CANADIAN THESES

THÈSES CANADIENNES

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 an inferior 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.

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 qualité inférieure.

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.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

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

Canada

THE UNIVERSITY OF ALBERTA

CONCURRENCY AND CONSISTENCY IN DATABASE SYSTEMS

by

Abdel Aziz Farrag

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1986

ISBN 0-315-30323-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Abdel Aziz Abdel Hamid M. Farrag

TITLE OF THESIS Concurrency and consistency in database systems

DEGREE FOR WHICH THESIS WAS GRANTED Doctor of Philosophy

YEAR THIS DEGREE WAS GRANTED 1986

(Signed) A. Farrag

Permanent address:
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada

Dated Jan. 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommended to the Faculty of graduate studies and research, for acceptance, a thesis entitled "CONCURRENCY AND CONSISTENCY IN DATABASE SYSTEMS" submitted by Abdel Aziz A. Farrag in partial fulfillment of the requirement for the degree of Doctor of Philosophy.

.....

Supervisor

N. Jaishankar Menon

.....

.....

.....

Date 2 Jan 86 W. A. Dawson

Dedication

To my mother in Egypt, the heroine whose name has mistakenly disappeared from all historical books, and to my wife Monica, the angel whose name has not been mentioned in any religious book.

ABSTRACT

This thesis is concerned with concurrency control in database systems. The main objective is to explore new concurrency control mechanisms that provide more concurrency than the currently available approaches. Two topics are examined and several new results are presented. The first topic deals with the concurrency control problem when the only information available about the transactions is syntactic information. A new concurrency control mechanism is presented and shown that it works correctly. The proposed mechanism is more general than any previously introduced mechanism, because it contains as many special cases as the degree of multiprogramming of the system. In particular, two-phase locking and timestamp ordering are special cases of the proposed mechanism. Several other special cases of the proposed mechanism are also presented. Each of these special cases can be selected in advance, and can even be changed dynamically during execution.

The second topic deals with the concurrency control problem when semantic information is available about the transactions. This semantic information takes the form of transaction types, transaction steps, and transaction breakpoints. A new class of "safe" schedules called relatively consistent (RC) schedules is defined, and a new concurrency control mechanism which allows only RC schedules is proposed. The class of RC schedules contains serializable and non-serializable schedules, and the proposed mechanism assumes fewer restrictions on the interleaving among transactions than the previously proposed mechanisms which allow non-serializable schedules. Consequently, our mechanism achieves higher level of concurrency.

Acknowledgement

I would like to thank professors W. Davis, E. Chan, and G. Fisher for their encouragement and useful comments. Special thanks also go to professors S. Cabay, C. Addison, T. Hanson, J. Tartar and T, Marsland for their support. Prof. S. Cabay has always been very friendly since I moved to University of Alberta. His encouragement, helpful advice, and broad knowledge have significantly improved my research skills.

My supervisor professor M. T. Ozsu has been extremely helpful during the process of writing this thesis. He has provided me with very useful comments and helped me in developing many of the ideas reported in this thesis. His continuous support and very friendly attitude made the task of completing my Ph.D. degree much easier. He has also helped me in conducting my job interviews, in presenting my research work and even in answering the questions about my thesis. For all these things I express my most sincere thanks.

My mother, in Egypt, has struggled for many years to give her children the chance to continue their education. She had to depend on very low widow's pension to raise eight children until they finished their university degrees. I owe her every success in my life, and without whose continuous support I would never have been able to finish any degree.

My wife, Monica, has greatly contributed to this thesis. Since I met with her in September 1982, my life has been completely changed and became easier, sweeter, and more enjoyable. She has always been my closest friend and my greatest supporter. Her kindness and tireless patience were the main reasons for my success in completing my Ph.D. degree.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The problem of coordinating concurrent accesses to a database system has been studied by many researchers and several concurrency control algorithms have been introduced [Esw76, Far85, Gar83, Kun81, Ree78, Ros78]. The task of a concurrency control algorithm is to ensure the consistency of the database while allowing a set of transactions to execute concurrently. A *database* is a collection of entities named x, y, z, etc. Each entity has a single value. The database is said to be *consistent* if the values of its entities satisfy a set of constraints called *integrity* constraints. Examples of integrity constraints are as follows.

1. In a banking system, the sum of the values assigned to the loans for a particular account must be less than the line of credit of that account.
2. In an airline reservation system, the number of assigned seats must not be greater than the capacity of the aircraft.

Each *transaction* is assumed to be a correct computation which preserves the integrity constraints of the database, i.e., when executed by itself it transforms a consistent database state into a new consistent state. It is also assumed that the concurrency control mechanism does not know the integrity constraints of the database, but operates in such a manner which preserves these constraints.

The following example demonstrates the importance of the problem and shows that in the absence of concurrency control, concurrent execution of a set of transactions can transfer a consistent database state into an inconsistent state.

Example 1.1:

Consider the two transactions T_1 and T_2 described below.

T_1: x ← x+1,

y ← y+1.

T_2: y ← 2*y,

x ← 2*x.

Suppose that the consistency constraint on x and y is that x = y. Consider the following sequence of execution:

T_1: x ← x+1,

T_2: y ← 2*y,

T_1: y ← y+1,

T_2: x ← 2*x.

It is not difficult to see that after executing T_1 and T_2 concurrently as described above, the values of x and y will not be consistent (i.e., the values of x and y will not be the same).

In practice, the entities of the database may represent accounts in a bank, reserved seats on a flight of an airline, etc. An inconsistent value for an entity x in a banking system (or airline reservation system) will mean loss of money (or loss of seats, respectively). In such systems, the database must always be consistent, even in the face of failure [Ber83, Tra82].

One obvious solution to the concurrency control problem is execute the transactions sequentially, that is, one at a time. Although this solution guarantees the consistency of the database, it eliminates concurrency and hence causes serious performance degradation.

In this thesis, we are interested in concurrency control mechanisms which ensure the consistency of the database while allowing as much concurrency as possible. Moreover, we are also interested in those database systems in which the concurrency control mechanism performs at the system level, i.e., is invisible the users (see Fig. 1).

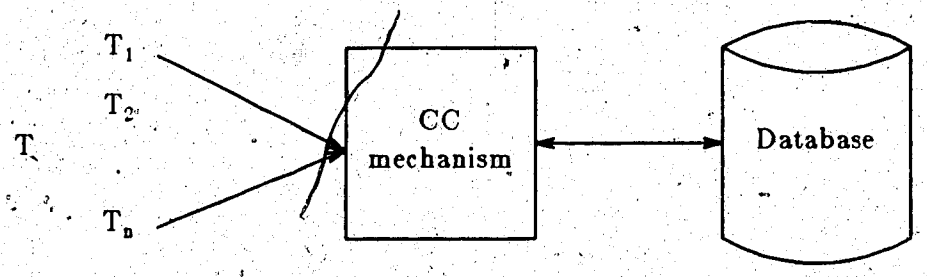

Figure 1 Database system model.

During the concurrent execution of a set of transactions, the concurrency control mechanism must ensure the following.

1. Each transaction sees a consistent database.
2. Each transaction eventually terminates.
3. The final database state after all transactions terminate is consistent.

The first condition indicates that although transactions are executed concurrently, each transaction will have the illusion that it is executing alone on a

dedicated database.

The second condition means that the mechanism must ensure the elimination of deadlock (and any other problem) which could prevent the termination of the transaction.

The third condition indicates that each concurrent execution allowed by the mechanism cannot violate consistency.

1.1. Objectives of this thesis

The main objectives of this thesis are to explore new mechanisms for concurrency control, which are more general than the currently available mechanisms and which provide more concurrency. Increasing concurrency results in increasing the system throughput and will improve resource utilization. Furthermore, since transaction delays and abortions are reduced, one would expect the transaction response times to decrease as well.

It should be noted, however, that the actual performance of various mechanisms is not the primary concern of this thesis. We have not run any realistic performance experiments. The emphasis is purely on developing the theoretical basis underlying these mechanisms. We only touch upon the performance issue in terms of the types of experiments that need to be run to evaluate the techniques proposed in this thesis. Therefore, even though, the mechanisms proposed here are more general, no claim is made as to their performance in comparison to more restricted algorithms.

We examine the concurrency control problem for two cases: when the only information available about the transactions is syntactic information, and when there is some semantic information available about the transactions. The semantic informa-

tion considered is transaction types, transaction steps and transaction breakpoints.

1.1.1. Using syntactic information of transactions

When the only information available about the transactions is syntactic information, *serializability* [Esw76] is the main correctness criterion for concurrency control. Serializability means equivalence to a *serial* execution of the transactions. A concurrent execution of a set of transactions or a *schedule* is said to be *serializable* if it is equivalent to (i.e., has the same effect on the database as) serial schedule in which the transactions are executed sequentially, in some order. Since the execution of a set of transactions serially cannot violate the integrity constraints of the database, the execution of a serializable schedule also preserves these constraints. The concurrency control mechanism performs its job by producing only serializable schedules.

The means for guarantying the serializability of the executing transactions are diverse. The most popular means are two-phase locking [Bay80, Esw76, Gra78] and timestamp ordering [Ber81, Lam76, Ree78].

In this thesis, a new concurrency control algorithm is proposed and proven to work correctly, i.e., produces only serializable schedules. The proposed mechanism is more general than any previously introduced concurrency control algorithm. This is because it has as many special cases as the multiprogramming level of the system. In particular, two-phase locking and timestamp ordering, which are considered to be the most popular (and practical) concurrency control algorithms, represent special cases of the proposed mechanism.

The set of transactions executing at any time under the proposed concurrency control algorithm can be partitioned into a set of classes called strict classes. Two-phase locking corresponds to a special case in which all executing transactions will

belong to the same strict class and timestamp ordering corresponds to a special case in which every executing transaction will belong to a different strict class. The maximum number of transactions which can belong to the same strict class is called the strictness level of the concurrency control algorithm. The value of the strictness level can be specified in advance and can be modified during execution.

We show that two-phase locking and timestamp ordering represent the two end points of a series of concurrency control algorithms. Each results from choosing a different value for the strictness level and is considered to be a special case of the general concurrency control algorithm described in this thesis.

1.1.2 Using semantic information of transactions

The requirement of serializability that the only acceptable schedules are those which are serializable may be reasonable when transactions are relatively short. But when transactions are long, this requirement is too strong and can lead to increased transaction response time.

Under the assumption that some semantic information is available about the transactions, the notion of serializability can be weakened to allow more concurrency [Fis81, Gar83, Lyn83]. The main idea is to allow controlled non-serializable schedules (which do not violate consistency). This represents the subject of the second topic reported in this thesis. We illustrate this idea further by the following example. (For more detailed examples the reader can also refer to [Gar83]).

Example 1.2:

Consider a banking system in which a transfer transaction can be divided into two consecutive steps, a withdrawal step followed by a deposit step. In order to

improve performance we may allow a transfer transaction to be interrupted between the withdrawal and deposit steps by another transfer transaction. This interleaving can be specified by inserting a certain command at the end of the withdrawal step which instructs the concurrency control mechanism that interleaving is allowed at this point for a transfer transaction. We will call such a command a breakpoint. In this case, a transfer transaction will appear to another transfer transaction as consisting of two (consecutive) atomic steps.

Similarly, consider another audit transaction which reads the balances of all accounts in the system. In order for the audit transaction to return consistent values, the transfer and the audit transactions must appear to one another as a single atomic step. That is, the audit transaction will be prohibited from interrupting the transfer transaction and vice versa.

Thus, the atomicity of the same transfer transaction will be viewed differently by different types of transactions. This concept is known in the literature as "relative atomicity" [Lyn83]. Which means that the atomicity of a transaction T_i viewed by another transaction T_j will only depend on the breakpoints inserted in T_i.

This thesis formalizes the concurrency control problem when semantic information are available about the transactions. A new class of "safe" schedules called relatively consistent schedules (or RC schedules for short) is defined. This class contains serializable as well as non-serializable schedules. It is proven that an RC schedule cannot violate consistency. Moreover, a new concurrency control mechanism which produces only RC schedules is proposed. The proposed mechanism achieves high level of parallelism by allowing serializable as well as non-serializable schedules to be produced.

The organization of the rest of this thesis is as follows. The first topic is dis-

cussed in Chapters 2, 3, and 4. Chapter 2 defines the basic concepts and describes the database system model used in formalizing the first topic. This chapter also reviews the previously introduced mechanisms. Chapter 3 describes a new concurrency control mechanism and proves that it works correctly, that is, produces only serializable schedules. This chapter also discusses the generality of the proposed mechanism and describes its special cases. It is shown that two-phase locking and timestamp ordering represent the two end points of a series of concurrency control mechanisms. Each of them is a special case of the proposed concurrency control mechanism. Chapter 4 discusses some related problems that are associated with the proposed mechanism and outlines some solutions to these problems.

The second topic is discussed in Chapters 5, 6, and 7. Chapter 5 introduces the concept of allowing non-serializable schedules. Chapter 6 describes the basic model that will be used in formalizing the second topic. This chapter defines a class of schedules called correct schedules. This class of schedules generalizes the class of "semantically consistent schedules" defined in [Gar83] and the class of "multilevel atomic schedules" defined in [Lyn83]. This chapter also defines a directed graph called the precedence graph. This graph represents the precedence relations among the steps of the executing transactions. This graph is used to define a new class of schedules called relatively consistent (RC) schedules. It is proven that an RC schedule cannot violate consistency. Chapter 7 proposes a new concurrency control mechanism which makes use of the semantic information available about the transactions to allow serializable and non-serializable schedules to be produced. The proposed mechanism assumes fewer restrictions on the interleavings among transactions than the locking mechanism proposed by Garcia-Molina [Gar83]. Consequently, our mechanism achieves higher level of parallelism. This chapter also proves the correctness of the proposed concurrency control mechanism. Chapter 8 concludes the thesis and discusses future work.

CHAPTER 2

BASIC CONCEPTS

This chapter defines some basic concepts which will be used throughout the thesis and reviews previously introduced mechanisms.

A *database* is a collection of entities named x, y, z, etc. A *transaction* is any sequence of read and/or write operations which preserves the integrity constraints of the database. A read operation by a transaction T_i on an entity x returns the current value of x. Similarly, a write operation on x by T_i updates the current value of x. An entity x can be read or written at most once by the same transaction. The read and the write operations of transaction T_i will be denoted $R_i(x)$ and $W_i(x)$, respectively. A transaction is assumed to be a correct computation, that is, either all its updates must be reflected in the database or none of them. A transaction is said to be *terminated* (or *committed*) when all its operations have been accepted by the concurrency control mechanism. If an operation by the transaction is rejected, the transaction must be aborted.

The above model of a transaction is called the *general model*. This model contains several other models of a transaction which appeared in the literature. The principal models are shown in Figure 2. In this figure the generality increases upwards.

If the transaction is restricted so that all the read operations appear before all the write operations, the *two-step model* [Pap79] is obtained (refer to Figure 2). Similarly, if the transaction is restricted so that an entity must be read before it is written, the *restricted model* (also called the *read-before-write model*) [Ste76] is obtained. Further, if each transaction is restricted so that an entity must be read before it is

written and all read operations appear before all the write operations, the *restricted two-step model* is obtained. Finally, we have the *action model*, where each step is an indivisible execution of a read and a write operation [Kun79]. That is, the action model is equivalent to a transaction model in which each write operation is immediately preceded by a read operation.

An example of a transaction of each of the previous models is shown below.

$T_1 = R_1(x)W_1(y)W_1(x)R_1(z)W_1(z)$. "General model"

$T_2 = R_2(x)R_2(z)W_2(y)W_2(x)W_2(z)$. "Two-step model"

$T_3 = R_3(x)R_3(y)W_3(y)W_3(x)R_3(z)$. "Restricted model"

$T_4 = R_4(x)R_4(y)R_4(z)W_4(x)W_4(y)$. "Two-step restricted model"

$T_5 = R_5(x)W_5(x)R_5(y)W_5(y)$. "Action model"

Definition 2.1: A *schedule* H of a set of transactions $T = \{T_1, T_2, .., T_n\}$ is an interleaved sequence of the operations of the transactions in T. Consider, for example, the following three transactions:

$T_1 = R_1(x)W_1(x)R_1(y)$.

$T_2 = W_2(x)W_2(y)$.

$T_3 = R_3(x)$.

An example of a schedule H of $\{T_1, T_2, T_3\}$ is

$H = R_1(x)R_3(x)W_1(x)W_2(x)R_1(y)W_2(y)$.

The order of the operations in H increases from left to right. Note that the

order of operations of the same transaction must be preserved in H. When the operations of each transaction appear consecutively (i.e., without interleaving with the other transactions) the schedule is said to be *serial*. Each serial schedule is correct, that is, its execution cannot violate the integrity constraints of the database. An example of a serial schedule is

$$H = R_1(x)W_1(x)R_1(y)R_3(x)W_2(x)W_2(y).$$

An operation $R_i(x)$ in a schedule H returns the value of x written by the write operation $W_j(x)$ if $W_j(x)$ precedes $R_i(x)$ in H and no other write operation on x appears between $W_j(x)$ and $R_i(x)$. $R_i(x)$ returns the initial value of x (i.e., the value which exists in the database before executing the schedule) if no write operation on x precedes $R_i(x)$.

Definition 2.2: Two schedules H_1 and H_2 for the same set of transactions are said to be *equivalent* if the following two conditions are satisfied.

(1) For each read operation $R_i(x)$, either $R_i(x)$ returns the value of x written by the same write operation in H_1 and H_2, or $R_i(x)$ is not preceded by any write operation on x in H_1 and H_2.

(2) For each entity x updated in both schedules, the last write operation on x must be the same in H_1 and H_2.

Condition (1) ensures that each read operation in H_1 returns the same value returned by the corresponding read operation in H_2. Condition (2) ensures that the final value written for each entity is the same in both schedules. These two conditions guarantee that H_1 has the same effect on the database as H_2. The following two schedules, for example, are equivalent.

$$H_1 = W_2(x)W_1(x)R_3(x)R_1(z)W_2(y)R_3(y)R_3(z)R_2(z)W_4(z).$$

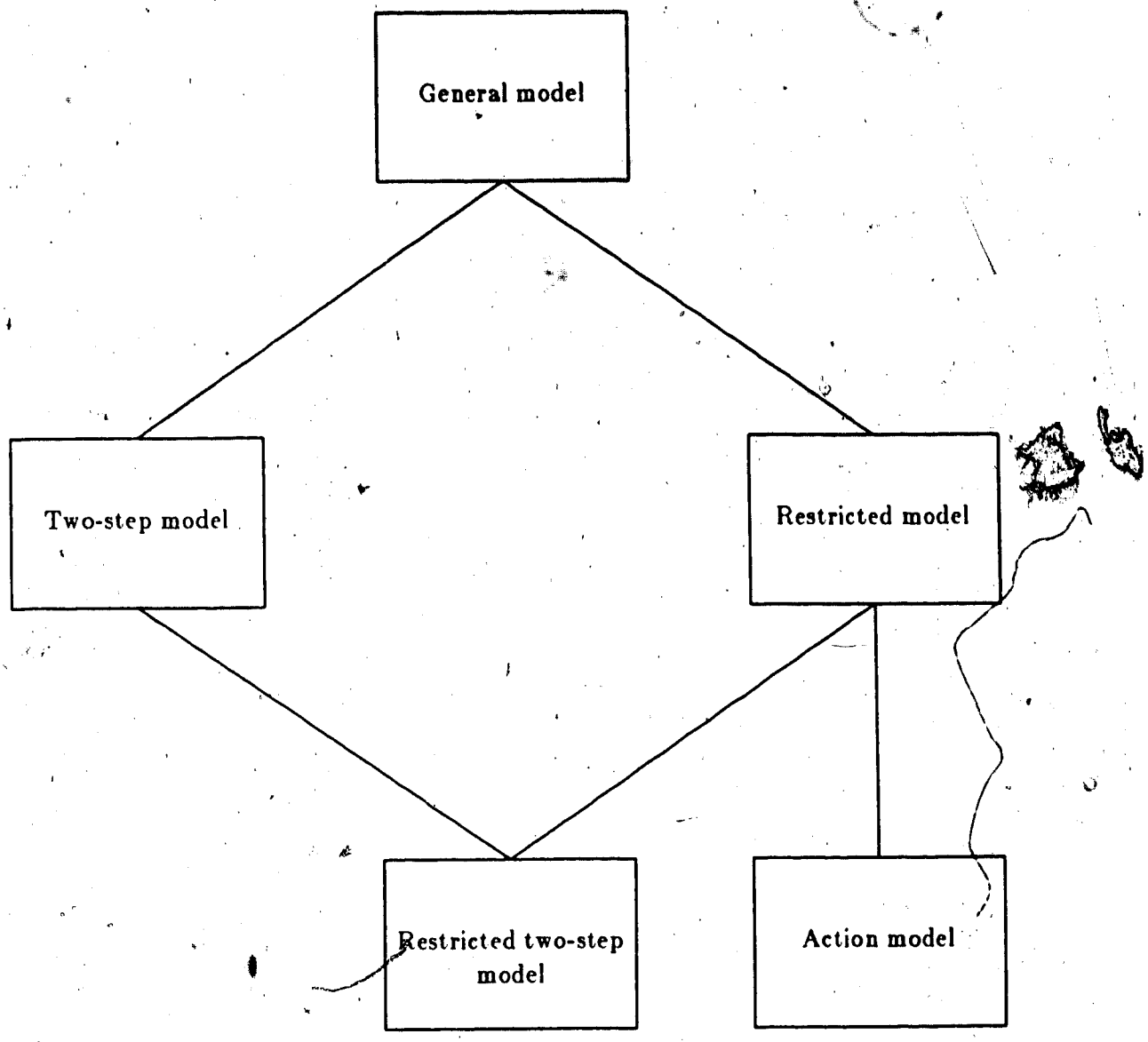

Fig. 2 Various transaction models.

$H_2 = W_2(x)W_2(y)R_2(z)W_1(x)R_1(z)R_3(x)R_3(y)R_3(z)W_4(z)$.

Definition 2.3: A schedule H is said to be *serializable* iff it is equivalent to a serial

schedule.

In the above example, the schedule H_1 is serializable since it is equivalent to the serial schedule H_2.

Definition 2.4: Two operations belonging to two different transactions T_i and T_j are said to be in *conflict* iff both operations access the same entity and at least one of them is a write operation. The transactions T_i and T_j which issue these operations are said to be *conflicting transactions.*

Intuitively, the conflict among two operations $R_i(x)$ and $W_j(x)$ indicates that the order among them is significant. Note that by the above definition read operations do not conflict and the order among them is insignificant.

2.1. Dependency graph and serializable schedules

This section defines a directed graph called the dependency graph. This graph will be used in recognizing a class of schedules called serializable schedules.

2.1.1. Dependency graph

The dependency graph DG(H) of a schedule H is a directed graph representing the conflict relations among the transactions in H. The nodes of this graph is the set of all transactions in H (i.e., each transaction T_i is represented by a node T_i). An arc (T_i,T_j) exists in DG(H) iff there is an operation in T_i which conflicts with and precedes another operation in T_j.

For example, consider the following schedule.

$$H = W_2(x)W_1(x)R_3(x)R_1(z)W_2(y)R_3(y)R_3(z)R_2(z)W_4(z).$$

The dependency graph DG(H) is constructed in Figure 3.

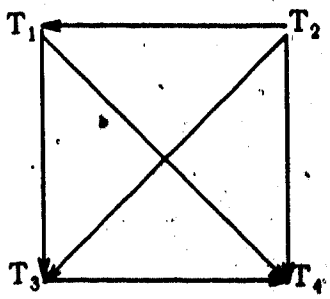

Figure 3 The dependency graph of H.

2.1.2. Serializable schedules

In order for the concurrency control mechanism to ensure the consistency of the database, it must allow only those schedules which do not violate consistency to be produced. A serializable schedule which has the same effect on the database as a serial schedule cannot violate the consistency of the database. The means for guaranteeing the serializability of the executing transactions are diverse. The most popular means are two-phase locking and timestamp ordering. The two-phase locking and the timestamp ordering mechanisms will be reviewed in later sections. But first, a theorem will be introduced for characterizing serializable schedules by checking for the absence of

a cycle in the dependency graph. This theorem will be used to analyze the concurrency control mechanisms presented in later sections and to prove that they work correctly, i.e., produce only serializable schedules.

Theorem 2.1: Suppose that the dependency graph DG(H) of a schedule H is acyclic, then H is serializable.

Proof: Let H_1 be a serial schedule formed by sorting the nodes of DG(H) topologically [Aho74] and replacing each node T_i with the operations of T_i. We claim that H and H_1 are equivalent. To prove this claim, suppose to the contrary that H and H_1 are not equivalent. For example, suppose that condition (1) of schedule equivalence is not satisfied. Let $R_i(x)$ return the value of x written by $W_j(x)$ in H and suppose first that $R_i(x)$ is not preceded by any write operation on x in H_1. Since $W_j(x)$ precedes and conflicts with $R_i(x)$ in H, there is an arc (T_j,T_i) in DG(H). This arc also implies that $W_j(x)$ precedes $R_i(x)$ in H_1 and leads to a contradiction.

A similar contradiction arises if $R_i(x)$ is preceded by a write operation on x in H_1 and is not preceded by any write operation on x in H. Similarly, suppose that $R_i(x)$ returns the value of x written by $W_j(x)$ and $W_k(x)$ in H and H_1, respectively (where $W_j(x) \neq W_k(x)$). In H, either $R_i(x)$ precedes $W_k(x)$ or vice versa. If $R_i(x)$ precedes $W_k(x)$ in H, then there is an arc (T_i,T_k) in DG(H). But this arc implies that $R_i(x)$ precedes $W_k(x)$ in H_1 and leads to a contradiction. If, on the other hand, $W_k(x)$ precedes $R_i(x)$ (and $W_j(x)$) in H, then there are arcs (T_k,T_j) and (T_j,T_i) in DG(H). These arcs also imply that $W_j(x)$ appears between $W_k(x)$ and $R_i(x)$ in H_1; a contradiction. In each of the above cases a contradiction arises, which implies that our initial assumption that condition (1) is not satisfied cannot be true.

Similarly, suppose that condition (2) of schedule equivalence is not satisfied

and let $W_j(x)$ and $W_k(x)$ be the last write operations on x in H and H_1, respectively. Since $W_k(x)$ conflicts with and precedes $W_j(x)$ in H, then there is an arc (T_k,T_j) in DG(H). But this arc implies that $W_k(x)$ precedes $W_j(x)$ in H_1 and leads to a contradiction. Therefore, condition (2) is also satisfied and H is equivalent to H_1, i.e., H is serializable. □

2.2. Two-phase locking

Two-phase locking is the most well-known mechanism for controlling concurrency. In this mechanism transactions are required to obtain locks on the entities of the database before accessing (i.e., reading or updating) these entities. In order to increase concurrency, two types of locks are allowed. These types will be denoted read-locks (also called shared-locks) and write-locks (also called exclusive-locks). Each transaction must obtain a read-lock or a write-lock on an entity x before reading or writing it, respectively.

The conditions for granting a read-lock or a write-lock on an entity x are as follows. When a transaction T_i requests a read-lock on x, this lock will only be granted if no other transaction already holds a write-lock on x. Similarly, when T_i requests a write-lock on x, this lock will only be granted if no other transaction already holds a read-lock or a write-lock on x. That is, read-locks are shared locks, i.e., several transactions can obtain a read-lock on the same entity concurrently, while write-locks are exclusive locks, i.e., no more than one transaction can obtain a write lock on x at the same time. If a read-lock or write-lock on some entity cannot be granted because the entity is already locked, the transaction which requested the lock has to wait until the lock is released.

Eswaran, et al. [Esw76] have shown that in order to guarantee consistency,

each transaction must be well-formed and two-phase. A transaction is well-formed if the following conditions are satisfied.

1. The transaction must lock an entity before accessing it.
2. The entities must be locked according to the rules described above.
3. The transaction must unlock all the entities before it terminates.

Each transaction must have two phases; a growing phase during which locks can only be obtained, followed by a shrinking phase during which locks can only be released. The point at which the transaction releases its first lock delimits the two phases. This point is called the locked point of the transaction.

In some systems, the transaction may not need to use explicit commands (like Lock(x) and Unlock(x)) to lock and release the entities it accesses. Instead, requesting or releasing locks are performed on the system level (i.e., invisible of the users). A read-lock or a write-lock is requested implicitly when the transaction submits a read or write operation, respectively, to the concurrency control mechanism. This lock will be granted if the operation is accepted; otherwise the operation is delayed. When all the operations of the transaction succeed (i.e., have been accepted), all locks are released by a single atomic action.

The above (restricted) form of two-phase locking is called *strict two-phase locking* [Bay80, Ros78] because each transaction maintains all locks until termination. Throughout this thesis the term two-phase locking will refer to a strict two-phase locking mechanism in which locks are granted and released as described above.

Two-phase locking has the drawback of reducing the level of concurrency. This is because transactions can be arbitrarily long (but finite) and each transaction must maintain all the locks it obtains during execution until it is terminated (or aborted). Other locking mechanisms which allow higher level of concurrency than two-phase

locking have been proposed [Ked79, Ked80, Sil80]. However, these mechanisms require a priori knowledge of the organization of the entities of the database and will not be considered in this thesis.

Another (and more serious) problem is *deadlock* [Gra78, Hol72].

Theorem 2.2: Every schedule produced by the two-phase locking mechanism is serializable [Esw76, Pap79].

2.2.1. Deadlock

Deadlock can occur because transactions wait for one another. Informally, a deadlock situation is a set of requests which can never be granted by the concurrency control mechanism. The following example shows how a deadlock can occur.

Consider a situation in which transactions T_1 and T_2 are currently holding write-locks on the entities x and y, respectively. Suppose that transaction T_1 requests a write-lock on y. Since y is currently locked by transaction T_2, then transaction T_1 will have to wait for transaction T_2 to release its lock on y. Suppose that while transaction T_1 is waiting for transaction T_2, transaction T_2 requests a write-lock on the entity x. Since x is currently locked by transaction T_1, then transaction T_2 must wait for transaction T_1 to release its lock on x. In this case, the two transactions may have to wait for each other indefinitely unless the mechanism has detected and resolved the problem (see Figure 4).

There are two approaches for solving the deadlock problem. The first approach is called deadlock detection and the second approach is called deadlock prevention.

Detecting a deadlock is simple and can be done by maintaining a directed

graph called the *wait-for graph*. This graph represents the wait-for relationship among the transactions. The set of nodes of this graph represents the set of transactions. An arc $T_i \rightarrow T_j$ exists in the graph if T_i is waiting for T_j to release its lock on some entity. For example, the wait-for graph for the deadlock situation described above is shown in Figure 4.

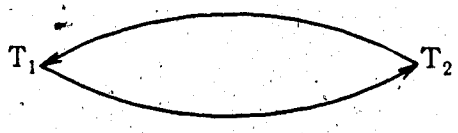

Figure 4 Deadlock situation.

A deadlock occurs when the wait-for graph contains a cycle. Once a deadlock is detected, the cycle may be broken by aborting one (or more) transaction (Normally, the last transaction which caused the deadlock will be aborted). Under the assumption that the cost of preempting each member of a set of deadlocked transactions is known, the problem of selecting the minimum total-cost set for breaking the deadlock cycle has been shown to be a difficult (NP-complete) problem [Leu79].

Deadlock can also be broken by using time-outs to revoke waits after a specified time interval. However, in practice it may be difficult to choose a workable time interval. If the time-out interval is made longer, more time will be wasted before deadlocked transactions are restarted. Furthermore, as the time-out interval is made shorter, several transactions can be restarted unnecessarily.

The deadlock problem is more difficult to handle in distributed systems. This is because some of the waiting transactions may be residing on different sites. That is, local wait-for graphs of the individual sites may not be sufficient for characterizing all deadlock situations. In this case, a deadlock can be detected by designating one site in the distributed system as a deadlock detector [Sto79]. Menasce, et al. [Men79] have shown that it is possible to detect a deadlock in distributed system without maintaining a centralized wait-for graph. This thesis does not review all mechanisms for detecting deadlock, but interested readers can refer to [Gli80, Men79, Obe82].

In order to avoid unnecessary abortion, deadlock avoidance mechanisms can be used. These mechanisms usually require transactions to preclaim all entities that will be accessed before starting executions. Preclaim strategies reduce the level of concurrency and are not feasible in environments where the entities to be accessed are not known in advance. This is particularly true for database systems designed to support concurrent user transactions that are allowed to query and update the database interactively.

2.3. Timestamp ordering

In a timestamp ordering mechanism each transaction is assigned a unique timestamp (i.e., a number) when it starts. The concurrency control mechanism uses these timestamps to resolve the conflicts among the executing transactions and to ensure that their execution cannot violate consistency. The timestamp of each transaction T_i will be attached to every read or write operation issued by the transaction T_i.

The basic timestamp odering mechanism maintains two values for each entity x, denoted TSR(x) and TSW(x). These values record the largest timestamp of any read and write operations processed on x, respectively. Timestamp ordering guaran-

tees the serializability of the executing transactions by producing all conflicting operations in timestamp order. The basic timestamp ordering mechanism processes read and write operations as follows.

When a read operation $R_i(x)$ by a transaction T_i is received, the timestamp of T_i is compared with the value TSW(x). If it is smaller, the operation is rejected. Otherwise, (i.e., if the timestamp of T_i is larger than TSW(x)), the operation is accepted. Similarly, when a write operation $W_i(x)$ is received, the timestamp of T_i is compared with the value Max[TSR(x),TSW(x)]. If it is smaller, the operation is rejected. Otherwise, it is accepted.

When an operation is rejected, the transaction which issued the operation must be aborted. Aborting a transaction will involve undoing all its previous steps which have been executed and restarting the transaction from the beginning. Abortion is the main drawback of the timestamp ordering mechanism. If it occurs frequently, the performance can be degraded. Other timestamp-based mechanisms which necessitate lesser abortion than the basic mechanism described above have been proposed [Ber80, Ree78, Tho78].

The conservative timestamp ordering [Ber80] avoids abortion by delaying the operations instead. An operation is delayed until the concurrency control mechanism is sure that accepting the operation will cause no future operations to be rejected. Although this mechanism necessitates lesser abortion than the basic timestamp ordering, it requires more delay to process the executing transactions. Variations on the conservative timestamp ordering are discussed in [Ber80].

Thomas [Tho78] has improved the performance of the basic timestamp ordering mechanism by introducing a rule (called Thomas write rule) for processing write operations. According to this rule, a write operation will only be rejected when it

invalidates a previous read operation. This thesis will only consider the basic timestamp ordering mechanism described previously.

Timestamp ordering mechanism has the advantage that it is deadlock free. This makes the implementation of the mechanism simple in centralized as well as in distributed databases. Moreover, timestamp ordering mechanism achieves a higher level of concurrency than two-phase locking. This is because transactions never wait. However, the basic timestamp ordering mechanism tends to reject many operations (that arrive late).

The idea of utilizing timestamps in a database system has also been used for several other purposes. For example, in some systems they are used to assign priorities to the executing transactions to prevent the deadlock problem (see [Ros78]). In [Ste76] timestamps are used to solve the starvation problem.

Theorem 2.3: Every schedule produced by the timestamp ordering mechanism is serializable [Ber81].

2.4. Serialization graph mechanism

A serialization graph mechanism works by explicitly building a dependency graph and checking it for cycles. The mechanism updates its dependency graph whenever one of the following conditions takes place.

1. A new transaction T_i starts in the system.
2. A read or a write operation is received by the mechanism.
3. A transaction T_i is terminated.
4. A transaction T_i is aborted.

When a transaction T_i begins, a new node T_i representing the transaction T_i will be added to the dependency graph. When a read or write operation by a transaction T_i is received, a set of edges of the form $T_j \rightarrow T_i$ will be added to the graph, where T_j is a different transaction which issues previous operation that conflicts with the operation of T_i. After adding these edges, the mechanism will check the graph for cycles. If the graph is acyclic, the operation will be accepted. Otherwise (i.e., if the graph is cyclic) the operation will be rejected. In the latter case the mechanism will abort T_i and abort all other transactions that are dependent on transaction T_i (i.e., which read some entity written by T_i). The mechanism must also delete the node of every transaction which has been aborted to make its graph acyclic.

When all operations of the transaction T_i have been accepted, the mechanism will check to see whether the node T_i is a source node in the dependency graph. If it is, then T_i will be terminated and its node will be removed from the dependency graph. Otherwise, transaction T_i has to wait.

In order to reduce the number of transactions that are required to be backed-up after aborting the transaction T_i, the mechanism must distinguish between the arcs of the dependency graph. An arc (T_i,T_j) in the dependency graph is called a *primary arc* [Far82] if it is due to a conflict among read and write operations in transactions T_j and T_i, respectively. This primary arc implies that T_j is dependent on T_i, that is, aborting transaction T_i will necessitate aborting transaction T_j. An arc that is not primary arc is called a *secondary arc*. A secondary arc (T_i,T_k) does not necessitate aborting T_k.

Since the mechanism does not accept any operation which creates a cycle in the dependency graph, therefore every schedule produced by mechanism is serializable.

This serialization graph mechanism has several drawbacks. For example, a

transaction may have to wait even if all its read and write operations have been accepted by the mechanism. A transaction can only be deleted if it is a source node in the dependency graph. (A source node is a node without incoming arcs. Note that the reason for this delay is that even if all operations of the transaction have been accepted, the transaction may be aborted).

Another problem is that abortion can propagate when a transaction is aborted. Moreover, maintaining and updating the dependency graph is costly (especially in distributed systems). The previous problems make the implementation of the serialization graph mechanism more difficult in distributed systems.

Despite the above problems, some researchers have already shown that maintaining a dependency graph can increase the level of concurrency. In particular, Bayer et al. [Bay80] have introduced a locking mechanism which maintains a dependency graph, and they have also shown that their mechanism achieves a higher level of concurrency than two-phase locking. In the proposed mechanism, two versions of each entity are maintained. When the mechanism receives a read operation, it searches its dependency graph to find which version must be returned by this operation. The mechanism increases concurrency by assigning some read operations the old version of the entity.

2.5. Optimistic concurrency control mechanism

The optimistic concurrency control mechanism [Kun81] uses transaction backup as the main strategy for controlling concurrency. Unlike all mechanisms mentioned previously, the optimistic mechanism never delays or rejects an operation submitted by a transaction. Instead, the read and the write operations of each transaction are processed freely without updating the actual database. At the end of each transac-

tion the mechanism will decide whether or not the transaction can be terminated. If the decision is affirmative, the transaction updates will be stored permanently in the database. Otherwise, the transaction will be backed up.

Each update transaction (i.e., a transaction which updates at least one entity of the database) must consist of three phases; a read phase, a validation phase and a write phase. A read only transaction (i.e., a transaction which reads only entities of the database without updating any of them) does not need a write phase.

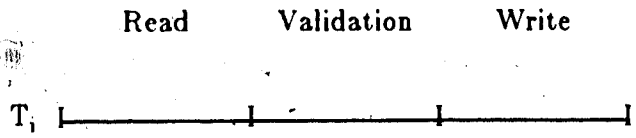

Figure 5 The three phases of transaction T_i.

All read and write operations of the transaction are contained in the read phase. During this phase, a write operation by a transaction T_i on an entity x creates a new value for x in the private work space area of T_i. This value of x will not be accessible to the other transactions. When all operations of the transaction have been processed, the transactions will enter its validation phase.

The purpose of the validation phase is to check whether the transaction can be

terminated without violating the integrity constraints of the database. If the validation phase succeeds, the transaction will enter its write phase. In the write phase all the local updates of the transaction will be made global updates by storing these updates permanently in the database). If the validation phase fails, the transaction will be backed up (possibly to the beginning.

The optimistic mechanism ensures that the updates of every executed transaction will not violate the integrity constraints by using a validation condition which guarantees the serializability of the executing transactions. This can be described as follows. Each transaction T_i will be assigned a unique timestamp $t_s(T_i)$ before it is validated. When a transaction T_j enters its validation phase, T_j must satisfy the following validation condition. For every transaction T_i with $t_s(T_i)<t_s(T_j)$, one of the following conditions must hold.

(1) T_i completes its write phase before T_j starts its read phase.

(2) The write set of T_i does not intersect the read set of T_j and T_i completes its write phase before T_j starts its write phase. (The read and the write sets of a transactions are the sets of entities read and written by the transaction, respectively).

(3) The write set of T_i does not intersect the read or the write set of T_j and T_i completes its read phase before T_j completes its read phase.

Condition (1) states that T_i actually completes before T_j starts (see Figure 6(a)). Condition (2) states that none of the entities updated by T_i is read by T_j and that T_i finishes writing its updates into the database before T_j starts writing. Thus, the updates of T_j will not be overwritten by the updates of T_i (see Figure 6(b)). Condition (3) is similar to condition (2) but does not require that T_i finish writing before T_j starts writing. It simply requires that the updates of T_i do not affect the read phase or the write phase of T_j (see Figure 6(c)). (Note that T_j cannot affect the read phase of

T_i by the last part of the condition).

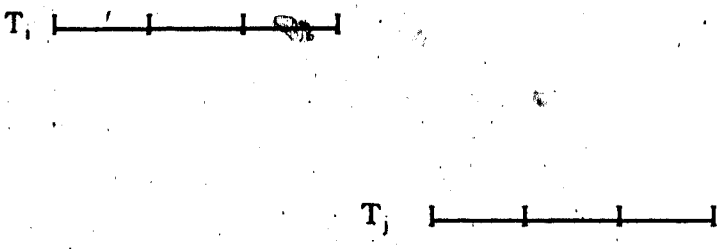

Fig. 6(a) Condition (1).

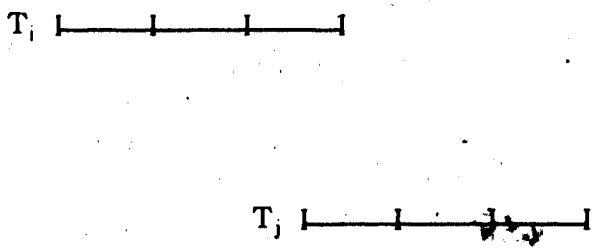

Fig. 6(b) Condition (2).

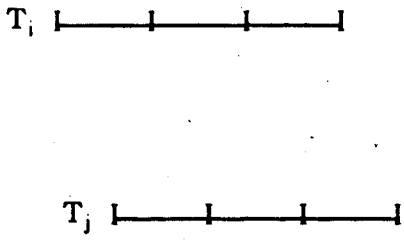

Fig. 6(c) Condition (3).

Fig. 6. The conditions for validations.

Since every executed transaction has to satisfy the validation condition, it is not difficult to see that the overall execution of the set of transactions is equivalent to a serial execution in which transaction T_i comes before transaction T_j whenever $t_s(T_i) < t_s(T_j)$.

The optimistic mechanism has the advantage that it allows a high level of concurrency. Kung, et al. [Kun81] have already shown that when transaction conflict is very rare, the optimistic mechanism will perform better than locking. However, the performance of the optimistic mechanism has not been examined for the general case in which transaction conflict may not be very rare.

The optimistic mechanism has several drawbacks. In order to validate a transaction, the optimistic mechanism has to store the read and the write sets of several other terminated transactions. This obviously increases the storage cost and makes the implementation of the algorithm more difficult in distributed systems.

Another problem is "starvation". Consider a situation in which the validation phase of a long transaction fails. Then, the transaction will be backed up and restarted again. Since the transaction is long, it is possible that the validation phase of the transaction will fail repeatedly. The following solution has been proposed by Kung, et al. [Kun81]. Each transaction will be assigned a counter which keeps track of the number of times the validation phase for the transaction will fail. If the validation phase has failed a specified number of times, then the starving transaction will be restarted and executed to completion by locking all entities accessed by the transaction.

Although the above solution resolves the starvation problem, it is still possible that a long transaction may be repeated several times before it is terminated. (The reader interested for more solutions to the starvation problem may refer to [Ste76]).

CHAPTER 3

THE GENERAL CONCURRENCY CONTROL ALGORITHM

This chapter describes the proposed concurrency control algorithm and proves that it works correctly, i.e., produces only serializable schedules.

The proposed concurrency control algorithm is more general than any previously introduced mechanism. This is because it contains as many special cases as the degree of multiprogramming of the system (This statement will be proven later). In particular, two-phase locking and timestamp ordering, which represent the most practical concurrency control mechanisms, are special cases of the proposed mechanism.

The general concurrency control mechanism collects the executing transactions in a set of classes, which will be called *strict classes*. The set of all transactions belonging to the same strict class are characterized by a unique *global timestamp* (i.e., number). Furthermore, each of these transactions is identified by a different *local timestamp*. The method used for assigning timestamps in our mechanism is valid, that is, the method satisfies the validation condition mentioned in [Ros78]. According to this condition, a method used for generating timestamps is said to be valid, if timestamps are generated in a monotonically increasing sequence. This sequence of timestamps must reflect the starting time of the transactions.

3.1. Transaction timestamps

Each transaction T_i is assigned a unique timestamp $t_s(T_i)$, when it starts. The timestamp $t_s(T_i)$ is a pair $(t_g(T_i),t_l(T_i))$, where $t_g(T_i)$ and $t_l(T_i)$ are called the global

and the local timestamps of T_i, respectively. Several transactions can have the same global timestamp simultaneously during execution. Each of these transactions must have a different local timestamp. The meaning of the local and the global timestamps will become apparent later when we describe how the mechanism processes read and write operations.

The method used for assigning timestamps assumes that there is some integer L, which we will call the strictness level, representing the maximum number of transactions which can have the same global timestamp simultaneously during execution. That is, at any time during execution, the number of transactions which can have the same global timestamp cannot exceed the value of L. This chapter assumes an arbitrary value for L. Moreover, the value of L is assumed to be fixed during execution. Choosing the value of L and its impact on the performance of the concurrency control algorithm will be discussed later.

The method used for assigning timestamps utilizes four counters C_1, C_2, C_3 and C_4. The current value of each counter C_i is denoted $V(C_i)$. C_1 and C_2 are used to generate the global and the local timestamps, respectively. C_3 is used to keep track of the number of transactions which have the current global timestamp $V(C_1)$. $V(C_3)$ cannot exceed the value of L. C_4 is used to keep track of the number of all executing transactions. $V(C_4)$ must always be $\leq M$, where M is some integer representing the multiprogramming level of the system. It is assumed that if $V(C_4) = M$, then no other transaction can start execution until one of the currently executing transactions is terminated or aborted. Timestamps are assigned as described below. Initially, all counters contain the value 0.

3.1.1. Starting or restarting a transaction

When a new transaction T_i arrives in the system (or when T_i is restarted) the values of $t_g(T_i)$ and $t_l(T_i)$ are assigned as follows.

1. If $V(C_4) = M$, then T_i has to wait until one of the currently executing transactions is terminated or aborted. Otherwise, $V(C_4) = V(C_4)+1$.
2. If $V(C_3) < L$, then $V(C_3) = V(C_3)+1$ and $V(C_2) = V(C_2)+1$. Otherwise, $V(C_3) = V(C_2) = 1$ and $V(C_1) = V(C_1)+1$.
3. $t_g(T_i) = V(C_1)$ and $t_l(T_i) = V(C_2)$.

Thus, the transaction T_i will be assigned the current global timestamp recorded in the counter C_1 as long as there are less than L executing transactions with the current global timestamp $V(C_1)$ and the total number of executing transactions is less than M. Otherwise, i.e., if there are L transactions with the current global timestamp $V(C_1)$, the transaction T_i will be assigned a new global timestamp (after incrementing the counter C_1).

3.1.2. Terminating or aborting a transaction

When a transaction T_i is terminated or aborted, the values of $V(C_3)$ and $V(C_4)$ will be modified as follows.

1. If $t_g(T_i) = V(C_1)$ at the time of termination or abortion, then $V(C_3) = V(C_3)-1$.
2. $V(C_4) = V(C_4)-1$.

Thus, when the transaction T_i is terminated (or aborted), the value of the counter C_4 must be decremented to reflect the correct number of executing transactions. Moreover, if the value $V(C_1)$ equals to the global timestamp of T_i, the counter C_3 will also be decremented by 1.

3.2. The values associated with the entities

For each entity x, the concurrency control algorithm maintains the following values: GTSW(x), LTSW(x), GTSR(x) and LTSR(x). The mechanism uses these values to decide whether a read or a write operation should be accepted, rejected, or delayed (as described later). These values are described as follows.

(i) GTSW(x) and LTSW(x)

GTSW(x) records the largest global timestamp of any transaction which has written x. This value is recorded when the write operation of the transaction is accepted. The local timestamp of the transaction will be recorded, at the same time, in LTSW(x). If the transaction is terminated (or aborted) and its global and local timestamps are still recorded in GTSW(x) and LTSW(x), respectively, its local timestamp will be deleted from LTSW(x).

(ii) GTSR(x) and LTSR(x)

GTSR(x) records the largest global timestamp of any transaction which read x. This value is recorded when the read operation of the transaction is accepted. The local timestamp of the transaction is recorded, at the same time, in LTSR(x). In general, several transactions with the same or different global timestamp can read x simultaneously. LTSR(x) records the local timestamp of every transaction whose global timestamp is recorded in GTSR(x) and read x. If a transaction is terminated or aborted and its global and local timestamps are recorded in GTSR(x) and LTSR(x), respectively, its local timestamp is deleted from LTSR(x).

Initially, i.e., before starting execution, LTSW(x) and LTSR(x) are empty and GTSW(x) and GTSR(x) record the initial value of the counter C_1.

3.3. Processing read and write operations

This section describes how the concurrency control algorithm processes read and write operations, and proves that the concurrency control algorithm produces only serializable schedules.

3.3.1. Processing read operations

When the concurrency control algorithm receives a read operation $R_i(x)$, one of the following conditions must be true.

(1) $t_g(T_i) < GTSW(x)$. This condition means that x has been written by a transaction which has a larger global timestamp than T_i. In this case $R_i(x)$ is rejected.

(2) $t_g(T_i) > GTSW(x)$. This condition means that x has not been written by any transaction whose global timestamp is larger than or equal to the timestamp of T_i. In this case $R_i(x)$ is accepted.

(3) $t_g(T_i) = GTSW(x)$. This condition means that x has been written by one (or more) transaction which has the same global timestamp as T_i. In this case, one of the following conditions must be true.

(a) LTSW(x) is empty. This means that any transaction with the same global timestamp as T_i and has written x has been terminated. In this case $R_i(x)$ is accepted.

(b) LTSW(x) is not empty. This means that x has been written by another transaction which has the same global timestamp as T_i and this transaction is still in the system. In this case $R_i(x)$ is delayed.

When the mechanism delays a read or write operation by a transaction T_i because it conflicts with a previously accepted operation by a different transaction T_j, T_i has to wait for T_j until it is terminated or aborted (and, therefore, the conflict no longer exists). If another operation with a larger global timestamp than the waiting operation and conflicting with it has been accepted, the waiting operation will be rejected (i.e., the transaction T_i which issued the operation will be aborted).

3.3.2. Processing write operations

When the concurrency control mechanism receives a write operation $W_i(x)$, one of the following conditions must be true.

(1) $t_g(T_i) <$ Maximum[GTSR(x),GTSW(x)]. This condition means that x has been read or written by a transaction which has a larger global timestamp than T_i. In this case, $W_i(x)$ is rejected.

(2) $t_g(T_i) >$ Maximum[GTSR(x),GTSW(x)]. This condition means that x has not been read or written by any other transaction whose global timestamp is larger than or equal to the global timestamp of T_i. In this case, $W_i(x)$ is accepted.

(3) $t_g(T_i) =$ Maximum[GTSR(x),GTSW(x)]. This condition means that x has been read or written by one (or more) transaction which has the same global timestamp as T_i. In this case, one of the following conditions is true.

(a) GTSW(x) > GTSR(x). This means that x has not been read by any transaction which has the same global timestamp as T_i. In this case, the mechanism examines LTSW(x). If it is empty, then any transaction which wrote x has been terminated. Therefore, $W_i(x)$ is accepted. Otherwise (i.e., if LTSW(x) is

not empty) $W_i(x)$ is delayed.

(b) GTSW(x) < GTSR(x). This means that x has not been written by any transaction which has the same global timestamp as T_i. In this case the concurrency control mechanism examines LTSR(x). If LTSR(x) is empty or contains only the local timestamp of T_i, then any other transaction which read x has been terminated. Therefore, $W_i(x)$ is accepted. Otherwise, $W_i(x)$ is delayed.

(c) GTSW(x) = GTSR(x). In this case the concurrency control mechanism examines LTSW(x) and LTSR(x). If LTSW(x) is empty and LTSR(x) is empty or contains only the local timestamp of Ti, then $W_i(x)$ is accepted. Otherwise, $W_i(x)$ is delayed.

3.3.3. The correctness of the mechanism

This section proves that the mechanism works correctly, i.e., produces only serializable schedules. The given proof is based on the (serializability) theorem given in Section 3.1.

Theorem 3.1: Every schedule produced by the proposed mechanism is serializable.

Proof: Let H be a schedule produced by the mechanism and let T_i and T_j be arbitrary transactions in H such that there is an operation in T_i which accesses the entity x, and which conflicts with and precedes an operation in T_j. Suppose first that the conflicting operation of T_j is a read operation on x. According to the procedures described previously for processing the (read and write) operations, one of the following conditions must be true at the time the conflicting operation of T_j is received:

(a) $t_g(T_j) > GTSW(x) \geq t_g(T_i)$.

(b) $t_g(T_j) = GTSW(x) \geq t_g(T_i)$ and LTSW(x) is empty.

The above conditions imply that either $t_g(T_i) < t_g(T_j)$, or $t_g(T_i) = t_g(T_j)$ and T_i terminates before T_j. It is not difficult to show that this is also true even if the conflicting operation of T_j is a write operation. This actually implies that for each arc $T_i \rightarrow T_j$ in DG(H), either $t_g(T_i) < t_g(T_j)$ or $t_g(T_i) = t_g(T_j)$ and T_i terminates before T_j. Let T_k and T_l be two arbitrary nodes in DG(H) such that there is a path (of length greater than zero) from T_k to T_l. Then, by transitivity, either $t_g(T_k) < t_g(T_l)$ or $t_g(T_k) = t_g(T_l)$ and T_k terminates before T_l.

Suppose first that $t_g(T_k) < t_g(T_l)$. Another path from T_l to T_k will lead to the contradiction that $t_g(T_l) < t_g(T_k)$. Similarly, suppose that $t_g(T_k) = t_g(T_l)$ and T_k terminates before T_l. Another path from T_l to T_k will lead to the contradiction that T_l terminates before T_k. In either case, another path from T_l to T_k cannot exist because it leads to a contradiction. Therefore, DG(H) is acyclic, i.e., H is serializable. □

3.4. Special cases of the proposed mechanism

This section describes the special cases of the proposed concurrency control algorithm and shows that two-phase locking and timestamp ordering represent the two end points of a series of concurrency control algorithms. Each of them is a special case of the proposed algorithm

The set of transactions executing at any time under the proposed algorithm can be partitioned into a set of disjoint classes; each class containing the set of transactions with the same global timestamp. This set of classes changes dynamically during execution (i.e., when a transaction is started, terminated or aborted). These classes

will be called strict classes. The value chosen for the strictness level determines the maximum number of transactions which can have the same global timestamp at any time during execution (i.e., which can belong to the same strict class).

3.4.1. The two-phase locking special case

Consider a special case of the proposed algorithm in which $L \geq M$, where L and M are as defined previously. According to the procedure described in Section 3.1 for assigning timestamps, when a new transaction starts in the system, the counter C_1 will only be incremented if $V(C_3) = L$ (i.e., if there are L executing transactions in the system with the current global timestamp $V(C_1)$). Since $L \geq M$ and the total number of executing transactions cannot exceed M, the executing transactions will always have the same global timestamp. This global timestamp will equal to the initial value of the counter C_1. In this case, the number of strict classes at time during execution will equal 1.

In the procedure described in the previously for processing read operations only condition (3) (i.e., $t_g(T_i) = GTSW(x)$) can be satisfied. In this case, a read operation $R_i(x)$ will be accepted if LTSW(x) is empty. Otherwise, $R_i(x)$ will be delayed. That is, $R_i(x)$ will only be delayed if it conflicts with a previous write operation issued by a transaction which has not been terminated.

Similarly, in the procedure described for processing write operations only condition (3) (i.e., $t_g(T_i) = Maximum[GTSW(x),GTSR(x)]$) can be satisfied. In this case, a write operation $W_i(x)$ will be accepted if LTSW(x) is empty and LTSR(x) is empty or contains the local timestamp of T_i. This is because the values of GTSR(x) and GTSW(x) will always be the same. Otherwise, $W_i(x)$ will be delayed. That is, $W_i(x)$ will be delayed if it conflicts with a previous read or write operation issued by a tran-

saction which has not been terminated.

It is not difficult to see that in this particular case the processing of read and write operations is performed as in the two-phase locking mechanism. In this special case the conflicting operations cannot be processed concurrently. Instead, the conflicts will be resolved by delaying the operations.

3.4.2. The timestamp ordering special case

Consider another special case in which L = 1. In the procedure described for assigning timestamps, when a new transaction T_i starts, the value of the counter C_1 will only be incremented if $V(C_3) = L$ (i.e., if there are L executing transactions with the current global timestamp). Since L = 1, then the executing transactions will always have different global timestamps, i.e., the number of strict classes at any time during execution will equal to the number of transactions executing at that time.

In the procedure described for processing read operations only condition (1) (i.e., $t_g(T_i) < GTSW(x)$) or condition (2) (i.e., $t_g(T_i) > GTSW(x)$) can be satisfied. In this case, a read operation $R_i(x)$ will be accepted if condition (2) is satisfied, otherwise $R_i(x)$ will be rejected.

Similarly, in the procedure described for processing write operations condition (3) (i.e., $t_g(T_i) = Maximum[GTSW(x),GTSR(x)]$) can only be satisfied if there is a read operation $R_i(x)$ which has been accepted by the mechanism. In this case, a write operation $W_i(x)$ will be rejected if condition (1) is satisfied (i.e., if $t_g(T_i) < Maximum[GTSW(x),GTSR(x)]$). Otherwise (i.e., if condition (2) or condition (3) is satisfied) $W_i(x)$ will be accepted.

It is not difficult to see that in this particular case the processing of read and

write operations will be performed as in the timestamp ordering mechanism. Unlike the previous special case, the conflicting operations will never be delayed. Instead, each operation is either accepted or delayed, depending on the condition satisfied when the operation is received.

3.4.3. Other special cases

Several other special cases of the proposed mechanism arise for L = 2,3,..,M-1. Processing read and write operations in each of these cases is performed by the concurrency control mechanism as if it is a combination of the two-phase locking and the timestamp ordering mechanism. An operation will be processed relative to another operation which has the same global timestamp as if the concurrency control is two-phase locking and relative to another operation which has different global timestamp as if the concurrency control is timestamp ordering. The following lemma gives the minimum and the maximum number of strict classes at any time during execution for any value of L, $1 \leq L \leq M$.

Lemma 3.1: Let C refer to the number of strict classes at any given time during execution and let L, M and $V(C_4)$ be as defined previously. Then, $\lceil V(C_4)/L \rceil \leq C \leq$ Minimum$[V(C_4), M-L+1]$.

Proof: The proof of the above Lemma is an immediate consequence of the method described previously for assigning timestamps. The minimum number of strict classes corresponds to an instant in which at most one class has less than L transactions. The maximum number of strict classes corresponds an instant in which M-L transactions (or every transaction if $V(C_4) \leq M-L$) will belong to M-L ($V(C_4)$) different classes and the remaining transactions will belong to another strict class. □

Thus, two-phase locking and timestamp ordering represent the two end points of a series of concurrency control algorithms. Each results from choosing a different value for the strictness level and is considered to be a special case of the general concurrency control mechanism described previously (see Figure 7). Although we have previously assumed that the value of L is fixed during execution, this is not necessary. In fact, the value of L can change from time to time (during execution) without affecting the correctness of the proposed mechanism.

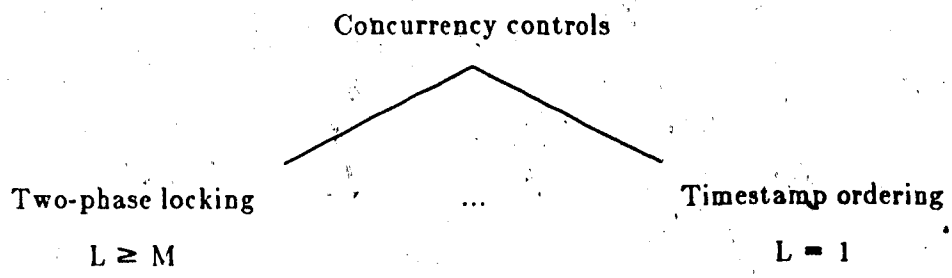

Figure 7 Special cases of the general mechanism.

3.5. The generality feature of the proposed mecha[illegible]

Integrating a series of concurrency control mechanisms, which include two-phase locking and timestamp ordering, within one general mechanism has several advantages. The most obvious advantage is that the general mechanism provides several special cases; any of them can be selected in advance and can even be changed during execution.

Most implementations of database management systems that are currently in use support only unique concurrency control mechanism, such as two-phase locking, or timestamp ordering. This mechanism is chosen when the database system is designed. If the database designers decide at a later time to replace the implemented concurrency control mechanism by a different mechanism, a lot of work needs to be done in order to implement the new mechanism. This problem is completely eliminated in the proposed approach. In order to change the currently used mechanism, we only need to change the current value of L.

The two critical factors in evaluating the performance of a concurrency control mechanism are the level of concurrency allowed by the mechanism and the likelihood of transaction conflict. The first factor depends on the mechanism used , while the second factor is application dependent, i.e., it varies from application to application. Two-phase locking and timestamp ordering mechanisms represent the two extremes in terms of allowing concurrency in the proposed mechanism. That is, two-phase locking allows the lowest level of concurrency, while timestamp ordering allows the highest level of concurrency. The other special cases allow intermediate levels of concurrency.

When the likelihood of transaction conflict is very low, we believe that the timestamp ordering mechanism will perform better. Under such conditions it is unlikely that many conflicting operations will arrive out of order. In this case delaying

the operations will not be necessary. Similarly, when the likelihood of transaction conflict is very high, we believe that two-phase locking mechanism will perform better. In this case many operations may arrive out of order, and consequently many transactions may be aborted if the operations are not delayed. Between these two extremes of transaction conflict, the intermediate special cases of the proposed mechanism are expected to perform better than two-phase locking and timestamp ordering.

CHAPTER 4

RELATED PROBLEMS

This chapter discusses some problems that are associated with the mechanism presented in in the previous chapter and outlines some solutions to these problems.

4.1. Deadlock

Deadlock can only occur if the strictness level is greater than 1 (i.e., for any mechanism except timestamp). Moreover, when it occurs it only involves transactions which have the same global timestamp. This is because a transaction never waits for any other transaction which has a different global timestamp.

The deadlock problem can be resolved by detecting deadlock and aborting one (or more) transaction [Obe82]. Detecting deadlock requires maintaining a wait-for graph similar to the one described in Chapter 3. Similarly, the deadlock problem can also be resolved by preventing deadlock from occurring (as described in [Ros78, Ryp79]).

4.2. Assigning timestamps in distributed systems

Assigning timestamps in a centralized system is simple and straightforward. This is due to the fact that the values of all counters (used for assigning timestamps) and the parameters L and M will be available at the same (centralized) site. Assigning timestamps in a distributed system will depend on the way in which the concurrency control will be implemented. The main approaches for implementing a concurrency

control mechanism in a distributed system are the *centralized approach* and the *distributed approach.*

(1) The centralised approach

In this approach the concurrency control mechanism resides on one (designated) site in the distributed system. This master site coordinates the activities of all transactions in the distributed system. The read and the write operations of all transactions in the system will be directed to the master site and the decision of accepting, delaying, or rejecting the operations will also be taken at that site. Such a design of a distributed system is actually a centralized control whose database happens to be distributed.

Assigning timestamps in this case is simple and similar to the centralized case. The values of all counters (used for assigning timestamps) and parameters L and M will be stored at the master site. The value of L will be chosen (and can only be modified) by the master site. The value of M in this case represents the maximum number of transactions that can be executed concurrently at all sites in the system.

When a transaction T_k is initiated at any site, the local and the global timestamps of T_k will be computed at the master site using the same procedure described in the previous chapter. Similarly, when T_k is terminated (or aborted) the counters C_3 and C_4 will also be modified as described in previously.

(2) The distributed approach

In this approach the concurrency control mechanism resides on every site in the distributed system. The local mechanism at each site S_i will be responsible for resolving all the conflicts at S_i, that is, accepting, delaying, or rejecting an operation

on some entity x stored at S_i will be decided by the local mechanism independently of all other sites in the system. When communication cost is high, such a design of a distributed system becomes more attractive than the centralized design described previously.

In this case, assigning timestamps can be done locally as follows. Let M_i represents the maximum number of transactions that can be executed concurrently at the site S_i. Similarly, let M represents the maximum number of transactions that can be executed concurrently at all sites (that is, $M = \sum M_i$ for all sites).

The values of the counters C_1, C_2, and C_3 and the parameters L and M will be stored at a particular site in the distributed system. This site must be known to all the other sites. (In order to increase the reliability of the distributed system, these values may also be stored at several other sites in the system. If one of these sites fail, these values can be found on another site that is still operating). The counter C_4 in this case will be replaced with a set of counters, one counter C_{4i} for each site S_i. The counter C_{4i} will be stored at the site S_i. Moreover, C_{4i} will only be accessible to the local concurrency control residing at S_i. $V(C_{4i})$ will represent the number of transactions currently executing at S_i. (Note that some transactions may be executed at more than one site).

When a transaction T_k starts at S_i, the local concurrency control mechanism at S_i assigns new (local and global) timestamps to T_k as follows. First, the mechanism must obtain exclusive locks on C_1, C_2, C_3, and L. Then, the local and the global timestamps of T_k will be computed using the same procedure described in the previous chapter after replacing M and $V(C_4)$ in this procedure with M_i and $V(C_{4i})$, respectively.

Similarly, when T_k is terminated (or aborted), the mechanism must obtain

exclusive locks on C_1 and C_3. Then, the values of the counters will be modified using the procedure described in the previous chapter after replacing $V(C_4)$ with $V(C_{4i})$ in this procedure.

The initial value of L in this case must be accepted to all sites. If one of the sites wants to change the current value of L, it must first notify all other sites. If the response of all sites is "Yes", i.e., if all sites agree on the new value, then this site can modify the value of L.

CHAPTER 5

ALLOWING NON-SERIALIZABLE SCHEDULES

In order to allow a set of transactions to access (i.e., retrieve and update) the entities of a database concurrently, a concurrency control mechanism is needed to resolve the conflicts that might arise among transactions and to ensure that the overall effect of their execution is correct (i.e., transforms a consistent database state into a new consistent state). Several concurrency control mechanisms that allow only serializable schedules have been introduced. Designing these mechanisms has proven to be simple. However, very little work has been done on designing concurrency control mechanisms which allow non-serializable schedules.

The main motivation behind allowing non-serializable schedules (which do not violate consistency) is improving performance. Several researchers [Gar82, Gar83, Gra76, Lyn83] have already noticed that in some applications, allowing only serializable schedules may lead to reducing the level of concurrency and increasing transaction response time. Allowing non-serializable schedules, on the other hand, increases concurrency and improves performance. These facts are well known [Kun79, Lyn83].

There are two main difficulties associated with the concept of allowing non-serializable schedules. The first is that designing concurrency control mechanisms that accept non-serializable schedules is not simple. The second problem is that specifying these schedules in a way which is suitable for use by a concurrency control mechanism may be difficult.

One simple way to improve performance and to allow non-serializable schedules is to extend the basic serializability theory to allow several degrees (or levels) of consistency [Gar82, Gra76]. That is, serializability can be considered the strongest degree of consistency and other (weaker) degrees of consistency, that permit non-serializable schedules, can be allowed. We illustrate this idea by the following example.

Example 5.1:

Consider the schedule H described below. H is not serializable (i.e., H is not equivalent to a serial schedule of the transactions). However, all update-transactions of H appear in a serial order, i.e., do not overlap. (Note that an update-transaction is a transaction which updates some entities of the database). Clearly, the execution of the schedule H cannot violate consistency, even though H is non-serializable schedule.

$H = W_2(x)R_1(x)W_3(x)W_3(y)R_1(y)$.

The basic idea of allowing non-serializable schedules was first introduced in [Gra76]. Several concurrency control mechanisms that make use of this idea have also been proposed in [Gar82].

In this thesis, we are interested in other approaches which make use of the semantic knowledge available about the transactions in allowing non-serializable schedules. This idea has been recently investigated by Garcia-Molina [Gar83] and an interesting preliminary work has been proposed for designing a concurrency control mechanism which accepts a certain class of non-serializable schedules called "semantically consistent" schedules. The details of this class of schedules, as well as the locking mechanism proposed by Garcia-Molina [Gar83], will be reviewed in Appendix I.

The basic idea behind the locking mechanism proposed by Garcia-Molina

[Gar83] is dividing the executing transactions into a set of disjoint classes such that the transactions which belong to the same class are "compatible", i.e., can interleave arbitrarily, and the transactions which belong to different classes are "incompatible", i.e., cannot interleave at all.

The compatibility concept defined in [Gar83] has been refined further by Lynch [Lyn83]. In this formalization, several levels of compatibility among transactions are defined. At each level, the transactions are grouped into a set of classes called "nested classes". The main idea is that the transactions which belong to the same class at a high level can have many relevant breakpoints (i.e., can interleave a great deal), while the transactions that belong to the same class at a low level might have fewer relative breakpoints (i.e., cannot interleave very much). Although the class of correctable schedules defined in [Lyn83] can be recognized in polynomial time, it is still an open problem whether there is an efficient on-line scheduler for this class.

The remaining chapters of this thesis examine the concurrency control problem when semantic information is available about the transactions. The formalization presented here is more general than those used in Garcia-Molina [Gar83] and Lynch [Lyn83]. This is because there is no assumption regarding any structural rule for the interleaving among the transactions. We shall describe a concurrency control mechanism which improves performance by allowing serializable and non-serializable schedules to be produced. The proposed mechanism assumes fewer restrictions on the interleaving among the transactions than those assumed in [Gar83], and therefore achieves a higher level of concurrency. The proposed mechanism also avoids other problems that are associated with the locking mechanism of Garcia-Molina regarding transaction failure. In the "semantically consistent transaction classes" formalization, the only way to handle transaction failure is to run another compensating transaction. In some cases, writing a compensating transaction may not be simple, and may itself

be the source of inconsistencies. Although the concept of compensating transactions can also be used in our mechanism, another approach is also presented which avoids the problems of the compensating transactions.

CHAPTER 6

THE MODEL

This chapter describes the database system model used in formalizing the second topic in this thesis, and defines a new class of schedules called relatively consistent (RC) schedules.

A *database* is a set D of entities. The entities of D are named x, y, z, etc. The values of these entities can be read and modified by a set of application programs, which will be called *transactions*. It is assumed that each of these transactions is a correct computation which preserves the integrity constraints of the database. These transactions are divided into a set of *types*, based on their semantic knowledge and the actions they perform. The type of a transaction T_i is denoted $t(T_i)$ and the set of all types of transactions is denoted TP. The elements of the set TP are named TP1, TP2, TP3, etc. (For example, in a banking system the elements of the set TP might represent transfer transactions, audit transactions, creditor transactions, etc).

The idea of classifying the transactions into a collection of types has also been used in [Ber80a, Gar83, Lyn83]. The classification used in [Gar83, Lyn83] is similar to ours, that is, this classification is based on the semantic information of the transactions, while the classification proposed in [Ber80a] is based on the syntactic information of the transactions.

Each transaction T_i is modeled as a sequence of steps and a termination command $TR(T_i)$ at the end of this sequence, that is, $T_i = S_{i1}S_{i2}S_{i3}...S_{im}TR(T_i)$. Each step S_{ij} in T_i consists of a sequence of atomic actions and a breakpoint at the end of these actions. An action is either a read operation, denoted $R_i(x)$, or a write operation,

denoted $W_i(x)$, on some entity x. Each step must have at least one operation, and an entity x can be accessed in at most one step in the same transaction. The breakpoint of the step S_{ij} represents the termination point of S_{ij}. This breakpoint will be denoted B_{ij}. The termination command $TR(T_i)$ represents the termination point of (all steps of) T_i. An example of a transaction is shown below.

$T_1 = R_1(x)W_1(x)B_{11}R_1(y)W_1(y)B_{12}TR(T_1)$.

The steps of T_1 are

$S_{11} = R_1(x)W_1(x)B_{11}$.

$S_{12} = R_1(y)W_1(y)B_{12}$.

Associated with each breakpoint B_{ij} is a set $t'(B_{ij})$, which denotes the types of all transactions that are allowed to interleave at B_{ij} ($t'(B_{ij}) \subseteq TP$). No restriction will be assumed on the breakpoints of a transaction (until Chapter 10).

An interleaved sequence of the operations and the breakpoints of a set of transactions is called a *schedule*. Consider, for example, the three transactions described below.

$T_1 = R_1(x)B_{11}R_1(y)B_{12}TR(T_1)$.

$T_2 = W_2(x)B_{21}W_2(y)R_2(z)B_{22}TR(T_2)$.

$T_3 = W_3(z)B_{31}W_3(y)B_{32}TR(T_3)$.

An example of a schedule H of $\{T_1, T_2, T_3\}$ is

$H = R_1(x)B_{11}W_2(x)B_{21}W_2(y)R_2(z)B_{22}TR(T_2)W_3(z)B_{31}W_3(y)B_{32}TR(T_3)R_1(y)B_{12}TR(T_1)$.

The order of operations and breakpoints in H (and in the transactions) increases from left to right. We require the order of operations (and breakpoints) of the same transaction to be preserved in H. We say that a step S_{ij} precedes a step S_{kl}, if the breakpoint B_{ij} precedes the first operation of S_{kl} in H. A read operation $R_i(x)$ returns the value of x written by the write operation $W_j(x)$, if $W_j(x)$ precedes $R_i(x)$ and no other write operation on x appears between $R_i(x)$ and $W_j(x)$. If no write operation

on x precedes $R_i(x)$ in H, $R_i(x)$ returns the initial value of x, i.e., the value which exists in the database before executing the schedule H.

We shall use the symbols "$<$" and "$\leq$" to denote the precedence relations among the operations and the breakpoints, where "$<$" means "precedes" and "$\leq$" means "precedes or the same as". We shall also use the symbol "S_{ij}" to indicate "the series of operations and breakpoint of S_{ij}."

Definition 6.1: Two schedules H_1 and H_2 for the same set of transactions are said to be *equivalent* if the following two conditions are satisfied.

(1) For each read operation $R_i(x)$, either $R_i(x)$ returns the value of x written by the same write operation in H_1 and H_2, or $R_i(x)$ is not preceded by any write operation on x in H_1 and H_2.

(2) For each entity x updated in both schedules, the last write operation on x must be the same in H_1 and H_2.

Condition (1) ensures that each read operation in H_1 return the same value returned by the corresponding read operation in H_2. Condition (2) ensures that the final value written for each entity x is the same in both schedules. These two conditions guarantee that each transaction will see the same database in both schedules.

Note that our definition of schedule equivalence is identical to the one used for serializability [Esw76]. This is to be expected, because the insertion of breakpoints in a transaction cannot change the meaning of equivalence. For example, the following two schedules are equivalent.

$H_1 = R_1(x)B_{11}W_2(y)R_2(z)B_{21}W_2(x)B_{22}TR(T_2)W_3(z)B_{31}W_3(y)B_{32}TR(T_3)R_1(y)$
$B_{12}TR(T_1)R_4(z)B_{41}TR(T_4)$.

$H_2 = W_2(y)R_2(z)B_{21}R_1(x)B_{11}W_3(z)B_{31}W_2(x)B_{22}TR(T_2)W_3(y)B_{32}TR(T_3)R_4(z)$
$B_{41}TR(T_4)R_1(y)B_{12}TR(T_1)$.

Definition 6.2: A schedule H is said to be *correct* if the following two conditions are satisfied.

(1) H is a stepwise serial schedule [Gar83], i.e., the operations and the breakpoint of each step in every transaction appear without interleaving.

(2) For every pair of steps S_{ij} and S_{ij+1} (of the same transaction T_i) and for every different step S_{kl} appearing between S_{ij} and S_{ij+1} (if any), $t(T_k) \in t'(B_{ij})$.

Condition (1) prohibits interleaving inside the same step and condition (2) ensures that the interleaving among the steps is allowed by the system. These two conditions guarantee that the execution of a correct schedule cannot violate consistency.

Note that adding the restriction that the breakpoint of the last step of each transaction allows all types in TP to interleave cannot change the set of correct schedules. In order to make our analysis consistent with the implementation discussed in Chapter 10, such a restriction will not be added.

Note also that in our definition of the class of correct schedules, we have not assumed any structure on the way the transactions can interleave. This actually implies that the class of correct schedules generalizes the class of "semantically consistent schedules" defined in [Gar83] and the class of "multilevel atomic schedules" defined in [Lyn83]. According to the definition of these classes, the interleavings among transactions must obey a certain structure, that is, must satisfy a set rules which define the "compatibility" among transactions.

Definition 6.3: An operation in step S_{ij} is in *conflict* with another operation in step S_{kl}, where $T_i \neq T_k$, iff both operations access the same entity and one of them is a write operation. The steps S_{ij} and S_{kl} (and the transactions T_i and T_k) are also said to be conflicting or interacting steps (transactions).

Definition 6.4: For each step S_{ij} and transaction T_k we define two symbols, denoted $P(S_{ij},T_k)$ and $F(S_{ij},T_k)$, which represent the beginning and the end of a series of steps in T_i that must be viewed by T_k as an atomic action. (The significance of defining these symbols will become apparent later). $P(S_{ij},T_k)$ is defined as follows.

(1) If $j=1$, or $t(T_k) \notin t'(B_{ip})$ for any $B_{ip}<B_{ij}$ in T_i, then $P(S_{ij},T_k)=S_{i1}$. Otherwise,

(2) let $t(T_k) \in t'(B_{ip})$, where $B_{ip}<B_{ij}$ in T_i, and for each breakpoint B_{iq} appearing between B_{ip} and B_{ij} in T_i (i.e., $B_{ip}<B_{iq}<B_{ij}$), $t(T_k) \notin t'(B_{iq})$, then $P(S_{ij},T_k)=S_{ip+1}$.

Similarly, $F(S_{ij},T_k)$ is defined as follows.

(1) If S_{ij} is the last step in T_i, or $t(T_k) \notin t'(B_{ip})$ for any $B_{ij} \leq B_{ip}$ in T_i, then $F(S_{ij},T_k)$ equals to the last step in T_i. Otherwise,

(2) let $t(T_k) \in t'(B_{ip})$, where $B_{ij} \leq B_{ip}$ in T_i, and for each breakpoint B_{iq} appearing between B_{ij} and B_{ip} in T_i (if any), $t(T_k) \notin t'(B_{iq})$, then $F(S_{ij},T_k)=S_{ip}$.

Example 6.1:

Consider for example the following two transactions:

$T_1=R_1(x)W_1(x)B_{11}R_1(y)W_1(y)B_{12}R_1(z)W_1(z)B_{13}TR(T_1)$.

$T_2=R_2(z)R_2(y)B_{21}TR(T_2)$.

Assume the following: $t(T_2) \in t'(B_{11})$, $t(T_2) \notin t'(B_{12})$, and $t(T_2) \in t'(B_{13})$. Then, $P(S_{13},T_2)=S_{12}$, and $F(S_{13},T_2)=S_{13}$.

6.1. Relatively consistent schedules

This section defines a new class of schedules called relatively consistent (RC) schedules. This class contains serializable and non-serializable schedules.

6.1.1. Precedence graph

The *precedence graph* of a schedule H is a directed graph PG(H)=(S,A), where S is a set of nodes representing the set of all steps of the transactions in H and A is a set of arcs defined as follows.

(1) For each pair of consecutive steps, S_{ij} and S_{ij+1}, of the same transaction T_i, an arc (S_{ij},S_{ij+1}) exists in A.

(2) For each pair of steps, S_{ij} and S_{kl}, belonging to two different transactions, an arc $(F(S_{ij},T_k),P(S_{kl},T_i))$ exists in A if there is an operation in S_{ij} which conflicts with and precedes (in H) another operation in S_{kl}.

(3) The remaining arcs of PG(H) are added recursively according to the following condition. For each pair of steps, S_{ij} and S_{kl}, belonging to two different transactions such that there is a path from S_{ij} to S_{kl} in PG(H), an arc $(F(S_{ij},T_k),P(S_{kl},T_i))$ exists in A.

Intuitively, an arc satisfying condition (1) introduces precedence relation among two steps belonging to the same transaction. We will call such arc an *internal*-arc (or *I*-arc, for short). Similarly, an arc satisfying condition (2) or (3) introduces precedence relation among two steps belonging to different transactions. We will call the arcs satisfying conditions (2) and (3) *conflict*-arcs (*C*-arcs) and *transitive*-arcs (*T*-arcs), respectively.

The *C*-arcs represent the conflict relationships among the transactions, while the *T*-arcs are induced arcs, added to the precedence graph to enforce the view of atomicity among the interacting transactions. (Note that there is at most one arc between any pair of nodes in PG(H), i.e., some of the arcs may satisfy conditions (2) and (3)).

Example 6.1:

consider the schedule H described below.

$H = R_1(x)B_{11}W_2(x)B_{21}W_2(y)R_2(z)B_{22}TR(T_2)W_3(z)B_{31}W_3(y)B_{32}TR(T_3)R_4(z)$
$B_{41}TR(T_4)R_1(y)B_{12}TR(T_1)$.

Assume the following:

(a) $TP = \{TP1, TP2, TP3, TP4, TP5\}$.

(b) $t(T_1) = TP1$; $t(T_2) = TP4$; $t(T_3) = TP5$; $t(T_4) = TP3$.

(c) $t'(B_{11}) = \{TP4, TP5\}$; $t'(B_{21}) = \{TP2\}$; $t'(B_{31}) = \{TP3, TP4\}$.

$t'(B_{12}) = t'(B_{22}) = t'(B_{32}) = t'(B_{41}) = TP$.

The precedence graph PG(H) is constructed in Figure 8.

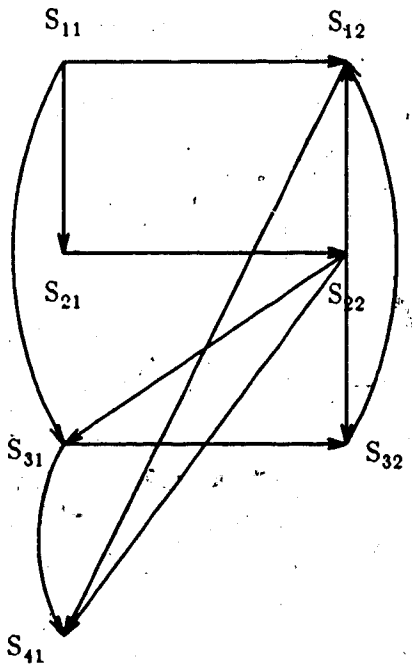

Figure 8 The precedence graph of H.

Definition 6.5: A schedule H is said to be relatively consistent (RC) schedule iff

(1) PG(H) is acyclic, and

(2) there is a topological sort for the nodes of PG(H) which yields a correct schedule, that is, a correct schedule H_1 can be formed by sorting the nodes of PG(H) topologically, replacing each node S_{ij} with the operations and breakpoint of S_{ij}, and adding the termination command $TR(T_i)$ at the end of each transaction T_i.

Example 6.2:

The schedule H of Example 6.1 is an RC schedule. This is because the nodes of PG(H) can be sorted to obtain the following correct schedule.

$H_1 = R_1(x)B_{11}W_2(x)B_{21}W_2(y)R_2(z)B_{22}TR(T_2)W_3(z)B_{31}W_3(y)B_{32}TR(T_3)R_1(y)$
$B_{12}TR(T_1)R_4(z)B_{41}TR(T_4)$.

The following Lemma proves that the acyclicity of the precedence graph is not a sufficient condition for finding a topological sort which yields a correct schedule. (In the implementation discussed in Chapter 10, further restrictions will be added on the interleavings among transactions. These restrictions will make the acyclicty of the dependency graph sufficient condition for finding a topological sort that yields a correct schedule).

Lemma 6.1: There is a schedule H such that PG(H) is acyclic and none of the topological sorts of PG(H) yield a correct schedule.

Proof: Consider, for example, the schedule H described below.

$H = W_4(p)B_{41}W_3(x)B_{31}R_1(y)R_1(z)B_{11}R_1(p)R_1(x)B_{12}TR(T_1)R_2(y)R_2(z)B_{21}$
$R_2(p)R_2(x)B_{22}TR(T_2)W_3(y)B_{32}TR(T_3)W_4(z)B_{42}TR(T_4)$

Assume the following:

(a) $TP = \{TP1, TP2, TP3\}$

(b) $t(T_1) = t(T_2) = TP1$; $t(T_3) = TP2$; $t(T_4) = TP3$.

(c) $t'(B_{11}) = t'(B_{21}) = \{TP2, TP3\}$; $t'(B_{31}) = t'(B_{41}) = \{TP1\}$;
$t'(B_{12}) = t'(B_{22}) = t'(B_{32}) = t'(B_{42}) = TP$.

The precedence graph of the above schedule is shown in Figure 9. It is not difficult to show that none of the topological sorts of PG(H) yield a correct schedule. This is because in any topological sort for PG(H), either the steps of the transactions T_1 and T_2 must interleave, or the steps of the transactions T_3 and T_4 must interleave. □

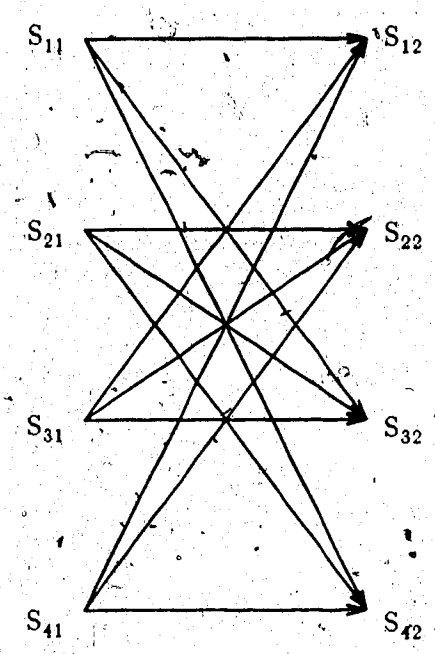

Figure 9 An acyclic precedence graph PG(H) such that none of the topological sorts of PG(H) yield a correct schedule.

Theorem 6.1: Suppose that H is an RC schedule, then H is equivalent to a correct schedule.

Proof: Since H is an RC schedule, therefore there is a topological sort for the nodes of PG(H) which yields a correct schedule H'. We prove that H and H' are equivalent. To prove that condition (1) of schedule-equivalence is satisfied let $R_i(x)$ be an arbitrary read operation in H and suppose that $R_i(x)$ belongs to the step S_{ij}. Suppose first that $R_i(x)$ is not preceded by any write operation on x in H. Then, for each write operation on x belonging to a step S_{kl} (where $k \neq i$), there is an arc $(F(S_{ij},T_k),P(S_{kl},T_i))$ in PG(H). This arc implies that there is a path from S_{ij} to S_{kl} in PG(H) and that S_{ij} precedes S_{kl}

in H', which also implies that $R_i(x)$ cannot be preceded by any write operation on x in H'. Similarly, suppose that $R_i(x)$ returns the value of x written by $W_p(x)$ in H and that $W_p(x)$ belongs to the step S_{ph}. Then, there is an arc $(F(S_{ph},T_i),P(S_{ij},T_p))$ in PG(H). For each write operation $W_k(x)$ belonging to a step S_{kl} (where $k \neq i$) such that $R_i(x) < W_k(x)$ in H, there is an arc $(F(S_{ij},T_k),P(S_{kl},T_i))$ in PG(H). Further, for each write operation $W_q(x)$ belonging to a step S_{qr} (where $q \neq p$) such that $W_q(x) < W_p(x)$ in H, there is an arc $(F(S_{qr},T_p),P(S_{ph},T_q))$ in PG(H). This actually implies that the following paths exist in PG(H): a path from S_{qr} to S_{ph}, a path from S_{ph} to S_{ij}, and a path from S_{ij} to S_{kl}, which also implies that the following must be true in the schedule H', S_{qr} precedes S_{ph} precedes S_{ij} precedes S_{kl}. This obviously indicates that $R_i(x)$ returns the value of x written by $W_p(x)$ in H'. In either case, $R_i(x)$ returns the same value in H and H', which implies that condition (1) of schedule equivalence is satisfied.

Similarly, to prove that condition (2) of schedule equivalence is satisfied let $W_i(x)$ be the last write operation on x in H and suppose that $W_i(x)$ belongs to the step S_{ij}. For each write operation $W_k(x)$ belonging to a step S_{kl} (where $k \neq i$) such that $W_k(x) < W_i(x)$ in H, there is an arc $(F(S_{kl},T_i),P(S_{ij},T_k))$ in PG(H). This arc implies that S_{kl} must precede S_{ij} in H', which also implies that $W_i(x)$ is the last write operation on x in H'. Therefore, condition (2) is also satisfied and H is equivalent to H'. □

CHAPTER 7

THE CONCURRENCY CONTROL MECHANISM

This chapter describes a locking mechanism based on the concepts mentioned in the previous chapters. The proposed mechanism achieves high level of parallelism by allowing serializable as well as non-serializable schedules (which do not violate consistency). The proposed mechanism assumes fewer restrictions on the interleavings among transactions than those assumed in the locking mechanism proposed by Garcia-Molina [Gar83].

7.1. Locking modes

Each transaction is required to obtain a lock on an entity x before accessing (i.e., reading or writing) x. Locks are not requested or released explicitly by the transaction, instead locks are requested and released implicitly by the mechanism in response to the operations and the breakpoints issued by the transaction.

The mechanism utilizes four types (or modes) of locks, denoted exclusive (E), shared (S), relatively exclusive (RE) and relatively shared (RS). Each transaction T_i must obtain an S-lock or E-lock on an entity x before reading or writing x, respectively. (The conditions for granting this lock will be described in detail later). An S-lock on x indicates that x can only be read by any other transaction, while an E-lock on x indicates that x cannot be read or written by any other transaction.

When the transaction T_i issues a breakpoint B_{ij}, each S-lock or E-lock obtained by T_i on an entity x during the execution of the step S_{ij} will be modified to an

RS-lock or an RE-lock, respectively. This new RS-lock (or RE-lock) will be associated with the breakpoint B_{ij} and will be viewed differently by different transactions. Those transactions that are allowed to interleave at B_{ij} see x as an unlocked entity, while those that are not allowed to interleave at B_{ij} see an S-lock (or E-lock) on x.

The mechanism uses the S-locks and the E-locks to ensure that each step in every transaction is processed as an atomic action. Similarly, the mechanism uses the RS-locks and the RE-locks to ensure that the interleavings among the steps cannot violate consistency.

7.2. The values maintained by the mechanism

We describe the values maintained by the mechanism. These values will be used by the mechanism in processing the operations, the breakpoints, and the termination command of the transaction.

L(x): The lock of x. L(x) is a boolean variable indicating whether x is currently locked.

$M(T_i,x)$: The mode of the lock of T_i on x. $M(T_i,x)$ is a variable denoting the mode of the lock of transaction T_i on x. $M(T_i,x)$ will only be needed if T_i is currently holding a lock on x.

LS(x): The lock set of x. LS(x) contains all transactions currently holding a lock on x.

$IS(T_i,x)$: The interleaving set of T_i on x. T_i accumulates the types of transactions that are allowed to interleave on x in $IS(T_i,x)$. $IS(T_i,x)=\Phi$ when T_i obtains its lock on x. More types will be added to $IS(T_i,x)$ as T_i encounters more breakpoints. $IS(T_i,x)$ will only be needed if T_i is currently holding a lock on x.

$End(T_i)$: A boolean variable indicating whether T_i has reached its termination point,

that is, whether T_i has issued the termination command $TR(T_i)$. The locks of T_i may not be necessarily released at this point. (The condition under which the locks of T_i can be released will be given later).

RG: The release graph of the executing transactions. The mechanism uses this directed graph to decide whether the locks of a certain transaction can be released. Each executing transaction is represented by a node in RG. This node will be added to the graph when the transaction starts. An arc (T_i,T_j) in the graph indicates that the locks of transaction T_i cannot be released before T_j reaches its termination point. This arc will be added to RG iff one of the following conditions is true:

1. T_i obtains an S-lock on an entity x at the time T_j is holding an RE-lock on x.
2. T_i obtains an E-lock on an entity x at the time T_j is holding an RS-lock or RE-lock on x.

The node of the transaction T_i (as well as all other values associated with T_i) will be deleted when all the locks of T_i have been released.

$Rel(T_i)$: The release indicator of T_i. $Rel(T_i)$ is a boolean variable; when it is true, then the release graph need to be checked to see whether the locks of T_i can be released, and when it is false the locks of T_i cannot be released.

In addition to the values described above, we define two sets called the destination set $DS(T_i)$ and the source set $SS(T_i)$ of the transaction T_i. None of these sets will be maintained by the mechanism and they are only defined to simplify some of the concepts presented in this chapter.

$DS(T_i)$: $DS(T_i)$ contains all types of transactions that are allowed to interleave at any breakpoint in T_i. That is, for each breakpoint B_{ij} in T_i, $TP1 \in t(B_{ij})$ implies that $TP1 \in DS(T_i)$.

$SS(T_i)$: $SS(T_i)$ contains all types of transactions that allow $t(T_i)$ to interleave at any breakpoint. That is, for each breakpoint B_{kl} in a transaction T_k, $t(T_i) \in t'(B_{kl})$ implies that $t(T_k) \in SS(T_i)$.

7.3. Assumptions on the breakpoints.

We add the following constraints on the breakpoints of the transactions.

Assumption 7.1: If $t(T_i) \in t'(B_{kl})$, then $DS(T_i) \subseteq t'(B_{kl})$.

Assumption 7.1 makes the interleaving among the transactions transitive, that is, if transaction T_i can interleave at a breakpoint B_{kl} in transaction T_k and transaction T_j can interleave at any breakpoint in transaction T_i, then transaction T_j can also interleave at the breakpoint B_{kl}.

Assumption 7.2: If $t(T_i) \in t'(B_{kl})$ and $t(T_k) \in t'(B_{ij})$, then $SS(T_i) \subseteq t'(B_{kl})$. That is, if transaction T_k can interleave at any breakpoint B_{ij} in transaction T_i, then any breakpoint in T_k that allows T_i to interleave also allows all types in $SS(T_i)$ to interleave.

The above assumptions are less restricted than those assumed in [Gar83]. According to the assumptions in [Gar83], if T_i can interleave (at any breakpoint) with T_j, then T_j can interleave (at every breakpoint) with T_i and the set of types that is associated with any breakpoint in T_i or T_j is the same. In other words, transactions are divided into a set of (disjoint) classes such that the transactions that belong to the same class can interleave arbitrarily, while the transactions that belong to different classes cannot interleave at all. Our assumptionss allow other interleavings which are not allowed in [Gar83].

7.4. Processing the operations and managing the locks

This section describes how the locking mechanism manages the locks, and outlines the procedures used for processing the operations, the breakpoints, and the termination command. We assume that the values associated with each entity x will be initialized as follows: L(x) is false and LS(x) is empty. We also assume that when a transaction T_i starts, End(T_i) and Rel(T_i) are false.

(1) Processing a read operation and granting an S-lock: When a read operation $R_i(x)$ is received by the mechanism, the mechanism will respond by either accepting the operation and granting T_i an S-lock on x, or delaying the operation. The conditions under which each of the previous responses will be taken are described in the following procedure.

```
Procedure Process-read (Ri(x))
begin
 if L(x)= true
  then begin
       for each Tj ∈ LS(x) do
       if ((M(Tj,x)= E) or (M(Tj,x)= RE and t(Ti) ∉ IS(Tj,x)))
        then begin
                delay Ri(x) (until Tj releases its lock or issues
                a breakpoint that allows t(Ti) to interleave);
                exit {*exit from procedure*}
               end
       accept Ri(x)
      end
  else accept Ri(x);
 L(x) ← true;
 LS(x) ← LS(x) ∪ {Ti};
 M(Ti,x) ← S;
 IS(Ti,x) ← Φ;
 for each Tj such that M(Tj,x)= RE do
  add the arc (Ti,Tj) to RG
end;
```

(2) Processing a write operation and granting an E-lock: When a write operation $W_i(x)$ is received by the mechanism, the mechanism will respond by either accepting the operation and granting T_i an E-lock on x, or delaying the operation. The condi-

tions under which each of the previous responses will be taken are described in the following procedure.

```
Procedure Process-write (W_i(x))
begin
 if L(x)=true
  then begin
       for each T_j ∈ LS(x) do
       if ¬ ((T_j=T_i) or (M(T_j,x)=RS and t(T_i) ∈ IS(T_j,x)) or
            (M(T_j,x)=RE and t(T_i) ∈ IS(T_j,x)))
        then begin
               delay W_i(x) (until T_j releases its lock or issues
               a breakpoint that allows t(T_i) to interleave);
               exit
             end
       accept W_i(x)
      end
  else accept W_i(x);
 L(x) ← true;
 LS(x) ← LS(x) ∪ {T_i};
 M(T_i,x) ← E;
 IS(T_i,x) ← Φ;
 for each T_j such that M(T_j,x)=RS or RE do
  add the arc (T_i,T_j) to RG
end;
```

(3) Processing a breakpoint B_{ij}: When a breakpoint B_{ij} is received by the mechanism, the mechanism will respond by modifying the locks obtained during the execution of the step S_{ij} as described in the following procedure.

```
Procedure Process-breakpoint (B_ij)
begin
 for each entity x such that T_i ∈ LS(x) do
  if M(T_i,x)=S
   then begin
        M(T_i,x) ← RS;
        IS(T_i,x) ← t(B_ij)
        end
   else if M(T_i,x)=E
        then begin
             M(T_i,x) ← RE;
             IS(T_i,x) ← t(B_ij)
             end
        else IS(T_i,x) ← IS(T_i,x) ∪ t(B_ij)
end;
```

(4) Processing the termination command and releasing locks: When the termination command TR(T_i) is received by the mechanism, the mechanism will check the releases graph (RG) to see whether or not the locks of T_i can be released. The conditions under which the locks of T_i can be released are described in the following procedure. (In this procedure, we will use the symbol $R'(T_i)$ to denote the set of all nodes in RG that are reachable from the node T_i)[1].

```
Procedure Process-terminate (TR(Ti))

  Procedure Release-locks (Ti)
  begin
  for each entity x such that Ti ∈ LS(x) do
   begin
    LS(x) ← LS(x)-{Ti};
   if LS(x)=Φ
     then L(x) ← false;
   end
  end;

  Procedure Check-release (Ti)
  begin
    for each Tj∈R'(Ti) do
     if End(Tj)=false then
       begin
         delay (releasing the locks of) Ti;
         for each Tj∈R'(Ti) do Rel(Tj) ← false;
         exit
       end
    for each Tj∈R'(Ti) do
     begin
      Release-locks (Tj);
      for each Tk such that (Tk,Tj) is in RG and Tk∉R'(Ti) do
        begin
        delete the arc (Tk,Tj) from RG;
        if End(Tk)=true then Rel(Tk) ← true
        end
     end
    delete all nodes Tj∈R'(Ti) (and the values associated with them)
  end

begin
   End(Ti) ← true;
   Rel(Ti)← true;
   while there is a node Tk in RG such that Rel(Tk)=true do
```

[1] This set can be found by performing depth-first-search [Aho74] starting at the node T_i.

Check-release(T_k)
end;

7.5. Commitment of transaction

So far we have not mentioned when the transaction can be committed, i.e., when the user will be acknowledged that his transaction has been successfully completed. The answer to this question is dependent on the approach used for handling transaction failure. A transaction may fail for several reasons (like integrity violation, system failure, hardware failure, etc.). When a transaction T_i fails, T_i may be backed up. Backing up T_i may necessitate backing up any other transaction T_j which read an entity written by T_i. Similarly, backing up T_j may also necessitate backing up other transactions, and so on. That is, a cascade of back-ups might occur. (In the proposed mechanism, the set of transactions that must be aborted after backing up T_i can be found by searching the release graph. These transactions can be minimized by distinguishing among the arcs of the release graph as described in [Far82]).

Allowing a cascade of back-ups to occur can degrade performance. Moreover, it will also prevent the commitment of a transaction T_i until all the locks held by T_i have been released.

There are also other approaches which avoid a cascade of back-ups and allow the transaction to be committed when it reaches its termination point. One approach is based on the concept of "countersteps" used in [Gar83]. In this approach, undoing the effect of a step S_{ij} can be achieved by requiring the user to provide a counterstep C_{ij}. Similarly, to undo the effect of the whole transaction, the user must provide a counterstep for each step in the transaction.

The above approach has the drawback that in some applications, in which transactions may be long and complicated, the task of analyzing the transactions and

writing countersteps may be difficult to be done by an inexperienced user. Furthermore, relying on such a user may add the risk of destroying the consistency of the database.

Another solution (which reduces concurrency) is to modify the procedures given previously to make the S-lock and the RE-lock "incompatible", i.e., cannot be held on the same entity concurrently. In this case, backing up a transaction T_i cannot lead to backing up any other transaction.

7.6. Other problems

One problem which has not been discussed and which is associated with the proposed mechanism (and with most of the other locking-based mechanisms) is deadlock [Gra78, Hol72]. Deadlock can occur because transactions wait for one another. The main approaches for solving the deadlock problem are deadlock detection [Gra78] and deadlock prevention [Ros78, Ryp79]. In the first approach the problem will be resolved by detecting the deadlock and aborting one (or more) transaction. Detecting a deadlock requires maintaining a directed graph called the wait-for graph which represents the wait-for relationships among the executing transactions. This graph has been described in Chapter 4.

In the second approach, the problem will be resolved by preventing the deadlock from occurring. This can be achieved by assigning timestamps to the transactions and using these timestamps to decide whether a transaction T_i can wait for another transaction T_j (as in [Ros78]). Other methods for avoiding deadlock require that the transactions preclaim all the entities that will be accessed in advance [Ryp79], or that the transactions access the entities in a certain (predefined) order.

7.7. The correctness of the proposed mechanism

This section proves the correctness of the locking mechanism, that is, it proves that the mechanism produces only RC schedules.

First, we define two sets for each transaction T_1, called the input set $IN(T_1)$ and the output set $OT(T_1)$. None of these sets need to be maintained by the mechanism and they are only defined to simplify the proof of correctness. Let H be a schedule of a set of transactions $\{T_1,T_2,T_3$, etc.$\}$ produced by the locking mechanism. The input set $IN(T_1)$ is defined as follows:

(1) Each step in T_1 is also in $IN(T_1)$.

(2) For each *C*-arc (S_{ij},S_{kl}) in PG(H), if $S_{kl} \in IN(T_1)$, then every step in T_i is also in $IN(T_1)$.

Similarly, the output set $OT(T_1)$ is defined as follows:

(1) Each step in T_1 is also in $OT(T_1)$.

(2) For each *C*-arc (S_{ij},S_{kl}) in PG(H), if $S_{ij} \in OT(T_1)$, then every step in T_k is also in $OT(T_1)$.

Observation 7.1: If $IN(T_1) \cap OT(T_2) \neq \Phi$, then transaction T_1 cannot release any lock before transaction T_2 reaches its termination point.

In the following lemma and theorems, we may use the symbol $TR(S_{ij})$ to denote the breakpoint of the step S_{ij}.

Lemma 7.1: Let H be a schedule produced by the mechanism and let (S_{1i},S_{2l}) be a *C*-arc in PG(H). (S_{1i},S_{2l}) will be called C_1-arc if $t(T_2) \notin DS(T_1)$. Otherwise, (S_{1i},S_{2l}) will be called C_2-arc. Let $(F(S_{1i},T_2),P(S_{2j},T_1))$ be any *T*-arc in PG(H), then one of the following two conditions is true:

(1) There is a C_1-arc (S_{kl},S_{pq}) such that S_{kl} and S_{pq} belong to $OT(T_1) \cap IN(T_2)$.

(2) There is a path from $F(S_{1i},T_2)$ to $P(S_{2j},T_1)$ such that each arc along this path is an I-arc or C_2-arc.

Proof: First, we introduce a recursive definition for the T-arcs as follows. Let $(F(S_{1i},T_2),P(S_{2j},T_1))$ be a T-arc in PG(H), then

(1) $(F(S_{1i},T_2),P(S_{2j},T_1))$ will be called T_1-arc, if for each arc (S_{kl},S_{pq}) along the path from S_{1i} to S_{2j} (which caused the T-art $(F(S_{1i},T_2),P(S_{2j},T_1))$), (S_{kl},S_{pq}) is I-arc or C-arc.

(2) Similarly, $(F(S_{1i},T_2),P(S_{2j},T_1))$ will be called T_n-arc, $n>1$, if for each arc (S_{kl},S_{pq}) along the path from S_{1i} to S_{2j} (which caused the T-arc $(F(S_{1i},T_2),P(S_{2j},T_1))$), (S_{kl},S_{pq}) is I-arc, C-arc, T_1-arc, T_2-arc,..., or T_{n-1}-arc.

We prove this lemma by induction on the T_n-arcs ($n \geq 1$). Suppose that $(F(S_{1i},T_2),P(S_{2j},T_1))$ is T_1-arc. If the path from S_{1i} to S_{2j} contains a C_1-arc, then it is obvious that $(F(S_{1i},T_2),P(S_{2j},T_1))$ satisfies condition (1) of the lemma. Otherwise, let each C-arc along the path from S_{1i} to S_{2j} be C_2-arc (as shown in Fig. 10)[2]. Suppose (without loss of generality) that the first and last C_2-arcs along the path from S_{1i} to S_{2j} are (S_{1k},S_{3p}) and (S_{4q},S_{2l}), respectively (refer to Fig. 10). Since there is a path from S_{1k} to S_{2l} such that each C-arc along this path is C_2-arc, therefore by assumption 4.1; $t(T_2) \in t(B_{1k})$. Further, Since S_{1i} precedes S_{1k}, then

$F(S_{1i},T_2)$ precedes or the same as S_{1k}. Similarly, by assumption 4.2, the path from S_{1k} to S_{2l} also implies that S_{2l} precedes or the same as $P(S_{2j},T_1)$. But since there is a path from S_{1k} to S_{2l} (such that each arc along this path is I-arc or C_2-arc), therefore there is also a path from $F(S_{1i},T_2)$ to $P(S_{2j},T_1)$ satisfying condition (2) of the lemma. Thus,

[2] The precedence graphs shown on all figures in this chapter may not contain all necessary arcs. The figures show only the arcs which are useful in simplifying the proof.

T_1-arcs satisfy the lemma.

Similarly, suppose that the lemma is true for any T_n-arc, where $n \geq 1$. Let $(F(S_{1i},T_2),P(S_{2j},T_1))$ be T_{n+1}-arc. If there is a C_1-arc along the path from S_{1i} to S_{2j}, or there is a T-arc along the path which satisfies condition (1) of the lemma, then it is obvious that $(F(S_{1i},T_2),P(S_{2j},T_1))$ also satisfies condition (1) of the lemma. Otherwise, let every C-arc along the path be C_2-arc and every T-arc along the path satisfy condition (2) of the lemma, then it is not difficult to see that $(P(S_{1i},T_2),P(S_{2j},T_1))$ must also satisfy condition (2) of the lemma. □

Observation 7.2: If (S_{ij},S_{kl}) is T-arc in PG(H), then $S_{ij} \in IN(T_k)$ and $S_{kl} \in OT(T_i)$.

Theorem 7.1: Let H be a schedule produced by the mechanism, then PG(H) is acyclic.

Proof: Suppose to the contrary that PG(H) is cyclic. Moreover, suppose (without loss of generality) that the set $E=\{(S_{1i},S_{2j}), (S_{2k},S_{3l}), (S_{3p},S_{4q}), (S_{4q},S_{1i})\}$ represents the set of all arcs along the cycle that connect nodes belonging to different transactions (refer to Fig. 11).

Suppose first that every arc in E is *C-arc*. Suppose further that E contains one (or more) C_1-arc. For example, let (S_{2k},S_{3l}) be C_1-arc. The arc (S_{2k},S_{3l}) implies that transaction T_2 releases all its locks before transaction T_3 reaches its termination point. But since S_{3l} is an ancestor of S_{2k} in PG(H), then transaction T_2 cannot release any lock before T_3 reaches its termination point; a contradiction. Otherwise, suppose that every arc in E is C_2-arc. Then, the arc (S_{1i},S_{2j}) implies that $TR(S_{1i}) < TR(F(S_{2j},T_1))$ in H. Since S_{2j} precedes S_{2k}, then $TR(S_{1i}) < TR(F(S_{2k},T_i))$. Further, since there is a path from S_{2k} to S_{1i} such that each C-arc along this path is C_2-arc, therefore by assumption 4.1, $TR(F(S_{2k},T_1))=TR(S_{2k})$. This obviously implies

that $TR(S_{1i}) < TR(S_{2k})$ in H. In a similar way, we can easily prove that the arcs of the set E imply the following: $TR(S_{1i}) < TR(S_{2k}) < TR(S_{3p}) < TR(S_{4q}) < TR(S_{1i})$. That is, $TR(S_{1i}) < TR(S_{1i})$; a contradiction. Therefore, there cannot be a cycle in PG(H), such that each arc along the cycle is *I*-arc or *C*-arc.

Otherwise, let E contains one (or more) *T*-arcs. Suppose (without loss of generality) that (S_{2k},S_{3l}) is the only *T*-arc in E. Suppose first that (S_{2k},S_{3l}) satisfies condition (1) of Lemma 7.1 and let (S_{5p},S_{6q}) be a C_1-arc such that S_{5p} and S_{6q} belong to $OT(T_5) \cap IN(T_3)$. The arc (S_{5p},S_{6q}) implies that transaction T_5 releases all its locks before transaction T_6 reaches its termination point. Further, since there is a path from S_{3l} to S_{2k} and $OT(T_2) \cap IN(T_3) \neq \Phi$, therefore, $OT(T_6) \cap IN(T_5) \neq \Phi$. But this actually implies that T_5 cannot release any lock before T_6 reaches its termination point; a contradiction. Similarly, suppose that (S_{2k},S_{3l}) satisfies condition (2) of Lemma 7.1. This actually implies that there is a cycle in PG(H) such that each arc along the cycle is *I*-arc or *C*-arc; a contradiction. Therefore, our initial assumption that PG(H) is cyclic cannot be true. □

Theorem 7.2: Let H be a schedule produced by the mechanism. Then, there is a topological sort for the nodes of PG(H) which yields a correct schedule.

Proof: We prove this theorem by induction on the number of topological sorts for PG(H). Suppose first that there is only one topological sort for PG(H) and let H_1 be the schedule that corresponds to this topological sort (that is, H_1 is formed by replacing each step S_{ij} in the sort with the operations and the breakpoint of S_{ij}, and then adding the termination command at the end of each transaction). We claim that H_1 is a correct schedule.

Suppose that our claim is not true and let S_{1i} and S_{1i+1} be a pair of steps in H_1

such that S_{2j} appears between S_{1i} and S_{1i+1} and $t(T_2) \notin t'(B_{1i})$. Since PG(H) has only one topological sort, therefore there is a path between any pair of nodes in PG(H) (in one direction). Further, since S_{1i} appears before S_{2j} in H_1, then there is a path from S_{1i} to S_{2j} in PG(H). Similarly, since S_{2j} appears before S_{1i+1} in H_1, then there is also a path from S_{2j} to S_{1i+1}. The path from S_{1i} to S_{2j} also implies that there is a T-arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ in PG(H). Moreover, since $t(T_2) \notin t'(B_{1i})$, then the T-arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ implies that there is a path from S_{1i+1} to S_{2j}, that is, PG(H) is cyclic; a contradiction. Therefore, H_1 must be a correct schedule.

Similarly, suppose that the theorem is true for any precedence graph with n topological sorts, where $n \geq 1$, and let PG(H) have n+1 topological sorts. Now, we prove that PG(H) has a topological sort that yields a correct schedule by showing that a new precedence graph PG(H)' which has less than n+1 topological sorts can be constructed by adding more arcs to PG(H). Let S_{1i} and S_{2j} be arbitrary nodes in PG(H) such that there is no path from S_{1i} to S_{2j} or vice versa. Consider the following cases:

(1) $OT(T_1) \cap IN(T_2) = \Phi$ (see Fig. 12(a)).

(2) There is a C_1-arc (S_{ij},S_{kl}) such that S_{ij} and S_{kl} belong to $OT(T_1) \cap IN(T_2)$ (see Fig. 12(b)).

(3) $OT(T_1) \cap IN(T_2) \neq \Phi$ and for each C-arc (S_{ij},S_{kl}) such that S_{ij} and S_{kl} belong to $OT(T_1) \cap IN(T_2)$, (S_{ij},S_{kl}) is C_2-arc (see Fig. 12(c)).

Suppose that case (1) is true and let the arc $(F(S_{2j},T_1),P(S_{1i},T_2))$ be added to PG(H). (This arc is not shown in Fig. 11.3(a)). The addition of this arc will lead to adding a new set of T-arcs (according to condition (3) of the precedence graph). It is not difficult to see that each arc to be added (including the arc $(F(S_{2j},T_1),P(S_{1i},T_2))$) will be directed from a node in $IN(T_2)$ to another node in $OT(T_1)$ (refer to Fig. 12(a)). But since that $OT(T_1) \cap IN(T_2) = \Phi$, then the addition of all new arcs cannot create a

cycle.

Similarly, suppose that case (2) is true and let (S_{3l},S_{4k}) be a C_1-arc such that S_{3l} and S_{4k} belong to $OT(T_1) \cap IN(T_2)$ (as shown in Fig. 12(b)). Then, it is not difficult to see that $OT(T_4) \cap IN(T_3)=\Phi$. (Otherwise, i.e., if $OT(T_4) \cap IN(T_3)\neq\Phi$, transaction T_3 cannot release any lock before T_4 reaches its termination point, contradicting the meaning of the C_1-arc (S_{3l},S_{4k})). Suppose that the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ is added to PG(H). (This arc is not shown in Fig. 11.3(b)). The addition of this arc will lead to adding a new set of T-arcs in PG(H). It is not difficult to see that each arc to be added (including the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$) will be directed from a node in $IN(T_3)$ to another in $OT(T_4)$. But since $OT(T_4) \cap IN(T_3)=\Phi$, then the addition of the new arcs cannot create a cycle.

Finally, suppose that case (3) is true and let the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ be added to the graph. (This arc is not shown in Fig. 11.3(c)). If adding the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ alone creates a cycle in PG(H), then there is already a path from $P(S_{2j},T_1)$ to $F(S_{1i},T_2)$. This path also implies that there is a T-arc $(F(S_{2j},T_1),P(S_{1i},T_2))$ in PG(H). But this arc implies that there was already a path from S_{2j} to S_{1i} before adding the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ to PG(H); a contradiction. Therefore, the addition of the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ alone cannot create a cycle in PG(H). It remains to show that any T-arc which need to be added after the addition of the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$ cannot create a cycle.

Let S_{3l} and S_{4k} be arbitrary nodes in PG(H) such that there is a path from S_{3l} to $F(S_{1i},T_2)$ and there is another path from $P(S_{2j},T_1)$ to S_{4k}. These two paths imply that after adding the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$, there will be a path in PG(H) from S_{3l} to S_{4k}. The path from S_{3l} to S_{4k} implies that the T-arc $(F(S_{3l},T_4),P(S_{4k},T_3))$ need to be added to the graph. If one of the arcs along the path from S_{3l} to S_{4k} is C_1-arc, or T-

arc which satisfies condition (1) of Lemma 7.1, then it is easy to show that the addition of the arc $(F(S_{3l},T_4),P(S_{4k},T_3))$ cannot create a cycle (the proof is simple and similar to case (2)). Otherwise, let each arc along the path from S_{3l} to S_{4k} be C_2-arc, or T-arc which satisfies condition (2) of Lemma 7.1. In this case, we prove that the addition of the arc $(F(S_{3l},T_4),P(S_{4k},T_3))$ cannot create a cycle by showing that there is already a path from $F(S_{3l},T_4)$ to $P(S_{4k},T_3)$ before adding the arc $(F(S_{3l},T_4),P(S_{4k},T_3))$ (and after adding the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$. Since there is a path from S_{3l} to $F(S_{1i},T_2)$, then there is a T-arc $(F(S_{3l},T_1),P(F(S_{1i},T_2),T_3))$ in PG(H) (refer to Fig. 12(c)). Further, by assumption 4.1, the path from S_{3l} to S_{4k} implies that $F(S_{3l},T_4)$ precedes or the same as $F(S_{3l},T_1)$. This obviously implies that there is a path from $F(S_{3l},T_4)$ to $F(S_{1i},T_2)$ in PG(H). Similarly, the path from $P(S_{2j},T_1)$ to S_{4k} implies that the T-arc $(F(P(S_{2j},T_1),T_4),P(S_{4k},T_2))$ exists in PG(H) (refer to Fig. 12(c)). Moreover, by assumption 4.2, the path from S_{3l} to S_{4k} implies that $P(S_{4k},T_2)$ precedes or the same as $P(S_{4k},T_3)$. Which also implies that there is a path from $P(S_{2j},T_1)$ to $P(S_{4k},T_3)$. Therefore, the following three paths exist in PG(H) before adding the arc $(F(S_{3l},T_4),P(S_{4k},T_3))$, (and after adding the arc $(F(S_{1i},T_2),P(S_{2j},T_1))$), a path from $F(S_{3l},T_4)$ to $F(S_{1i},T_2)$, a path from $F(S_{1i},T_2)$ to $P(S_{2j},T_1)$, and a path from $P(S_{2j},T_1)$ to $P(S_{4k},T_3)$. Which also implies that the addition of the arc $(F(S_{3l},T_4),P(S_{4k},T_3))$ cannot create a cycle.

The previous argument can be applied recursively (that is, if S_{5p} and S_{6q} are arbitrary nodes in PG(H) such that there is a path from S_{5p} to $F(S_{3l},T_4)$ and there is another path from $P(S_{4k},T_3)$ to S_{6q}, then we can easily prove that the addition of the arc $(F(S_{5p},T_6),P(S_{6q},T_5))$ cannot create a cycle.) This actually implies that the addition of all new arcs cannot create a cycle.

Thus, in each of the previous cases, a new precedence graph PG(H)' can be constructed from PG(H) by adding more arcs (which do not create a cycle). PG(H)' has

a topological sort which yields a correct schedule. But since each topological sort for PG(H)' is also a topological sort for PG(H), then PG(H) also has a topological sort which yields a correct schedule. □

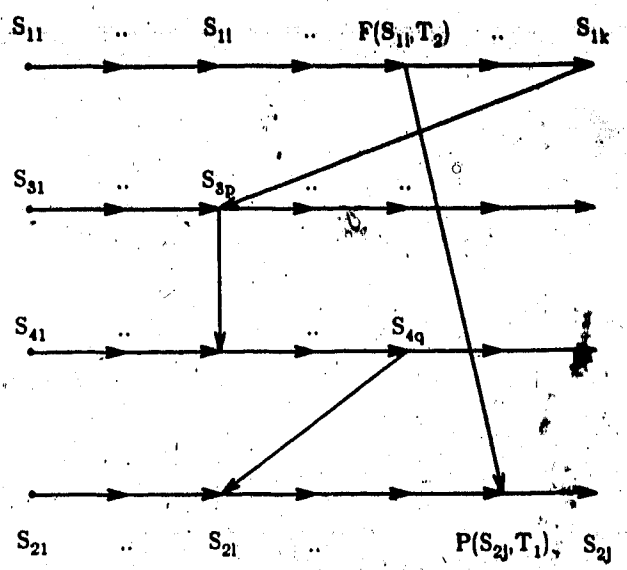

Fig. 10 T-arc $(F(S_{1l},T_2),P(S_{2j},T_1))$ satisfying condition

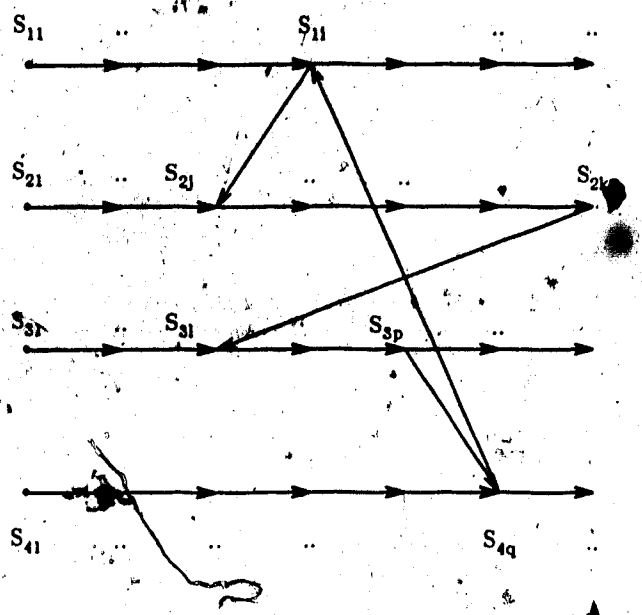

Fig. 11 The arcs of the (assumed) cycle of PG(H).

Fig. 12(a) Case (1): $OT(T_1) \cap IN(T_2) = \Phi$.

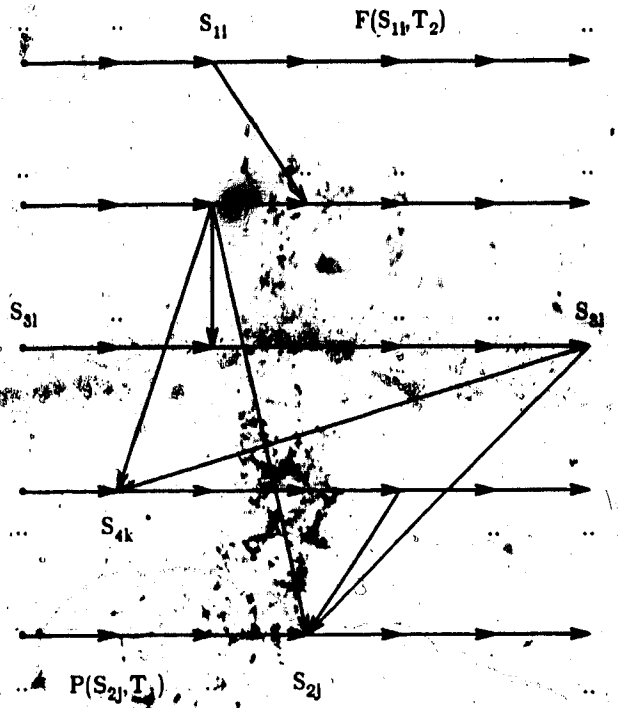

Fig. 12(b) Case (2): There is a C_1-arc (S_{3l},S_{4k}) such that S_{3l} and S_{4k} belong to $OT(T_1) \cap IN(T_2)$.

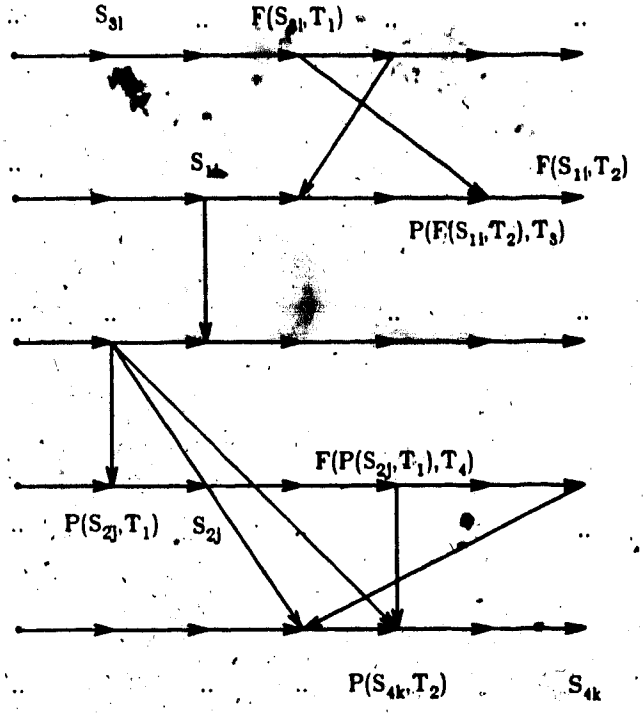

Fig. 12(c) Case (3): $OT(T_1) \cap IN(T_2) \neq \Phi$ and for each C-arc (S_{ij},S_{kl}) such that S_{ij} and S_{kl} belong to $OT(T_1) \cap IN(T_2)$, (S_{ij},S_{kl}) is C_2-arc.

CHAPTER 8

CONCLUSIONS

The concurrency control problem in database systems has been examined by many researchers and several concurrency control algorithms have been proposed. The task of a concurrency control algorithm is to ensure the consistency of the database while allowing a set of transactions to execute concurrently. The consistency of the database is defined in terms of a set of constraints called consistency or integrity constraints. The database is consistent whenever the values of its entities satisfy the integrity constraints; otherwise, the database is inconsistent.

In this thesis, two topics have been examined. The first topic deals with the concurrency control problem when the only information available about the transactions is syntactic information. In this topic serializability is the correctness criterion for concurrency control. Serializability requires that the only acceptable schedules are those which provide equivalent to a serial execution of the transactions.

The means for achieving serializability are diverse; the most popular means are two-phase locking and timestamp ordering mechanisms. In this thesis these two mechanisms have been reviewed and proven to be special cases of a general concurrency control mechanism. This general mechanism has been described in detail and proven to work correctly (i.e., produces only serializable schedules).

It has been shown that two-phase locking and timestamp ordering represent the two end points of a series of concurrency control mechanisms. Each of these mechanisms results from choosing a different value for the strictness level and is considered to be a special case of the general concurrency mechanism proposed in this

thesis. The value of the strictness level can be specified in advance and can be modified during execution (without affecting the correctness of the proposed mechanism).

In the second topic, the concurrency control problem has been examined when semantic information is available about the transactions. The semantic information considered is transaction types, transaction steps, and transaction breakpoints. A new class of schedules called relatively consistent (RC) schedules has been defined. This class of RC schedules contains serializable and non-serializable schedules.

The main motivation behind allowing non-serializable schedules is to improve performance. Several researchers have noticed that in some applications allowing only serializable schedules can reduce the level of concurrency and increase transaction response time. One way to increase concurrency is to allow non-serializable schedules that do not violate consistency. A new concurrency control mechanism that allows serializable and non-serializable schedules (which do not violate consistency) has been proposed. The presented mechanism allows more interleavings among the executing transactions than the locking mechanism proposed by [Gar83]. Consequently, the proposed mechanism achieves higher level of concurrency.

8.1. Future work

There are still other questions that remain open for future research. One problem that is very important and which has not been discussed in detail, is performance. We believe that the performance of a concurrency control mechanism must depend on both the likelihood of transaction conflict and the level of concurrency allowed by the mechanism.

The likelihood of transaction conflict is application dependent, that is, it varies from application to application, while concurrency depends on the mechanism

used. Some mechanisms allow more concurrency than others. For example, two-phase locking does not allow two conflicting transactions to execute concurrently, that is, if two transactions are in conflict, then one of them must wait for the other. Other mechanisms (like timestamp ordering) may allow two (or more) transactions to execute concurrently as long as their conflict cannot violate consistency. Several other mechanisms exist between these two extremes (see Chapter 3).

In the mechanism proposed in Chapter 3, the level of concurrency can be changed by modifying the current value of L. The smaller the value of L, the greater the "expected" value of strict classes (as indicated by Lemma 3.1), the more conflicting transactions can proceed execution together. Thus, the performance can be tuned by modifying the value of L. The exact relationship between performance and the previous two concepts (i.e., transaction conflict and concurrency) is an interesting problem that deserves further research.

There are also other problems related to the second topic. One problem is extending our work for new transaction models. Another problem is handling different types of failures efficiently (such as transaction failure, system failure, etc.). A third problem is comparing the performance of the concurrency control mechanisms that allow serializable and non-serializable schedules against those that allow only serializable schedules. Recent results on this area are encouraging [Cor85]. These results show that when transaction conflict is relatively high, the locking mechanism of Garcia-Molina [Gar83] outperforms the conventional two-phase locking mechanism in terms of transaction response time. Since our mechanism does not require each transaction to obtain all the locks that it needs before execution (as in [Gar83]), as well as allowing more interleaving among the transactions, we expect that transaction response time will be reduced further in our mechanism.

REFERENCES

[Aho74] A. Aho, J. Hopcroft and J. Ullman, "*The Design and Analysis of Computer Algorithms*", Addison-Wesley, 1974.

[Bay80] R. Bayer, H. Heller and A. Reiser, "Parallelism and Recovery in Database Systems", *ACM Trans. Database Systems* 5, 2 (June 1980), 139-156.

[Ber80] P. Bernstein and N. Goodman, "Timestamp Based Algorithm for Concurrency Control Problem for Multiple Copy Databases", *Proceedings of the 6th International Conference on Very Large Databases*, October 1980.

[Ber80a] P. Bernstein, D. Shipman and J. Rothnie, "Concurrency Control in a System for a Distributed Databases (SDD-1)", *ACM Trans. Database Systems* 5, 1 (March 1980), 81-51.

[Ber81] P. Bernstein and N. Goodman, "On Concurrency Control in Distributed Database Systems", *ACM Computing Surveys* 13 , 2 (June 1981), 185-221.

[Ber83] P. Bernstein and N. Goodman, "*Concurrency Control and Recovery for Replicated Distributed Databases*", Tech. Rep.-20, Centre for Research in Computing Technology, Harvard University, July 1983.

[Cor85] R, Cordon, "*Using Semantic Knowledge for Transaction Processing*", Dept. of Electrical Engineering and Computer Science, Princeton University, Ph.D. thesis October 1985.

[Esw76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Comm. ACM*

19, 11 (Nov. 1976), 624-633.

[Far82] A. A. Farrag and T. Kameda, "*On Concurrency Control Using Multiple Versions*", Dept. of Computing Science, Simon Fraser University, Tech. Rep. 1982-13, Aug. 1982.

[Far85] A. A. Farrag and M. T. Ozsu, "A General Concurrency Control Algorithm for Database Systems", *Proceedings of the National Computer Conference*, Chicago, Illinois , July 1985.

[Fis81] M. Fischer and N. Lynch, "Global States of Distributed System", *IEEE Trans. on Software Eng.* 8, 3 (July 1981), 198-202.

[Gar82] H. Garcia-Molina and G. Wiederhold, "Read-only Transactions in Distributed Database", *ACM Trans. Database Systems* 7, 2 (June 1982), 209-234.

[Gar83] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database", *ACM. Trans. Database Systems* 8, 2 (June 1983), 186-213.

[Gli80] V. D. Gligor and S. H. Shattuck, "On Deadlock Detection in Distributed Systems", *IEEE Trans. on Software Eng.* 6, 5 (Sept. 1980), 435-440.

[Gra76] J. N. Gray, R. A. Lorie, G. R. Putzolu and L. I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database", *Proc. IFIP Working Conf. on Modeling of Database*, January 1976, 695-723.

[Gra78] J. N. Gray, "Notes on database operating systems", *in Operating systems: An advanced course, Lecture notes in computer science*, Springer-Verlag, N.Y., 1978, 393-481.

[Hol72] R. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Survey* 4, 3 (December 1972), 179-195.

[Ked79] Z. Kedem and A. Silberschatz, "Controlling Concurrency Using Locking", *Proceedings of the 20th International Conference on Foundation of Computer*, October 1979, 274-285.

[Ked80] Z. Kedem and A. Silberschatz, "Non-two-phase Locking Protocols with Shared and Exclusive Locks", *Proceedings of the 6th International Conference on Very Large Databases*, October 1980.

[Kun79] H. T. Kung and P. H. Papadimitriou, "An Optimality Theory of Database Concurrency Control", *Proceedings of the ACM-SIGMOD*, 1979, 116-126.

[Kun81] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Trans. Database Systems* 6 , 2 (June 1981), 213-226.

[Lam76] L. Lamport, "*Towards a Theory of Correctness of Multi-user Database Systems*", Massachusetts Computer Associates Tech. Rep. CA 7610-0712 , October 1976.

[Leu79] J. Y. Leung and E. K. Lai, "On Minimum Cost Recovery From System Deadlock", *IEEE Trans. on Computers* 28, 9 (Sept. 1979), 671-677.

[Lyn83] N. Lynch, "Multilevel Atomicity-A New Correctness Criterion for Database Concurrency Control", *ACM Trans. Database Systems* 8, 4, (December 1983), 484-502.

[Men79] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distri-

buted Databases", *IEEE Trans. on Software Eng.* 5, 3 (May 1979), 195-201.

[Obe82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Trans. Database Systems* 7, 2, (June1982), 187-208.

[Pap79] C. Papadimitriou, "The Serializability of Concurrent Database Updates", *J. ACM* 26, 4 (Oct. 1979), 631-653.

[Ree78] D. P. Reed, "*Naming and Synchronization in Decentralized Computer Systems*", Dept. Electrical Engineering and Computer Science, MIT Tech. Rep.-205, September 1978.

[Ros78] D. Rosenkrantz, R. Stearns and P. Lewis, "System Level Concurrency Control for Distributed Database Systems", *ACM Trans. Database Systems* 3, 2 (June 1978), 178-198.

[Ryp79] D. Rypka and A. P. Lucido, "Deadlock Detection and Avoidance for Shared Logical Resources", *IEEE Trans. on Software Eng.* 5, 5 (September 1979), 465-471.

[Sil80] A. Silberschatz and Z. Kedem, "Consistency in Hierarchical Database System", *J. ACM* 27, 1 (January 1980), 72-80.

[Ste76] R. Stearns, P. Lewis and D. Rosenkrantz, "Concurrency Control for Database systems", *Proceedings of the IEEE-Annual Symp. on Foundations of Computer Science*, 1976, 19-32.

[Sto79] M. Stonebraker,"Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Trans. on Software Eng.* 5, 3 (May 1979), 180-194.

[Tho78] R. H. Thomas, "A Solution to the Concurrency Control Problem for Multiple Copy Databases", *Proceedings of the COMPCON Conference*, N.Y., 1978.

[Tra82] I. Traiger, J. Gray, C. Galthier and B. Lindsay, "Transactions and Consistency in Distributed Database Systems", *ACM Trans. Database Systems* 7, 3 (September 1982), 180-209.

APPENDIX I

SEMANTICALLY CONSISTENT SCHEDULES

Garcia-Molina [Gar83] has defined a new class of schedules called semantically consistent schedules. This class contains serializable and non-serializable schedules (which do not violate consistency). A concurrency control mechanism that accepts only semantically consistent schedules has also been proposed in [Gar83]. This chapter reviews the work that has been done by Garcia-Molina [Gar83]. First, we introduce the following definitions.

Definition I.1: We modify slightly our previous definition of a transaction given in Chapter 8. In the new definition, a transaction will not contain breakpoints. Instead, each transaction will consist of a series of steps. (The beginning and the end of each step may be identified by a special symbol).

Definition I.2: A schedule is defined as in Chapter 8, except that we will only allow step-wise serial schedules. That is, each step in every transaction will be executed atomically.

Definition I.3: A schedule H is said to be semantically consistent iff the following conditions are true.

(1) The execution of H maintains the consistency of the database (i.e., S transforms a consistent database state into a new consistent state).

(2) Any transaction in H that displays data to users must obtain a consistent view of the database, that is, all consistency constraints that can be evaluated by the data accessed by the transaction must evaluate to true.

(3) For certain distinguished entities of the database, the execution of H must have the same effect on these entities as a serial execution of the transactions.

I.1. Specifying semantically consistent schedules

Semantically consistent schedules are defined in terms of a collection of sets, called compatibility sets. These sets describe all the rules that govern the interleaving among the transactions. Any schedule H that satisfies these rules is semantically consistent (that is, H will satisfy the conditions described previously). There is a compatibility set CS(TP1) for each type TP1. The definition of the compatibility sets will only depend on the semantic information available about the transactions.

Example I.1:

Consider the compatibility set described below.

CS(TP1)={{TP1,TP2},{TP1,TP3}}

The above set implies the following:

1. A transaction of type TP1 can interleave with a transaction of type TP2 or type TP3.
2. Transactions of type TP2 can interleave.
3. Transactions of type TP3 can interleave.
4. A transaction of type TP2 cannot interleave with another transaction of type TP3.

The elements of CS(TP1) are called interleaving descriptors. Note that if the interleaving descriptors of a certain type are empty, then the transactions of that type cannot interleave at all.

I.2. The locking mechanism

Garcia-Molina [Gar83] has described a locking mechanism that produces only semantically consistent schedules. Several assumptions have been introduced to simplify the mechanism. Before outlining the basic features of this mechanism, we introduce the following definitions.

Definition I.4: A type TP1 is said to be local type, if for every transaction T_i with $t(T_i)$ = TP1, T_i consists only of one step. Otherwise, TP1 is nonlocal type.

Definition I.5: A transaction T_i is said to be local transaction, if $t(T_i)$ is local type. Otherwise, T_i is nonlocal transaction.

(The above definitions must not imply that every single step transaction is local).

Definition I.6: A transaction T_i is said to be compatible with a transaction T_j, iff there is an interleaving descriptor h in $CS(t(T_j))$ such that $t(T_i) \in h$.

The following assumptions are assumed to hold true.

Assumption I.1: The compatibility sets are assumed to be sound, i.e., the compatibility sets must satisfy the following conditions.

(1) There is no CS(TP1) such that $g,h \in CS(TP1)$ and $g \subseteq h$ (i.e., there are no redundant descriptors).

(2) $h \in CS(TP1)$ and $TP2 \in h$, then $h \in CS(TP2)$ (i.e., the descriptors are mutually consistent).

Assumption I.2: If $h \in CS(TP1)$ and $h \neq \Phi$, then $TP1 \in h$. This actually means that if transaction T_i can interleave with another transaction of different type than $t(T_i)$, then T_i can also interleave with any other transaction of the same type.

Assumption I.3: If a transaction T_i is compatible with a transaction T_j, then transaction T_j is compatible with T_i.

Assumption I.4: If TP1 is nonlocal type, then |CS(TP1)| = 1. That is, the compatibility set of a nonlocal type contains only one interleaving descriptor. (Local types do not have this restriction).

The above assumptions divides the nonlocal types into a set of disjoint classes called equivalent classes. Two types TP1 and TP2 belong to the same equivalent class iff CS(TP1)=CS(TP2). (Note that there is one class for the types whose compatibility sets contain only the empty descriptor). If $t(T_i)$ and $t(T_j)$ belong to the same equivalent class, then T_i and T_j can interleave arbitrarily. Otherwise, T_i and T_j cannot interleave.

I.2.1. Types of locks

The locking mechanism proposed in [Gar83] utilizes two types of locks, called local and global locks. Local locks are used by the mechanism to ensure that each step is executed as atomic action. Similarly, global locks are used by the mechanism to ensure that the interleaving of the atomic steps cannot violate consistency.

Each transaction must first obtain a global lock on an entity x and then a local lock on x before accessing (i.e., reading or writing) x. The local locks obtained during the execution of each step in the transaction will be released when the step finishes. Associated with each global lock on an entity x is an interleaving set denoted ID(x). ID(x) represents the types of all transactions that are currently holding global lock on x. The global lock on x can only be released when all these transactions terminate.

I.2.2. Locking by nonlocal transactions

Before a nonlocal transaction T_i accesses an entity x, it first checks whether x is currently locked. If x is unlocked, then T_i will obtain a global lock on x and will set

ID(x)=CS(t(T_i)) (indicating that x can also be locked by any other transaction that is compatible with T_i). Otherwise, i.e., if x is currently locked, then T_i checks the interleaving set ID(x). If ID(x)=CS(t(T_i)), then T_i can interleave (i.e., will be granted a global lock on x), otherwise, T_i has to wait.

T_i will also need to obtain a local lock on x before accessing x, but this lock will not be granted if there is already a local lock on x (that is not compatible with the lock of T_i).

I.2.3. Locking by local transactions

A local transaction may have more than one interleaving descriptor and, therefore, can choose its interleaving during execution. (Note that a local transaction has only one step which must be executed atomically). Locking by local transactions can be described as follows. Let T_i be a local transaction and let t(T_i)=TP1 and CS(TP1)={{TP1,TP2},{TP1,TP3}}. Transaction T_i must decide which of the two interleavings ({TP1,TP2} or {TP1,TP3}) it will participate in. This decision will be deferred until T_i encounters the first globally locked entity x. If ID(x)={TP1,TP2}, then T_i chooses {TP1,TP2} as its own interleaving (i.e., if later it encounters an entity y such that ID(y)={TP1,TP2}, it must wait). Similarly, if ID(x)={TP1,TP3}, then T_i chooses {TP1,TP3} as its own interleaving.

Before T_i encounters any interleaving, it sets on every entity x to be globally locked by T_i the ID(x)=Φ. When a decision is reached, all these descriptors (Φ) are set to the decided interleaving set. The meaning of setting an ID(x)=Φ by T_i is that T_i will not allow any other (local or nonlocal) transaction to interleave on x.

Theorem I.1: Every schedule produced by the locking mechanism described previously is a semantically consistent schedule [Gar83].

I.3. Transaction failure

Garcia-Molina [Gar83] has proposed the concept of counterstep for handling transaction failure. In order to undo the effect of a certain step S_{ij} a counter step C_{ij} must be executed. Similarly, to undo the effect of the whole transaction T_i, we must undo the effect of each step in T_i.