CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

0-315-10855-X

58768

National Library of Canada | Bibliothèque nationale du Canada | CANADIAN THESES ON MICROFICHE | THÈSES CANADIENNES SUR MICROFICHE

NAME OF AUTHOR/*NOM DE L'AUTEUR* Abdel Aziz Farrag

TITLE OF THESIS/*TITRE DE LA THÈSE* An Improved Multi-version scheme for controlling Concurrent Accesses To A Database System.

UNIVERSITY/*UNIVERSITÉ* SIMON FRASER UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/ *GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE* M.SC

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE GRADE* 1982

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* Tiko Kameda

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

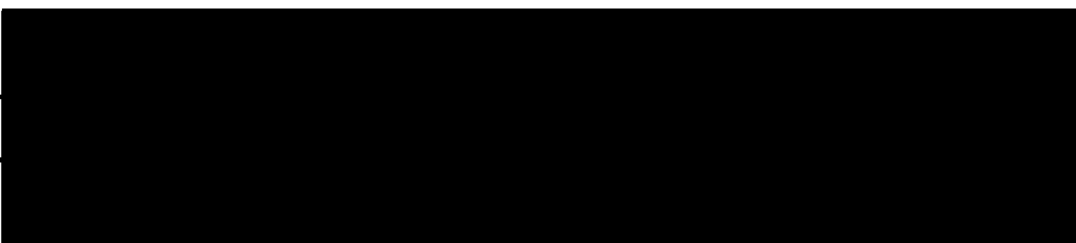
The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/*DATÉ* Sept. 30, 1982 SIGNED/*SIGNÉ*

PERMANENT ADDRESS/*RÉSIDENCE FIXÉ*



AN IMPROVED MULTI-VERSION SCHEME FOR CONTROLLING CONCURRENT ACCESSES TO A DATABASE SYSTEM

by

Abdel Aziz Farrag

B.Sc., Alexandria University, 1978

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

© Abdel Aziz Farrag 1982

SIMON FRASER UNIVERSITY

August 1982

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.

APPROVAL

Name : Abdel Aziz A. H. Farrag

Degree : Master of Science

Title of Thesis: An Improved Multi-Version Scheme For Controlling Concurrent Accesses To A Database System

Examining Committee:

Chairperson: Nick Cercone

T. Kameda
Senior Supervisor

W. S. Luk

L. Hafer

Toshimi Minoura
External Examiner
professor
University of Oregon

Date Approved: Aug 25, 1982

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

An Improved multi-version scheme for controlling concurrent accesses to a database system

Author:

(signature)

Abdel Farag

(name)

Sept. 30, 1982

(date)

An Improved Multi-Version Scheme For Controlling Concurrent Accesses To A Database System

Abstract

In many database system applications user transactions access the database concurrently. Since the operations of these transactions may interleave in any order there must be some kind of coordination to ensure database consistency. This thesis introduces a new scheme for controlling concurrent accesses to a database system. "Read" operations are always granted and each "read" operation returns an appropriate version without causing inconsistency. A "write" operation creates an appropriate version provided it does not cause inconsistency. An efficient algorithm for selecting the appropriate version is also introduced.

Unlike previously introduced schemes, a "write" operation may be allowed to create an "earlier" version. It is also shown how this new concept will enlarge the set of executions to be accepted by this scheme. A simple solution is proposed for the cyclic restart problem.

Acknowledgement

I would like to thank Professors W. S. Luk and L. Hafer for their encouragement and helpful advice.

Professor T. Kameda has greatly contributed to this research, both through his initial arousal of my interest in the problem of concurrency control in database systems and through his critical guidance in the development of the ideas in this thesis. His tireless patience and encouragement have made this task much easier. For all these things I express my most sincere thanks.

This thesis is dedicated to my mother, without whose continous support I would not have been able to study.

Table of contents

Approval page ii

Abstract iii

Acknowledgement iv

List of figures vii

1. Introduction 1

1.1 Concurrency control and consistency problem.........1

2. Database system model and execution 8

2.1 Database system model 8

2.2 Execution 10

2.2.1 Significance of the two functions F1 and F2..10

2.2.2 Late operations 13

2.2.3 Normalized execution 13

2.2.4 Serial execution 14

2.2.5 Two fictitious transactions.. 14

2.2.6 Augemented execution 15

3. Flow graph and active flow graph 16

3.1 Flow graph 16

3.1.1 Live operations 17

3.2 Multi-version serializability equivalence and the active flow graph 17

3.3 MVSR-executions 18

3.3.1 The fixpoint set 22

4. IMVSR-executions and the dependency graph 24

4.1 The dependency graph 24

4.1.1 The dependency graph for a serial execution..27

4.2 Selecting an appropriate version 33

5. IMVSR-scheduler 40

5.1 Processing the begin operation 40

5.2 Processing the read operation 41

5.3 Processing the write operation 42

5.4 Processing the commit operation 43

5.4.1 The deletion condition 44

6. Cyclic restart and long transactions 47

6.1 Detecting a cyclic restart 56

7. Conclusions 63

References 65

List of figures

Figure 1 Concurrent execution of T1 and T2 3

Figure 2 Flow graph for e 19

Figure 3 Flow graph for e1 21

Figure 4 Dependency graph for e 30

Figure 5 A cyclic restart 48

Figure 6 Ti is involved with Tj in a cyclic restart and with Tk in another cyclic restart 50

Figure 7 A cyclic restart involving three transactions Ti, Tj, and Tk 52

Figure 8 Transaction Ti and Transaction Tj are involved in a cyclic restart in which the two transactions will be aborted simultaneously at time t=t1,2t1,3t1,.. 54

Figure 9 transaction Ti and transaction Tj are aborted due to conflict with each other and they subsequently proceed to completion 56

Figure 10 transaction Ti is aborted because of transaction Tj and transaction Tk 57

1. Introduction

1.1 Concurrency control and consistency problem

A database is said to be in a consistent state if all the data objects satisfy a set of established assertions or integrity constraints. A database subject to multiple accesses requires that access to it be properly coordinated in order to preserve consistency.

In many database system applications it is desirable that the database system be shared by a set of user transactions. In such a system steps of transactions may occur in any interleaved order.

Even if each transaction is correct in the sense that it preserves the consistency of the database when executed by itself, the concurrent execution of correct transactions in an interleaved order may transform the database from a consistent state into an inconsistent state.

Example 1 (Lost update)

Suppose we have two transactions T1 and T2 accessing the same data object X, incrementing X by 1, and writing the new value of X. In the absence of concurrency control, these two transactions may interleave in an unacceptable order, so that the net effect is incorrect as shown in figure 1. (Note that R1(X) means a read request from transaction T1 for the data item X. Similary W1(X) means a write request from transaction T1 for the data item X).

Suppose that the initial value of X=5. Then the final value of X=6, i.e., the database reflects only one of the two updates.

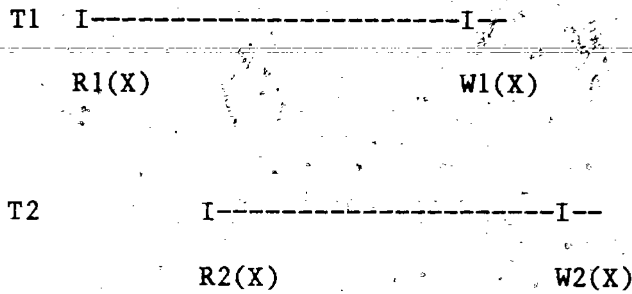


Figure 1. Concurrent execution of T1 and T2

Serializability still remains the most popular approach for ensuring database consistency. Databases are intended to be faithful models of some parts of the real world and the user transactions represent instantaneous changes in the world. Since the user transactions may interleave in any order, the only acceptable interleavings are those that are "equivalent" to some sequential execution of these transactions.

It is the task of the concurrency control scheme to safeguard the database consistency by properly granting or rejecting requests.

Concurrency control has been actively investigated for the past several years and several schemes have been proposed. These schemes can be broadly classified into two classes. The first class contains all the single version schemes while the second class contains all the multi-version schemes.

In this thesis we introduce a new multi-version scheme, or a new member in the second class.

The idea of using multiple versions of data items in a database system was first formalized by Stearns, et al. [STEA-76]. In their model a transaction Ti must read a data item in order to update it.

Bayer, et al. [BAYE-80] and Kessels [KESS-80] took notice of the fact that most database systems maintain two versions, the old and the new versions, of each object for recovery reasons while a transaction is modifying it. In the concurrency control scheme of [BAYE-80], a read operation reads either the old or the new version, depending on the state of the object with respect to updating. This clearly increases concurrency.

Reed [REED-79] has also proposed a multi-version concurrency control scheme for distributed database systems based on timestamping.

In section 2, we describe the database system model used in this thesis. It is similar to the model used by Muro, et al. [MURO-81, MURO-82], but we may allow a write operation to create an earlier version.

In section 3, we introduce two important graphs, the <u>flow graph</u> and the <u>active flow graph</u>. The latter is used in defining multi-version equivalence of two executions.

In section 4, we introduce a useful tool called the <u>dependency graph</u> [ESWA-76] used for recognizing the <u>IMVSR</u> - executions. We also show how the new concept of allowing a write operation to create an earlier version (provided it does not cause inconsistency) enlarges the fixpoint set (set of all executions accepted by the scheduler without aborting or delaying

any operation) of the IMVSR-scheduler without increasing overhead.

In section 5, we describe the new concurrency control algorithm (IMVSR-scheduler). In this section we show how the scheduler processes read and write operations.

In section 6, we explain the "cyclic restart problem" [STEA-76] and introduce a simple solution to this problem. Section 7 contains conclusions.

The main contributions of this thesis are.

1- Introducing a new multi-version scheme for controlling concurrent accesses to a database system.

2- Proving that read requests can always be granted.

3- Devising an efficient algorithm for selecting the appropriate version to be read or to be created.

4- Proving that the fixpoint set of our scheme contains the fixpoint sets of previously introduced multi-version schemes [STEA-76, REED-79, MURO-81, MURO-82, PAPA-82].

5- Proposing a simple solution for the "cyclic restart problem".

2. Database system model and execution

2.1 Database system model

Our database system model consists of a set E of data items, a set of transactions T={T1,T2,...,Tn}, and a scheduler.

We consider a transaction as a sequence of steps. Each step is either read(X) or write(X), where X is a data item in E. A read operation Ri(X) by a transaction Ti returns an appropriate version of X. A write operation Wi(X) by a transaction Ti creates a new version of X. Each data item is accessed by at most one read and one write operation of each transaction.

For each data item X, we maintain multiple versions of X, each of which corresponds to a write operation. Associated with each version of X is a version number which represents a proper order for this version among the other versions of X which already exist in the system. For each data item X, its initial value is defined to be X0 (version 0 of X). Assuming we had n versions of X {X0,X1,....,Xn-1}, when the scheduler received a read request for X, Ri(X), the scheduler would assign an appropriate version of X to the read request (not necessarily the latest version Xn-1).

If the scheduler receives a write operation , Wi(X), a new version will be created (this version will be assigned a number

not necessarily larger than other numbers assigned to the versions created before). When a transaction Ti creates a version of X and transaction Tj reads this version we say that transaction Tj reads X from Ti. When a transaction Tj reads a data item from a transaction Ti then transaction Tj is said to be strongly dependent on Ti. Other transactions which read a data item from Tj are also strongly dependent on Ti. The dependency relation among transactions will be studied in detail in section 4.

A transacion Ti is said to be a read only transaction for X if it reads X without updating it. Similarly, if Ti updates X without reading it, we say that Ti is a write only transaction for X. However, if Ti reads and updates X, we say that Ti is a read-write transaction for X.

2.2 Execution

An *execution* is a triple e=(O(T),F1,F2), where O(T) is the set of operations of the transactions in T, F1 is a mapping from O(T) to the set of positive integers {1,2,3,........}, and F2 is a mapping from O(T) to the set of nonnegative real numbers.

2.2.1 Significance of the two functions F1 and F2

The function F1 represents a permutation on the set of operations O(T). If F1(oi) is less than F1(oj) we say that oi *precedes* oj. If there is no operation between oi and oj, i.e., if F1(oj)=F1(oi)+1 we say that oi *immediately precedes* oj.

For each read operation, F2 determines the version to be returned by this operation. If F2(Ri(X))=1 for example, we say that Ri(X) *returns* version 1 of X, i.e., X1. If F2(Wj(X))=2, we say that Wj(X) *creates* version 2 of X. Different read operations which access the same data item may return the same version while different write operations always create different versions.

If F2(Ri(X))=F2(Wj(X)), then transaction Ti reads X from transaction Tj.

In our scheme, each read operation accessing a data item X returns an appropriate version of X created before, not necessarily the latest version of X. This means that

if $F2(Ri(X))=F2(Wj(X))$ then $F1(Wj(X))<F1(Ri(X))$

A write operation creates a new version of X. This version will be assigned a number not necessarily larger than other numbers assigned to the versions created before.

The way we assign a version number to represent a version created by a write request (or to be selected by a read request) depends mainly on the relation between the transactions as we explain later.

Example 2:

Let e=(O(T),F1,F2), where

1- O(T)={R1(X),W1(X),W2(Y),R1(Y),W2(X),R3(X)}

2- F1(R1(X))=1

F1(R3(X))=2

F1(W1(X))=3

F1(W2(Y))=4

F1(R1(Y))=5

F1(W2(X))=6

3- F2(R1(X))=0

F2(R3(X))=0

F2(W1(X))=1

F2(W2(Y))=1

F2(R1(Y))=0

F2(W2(X))=2

We shall also write e in a compact form as follows.

e=R1(X0)R3(X0)W1(X1)W2(Y1)R1(Y0)W2(X2)

The order of appearance of each operation represents the function F1 and the number associated with each data item represents the function F2. For example, the operation W1(X) is the third operation and creates version 1 of X.

2.2.2 Late operations

Suppose that the scheduler has received a read operation from a transaction Ti for the data item X, and suppose that the selected version of X is k, i.e., the read operation returns Xk. Then if k is the largest version number of X, we say that the read operation came in time and if k is not the largest version number of X we say that this read operation came late.

Similary, if the write operation Wi(X) may create a new version of X with a number larger than any version number created before, we say that the write operation came in time and if it must create a new version with a number less than some version number created before we say that the operation came late. For example, in the following execution the last two operations R1(Y) and W1(Y) came late.

e=R1(X0)W2(X1)W2(Y1)R1(Y0)W1(Y.5)

Here Y0.5 is a new version of Y with version number 0.5.

2.2.3 A normalized execution

An execution is said to be normalized [MURO-81] if the following two conditions are satisfied

a- If $F1(Wi(X))<F1(Wj(X))$ then $F2(Wi(X))<F2(wj(X))$.

b- If $F1(Wi(X))<F1(Rj(X))$ then $F2(Wi(X))\leq F2(Rj(X))$.

Condition a means that if Wi(X) precedes Wj(X) then Wj(X) creates a version of X with version number larger than that of the version created by Wi(X).

Condition b means that each read operation Ri(X) in a normalized execution returns the latest version created before this read operation.

It is not difficult to see that an execution in the conventional "single-version" database is actually a special case of an execution in the multi-version database (we refer to it as a normalized execution).

2.2.4 <u>Serial execution</u>

An execution is said to be <u>serial</u> if it is normalized and all the operations of each transaction appear consecutively (without interleaving with the operations of the other transactions). For example, the following execution es is serial.

es=W1(X1)R2(X1) R2(Y0) W2(X2) R3(X2)=T1T2T3.

2.2.5 <u>Two fictitious transactions</u>

a- <u>Initial transaction To</u>

This is a write only transaction, which writes the initial version of each data item.

b- <u>Final transaction Tf</u>

This is a read only transaction which reads the final version of each data item (i.e., the final result of the

execution of all transactions).

2.2.6 Augmented execution (ae)

An augmented execution ae consists of the execution e concatenated with the initial transaction T0=W0(X)W0(Y)... at the beginning and the final transaction Tf=Rf(X)Rf(Y)... at the end.

3. Flow graph and active flow graph

3.1 Flow graph

We construct for an execution e=(O(T),F1,F2), a directed graph called the flow graph. The set of nodes of this directed graph is the set of operations in O(T) together with the operations of the two fictitious transactions To and Tf. The arcs of this graph are given as follows [PAPA-82].

1- (Wj(X),Ri(X)) is an arc iff F2(Ri(X))=F2(Wj(X)), i.e., Ri(X) returns the version created by Wj(X).
2- (Ri(Y),Wi(X)) is an arc iff F1(Ri(Y))<F1(Wi(X)), i.e., Ri(Y) and Wi(X) belong to transaction Ti and Ri(Y) precedes Wi(X).

We can simply construct the flow graph FG(e) for an execution e as follows. Represent each operation o∈O(T′) by a node(where T′=TU{To,Tf}). Join each read node Ri(X) with a write node Wj(X) if Ri(X) returns the version created by Wi(X). Join each write node Wi(X) with all the read operations which belong to transaction Ti and precede the write operation.

Intuitively, an arc of type 1 represents a read from relation [PAPA-79, PAPA-82], i.e., if there is an arc from Wj(X) to Ri(X) in FG(e) then transacion Ti reads X from transaction Tj. An arc of type 2 indicates that the value to be created by a write operation Wi(X) presumably depends on the values returned by the previous read operations which precede Wi(X) in Ti.

3.1.1 Live operations [PAPA-82]

The live operations are defined as follows

a- all the (read) operations of the final transaction Tf are live operations

b- a write operation Wi(X) is live if there is an arc (Wi(X),Rj(X)) where Rj(X) is a live read.

c- a read operation Ri(Y) is live if there is an arc (Ri(Y),Wi(X)) where Wi(X) is a live write.

An operation is said to be dead if it is not live. Intuitively all the dead operations have no effect on the final values of the data items (the values read by the final transaction Tf).

3.2 Multi-version serializability equivalence and the active flow graph

We call the subgraph of the flow graph defined by the nodes corresponding to the live operations active flow graph AFG(e).

Two executions e1=(O(T),F1,F2) and e2=(O(T),F1′,F2′) are said to be multi-version equivalent (MV-equivalent, for short) if their active flow graphs are identical [PAPA-82]. Obviously this definition means that the two executions have the same set of live operations and each live read in e1 returns the same value as the corresponding read in e2.

3.3 MVSR-executions

An execution e is said to be <u>multi-version</u> <u>serializable</u> (MVSR for short) if it is MV-equivalent to a serial execution.

We define the set MVSR to be the set of all multi-version serializable executions and the set SR to be the set of all (single version) serializable executions [PAPA-82].

Example 4

We construct the flow graph for the following execution and determine the live operations and the dead operations.

e=W1(X1)R2(X1)W2(X2)R2(Y0)W2(Y1)W1(Y0.5)W3(Y2)

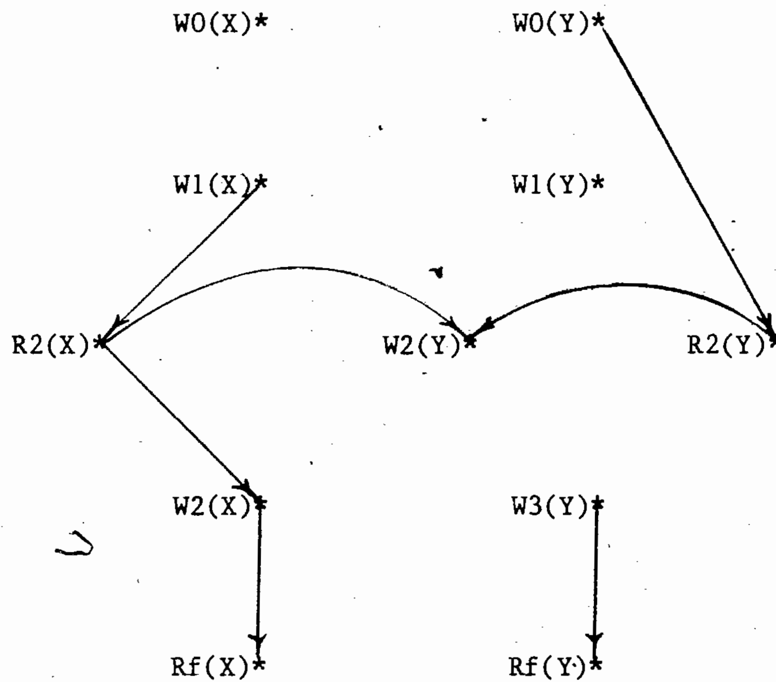


Figure 3. Flow graph for e

The live operations are
W1(X), R2(X), W2(X), W3(Y).

The dead operations are
R2(Y), W2(Y), W1(Y).
(Note that we did not mention the operations of the fictitious transactions).

It is clear that the execution e in this example is a multi-version serializable execution since its active flow graph is the same as the active flow graph of the serial execution es, where
es=T1T2T3=W1(X1)W1(Y1)R2(X1)W2(X2)R2(Y1)W2(Y2)W3(Y3).

<u>Example 4</u>

We construct the flow graph for the following execution and show that it is an MVSR-execution.

e1=R3(Y0)W3(Y1)R1(Y1)W2(X1)R1(X1)W3(X2)R5(X2)W5(Z1)W1(Y2)W4(X3).

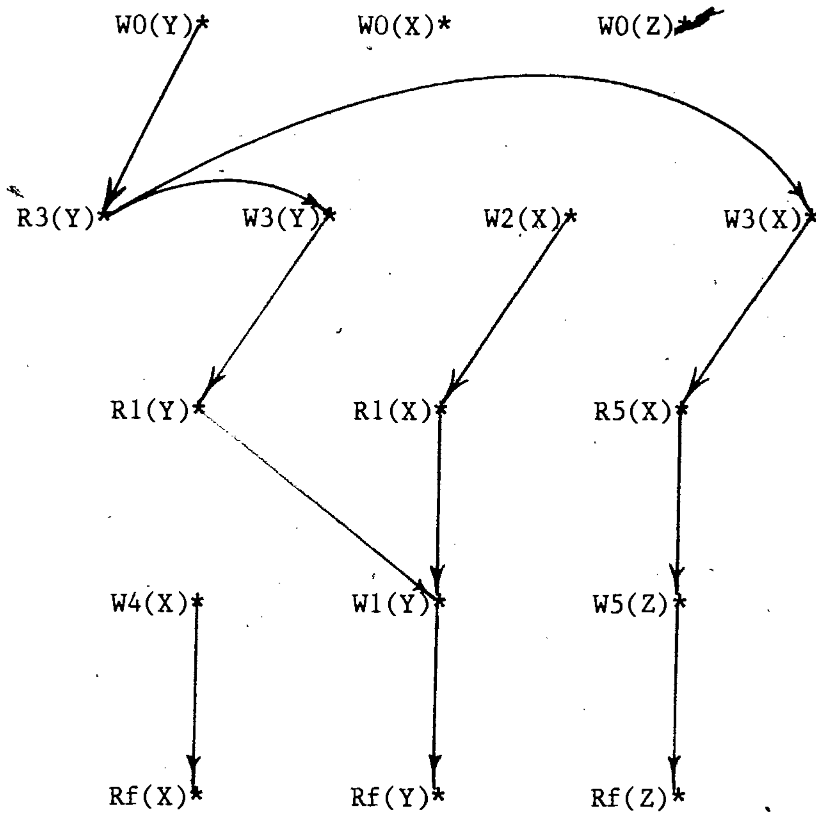


Figure 3. Flow graph for e1

It is clear that all the operations of e are live operations. It is also not difficult to see that the flow graph of the serial execution es=T3T5T2T1T4 is identical to the flow graph of e. This means that e is an MVSR-execution.

3.3.1 The fixpoint set

Associated with each scheduler S is a fixpoint set [KUNG-79] F(S). An execution e belongs to the fixpoint set F(S) of the scheduler S if it can be accepted without delaying or rejecting any operation of e.

A scheduler S may be viewed as a mapping or a transformation which receives as input any execution e, and produces as output an execution $e' \in F(S)$. However if the scheduler S is fed by a member of F(S) it will leave it intact.
Obviously the larger the fixpoint set the better is the scheduler in some sense. One would want to have a scheduler S with the fixpoint set F(S) as large as the set MVSR.

It has been proved that serializability test (and therefore multi-version serializability test) is NP-complete (see [PAPA-79, PAPA-82]), which implies that the implementation of the MVSR-scheduler (i.e., a scheduler which recognizes any MVSR-execution) is impractical. In fact, all the schedulers previously introduced (based on single version or multi-version) [STEA-76, REED-79, PAPA-79, BAYE-80, MURO-81, BERN-81, MURO-82, PAPA-82] which can be implemented in polynomial time recognize only a subset of the set MVSR for the multi-version case or the set SR for the single version case.

We are introducing an improved multi-version concurrency control scheme (IMVSR-scheduler) which can be implemented in polynomial time and its fixpoint set F(S) contains the fixpoint sets of previously introduced multi-version schemes [STEA-76, MURO-81, BERN-81, MURO-82].

In general, a concurrency control scheme (scheduler) is said to be <u>efficient</u> if it can be implemented in polynomial time.

4. IMVSR-executions and the Dependency Graph

In this section we define a new class (or a set) of executions called the IMVSR-executions and show that this class represents a subset of the set MVSR. We also introduce a useful tool called the dependency graph used in recognizing the IMVSR-executions.

4.1 The dependency graph

We say that a transaction Ti *conflicts* with a transaction Tj if both transactions access the same data item X and at least one of them creates a new version of X.

We construct for an execution e, a directed graph called the *dependency graph* DG(e) which represents the "dependence" relation among the transactions [ESWA-76]. The set of nodes of this graph is the set of transactions T={T1,T2,....,Tn} and the set of arcs are defined as follows.

An arc is directed from Ti to Tj ($i \neq j$) if any of the following conditions holds for some $X \in V$, where V is the set of data items accessed by the the set of operations in O(T).

a- There exist two operations Wi(X) and Rj(X) in O(T) such that $F2(Wi(X)) \leq F2(Rj(X))$

b- There exist two operations Wi(X) and Wj(X) in O(T) such that $F2(Wi(X)) < F2(Wj(X))$

c- There exist two operations Ri(X) and Wj(X) in O(T) such that $F2(Ri(X)) < F2(Wj(X))$

If there is an arc from Ti to Tj then we say that Tj is *dependent on* Ti. Other transactions reachable in DG(e) from Tj are also said to be dependent on Ti.

An arc from Ti to Tj which exists because $F2(Rj(X)) = F2(Wi(X))$ is called a *primary* arc [MURO-82]. It may also satisfy conditions b and/or c in addition to a. If there is a primary arc from Ti to Tj we say that Tj is *strongly dependent* on Ti.

We define strong dependence to be transitive. Intuitively, if Tj is strongly dependent on Ti, then some information is actually transfered from Ti to Tj.

In general, an execution e is said to be an *IMVSR-execution* if its dependency graph is *acyclic*.

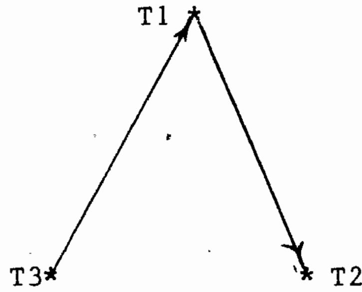
Note that an arc from Ti to Tj due to condition b in the scheme introduced by Muro, et al. [MURO-81, MURO-82] or by Papadimitriou, et al. [PAPA-82] implies that Wi(X) arrived before Wj(X). This is not the case in our scheme since we may allow a write operation to create an earlier version.

Example 6

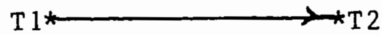
We construct the dependency graphs for the following executions.

e1=R1(X0)R3(X0)W1(X1)W2(Y1)W2(X2)R1(Y0)

e2=W1(X1)W2(X2)W2(Y1)W1(Y.5)



DG(e1)



DG(e2)

4.1.1 The dependency graph for a serial execution

In a serial execution all the operations of each transaction appear consecutively. Suppose that e is a serial execution and without loss of generality suppose further that e=T1T2...TiTi+1...Tn (i.e., the operations of T1 first and then the operations of T2 and so on). If transaction Ti and transaction Tj conflict and transaction Ti appears before transaction Tj (i.e., all the operations of Ti appear before all the operations of Tj) then an arc will be directed from Ti to Tj in the dependency graph. This means that all the arcs of DG(e) will be directed from left to right if the nodes are arranged from left to right in the order T1,T2,...,Tn. It follows that DG(e) cannot have a cycle if e is serial.

Lemma 1

Any IMVSR-execution is also an MVSR-execution.

Proof

Suppose that e is an IMVSR-execution. Then by definition DG(e) is acyclic, and therefore any topological sort of the set of nodes {T1,T2,..,Tn} yields a serial execution es. In order to prove this lemma we will show that FG(e)=FG(es), i.e., the arcs of type 1 and those of type 2 are the same for both flow graphs FG(e) and FG(es) (refer to section 3.1).

Suppose that there is an arc (Wi(X),Rj(X)) in FG(e). Then there must be an arc (Ti,Tj) in DG(e). To prove that there must be the corresponding arc (Wi(X),Rj(X)) in FG(es) we need only prove that there cannot be any other transaction, say Tk, where Wk(X)∈Tk between Ti and Tj (Ti..Tk..Tj) in the previous sort. In order to prove this claim, assume the existence of such a transaction. This means that there is a path in DG(e) from Ti to Tk and a path from Tk to Tj. If the version number of the version of X created by Tk is larger than that of the version created by Ti a path must exist from Tj to Tk (due to condition c on page 24), and therefore, a cycle exists since there is another path from Tk to Tj. If it is smaller, on the other hand, then a path must exist from Tk to Ti, and therefore, a cycle exists since there is another path from Ti to Tk. In either case a cycle exists in DG(e), which contradicts our assumption that DG(e) is acyclic. Then such a transaction Tk cannot exist and therefore Tj reads X from Ti in es, i.e., an arc (Wi(X),Rj(X)) is in FG(es). Since the two executions have the same set of read operations, the arcs of type 1 are the same for FG(e) and FG(es).

Moreover, since the order of operations of the same transaction does not change (i.e., if o1 and o2 are two operations belonging to transaction Ti and o1 precedes o2 in e then o1 also precedes o2 in es) arcs of type 2 are also the same for FG(e) and FG(es). This means that the two flow graphs are identical and therefore e is MV-equivalent to a serial execution

es. It follows that e is an MVSR-execution. (Q.E.D.)

Corollary 1

Checking whether an execution e=(O(T),F1,F2) is an IMVSR-execution can be done in O(|V|*n**2) time, where V is the set of data items accessed by the set of operations in O(T), and n is the number of transactions.

Proof

Given an execution e, we want to test if DG(e) is acyclic. In order to construct DG(e), we check if the condition a,b, or c is satisfied for each X∈V. For each X∈V, these conditions can be tested in O(n**2) time. Therefore altogether, DG(e) can be constructed in O(|V|*n**2) time. Test for acyclicity can be performed in time linear in the number of vertices and arcs in the dependency graph. (Q.E.D.)

In order to show that IMVSR is a proper subset of the set MVSR we shall give some execution which belongs to the set MVSR but does not belong to the set IMVSR.

Example 7

We construct the dependency graph for the following execution

e=R3(Y0)W3(Y1)R1(Y1)W2(X1)R1(X1)W3(X2)R5(X2)W5(Z1)W1(Y2)W4(X3).

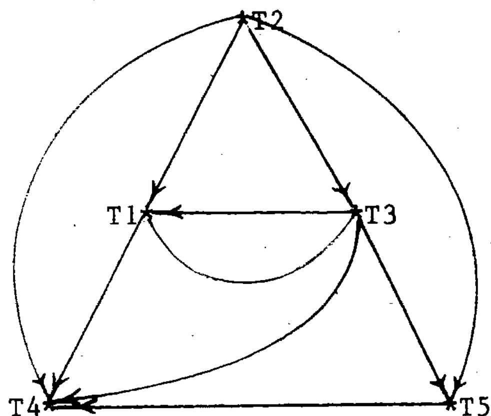


Figure 4. Dependency graph for e.

Since the dependency graph for e is cyclic, e is not an IMVSR-execution, i.e., $e \notin IMVSR$.

Note that we have already proved in section 2 (Example 3) that e is a multi-version serializable execution, i.e., $e \in MVSR$.

In the following two theorems (Theorem 1 and Theorem 2), we assume that we have a set of versions of the data item X, say X0,X1,..,Xk-1,Xk created by a set of transactions D={d0,d1,..,dk-1,dk} where di $(0 \le i \le k)$ denotes the transaction which created version Xi. We will refer to the set D as the <u>destination set</u>.

<u>Theorem 1 [MURO-81]</u>

Read operations can always be granted without creating a cycle in the dependency graph.

<u>Proof</u>

The following set of arcs, among others, exist before receiving the read request Ri(X)
(d1,d2),(d2,d3),...,(di,di+1),..,(dk-1,dk).

If there is no path from Ti to node dj or from dj to Ti for all j $(1 \le j \le k)$, then Ri(X) can be assigned any version of X without creating a cycle in the dependency graph.

Otherwise, let j be the least version number of X such that there is a path from Ti to dj and let j' be the largest version number of x such there is a path from dj' to Ti. We have j'<j, since otherwise there would be a cycle. Then Ri(X) can be assigned any version Xl $(j' \le l < j)$ without creating a cycle in the dependency graph. (Q.E.D.)

According to Theorem 1, if there is a path from Ti to each node di (i.e., if j=1) then Ri(X) can be assigned X0. However, if there is a path from each di to Ti (i.e.,if j′=k) then Ri(X) can be assigned Xk.

4.3 Selecting an appropriate version

Although Theorem 1 ensures that read operations can always be granted without creating a cycle in the dependency graph, it does not specify how we can obtain the boundaries j and j′ (refer to Theorem 1).

In this subsection we introduce an efficient algorithm for determining these boundaries by modifying depth first search. We only show how we can determine j, i.e., the least version number of X such that there is a path from Ti to dj, where dj is the transaction which created Xj of X. In order to determine j′ we can apply the same algorithm for the converse graph. (Note that the converse graph of a directed graph G=(V,E) can be obtained by only reversing the direction of the arcs of E, i.e., (a,b) is an arc in the converse graph iff (b,a) is an arc in G.).

Suppose that we have k+1 versions of X {X0,X1,X2,.....,Xk} at the time a read request for X is received and suppose these versions were created by a set of transactions D (the destination set)={d0,d1,d2,....,dk}, where di (i=0,1,2,...,k) refers to the transaction which created the version i of X. We call the node of the transaction which issued the read request the <u>origin node</u> (O).

We perform depth first search of DG starting at the origin node (O), with the following modification. Each time a node is visited, this node is tested to see if it is a destination node.

If so, then we record this node and immediately backtrack without searching through it (it cannot lead us to another node which created a smaller version).

The search algorithm can be described precisely by the following recursive algorithm.

<u>Algorithm SEARCH(0)</u>

1- For each node vi adjacent to 0 do the following.

2- Check to see if vi is marked.

3- If vi is marked, then select another node adjacent to 0.

4- If vi is not marked, then mark it and check to see if it is a destination node.

5- If vi is a destination node, then add it to the output set and go to step 1.

6- If vi is not a destination node, then call SEARCH(vi).

The above search finishes when all the paths starting from 0 are explored. Each time a destination node is found we add it to the output set. Suppose that the search is finished and the output set is,

$D'=\{di1,di2,di3,...\}$

where $i1<i2<i3<...$ Let j be as defined in the proof of Theorem 1. Then $j=i1$ and therefore the read operation Ri(X) can return any version of X with version number between j′ and i1, where j′ is the other boundary to be obtained from the converse graph (refer

to Theorem 1).

If the search is finished without getting any destination then Ri(X) can return any version of X with version number between j′ and k. Obviously, in the previous algorithm if the destination set D contains only one node, i.e., contains d0, we do not need to search using the previous algorithm. We simply select d0, i.e., the read operation returns version X0. This clearly does not create a cycle in the dependency graph.

Lemma 2

Selecting the appropriate version can be done in time linear in the number of arcs in the current dependency graph.

Proof

Since in the previous search algorithm we do not search through each node more than once then this algorithm works in time linear in the number of arcs in the dependency graph. (Q.E.D.)

The algorithm for determining the appropriate version number for a new version to be created by a write operation Wi(X) is quite similar to the algorithm used for the read operation. If Ti is a write only transaction for X then we can determine j and j′ as described previously. Otherwise, if Ti is a read-write transaction for X and Ri(X) returned a version of X, say Xn, before the write operation Wi(X), then there must be an arc from each node dl ($l \leq n$) to node Ti and from node Ti to each node dl′ ($l' > n$) according to rules a and c mentioned on page 25. This actually means that n is the largest version number of X such that there is a path from dn to Ti and n+1 is the least version number of X such that there is a path from Ti to dn+1 where dn and dn+1 are the transactions which created versions Xn and Xn+1, respectively. In this case j′=n and j=n+1.

In either case, Wi(X) would create a version of X with version number between j′ and j. The following theorem (Theorem 2) shows the cases in which the creation of such transaction will (or will not) create a cycle in the dependency graph.

<u>Theorem 2</u>

Let Xj be the version of X with the least version number such that there is a path from Ti to dj and there is no path from Ti to any other transaction, if any, that read Xj-1 and let Xj′ be the version of X with the largest version number such that there is a path from dj′ to Ti, where dj and dj′ refer to the

transactions which created versions Xj and Xj′, respectively.

A late write operation Wi(X) creating an earlier version of X, say Xi′, directly before Xm (and after Xm-1) does not create a cycle iff dm is a write only transaction for X where $j' < i < m \leq j$.

Proof

Suppose that dm is a write only transaction for X. Then creating Xi′ will introduce new arcs in the dependency graph as follows (provided that those arcs do not already exist).

1- An arc is introduced from each node dl to node Ti where dl is a transaction which created (or returned) a version of X with version number less than or equal to m-1.

2- An arc is introduced from node Ti to each node dl′ where dl′ is a transaction which created (or returned) version of X with version number greater than or equal to m.

The newly introduced arcs represent a path from each node dl to a node dl′ through Ti, where dl and dl′ are defined as above. But since there was no path from node Ti to any node dl or from any node dl′ to node Ti and there was already a path from each node dl to each node dl′ before the write operation Wi(X) then the newly introduced arcs do not create a cycle in the dependency graph.

Suppose that creating Xi′ ($j'<i'<m\leq j$) does not create a cycle, i.e., the newly introduced arcs due to the write operation do not create a cycle. In order to prove that dm must be a write only transaction for X we will assume first that dm is a read-write transaction for X. Since dm returned a version of X with version number less than i′, therefore there must be an arc from dm to Ti. But since Ti created a version of X with version number less than the version number of the version created by dm, therefore there must be another arc from Ti to dm, i.e., the newly introduced arcs create a cycle in the dependency graph which contradicts our initial assumption that the newly introduced arcs do not create a cycle in the dependency graph. (Q.E.D.)

The above theorem implies that the set IMVSR contains all the schedules (executions) which can be accepted by previously introduced multi-version schemes [STEA-76, MURO-81, MURO-82].

In the multi-version scheme introduced by Stearns, et al. [STEA-76], a transaction Ti must read a data item X in order to update it. We have removed this restriction and proved that the removal of this restriction can lead to accepting a class of write operations which would otherwise not be accepted.

In the multi-version scheme introduced by Muro, et al. [MURO-81, MURO-82], the write operations create version numbers according to the arrival time, i.e., if Wi(X) precedes Wj(X) then

F2(Wi(X))<F2(Wj(X)). It is easy to see that all the late write operations would be rejected and any execution which can be accepted by their scheme can also be accepted by our scheme.

Papadimitriou, et al. [PAPA-82] defines a set DMVSR of multi-version schedules in terms of a polynomial time algorithm for testing memebership in it. Since the input to the algorithm is an execution without the function F2, we cannot use their algorithm as a scheduling algorithm. Here we shall show that there is an execution e in IMVSR which is not in DMVSR if F2 is removed from e. Consider, for example, the following execution (we express it without the function F2)
e=W1(Y)W2(Y)W2(X)R1(X)W1(X).

There would be a cycle due to arcs from T1 to T2 (since T2 creates the next version of Y after T1) and from T2 to T1 (since T1 returns the version of X created by T2). However, this execution can be accepted in our scheme in this way
e=W1(Y1)W2(Y2)W2(X1)R1(X0)W1(X.5).

If we constructed the dependency graph for this execution it would be acyclic (only one arc from T1 to T2).

5. IMVSR-Scheduler

In this section, we describe the IMVSR-scheduler and explain how the scheduler responds to each input request. We will modify the transaction model given in section 2 to include two additional operations.

1- Begin transaction, B(T).

2- Commit transaction, C(T).

In the new model each transaction Ti begins with B(Ti) and ends with C(Ti).

The input to the IMVSR-Scheduler is the sequence of arriving requests from user transactions including the begin and the commit operations.

5.1 Processing the begin operation

Processing the begin operation is trivial. The begin operation indicates the arrival of a new transaction. In response the each begin operation B(Ti) the scheduler creates a new node for transaction Ti.

5.2 Processing the read operation

The read operations in our scheme are always granted and each read operation returns an appropriate version without causing inconsistency (creating a cycle in the dependency graph). In response to each read request, Ri(X), the scheduler searches the versions of X currently maintained in the system and selects the appropriate version which can be assigned to Ri(X) without creating a cycle in the dependency graph (refer to Theorem 1).

5.3 Processing the write operation

Processing a write operation is quite similar to processing a read operation, but instead of choosing the appropriate version the scheduler finds the version number of the new version to be created by the write operation.

The new concept we are using in processing a write operation is that version numbers do not correspond to the order of the arrival times. Instead we may allow a late write Wi(X) to create an earlier version of X provided it does not create a cycle in the dependency graph.

When the scheduler receives a write request Wi(X), it updates its dependency graph as if it granted the write operation. If the newly introduced arcs do not create a cycle in the dependency graph, then the scheduler officially grants this operation and creates a new version of X. And if the new arcs cause a cycle to be created then the partial execution received so far is not an IMVSR-execution if Wi(X) is actually appended to it. In this case the scheduler rejects the write operation and initiates the abortion process (aborting transaction Ti which issued Wi(X) and all transactions strongly dependent on Ti). The abortion process may propagate to include many other transactions.

5.4 Processing the commit operation

The commit operation is the last operation of each transaction. If the scheduler grants a commit request by a transaction Ti, we say that Ti has been committed, which means that all the operations of Ti have been successfully processed, Ti will not be aborted in the future, and the effects of Ti will be made permanent. However, there is an important condition which must be satisfied by any transaction to be deleted from the dependency graph. When the scheduler receives a commit request from a transaction Ti, it checks to see if Ti satisfies this condition. If the condition is not satisfied then Ti must wait (add it to a wait list) until the condition is satisfied. If the condition is satisfied then the scheduler deletes Tk from the dependency graph together with all the arcs directed from it (outgoing arcs). However, a transaction may be committed before it is deleted.

A source node

We call a node v of a directed graph a source node if it has no incoming arcs, i.e., there is no arc directed from any other node to v.

Obviously from the previous definition if a node Tk is a source node in the dependency graph then transaction Tk is not dependent on any other transaction in the system.

5.4.1 The deletion condition

A transaction Tk is said to satisfy the deletion condition after Commit(Tk) has been received by the scheduler if its node in the dependency graph is a source node.

Commit request by a transaction Tk may be granted if it is a source node with respect to the primary arcs, before it is deleted.

Aborting or deleting a transaction Tk may make another transaction Tj satisfy the deletion condition. In our scheme, after a deletion or abortion the scheduler checks the wait list to see if any transaction (or a set of transactions) satisfies the deletion condition.

Clearly, a source node in the dependency graph cannot be involved in a cycle since it has only outgoing arcs. Moreover, if a transaction Tk satisfies the deletion condition (i.e., its node in the dependency graph is a source node and Commit(Tk) has been received by the scheduler) its node in the dependency graph will remain a source node since it will not issue any new request. Thus, the deletion of transaction Tk satifying the deletion condition will not cause any problem in the future (i.e., Tk cannot get involved in a cycle).

When we delete a transaction Tk we delete its node in the dependency graph and all arcs going out of this node. If

transaction Tk created a version of a data item X, say Xk′, we also delete any version of X with number less than k′. Obviously there will be only one version of X with number less than k′ (otherwise the node Tk would not be a source node). This version may have been created by the fictitious transaction T0 or any other previously committed transaction.

Theorem 1 was proved with the implicit assumption that all versions are kept and the dependency graph contains all transactions. Let Ti and dj be as defined in the proof of Theorem 1. Since only source nodes are deleted, if there is a path from Ti to dj in the complete dependency graph, then the same path must be in the current (pruned) dependency graph. Therefore the version Xj in the proof of Theorem 1 can be found using the current dependency graph. We show here furthermore that each read request can be granted a version that is still kept by the system.

Let Tk and Xk′ be as defined above (i.e., transaction Tk satisfies the deletion condition and created version Xk′ of X) and let Xk1′ be a version of X created by a previously deleted transaction, i.e., $k1' < k'$. If a read operation can be assigned Xk1′ without creating a cycle in the (complete unpruned) dependency graph, then k1′ must satisfy the boundary constraints of Theorem 1, i.e., $j' \leq k1' < j$, where j and j′ are defined as in the proof of Theorem 1. But since Tk is a source node in the current dependency graph, there can be no path from Ti to Tk.

This implies $k'<j$. We thus have $j'\leq k1'<k'<j$, i.e., k' also satisfies the the constraints of Theorem 1. This implies that a read operation which could be assigned a deleted version without creating a cycle in the dependency graph can also be assigned Xk' without creating a cycle in the dependency graph.

6. Cyclic restart and long transactions

Suppose that we have two transactions executing concurrently as shown in figure 5, and also that the two transactions access the data items X and Y according to the timing pattern.

Consider the data item X. Ti reads X0 and writes X1. Tj, on the other hand, reads X0 and tries to write a new version of X at time t=t1. Since granting Wj(X) will create a cycle involving Ti and Tj, the scheduler will abort Tj. Assuming that the restart for Tj begins approximately at time t=t1, at time t=t2 the scheduler will abort Ti, and subsequently at time t=t3 the scheduler will abort Tj again. The abortion of Ti and Tj may repeat itself forever without Ti or Tj being committed. This is an example of the "cyclic restart problem" [STEA-76].

Stearns, et al. have observed that minor changes in time may not prevent cyclic restart.

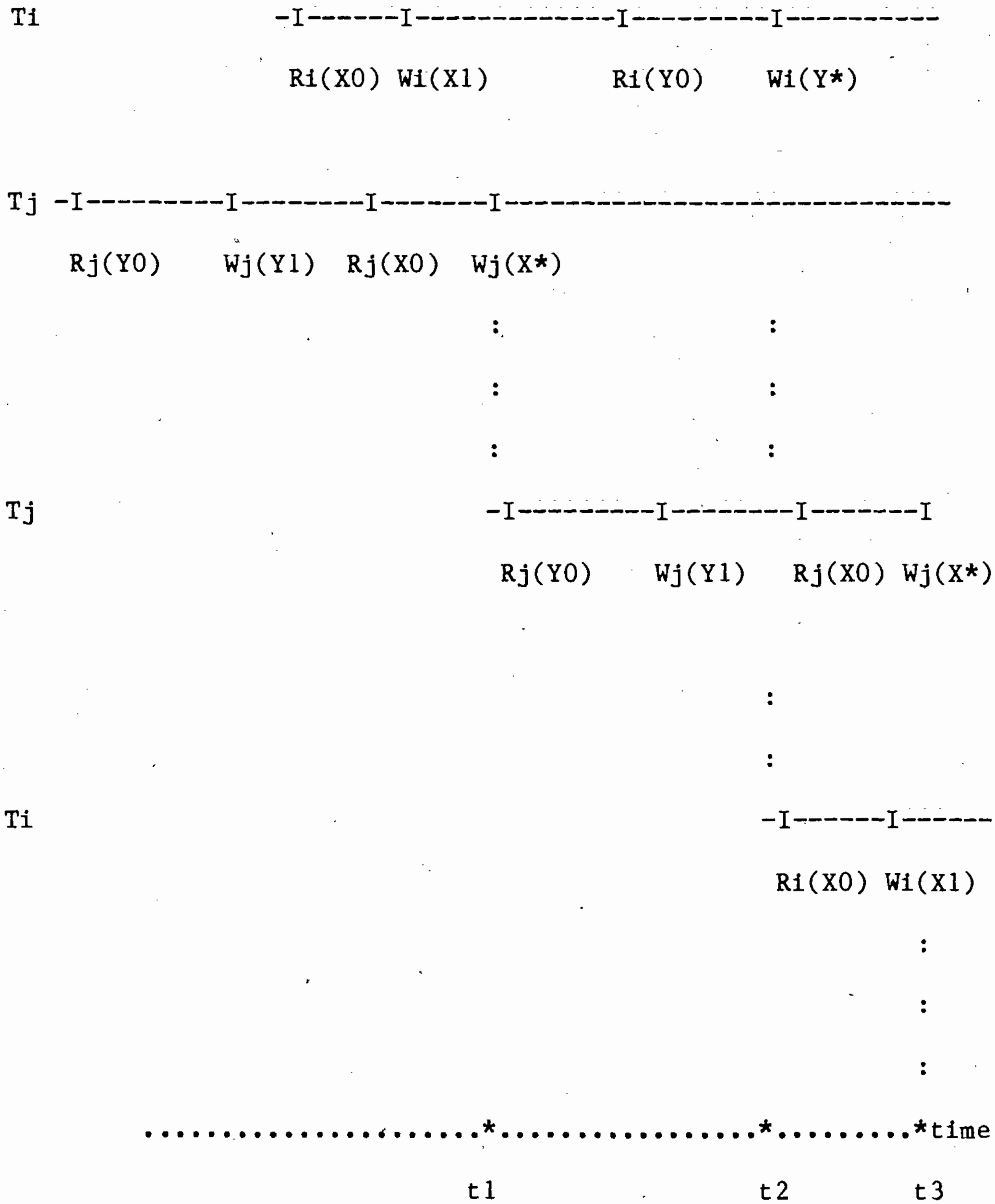


Figure 5. A cyclic restart

(Note that * associated with a write operation means rejected operation and a dashed line is drawn beginning at the start of a transaction.)

A cyclic restart may take several forms. For example, a transaction T_i may be involved in a cyclic restart with two or more different transactions individually as shown in figure 6.

```
               Ri(L0)     Wi(L1)            Ri(Z0)
Ti            -I-I-----I-I-------------I-I-------I-------------
                 Ri(X0)Wi(X1)          Ri(Y0)    Wi(Y*)
                                                 :
Tk -I---------I---------I-------I--------------------------------
    Rk(Z0)    Wk(Z1)    Rk(L0)  Wk(L*)
                                :
Tj -I---------I-------I-------I---------------------------------
    Rj(Y0)    Wj(Y1)  Rj(X0)  Wj(X*)
                              : :                :
                              : :                :
Tj                            -I---------I---------I-------I---
                               Rj(Y0)    Wj(Y1)    Rj(X0)  Wj(X*)
Tk                              -I---------I---------I-------I---
                                 Rk(Z0)    Wk(Z1)    Rk(L0)  Wk(L*)
                                                 :         : :
                                                 :         : :
Ti                                               -I-I-----I-I----
                                                  Ri(L0)
                                                           : :

                                                           : :
        ....................*.*................*.........*.*.t
                            t1t2               t3        t4t5
```

Figure 6. Ti is involved with Tj in a cyclic restart and with Tk in another cyclic restart.

It is also possible for a cyclic restart to involve a large number of transactions. For example the three transactions Ti,Tj, and Tk are involved in a cyclic restart in figure 7.

```
Ti  -----I----------I---------I---------I----------------------

         Ri(X0)     Wi(X1)    Ri(Y0)    Wi(Y*)

                                        :

Tj                 -------I---------I---------I---------I--------

                          Rj(Y0)    Wj(Y1)    Rj(Z0)    Wj(Z*)

                                    :                   :

Tk                           -------I---------I---------I---------I-------

                                    Rk(Z0)    Wk(Z1)    Rk(X1)    Wk(X*)

                                    :                   :         :

Ti                                  -----I---------I---------I---------I--

                                         Ri(X0)    Wi(X1)    Ri(Y0)    Wi(Y*)

                                    :              :              :    :

Tj                                                 -----I---------I------

                                                        Rj(Y0)    Wj(Y1)

Tk                                                                -----I---

                                                                       Rk(Z0)

                                    :              :              :    :

Ti                                                                     ---

                                    :              :              :    :

                                    t1             t2             t3   t4
```

Figure 7. A cyclic restart involving three transactions Ti, Tj, and Tk.

A cyclic restart does not necessarily mean aborting one transaction at a time. In some cases two or more transactions may be involved in a cyclic restart in such a way that all the transactions may be aborted at the same time. For example, in figure 8 transaction Ti and transaction Tj are involved in a cyclic restart in which the two transactions will be aborted simultaneously at time t=t1,2t1,3t1,... (Note that when the scheduler aborts Ti it also aborts Tj since Tj is strongly dependent on Ti).

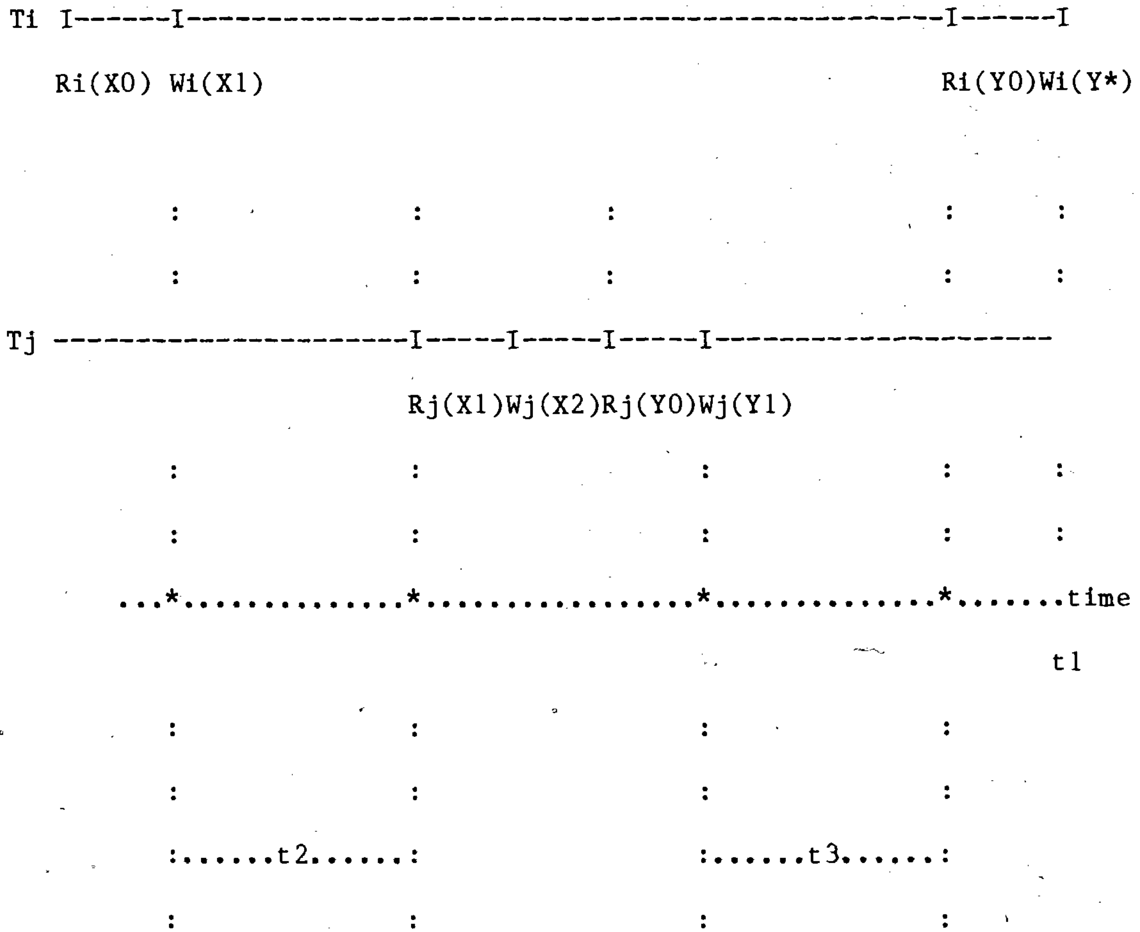


Figure 8. Transaction Ti and transaction Tj are involved in a cyclic restart in which the two transactions will be aborted simultaneously at time t=t1,2t1,3t1,....
The bottom part of this figure will be used later.

From the above examples we conclude that a transaction Ti may be involved in one or more cyclic restarts simultaneously. If a transaction is involved in a cyclic restart then it may stay in the system forever.

We may also ask whether it is guaranteed for each transaction to commit if it is not involved in a cyclic restart. Unfortunately, the answer is no. For example a transaction Ti (most likely, a long transaction) may stay in the system forever without being involved in a cyclic restart because it conflicts with different transactions each time [KUNG-81]. Ti may be aborted repeatedly without ever finishing. At the same time the existence of such a transaction may cause several transactions to be aborted.

Therefore the existence of a transaction which continually gets aborted does not necessarily imply a cyclic restart. This fact implies that the detection of a cyclic restart may be very difficult, if not impossible. We thus must cure a "desease" without knowing that it is there.

6.1 Detecting a cyclic restart

Since a cyclic restart may take many different forms, detecting a cyclic restart may not be an easy task. We shall give some examples to show that detecting a cyclic restart is indeed very difficult.

Transaction Ti and transaction Tj may be aborted due to conflicts with each other one or more times and they may subsequently proceed to completion as shown in figure 9.

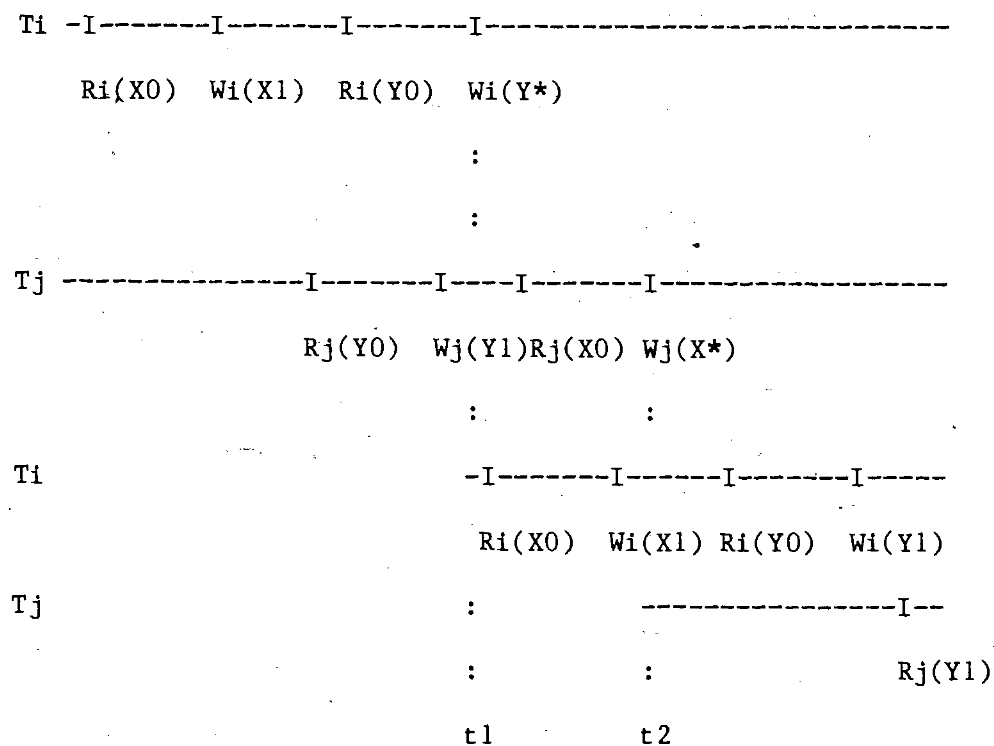


Figure 9. Transaction Ti and transaction Tj are aborted due to conflict with each other and they subsequently proceed to completion.

Transaction Ti may be aborted due to conflict with more than one transaction as shown in figure 10.

A long transaction may also be aborted due to conflict with one or more transactions and may cause other transactions to be aborted and still this case may not be a cyclic restart.

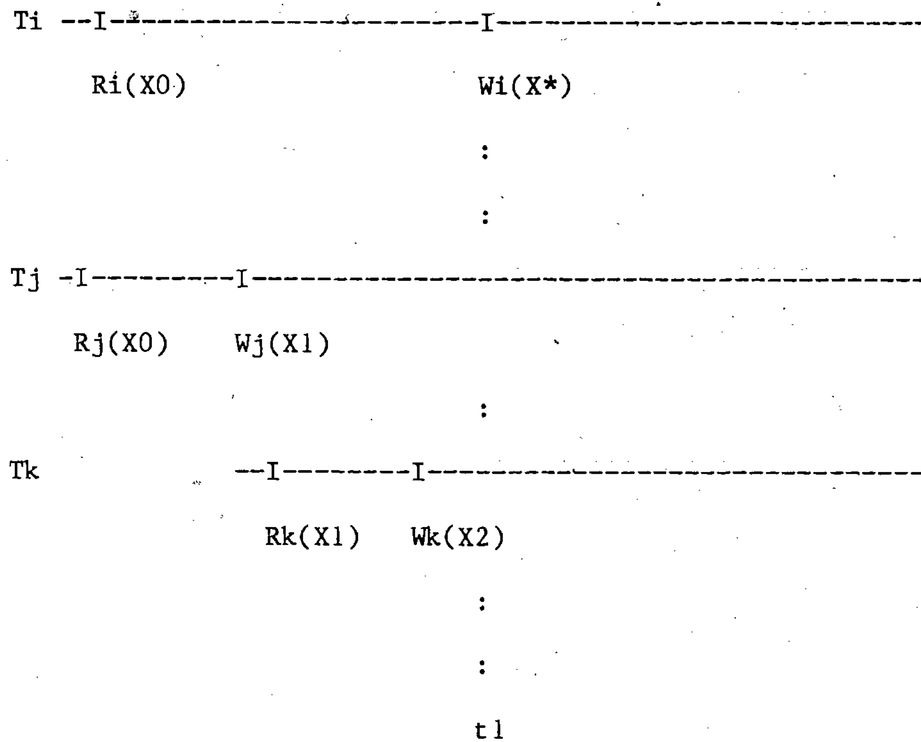


Figure 10. Transaction Ti is aborted because of transaction Tj and transaction Tk.

How can we prevent a cyclic restart from repeating itself forever and guarantee long transactions to commit?

The major difficulty in coping with cyclic restart is our inability to determine the cause of an abortion as cyclic restart, as the above examples illustrate. Therefore we cannot directly deal with cyclic restarts. Our approach will be, of necessity, indirect.

One approach to solving this problem is to assign different priorities to different transactions (Bernstein, et al. [BERN-81], this method was proposed for solving cyclic restarts due to deadlock) and to test priority to decide which of the conflicting transactions to abort. For example, we would abort Ti for Tj only if Tj had a higher priority than Ti (i.e., P(Ti)<P(Tj), where P(Ti) is the priority assigned to Ti).

One problem with this technique is that some unfortunate transaction (most likely long transaction) may stay in the system forever because it conflicts with a different transaction each time.

Another problem may arise when a transaction Ti is involved in a cyclic restart with two transactions Tj and Tk where P(Tj)<P(Ti)<P(Tk). In this case we cannot easily apply the previous policy for deciding which transaction to abort. This is because if Tj and Tk are strongly dependent on Ti, the scheduler

will abort the three transactions. Note that selecting Tj for abortion may lead to aborting Tk, which has a higher priority than Ti, if Tk is strongly dependent on Tj.

Another approach is to use timestamp (cf [STEA-76]). In this approach each transaction is given a unique timestamp when it arrives. This timestamp is greater than any timestamp given to previously received transaction. If two transactions Ti and Tj conflict, the order of time stamps between Ti and Tj is used to determine whether Ti or Tj should be restarted.

This techinque is quite similar to the priority technique except that we are assigning a timestamp instead of a priority (note, in the previous technique a new transaction may be given a higher priority than an old transaction which arrived earlier).

As a result, it has the same disadvantages except that long transactions are guaranteed to commit.

Another possibility for solving the cyclic restart problem might be to use a random time delay. Whenever a transaction Ti is aborted we randomly delay this transaction instead of restarting it immediately. Obviously in this method if Ti and Tj are involved in a cyclic restart it is not guaranteed to break this cycle after the first delay and perhaps we may abort (and randomly delay) Ti and Tj many times before breaking the cycle.

In some cases this method may even fail to prevent the cyclic restart from repeating itself forever. For example, in the cyclic restart shown in figure 8, Ti and Tj will be aborted (and randomly delayed) at the same time. If the difference between the time delays of Ti and Tj is such that Wi(X) precedes Rj(X) and Wj(Y) precedes Ri(Y) the two transactions may stay forever (refer to figure 8). That is because the same conflicts between Ti and Tj will occur again.

We may also delay transactions unnecessarily even if the abortion is not due to a cyclic restart. Another problem with this scheme is that long transactions may stay in the system forever.

A simple solution for the previous problems (a cyclic restart and long transactions) is to assign each transaction a counter (abortion counter) which indicates the number of times this transaction has been aborted. When this count exceeds a specific limit, it will not be restarted again. This transaction will wait until all other transactions aboted before (presumably because they conflict with this transaction) will be executed for completion. After that this transaction will be executed without abortion. If during the time this transaction waits one of those transactions exceeds the limit, it is not restarted immediately, but will be executed after the execution of this transaction. The scheduler will not allow two (or more) transactions which exceeded the abortion limit to execute together (otherwise they

may conflict with each other). Executing this transaction without abortion does not mean executing it alone. It simply means that when this transaction is executed, the scheduler will abort any other transaction that conflicts with it.

This solution is not too restrictive compared to some other methods of control. For example, in some variations of <u>two-phase locking</u> a transaction may hold all locks until termination (see Bernstein, et al. [BERN-81]).

It is clear that this solution prevents cyclic restart and guarantees long transactions to commit. It can also deal with all the different forms of cyclic restart mentioned before. A long transaction Ti involved with one or more transactions in a cyclic restart is executed later (it is most likely for Ti to exceed the limit before the other transactions).

Kung, et al. [KUNG-81] have introduced a solution for a similar but different problem, i.e., "starving" long transactions, based on keeping track of the number of times a transaction is restarted. When this count exceeds a limit, the corresponding transaction gets the highest priority and is executed alone while all other transactions wait for its completion. Obviously, their solution is the opposite of ours. Our justification is based on the idea of isolating the cause of the problem (presumably the long transaction which exceeded the limit). Stated another way, it is better to make only one

transaction wait and to achieve as much concurrency as possible.

Unlike their method, we let a transaction which exceeds the limit execute with other transactions.

The disadvantage of our solution is that conflicting transactions may be restarted several times before they proceed to completion. But since detecting cyclic restarts is very difficult as we have shown, this problem may arise in most of the other solutions.

There is probably no single best method for resolving cyclic restart and starving transactions. The performance of each method will depend on characteristics, such as lengths or frequency of the write operations of the transactions. A possible further research would be to compare the different methods mentioned above for various sets of transactions.

7. Conclusions

We have presented a new multi-version scheme for controlling concurrent accesses to a database system. Multiple versions of each data item are maintained. When the scheduler receives a read operation for a data item X, Ri(X), it searches the versions of X currently maintained in the system and selects the appropriate version which can be assigned to Ri(X) without creating a cycle in the dependency graph.

Processing a write operation is quite similar to processing a read operation but instead of selecting the appropriate version the scheduler finds an appropriate version number for the new version to be created (refer to Theorem 2).

Unlike previously introduced multi-version schemes [STEA-76, MURO-81, MURO-82], a write operation may be allowed to create an earlier version. If the write operation comes in time it creates a new version with the largest version number. However, if it comes late it creates an "earlier" version, provided it does not create a cycle; otherwise the issuing transaction is aborted. The abortion process may propagate to include some other transactions.

An efficient algorithm for selecting the appropriate version to be read or to be created was introduced. This algorithm works in time linear in the number of arcs in the dependency graph.

We have proved that the fixpoint set of our scheduler contains those of the other schemes and therefore our scheme achieves more concurrency.

We have also analyzed the cyclic restart problem [STEA-76] and introduced a simple solution.

References

[BAYE-80] Bayer, R., Heller, H., and Reiser, A., Parallelism and recovery in database systems, ACM TODS 5, 2 (June 1980), 139-156.

[BERN-81] Bernstein P.A. and Goodman N., On concurrency control in distributed database systems, ACM Computing Surveys 13 (June 1981), 185-221.

[ESWA-76] Eswaran, K. P., Gray, J.N., Lorie, R.A., and Traiger, I.L., The notions of consistency and predicate locks in a database system, CACM 19, 11 (Nov. 1976), 624-633.

[KESS-80] Kessels, J. L. W., The readers and writers problem avoided, Info. Process. Letts. 10, 3 (April 1980), 159-162.

[KUNG-79] Kung, H. T., and Papadimitriou, C. H., An optimality theory of concurrency control for databases, Proc. ACM-SIGMOD, Intl. conf. on management of data, (May 1979), PP. 116-126.

[KUNG-81] Kung, H. T., and Robinson, J. T., On optimistic methods for concurrency control, ACM TODS 6, 2 (june 1981), 213-226.

[MURO-81] Muro, S., Minoura, T., and Kameda, T., Multi-version concurrency control for a database system, CCNG Report E-98, Computer Communications Networks Group, University of Waterloo, (Aug. 1981).

[MURO-82] Muro, S., Kameda, T., and Minoura, T., Multi-version concurrency control scheme for a database system, TR 82-2, Department of Computing Science, Simon Fraser University, (February, 1982).

[PAPA-79] Papadimitriou, C. H., The serializability of database updates, JACM 26, 4 (Oct. 1979), 631-653.

[PAPA-82] Papadimtriou, C.H. and Kanellakis, P.C., On concurrency control by multiple versions, Proc. ACM Symp. on Principles of Database Systems (March 1982), 76-82.

[REED-79] Reed, D.P., Implementing atomic actions on decentralized data, Proc. 7th ACM Symp. on Operating Systems Principles, (Dec.1979), 66-74.

[STEA-76] Stearn, R., Lewis, P., and Rosenkrantz, D., Concurrency control for database systems, Proc. IEEE Symp. Foundation of Comp. Sci. Houston, Texas, (Oct. 1976), 19-32.