2010 International Conference on Data Storage and Data Engineering

# Optimization of Detected Deadlock Views of Distributed Database

B.M. Monjurul Alom
School of Electrical engineering and Computer Science
University of Newcastle, NSW 2308
Australia
Monjurul.Alom@newcastle.edu.au

Frans Henskens, Michael Hannaford
School of Electrical engineering and Computer Science
University of Newcastle, NSW 2308
Australia
Frans.Henskens, Michael.Hannaford@newcastle.edu.au

*Abstract*—**Deadlock is one of the most serious problems in multitasking concurrent programming systems. The deadlock problem becomes further complicated when the underlying system is distributed and when tasks have timing constraints. Deadlock detection and optimization is very difficult in a distributed database system because no controller has complete and current information about the system and data dependencies. The deadlock problem is intrinsic to a distributed database system which employs locking as its concurrency control algorithm. In this paper, an optimization technique for the detected deadlock is presented which minimizes the abortion of the selected victim transactions. The optimization technique is concerned with the detection of the transactions which are the basis for the most of the deadlock cycles (either local or global) in the system. The presented technique aborts the transaction's requests which are really to blame for the formation of many deadlock cycles. Also the presented deadlock detection algorithm does not detect any false deadlock or exclude any really existing deadlocks. In this technique global deadlock is not dependent on the local deadlock system.**

***Keywords-TWFG; Priority_Id; Transaction Manager (TM) ; Transaction Queue; Most Deadlock Creator (MDC).***

## I. INTRODUCTION

Concurrency control and deadlock detection is the most important problem that must have a powerful attention when sharing information in distributed systems. The maturation of database management system (DBMS) technology has synchronized with significant developments in computer network and distributed computing technologies. A distributed database (DDB) is a collection of multiple logically interrelated databases distributed over a computer network [1]. A distributed database management system (DDBMS) is the software that permits the management of the DDB and makes the distribution transparent to the users.

In modern computer systems, several transactions may compete for a finite number of resources [2]. Upon requesting a resource, a transaction enters a wait state if the request is not granted due to non-availability of the resource. A situation may arise wherein waiting transactions may not ever get a chance to change their states. This can occur if the requested resources are held by other similarly waiting transaction. This situation is called deadlock.

During the last decade computing systems have undergone substantial development, which has greatly impacted on distributed database systems. While commercial systems are gradually maturing, new challenges are imposed by the world-wide interconnection of computer systems [3]. This creates an ever growing need for large scale enterprise-wide distributed solutions. In the future, distributed database systems will have to support hundreds or even thousands of sites and millions of clients and, therefore, will face tremendous scalability challenges with regard to performance, availability and administration [3]. Deadlocks can arise in each database system that permits concurrent execution of transactions using locking protocols, which is the case in most of today's (distributed) database systems.

In a distributed database system, data access by concurrent transactions are synchronized in order to preserve database consistency [4]. This synchronization can be achieved using concurrency control algorithms such as two phase locking (2PL), timestamp ordering [5], optimistic concurrency control [6] or a variation of these basic algorithms. In practice, the most commonly used concurrency control algorithm is 2PL. However, if locking is used, a group of transactions may become involved in a deadlock [4]. Consequently, some form of deadlock resolution must accompany 2PL.

In a distributed database system, although a transaction may perform all of its actions at the site in which it originates, it may also perform actions (or actions may be performed on behalf of it) at other than the original site. If this happens, an agent [7] is created at the remote site to represent the transaction at that site. This agent becomes part of the original transaction for concurrency control and recovery purposes. Many algorithms have been proposed to detect deadlocks in distributed database systems [3, 4, 8-18] . Some methods are based on transmitting probes between sites. Probes are special messages used to detect deadlocks. Probes (these messages) follow the edges of the wait-for graph without constructing a separate representation of the graph [9, 10, 17]. The advantage of this approach is that probe algorithms are more efficient than wait-for-graphs. The disadvantage of the probe approach is that after deadlock is detected, the constituents of the cycle remain to be discovered.

The distributed deadlock detection algorithms that have been proposed are divided into two categories. Algorithms that belong to the first category pass information about transaction requests to maintain a global wait-for-graph. In the algorithms in the second category, simpler messages are sent among transactions; no global wait-for-graph is explicitly constructed. However, a cycle in the graph will ultimately cause messages to return to the initiator of the deadlock detection message, signaling the existence of deadlock.

The authors presented the deadlock detection technique in [19] that is based on creating Linear Transaction Structure (LTS) to find local cycles for each site of distributed database systems (DDBS). To find the global deadlock cycle Distributed Transaction Structure (DTS) is used for DDBS. In each site, each transaction has a unique priority id assigned by the transaction manager (TM); priority id is used to find the youngest transaction which caused a deadlock cycle. Transaction Queues (TQ) are used to store the transaction's priority id which forms cycles in LTS and DTS. The proposed technique is efficient as it does not detect any false deadlock. But the problem is that the technique ( in [19] ) always aborts the youngest victim transaction from detected local and global cycles. So, always aborting youngest victim transactions, some (essential) requests from transactions can be excluded from the system. Therefore the transactions (are really supposed to be expelled from the system) which are responsible for the creation of many deadlock cycles may still exist in the system.

In this paper, we describe the deadlock optimization technique which is concerned with the detection of the transactions which are the source for the most of the deadlock cycles (either local or global) in the system. The presented technique aborts the transaction's requests which are really responsible for the formation of many deadlock cycles. The optimization technique based on some definitions which execute mathematical functions on the detected local and global deadlock cycles. The technique can be applied to all different types of complex transaction wait for graph (TWFG).

The remainder of this paper is organized as follows: The Framework of the Distributed Deadlock Optimization is presented in section II. Explanation of the Deadlock Optimization is described in section III. The paper concludes with a discussion and final remarks in section IV.

## II. FRAMEWORK OF DEADLOCK OPTIMIZATION

The authors presented the deadlock detection technique in [19] that is based on the following Rules (1-6):

**Rule-1:** Each Local deadlock cycle $LD_i$ is detected from the values of $LTS_i$ . A global deadlock cycle GDC is calculated from a set of $\{ DTS_i , DTS_{i+1}, DTS_{i+2} , \ldots\ldots\ldots\ldots DTS_n \}$ . GDC is not dependent on the set of $\{LD_i , LD_{i+1} , LD_{i+2} \ldots\ldots\ldots LD_n\}$.

**Rule-2:** $\{( \forall q \in LTS_i\ (p,q) \mid \exists\ LD_i$ ; iff $(LTS_i[q_k]= LTS_i[p_j] \vee LTS_i[p_{j+1}] \vee LTS_i[p_{j+2}] \vee \ldots\ldots \vee LTS_i[p_{k-1}] ) \wedge ( LTS_i(q_j)= LTS_i[p_{j+1}] \wedge LTS_i(q_{j+1})= LTS_i[p_{j+2}], \wedge LTS_i(q_{j+2})= LTS_i[p_{j+3}] \wedge \ldots\ldots \wedge LTS_i(q_{k-1})= LTS_i[p_k]) \bullet LD_i =\{ LTS_i[p_j],LTS_i[q_j], LTS_i[q_{j+1}], LTS_i[q_{j+2}] \ldots\ldots LTS_i[q_{k-1}]( LTS_i[p_k] ), LTS_i[q_k] \}; 1<= i<=n; j<=k<=N; j>0 \}$.

**Rule-3:** $\{\exists\ yvictim(T_{idv1}): TQ[PT_{id1}, \ldots\ldots\ldots PT_{id(n-1)}, PT_{id(n)}] \mid$ iff $(PT_{id1}>PT_{id2} > \ldots\ldots\ldots\ldots >PT_{idn} ) \bullet yvictim\ (T_{idv1}) =PT_{id(n)}; PT_{id1} \Rightarrow PT_{id}(LTS_i[p_j]), PT_{id2} \Rightarrow PT_{id}\ (LTS_i[q_j], PT_{id3} \Rightarrow PT_{id}\ (LTS_i[q_{j+1}]), \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$
$PT_{id(n-1)} \Rightarrow PT_{id}\ (LTS_i[q_{k-1}]) \vee PT_{id}\ (LTS_i[p_k]) , PT_{id(n)} \Rightarrow PT_{id}\ (LTS_i[q_k]) \}$.

**Rule-4:** Abort the request from yvictim (youngest victim) transaction; from each calculated $LD_i$ through all distributed sites of the database systems.

**Rule-5:** $\{ \forall q \in LTS_i\ (p,q) , \forall p \in LTS_{i+1}\ (p,q) \mid \exists\ q_k=p_k \bullet LTS_i \leftrightarrow LTS_{i+1} \} \vee \{ \forall q \in LTS_{i+1}(p,q) , \forall p \in LTS_i\ (p,q) \mid \exists\ q_k=p_k \bullet LTS_{i+1} \leftrightarrow LTS_i \} \vee \{ \forall q \in LTS_i(p,q) , \forall p \in LTS_n\ (p,q) \mid \exists\ q_k=p_k \bullet LTS_i \leftrightarrow LTS_n\}$.

**Rule-6:** if $LTS_i \leftrightarrow LTS_{i+1}$ ; at first, DM starts storing the transaction's (those are connected to other sites.) intra requests (connectivity) into DTS. Data Manager (DM) then records the request of transactions between $LTS_i$, $LTS_{i+1}$ into DTS. GDC is detected applying definition 2 to all of the calculated DTS from the whole distributed database system (DDBS), yvictimpair is aborted from each DTS applying definition 3, to free from (GDC) global deadlock cycle.

In the deadlock detection and optimization technique, a Linear Transaction Structure (LTS) is maintained for the transactions of each local site of the database systems and Distributed Transaction Structure's (DTS) are used to handle the global deadlock among the distributed sites. Each transaction is assigned a unique Priority Id from the local Transaction Manager (TM) of the systems. All the local TM's are controlled by Global TM (GTM). GTM also manages the TQ for DTS. The existence of the cycle from in local LTS represents a local deadlock and the existence of a cycle in the DTS represents a global deadlock. The proposed technique uses a graph (TWFG) to represent the transaction's data request; that is maintained by Global Scheduler (GS). GS also collects the request (information) of the transaction from each local scheduler (S).

This technique assures that global deadlock is not dependent on local deadlock. On the other words, local deadlock does not cause the global deadlock. The victim transactions are (youngest transactions) decided based on priority id (from Transaction Queue (TQ)) from the detected cycles. This technique stores the id (number) of each transaction to their corresponding LTS and DTS whilst finding the local and global deadlock cycles. According to this approach, no false deadlock is detected or does not exclude any really deadlocked cycle.

If any transaction $T_p$ requests a data item that is held by another transaction $T_q$ , this technique stores the values of p and q to the linear transaction structure (LTS), where p and q represents their corresponding transaction number. Each row of the LTS stores a pair of values (p, q). Each site must have its own LTS. It is assumed that the total no of distributed sites are n, total number of transaction pairs in each LTS are N.

Distributed Transaction Structure (DTS) generally stores all the transactions which are interconnected (requests for data item from other sites) from one site to another site. DTS also records the transaction's (those are connected to other sites) intra requests (connectivity). DTS is managed by Data Manager (DM). Each transaction should have a unique priority transaction id named $PT_{id}$ from transaction manager (TM). Transaction queue (TQ) is used to store the $PT_{id}$ for the abortion of the youngest (victim) transaction from a deadlocked cycle.

To find local deadlock, DM starts storing transaction's requests for data item in LTS, from any transaction in any site. TM records priority transaction id in TQ for those transactions which form cycles in LTS; it is recommended that (any) starting transaction in the cycle has the highest priority Id (but the starting transaction in LTS could be any transaction). Whilst detecting global deadlock, at first Data Manager (DM) starts storing the intra connected transaction's (those are connected to other sites) request in DTS from any site. After then, DM records the transaction requests from site to site. This is to provide less priority id for the transaction's data request from one site to another site, as in general , global deadlock cycles become free from deadlock after aborting the transaction's data request from one site to another site. Similarly GTM records priority transaction id in TQ for those transactions which form cycles in DTS. Generally in TQ, the priority id in the first position has the leading priority and the priority id in the last position has lowest priority. The priority id which has lowest priority is the youngest transaction.

The following Rules (7-15) are also necessary to implement the optimization technique within the detected deadlocks:

**Rule-7**:

Let each distributed site has more than one local deadlock cycles ($LD_i$ , $LD_{i+1}$ … $LD_n$) and MLDC be the most local deadlock creator which is a set of at least one or more transaction pair; MLDC is defined as follows:

$MLDC_i = \{LD_i \cap LD_{i+1} \}$

$MLDC_{i+1} = \{LD_{i+2} \cap LD_{i+3} \}$

………………………………

$MLDC_n= \{LD_{n-1} \cap LD_n \}$; where $LD_i$ , $LD_{i+1}$ , $LD_{i+2}$ ……$LD_n$ are (a set of transactions which form a local cycle) detected from the values of $LTS_i$ , $LTS_{i+1}$ , $LTS_{i+2}$ …………………… $LTS_n$ respectively.

**Rule-8:**

If there exists more than one transaction pair in each of MLDC set, abort any one of the data request (from the transaction pairs) from each of the {$MLDC_i$, $MLDC_{i+1}$ ….. $MLDC_n$ }; otherwise abort the single existing data request from the transaction pairs.

**Rule-9:**

Let, $K_i=\{ LD_i \cup LD_{i+1} \}\backslash MLDC_i \}$

$K_{i+1} = LD_{i+2} \cup LD_{i+3} \}\backslash MLDC_{i+1}\}$

………………………………………………

$K_n=\{\{ LD_n \text{ U } LD_{n-1} \}\backslash MLDC_n \}$

If ($\exists$ a cycle in $K_i \vee K_{i+1} \vee$..........$\vee$ $K_n$), apply rule-7 and 8 for the minimization of the local victim transaction; otherwise deadlock free.

**Rule-10:**

Let two distributed sites have more than one global deadlock cycles (GDC) and MGDC be the most global deadlock creator which is a set of at least one or more transaction pair; MGDC is defined as follows:

$MGDC_i = \{GDC_i \cap GDC_{i+1} \}$

$MGDC_{i+1} = \{GDC_{i+2} \cap GDC_{i+3} \}$

………………………………

$MGDC_n= \{GDC_{n-1} \cap GDC_n \}$; where $GDC_i$ , $GDC_{i+1}$ , $GDC_{i+2}$ ……..$GDC_n$ are (a set of transactions which form a global cycle) detected from the values of $DTS_i$ , $DTS_{i+1}$ , $DTS_{i+2}$ …………………… $DTS_n$ respectively.

**Rule-11:**

If there exists more than one transaction pair in each of MGDC set, abort any one of the data request (from the transaction pairs) from each of the {$MGDC_i$, $MGDC_{i+1}$ ….. $MGDC_n$ }; otherwise abort the only existing data request from the transaction pairs.

**Rule-12:**

Let, $P_i=\{ GDC_i \cup GDC_{i+1} \}\backslash MGDC_i \}$

$P_{i+1} = \{GDC_{i+2} \cup GDC_{i+3} \}\backslash MGDC_{i+1}\}$

…………………………………………………

$P_n=\{\{ GDC_n \cup GDC_{n-1} \}\backslash MGDCC_n \}$

If ($\exists$ a cycle in $P_i \vee P_{i+1} \vee$..........$\vee$ $P_n$) , then apply definition 10 and 11 for the minimization of the global victim transaction.

**Rule-13:**

If there exist only one global deadlock cycle ($GDC_i$) between two sites (say $S_i$ and $S_k$ ) and only one local deadlock cycle ($LD_i$ or $LD_k$) in any of the sites, then most deadlock creator (MDC) is optimized as follows:

$MDC= GDC_i \cap LD_i \vee GDC_i \cap LD_k$ . Abort the MDC transaction pair from the two distributed sites of the database systems.

**Rule-14**

If there exist only one global deadlock cycle ($GDC_i$) between two sites ($S_i$ and $S_k$) and one local deadlock cycle in both of sites ($S_i$ and $S_k$ ), then most deadlock creator (MDC) is optimized as follows:

$MDCS_i= GDC_i \cap LD_i \wedge MDCS_k = GDC_i \cap LD_k$

Abort the transaction pair from $MDCS_i$ and $MDCS_k$ between the two distributed sites of the database systems.

**Rule 15:**

If any distributed site has no local deadlock but have the global deadlocks between two distributed sites, then apply rule 10 and 11.If there are no global deadlocks among the distributed sites but have more than one local deadlock in each of the site, then apply rule-7 and 8.

## III. Explanation of the optimization Technique

The functionality of the optimization of detected deadlock is demonstrated in the following example considering Figure 1. In Figure 1, site-1, site-2, and site-3 have 4, 3, and 1 local deadlock respectively. There are 2 global deadlocks between site-1 and site-2 and 1 global deadlock between site-2 and site-3. According to definition 7 and 8, the data (resource) request from the transaction pairs (which create most local deadlock cycle) { $T\text{-}6 \longrightarrow T\text{-}4$ },{ $T\text{-}5 \longrightarrow T\text{-}7$ },{ $T\text{-}14 \longrightarrow T\text{-}11$ },and { $T\text{-}14 \longrightarrow T\text{-}16$ } are aborted . According to definition 10 and 11, the data request of the transaction pair { $T\text{-}9 \longrightarrow T\text{-}11$ } is aborted to free from two global deadlock cycles. According to definition 13, the data (resource) request from the transaction pair { $T\text{-}20 \longrightarrow T\text{-}19$ } is aborted to minimize the deadlock. The optimized deadlock free graph is presented in Figure 2.

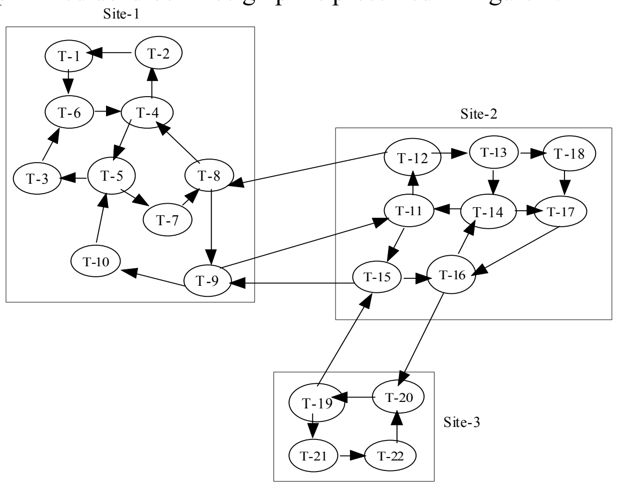

Figure 1. Complex TWFG consist local and global deadlock within three sites S1, S2 and S3.

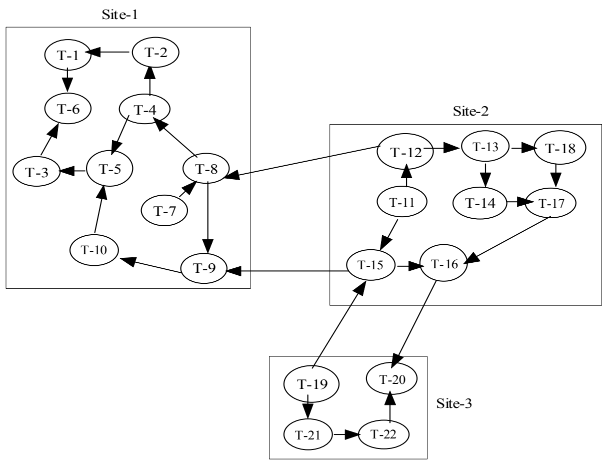

Figure 2. Optimized TWFG free from deadlock within three sites S1, S2 and S3.

## IV. PREVIOUS WORK

A lot of works regarding distributed deadlock detection are elaborated in [3, 8-10, 12, 14, 20-23]. To the best of our knowledge, very few techniques have been proposed for deadlock optimization within the detected deadlocks. Some of the algorithms based on deadlock detection are precisely described in the following:

### *A. Chandy & Mishra Algorithm [9]*

This algorithm uses transaction wait for graphs (TWFG) to represent the status of transactions at the local sites and uses probes to detect global deadlocks. The algorithm by which a transaction $T_i$ determines if it is deadlocked is called a probe computation. A probe is issued if a transaction begins to wait on another transaction and gets propagated from one site to another based on the status of the transaction that received the probe. The probes are meant only for deadlock detection and are distinct from requests and replies. A transaction sends at most one probe in any probe computation. If the initiator of the probe computation gets back the probe, then it is involved in a deadlock. This scheme does not suffer from false deadlock detection even if the transactions do not obey the two-phase locking protocol.

### *B. Sinha's Scheme[11]*

This algorithm, an extension to Chandy's scheme [9] is based on priorities of transactions. Using priorities, the number of messages required for deadlock detection is reduced considerably. An advantage of the scheme is that the number of messages in the best and worst cases can be easily determined.

### *C. Obermack's Algorithm[11]*

This algorithm at each site builds and analyzes directed TWFG and uses a distinguished node at each site. This node is called "external" and is used to represent the portion of TWFG that is external (unknown) to the site. This algorithm does not work correctly; it detects false deadlocks because the wait-for graphs constructed do not represent a snap-shot of the global TWFG at any instant.

### *D. Menasce's Scheme [10]*

This algorithm was the first to use a condensed transaction-wait-for graph (TWFG) in which the vertices represent transactions and edges indicate dependencies between transactions. This algorithm can fail to detect some deadlocks and may discover false deadlocks.

### *E. Ho's Algorithm [12]*

According to this [12] approach, the transaction table at each site maintains information regarding resources held and waited on by local transactions. The resources table at each of the sites maintains information regarding the transactions holding and waiting for local resources. Periodically, a site is chosen as a central controller responsible for performing deadlock detection. The drawback of this scheme is that it requires 4n messages, where n is the number of sites in the system.

### *F. Kawazu's Algorithm [13]*

The algorithm is based on two phases. In the first phase local deadlocks are detected, and in the second phase global deadlocks are detected in the absence of local deadlocks. This scheme suffers from phantom deadlocks, because each local wait-for graph is not collected at the same time due to communication delays. Also, in case a transaction simultaneously waits for more than one resource, some global deadlocks may go undetected since the global deadlock detection is initiated only if no local deadlock is detected. Also Bracha's [14], Mitchell's [15] and Krivokapic's [1] algorithms have been described to detect deadlock in distributed database.

## V. CONCLUSION AND FUTURE WORK

The deadlock problem involves a circular waiting where one or more transactions are waiting for resources to become an available and those resources are held by some other transactions that are in turn blocked until resources held by the first transaction or transactions are released. We presented an approach to detect both local deadlocks and global deadlocks in [19]. The technique uses TQ (Transaction queue) to store the priority id for all transactions which are in local deadlock cycles or in global deadlock cycles. Based on the priority id, the youngest transactions are aborted to free, the system from deadlock cycles. As a result, some crucial requests from the transactions can be expelled from the system because of the abortion of youngest victim pair. Therefore the transactions which are really responsible for the creation of major deadlock cycles may remain in the system. To handle this problem, this paper describes a technique to optimize the detected deadlock.

In a DBMS, deadlock resolution means that one or more of the participating transactions, the victim, is chosen to be aborted, thereby resolving the deadlock [3]. But when more than one deadlock cycle is involved in any distributed site or among the sites, it is required to optimize the request of the transactions which are involved for the cause of the major deadlock cycles. Handling deadlock involves two problems: deadlock detection and deadlock resolution. A deadlock detection algorithm or technique is correct if it satisfies two conditions: (1) every deadlock is eventually detected, and (2) every detected deadlock really exists, i.e., only genuine deadlocks are detected. Our proposed optimization technique also maintains these major features whilst minimizing the detected deadlock cycles.

## REFERENCES

[1] P. Valduriez and T. Ozsu, "Principle of Distributed Database Systems.," Prentice Hall, 1999.

[2] A. K. Elmagarmid, "A Survey Of Distributed Deadlock Detection Algorithms," *Sigmod* vol. 15: 3, pp. 37-45, 1986.

[3] N. Krivokapi, A. Kemper, and E. Gudes, "Deadlock Detection in Distributed Database Systems: A new algorithm and a comparitive performance analysis " *VLDB Journal* vol. 8, pp. 79-100, 1999.

[4] A. N. Choudhary, "Cost of Distributed Deadlock Detection: A performance Study," in *IEEE*, 1990.

[5] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM,* vol. 13:2, pp. 186-221, 1981.

[6] H. T. Kung and J. T. Robinson, "Optimistic Methods for Concurrency Control," *ACM Transaction on Database Systems,* vol. 6, pp. 213-226, 1981.

[7] J. N. Gray, "A discussion on distributed systems," IBM Research Division, 1979.

[8] G. Alkhatib and R. S. Labban, "Transaction Management in Distributed Database Systems: the Case of Oracle Two-Phase Commit," *The Journal of Information Systems Education,* vol. 13:2, pp. 95-103, 1995.

[9] K. M. Chandy, J. Misra, and L. M. Hass, "Distributed Deadlock Detection," *ACM Transaction on Computer Systems,* vol. 1:2, pp. 144-56, 1983.

[10] X. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems " in *ACM*, 1982.

[11] G. S. HO and C. V. RAMAMOORTHY, "Protocols for Deadlock Detection in Distributed Database Systems " *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,,* vol. 8:6, pp. 554-557, 1982.

[12] S. Kawazu, S. Minami, K. Itoh, and K. Teranaka, "Two-Phase Deadlock Detection Algorithm in Distributed Databases " in *IEEE*, 1979.

[13] D. P. Mitchell and M. J. Merritt, "A Distributed Algorithm for Deadlock Detection and Resolution," in *ACM*, 1984.

[14] J. Nummenmaa, "Distributed Deadlock Management," in *http://www.cs.uta.fi/~jyrki/ds01*, 2002.

[15] R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Transaction on Database Systems,* vol. 7:2, pp. 187-208, 1982.

[16] A. G. Olson and B. L. Evans, "Deadlock Detection For Distributed Process Networks," in *ICASSP*, 2005, pp. 73-76.

[17] M. K. Sinha and N. Natarjan, "A Priority Based Distributed Deadlock Detection Algorithm " *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 11:1, pp. 67-80, 1985.

[18] H. Wu, W.-N. Chin, and J. Jaffar, "An Efficient Distributed Deadlock Avoidance Algorithm for the AND Model," *IEEE Transactions on Software Engineering,* vol. 28:1, pp. 18-29, 2002.

[19] B. M. M. Alom, F. Henskens, and M. Hannaford, "Deadlock Detection Views of Distributed Database," in *International conference on Information Technology & New Generartion (ITNG-2009)* Las Vegas, USA: IEEE Computer Society, 2009.

[20] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases " *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 5:3, pp. 195-202, 1979.

[21] S. Bhalla and M. Hasegawa, "Automatic Detection of Multi-Level Deadlocks in Distributed Transaction Management Systems," in *International Conference on Parallel Processing Workshops (ICPPW)*: IEEE, Computer Society, 2003.

[22] G. Bracha and S. Toueg, "Distributed Algorithm for Generalized Deadlock Detection," in *ACM Symposium on Principles of Distributed Computing*, 1984.

[23] N. Farajzadeh, M. Hashemzadeh, M. Mousakhani, and A. T. Haghighat, "An Efficient Generalized Deadlock Detection and Resolution Algorithm in Distributed Systems," in *International Conference on Computer and Information Technology*, 2005.