with only one TRRP.

(3) Each cycle in H/W-TWBG consists of at least two TRRPs.

(4) There exists a cycle in H/W-TWBG if and only if there exists a deadlock in the database system.

Due to the third and the fourth properties, the deadlock detection problem corresponds to finding those TRRPs which constitute a cycle.

Definition 4.1 Let two consecutive TRRPs in H/W-TWBG be TRRP1 (Ti*...*Tj) and TRRP2(Tj*...*Tk) where Tj is blocked at a resource Rx. The TRRP disconnection rule (TDR) is defined as follows.

(1) Abort Tj, or

(2) If the last edge of TRRP1 is W-labeled and the blocked mode of Tj is

compatible with the total mode of Rx, then starting from the first request up to the request of Tj in the queue of Rx, let the set of requests whose blocked modes are compatible (incompatible) with the total mode of R_x be AV(ST, respectively). Reposition those requests in ST right after all the requests in AV in the queue.

Lemma 4.1 Once TDR-2 is applied to a resource R_x, those requests in AV cannot be involved in any deadlock cycle.

proof: Let those requests in AV be (T_1, bm_1), (T_2, bm_2),...and (T_r, bm_r) in the sequence of the queue. T_1 cannot be in any deadlock cycle because T_1 is not blocked at any other resource and bm_1 is compatible with the total mode of resource R_x. Because T_{i-1} is not involved in any deadlock cycle and the only incomming edge of T_i is from T_{i-1}(because all the requests in AV are repositioned not to have any incomming edge from any holder of R_x), T_i cannot be involved in any deadlock cycle$(1 \le i \le r)$. QED

Theorem 4.1 A deadlock cycle in H/W-TWBG can be resolved by applying TDR.

proof: Application of TDR-1(i.e. aborting a transaction) is straightforward and Lemma 4.1 justifies the application of TDR-2. QED

As for our victim selection strategy, the victim candidate and its cost are defined as follows: When TDR-1 is applied, the transaction to be aborted is the victim candidate and the summation of the cost of each transaction in ST divided by 2 is the cost of the candidate because those transactions in ST are not aborted but their executions are simply delayed. Among multiple victim candidaes in a cycle, one with the minimal cost is selected as a victim and when the victim comes from TDR-1(TDR-2), the corresponding transaction is aborted(the resqusts in ST are repositioned right after those of AV, respectively).

Example 4.1 Assume the following situation.

R1[SIX]; Holder((T_1, IX, SIX) (T_2, IS, S) (T_3, IX, NL) (T_4, IS, NL))

Queue((T_5, IX) (T_6, S) (T_7, IX))

R2[IS]: Holder((T_7, IS, NL)) Queue ((T_8, X) (T_9, IX) (T_3, S) (T_4, X))

According to UPR-2 given in Section 3, the entry of T1 precedes that of T2 in the holder list. Note that T4 does not block any request and the request of T2 cannot be granted in the presence of T1. Figure 4.1 shows H/W-TWBG for the above situation.

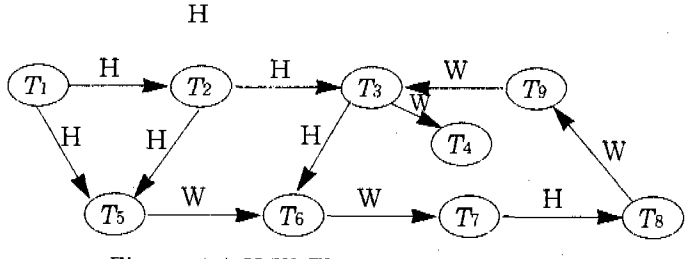

Figure 4.1 H/W-TWBG for Example 4.1

There are four cycles in Figure 4.1. Let's consider one of them which consists of four TRRPs: (T_1, T_2), (T_2, T_5, T_6, T_7), (T_7, T_8, T_9, T_3) and (T_3, T_1). According to TDR, there are four victim candidates from TDR-1 $\{T_1, T_2, T_7, T_3\}$ and there is one victim candidate from TDR-2 $\{T_8\}$. When one of $\{T_7, T_3\}$ is aborted or the request of T_8 is repositioned right after that of T_3 in the queue, all the deadlocks are resolved. When one of $\{T_1, T_2\}$ is aborted however, there still exists some deadlocks. Note that the remaining cycles are not newly formulated one but they were already there. When T_8 is selected as a victim, AV is $\{(T_9, IX), (T_3, S)\}$ and ST is $\{(T_8, X)\}$. By repositioning (T_8, X) right after (T_3, S) in the queue, the following modified situation for resource R2 can be obtained.

R2[IS]:Holder((T7, IS, NL)) Queue ((T9, IX) (T3, S) (T8, X) (T4, X))

According to our scheduling policy, the request of T_9 is granted but that of T_3 cannot be granted.

R2[IX]:Holder((T9, IX, NL) (T7, IS, NL)) Queue((T3, S) (T8, X) (T4, X))

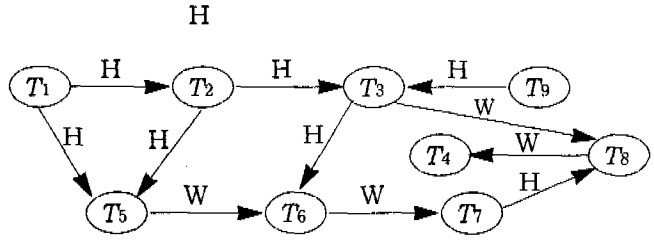

Figure 4.2 H/W-TWBG for the modified situation

The modified situats another H/W-TWGBG as shown in igure 4.2. Note that there is not any cycle in the figure.

5. Periodic Deadlock Detection and Resolution

In the periodic deadlock detection and resolution, the deadlock detection is done periodically different from the continuous case where the existence of a deadlock is checked and resolved whenever a lock request cannot be granted immediately[1, 5, 11]. An immediate question, therefore may be how the period has to be decided. By increasing the periodic interval, the cost of deadlock detection decreases but it will detect deadlocks late[5, 11]. The decision of the period is not of our main concern here, so we will focus on how fastly and correctly can we detect deadlocks and select victims to resolve deadlocks.

There can be several criteria for deciding a cost of each transaction, for example, number of locks it holds, starting time of it, the

amount of CPU and I/O time which has beem consumed and so on [2, 3, 5]. We assume that the cost of each transaction is determined by some combination of the above metrics and it is stored in a cost-table such that Cost(T_i) indicates the cost of transaction T_i. It is known to be NP-hard to find the minimal cost victims to break all cycles in a directed graph [2, 11]. Consequently, we presents a periodic deadlock detection and resolution algorithm whose time and space requirements are resonable and its solution is near optimal. Before we explain our periodic-detection-resolution algorithm, we need to present the internal data structure for the implementation of our scheduling policy and of H/W-TWBG.

Our internal structure called the lock table consists of two tables, namely the *resource status table* (RST) and the *transaction status table*(TST). The RST which has an entry for each locked resource is an array [0..M-1] of record with *rid* (resource identifier), tm(total mode), queue(queue pointer) and holder (holder list). Each holder of a resource is represented by a record with tid(transaction identifier), gm (granted mode), bm(blocked mode) and next(pointer to the next holder). Each transaction is assigned an integer value between 1 and N as its identifier which is used as an index for an entry in TST. The TST is an array [1..N] of record with ancestor, pr, waited and current. The variable waited maintains a linked list to keep all the incident edges of a vertex in H/W-TWBG and the variable pr keeps the position of a resource in RST where the corresponding transaction is blocked in the queue. The variables ancestor and current which are used for detection of cycles will be explained shortly.

Each edge in H/W-TWBG is represented by a record with lock(lock mode), tid(transaction identifier) and next(pointer to the next edge). More specifically, the edge $T_i \rightarrow T_j$ with H-label in H/W-TWBG is represented by an entry(NL, T_j, next) in TST[i]. waited. The W-labeled edges in H/W-TWBG are internaly represented by TST as follows: Let a resource say Rx have its entry in RST[k].

(1) TST[k].queue takes 0 or the identifier of the first transaction in the queue.

(2) For each request(T_i, bm_i) in the queue of R_x, TST[i].pr is set to k and an edge(bm_i, T_j, next) is put on the list of TST[i].waited if the request is followed by another request (T_j, bm_j). This is an edge $T_i \rightarrow T_j$ with W-label in H/W-TWBG as a matter of fact. However, for the last request in the queue of Rx, TST[i].pr is set to k and an edge(bm_i, 0, next) is put on the list of TST[i].waited.

It should be noted at this point all W-labeled edges in H/W-TWBG are present all the time in the system because the queue status needs to be maintained continulusly for our scheduling policy. Different from that, all H-labeled edges in H/W-TWBG are made to be present in TST

while our periodic-detection-resolution algorithm is activated. For maintain-ing the queue of a resource easily and to apply TDR systematically in our periodic-detection-resolution algorithm, when multiple entries are put on the list maintained by a waited in TST, the edge whose lock is not NL is put at the front of the list. For the situation given in Example 4.1, the corresponding RST and TST are shown in Figure 5.1.

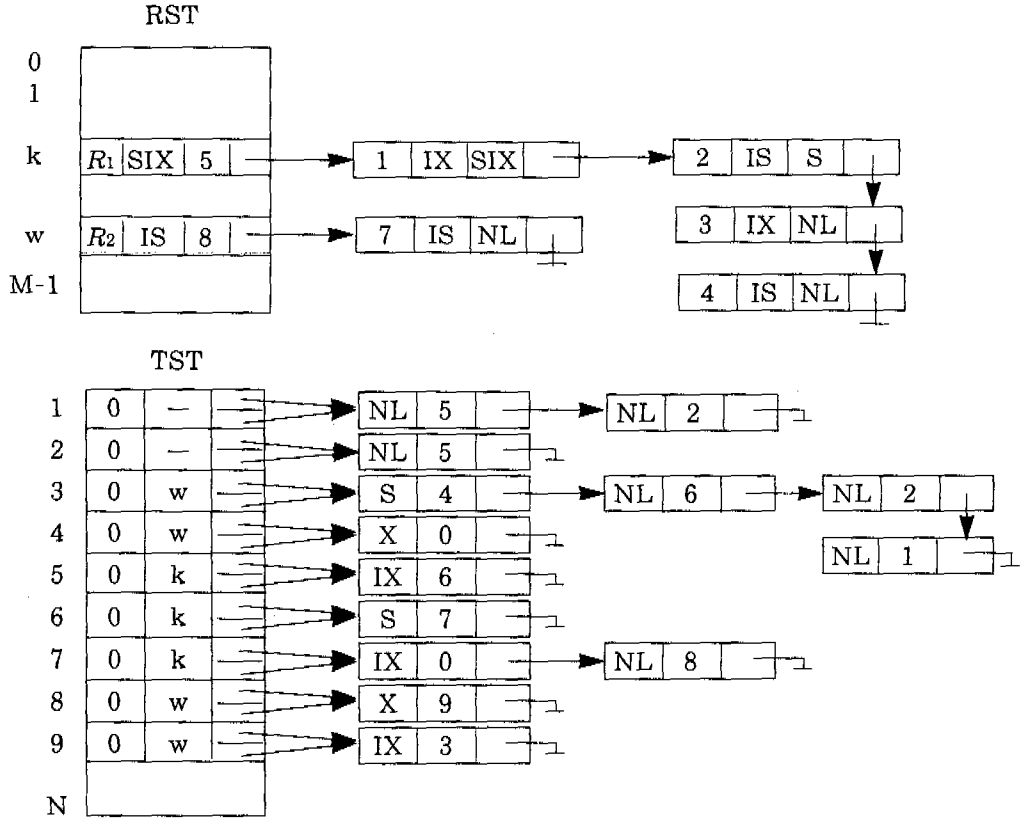

Figure 5.1 RST and TST for Example 4.1

The overall process of our periodic deadlock detection and resolution scheme can be summarized as the following three steps which are repeated periodically. These three steps will be refined shortly.

Algorithm periodic-detection-resolution;

Step 1: Construct H-labeled edges and initialize some variables.

Step 2: Find cycles and select victims to resolve them.

Step 3: Confirm transactions to be aborted and granted.

In Step 1 only H-labeled edges are

constructed into TST by applying ECR-1 and 2 for each resource in RST because all W-labeled edges are always present in TST. The algorithm employs two variables ancestor and and current for each transaction node in TST. The ancestor which is initialized to O is used to mark trarersed vertices in the current subgraph as well as is used for the backtracking of the depth-first-search. The current of vertex v which is initialized to waited indicates the next edge to be searched from v. In addition to that some global variables (abortion-list, change-list and grant-list) which will be explained in the next steps are initialized to be empty. As a matter of fact, Figure 5.1 shows the result of Step 1 for the situation given in Example 4.1. The whole process of Step 1 is refined below.

```
Step 1:/* Initialization*/
  for v:= 0 to M-1 do
      Construct H-labeled edges into
      TST by applying ECR-1 and 2
      for RST[v].
  for v:= 1 to N do begin
      TST[v].ancestor:=0;
      TST[v].current:=TST[v].waited;
      end;
abortion-list:=φ; change-list:=φ;
grant-list:=φ;
```

In Step 2 a directed walk in TST starting from vertex v always either terminates at v or takes us back to a vertex marked a non-zero ancestor. The first condition corresponds to the case in which there is no more cycle reachable from v in the following senses: (1) There is no cycle in the beginning in that subgraph, (2) There was a cycle which had beem already resolved by previous another directed walk, or (3) The walk has already resolved a cycle, if exists, so at this point there is no more cycle reachable from the root node. The second condition is satisfied when there is a cycle in it. During the directed walk, we backtrack to the ancestor and then proceed to the next incident vertex when we cannot move forward anymore. That condition is expressed in our internal structure in such a way the current value of a vertex is nil. That internal condition takes place when all reachable cycles, if any, have beem resolved by selecting some vertices or the corresponding vertex as victim(s). The whole process of Step 2 is refined below.

```
Step 2:/* Cycle detection and victim
selection*/
   for v:= 1 to N do begin
    TST[v].ancestor:=-1;
    while v≠ -1 do begin
   if TST[v].current = nil then begin
     w:=TST[v].ancestor;
     TST[v].ancestor:=0; v:=W;
   end else begin
     Let the edge pointed to by TST
     [v].current be(Lock, w, link).
    if(w=0) or (TST[w].current=nil)
    then TST[v].current:=link
    else if TST[w].ancestor≠0 then
    begin
       victim-selection(v); v:=w;
    end else begin
```

```
      TST[w].ancestor:=v; v:=w;
    end
   end
   end-while
end-for;
```

Based on TDR and our victim selection strategy, the process of victim-selection can be explained as follows. More detailed one which is written in an algorithmic language can be found in [16]. Let an edge $v \rightarrow w$ be the one which detects a cycle in our directed walk. Starting from v, we backtrack until w is visited again while applying TDR and our victim selection strategy to find a victim in the cycle. During backtracking, the ancestor of each backtracked vertex is cleared except for w. On finishing backtracking, if the victim is decided by TDR-1, then the current of the victim is set to nil and the victim is added to the abortion-list which keeps all the victims to be aborted. When it is selected by TDR-2 however, the followings take place: First, the requests in ST are repositioned right after those of AV. Second, the queue of the resource in RST is set to the first request in AV. Third, in order to prevent the requests in ST from the repeated application of TDR-2, the cost of each ransaction in ST is incremented by "some" value which might be determined according to the current cost of the transaction and the period of the deadlock detection. Fourth, the resource identifier which they are blocked is added to the change-list which is taken care of at Step 3, Finally, becaue they cannot be involved in any deadlock cycle, the current of each transaction in AV is set to nil.

During our directed walk, once a cycle is detected with an edge $(v \rightarrow w)$ and resolved appropriately with backtracking, the walk resumes at the vertex w. It should be noted that we may traverse more than once a subset of transactions in the cycle which has been resolved just before to check undetected cycles reachable from the root.

In Step 3, with the abortion-list and the change-list obtained from Step 2, we confirm transactions to be aborted and granted in addition to modifying RST and TST. The details for Step 3 is given below.

```
Step 3:/*Confirm transactions to be aborted and granted*/
    for v:= 1 to N do
        delete all the H-labeled edges
        in TST[v].
    for each transaction v in abortion-
        list do
      if v is in grant-list
      then delete v from abortion-list
       else for each resource Rv to
       which v is related do
       delete v from Rv and construct
       grant-list according to our
       scheduling policy.
   for each resource in change-list do
     construct grant-list according to
     our scheduling policy.
```

As results of Step 3 (in fact, the result of periodic-detection-resolution

algorithm), we get the abortion-list which contains transactions to be aborted and the grant-list whose constituting transactions are to be granted. One comment might be in order for Step 3 in which a transaction Ti in the abortion-list is not aborted if it exists in the grant-list. The reason for this can be clarified through the following example.

Example 5.1 Suppose that two resources R1 and R2 are requested as follows:

R_1[S]:Holder((T_1, S, NL)) Queue(T_2 (X), T_2(S))

R_2[S]:Holder((T_2, S, NL), (T_3, S, NL)) Queue(T_1(X))

Then the above situation has two cycles $\{T_1, T_2, T_3\}$ and $\{T_1, T_2\}$ as shown in Figure 5.2.

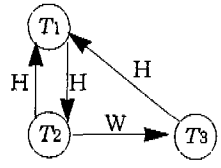

Figure 5.2 An example of a deadlock

Assume that the cycle search started from T_1 at Step 2 and cost for each transaction T_1, T_2, and T_3 is 6, 4, and 1 respectively. Because W-labeled edge, if any, precedes H-labeled edges in an entry of TST, the cycle $\{T_1, T_2, T_3\}$ is detected first and according to our deadlock resolution rule, T_3 is selected as a victim to be aborted and added to the abortion-list. When the cycle $\{T_1, T_2\}$ is detected next, T_2 is selected as a victim to be aborted and added to the abortion-list. At Step 3 of our periodic-detection-resolution algorithm with the abortion-list $\{T_2, T_3\}$, suppose that T_2 is examined first. After deleting T_2's requests from R1 and R_2, according to our scheduling policy, T_3 has to be granted at R_1 such that it is put in the grant-list. When T_3 is checked from the abortion-list, it has to be deleted from the abortion-list because T_3 is not involved in deadlock any more. As results of Step 3 in this example, the abortion-list is $\{T_2\}$, the grant-list is $\{T_3\}$ and the situation is modified as follows.

R_1[S]:Holder((T_3, S, NL), (T_1, S, NL)) Queue( )

R_2[S]:Holder((T_3, S, NL)) Queue(T_1 (X))

Before we close this section, the space and the time complexities of our algorithm are discussed. Space complexity is $O(n+e)$, where n is the number of transactions in TST and e is the number of edegs in it. When there is not any cycle in the graph, $O(n+e)$ time is required to search all the vertices and all the edges in TST. Different from listing all the elementary cycles in a directed graph as in Johnson's algorithm[15], when a cycle is found at Step 2, it is searched again to find victim; transaction(s) in the victim are marked not to search the same cycle again; and the search resumes at the vertex where the cycle is found. By doing this way, the total number of cycles searched(c') cannot exceed the number of elementary cycles (c) and also cannot be greater than the

number of transactions (n). Because procedure victim-selection can be done in $O(n)$ time, the total time complexity is $O(n+e^*(c'+1))$.

6. Closing Remarks and Disscussion

In this work, we developed a new deadlock detection graph, called H/W-TWBG, in the environment of sequential transaction processing with supporting multiple lock modes and permitting lock conversions. We also established the identification principles of the victim candidates: one by aborting a transaction in a cycle and the other by switching the order of lock requests in the queue of a resource without having the risk of the live-lock.

Based on the graph and the victim selection strategies, we have developed an efficient deadlock detection and resolution algorithm which has $O(n+e)$ storage space and $O(n+e^*(c'+1))$ time complexity, where e is the number of waited-by edges, n is the number of transactions in the database system and c' is the number of cycles which are actually searched such that c' is not greater than c, the number of elementary cycles in the graph, and also not greater than n.

References

1. R. Agrawal, M.J. Carey and D.J. DeWitt, "Deadlock Detection is Cheap," *ACM SIGMOD RECORD*, Vol.13, No.2, pp.19-34, January 1983.
2. R. Agrawal, M.J. Carey and L.W. McVoy, The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Trans. on Software Eng.*, Vol.SE-13, No.12, pp.1348-1363, December 1987.
3. R. Agrawal, M.J. Carey and M. Linvy, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM TODS*, Vol.12, No.4, pp.609-654, December 1987.
4. C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm," in *Proc. of 7th Intl. Conf. on VLDB*, pp.166-178, 1981.
5. P.A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.
6. W.N. Chin, "Some comments on ' Deadlock Detection is Cheap' in SIGMOD Record Jan. 83," *ACM SIGMOD RECORD*, pp.61-63, March 1984.
7. C.J.Date, "An Introduction to Database Systems: Vol II," *Addisin-Wesley*, 1983.
8. A.K. Elmagarmid, Deadlock Detection and Resolution in Distributed Processing Systems, Ph.D Dissertation, The Ohio State University, 1985.
9. S. Even, "Graph Algorithms," *Computer Science Precess*, 1979.
10. J.F. Garza and W. Kim,

"Transaction Management in an Object-Oriented Database System", in Proc. of ACM SIGMOD Intl. Conf., pp. 37-45, 1988.

11. J. Gray, "Notes on Database Operating Systems," in Lecture Notes in Computer Science 60, Advanced Course on Operating Systems, ed. G. Seegmuller, Springer Verlag, New York, pp. 393-481, 1978.
12. J. Gray and A. Reuter, "Transaction Processing," Version I of the Slides, 1987.
13. T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, pp. 287-317, 1983.
14. B. Jiang, "Deadlock Detection is Really Cheap," *ACM SIGMOD RECORD*, Vol. 17, No. 2, pp. 2-13, June 1988.
15. D. B. Johnson, "Finding all the Elementary Circuits of a Directed Graph," *SIAM J. Computing*, Vol. 4, No. 1, pp. 77-84, March 1975.
16. Y. C. Park, Deadlock Detection and Resolution in Database Management Systems: A Comprehensive Approach, Ph. D Dissertation, Northwestern University, 1989.
17. Y. C. Park and P. Scheuermann, "A Deadlock Detection and Resolution Algorithm For Sequential Transaction Processing with Multiple Lock Modes," to appear in *COMPSAC* 1991.
18. K. H. Pun and G. G. Belford, "Performance Study of Two Phase Locking in Single-Site Database Systems," *IEEE Trans. Software Eng.*, Vol. SE-14, No. 12, pp. 1313-1328, December 1987.

Appendix

Recall the major features in our model described in Section 2. One important feature of our model is that a transaction can wait for at most one resource to be released. The consequesnt observation of the previous statement is formally stated in Axiom 1.

Axiom 1 No transaction appears more than once in the queue of the whole system.

A transaction, however can appear in a queue of a resource R_1 and at the same time appear in the holder list of a resource R_2 because being in the holder list does not necessarily implies that the transaction is blocked.

Lemma 1 No cycle can be made without any H-labeled edge.

proof: Suppose that there is a cycle with W-labeled edges only. Because the only way to draw a W-labeled edge in H/W-TWBG is to use the ECR-3 which is applicable for transactions in the queue of a resource, there must be a transaction T such that either T appears more than once at different positions of the queue of a resource or T appears in the queue of a resource R_1 and at the same time appears in the queue of another resource R_2. Both cases contradict Axiom 1.

QED

Lemma 2 No cycle can be made

with only one TRRP.

proof: Suppose that there is a cycle only with one TRRP, say $TRRP_1$, in H/W-TWBG of a resource R1. The definition of TRRP says that $TRRP_1$ consists of one H-labeled edge and following multiple (possibly empty) W-labeled edges. By Lemma 1, $TRRP_1$ should have at least one W-labeled edge to form a cycle. Let the tail transaction of H-labeled edge be T. In order for $TRRP_1$ to be a cycle, one of the following cases is true. 1) T appears in the queue of R1 again or 2) There is a transaction T' such that T' appears in the queue of resource R_1 and at the same time T appears in the queue of another resource. Both cases contradict Axiom 1. QED

Lemma 3 Each cycle in the H/W-TWBG consists of at least two TRRPs.

proof: First of all, Lemma 2 excludes a cycle with one TRRP in the H/W-TWBG. Suppose followings. 1) There is a path $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ with k TRRPs ($K \geq 1$) in H/W-TWBG, 2) T_n holds a resource Rn, and 3) T_1 then requests a lock for the resource Rn which turns out to be blocked due to lock conflicts. Then there will be a H-labeled edge from T_n to T_1, which is as a matter of facts a TRRP in our definition. The resulting sequence forms a cycle which consists of k+1 TRRPs. Therefore a cycle in H/W-TWBG can be made with n TRRPs where $n \geq 2$.

QED

Definition 1 (Slight modifications of the definition [4]) Let T be all transactions in a system. The system is in a deadlock if there is a subset T' of transactions such that all transactions of T' have outstanding requests and even all transaction T-T' were removed and their resources were released, no request of a transaction of T' could be completely satisfied (none of them could start to run). We call such a set T' a *deadlock set.*

Definition 2 A deadlock set T is minimal if and only if no proper subset of T is a deadlock set any more.

The rationale behind the definition of a minimal deadlock set is that we are mainly interested in a single cycle even though Beeri and Obermarck [4] said that deadlock sets are closed under union. The case in which a number of cycles exist is easily extended once we have established a firm theory on a single cycle case. Next is another definition of a deadlock state which will be used afterwards.

Definition 3 The system is in a deadlock if and only if its minimal deadlock set is non-empty.

Lemma 4 Let MDS be a minimal deadlock set. Then the incomming edge and the outgoing edge of each transaction of MDS in H/W-TWBG are unique.

proof: Let us consider the incomming edge first. Assume that a transaction T_1 is blocked for a resource R_1. As far as R_1 goes, T_1 has a unique incomming edge. Suppose otherwise.

Case 1:There is no incomming edge to T_1 at R_1.

If T_1 is a member of holders of R_1, then there is at least one transaction $T2(T_1 \neq T_2)$ whose granted mode is incompatible with the blocked mode of T_1. If T_1 is the first transaction in the queue of R_1, then there is at least one transaction T_3 whose granted mode or blocked mode is incompatible with the blocked mode of T_1. If T_1 is in the queue and not the first transaction in the queue, then let T_4 be a transaction preceding T_1 in the same queue. In all cases, our ECR rule draws an edge to T_1 from T_2, T_3 or T_4. A contradiction arises.

Case 2:There is more than one incomming edge to T_1.

If there is more than one incomming edge to T_1, then deletion of all incomming edges except one does not allow T_1 to proceed since T_1 continues to be blocked. This contradicts the definition of MDS.

Therefore there is only one incomming edge for each transaction in MDS. As for the outgoing edge, if T_1 in MDS has more than one outgoing edge, sat $k(k \geq 2)$, then there must be k distinct transactions which is blocked by T_1 because there is only one incomming edge for each transaction. Assume all (k-1) transactions are aborted. A deadlock is still there, which contradicts the definition of MDS. The proof is completed. QED

Now we present the main theorem which states the deadlock condition in H/W-TWBG.

Theorem 1 There exists a cycle in H/W-TWBG if and only if there exists a deadlock in the database system.

proof: (<---) Suppose that there is a deadlock in the system. Then there is a minimal deadlock set MDS $\{T_1, T_2 \ldots, T_n\}$. Assume that all transactions except ones in MDS are sucessfully committed to release all their locks. By Lemma 4, the incomm-ing edge and the outgoing edge of T_i are unique. The only way to construct edges among those transactions while satisfying the incomming edge and the outgoing edge uniqueness for each $T_i(1 \leq i \leq n)$ in MDS is to make a cycle. Whenever there is a non-empty MDS in the system (in other words, there is a deadlock), there is a cycle in H/W-TWBG.

(--->) Suppose that there is a cycle in H/W-TWBG. Without loss of generality, assume that the cycle is an elementary cycle. Lemma 3 says the cycle consists of at least two TRRPs. Note that each transaction in a TRRP except the tail transaction of the H-labeled edge (which is the first edge in the TRRP) is blocked by all the preceding transactions in the TRRP. Because all TRRPs constitutes a cycle, no transaction involved in the cycle can proceed without outer intervention (in other words, transactions in the cycle form a minimal deadlock set). Such condition is a deadlock. QED