TOWARDS ADAPTIVE CONCURRENCY CONTROL IN DATABASE SYSTEMS

by

Biaodong Cai

B.Sc. Xidian University, China 1984

M.Sc. Xidian University, China 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
March 1992

ISBN 0-315-83663-6

APPROVAL

Name:	Biaodong Cai
Degree:	Master of Science
Title of thesis:	Towards Adaptive Concurrency Control in Database Systems

Examining Committee: Dr. Ramesh Krishnamurti
Chair

Dr. Tiko Kameda, Senior Supervisor

Dr. Jia-Wei Han, Supervisor

Dr. Peter Triantafillou, External Examiner

Date Approved: March 4, 1992

Abstract

In this thesis, we propose a new approach to adaptive concurrency control for database systems. Unlike previous concurrency control schemes, ours is "data-oriented." We partition a database into two parts, OPT and PES, depending on the conflict rate on each data item. Access to data items in OPT is governed by an optimistic method while access to items in PES is governed by two-phase locking. Therefore, our scheme takes advantage of both optimistic and locking methods. The partition can be changed dynamically to adapt to the changing conflict rates.

Based on a systematic study on optimistic methods, we develop several hybrid concurrency control algorithms that combine optimistic and two-phase locking methods. We also develop a systematic procedure to produce such hybrid algorithms. We design a re-partition algorithm that can be executed concurrently with transaction processing, and a mechanism that traces conflict rate changes. In addition, we propose an implementation for an adaptive concurrency control scheme. We further extend our approach to multi-version database systems.

To my parents, Huizhen, and Xin-Xin.

Acknowledgements

I wish to express my appreciation and gratitude to my supervisor, Dr. Tiko Kameda, for his advice, support, and patience. In addition to being a superb advisor aiding me with the technical contents of this thesis, Dr. Kameda have given me a great help in revisions and the text processing of this thesis. It was his insistence on precision and perfection that ensured the quality of the thesis.

I would like to thank Dr. Jia-Wei Han for his constant encouragement and help during the years of my study at Simon Fraser. He also spent valuable time helping me searching for a thesis topic. I would also like to thank Dr. Peter Triantafillou for his valuable comments on my work. Every discussion with him was pleasent, for I always came back with interesting suggestions.

I am grateful to graduate students and professors in the School of Computing Science. They made my first years in Canada memorable.

Finally, my thanks go to my family. I am indebted to my wife Huizhen for her support, self-restraint, and self-sacrifice. I am also indebted to my parents for their love, education, and understanding.

Contents

Chapter 1

Introduction

In this thesis, we propose a new approach to adaptive concurrency control for database systems. Our approach combines optimistic and two-phase locking schemes, and is dynamically adaptable to system workloads. In addition, our approach allows the database system administrator to modify some parameters during its operation. The major contribution of our work is that it proposes a new view point to study concurrency control. We claim that our approach is "data-oriented," in contrast to the previous studies, which were "transaction-oriented."

In this introductory chapter, we first briefly review the major existing concurrency control schemes. Then, we introduce the new approach by presenting our basic assumptions, describing the way we attack the concurrency control problem, and exhibiting the special features of the new approach. Finally, we present the organization of the thesis.

1.1 Concurrency Control Schemes

Concurrency control for database transactions has been studied for more than fifteen years [14, 18, 32]. There are three major concurrency control schemes currently known: (two-phase) locking [14], time-stamp ordering [30, 31, 33], and optimistic schemes [22]. In this section, we briefly describe the basic concepts of the three schemes. Locking and optimistic schemes will receive a more detailed discussion in Chapters 2 and 3, respectively. Technical terms used here without definition, such as transaction and conflict, will be formally defined in later chapters.

Locking is the most commonly used among the three schemes. In this scheme, a transaction must acquire a lock on a data item in the database before it can access the item. Among locking methods, *two-phase locking* (2PL) [14] is the most important. In 2PL, a transaction cannot acquire any more locks once it releases a lock. So, a transaction has two phases, an expanding phase during which it acquires all the locks it requires, followed by a shrinking phase during which it releases the locks it has acquired. 2PL orders transactions according to the orders of operations in conflict. It is not deadlock-free. In this thesis, "locking scheme" will mean two-phase locking.

Unlike the locking scheme, *time-stamp ordering* (TO) orders transactions by the time points at which transactions start execution. A transaction, when it starts execution, is assigned a unique time-stamp. Attached to each data item are two time-stamps, a read stamp and a write stamp. The read (write) stamp on a data item records the time-stamp of the last transaction that reads (writes) it. When a transaction requests to access a data item, the scheduler first compares the transaction's time-stamp with the time-stamps of that item. If it finds that the item has been accessed by another transaction with a newer time-stamp in a conflicting access mode, the scheduler will abort the requesting transaction. Otherwise, the request is granted. This scheme is deadlock-free.

Both 2PL and TO are *pessimistic* schemes in the sense that they are always prepared for conflicts. They check for conflict for every access request and grant the request only when granting it will not violate serial correctness. The *optimistic* scheme is different from them, for it explicitly assumes that conflicts among transactions are rare. An access request is granted immediately without any conflict checking. Concurrency control is deferred until the end of a transaction, when checking for potential conflicts takes place. If a conflict is detected, one or more transactions are aborted. The optimistic scheme is also deadlock-free.

1.2 Our Approach

Basic Assumptions

Our approach is based on four basic assumptions presented below. The order of the assumptions reflects, to some degree, the relative importance of the assumptions to our approach. They will be discussed in detail in Chapter 4.

It is commonly agreed that, when conflict rates on data are low, optimistic methods perform best, and when conflict rates are medium or high, locking methods perform best. This is our first assumption and is called the *performance assumption.*

It is likely that, in a database, conflicts are distributed unevenly over the data items for a period of time. In other words, the conflict rate varies from data item to data item. We call it the *non-uniform access distribution.*

Given a specific application, the conflict rate, or the relative conflict rate on a data item, may be roughly predictable for a short period of time. The prediction may be based on the experience and knowledge of the database system administrator and the execution histories. The prediction need not be precise, it need only give a general picture about whether the conflict rate is low or not low. The *access locality* is the third assumption of our approach.

The last assumption is that, when the number of data items in a database is sufficiently large, it is very likely that, at any time, there is a large portion of data items in the database on which conflict rates are low enough to make optimistic scheme the best concurrency control scheme for them. We call it the *low conflict rate assumption.*

There will be more discussions on these assumptions.

Concurrency Control System

Based on the above four assumptions, we develop an approach to concurrency control that takes advantage of the assumptions and avoids some shortcomings that the assumptions suggest. Our approach is data-oriented because we take into account the conflict rate on each data item. Actually, the conflict rate on each data item is the deciding factor in choosing a particular concurrency control method for the data item. More specifically, we partition the database into two parts, OPT and PES, where OPT consists of those data items with low conflict rates, and PES consists of those data items with medium or high conflict rates. As suggested by their names, access to data items in OPT is governed by an optimistic method, while access to items in PES is governed by a 2PL method. Therefore, we can take advantage of both optimistic and locking methods.

Conflict rates on data items may change from time to time. Therefore, we should provide mechanisms to re-partition the database so that the partition reflects the up-to-date conflict distribution. We propose a concurrency control system which contains two more functional components in addition to the scheduler. The system is depicted in Fig. 1.1.

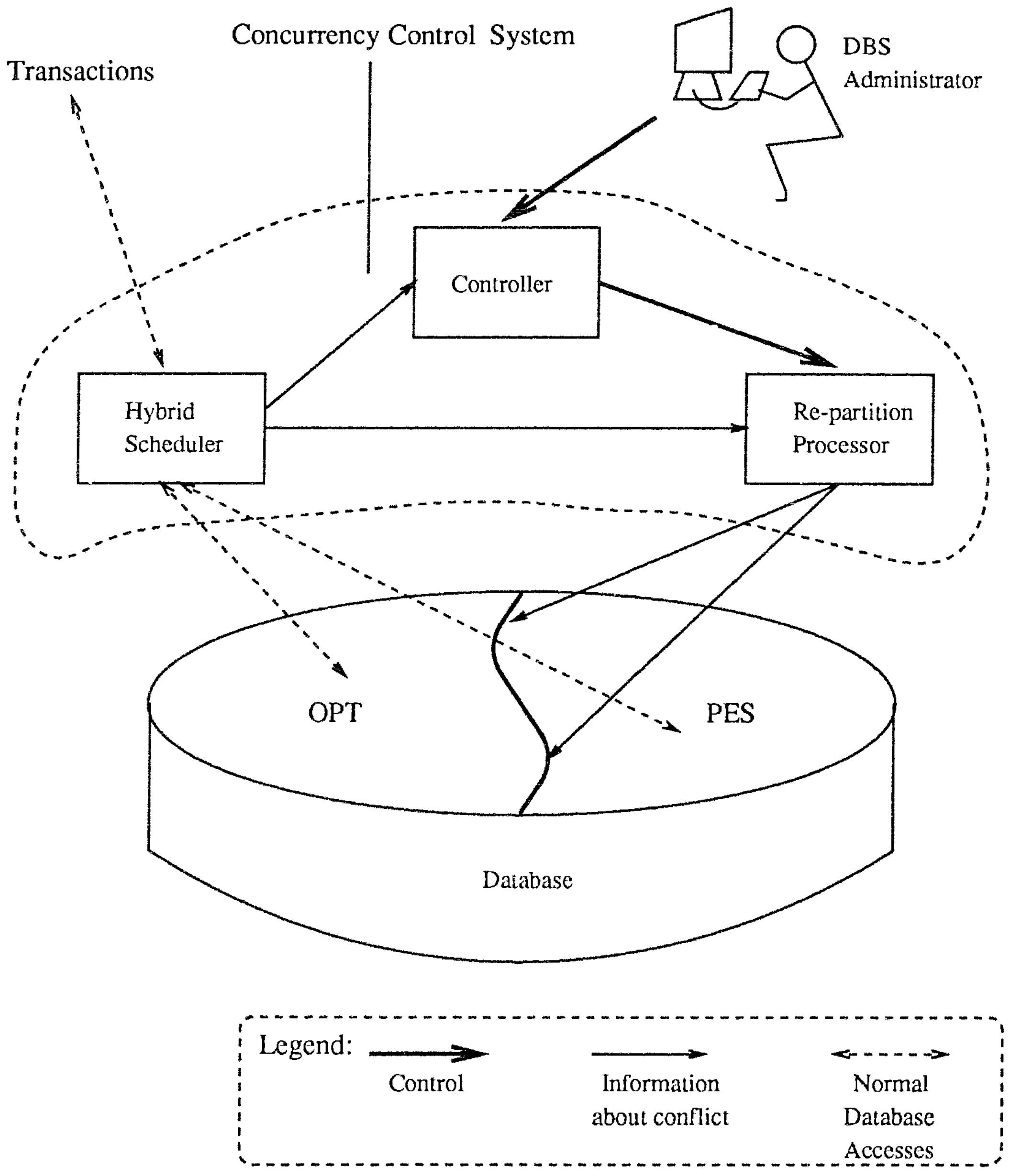


Fig. 1.1 Concurrency Control System

The controller is responsible for keeping the partition up-to-date. It may have functions such as tracing conflict rate changes, making decisions about when a re-partition should start and which items are to be involved in the re-partition. It can accept commands from the database system administrator (DBA), so, the DBA may start a re-partition through the controller. The controller collects conflict rate information from the scheduler and sends re-partition commands to the re-partition processor. The re-partition processor is responsible for re-partitioning. Re-partitioning must be executed concurrently with transaction execution.

Major Problems to be Solved

Corresponding to the three system components, there are three major problems to be solved. In our approach, a transaction may access different data items under different concurrency control methods. Our first problem is how to coordinate optimistic and pessimistic methods, two seemingly conflicting methods, so that serializability is ensured. The second problem is how to dynamically re-partition a database while transactions are being executed. The third problem is how to predict the conflict rate on a data item, and what criteria to use in putting a data item in OPT or PES. These problems will be addressed in Chapters 4, 5, and 6, respectively.

Special Features

Our system has the following special features:

- When the database is properly partitioned, it takes advantage of both optimistic and locking schemes.
- It is adaptive, and is virtually continuously adjustable. The number of different partitions is 2^n, where n is the number of data items. So, when the number of data items is large, the concurrency control policy can be adjusted virtually continuously.
- It provides an interface to the DBA. An important consequence of this is that the DBA's knowledge can be utilized to achieve more efficient concurrency control.

- The hybrid scheduler introduces only a little amount of additional overhead.
- A pure locking or optimistic scheduler can be realized as a special case.

1.3 The Organization of the Thesis

Chapters 2 and 3 provide preliminaries. In Chapter 2, we introduce a concurrency control theory which will be used as the formal framework for the subsequent chapters. In Chapter 2, we also discuss the locking scheme using this framework. Chapter 3 is a systematic study of the optimistic scheme.

Chapters 4 to 7 constitute the main body of this thesis. In Chapter 4, we discuss our motivation, present several hybrid schedulers, and develop a systematic procedure for combining optimistic and locking schemes. In Chapter 5, we develop algorithms for dynamic re-partitioning of a database, and in Chapter 6, we discuss some issues for the controller and propose an automatic controller. In Chapter 7, we propose an implementation for adaptive concurrency control.

Finally, in Chapter 8, we conclude our discussion and point to possible future work.

Chapter 2

Concurrency Control Theory

In this chapter we provide a formal framework for the subsequent chapters. We present models, definitions, and basic theorems of concurrency control theory. The framework is strongly influenced by [6, 7, 9].

Later in this chapter we formally discuss locking methods as an application of the framework. The discussion will also serve as a preliminary to Chapter 4, where we combine locking and optimistic schemes.

2.1 Database Systems

In this section, we present a database system model for the study of concurrency control.

A *database* consists of a set of named *data items*. We denote data items by lower case letters, x, y, z, etc. Each data item has a *value*. The values of the data items at any time comprise the *state* of the database. Among the possible states of a database, there are a set of states that reflect the "correct" information of the application. We call them *consistent states*.

Users access a database by means of transactions. A *transaction* consists of a *Begin* command, followed by a sequence of *Read* and/or *Write* commands, which are followed by an *End*. We use T_1, T_2, $\ldots$, T_i, T_j, $\ldots$ to denote transactions. The *Begin* and *End* commands mark the beginning and end of a transaction. A *Read* command, *Read(x)*, returns the value of data item x in the current database state. A *Write* command, *Write(x, new-value)*, creates a new database state in which x has the value *new-value*. Each transaction represents a self-contained computation. It is assumed to be "correct," i.e., a transaction, when executed alone on a consistent database state, will take the database to a new consistent state. An incomplete execution of a transaction may, however, put the database in an inconsistent state. Therefore, the atomicity of transactions must be ensured, i.e., either its full effects must be reflected in the database or nothing at all.

A transaction, however, *cannot* directly access the database. It only submits, as requests, its commands to a database management module known as the transaction manager (TM), which is part of our database system model.

A *database system* (DBS) contains four components (see Fig. 2.1): a *transaction manager*, a *scheduler*, a *data manager* (DM) and a database. The TM receives commands from transactions and passes them as requests to the scheduler. It also manages private work space for transactions. A scheduler controls the concurrent execution of transactions. It receives *Begin*, *Read*, *Write*, and *End* requests from the TM and issues *dm-read, dm-write, prewrite, abort*, and *commit* operations in responding to the requests. *dm-reads* and *dm-writes* are sent to the DM; *prewrites* and *aborts* are sent to the TM; and *commits* are sent to both the TM and the DM. The DM is responsible for accessing data items in the database. It provides two data manipulation operations: *dm-read(x)*, which reads data item x; and, *dm-write(x, new-value)*, which assigns the value *new-value* to data item x. Note that *dm-write(x, new-value)* is a logical operation recognized by the DM. It does not necessarily mean "write to the database directly." It only makes x's *new-value* generally visible, and eventually stores *new-value* in x in the database. In fact, the DM may first write *new-value* to the cache and then flush it to the database. We assume once a value is written in the cache, it is generally visible.

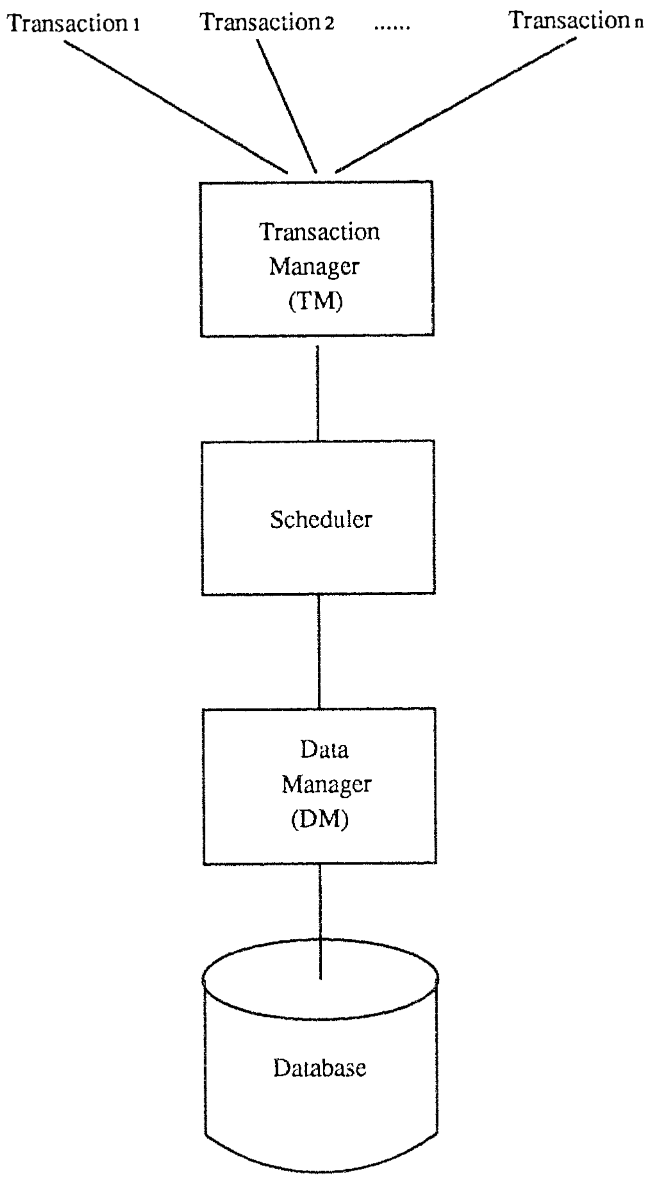


Fig. 2.1 Database System

The actions taken by the DBS upon receipt of the four types of commands, *Begin*, *End*, *Read*, and *Write*, from a transaction are described below:

Begin: The TM assigns a "transaction id," and initializes a private work space for the transaction.

Read(x): If x is already in the transaction's private work space, then its value is returned to the transaction by the TM. Otherwise, the TM passes it as a request to the scheduler which will decide whether to grant it immediately or to delay it. If it is granted, the scheduler issues a *dm-read(x)* operation to the DM. The DM returns the current value of x to the TM, which copies it in the transaction's private work space and gives it to the transaction. If delayed, the request is placed on a queue internal to the scheduler. Later, the scheduler may decide to grant it.

Write(x, new-value): The scheduler first decides if the request should be granted. If so, the TM writes *new-value* in the transaction's private work space by executing *prewrite(x, new-value)*. This has the effect of overwriting the previous value of x in the private work space, if a copy of x exists in the private work space. Otherwise, x is created in the work space with the value *new-value*. Note that it does not alter any values in the database.

End : The scheduler checks whether allowing the transaction to commit (by making its changes permanent in the database) will leave the database in a consistent state. We call this step *validation*. In the event that it will not, the scheduler issues an abort operation. The transaction (maybe some others) will then be aborted. The data structures in the scheduler, such as read-/write-locks and read/write sets (to be introduced in later chapters), for the aborted transaction will be discarded. The private work space for the aborted transaction is cleared. Otherwise, the scheduler issues a dm-write operation for every data item in the transaction's private work space whose value has been created or changed by a prewrite operation. This has the effect of making the last change to x in the private work space visible by other transactions and causing it eventually to become a permanent value in the database. After all dm-writes have been carried out, a commit operation is issued, the private work space is discarded and the transaction is completed. We assume that once a dm-write is issued, its effect will be eventually seen in the database and the transaction cannot be aborted thereafter. Because a partial result of a transaction is not allowed to exist in the database, the DBS ensures that, once a dm-write is issued for a transaction, all the dm-writes for the transaction will eventually be issued. In other words, the transaction will be committed. The mechanism ensuring this property is the recovery mechanism that is beyond the scope of this thesis.

To simplify the discussion of our algorithms, we adopt the following assumption.

Assumption 2.1 *A data item does not exist in a transaction's private work space until and unless it has been read or written explicitly by that transaction.*

2.2 Serializability

Serializability is the most-commonly used correctness criterion. We define serializability in terms of conflicts.

It is dm-reads and dm-writes that actually access the database. Therefore, we model the execution of transactions using dm-reads and dm-writes rather than *Reads* and *Writes*. We refer to dm-reads and dm-writes as *dm-operations*, and simply call them *operations* when it is clear from the context. We say that two dm-operations *conflict* if they are from different transactions,[1] they operate on the same data item, a.'d at least one of them is a dm-write. We also say that two transactions *conflict* with each other if they have operations that conflict with each other.

[1]This condition is just for the convenience of discussion. We do not consider the intra-transaction concurrency control.

A transaction can exist in one of the three states: *active*, *committed(C)*, and *aborted(A)*. Committed and aborted are permanent, stable states. Active is a temporary state. Eventually it will be converted to either committed or aborted. A *transaction*, Ti, is formally modeled as a 3-tuple (O_i, S_i, $<_i$), where

- O_i is the set of all dm-operations issued on behalf of T_i.
- S_i is a set of *synchronization events* which contains a A_i or a C_i, and some other events such as locking and unlocking. The types of events depend on the concurrency control method used.
- $<_i$ is a partial order over $O_i \cup S_i$ such that
  1. all the dm-reads precede all the dm-writes.
  2. if $A_i \in S_i$, then no dm-write belongs to O_i, and for any dm-read r, $r <_i A_i$.
  3. if $C_i \in S_i$, then for any dm-operation $p \in O_i$, $p <_i C_i$.
  4. if $A_i, C_i \notin S_i$, there are only dm-reads in O_i.

The conditions for $<_i$ reflect the discussion of *End* command in the last section. As described in the last section, all the dm-reads precede the validation of a transaction which precedes all the dm-writes. The validation has not been represented in the formal model yet. It should take place where the processing of *End* command starts, and it can be modeled by putting some synchronization events in S_i. When a transaction passes its validation, it is certain that committing the transaction will take the database to a new consistent state. For an active transaction, such certainty does not exist. This modeling of transactions is open-ended. The synchronization event set of a transaction is not completely specified. We have devised the above model because our discussion will refer to different concurrency control schemes and their combinations. We call both an operation and a synchronization event *actions*. We will omit the subscript of O_i, S_i, and $<_i$ when it will cause no confusion.

Let T= $\{T_1 = (O_1, S_1, <_1), \ldots, T_n = (O_n, S_n, <_n)\}$ be a set of transactions. A *history* H of an execution of T is defined as a 3-tuple (O_H, S_H, $<_H$) such that

- $O_H = \bigcup_{i=1}^{n} O_i$
- $S_H = \bigcup_{i=1}^{n} S_i$, and
- $(\bigcup_{i=1}^{n} <_i) \subseteq <_H$, and for every pair of conflicting operations p and q in O_H either $p <_H q$ or $q <_H p$.

As for transactions, the subscript H may by omitted from O_H, S_H, and $<_H$.

A *complete history* is a history with no active transactions. The *commit projection* of a history H, denoted Commit(H), is the history obtained by deleting all the actions of uncommitted (i.e., aborted or active) transactions from H. Clearly, a commit projection of any history is a complete history.

Two histories H and H' are said to be *equivalent* if

- $O_H = O_{H'}$, and
- $p_i <_H q_j$ if and only if $p_i <_{H'} q_j$, where p_i and q_j are any conflicting operations belonging to transaction T_i and T_j, respectively, such that $A_i, A_j \notin H$.

There is no condition on synchronization events for equivalence. So histories produced by schedulers using different concurrency control schemes could be equivalent. This notion of equivalence is the so-called *conflict equivalence*. Two histories are equivalent under this notion if their orders of the conflicting operations are consistent with each other.

A complete history H is *serial* if for every two transactions T_i and T_j that appear in H, either all operations of T_i appear in H before all operations of T_j or vice versa.

A single transaction is assumed correct. By induction, a serial history is also correct, i.e., it will take the database from one consistent state to another consistent state. A concurrent execution of a set of transactions would also be considered correct, if its effect is the same as a serial execution of the same set of transactions. Formally, a history H is *serializable* if Commit(H) is equivalent to a serial history. The task of concurrency control is to schedule concurrent transactions so that the resulting history is serializable.

We can determine whether a history is serializable by analyzing a graph derived from the history called the serialization graph. Let H be a history over T= $\{T_1, \ldots, T_n\}$. The *serialization graph* for H, denoted SG(H), is a directed graph whose nodes are transactions in T that are committed in H and it has an edge $T_i \rightarrow T_j$ if and only if one of T_i's operations precedes and conflicts with one of T_j's operation in H. An edge $T_i \rightarrow T_j$ implies that T_i must appear before T_j in any serial history that is equivalent to Commit(H). If we can find a serial history H_s over the committed transactions in H consistent with all edges in SG(H), then H_s is equivalent to Commit(H), and so H is serializable. As stated in the theorem below, we can do this as long as SG(H) is acyclic.

Theorem 2.1 *[The serializability theorem] [7] A history H is serializable if and only if SG(H) is acyclic.* □

For a serializable history H, the edges in SG(H) represent a partial order over transactions, and any total order on the nodes of SG(H) that is compatible with the partial order is called a *serialization order*.

2.3 The Locking Scheme

In this section, we discuss 2PL using the concurrency control theory just presented.

2PL synchronizes *Reads* and *Writes* by explicitly detecting and preventing conflicts between concurrent operations. It uses two types of locks, read-locks and write-locks, to synchronize the conflicting operations. A read-lock and a write-lock on the same data item conflict with each other. So do a write-lock and another write-lock on the same data item. The ownership of locks is governed by 4 rules:

1. Before reading data item x from the database (more precisely, before a *dm-read(x)* is issued), a transaction must own a read-lock on x. Before writing a new value of x into the database (more precisely, before a dm-write(x) is issued), a transaction must own a write-lock on x.

2. Different transactions cannot simultaneously own conflicting locks on the same data item.

3. A transaction cannot release a lock it owns until the corresponding operation has been processed by the DM.

4. Once a transaction releases a lock, it may not subsequently obtain any more locks on any data item.

The last rule above causes every transaction to obtain locks in two phases. During the growing phase, the transaction obtains locks without releasing any locks. By releasing a lock the transaction enters the shrinking phase. During this phase the transaction releases locks, and, by rule 4, is prohibited from obtaining additional locks. There are two important points: the time at which the transaction has acquired all the locks it requires and the time at which it starts releasing its locks. The former is called the *locked point* (denoted as LP) and the latter is called the *unlocking point* (denoted as UP). When the transaction terminates (commits or aborts), all remaining locks are automatically released.

We show that 2PL is correct in our framework. That is, every history produced by a 2PL scheduler (called a 2PL history) is serializable. To see this, let us study the properties of a 2PL history.

The synchronization event set, S_i, of a transaction, T_i, scheduled by a 2PL scheduler contains the following four types of synchronization events, where x is a data item

$rl_i(x)$— the scheduler sets a read-lock on x on behalf of T_i;
$wl_i(x)$— the scheduler sets a write-lock on x on behalf of T_i;

$ru_i(x)$— the scheduler releases (i.e., unlocks) a read-lock on x on behalf of T_i;

$wu_i(x)$— the scheduler releases a write-lock on x on behalf of T_i.

Let $o_i(x)$ be a dm-operation of T_i on data item x, where o stands for either r or w. By rule 1, T_i must own a lock on x, before it executes $o_i(x)$, i.e., $ol_i(x) < o_i(x)$. By rule 3, T_i cannot release the lock on x before $o_i(x)$ is processed. That means $o_i(x) < ou_i(x)$. Formally we have

Proposition 2.1 *Let H be a 2PL history. If $o_i(x)$ is in Commit(H), then $ol_i(x)$ and $ou_i(x)$ are in Commit(H) and $ol_i(x) < o_i(x) < ou_i(x)$.*

Suppose there are two operations, $p_i(x)$ of T_i and $q_j(x)$ of T_j, that conflict. Thus the locks that correspond to these operations also conflict. By rule 2, T_i and T_j cannot simultaneously own these locks. Therefore, the scheduler must release the lock corresponding to one of the operations before it sets the lock for the other. In terms of the precedence relation $<$, we have either $pu_i(x) < ql_j(x)$ or $qu_j(x) < pl_i(x)$.

Proposition 2.2 *Let H be a 2PL history. If $p_i(x)$ and $q_j(x)$ $(i \neq j)$ are conflicting operations in Commit(H), then either $pu_i(x) < ql_j(x)$ or $qu_j(x) < pl_i(x)$.*

Now let us look at rule 4. It is equivalent to saying that every locking operation of a transaction must precede any unlocking operation of that transaction. In terms of the precedence relation, $pl_i(x) < qu_i(y)$ for all operations p_i and q_i.

Proposition 2.3 *Let H be a 2PL history. For any $pl_i(x)$ and $qu_i(y)$ of T_i, $pl_i(x) < qu_i(y)$ for any data items x and y.*

Now we show that a 2PL history is serializable by showing that its serialization graph is acyclic. Recall that SG(H) has only committed transactions as its nodes.

Lemma 2.1 *Let H be a 2PL history, and suppose $T_i \rightarrow T_j$ is in SG(H). Then, for some data item x and some conflicting operations $p_i(x)$ and $q_j(x)$ in H, $pu_i(x) < ql_j(x)$.*

Proof: Since $T_i \rightarrow T_j$, there must exist conflicting operations $p_i(x)$ and $q_j(x)$ such that $p_i(x) < q_j(x)$. By Proposition 2.1,

1. $pl_i(x) < p_i(x) < pu_i(x)$, and
2. $ql_j(x) < q_j(x) < qu_j(x)$.

By Proposition 2.2, either $pu_i(x) < ql_j(x)$ or $qu_j(x) < pl_i(x)$. In the latter case, by (1) and (2) and transitivity, we would have $q_j(x) < p_i(x)$, which is ruled out. Thus, $pu_i(x) < ql_j(x)$, as desired. □

Corollary 2.1 *Let H be a 2PL history, and let $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ be a path in SG(H), where $n > 1$. Then, for some data items x and y, and some operations $p_1(x)$ and $q_n(y)$ in H, $pu_1(x) < ql_n(y)$.* □

Theorem 2.2 *Every 2PL history H is serializable.*

Proof: Suppose, by way of contradiction, that SG(H) contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Corollary 2.1, for some data items x and y, and some operations $p_1(x)$ and $q_1(y)$ in Commit(H), $pu_1(x) < pl_1(y)$. But this contradicts Proposition 2.3. Thus SG(H) has no cycles and so, by the Serializability Theorem, H is serializable. □

2PL has a well-known drawback of causing deadlocks. We consider that a deadlock happens because a 2PL scheduler fails to schedule concurrent operations geniusly. So, a 2PL scheduler needs strategies for detecting and resolving deadlocks. The overhead incurred in these tasks should be counted as the overhead of 2PL method. It is worth mentioning that, when deadlocks happen, some transactions muxt be aborted. Thus rollbacks cannot be avoided if a 2PL scheduler is used.

Now we present a demonstrative implementation of the 2PL scheme, in order to give the flavor of algorithm descriptions in the subsequent chapters.

- When it receives a *Read(x)* request from transaction T_i, the scheduler does the following:

```
if $x$ is in $T_i$'s private work space %By Assumption 2.1, $x$ was read or
                                        %written by $T_i$. So $x$ is already
                                        %locked on behalf of $T_i$.
  then read $x$ from there
  else if $x$ is write-locked by some other transaction
         then block $T_i$ until the read-lock can be set on $x$
     set read-lock on $x$
     dm-read($x$)
```

- When it receives a *Write(x, new-value)* request from transaction T_i, the scheduler does the following:

```
if $x$ is read- or write-locked by some other transaction
  then block $T_i$ until the write-lock can be set on $x$
set write-lock on $x$
prewrite($x$, new-value)
```

- When it receives an *End* command from transaction T_i, the scheduler does the following:

```
release all the read-locks
```

```
use dm-writes to reflect $T_i$'s updates to the database
release all the write-locks
commit
```

Chapter 3

Optimistic Scheme Revisited

Among the three schemes of concurrency control, it appears that the optimistic scheme has been studied less extensively than the other two. In this chapter we present systematic discussions on optimistic concurrency control. We start with a discussion of the general characteristics of optimistic scheme. We then examine the read phase. All of the optimistic methods to be discussed in this thesis have many features in common in this phase. Next, we discuss the validation-and-write phase, and classify optimistic methods. Finally, each class is studied in detail.

3.1 Principles and Classification

It is claimed in [22] that the locking scheme has the following inherent disadvantages:

- Lock maintenance and deadlock detection incur a substantial overhead, e.g., 10% of the total execution time in System R [17].
- There are no general purpose deadlock-free locking methods that always provide a high degree of concurrency.

- Concurrency is significantly lowered whenever it is necessary to leave some hot-spot data items locked while waiting for a secondary memory access.
- Because of the possibility of failures, a strict two-phase locking protocol [7] has to be applied to ensure recoverability [7], that is, locks have to be kept until the transaction commits.
- Locking may be necessary only in the worst case, that is, in most cases locking is too strong a preventive measure.

The optimistic concurrency control scheme is designed to get rid of the locking overhead. It is optimistic in the sense that it explicitly assumes that conflicts among transactions are rare. Thus it relies for efficiency on the hope that conflicts will not occur frequently. Since no blocking is possible, optimistic methods are deadlock-free. Concurrency control is deferred until the end of a transaction, when some checking for potential conflicts has to take place. If a conflict is detected, a "pessimistic" view is taken: the conflict is resolved by aborting the transaction. Hence, this scheme relies on transaction rollback as a control mechanism.

The execution of a transaction consists of 2 phases: a read phase and a validation-and-write phase[1]. In its *read phase*, a transaction reads data items, performs required computation, and writes new values of data items into the private work space by prewrite operations. When it finishes all its activities and is ready to commit, the transaction issues an *End* request and proceeds to its *validation-and-write phase*. The scheduler checks whether or not the transaction was in conflict with any of the transactions operating concurrently. Since no locks are held, the data item read by a transaction may have been modified by concurrent transactions. If so, some conflict resolution policy has to be applied. If no conflict is detected, the scheduler *reflects* the transaction's modification in the database by executing dm-write operations and then commits the transaction.

[1]In the literature, validation and write are usually two separate phases. For the convenience of discussing combined algorithms, we merge these two phases into one. In Chapter 4, the validation-and-write phase will be subdivided into subphases.

3.1.1 Read Phase

The part of an optimistic scheduler responsible for the read phase of a transaction consists of procedures invoked upon receipt of *Begin*, *Read*, and *Write* requests. The procedures for *Read* and *Write* are the same for all the optimistic methods and are presented here separately from the procedures for the other requests. The procedure for *Begin* varies from method to method and is discussed together with the corresponding validation-and-write phase.

To detect conflicts, an optimistic scheduler maintains two sets: *read-set* (RS_i) and *write-set* (WS_i) for each transaction T_i. Upon receipt of a *Read* or *Write*, the scheduler reacts as follows:

- When it receives a *Read(x)* request from transaction T_i, the scheduler does the following:

```
if $x$ is in $T_i$'s private work space
    then read $x$ from there
    else dm-read($x$)
        $RS_i := RS_i \cup \{x\}$ % See Remark 1.
```

- When it receives a *Write(x, new-value)* request from transaction T_i, the scheduler does the following:

```
$WS_i := WS_i \cup \{x\}$
prewrite($x$, new-value)
```

Remark 1: By Assumption 2.1, x was previously read or written by T_i if x is in T_i's private work space. If x was read, it is already in RS_i. If x was written, this *Read* request will get the value written by T_i itself.

A property of an optimistic scheduler that is worth mentioning here is that for every dm-operation there will be an addition of the data item to the corresponding read-set or write-set.

3.1.2 Validation-and-Write Phase: a Classification

The read phase of a transaction, as just discussed, is fairly unrestricted. A *Read* or *Write* request is immediately processed without any checking for conflicts or delaying. The burden of ensuring serializability is left to the validation-and-write phase. Ensuring serializability involves two distinct tasks: detecting conflicts that may possibly violate serializability, and resolving them if there are any.

Now let us examine the first task, i.e., detecting conflicts. Suppose transaction T_i, with read-set RS_i and write-set WS_i, is at the beginning of its validation-and-write phase. Let T_j be an arbitrary transaction that runs concurrently with T_i, and let RS_j, WS_j be its read- and write-sets, respectively. A straightforward way to detect conflicts involving T_i is to examine $RS_i \cap WS_j$ and $WS_i \cap (RS_j \cup WS_j)$ for every such T_j. A non-empty intersection indicates that conflicts exist between T_i and T_j. However, not every conflict would result in the violation of serializability. In fact, if conflicts occur in an order consistent with the serialization order, then no harm is done. In determining the serialization order, we take advantage of the fact that in every transaction all the dm-reads happen in its read phase and all dm-writes in its validation-and-write phase. Consider the following scenario (Fig. 3.1):

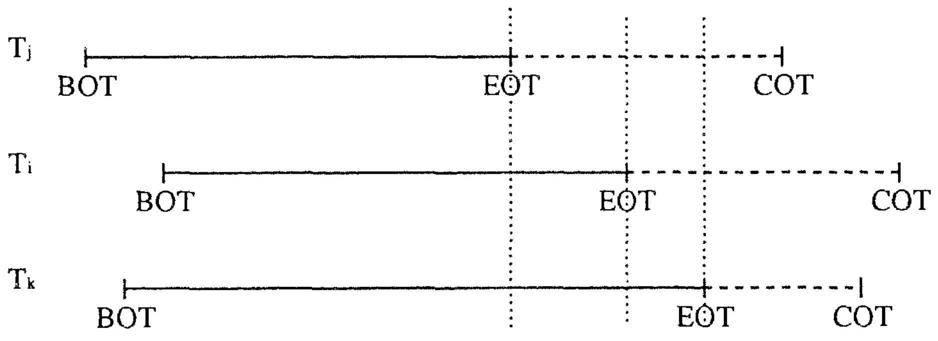


Fig. 3.1

In Fig. 3.1, T_i, T_j, and T_k are transactions, and BOT, EOT COT are synchronization events. BOT marks the beginning of a transaction. It is also the beginning of the read phase of the transaction. EOT marks the end of the read phase and the beginning of the validation-and-write phase. COT marks the completion of the transaction. T_j represents any transaction whose validation-and-write phase overlaps the read phase of T_i, and T_k represents any transaction whose read phase overlaps the validation-and-write phase of T_i. Suppose T_i, T_j, and T_k will all commit eventually. As the serialization order (if any) we use the order in which they come across their EOTs. That is, T_j is ordered before T_i which is ordered before T_k. Imagine the moment when T_i comes across its EOT and enters its validation-and-write phase. Having adopted the above order, we must make sure that there are no conflicting operations $p_i \in O_i$ and $q_j \in O_j$ such that $p_i < q_j$. For this purpose, we need only check for $RS_i \cap WS_j = \phi$ and $WS_i \cap WS_j = \phi$. We need not check $RS_j \cap WS_i = \phi$ because all the dm-reads of T_j precede all the dm-writes of T_i, i.e., the order of any conflicting operations in RS_j and WS_i is consistent with the serialization order of T_i and T_j. Similarly, we need to check $WS_i \cap RS_k = \phi$ and $WS_i \cap WS_k = \phi$ with T_k, but not $RS_i \cap WS_k = \phi$. Further, since every transaction is checked for conflicts with the other concurrent transactions, duplicated checking should be eliminated. T_i, for example, need only be checked for either $RS_i \cap WS_j = \phi$ and $WS_i \cap WS_j = \phi$, or $WS_i \cap RS_k = \phi$ and $WS_i \cap WS_k = \phi$. The former, i.e., checking $RS_i \cap WS_j = \phi$ and $WS_i \cap WS_j = \phi$, is called *backward checking* and the latter is called *forward checking*. The terms forward and backward are from [20]. [22], as well as other papers [3, 23, 12, 13], discusses only backward checking. Checking can be done serially or in parallel. Therefore, there are four combinations of checking strategies: *serial forward*, *serial backward*, *parallel forward*, and *parallel backward*. Backward checking with serial and parallel validation will be discussed in Section 3.2, while forward checking will be discussed in Section 3.3.

Once a conflict that may violate serializability is discovered, resolution is straightforward. Since the conflict has already happened, there exists no alternative but to rollback some involved transactions. This topic will be discussed in detail in the following sections.

3.2 Backward Checking

Starting from this section, we will study optimistic algorithms one by one. Serial and parallel backward checking strategies will be examined in this section. Forward checking will be discussed in the next section. Finally, we compare these algorithms in the last section of this chapter.

3.2.1 Serial Validation

As discussed above, transaction T_i is checked for $RS_i \cap WS_j = \phi$ and $WS_i \cap WS_j = \phi$ in backward checking when it enters its validation-and-write phase, where T_j is any transaction whose validation-and-write phase overlaps T_i's read phase. Meanwhile, another transaction may enter its validation-and-write phase while the validation for T_i is proceeding. We can simplify the problem by the rule that there be at most one transaction being validated at any time. All the validation-and-write phases are therefore executed serially. Hence the name serial validation. As a result, there is no need to check $WS_i \cap WS_j = \phi$, because, according to this rule, all the dm-writes are performed in an order consistent with the serialization order.

We now formally describe concurrency control based on serial validation and backward checking. In Algorithm 3.1 given below, *tnc* is the transaction number counter maintained by the scheduler. committed transaction It is incremented just before it is assigned as the transaction number to a newly committed transaction. We use *tn(T_i)* to denote the transaction number of T_i. The scheduler uses transaction numbers to represent a total order among the transactions it has scheduled. This order is used as the serialization order. Transaction numbers and *tnc* are also used to identify the transactions that should be checked in validating T_i. The transactions that performed dm-writes while T_i was in its read phase are ordered before T_i in the total order, and so must be checked for conflicts with T_i. These transactions have the characteristic that their transaction numbers will be greater than or equal to the value of *tnc* when T_i enters its read phase, but less than or equal to the value of *tnc* when T_i enters its validation-and-write phase. As we will see later, their COTs are between T_i's BOT and EOT.

In this thesis, we assume that immediately after a transaction is aborted, the locks it holds, if any, will be released and its read-/write-sets will be deleted. If it is in a critical section for the transaction, the scheduler will exit from its critical section. And any step in the scheduling algorithm after the point where the transaction is aborted will not be executed by the scheduler. In other words, an abort operation means an exit from the algorithm.

Algorithm 3.1: Serial Backward-Checking Optimistic Algorithm (SBO)

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

```
$start\text{-}tn_i := tnc$ % $start\text{-}tn_i$ will be used to determine
% those transactions involved in $T_i$'s validation.
$RS_i := WS_i := \phi$
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
begin critical section
    $finish\text{-}tn_i := tnc$
    for every $T_j$ such that $start\text{-}tn_i + 1 \leq tn(T_j) \leq finish\text{-}tn_i$ do % See Remark
        if $WS_j \cap RS_i \neq \phi$
            then abort $T_i$and exit
    for every $x \in WS_i$ issue a $dm\text{-}write(x)$          %Reflecting
```

```
        $tnc{:=}tnc+1$
        $tn(T_i){:=}tnc$
    end critical section
    commit
```

Remark: T_j is a transaction that performs dm-write after T_i started but before T_i entered its validation-and-write phase.

Now we want to show that the algorithm SBO is correct, i.e., it produces only serializable histories.

The synchronization event set, S_i, of a transaction T_i scheduled by an optimistic scheduler contains BOT_i, EOT_i, and COT_i that are shown in Fig. 3.1. One important feature of SBO is that, for any two committed transactions T_i and T_j, either $COT_i < EOTj$ or $COTj < EOTi$ holds. We associate BOT_i with "*start-tn*$_i$*:=tnc*," EOT_i with "*finish-tn*$_i$*:=tnc*," and COT_i with "$tn(T_i){:=}tnc$." For the partial order "$<_i$" of T_i, we require that

$$BOT_i <_i r_i(x) <_i EOT_i <_i w_i(y) <_i COT_i, \tag{3.1}$$

where x and y are any data items read and written by T_i, respectively, and $r_i(x)$ and $w_i(y)$ are the corresponding dm-read and dm-write, respectively.

Proposition 3.1 *Let H be a history produced by SBO, T_i be a transaction, and T_j be a committed transaction in H. If $BOT_i < COT_j < EOT_i$, then BSO checks $WS_j \cap RS_i = \phi$ when validating T_i.*

For a pair of committed transactions, the order of their EOTs is consistent with the order of their conflicting operations. This is formulated in the following lemma.

Lemma 3.1 *Let H be a history produced by SBO, and let $p_i(x)$ and $q_j(x)$ $(i \neq j)$ be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$ then $EOT_i < EOT_j$.*

Proof: Since $p_i(x) < q_j(x)$, from relation (3.1), we have $BOT_i < p_i(x) < q_j(x) < COT_j$. Now we examine each type of conflict.

1. $p_i = r_i$, $q_j = w_j$.
   Assume $EOT_j < EOT_i$. Serial validation implies $COT_j < EOT_i$. Also since $BOT_i < COT_j$, it follows from Proposition 3.1 that, during T_i's validation-and-write phase, $WS_j \cap RS_i = \phi$ would have been checked; it was not empty because $x \in RS_i$ due to $p_i(x)$ and $x \in WS_j$ due to $q_j(x)$. Therefore, T_i would have been aborted, a contradiction to the assumption that T_i is committed. Therefore, $EOT_i < EOT_j$, since EOTs are totally ordered by $<$.

2. $p_i = w_i$, $q_j = r_j$.
   By relation (3.1), $EOT_i < p_i(x) < q_j(x) < EOT_j$.

3. $p_i = w_i$, $q_j = w_j$.
   From relation (3.1), we have $EOT_i < p_i(x) < COT_i$ and $EOT_j < q_j(x) < COT_j$. Because $p_i(x) < q_j(x)$, serial validation implies $COT_i < EOT_j$. Therefore, $EOT_i < COT_i < EOT_j$. □

Theorem 3.1 *SBO produces only serializable histories.*

Proof: Immediate from Lemma 3.1, since EOTs are totally ordered by transaction numbers, and the total order can represent a serialization order. □

3.2.2 Parallel Validation

Serial validation is well-suited to the situation where the validation-and-write phase is short compared to the read phase. If it is not, concurrent validation-and-write phases are desired. The algorithm presented below is devised to exploit parallel execution of validation-and-write phases.

Algorithm 3.2, Parallel Backward-Checking Optimistic Algorithm (PBO)

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

```
$start\text{-}tn_i := tnc$
$RS_i := WS_i := \phi$
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
begin critical section
    $finish\text{-}tn_i := tnc$
    $my\text{-}committing := committing$
    $committing := committing \cup \{T_i\}$ % See Remark 1
end critical section
for $T_j$ such that $start\text{-}tn_i + 1 \leq tn(T_j) \leq finish\text{-}tn_i$ do
    if $WS_j \cap RS_i \neq \phi$
      then $committing := committing - \{T_i\}$
           abort and exit
for $T_j \in my\text{-}committing$ do
    if $WS_j \cap (RS_i \cup WS_i) \neq \phi$
      then $committing := committing - \{T_i\}$
           abort and exit
for every $x \in WS_i$ issue a $dm\text{-}write(x)$          %Reflecting
begin critical section
    if $WS_i \neq \phi$
      then $tnc := tnc + 1$
           $tn(T_i) := tnc$ % See Remark 2
    $committing := committing - \{T_i\}$
end critical section.
commit
```

Remark 1: *committing* is the set of transactions which have already started their validation-and-write phase but have not committed yet.
Remark 2: A transaction gets a transaction number only if it updates the database.

As in SBO, *start-tn*$_i$ and *finish-tn*$_i$ are used to determine the transactions that were committed when T_i was in its read phase. That is, those T_j's with $BOT_i < COT_j < EOT_i$, which may conflict with T_i. Proposition 3.1 is also valid for PBO. This time, unlike SBO, validation-and-write phases are not protected by critical sections. They may run in parallel to each other. T_i's validation should also check those transactions which enter their validation-and-write phases before T_i does and execute their validation-and-write phases concurrently with T_i's. In other words, those T_j's such that $EOT_j < EOT_i < COT_j$ are also checked in addition to these checked by SBO. Note that COT_i and COT_j need not be ordered relative to each other. Those transactions are identified by the set *my-committing*. The scheduler maintains a set *committing* which contains all the transactions that are currently in their validation-and-write phases. By copying *committing* set at the beginning of its validation-and-write phase under the protection of critical section, T_i's *my-committing* set contains exactly all the transactions T_j such that $EOT_j < EOT_i < COT_j$. Furthermore, because of concurrent validation-and-write phases, write-write conflicts between T_i and these T_j's must also be checked. Similar to Proposition 3.1, we make the following observation.

Proposition 3.2 *Let H be a history produced by PBO, and let T_i and T_j be transactions in H. If $EOT_j < EOT_i < COT_j$, then PBO checks $WS_j \cap (RS_i \cup WS_i) = \phi$ when validating T_i.*

To show the correctness of PBO, we first prove a lemma analogous to Lemma 3.1.

Lemma 3.2 *Let H be a history produced by PBO, and let $p_i(x)$ and $q_j(x)$ $(i \neq j)$ be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$ then $EOT_i < EOT_j$.*

Proof: Since $p_i(x) < q_j(x)$, from relation (3.1), we have $BOT_i < p_i(x) < q_j(x) < COT_j$. Now we examine each type of conflict.

1. $p_i = r_i$, $q_j = w_j$.
   Assume $EOT_j < EOT_i$. There are two cases: 1, $COT_j < EOT_i$, and 2, $EOT_i < COT_j$. We show that both cases lead to contradictions. In case 1, $BOT_i < COT_j < EOT_i$, by Proposition 3.1, in T_i's validation-and-write phase, $WS_j \cap RS_i = \phi$ would have been checked and it was not satisfied because $x \in RS_i$ due to $p_i(x)$ and $x \in WS_j$ due to $q_j(x)$. Therefore, T_i would have been aborted, a contradiction to the assumption that T_i is committed.

   In case 2, we have $EOT_j < EOT_i < COT_j$. Proposition 3.2 applies. In T_i's validation-and-write phase, $WS_j \cap (RS_i \cup WS_i) = \phi$ would have been checked and it was not empty, because $x \in RS_i$ due to $p_i(x)$ and $x \in WS_j$ due to $q_j(x)$. Therefore, T_i would have been aborted, a contradiction again. Therefore, $EOT_i < EOT_j$, since EOTs are totally ordered.

2. $p_i = w_i$, $q_j = r_j$.
   Proved as in the proof of Lemma 3.1.

3. $p_i = w_i$, $q_j = w_j$.
   From relation (3.1), we have $EOT_i < p_i(x) < q_j(x) < COT_j$. Assume $EOT_j < EOT_i$. Then, by Proposition 3.2, in T_i's validation-and-write phase, $WS_j \cap (RS_i \cup WS_i) = \phi$ would have been checked and it was not empty because $x \in WS_i$ due to $p_i(x)$ and $x \in WS_j$ due to $q_j(x)$. Therefore, T_i would have been aborted, a contradiction to the assumption that T_i is committed. Thus, $EOT_i < EOT_j$ must hold, since EOTs are totally ordered. □

Theorem 3.2 *PBO produces only serializable histories.*

Proof: Immediate from Lemma 3.2, since EOT's are totally ordered. □

3.2.3 Remarks

SBO and PBO presented above first appeared in [22] and were adopted in many subsequent papers. However, checking for $WS_j \cap RS_i = \phi$ in them is more restrictive than necessary. To illustrate this, suppose that there are two transactions T_1 and T_2. T_1 writes x and T_2 reads x. Further, suppose T_1 enters its validation-and-write phase and finally commits when T_2 is in its read phase, i.e., T_1 precedes T_2 in the serialization order (See Fig. 3.2). Consider the following two cases:

1. T_2 reads x (at position A in Fig. 3.2) before the value of x written by T_1 is reflected to the database. It is a violation of serializability and T_2 should be aborted.

2. T_2 reads x (at position B in Fig. 3.2) after the value of x written by T_1 has been reflected to database. This write-read conflict does not violate the serialization order. T_2 need not be aborted.

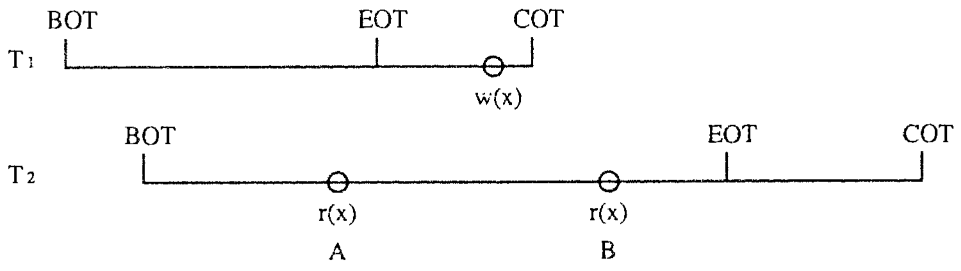


Fig. 3.2 Detecting conflicts

Pradel, et al., suggested a method to relax the restriction [27]. In their suggestion, the read set of a transaction also contains EOT's of other transactions, and its elements are ordered into a the sequence. When a transaction, say T_1 in Fig. 3.2, enters its validation-and-write phase, every other transaction currently in its read phase, say T_2 in Fig. 3.2, records EOT_1 in its read set. When T_2 checks for conflicts with T_1, it need only check the read operations in RS_1 which occur after EOT_1 against T_1's write set.

It is worth mentioning that the forward checking presented below does not have the above restriction.

3.3 Forward Checking

3.3.1 Serial Validation

In backward checking, a transaction is checked for conflicts with other transactions that have entered their validation-and-write phases before it does. As from Theorems 3.1 and 3.2, these transactions appear before T_i in serialization order. In forward checking, a transaction is checked for conflicts with other transactions that will appear after it in the serialization order. As in backward checking, we also use the logical time at which transactions enter their validation-and-write phases to order them. We present a serial forward checking validation algorithm below. The algorithm is straightforward compared to SBO and PBO. The scheduler maintains a set *Active* to record the transactions that are currently in their read phases. Since the transaction (T_i) in the validation-and-write phase is currently writing, it may have write-read conflict with any of the transactions in *Active*. Unlike the critical sections used in backward checking, the critical section used in the algorithm below is a system-wide critical section. The purpose of using such a strong critical section is to prevent the read set of any other transaction from being updated while the validation and reflecting of a transaction is taking place. Later, we will discuss some methods to relax this restriction.

Algorithm 3.3, Serial Forward-Checking Optimistic Algorithm (SFO)

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

```
$Active := Active \cup \{T_i\}$
$RS_i := WS_i := \phi$
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
begin (system-wide) critical section
    $Active := Active - \{T_i\}$
    $conflict$:=false
    for every $T_j \in Active$ do
        if $WS_i \cap RS_j \neq \phi$ then $conflict$:=true
    if $conflict$ then resolve the conflict by aborting either $T_i$ (and exit)
                       or all $T_j$'s such that $WS_i \cap RS_j \neq \phi$.
                       % The decision is made upon some cost criteria.
    for every $x \in WS_i$ issue a $dm\text{-}write(x)$         %Reflecting
end critical section
commit.
```

We associate BOT_i with "$Active := Active \cup \{T_i\}$," EOT_i with "$Active := Active - \{T_i\}$," and $COT_i$with the end of the critical section. Transaction T_j with $BOT_j < EOT_i$ and $EOT_i < EOT_j$,if it has EOT_j) will be in *Active* when transaction T_i is being validated, and for each $T_j \in Active$, SFO checks $WS_i \cap RS_j = \phi$ when validating T_i.

The correctness proof for SFO is similar to that for SBO.

Lemma 3.3 *Let H be a history produced by SFO, and let $p_i(x)$ and $q_j(x)$ $(i \neq j)$ be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$ then $EOT_i < EOT_j$.*

Proof: We only show that, if $r_i(x) < w_j(x)$, then $EOT_i < EOT_j$. The rest of the proof is similar to Lemma 3.1.

Since $r_i(x) < w_j(x)$, by relation (3.1), we have $BOT_i < r_i(x) < w_j(x) < COT_j$. Also, we have $EOT_j < w_j(x) < COT_j$. Since EOT_j and COT_j form the boundary of the critical section for T_j's validation-and-write phase, $r_i(x)$ can only occur before EOT_j, i.e., $r_i(x) < EOT_j$. So we have $BOT_i < r_i(x) < EOT_j$. Assume $EOT_j < EOT_i$. Since $BOT_i < EOT_j < EOT_i$, in T_j's validation-and-write phase, SFO would have checked $WS_j \cap RS_i = \phi$ and either T_i or T_j would have been aborted because $x \in (WS_j \cap RS_i)$, a contradiction. So, $EOT_i < EOT_j$. □

Theorem 3.3 *SFO produces only serializable histories.*

Proof: Immediate from Lemma 3.3, since EOT's are totally ordered. □

3.3.2 Replacing System-Wide Critical Sections

Imposing a system-wide critical section for a validation-and-write phase may not be acceptable, and should be avoided. Actually, the purpose of using system-wide critical section, instead of a simple critical section, is to prevent the situation where, after $WS_i \cap RS_j = \phi$ is checked for some T_j,[2] T_j reads a data item in WS_i before the new-value of the item is reflected on behalf of T_i. So, we can replace the system-wide critical section by an ordinary critical section (as used in backward checking) with some additional facility. The following are some alternatives.

1. After ensuring that $WS_i \cap RS_j = \phi$ is satisfied, block all the read requests from T_j until the validation-and-write phase of T_i ends.

2. Lock all the data items in WS_i at the beginning of the validation-and-write phase of T_i until the validation-and-write phase ends.

[2]We assume this checking is executed atomically.

3. From the beginning of the validation-and-write phase of T_i, collect all the items being read into a special read set RV in addition to adding them into the corresponding read sets. After T_i's modification is reflected, check RV against WS_i. If T_j has contributed an item in $WS_i \cap RV$, then abort T_j.

Alternatives 1 and 2 use locking to solve the problem. However, the duration of locking is short and it causes no deadlock problem. Both methods are pessimistic, in expecting that conflict may happen during validation-and-write phase. Alternative 3 is optimistic: "I did not meet any problem in my read phase, why should I worry about validation-and-write phase?"

3.3.3 Parallel Validation

Parallelizing forward checking is very difficult, though possible. This is easy to see from the following discussion.

In parallel *backward* checking, the set of the transactions that should be checked is fixed. Every transaction involved in validation has at least finished its read phase. Thus its read-set and write-set are also fixed. Therefore, backward checking examines "static" data. In contrast, the read sets and write sets involved in parallel *forward* checking may be changing, except those of the transaction being validated. Suppose we are validating T_i. Since all the other transactions involved in the validation could still be in their read phases, their read-sets might still be expanding during the validation. Because other transactions may enter their own validation-and-write phases during T_i's validation, the set of transactions whose write-sets should be checked for validating T_i is also expanding. To make things worse, a new transaction may start at any time during the T_i's validation-and-write phase. One can immediately see the difficulty of the validation.

One possible approach to parallel forward validation is as follows. The scheduler performs validation incrementally through several rounds of checking. In the first round which starts right after EOT_i, the scheduler checks T_i's write-set against the current read-set of every transaction which is in its read phase at the moment when EOT_i happens. Meanwhile, the scheduler collects all the information concerning the read operations that occur in the first round and all the transactions that start their validation-and-write phases at this time. If any conflict is detected, either T_i or all the transactions that conflict with T_i are aborted. If T_i survives from this round, it proceeds to the second round. In the second round, the scheduler uses the incremental information to validate T_i while, at the same time, collecting increments during this round which will be used in the third round. This time, not only read-write conflicts but also write-write conflicts are checked. Again, if T_i survives, it proceeds to the third round, and so on. In the last round, we have to protect the whole round in a global critical section to finish the validation. Reflecting T_i's modification to database can be done in the last round or the second last round. But once it is done, T_i can not be aborted.

Concurrency gained from parallelism is related to the number of rounds. However, on the assumption that conflicts are rare, there is probably no need to have more than two rounds. When there is only one round, it becomes serial validation. A scheduler based on parallel forward validation could be complex. The overhead of running it may offset the benefit of parallelism. We will not discuss this class any further.

3.4 Comparison

In this section we compare the three algorithms presented in this chapter, i.e., SBO, PBO, and SFO. Our comparison focuses on the differences between backward checking and forward checking. Differences between serial and parallel validation are fairly easy to see.

Difference 1 Forward checking resolves conflicts more flexibly than backward checking does. When a transaction, say T_i, discovers conflicts, forward checking can abort either T_i or the transactions which conflict with T_i, while backward checking has no choice but to abort T_i. This is because, in backward checking, all the transactions that are checked for conflicts with T_i either have already committed or may have started dm-writing. Besides, the transactions aborted by forward checking are still in their read phase. Some may have just started their execution. On the other hand, the transactions aborted by backward checking have already successfully finished their read phases. This implies that the abortions in forward checking are in general less expensive than those in backward checking.

Because of this difference, backward checking suffers from starvation problem but forward checking does not. This is important for long transactions, for, in backward checking, a long transaction may often starve.

Difference 2 Forward checking needs less checks than backward checking. Forward checking checks a write-set against a number, say N_f, of read-sets, while backward checking checks a read-set against a number, say N_b, of write-sets. There are three points to be noted here. 1) A transaction's write-set is often smaller than its read-set. Also, it is often that a transaction has an empty write-set, and therefore, does not have validation-and-write phase in forward checking. However, it is seldom the case that a transaction has an empty read-set. 2) The read-sets checked in forward validation are partial, because the transactions they belong to are still in their read phase. On the other hand, all the sets checked in backward checking are complete. They belong to some finished transactions. 3) N_f is limited. It is the number of "active" transactions in the system at the moment when the transaction in question entered its validation-and-write phase. It does not depend on the length of the transaction. N_b, on the other hand, depends on the length of the read phase of the transaction being validated. To make it clear, suppose the system can have at most m transactions executing simultaneously, and suppose that a transaction's read phase is n times longer than the average life-time of transactions. In forward checking, at most $m-1$ read-sets are checked, while in backward checking, there may be $(n+1)(m-1)$

write-sets to be checked.

Difference 3 Unlike forward checking, backward checking has to store the write-sets of committed transactions as long as there is an uncommitted transaction which was started before they were committed. When some transactions are long, this requires significant amount of storage space. On the other hand, forward checking have to deal with the problem of dynamic read sets.

Difference 4 Backward checking allows parallel validation, which is a very important advantage that forward checking does not have. Dm-writing to stable storage is sometimes time-consuming. Parallelization of writes is important for performance.

Chapter 4

Combining Optimistic and Locking Schemes

From this chapter on, we combine the locking and optimistic schemes, taking advantage of both schemes. We start with a discussion, motivating adaptive concurrency control algorithms. We then briefly survey the existing combined algorithms. Unfortunately, about half of these algorithms are not adaptive, and the rest of them are only adaptive to a very limited extent. Next, the motivation behind our approach is presented, based on the analysis of previous algorithms. An algorithm is presented to illustrate our approach, followed by a systematic procedure for combining locking and optimistic methods. Then, we discuss our algorithms in detail. Finally, we extend our approach to multiversion databases.

4.1 Introduction

4.1.1 Motivation

At system design time, a concurrency control algorithm (typically a 2PL variant) is adopted. This design decision may be made based on some *a priori* knowledge of the expected use of the system or simply because the algorithm may appear to be the best. Due to the complicated structure of a database management system, it is unlikely that the original algorithm incorporated into the system will ever be changed, despite the fact that the system may be used under changing workload conditions.

Studies has been done to compare several different algorithms in an attempt to reach some conclusion concerning their operational merits (e.g., [4, 8, 11, 24, 25]). Naturally, if a clear-cut conclusion could be drawn about one algorithm being the "best" under almost all conditions, then that algorithm should be employed by all database systems. However, as commented by Agrawal. et al. [4], the studies have tended to be contradictory, rather than being definitive. A common conclusion suggests that, while locking normally performs well (especially when conflict rates are medium or high), an optimistic method performs better when conflict rates are low. Anyway, the past studies are by no means the last words, since the simulation studies were not performed under a sufficiently wide variety of system workloads and parameters. Also, due to the changing application areas (e.g., artificial intelligence) the usefulness of some characterizations of workloads and parameters in these studies may be short-lived.

It is our opinion that the concurrency control module in a database management system should be a versatile piece of software that has the ability to adapt itself to the changing system workload and environment.

4.1.2 Hybrid Concurrency Control methods: A Survey

Combining different concurrency control schemes in one database management system has been investigated in [6, 9, 10, 15, 16, 23]. The motivation behind each such attempt is either achieving more concurrency or making the scheme more adaptive, i.e., getting better performance in different situations, or both. Interestingly, all the combined schemes proposed so far integrate locking with one of the two other schemes, i.e., time-stamp ordering or optimistic schemes. This is probably because locking is easy to understand and easy to implement.

In [6], Bernstein and Goodman suggest a systematic way to combine different concurrency control methods. They decompose the concurrency control problem into two subproblems: synchronization of read-write conflicts and synchronization of write-write conflicts. Different methods are used to synchronize these two types of conflicts, and some technique is used to integrate the two parts. For example, they discuss an algorithm that uses 2PL for read-write synchronization and TO for write-write synchronization, as well as an algorithm that uses TO for read-write synchronization and 2PL for write-write synchronization. They confine their discussion on this issue to locking and time-stamp ordering schemes. The algorithms they suggest can probably enhance concurrency to some degree, but are not adaptive.

Farrag and Ozsu [15, 16] suggest another way to combine locking and time-stamp ordering schemes. They use an integer L, called the strictness level. The set of transactions is divided into groups, each containing at most L transactions. For the intra-group conflicts (involving transactions within the same group), 2PL is used. For the inter-group conflicts (involving transactions from different groups) TO is used. No characteristics other than the arriving time are taken into account in deciding the membership of a transaction in a group. A transaction is simply put into the newest group. When the number of the transactions in this group reaches L, a new group is created to accommodate new transactions. When L is set to infinity, the scheduler is purely 2PL. When L is set to 1, it becomes purely TO. It is hard to see the advantages of this approach, except that changing the value of L can offer some flexibility.

Combining the locking and optimistic schemes was proposed by Lausen [23]. In his approach, a transaction can execute in one of two modes: optimistic, where the optimistic concurrency control principle applies, and locking, where the 2PL principle applies. An interesting application of his approach is that one can start a transaction in an optimistic view that it won't conflict with other concurrent transactions. When conflicts actually occur and the transaction is aborted, the view turns to pessimistic. When the transaction is restarted, it operates in the locking mode.

Another approach combining the locking and optimistic schemes is suggested by Boral and Gold [9]. They adopt Bernstein and Goodman's idea of decomposing concurrency control to read-write synchronization and write-write synchronization. However, they use the serialization graph to detect conflicts. Their approach can only be used in centralized systems, and the overhead for detecting conflicts is likely to be high.

Canning, Muthuvelraj, and Sieg [10] extend Lausen's approach, trying to design a more adaptive concurrency control algorithm. They group transactions into clusters. The transactions having data contention with each other are grouped into the same cluster. A cluster could have a status of optimistic, pessimistic, or something intermediate. Further, there is a threshold on the number of transactions in a cluster. When the number of transactions in a cluster exceeds the threshold, the cluster changes to an intermediate status tending to pessimistic. We think their approach is a poor extension of [23]. The maintenance and merging of clusters incurs significant cost relative to a possible gain in concurrency.

The combined algorithms surveyed above approach concurrency control from two different points of view.

1. Decompose concurrency control into synchronization of different types of conflicts. As in [6, 9], concurrency control is decomposed to read-write synchronization and write-write synchronization. One concurrency control method is applied to read-write synchronization and the other method to write-write synchronization. In

[15, 16], conflicts within a group are processed using one method and conflicts between groups are handled using the other method. So within one transaction, two methods work together to ensure serializability. Only [9] is adaptive. It is sensitive to the classes of transactions, but not to the change in the conflict rate.

2. Allow transactions using different concurrency control methods to run together. As in [10, 23], a transaction may run in either optimistic or pessimistic mode. Only one method applies to a given transaction. Some mechanism is used to coordinate the transactions. These algorithms are sensitive to the change of conflict rates in the whole database to some extent. However, they do not take into account the distribution of conflict rates over the set of data items. Also it is not sensitive to the classes of transactions.

4.2 Our Approach

The two points of view summaried at the end of the previous section focus on transactions, and do not take differences between data items into consideration. A unique concurrency control policy applies to all the data items. We call them *transaction-oriented*. However, conflicts occur on data. It is the contention on data that generates conflicts. And it is the access to data that determines the classes of transactions. Therefore, our approach focuses on properties of data. To illustrate the significance of shifting our focus in concurrency control onto properties of data, let us consider the following scenario:

> Suppose that a database consists of two disjoint sets of data items, say OPT and PES. Originally, the conflict rates on the data in OPT are low, while the conflict rates on the data in PES are high. We define *conflict rate on a data item* as the number of harmful conflicts on the item in unit time, where a *harmful conflict* is a conflict which may violate serializability. The precise definition of *harmful conflict* depends on the concurrency control

algorithm used. It will receive more discussion in Section 6.1.1. Suppose further that 2PL is used for concurrency control. A transaction, say T_i, holds some write locks on some data items in OPT and now wants some more locks on data items in PES. The transaction may wait for a long time to get all these locks. By the 2PL rule, it won't release any locks on OPT items until and unless it gets all the locks (if no transaction is rolled back due to deadlock). Some other transactions which want to access OPT items write-locked by T_i have to wait until T_i releases the locks on them. These transactions may also hold some locks on OPT items, which in turn will block some more transactions. Consequently, the conflict rate on OPT items may get higher and higher. We name this phenomenon *conflict escalation.*

Conflict escalation occurs when conflict distribution is uneven across the set of data items. Uneven conflict distribution is caused by non-uniform data accesses by transactions, which seem to be very common in practice. This is a fact overlooked by many performance studies on concurrency control. Also, it is easy to see that none of the "pure" (i.e., not combined) concurrency control methods works well under this circumstance. This also argues for adaptive algorithms. Back to the discussion about the significance of a data-oriented approach, it is hard for the transaction-oriented approaches to deal with uneven conflict distribution, because they cannot take conflict distribution into account. Semantics-based locking approaches [5, 29, 34, 35] could not consider this fact either, since their primary emphasis is on reducing conflict between operations by giving more semantic information about the data objects and the operations on them. If we want to face the problem caused by an uneven distribution of conflicts, we have to make more effort on data grouping than they did. For example, if we did not let T_i lock OPT data items but somehow still ensured serializability, we could avoid escalating conflict rates in OPT.

We should emphasize here that our approach is not merely for solving the uneven distribution problem. It is also an approach towards more adaptive concurrency control in general. We now elaborate on this point in some detail. First, based on the following arguments, we expect that OPT occupies a large portion of the database. Assuming that the granularity of data items chosen for concurrency control is not too big, say, at the record or page level, the number of the data items in the database will be sufficiently large. In this case, given a period of time, it is likely that only a very small portion of the database is subjected to frequent conflicts. Meanwhile, most of the data items have low conflict rates and many are even not accessed at all. Further, it may be usual that all the data items in the database have low conflict rates for some period of time. This is particularly likely in large databases. Even though many studies conclude that optimistic methods perform very well when conflict rates are low, one possible reason that they are not used widely in practice is that in many applications, where high conflicts do occur occasionally, optimistic methods perform poorly. With the idea of adaptive concurrency control, we can use an optimistic method in OPT and a locking method in PES. We can expand OPT or even let it take over the entire database when conflict rates are low for all data items; when conflict rates become higher, we just shrink OPT, even until it disappears.

In a particular application of a database management system, one may be able to predict an approximate conflict distribution or at least predict an approximate distribution of update operations for a certain period of time. For example, in a banking database system, a personal saving account may be updated at most once a day on average, but some internal variables, such as the total amount of money in a branch, will be frequently updated. In some databases, there could be some documentary data, such as employees' names, which remain almost unchanged once they are stored. It is also possible that the changes in conflict distribution is predictable. For example, more conflicts could occur during the day than at night; some data in a commercial database could have higher conflict rates at the end of a month than at other times. Most importantly, prediction could often be based on the history of a system. Since the conflict rate on a specific data item probably does not change very fast, one may obtain a good approximation to the conflict rate on that data item for a coming short period of time from the most recent history. In summary, prediction of conflict distribution with certain precision might be possible, and it is better than nothing for an adaptive concurrency control algorithm.

Now assume that a database is partitioned into OPT and PES, where the conflict rates on the data items in OPT (conflicts in OPT, for short) are low and the conflicts in PES are medium or high. Our approach is to choose a concurrency control method best suited to the conflict rates for each part of the database, say, an optimistic method for OPT and a locking method for PES. When a transaction accesses a data item in a part, it obeys the concurrency control rules governing that part. The transaction, therefore, may be managed by more than one subscheduler enforcing different rules. If the subschedulers can coordinate with each other to ensure serializability with small overhead, we can take advantage of various concurrency control methods.

With changes in conflict rates, a partition (into OPT and PES) may become out-of-date. Therefore, we should have a mechanism to keep the partition up-to-date. Its functions were briefly introduced in Chapter 1 and will be discussed in more detail in subsequent chapters. Now consider that concurrency control is characterized by the proportion of optimistic accesses over pessimistic accesses. When we change the membership of a data item from OPT to PES, or vice versa, we adjust the control a little bit. When the number of data items is large, such a change is so little from the global point of view that the control seems to be continuously tunable. This is a unique feature of our approach. We can even imagine such a scenario as the following:

> At 8:00 AM, a transaction has 89% of its accesses controlled optimistically and 11% pessimistically. At 2:00PM, the same transaction may have 73% of its accesses controlled optimistically and 27% pessimistically.

In the remainder of this chapter, we integrate different subschedulers.

4.3 A Combined Algorithm – An Example

To flesh out our ideas about data-oriented approach to concurrency control in concrete algorithms, we show how to integrate parallel backward checking (PBO) with two-phase locking. We first present the algorithm, and then show that straightforward composition can produce a correct algorithm. The proof will also give us hints on a systematic way to integrate optimistic methods with locking.

4.3.1 The Algorithm

As stated in the last section, we partition a database into two parts, OPT and PES. It is assumed that conflicts due to data items in OPT are infrequent and those due to data items in PES are not infrequent. We also assume that there exists an efficient method for the scheduler to determine if a data item is in OPT or PES. We leave suggestions for specific methods to a later chapter. We use WL_i (Write-Locked items) to denote the set of data items in PES written by T_i.

The following is a description of our algorithm:

Algorithm 4.1: (PBO + 2PL)

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

  $start\text{-}tn_i{:=}tnc$

  $RS_i := WS_i := WL_i := \phi;$

- When it receives a *Read(x)* request from transaction T_i, the scheduler does the following:

  check-member(x) % determine which part of database x belongs to

```
case R1: $x \in PES$ % Using locking in this case
    if $x$ is in $T_i$'s private work space
        then read $x$ from there
    else if $x$ is already write-locked by some other transaction,
            block $T_i$ until read-lock can be set on $x$
        set read-lock on $x$
        $dm\text{-}read(x)$
case R2: $x \in OPT$ % Using optimistic control in this case
    if $x$ is in $T_i$'s private work space
        then read $x$ from there
    else $dm\text{-}read(x)$
        $RS_i := RS_i \cup \{x\}$
```

- When it receives a *Write(x, new-value)* request from transaction T_i, the scheduler does the following:

```
check-member($x$)
case W1: $x \in PES$
    if $x$ is read- or write-locked by some other transaction,
        block $T_i$ until write-lock can be set on $x$.
    set write-lock on $x$
    $WL_i := WL_i \cup \{x\}$
    prewrite($x$, new-value)
case W2: $x \in OPT$
    $WS_i := WS_i \cup \{x\}$
    prewrite($x$, new-value)
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
begin critical section
```

```
    $finish\text{-}tn_i := tnc$
    $my\text{-}committing := committing$
    $committing := committing \cup \{T_i\}$
end critical section
release read-locks
for all $T_j$ such that $start\text{-}tn_i + 1 \leq tn(T_j) \leq finish\text{-}tn_i$ do
    if $WS_j \cap RS_i \neq \phi$
      then $committing := committing - \{T_i\}$
           abort $T_i$
for $T_j \in my\text{-}committing$ do
    if $WS_j \cap (RS_i \cup WS_i) \neq \phi$
      then $committing := committing - \{T_i\}$
           abort $T_i$
for every $x \in (WL_i \cup WS_i)$ issue a $dm\text{-}write(x)$
           % Reflecting. $(WL_i \cup WS_i)$ contains
           % all the items updated by $T_i$.
begin critical section
    if $WS_i \neq \phi$
      then $tnc := tnc + 1$
           $tn(T_i) := tnc$
    $committing := committing - \{T_i\}$
end critical section.
release write-locks
commit
```

One can see that the composition is quite straightforward. In face, it is almost just gluing two algorithms together. For a *Read* or *Write* request, the only additional step is checking (using *check-member(x)*) if x is in OPT or PES when it arrives. Then it follows optimistic or locking steps, depending on which part x belongs to. The procedure for *End* is similar to that in PBO, except that "release read-locks" and "release write-locks" are inserted at appropriate places. However, it is the positions of these steps that play a vital role in making Algorithm 4.1 correct. Unlike [23], there is no checking for conflicts between an optimistic read and a pessimistic write and the like. The simplicity can be attributed to the clear separation of OPT and PES.

Actually, we need not use prewrites for PES part. We can directly use dm-writes in PES, and therefore, do not need any private work space for PES. There are several impacts of this improvement. First, the buffering problem of optimistic scheme is eased. Second, the write subphase is considerably shortened. Third, the negative effect is that recovery is more costly. Since a transaction may be aborted due to conflicts in OPT, recovery problem could be serious.

4.3.2 Correctness

We can think of a transaction scheduled by Algorithm 4.1 as consisting of two phases: a read phase and a validation-and-write phase, separated by an *End* request. Let BOT, EOT, and COT stand for the same points (respectively) as that in PBO in Chapter 3. The synchronization event set of a transaction contains these events and the locking and unlocking events. As in Chapter 3, our correctness proof will show that if $p_i(x) < q_j(x)$ for any pair of conflicting operations, $p_i(x)$ and $q_j(x)$, then $EOT_i < EOT_j$ holds. This constitutes a proof since EOT's are totally ordered and their order can be considered as the serialization order. We show this fact first for any x in PES, and then for any x in OPT. Since Algorithm 4.1 is a straightforward combination of 2PL and PBO, the related proofs we used in Chapters 2 and 3 carry over.

The following proposition formulates the relation of EOT to locking and unlocking operations.

Proposition 4.1 *Let H be a history produced by Algorithm 4.1. Let T_i be a transaction in H and $o_i(x)$ be an operation in T_i, where $x \in PES$. If both $ol_i(x)$ and $ou_i(x)$ appear in H, then $ol_i(x) < EOT_i < ou_i(x)$.* □

We first deal with the case $x \in$ PES.

Lemma 4.1 *Let H be a history produced by Algorithm 4.1, and let $p_i(x)$ and $q_j(x)$ ($i \neq j, x \in PES$) be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$ then $EOT_i < EOTj$.*

Proof: From the proof of Lemma 2.1, if $p_i(x) < q_j(x)$ then $pu_i(x) < ql_j(x)$. Therefore, by Proposition 4.1, $EOT_i < pu_i(x) < ql_j(x) < EOTj$. □

Next, we consider the case $x \in$ OPT. In this case, Lemma 3.2 is applicable.

Lemma 4.2 *Let H be a history produced by Algorithm 4.1, and let $p_i(x)$ and $q_j(x)$ ($i \neq j, x \in OPT$) be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$ then $EOT_i < EOTj$.* □

From the above two lemmas, the correctness of Algorithm 4.1 follows immediately.

Theorem 4.1 *Algorithm 4.1 produces only serializable histories.* □

4.4 Systematic Procedure

In all the algorithms we have presented in Chapters 3 and 4 so far, the key to ensuring serializability is to arrange conflicting operations in consistence with the total order defined by the EOTs. We now consider a more general problem. Suppose a database is partitioned as before. In accessing OPT, an optimistic method is used, while in accessing PES, 2PL is used. We call a scheduler with this characteristic an O+P scheduler. The problem is how to combine the two methods so that serializability is guaranteed. To attack this problem, we consider that on each part of the database there is a *subscheduler* performing concurrency control. For example, in Algorithm 4.1, we consider PBO as the optimistic subscheduler and Strict 2PL as the locking subscheduler. We first investigate the properties of each kind of subscheduler.

4.4.1 Confining Sections

As was discussed in Chapter 3, an optimistic scheduler uses the time order of EOTs as the serialization order. From now on, the term "optimistic scheduler" refers to a scheduler using SBO, PBO, or SFO. In all of these algorithms, when a transaction, say T_i, enters its validation-and-write phase, it first enters a subphase in which the set of transactions involved in the validation is determined. In SBO, this subphase involves only one operation "*finish-tn*$_i$:=*tnc*," which assigns the value of *tnc* at EOT_i to *finish-tn*$_i$. In PBO, *tnc* is recorded and *committing* is recorded and updated in the subphase. In SFO, the subphase consists of "*Active*:=*Active*$-\{T_i\}$," which defines the set of transactions that are in their read phases at EOT_i. We call this subphase *confining section* (CS). The beginning of the CS for transaction T_i is marked by EOT_i. We use ECS_i to mark the end of CS_i. Since some global information is updated in CS's, CS's should be executed mutually exclusively. We extend the use of partial order "<" to the confining sections. $CS_i < CS_j$ means that the entire confining section of T_i precedes the confining section of T_j, in other words $ECS_i < EOT_j$. Here we consider ECS_i as a synchronization event. Let S_1 and S_2 be sections (i.e., intervals) such as confining sections and locked sections to be introduced later. Let B_1, B_2, and E_1, E_2 be the beginnings and ends of S_1 and S_2, respectively. We say S_1 and S_2 *overlap* if $B_1 < E_2$ and $B_2 < E_1$ (see Fig. 4.1). Our optimistic schedulers ensure that no two confining sections overlap, i.e.,

Proposition 4.2 *Optimistic schedulers SBO, PBO, and SFO enforce, either $CS_i < CS_j$ or $CS_j < CS_i$, for any transactions T_i and T_j ($i \neq j$) that have confining sections.*

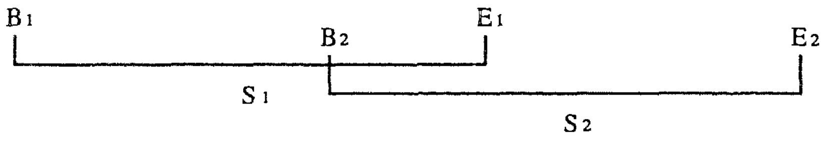


Fig. 4.1 Overlapping

Since an EOT stands for the beginning of a CS which is mutually exclusive with other CS's, one can substitute CS for EOT in Lemmas 3.1, 3.2 and 3.3 without affecting their correctness. We can thus rephrase the lemmas as follows.

Lemma 4.3 *Let H be a history produced by an optimistic scheduler, and let $p_i(x)$ and $q_j(x)$ ($i \neq j$) be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$ then $CS_i < CS_j$.* □

Therefore, the order of CSs is consistent with the serialization order generated by optimistic scheduler SBO, PBO or SFO.

4.4.2 Locked Sections

The locked point (LP) and unlocking point (UP) were defined in Section 2.3. They are synchronization events delimiting the *locked section* (denoted as LS). Similar to confining sections, we extend $<$ to the set of locked sections and write $LS_i < LS_j$ to mean $UP_i < LP_j$. Now we examine properties locked sections have.

For a transaction T_i, by definition, we have $pl_i(x) < LP_i$, for any locking event

$pl_i(x)$. Similarly, $UP_i < pu_i(x)$ holds, for any unlocking events $pu_i(x)$[1]. We formally state the above discussion in the following proposition.

Proposition 4.3 *Let H be a history produced by a 2PL scheduler, and let $p_i(x)$ be an operation in Commit(H). Then $pl_i(x) < LP_i < UP_i < pu_i(x)$.*

Lemma 2.1 says that if $p_i(x)$ conflicts with and precedes $q_j(x)$, then $pu_i(x) < ql_j(x)$. By the above proposition, this implies $UP_i < LP_j$. In terms of locked sections, we thus have $LS_i < LS_j$. We formulate this as follows.

Lemma 4.4 *Let H be a history produced by a 2PL scheduler, and let $p_i(x)$ and $q_j(x)$ be conflicting operations in Commit(H). If $p_i(x) < q_j(x)$, then $LS_i < LS_j$.*

Corollary 4.1 *The locked sections of conflicting transactions do* not *overlap. That is, if transaction T_i conflicts with transaction T_j, then either $LS_i < LS_j$ or $LS_j < LS_i$.*

We refer to confining sections and locked sections as *synchronizing sections.*

4.4.3 Integration

When two subschedulers are integrated into one, a transaction accessing both parts of a database will experience two different kinds of concurrency control, and, consequently, will have both a confining section and a locked section.

For an optimistic subscheduler and a pessimistic subscheduler to cooperate, they should interact in some way. As we could see from discussions given so far in this chapter, optimistic and locking methods share some important properties. They both

[1]Strictly speaking, one should use $\leq$ instead of $<$ in above relations. However, this implies introducing a new partial order. Therefore, we interpret "at which" in the definitions of LP and UP as "right after" and "right before," respectively, in order to use $<$ for LP's and UP's, instead of $\leq$.

arrange transactions according to conflicts. (Time-stamp ordering, in contrast, arranges transactions according to the transactions' arriving times.) They both have synchronizing sections whose order is consistent with a serialization order. To integrate them, we need only to ensure that the serialization orders generated by the optimistic and locking subschedulers are consistent with each other. In other words, we should prevent the situation where one transaction, say T_i, is ordered before another transaction, say T_j, by one subscheduler, but they are ordered in the reverse order by the other subscheduler. That is to say, we should prevent situations such as "$CS_i < CS_j$ and $LS_j < LS_i$" from happening. If we "stick" the confining section and locked section of a transaction together, then such a situation will not arise. Therefore, we add a restriction on the O+P class to form a subclass of O+P.

> Let S be an O+P scheduler. If S ensures that, for every transaction, its confining section and locked section (if both exist) overlap, then S is called an O+P^o scheduler, where "o" stands for overlap.

We now examine properties related to the confining sections and locked sections in a history produced by an O+P^o scheduler.

Property 4.1 *Let H be a history produced by an O+P^o scheduler, and let T_i and T_j be two transactions that both have confining and locked sections such that T_i and T_j conflict on a data item in PES. Then $LS_i < LS_j$ if and only if $CS_i < CS_j$.*

Proof: Only if part: Because LS_i and CS_i overlap, $EOT_i < UP_i$. Similarly, because LS_j and CS_j overlap, $LP_j < ECS_j$. If $LS_i < LS_j$, then $EOT_i < UP_i < LP_j < ECS_j$. This implies that $CS_j \not< CS_i$. By Proposition 4.2, CS_i and CS_j do not overlap, so $CS_i < CS_j$.

If part: If $CS_i < CS_j$ then $LS_j \not< LS_i$ follows immediately by exchanging the roles of confining sections and locked sections in the proof of the "Only if" part. By Corollary 4.1, if T_i and T_j conflict on a PES item, LS_i and LS_j do not overlap. Therefore, $LS_j \not< LS_i$ implies $LS_i < LS_j$. □

The property presented above shows the effect of overlapping the confining and locked sections of a transaction. In 2PL, the serialization order was defined based on the locked points of the transactions, and in an optimistic scheme the serialization order is defined based on EOT's. Here, we shall introduce a reference point in each transaction on which the serialization order for the O+P^o scheduler is based. The *sequencing point* (SP_i) of the transaction T_i coincides with EOT_i or LP_i, if T_i has either CS_i or LS_i but not both. If T_i has both CS_i and LS_i, then SP_i is defined to be the later of EOT_i and LP_i. (See Fig. 4.2.) In other words, in the latter case, SP_i is the starting point of the overlap between CS_i and LS_i. The following lemma shows the importance of sequencing points.

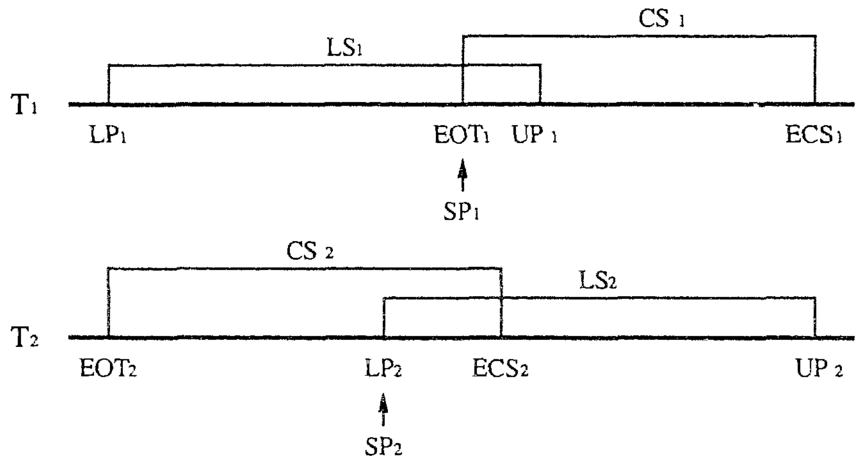


Fig. 4.2 Sequencing Points

Lemma 4.5 *Let H be a history produced by an O+P^o scheduler, and let T_i and T_j have two conflicting operations $p_i(x)$ and $q_j(x)$ in Commit(H), such that $p_i(x) < q_j(x)$. Then we have $SP_i < SP_j$.*

Proof: Since T_i and T_j have conflicting operations, either they both have confining sections, or they both have locked sections, or both. Therefore, from Lemmas 4.3 and 4.4, $CS_i < CS_j$ or $LS_i < LS_j$ holds. If $CS_i < CS_j$, then $ECS_i < EOT_j$. Since CS_i and LS_i overlap, we have $LP_i < ECS_i$ and $EOT_i < ECS_i$. Therefore $SP_i < ECS_i$ holds. Because SP_j is the later of LP_j and EOT_j, $ECS_i < SP_j$ follows from $ECS_i < EOT_j$. So we have $SP_i < ECS_i < SP_j$.

Similarly, when $LS_i < LS_j$, we can show that $SP_i < SP_j$. □

We can now state the following theorem.

Theorem 4.2 *An O+P° scheduler produces only serializable histories.*

Proof: A history H is, in general, a partial order on the set of operations and synchronization events. Let σ(H) be a (totally ordered) sequence of operations and synchronization events compatible with this partial order. Clearly, all SP's are totally ordered in σ(H). Consider this total order as the serialization order. The theorem now follows from Lemma 4.5. □

4.4.4 Necessity for LS-CS Overlap

Now we show that if the confining section and locked section of a transaction are separated, an O+P scheduler may not guarantee serializability.

Theorem 4.3 *Given an O+P scheduler S, there exist a transaction set* **T***, such that, for any transaction $T_i \in$* **T***, if S allows T_i's locked section and confining section not to overlap, then it may produce a non-serializable history.*

Proof: Let T = $\{T_1, T_2\}$ be a transaction set, where

T_1: $r_1(x)$, $w_1(y)$, and
T_2: $r_2(y)$, $w_2(x)$,
for some $x \in$ PES and $y \in$ OPT.

Assume that S allows LS_1 and CS_1 not to overlap. It is possible that $LS_1 < CS_1$. If $LS_1 < CS_1$, S may produce a history in which $LS_1 < LS_2$ and $CS_2 < CS_1$. This history contains $r_1(x) < w_2(x)$ and $r_2(y) < w_1(y)$. Therefore, it is not serializable.

Now assume that S allows LS_2 and CS_2 not to overlap. Therefore, $CS_2 < LS_2$ is possible. Similarly, S may produce a non-serializable history containing $r_2(y) < w_1(y)$ and $r_1(x) < w_2(x)$. □

4.4.5 Restricting Overlappings

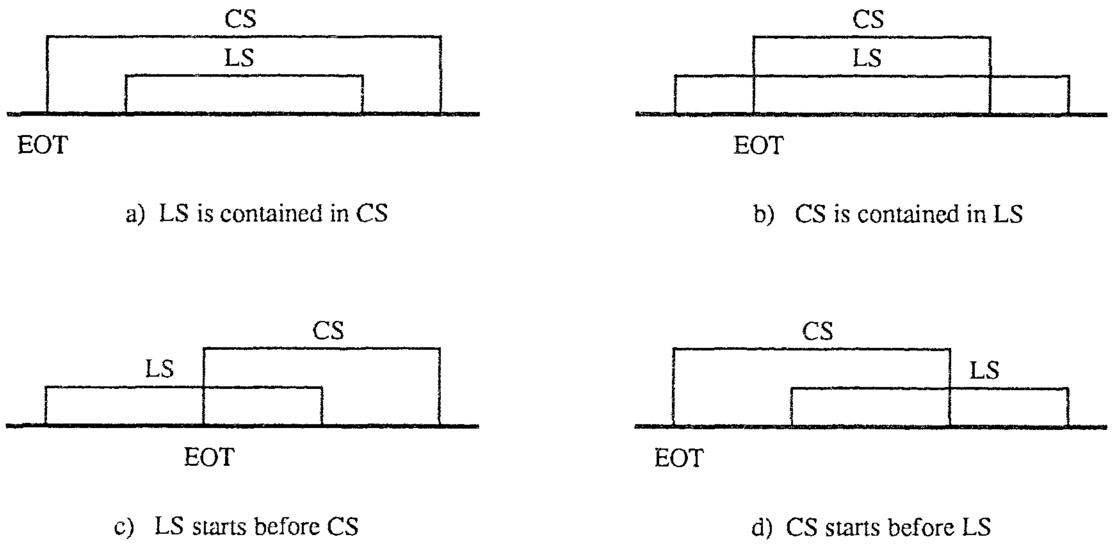


Fig. 4.3 Overlappings of CS and LS

In the above discussions, the way in which the locked section (LS) and the confining section (CS) overlap was not restricted. So there are four possible ways they can overlap (Fig. 4.3): (a) the entire LS is contained in the CS, (b) the entire CS is contained in the LS, (c) the LS starts before the CS, and (d) the CS starts before the LS. In a centralized database system, the scheduler cannot receive the *End* request of a transaction until and unless it has received all the read and write requests of the transaction. That is, the validation of a transaction should be after any prewrite of the transaction. Also, according to our model, all write requests of a transaction are first executed via prewrite operations to the transaction's private workspace, then, only after the scheduler receives an *End* command, are the writes reflected to the database (by dm-writes). Further, since the CS of a transaction should be executed mutually exclusively and is often implemented as critical section, waiting for a lock inside the CS may easily result in a deadlock, especially in serial validation. Therefore, we further restrict the way the two sections can overlap. Practically, overlappings (b) and (c) are more meaningful. What distinguishes (b) and (c) from (a) and (d) is that EOT takes place in the locked section. So, thereafter, we will concentrate on the development of combined schedulers with this characteristic, and call them O+P^r schedulers, where "r" stands for *restricted.*

4.5 Some Combined Algorithms

In this section, we present some combined algorithms. They are all in the class O+P^r.

4.5.1 Serial Forward Checking + 2PL

We present a composition of SFO and 2PL. The procedures for *Read* and *Write* requests are the same as those in Algorithm 4.1. We therefore present only the procedures for *Begin* and *End* requests.

Algorithm 4.2: (SFO + 2PL)

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

```
$Active := Active \cup \{T_i\}$ % should be executed atomically.
$RS_i := WS_i := WL_i := \phi$
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
C1 begin (system-wide) critical section
       $Active := Active - \{T_i\}$
       $conflict$:=false
       for every $T_j \in Active$ do
           if $WS_i \cap RS_j \neq \phi$ then $conflict$:=true
       if $conflict$ then resolve the conflict by aborting either $T_i$
                          or all $T_j$'s such that $WS_i \cap RS_j \neq \phi$.
                          The decision is made upon some cost criteria.
       for every $x \in WS_i$ issue a $dm\text{-}write(x)$        % Reflecting
   end critical section
C2 release read-locks
C3 for every $x \in WL_i$ issue a $dm\text{-}write(x)$
C4 release write-locks
C5 commit
```

The correctness of the algorithm follows from the correctness of SFO, Strict 2PL, and O+P^r class. There are a few things worth mentioning here. First, the critical section used here is a tool for achieving mutual exclusion for SFO activities. Therefore, we need not stop 2PL activities in the critical section. 2PL activities can take place in parallel with the critical section. Specifically, we may release read-locks held by a transaction at the beginning of the critical section before the validation of the transaction, so that the unlocked data items are available for other transactions sooner.

However, we should point out that, in practice, achieving mutual exclusion on all data items is much easier than achieving mutual exclusion on OPT only, especially when the border between OPT and PES is dynamically changeable.

Second, as stated in Difference 1 in Section 3.4, forward checking resolves conflicts more flexibly than backward checking. This has an added significance in combined schedulers. Consider long transactions. The longer a transaction, the greater is the chance it conflicts with other transactions in OPT, and the higher is the cost to abort it. A long transaction accessing both OPT and PES may be further delayed due to its waiting for locks. Therefore, the cost of abortion in backward checking is even higher, especially because, when a transaction is to be aborted, it has got all its locks. In forward checking, however, we can choose not to abort the transaction undergoing validation, instead, we can abort the transactions that conflict with it. It is interesting that this is achieved in the validation-and-write phase of the transaction. So, we need not even know that the transaction is a long one when it arrives at the system.

4.5.2 Serial Forward Checking + Deferred Write-Locking

As stated in Section 2.1, a write operation of a transaction only writes a new value in the transaction's private work space by a prewrite operation. The new value is not reflected to the database and is not visible to the other transactions until and unless the transaction passes its validation. Therefore, a transaction is two-phased no matter whether it is scheduled by a 2PL, an optimistic, or an O+P scheduler. To be consistent with the optimistic scheme, we use the terms *read phase* and *validation-and-write phase* to name the corresponding phases. In Algorithms 4.1 and 4.2, write-locks are set *too early*. Concurrency may be increased if we postpone setting write-locks as much as possible. Actually, we need not set write-locks in the read phase. We can do so in the validation-and-write phase. We present a combined scheduler with deferred write-locking below. In Chapter 8, we will combine deferred write-locking with PBO.

Algorithm 4.3: SFO + Deferred Write-Locking

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

```
    $Active := Active \cup \{T_i\}$
    $RS_i := WS_i := WL_i := \phi$
```

- When it receives a *Read(x)* request from transaction T_i, the scheduler does the following:

```
    check-member(x) % determine which part of database $x$ belongs to
    case R1: $x \in PES$
        if $x$ is in $T_i$'s private work space
            then read $x$ from there
        else if $x$ is already write-locked, block $T_i$
                until read-lock can be set on $x$.
            set read-lock on $x$
            dm-read(x)
    case R2: $x \in OPT$
        if $x$ is in $T_i$'s private work space
            then read $x$ from there
        else dm-read(x)
            $RS_i := RS_i \cup \{x\}$
```

- When it receives a *Write(x, new-value)* request from transaction T_i, the scheduler does the following:

```
    check-member(x)
    case W1: $x \in PES$
        $WL_i := WL_i \cup \{x\}$
        prewrite(x, new-value)
    case W2: $x \in OPT$
```

```
            $WS_i := WS_i \cup \{x\}$
            $prewrite(x, new\text{-}value)$
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
parallel-for $x \in WL_i$ do % See Remark
   if $x$ is locked then wait until write-lock can be set on $x$
end parallel-for
begin (system-wide) critical section          %Start validation.
  $Active := Active - \{T_i\}$          % $EOT_i$ is here.
  $conflict$:=false
  for every $T_j \in Active$ do
     if $WS_i \cap RS_j \neq \phi$ then $conflict$:=true
  if $conflict$ then resolve the conflict by aborting
                 either $T_i$ or all $T_j$ such that
                 $WS_i \cap RS_j \neq \phi$. The decision is made upon
                 some cost criteria.
  for every $x \in WS_i$ issue a $dm\text{-}write(x)$          % Start reflecting
end critical section
release read-locks
for every $x \in WL_i$ issue a $dm\text{-}write(x)$
release write-locks
commit.
```

Remark: Parallel-for can be thought of as a process for each x. These processes run concurrently.

First we show that the algorithm is correct. We need only to show that deferred write-locking is a two-phase locking algorithm. This can be done easily by verifying that deferred write-locking satisfies the four lock ownership rules in Section 2.3.

Write-locks are acquired in the last possible moment. They could be acquired in the critical section. But, this may cause a transaction to wait for a lock forever in the critical section, resulting in a deadlock. One can immediately see two improvements over Algorithm 4.2:

1. The duration that a write-lock is held by a transaction is likely to be shortened.
2. By collecting all the write-lock requests together and executing them in parallel, the time a transaction spends waiting for write-locks may also be shortened.

Since the duration that a write-lock is held is likely to be shorter, the possibility and time a read or write operation is blocked may also be decreased and shortened, respectively. Consequently, the duration of a transaction may be shortened and concurrency may be increased.

We think that deferred write-locking combines nicely with optimistic methods. We can see this from the view point of version control. A *prewrite(x)* will generate a version of x. This version is not visible to the other transactions (other than its creator) until and unless a corresponding *dm-write(x)* reflects it to the database. We call this version an *uncommitted version of x* when it is created. When the version is reflected to the database, we call it the *committed version of x.* In our model, there is only one committed version for each data item at any time, no matter what concurrency control method is used. (This is not the case for a mulit-version database discussed in Sec. 4.6.) In an optimistic method, there could be several uncommitted versions of a data item, say x, at a given time. Different transactions may read different uncommitted versions (created by themselves) at the same time. In the 2PL algorithms discussed in previous chapters and sections, there was at most one uncommitted version of x at any time, because a transaction must hold a write-lock before executing a prewrite, and keep the lock until the version it created was reflected to the database. Besides, only one version, either committed or uncommitted, was readable at any time. This is a kind of mismatch. In the deferred write-locking, however, the situation is the same as in an optimistic method. There are multiple uncommitted versions of a data item, each visible only to its creator. It seems more natural to use such locking to combine with optimistic algorithms.

One shortcoming of deferred write-locking is that it may cause more deadlocks than than the standard 2PL, and the damage of a deadlock caused by it may be more severe than that in the standard 2PL. For example, suppose that two transactions T_i and T_j both first read and then write a data item x ($x \in$ PES). Let $req\text{-}p_i(x)$ ($req\text{-}q_j(x)$) denote the time when the scheduler receives the request $p_i(x)$ ($q_j(x)$), and let End_i (End_j) denote the time the scheduler receives End from T_i (T_j). Consider the deadlock caused only by locking x for T_i and T_j. The condition for such a deadlock in the standard 2PL is $req\text{-}r_i(x) < req\text{-}w_j(x) \wedge req\text{-}r_j(x) < req\text{-}w_i(x)$, while the deadlock condition in deferred write-locking is $req\text{-}r_i(x) < End_j \wedge req\text{-}r_j(x) < End_i$. Apparently, the latter is much more easily satisfied than the former. Further, such a deadlock can be detected in standard 2PL at the last of $req\text{-}w_i(x)$ and $req\text{-}w_j(x)$, but in deferred write-locking, the last of End_i and End_j. So, the cost of recovering from such a deadlock in deferred write-locking is more severe than that in the standard 2PL. Because about 90% of deadlocks involve only two transactions [7], and because the deadlock discussed above is a common type of deadlock, this shortcoming of deferred write-lock is very serious to performance. We still need a simulation study to find out how deferred write-locking with the standard 2PL, especially, in the hybrid scheduler environment.

4.5.3 Relaxed Locking

Here we present a concurrency control algorithm called "Relaxed Locking" (RL), based on the ideas we have come across so far. It utilizes the overlapping of locked section and confining section, even though the database is *no longer* partitioned into OPT and PES. The idea of RL is as follows: A read operation, as in the optimistic scheme, does not block a conflicting write operation. On the other hand, a write operation, as in the locking scheme, will block any operation conflicting with it. Thus, the scheduler maintains only write-locks. No read-locks are ever set. Instead, read-sets and write-sets are used in validations to detect read-write conflicts. When a read request, say *Read(x)*, comes from T_i, the scheduler checks if x is (write-)locked by some other transaction. If so, it blocks T_i until the lock is released. When it is not locked, x is read for T_i and put in the corresponding read-set. For a write request, the scheduler does almost the same thing as it does for read request, except it has to set a lock for the prewrite operation. Eventually, when the *End* request comes, the scheduler uses the forward checking strategy to validate T_i. It checks the intersections of the write-set of T_i and the read-sets of other "active" transactions. The formal description is presented below.

Algorithm 4.4: Relaxed Locking (RL)

- When it receives a *Begin* request from transaction T_i, the scheduler does the following:

```
$Active := Active \cup \{T_i\}$
$RS_i := WL_i := \phi$
```

- When it receives a *Read(x)* request from transaction T_i, the scheduler does the following:

```
if $x$ is in $T_i$'s private work space
  then read $x$ from there
  else
      if $x$ is locked by some other transaction
        then block $T_i$ until the lock on $x$ is released
      begin critical section
        $RS_i := RS_i \cup \{x\}$
        $dm\text{-}read(x)$
      end critical section
```

- When it receives a *Write(x, new-value)* request from transaction T_i, the scheduler does the following:

```
if $x$ is locked
  then wait until lock can be set on $x$
set lock on $x$ for $T_i$
$WL_i := WL_i \cup \{x\}$
prewrite($x$, new-value)
```

- When it receives an *End* request from transaction T_i, the scheduler does the following:

```
begin critical section
  $Active := Active - \{T_i\}$
  conflict:=false
  for every $T_j \in Active$ do
     if $WL_i \cap RS_j \neq \phi$ then conflict:=true
  if conflict then resolve the conflict by aborting either $T_i$
                   or all $T_j$'s such that $WL_i \cap RS_j \neq \phi$.
                   The decision is made upon some cost criteria.
end critical section
for every $x \in WL_i$ issue a dm-write($x$)          % Reflecting
release locks
commit
```

Note that, unlike SFO, the critical section here is an ordinary critical section such as that in SBO and PBO. The algorithm has some special features: (1) It uses rollback to resolve the read-write conflicts and blocking to resolve the write-read and write-write conflicts. (2) It has less blocking and more rollbacks than 2PL algorithms. On the other hand, it has less rollbacks and more blocking than optimistic algorithms.

(3) A read-only transaction does not block any other transaction, but may be blocked by some locks.

We expect the algorithm to have a good performance in situations where conflicts are rare, but not rare enough to justify the use of any of the optimistic algorithms. The algorithm is correct, i.e., it generates only serializable histories. The proof of its correctness is straightforward after the proofs in the previous chapters and sections.

After RL was designed, it was discovered that Agrawal and El Abbadi [2] had developed an algorithm similar to RL under different motivation. Their simulation confirms our prediction about its performance [1].

4.6 Going into Multiversion World

In this section, we extend our hybrid scheme to multiversion databases. For multiversion databases and the locking scheme for them see [7]. For optimistic multiversion concurrency control see [3].

In a *multiversion database*, a data item may have more than one version simultaneously stored in the database. A read operation now reads a "version" of a data item. A write operation generates a new version of a data item, without overwriting an old one. Old versions are still accessible to transactions. Each data item, x, now has a list of versions. A version of x is denoted as x_i, where the subscript i is called the version number, which is the transaction number of the transaction that creates it. For two versions of x, $x_i <_v x_j$ if i is less than j.

A transaction with a (potential)[2] write operation is called an *updator*. A transaction that is not an updator is called a *query*. In other words, a query is a read-only transaction. We assume that, when a transaction is submitted to the TM, the TM

[2]A transaction whose program contains a write operation may actually not execute that write operation.

is informed or can find out easily whether the transaction is a query or an updator. Each transaction T_i is assigned a transaction number, denoted as *tn*(T_i). A query's transaction number is, however, assigned when the query starts.

A query will never be blocked or validated. It is executed asynchronously with respect to updators and other queries. This is achieved by letting a query read some old versions of data items. When a query, Q, starts, a transaction number *tn*(Q) is assigned to Q, such that any updator with its transaction number less than or equal to *tn*(Q) has been committed or aborted when Q starts. Later, we will show a technique to assign a *maximal* transaction number to a query so that it can read as up-to-date information as possible. For a *Read*(x) operation from Q, the scheduler will find a version of x with the largest version number less than or equal to *tn*(Q). By the above principle, the updator that created the version of x had already committed when Q started. In other words, Q will never "read from"[3] any updator executing concurrently with it. Therefore, there is no need to set a lock or modify a read-set for an operation from a query, and there is no need to distinguish an OPT item from a PES item.

To the updators, the database is still partitioned into OPT and PES as before. A read operation of an updator reads either the version it has created itself, or (if such a version does not exist) the newest version created by a committed updator. We call this version the *newest committed version*. The read set of an updator contains a set of data items together with their version numbers, instead of just a set of data items. The write-set of an updator, however, contains as before a set of data items. An updator gets its transaction number, which is the current value of *tnc*, in its confining section. Since updators may enter their confining section in a different order than they leave their validation-and-write phase, we cannot directly use *tnc* to assign transaction numbers to queries. To deal with this problem we use another transaction

[3]We say that a read operation *reads from* a write operation if (1) the two operations are from different transactions, and (2) the read operation reads the value created by the write operation. Note that there is no such relation if two operations operate on different versions. A transaction T_i *reads from* another transaction T_j, if T_i has a read operation that reads from a write operation of T_j.

number counter called *vtnc* (visible *tnc*). *vtnc* indicates the latest updator of which the result is available to queries. It is used to assign transaction numbers to queries. An committing queue (CQ) is maintained, which contains an entry for each updator in its validation-and-write phase. Each entry E in CQ contains a type field (E.*type*) and a number field (E.*num*) storing the transaction number. CQ is ordered on the *num* field of the entries (i.e., the order of entering the corresponding confining section). The value of a *type* field is either *VALIDATING* or *WRITTEN*. *VALIDATING* means that the updator it represents is being validated and *WRITTEN* means that the updates of the updator it represents are already reflected to the database and are available to queries and other updators.

Below, we present a hybrid multiversion concurrency control algorithm using backward checking. The multiversion algorithm using forward checking is straightforward from this.

Backward checking

When checking $RS_i \cap WS_j = \phi$ for updators U_i and U_j, the versions in RS_i whose version number is greater than or equal to $tn(U_i)$ are not considered, for these versions are read from U_j or some later transactions.

Given below is an algorithm description.

- When it receives a *Begin* request from updator U_i, the scheduler does the following:

```
        $start\text{-}tn_i := tnc$
        $WL_i := RS_i := WS_i := \phi$
```

- When it receives a *Read(x)* request from updator U_i, the scheduler does the following:

```
        $check\text{-}member(x)$
        case R1: $x \in PES$
            if $x$ is in $U_i$'s private work-space
```

```
                then read $x$ from there
                else if $x$ is write-locked by another updator, block $U_i$
                        until a read-lock can be set on $x$
                    set read-lock on $x$
                    $dm\text{-}read(x_k)$ where $x_k$ is the newest committed version of $x$
        case R2: $x \in OPT$
            if $x$ is in $U_i$'s private work-space
                then read $x$ from there
                else $dm\text{-}read(x_k)$ where $x_k$ is the newest committed version of $x$
                    $RS_i := RS_i \cup \{x_k\}$
```

- When it receives a *Write(x, new-value)* request from updator U_i, the scheduler does the following:

```
        $check\text{-}member(x)$
        case W1: $x \in PES$
            $WL_i := WL_i \cup \{x\}$
            $prewrite(x, new\text{-}value)$
        case W2: $x \in OPT$
            $WS_i := WS_i \cup \{x\}$
            $prewrite(x, new\text{-}value)$
```

- When it receives an *End* request from updator U_i, the scheduler does the following:

```
        C1, parallel-for $x \in WL_i$ do
                if $x$ is locked then wait until write-lock can be set on $x$
                set write-lock on $x$
            end parallel-for
        C2, begin critical section
                $finish\text{-}tn_i := tnc$
                $tnc := tnc+1$
            allocate entry E
            E.$type$:= $VALIDATING$
            E.$num$:=$tnc$
            append E to CQ
        end critical section
    C3, release read-locks
    C4, for $U_j$ such that $start\text{-}tn_i + 1 \leq tn(U_j) \leq finish\text{-}tn_i$ do
            $RS'_{i,j} := \{x_k \mid x_k \in RS_i \text{ and } k < tn(U_j)\}$
            % $RS'_{i,j}$ contains only those versions created by updators
            % preceding $U_j$ in equivalent serial history.
            if $WS_j \cap RS'_{i,j} \neq \phi$ % Version numbers are ignored in set operations
               then delete E from CQ
                    abort
    C5,     attach the version number ($finish\text{-}tn_i + 1$) to the copy of each item $x$,
                    $x \in WS_i \cup WL_i$, in $U_i$'s private work-space
            reflect $WS_i \cup WL_i$ to the database
                    by creating new versions in the database
    C6, release write-locks
    C7, begin critical section
            $tn(U_i) := finish\text{-}tn_i + 1$
            E.$type$:= $WRITTEN$
            while head(CQ).$type$ = $WRITTEN$ do
                  $vtnc$:=head(CQ).$num$ % Used for queries
                  delete head(CQ)
            end (while)
        end critical section
```

Correctness

The correctness proof for the algorithm is based on a model different from what we introduced in Chapter 2. The notion *conflict* cannot be used here. Instead, we have to use the notion *read from* (see the footnote in this section). Therefore, *conflict equivalence* should be replaced by *view equivalence* in discussing equivalence of histories. The details are omitted.

Chapter 5

Dynamic Re-partitioning of the Database

In this chapter, we study how to dynamically re-partition a database, in order to keep the up-to-date information of the conflict rate distribution reflected in the partition of the database. It constitutes an important part of our adaptive concurrency control scheme. What makes dynamic re-partitioning complex is that we should not stop the database system for re-partitioning. One way of carrying out re-partitioning is to use a group of transfer processes, which run concurrently with database transactions. A transfer process transfers a data item from OPT to PES, or vise versa. So, we must guarantee that, despite interference from transfers, the execution history is still serializable.

We study a specific transfer algorithm, i.e., one to be used in conjunction with Algorithm 4.2. The transfer algorithms for the other combined schedulers can be designed in a similar way. In section 5.1, we discuss interference of transfers to transaction execution. In Section 5.2 we present our transfer algorithm. In section 5.3, we rewrite Algorithm 4.2 to make it compatible with a transfer algorithm, and in Section 5.4, we give a correctness proof for it.

5.1 Managing Interference

Transferring a data item being accessed by transactions will definitely interfere with transaction execution. In the presence of transfers, data items are "on the run" from the viewpoint of transactions. For example, a transaction may read a data item from OPT, but later when it wants to write on the same item, it may be in PES. Therefore, the major problem in designing a transfer algorithm is to manage interference so that serializability is ensured and performance is not seriously degraded.

Performance can be affected in different ways. One can immediately work out some "brute force" transfer methods which guarantee serializability of transaction execution but have a bad impact on performance. Following are two such methods, which resort to drastic measures. Suppose we are transferring x from OPT to PES.

Bullying *Abort all transactions that are accessing x and then transfer x.*
This method is not acceptable, for it causes too many abortions.

Polite *Defer the transfer until there is no transaction accessing x.* This method is unacceptable either, because (1) it cannot guarantee that x is eventually transferred to PES, and (2) performance will be degraded by delaying the transfer of x to PES, since the reason to transfer x to PES is that the conflict rate on x is going higher or is already high enough not to justify the use of an optimistic method.

The two methods represent the two extremes of interference. The comments on the Polite suggest that we should "manage" interference rather than merely "minimize" it. We suggest that both of the following goals should be taken into account in designing a transfer algorithm: minimize the blocking of transactions due to data transfer, and transfer a data item to its destination as soon as possible.

To investigate the interference of transfers to transaction execution, we introduce a new term: "contention." We say that there is *contention* on data item x, if more than one currently active transaction is either accessing it or is trying to access it, and at least one of them wants to write it. Here, we say that a transaction T_i is "accessing x" if $x \in (RS_i \cup WS_i)$ for $x \in$ OPT, or T_i holds a lock on x for $x \in$ PES, and T_i is "trying to access x if it is waiting for a lock on x ($x \in$ PES).

Transferring a data item x from OPT to PES or vice versa, when no transaction is accessing or trying to access x is trivial, since the transfer will not interfere with transactions. x should be simply removed from the source part (OPT or PES) and added to the destination part. Transferring x when there is no contention is also relatively straightforward. In this case, however, besides changing the membership of x, some additional actions must be taken. When x was being accessed by T_i in OPT before transfer to PES, for example, an appropriate lock must be set on x on behalf of T_i.

Transferring x when there is contention on it requires careful consideration, for interference of the transfer is complicated. Since the way the contention-related information is stored for data items in PES is different from that for OPT, it is fairly messy to convert it from one to the other. The approach that we adopt therefore, is to avoid such conversion altogether, simply by disallowing a transfer of a data item in contention until the contention is resolved. To see the underlying motivation behind this principle, suppose that, presently, there is a contention on data item x. If x is transferred from PES to OPT at this time, then some transaction involved in the contention will be aborted. This can be avoided if x is not transferred immediately. Suppose x is transferred from OPT to PES at this time. We have to set a lock on it for one transaction and block the other. Since we have no idea which transaction should have a lock and which should be blocked, this may cause some unnecessary deadlocks. There is another problem more serious than unnecessary deadlocks. Suppose that T_i is involved in the contention due to operation $o_i(x)$. Further suppose that after the transfer of x to PES, some other transaction obtains a lock on x and T_i is left to wait for the lock. T_i may have another operation $o_i'(y)$ ($y \in$ PES) such that $o_i(x) < o_i'(y)$ and T_i has already obtained a lock on y. In other words, T_i has obtained the lock for the later operation but is waiting for the lock for the earlier operation. Even though the operation $o_i(x)$ has been processed, (so that it is not a logical problem for T_i,) when T_i gets the lock on x, it will resume its execution at some point other than $o_i(x)$, which will make the control flow more complex.

5.2 The Transfer Algorithm

Intuitively, transferring a data item back and forth between OPT and PES frequently would be counter-productive. Therefore, we impose a restriction on the frequency of transfers. Specifically, we make the following assumption.

Assumption 5.1 *The time interval between two consecutive transfers of an item is greater than the maximal transaction life-time.*

This assumption will not affect the correctness for transfers and transaction executions. However, it helps to make the correctness proof in Section 5.4 easier. This assumption also suppresses unnecessary transfers, which enhance performance. For this purpose, it is probably more desirable to extend the time interval to *two or three times of the maximal transaction life-time.* Choosing "two or three" is based on some "gut feeling" rather than scientific evidence.

There are two kinds of transfers: from OPT to PES and from PES to OPT. Both kinds of transfers will access some data structures used by the scheduler.

When a transfer process detects some contention on the data item being transferred, there are three alternatives:

1. abort all the transactions that are involved in the contention,
2. abort the transfer process itself, and
3. wait until the scheduler resolves the contention.

We discard the first choice, because it is costly and not interesting. When transferring a data item, say x, from PES to OPT, the conflict rate on x is presumably going lower, or is already low. It is likely that, when we transfer x some time later, there will be no contention on x. Since aborting a transfer process itself is cheap, it would be a good choice to abort the transfer process in this situation. We use this strategy in Case T2.4 in the transfer algorithm given below. However, the second choice is not always usable. When transferring x from OPT to PES, the conflict rate on it is going higher, or is already high enough not to justify the use of optimistic method. If we abort the transfer process and restart it later, it is very likely that the restarted process again detects contention on x, and the transfer process is aborted again. Further, due to a high conflict rate, many transactions may be aborted during this time. In this situation, we should transfer x to PES as soon as possible. We therefore choose the third choice, and make the transfer process wait. To prevent more transactions from being involved in the contention on x when the transfer process is waiting, we drop x from OPT but do not add it to PES immediately. Combined with Cases R3 and W3 in the revised Algorithm 4.2, this achieves the effect of locking. It allows those transactions that have already accessed x (i.e., those involved in the contention) to access x while blocking all the other transactions that want to access x. When these transactions that are involved in the contention all finish, the contention is already naturally resolved. The transfer process can resume and finish the transfer. In Cases T1.4 and T1.5 this strategy is used. We also extend the use of this strategy to Case T2.3, where there is no contention on x and the transaction that holds the write-lock on x has already finished its read phase. Since dropping x from OPT or PES works as a lock on x, it contributes to forming deadlocks.

Here is a description of our transfer algorithms.

(1) TRANSFER x FROM OPT TO PES

```
$morerw$:=$morew$:=false % See Remark 1
begin critical section
```

```
    compute
        $R := \{T_i \in Active | x \in RS_i\}$ % Recall $Active$ set used in Algorithm 4.2
        $W := \{T_i \in Active | x \in WS_i\}$
        $OPT := OPT - \{x\}$
    case T1.1: $R = W = \phi$ % $x$ is not being accessed
        $PES := PES \cup \{x\}$
    case T1.2: $R \neq \phi \wedge W = \phi$ % $x$ is being read but not written
        set read-lock on $x$ for each $T_i \in R$ % See Remark 2
        $PES := PES \cup \{x\}$
    case T1.3: $R = \phi \wedge W = \{T_j\}$ % $x$ is being written by one transaction.
        set write-lock on $x$ for $T_j$
        $WL_j := WL_j \cup \{x\}$
        $PES := PES \cup \{x\}$ % $x$ remains in $WS_j$.
    case T1.4: $R = \phi \wedge |W| > 1$
        $morew$:=true % See Remark 3
    case T1.5: $R \neq \phi \wedge W \neq \phi$
        $morerw$:=true
end critical section
T1.4a: if $morew$ then wait until all $T_j \in W$ abort or
                                        finish their step C1 of Algorithm 4.2
                    $PES := PES \cup \{x\}$
T1.5a: if $morerw$ then
            wait until all $T_j \in W$ abort or finish their step C1 % See Remark 3
            set read-lock on $x$ for each remaining $T_i \in R$
            $PES := PES \cup \{x\}$ % $x$ remains in $RS_i$
```

Remark 1: $morerw =$ true indicates that x is being read and written by more than one transaction. $morew =$ true indicates that $morerw$ is not true but x is being written by more than one transaction.

Remark 2: Read-locks may be set after the actual read operations. So, $rl_i(x) <$

$r_i(x) < ru_i(x)$ may not hold. x remains in RS_i.
Remark 3: Not adding x to PES immediately simply locks all the other transactions out.
Remark 4: It may be hard to detect when a transaction finishes its step C1. So, we can relax the condition to "until all $T_j \in W$ complete (either abort or commit)."

(2) Transfer x from PES to OPT

```
$wait$:=false
critical section
  check the locks on $x$ and the waiting queue for $x$. set
     $Rlock$: the set of transactions which hold read-locks on $x$
     $Wlock$: the set of transactions which hold write-lock on $x$.
                 Note that $Wlock$ contains at most one element.
     $Rwait$: the set of transactions waiting for read-locks on $x$.
     $Wwait$: the set of transactions waiting for write-locks on $x$.
     % See Remark 1
  case T2.1: $Rlock = Wlock = Rwait = Wwait = \phi$
       $PES := PES - \{x\}$
       $OPT := OPT \cup \{x\}$
  case T2.2: $Rlock \neq \phi \wedge Wwait = \phi$
       for every $T_i \in (Rlock \cap Active)$ do
          $RS_i := RS_i \cup \{x\}$ % $T_i$ continues to hold read-lock on $x$
          $PES := PES - \{x\}$
          $OPT := OPT \cup \{x\}$
  case T2.3: $Wlock = \{T_j\} \wedge Rwait = Wwait = \phi$
       if $T_j \in Active$ then $WS_j := WS_j \cup \{x\}$ % $T_j$ continues to hold
                                                    % write-lock on $x$.
                          $PES := PES - \{x\}$
                          $OPT := OPT \cup \{x\}$
```

```
                else $wait$:=true
                         $PES := PES - \{x\}$ % See Remark 2
    case T2.4: $(Rlock \neq \phi \wedge Wwait \neq \phi) \vee (Wlock \neq \phi \wedge (Rwait \cup Wwait) \neq \phi)$
        abort the transfer % Since the conflict on $x$ is known to be going low,
                           % Restarting transfer later would be a wise choice.
end critical section
T2.3a: if $wait$ then wait until $T_j$ finishes
              $OPT := OPT \cup \{x\}$
```

Remark 1: When $Rlock \neq \phi$, $Wlock$ must be empty, and vise versa. We assume that when $Rlock \neq \phi \wedge Wwait = \phi$, $Rwait = \phi$

Remark 2: T_j is in its write phase. The transfer process must be executing during the reflecting of T_j's modification in PES. x cannot be transferred to OPT at this moment, because validation has finished and if a transaction reads x from OPT, serializability may be violated when the read takes place before the reflecting of the item. So, we let T_j hold exclusive the lock on x until it completes.

5.3 Revision of Algorithm 4.2

To ensure that the concurrency control algorithm can run concurrently with a transfer algorithm, we have to protect the operations on the data structure shared with the transfer algorithm by putting them in a critical section and revise the algorithm slightly. The main change is in dealing with the case where the data item being accessed is "in transit," i.e., it belongs to neither PES nor OPT; it is in the process of being transferred from one to the other. (See cases R3 and W3 below). We assume that each single statement is atomic. The procedures for *Begin* and *End* remain the same as before. The procedures for *Read* and *Write* are described below. Most steps are straightforward, except for cases R3 and W3. When data item x is in transit, we let transactions that have accessed x proceed and access x in their private work space again, while preventing other transactions from accessing x. This is achieved through the transfer process by dropping x from OPT or PES, wherever it was (see the transfer algorithm in Section 5.2). This is equivalent to setting a lock on x.

- When it receives a *Read(x)* request from transaction T_i, the scheduler does the following:

```
block:=false          % See Remark 1
waittrf:=false if $x$ is in $T_i$'s private work space
                   then readwrkspc:=true
                   else readwrkspc:=false
begin critical section
  check-member(x)
  case R1: $x \in OPT$
       $RS_i := RS_i \cup \{x\}$
  case R2: $x \in PES$
       if not readwrkspc then
         if $x$ is write-locked
           then block:=true
           else set read-lock on $x$
  case R3: $x \notin (OPT \cup PES)$ % $x$ is in transit.
       if $x \in (RS_i \cup WS_i)$
         then $RS_i := RS_i \cup \{x\}$ % See Remark 2
         else waittrf:=true
end critical section
if block then wait until the read-lock can be set on $x$
           set read-lock on $x$
if waittrf then wait until $x \in (OPT \cup PES)$
             if $x \in OPT$ then $RS_i := RS_i \cup \{x\}$
             else if $x$ is write-locked then
                    wait until the read-lock can be set
                    set read-lock on $x$
    if $readwrkspc$          % $readwrkspc$=true implies $block$=false
                then read $x$ from $T_i$'s private work space
                else $dm\text{-}read(x)$.
```

- When it receives a *Write(x, new-value)* request from transaction T_i, the scheduler does the following:

```
    $block$:=false
    $waittrf$:=false
    begin critical section
      $check\text{-}member(x)$
      case W1: $x \in OPT$
          $WS_i := WS_i \cup \{x\}$
      case W2: $x \in PES$
          if $x$ is read- or write-locked
              then $block$:=true
              else set write-lock on $x$
                  $WL_i := WL_i \cup \{x\}$
      case W3: $x \notin (OPT \cup PES)$
          if $x \in (RS_i \cup WS_i)$
            then $WS_i := WS_i \cup \{x\}$ % See Remark 3
            else $waittrf$:=true
    end critical section
    if $block$ then wait until the write-lock can be set on $x$
                set write-lock on $x$
                $WL_i := WL_i \cup \{x\}$
    if $waittrf$ then wait until $x \in (OPT \cup PES)$
                if $x \in OPT$ then $WS_i := WS_i \cup \{x\}$
                            else if $x$ is locked, wait until the
                                write-lock can be set on $x$
```

```
                        set write-lock on $x$
                        $WL_i := WL_i \cup \{x\}$
        prewrite($x$, new-value).
```

Remark 1: These flags are used for pulling the read, write, and waiting operations out of the critical section. *block*=true means that T_i is blocked by a lock set on x. *waittrf*=true means that x is in transit.

Remark 2: Allow T_i to read x in transit. At this moment, T_i does not own any lock on x. If T_i owns a read lock on x, $x \notin (OPT \cup PES)$ cannot be true by T2.2 and T2.4 (see section 5.2). If T_i owns a write-lock on x, by T2.3 $T_i \notin Active$, i.e., T_i is in its validation-and-write phase. It cannot send a read request now. It is crucial to let T_i proceed (see T1.4, T1.5).

Remark 3: Allow T_i to write on the item being transferred if T_i accessed it before. See T1.4 and T1.5 in Section 5.2.

5.4 Correctness Proof

A transfer algorithm is said to be correct, if (1) it transfers a data item from one part to the other, and more importantly, (2) it does not interfere with the scheduler in such a way that it causes the scheduler to generate nonserializable histories. Proving the first for our transfer algorithm is straightforward. The second is what we are going to prove in this section.

First, to see intuitively that our transfer algorithm is correct, we need only to see that serializability is ensured before, during, and after a transfer. Consider transferring x from OPT to PES. Before the transfer, assume that serializability is ensured. During the transfer, only an optimistic method applies to x. Some transactions' requests for accessing x are delayed. But delays will not affect serializability. So, it is again ensured. After the transfer, any transaction that accessed x before the transfer and is still active at this time will own a lock on x. Since x is transferred to PES only if there is no contention on it, no conflicting locks are set as a result of the transfer. To any other transaction requesting to access x after the transfer started, it looks like that x is originally in PES. Serializability is then ensured by 2PL method. Note that, x may still resides in the read-sets or write-set of some active transactions after the transfer. However, this will never cause any transaction abortion, because (1) when x was transferred to PES, there was no contention on it, and (2) after the transfer, no transaction will put x in its read- or write-set.

Now we prove that our algorithm is correct more formally. As in Chapter 3, we want to show that if $p_i(x)$, an operation of T_i, conflicts with and precedes $q_j(x)$, an operation of T_j, then EOT_i precedes EOT_j. Since EOT's are totally ordered, the correctness follows immediately. Because of the complications due to data transfers, we break the proof into three lemmas, each dealing with a specific kind of conflict.

As seen in the description of the transfer algorithm, when x is transferred from OPT to PES, x is not deleted from each RS or WS which contains it. Similarly, when an item x is transferred from PES to OPT, the transfer algorithm does not release the locks on x immediately. Transactions continue to hold these locks until they complete. A transfer of a data item from OPT to PES does not set any conflicting locks, nor does a transfer from PES to OPT introduce any conflicting operations. We formulate these in the following five propositions.

Let $RS_i \leftarrow +x$ ($WS_i \leftarrow +x$) stand for $RS_i := RS_i \cup \{x\}(WS_i := WS_i \cup \{x\})$. We consider them as synchronization events.

Proposition 5.1 *After $RS_i \leftarrow +x$, $x \in RS_i$ continues to hold until EOT_i, and after $WS_i \leftarrow +x$, $x \in WS_i$ continues to hold until EOT_i.*

During a transfer, we are only interested in the contents of RS and WS of a transaction in *Active*, i.e., a transaction before its EOT. The RSs and WSs after EOTs are irrelevant to a transfer. So in the following discussion, when we say $x \in RS_i(WS_i)$,

$T_i \in Active$ is implied.

In the following propositions and lemmas, H stands for a history.

Proposition 5.2 *Let T_i be a transaction in Commit(H) with $r_i(x) \in T_i$. Then at least one of the following holds in Commit(H).*

1. $rl_i(x) < r_i(x) < ru_i(x)$
2. $RS_i \leftarrow +x < r_i(x)$

Proposition 5.3 *Let T_i be a transaction in Commit(H) with $w_i(x) \in T_i$. Then at least one of the following holds in Commit(H).*

1. $wl_i(x) < w_i(x) < wu_i(x)$
2. $WS_i \leftarrow +x < w_i(x)$

Proposition 5.4 *Let $p_i(x)$ and $q_j(x)$ be conflicting operations. If $pl_i(x)$, $pu_i(x)$, $ql_j(x)$, and $qu_j(x)$ are all in history H, then either $pu_i(x) < ql_j(x)$ or $qu_j(x) < pl_i(x)$ holds.*

Note that the above proposition is not trivial when there are concurrent data transfers.

Proposition 5.5 *Let T_i be a transaction in Commit(H).*

1. *If $r_i(x) \in T_i$ then $BOT_i < r_i(x) < EOT_i$, and if $w_i(y) \in T_i$ then $EOT_i < w_i(y) < COT_i$.*
2. *If $pl_i(x) \in S_i$ (synchronization event set of T_i) then $pl_i(x) < EOT_i < pu_i(x)$ holds.*

(1) is trivial. We have restated it for convenience. (2) follows from our deliberately not releasing locks during the transfer.

We define $req\text{-}o_i(x)$ as the time when T_i enters its critical section for the request $o_i(x)$. Let $StartTransf(x)$ denote the time when a transfer of x starts. Let $OtoP(x)$ denote the action of transferring x from OPT to PES. We also use it to denote the completion time of the transfer. The meaning and usage of $PtoO(x)$ are similarly defined. $req\text{-}o_i(x)$, $StartTransf(x)$, $OtoP(x)$, and $PtoO(x)$ are all considered as synchronization events. Therefore, our history, consists not only of actions of transactions and the scheduler, but also of actions of transfer processes.

Proposition 5.6 *Let $r_i(x) \in T_i$, and assume that $BOT_i < StartTransf(x)$ and $OtoP(x) < EOT_i$. Then $rl_i(x) \in S_i$.*

Proof: $T_i \in$ *Active* at $StartTransf(x)$, and T_i does not complete during $OtoP(x)$. Only cases T1.2 and T1.5 apply to this situation, and in either case a read-lock is set on x for T_i. □

Lemma 5.1 *Let T_i and T_j be in Commit(H). If $r_i(x) < w_j(x)$, then $EOT_i < EOT_j$.*

Proof: By Propositions 5.2 and 5.3, at least one of A and B, and one of C and D hold.

A) $rl_i(x) < r_i(x) < ru_i(x)$ B) $RS_i \leftarrow +x < r_i(x)$

C) $wl_j(x) < w_j(x) < wu_j(x)$ D) $WS_j \leftarrow +x < w_j(x)$

If we can show that under each minimal combination of the conditions, namely A and C, A and D, B and C, and B and D, $r_i(x) < w_j(x)$ implies $EOT_i < EOT_j$, then the lemma follows immediately.

A AND C

From Proposition 5.4, either $ru_i(x) < wl_j(x)$ or $wu_j(x) < rl_i(x)$ holds. Together with the condition of the lemma, $r_i(x) < w_j(x)$. We have

$$rl_i(x) < r_i(x) < ru_i(x) < wl_j(x) < w_j(x) < wu_j(x).$$

By Proposition 5.5 (2), $EOT_i < ru_i(x)$ and $wl_j(x) < EOT_j$. It follows that $EOT_i < EOT_j$.

B AND C

If A is also true, then so is 'A and C,' which has been dealt with. So, assume A is not true. Then $x \in OPT$ at $req\text{-}r_i(x)$. Since $wl_j(x) \in S_j$ by C, x must be transferred during the life-time of T_i or T_j. We consider the transfers $OtoP(x)$ and $PtoO(x)$ separately.

Case 1: $OtoP(x)$.

Consider the two subcases.

(1) $x \in WS_j$ at $StartTransf(x)$. Since $wl_j(x) \in S_j$, this implies $T_j \in Active$ at $StartTransf(x)$, i.e. $StartTransf(x) < EOT_j$. Note that $EOT_i < StartTransf(x)$ must hold. Otherwise, $T_i \in Active$ at $StartTransf(x)$, which implies that only Case T1.5 is possible. In Case T1.5, x will not be write-locked, a contradiction to C. So $EOT_i < StartTransf(x) < EOT_j$.

(2), $x \notin WS_j$ at $StartTransf(x)$. This implies $OtoP(x) < req\text{-}w_j(x)$. The only nontrivial case is that $OtoP(x) < EOT_i$. By Proposition 5.6, $rl_i(x)$ is in S_i, which must be set during the transfer. By Proposition 5.4 and 5.5(2), we get

$$rl_i(x) < EOT_i < ru_i(x) < wl_j(x) < EOT_j.$$

Case 2: $PtoO(x)$.

At $req\text{-}w_j(x)$, $x \in PES$, and $StartTransf(x) < EOT_j$ holds. Otherwise, in either Case T2.3 or T2.4, T_j would complete during or before the transfer, which means

$w_j(x) < r_i(x)$, a contradiction to the condition of the lemma. Under $StartTransf(x) < EOT_j$, there are two possibilities: T_j completes during the transfer or not. The first possibility will lead to a contradiction as just discussed. Under the second possibility, only Case T2.3 is possible and $x \in WS_j$. If $EOT_j < EOT_i$, $WS_j \cap RS_i = \phi$ would be checked in validating T_j, and one of T_i and T_j would be aborted, a contradiction. So, we have $EOT_i < EOT_j$.

A AND D

If C is also true, so is 'A and C,' which has been dealt with. So, assume C is not true. Then $x \in OPT$ at *req*-$w_j(x)$. Since $rl_i(x) \in S_i$, x is transferred. As with the case 'B and C,' consider the possible transfers separately.

Case 1: $PtoO(x)$

Trivial when $EOT_i < PtoO(x)$. Assume, therefore, $PtoO(x) < EOT_i$. Then $x \in RS_i$. If $EOT_j < EOT_i$, $WS_j \cap RS_i = \phi$ would be checked in validating T_j, and either T_i or T_j would be aborted, a contradiction. So, $EOT_i < EOT_j$.

Case 2: $OtoP(x)$

We show that this case is impossible by considering the following two cases.

(1), *req*-$r_i(x) < StartTransf(x)$. Then $x \in RS_i$ and $x \in WS_j$. Case T1.5 will prevail. Either, T_i aborts or finishes its step C1 during the transfer, which means no $rl_i(x)$ is set, a contradiction to A; or, T_i does not finish its step C1 during the transfer, which means $EOT_j < EOT_i$ and $WS_j \cap RS_i = \phi$ is checked so that one of T_i and T_j is aborted, a contradiction to the lemma's condition.

(2), $StartTransf(x) <$ *req*-$r_i(x)$. If T_j is not in *Active* at $StartTransf(x)$, then $w_j(x)$ already happened before $StartTransf(x)$. So, $w_j(x) < r_i(x)$, a contradiction. If T_j is in *Active* at $StartTransf(x)$, then either T_j finishes its step C1 during the transfer (Case T1.4 or T1.5) which means $w_j(x) < r_i(x)$, or T_j gets a write lock on x (Case T1.3), which implies C is true, a contradiction to the above assumption.

B AND D

If either A or C is true, it is already proved. Assume both A and C are false. So, $x \in OPT$ both at $req\text{-}r_i(x)$ and at $req\text{-}w_j(x)$. If x is not transferred from OPT to PES or is transferred after both EOT_i and EOT_j, the lemma is already proved in the last chapter. The case $EOT_i < StartTransf(x) < EOT_j$ is straightforward. The case $EOT_j < StartTransf(x) < EOT_i$ is not possible; otherwise, $WS_j \cap RS_i = \phi$ would be checked and one of them would be aborted. The only thing left is the case where $StartTransf(x)$ precedes both EOT_i and EOT_j. This is Case T1.5. EOT_j still cannot precede EOT_i since, otherwise, $WS_j \cap RS_i = \phi$ would be checked. □

Lemma 5.2 *Let T_i and T_j be in $Commit(H)$. If $w_i(x) < w_j(x)$, then $EOT_i < EOT_j$.*

Proof: As in the proof of Lemma 5.1, we prove that, under each minimal combination of $(A \vee B) \wedge (C \vee D)$, the lemma is true, where

A) $wl_i(x) < w_i(x) < wu_i(x)$ B) $WS_i \leftarrow +x < w_i(x)$
C) $wl_j(x) < w_j(x) < wu_j(x)$ D) $WS_j \leftarrow +x < w_j(x)$

A AND C

Similar to A and C of Lemma 5.1.

B AND C

Assume A is false. Then $x \in OPT$ holds at $req\text{-}w_i(x)$.

Case 1: $OtoP(x)$.

Assume D is also true. Only in Case T1.3 can both C and D hold. In Case T1.3, i.e., when $W = \{T_k\}$, conjunctively with C and D, implies $T_k = T_j$. So, $T_i \notin Active$ at $StartTransf(x)$, and therefore, $EOT_i < StartTransf(x) < EOT_j$.

Now assume that D is false. Then $OtoP(x) < wl_j(x)$. Since A is false by assumption, only Case T1.4 or T1.5 is possible. T_i will finish its step C1 during $OtoP(x)$ in both cases. It follows that $EOT_i < wl_j(x) < EOT_j$.

Case 2: $PtoO(x)$.

$PtoO(x) < req\text{-}w_i(x)$. $EOT_j < StartTransf(x)$ will lead to a contradiction since in both Cases T2.3 and T2.4, T_j would finish before the end of the successful transfer. That is, $w_j(x) < req\text{-}w_i(x) < w_i(x)$, which contradicts the condition of the lemma. Therefore, $StartTransf(x) < EOT_j$. Only in Case T2.3 will a transfer be successful in such a situation. x is in both WS_i and WS_j after the transfer. And both $w_i(x)$ and $w_j(x)$ will be executed in the corresponding C1 steps. Thus $EOT_i < w_i(x) < w_j(x) < EOT_j$.

A AND D

Assume C is false. $x \in OPT$ holds at $req\text{-}w_j(x)$.

Case 1: $PtoO(x)$.

Consider the following two cases:

(1) $EOT_i < StartTransf(x)$. Trivial, since T_i must complete before the end of the successful transfer.

(2) $StartTransf(x) < EOT_i$. Either T_i completes before the end of transfer, which is trivial; or $PtoO(x) < EOT_i$, which is similar to the discussion in Case 2 of B and C, i.e., $EOT_i = w_i(x) < w_j(x) = EOT_j$.

Case 2, $OtoP(x)$.

This case is impossible. We show that no matter whether $T_j \in Active$ or not at $StartTransf(x)$, it leads to a contradiction.

(1) $T_j \notin Active$. $EOT_j < w_j(x) < StartTransf(x) < w_i(x)$, a contradiction.

(2) $T_j \in$ *Active*. Since C is false by assumption, only Case T1.4 or T1.5 is possible. In either case, T_j will finish its step C1 before the end of transfer. Again, $w_j(x) < w_i(x)$, a contradiction.

B AND D

Similarly, assume both A and C are false. $x \in OPT$ both at $req\text{-}w_i(x)$ and at $req\text{-}w_j(x)$. If no transfer ever happens, $EOT_i = w_i(x) < w_j(x) = EOT_j$. Otherwise, since no lock on x is set for T_i or T_j, both T_i and T_j must have finished their step C1 before the end of transfer. So, $EOT_i < w_i(x) < w_j(x) < EOT_j$. □

Lemma 5.3 *Let T_i and T_j be in Commit(H). If $w_i(x) < r_j(x)$, then $EOT_i < EOT_j$.*

Proof: By Proposition 5.5(1), $EOT_i < w_i(x) < r_j(x) < EOT_j$. □

Theorem 5.1 *The transfer algorithm is correct. Under the interference of data transfer processes, the concurrency control algorithm still ensures serializability.*

Proof: Immediate from Lemmas 5.1 to 5.3, since EOT's are totally ordered. □

Chapter 6

Keeping the Partition Up-to-Date

The problems of how to predict conflict distribution and how to decide the membership of a data item are open and application dependent. There are many factors, such as the type of transactions, that can affect them. The database system administrator can also play an important role here. There are two principles that should govern the handling of these problems. The first is that the cost of running the controller should be low. The second is that the controller should invoke transfers as infrequently as possible. Otherwise, the overheads may well offset the benefit gained from our hybrid scheduler.

In this chapter, we present a solution based on history recording. It automatically traces changes in conflict rate and invokes transfers when the conflict rate reaches some thresholds. Practically, it is desirable to incorporate the method as a part of the scheduler. However, for the sake of simplicity, we consider it as a separate module. We call the module *re-partition controller* or just *controller*.

We first give a general picture of the controller and discuss some issues that should be of concern in the design. Then we build a specific controller. Finally, we discuss some correctness issues related to the controller.

6.1 Design Issues

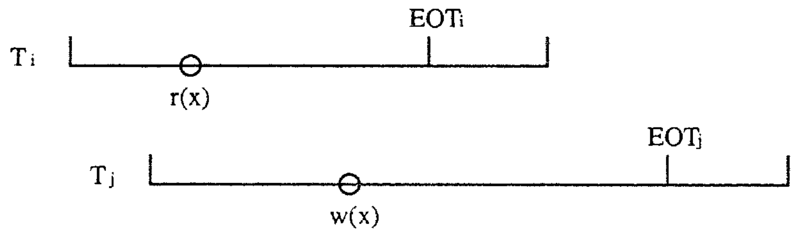
The automatic re-partition controller maintains a table to record potentially harmful conflicts and make decisions as to when transfers are needed. We call this table *conflict table* (CT). Each data item has an entry in it. The entry for x, denoted E(x), contains, among other information, (1) *status*, indicating if x is in OPT or PES from the stand-point of the controller, (2) *counter*, recording the number of potentially harmful conflicts on x in the current time interval, and (3) t_0, the last time the *counter* is reset. We refer to a field of E(x) by "E(x).", e.g., E(x).*counter*. To reduce the storage space needed, a data item with no co tention on it for some time interval, say θ, will be purged. Such a data item must be in OPT.

The controller provides three tunable parameters, θ, c_1, and c_2, to the database administrator. θ controls the frequency of resetting *counter*. The controller resets a counter every θ time units. c_1 and c_2 are thresholds measured by number of potentially harmful conflicts. For any data item x, once E(x).*counter* $> c_1$ within θ (may be less than θ), x should be transferred to PES if it is not there. Symmetrically, if E(x).*counter* $< c_2$ during a period of θ time units, x will be transferred to OPT if it is not there. We require that $c_1 \geq c_2$

6.1.1 Counting Conflicts

Detecting conflicts is an intrinsic function of the scheduler. So, the controller need not detect conflicts on its own. Rather, it is informed by the scheduler whenever a conflict is detected.

As stated in Section 4.2, the specific definition of harmful conflicts depends on the concurrency control algorithm used. In a 2PL method, every pair of conflicting operations from two concurrent transactions is considered as a harmful conflict, while in SFO, only a pair of conflicting operations whose order is different from the order of EOT's of their transactions is considered as a harmful conflict. Of course, the two transactions involved in a harmful conflict in SFO must be executing concurrently. In the remainder of this chapter and the next, we refer to a potentially harmful conflict simply as a conflict.



The conflict is not harmful in optimistic scheme, but is harmful in 2PL scheme.

Fig. 6.1 Difference in deciding harmful conflicts

It is interesting that there is a difference in counting conflicts between locking and optimistic methods. As illustrated in Fig 6.1, in SFO, the conflicts between T_i's reads and T_j's writes are not detected when $EOT_i < EOT_j$. However, in a 2PL method, these conflicts are detected. This difference should be considered in setting c_1 and c_2.

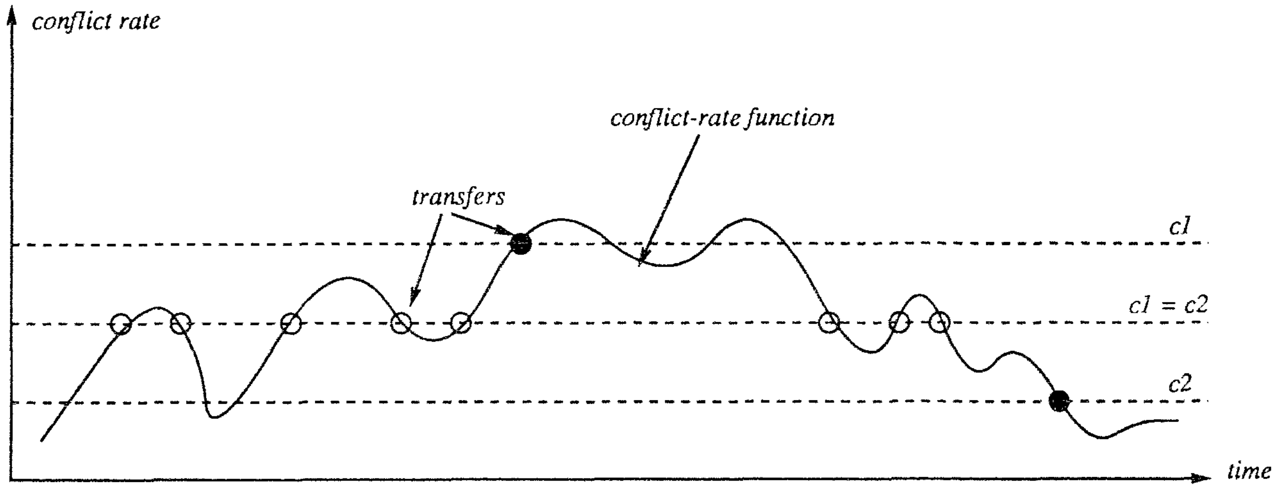
Counting conflicts in PES is easy. Whenever a lock cannot be set, the controller counts the number of the locks and ungranted lock requests conflicting with it. All the information is available from the locking table.

There is, however, some problem in counting conflicts in OPT. For SFO, there is no problem, since the scheduler does not make decision until it has checked all the conflicts. For backward checking algorithms such as SBO and PBO, because a scheduler aborts the transaction and stops the validation once a non-empty intersection is found, some conflicts are not counted in. One way to solve the problem is to let the scheduler check all the intersections. It will, however, incur some cost for information useful only to the controller. Another way is to let the inaccuracy exist and lower the parameter c_1 to compensate for the missed conflicts. We can even let $c_1 < c_2$. The problem of this approach is that it is hard to know the percentage of missed conflicts. So the parameter c_1 may very easily become meaningless.

6.1.2 Setting Parameters

The parameter θ should be large enough. By Assumption 5.1, θ should be greater than twice the maximum transaction life-time. The longer θ is, the less frequent transfers will be and the less overheads the controller will incur. A *counter* is a monotonically increasing function of time within a period of θ. So, it is easy to determine that a *counter* has reached the threshold c_1 and to initiate a transfer from OPT to PES. On the other hand, we can not say a *counter* reaches (less than) the threshold c_2 until the whole period of θ has elapsed. Therefore, a longer θ does not mean our system is less sensitive to conflict increase, but it does mean less sensitive to conflict decrease. In other words, a longer θ means more pessimistic view. However, if θ is too long, the conflict rate may vary a lot during θ. So, we simply lost sensitivity.

Let's consider c_1 and c_2. Not only are the absolute values of c_1 and c_2 important, but also is the difference between c_1 and c_2. If $c_1 = c_2$, some data items with conflict rates around c_1 will be subject to a lot of transfers between the two parts of the database. The larger $(c_1 - c_2)$ is, the smaller the number of transfers will be. Fig. 6.2 illustrates this. In the figure, the unfilled circles represent the transfers when $c_1 = c_2$. The filled circles represent the transfers when $c_1 > c_2$. We can see that the number of transfers when $c_1 = c_2$ is much greater than that when $c_1 > c_2$. That is the main reason for using two thresholds instead of one. The curve in the figure is *conflict-rate function* which is considered, for simplicity, to be a continuous function of time. However, we should point out that this figure is for illustrative purpose only . In fact, it is the integral of the conflict-rate function, i.e., the number of conflicts, rather than the conflict-rate function itself that should be compared with the thresholds. Generally speaking, the greater $(c_1 - c_2)$ is, the more stable the partition is, and so the less cost transfers incur. However, a larger $(c_1 - c_2)$ means more optimistic accesses to data items with higher conflict rates and more pessimistic accesses to data items with lower conflict rates. It will have negative impact on the performance when $(c_1 - c_2)$ is too large.



Conflict rate change on a data item and the transfers by different thresholds settings.

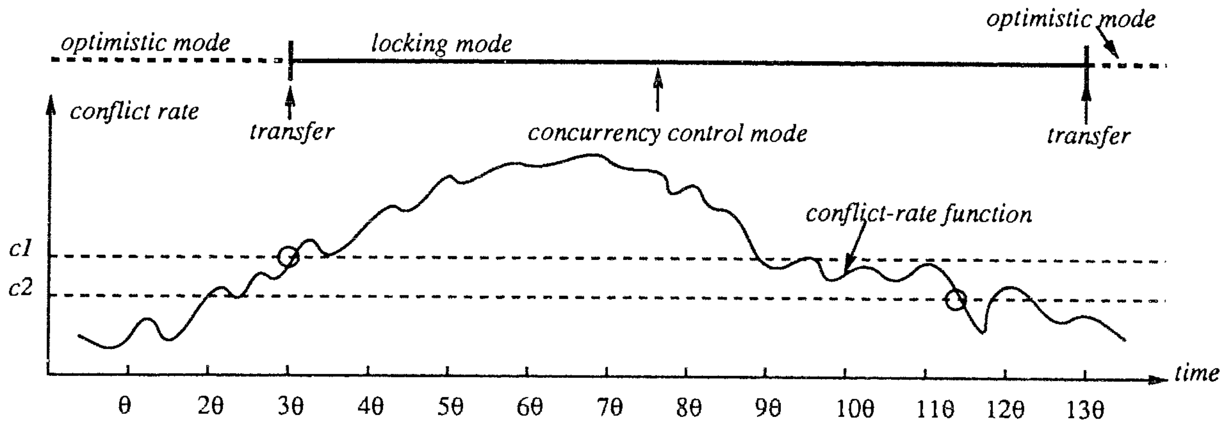
Fig. 6.2 Thresholds

The values of θ, c_1, and c_2 are related to each other. They are also application dependent. Some factors, such as resources available (number of CPU's and disks, etc.), are very important in determining them. We haven't done any study on establishing mathematical model for determining the value of the parameters. Here we just present a superficial understanding. Given a specific environment, suppose we know that an optimistic algorithm performs very well when conflict rate is less than r_1, and it performs tolerably when conflict rate is between r_1 and r_2 $(r_1 < r_2)$. Then for a hybrid scheduler using this optimistic algorithm, c_1 and c_2 should be related to θ, r_1, and r_2. For example, suppose conflict rate will increase from r_1 to r_2 in θ linearly with time. Then c_1 could be set as $(1/2)\theta(r_1 + r_2)$, i.e., the integral of conflict rate function over θ. Also we should consider the difference in counting conflicts between locking and optimistic schemes.

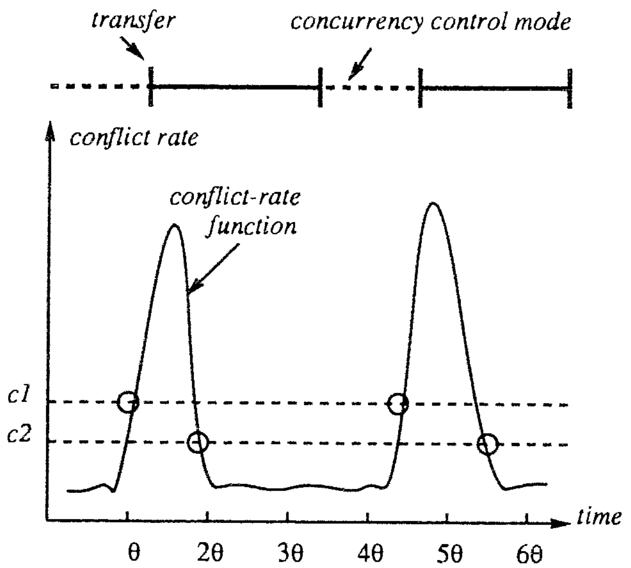
6.1.3 Working Environment

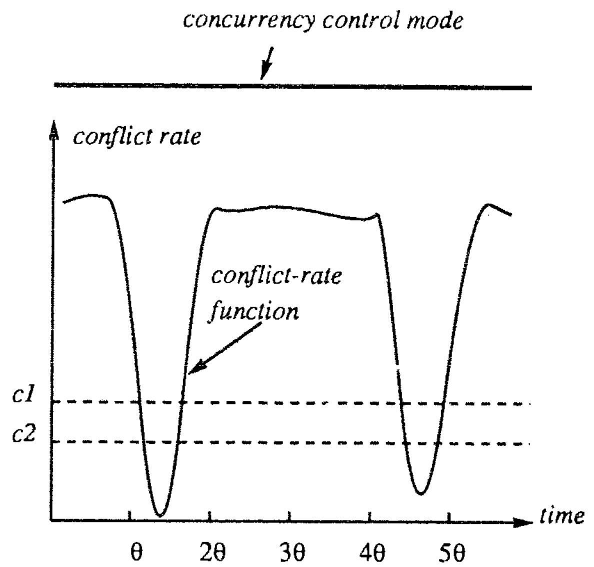
Our controller is based on the assumption that the conflict rate on any data item does not change sharply. As in Fig. 6.2, we consider the conflict rate on an item as a function of time. The controller will perform acceptably when the function changes slowly, as shown in Fig. 6.3.a. In Fig. 6.3, the curve still represents the conflict-rate function. The horizontal line above the figure represents the concurrency control mode on the data item. A dashed line indicates the optimistic mode, while the solid line indicates the locking mode, and a vertical bar represents a transfer. Fig. 6.3.a also shows that the controller is not sensitive to the small fluctuations in the conflict rate since it employs two thresholds.

For large "pulses", however, the controller behaves differently. It is insensitive to negative pulses, since we count the number of conflicts in θ time interval (see Fig. 6.3.c). On the other hand, it is sensitive to positive pulses and works poorly in this situation (see Fig. 6.3.b). Here, we say that the controller does not work well not only because it will start many transfers, but also because the concurrency control mode does not match the conflict rate. And the latter is more serious. When a positive pulse occurs, we are still in optimistic mode. Since conflicts are detected only during validation, detection tends to be late. When the controller discovers that $counter > c_1$ and starts transferring the item to PES, the transfer process has to wait at step T1.4a or T1.5a (see Chapter 5). Since, at this time, all the accesses causing the pulse are in optimistic mode, many of the transactions involved will be aborted. Only when all the transactions that are involved in the conflict have finished, can the transfer process finish. However, at this moment, the pulse has already peaked and the conflict rate is low again. Then we have to access the low conflict item under locking mode for a certain period of time until it is transferred back to OPT. However, such pulses are probably rare in practice. We even doubt they could actually happen. Furthermore, since we use the number of conflicts which is the integral of the conflict rate function, the harmful effect of positive pulse is not likely to be significant.



(a) Conflict-rate function and access mode (transfers), when conflict rate changes slowly. Performs well.





(b) Sensitive to positive pulses. Performs poorly.

(c) Insensitive to negative pluses. Performs well.

Fig. 6.3 Sensitivity of the method to conflict-rate functions

From the above discussion, we believe that our controller can perform well in practical applications.

6.2 Building the Controller

The controller employs a process called Record Conflict (RC). Whenever the scheduler detects a conflict on an item, say x, it asks the controller to start a RC process to record the conflict by increasing E(x).*counter*. If x is in OPT and E(x).*counter* $> c_1$, RC starts a transfer process to transfer x to PES and reset t_0 and *counter* to the current time and 0, respectively.

If the current time t is greater than or equal to $t_0 + \theta$, then the RC resets t_0 and *counter* to t and 0, respectively, and determines if a transfer of x to the other part of the database is needed.

Maintaining the CT only when a conflict is detected is not sufficient. There are still some problems left: (1) the time to start transferring x from PES to OPT, and (2) promptly finding those data items on which there has been no contention for a period of time. Let us analyze these problems in detail. For a PES item x, when a conflict on it is detected, the RC process may find that $t - \mathrm{E}(x).t_0 \geq \theta$ and E(x).*counter* $< c_2$. This is the condition to transfer x to OPT. However, this may not be the right time to do so. A transfer process may have to wait or abort itself due to the contention, or a transaction may be rolled back if we transfer x to OPT. This problem itself is not difficult to solve. Since a conflict always involves a write operation, in this case, when the write-lock involved in the conflict is released, we can start the transfer if it is still appropriate. The second problem is more difficult. For an item x in OPT, the condition for dropping E(x) is ($t - \mathrm{E}(x).t_0 \geq \theta$) and (E($x$).*counter* $= 0$). However, a RC process is started only when a conflict is detected. The condition (E(x).*counter* $= 0$) is not detectable by RC only. When $x \in$ OPT, we expect that the conflict on it will be rare. It is very likely that the accesses to x encounter no contention for a sufficient long period of time. It is even fairly likely that x is not accessed for a long time. Dropping E(x) in a timely manner has a favorable impact on performance of the controller. A similar situation occurs when $x \in$ PES. The conflict rate on a PES item may drop to 0 within θ. So, we need some type of process that is started even when there is no contention on x at all. We use a process called Transfer & Delete (TD). It is designed to solve both problems (1) and (2).

Now let us consider the appropriate time to start a TD process. As we discussed, the transfer of a data item from PES to OPT can be started when a write-lock on it is released. So, we can associate a TD with every write operation. It is, however, not sufficient to solve problem (2). Associating TDs with all read operations is, on the one hand, too frequent, and on the other hand, not sufficient when a data item is not accessed. Starting a TD periodically can guarantee sufficient checks regardless of whether a data item is accessed or not. But this method may become time-consuming if not properly designed. Here, we suggest a method based on periodical checking.

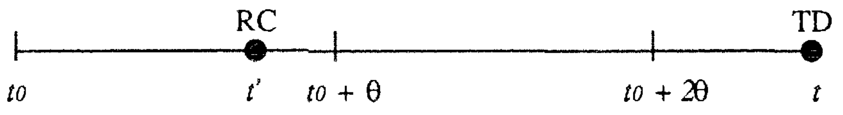


Fig. 6.4 Time for TD

Consider problem (2) only. An entry in the CT needs to be checked by a TD only if it is not checked by a RC for at least θ time units. So the problem is converted to selecting those entries that have not been checked by RCs for θ. Let t be be the time a TD checks an entry E(x). If $t - t_0 \geq 2\theta$ then E(x) must have not been checked by a RC for at least θ time units. Otherwise, t_0 would have been reset by the last RC (see Fig. 6.4). We "sort" all the entries in the CT according to field t_0, in ascending order. So, when we scan from the beginning, we reset the entries (and performs necessary transfers) as long as $t - t_0 \geq 2\theta$ holds for the entries, where t is the current time. Now let us see how to "sort" the CT. We link all the t_0 fields together to form a queue, denoted Q, with its elements arranged in the ascending order. Because the CT entries are created and the t_0 fields are reset all in a linear order, the queue can be created and maintained incrementally. When a RC creates or resets an entry, it links the entry to the end of the queue, because its t_0 field now has the largest value. This can be done very quickly. Every 2θ time units, we start a TD to scan the queue from the beginning. The entry scanned, say E(x), is either dropped if x is in OPT, or transferred to OPT if it is in PES.

To solve the problem (1), we use a small trick. When a RC finds that x should be transferred to OPT, it only clears *counter* without setting t_0. So, when a TD finds $x \in$ PES, either there is no new conflict on it for θ, or it was already checked by a RC and decide that it should be transferred.

In what follows, we describe RC and TD. For each combined concurrency control algorithm, RC and TD may be customized, just like transfer algorithms.

The following is a description of RC. *Rlock*, *Rwait*, *Wlock*, and *Wwait* are the same as those in the transfer algorithm given in the last chapter.

- When the scheduler cannot set a lock on x because of contention, a RC does the following:

```
begin critical section
    % All the fields that appear in this critical section are fields of E($x$)
    If E($x$) is not in the CT % See Remark 1
        then allocate an entry for it
            $status$:=OPT
            $t_0$:=$gettime()$
            $counter$:= 0
            link it to the end of Q
    If the scheduler wants to set a read-lock
        then $counter := counter + 1 + |Wwait|$
    If the scheduler wants to set a write-lock
        then $counter := counter + |Rlock| + |Wlock| + |Wwait| + |Rwait|$
    $t$:=$gettime()$
```

```
        If $t - t_0 \geq \theta$
          then if $counter \geq c_2$
               then $counter := 0$ % $x$ remains in PES
                    $t_0 := t$
                    link E($x$) to the end of Q
               else $counter := 0$ % See Remark 2
    end critical section
```

- When the scheduler validates a transaction T_i and finds conflicts, a RC does the following:

```
    begin critical section
        For every $x$ that appears in an intersection
            If $x$ appears in $n$ intersections
              then if $x$ has an entry in the CT
                     then E($x$).$counter$ := E($x$).$counter + n$
                     else allocate an entry in the CT for $x$
                          E($x$).$status$:= OPT
                          E($x$).$t_0$:= $gettime()$
                          E($x$).$counter$:= n
                          link E($x$)to the end of Q
            If E($x$).$counter > c_1$
              then E($x$).$counter$ := 0
                   E($x$).$t_0$ := $gettime()$
                   E($x$).$status$ := PES
                   start transferring $x$ to PES
                   link E($x$) to the end of Q
              else $t$:=$gettime()$
                   if $t - t_0 \geq \theta$
                     then E($x$).$counter$:= 0
                          E($x$).$t_0$:= $t$
```

```
                    link E($x$) to the end of Q
    end critical section
```

Remark 1: This "if" statement is very strange, especially because it sets the status to OPT. See the discussion in "Status and Memberships" later.
Remark 2: *counter* $< c_2$. x should be transferred to OPT. Only setting *counter* to zero will make $t - t_0 \geq 2\theta$ eventually become true. A TD will start a transfer at that time.

The following is a description of a TD

- When a TD is started

```
    E($x$):=head of Q
    begin critical section
        $t$:=$gettime()$
        while $t$ – E($x$).$t_0 \geq 2\theta$ do
            if E($x$).$status$= OPT
              then delete E($x$) from Q % See Remark 1
              else E($x$).$t_0$:=$t$
                 if E($x$).$counter < c_2$ % See Remark 2
                   then E($x$).$counter$:=0
                        E($x$).$status$:=OPT
                        start transferring $x$ to OPT
                   else E($x$).$counter$:=0 % See Remark 3
                 link E($x$) to the end of Q
            E($x$):=head of Q
            $t$:=$gettime()$
    end critical section
    store $t$. the next TD will be started at $t + 2\theta$
```

Remark 1: Let t' be the last conflict on x. Then $t' < t_0 + \theta$. Otherwise t_0 would have been reset by the RC. So, from t' to t $(t - t' > \theta)$, there was no new conflict on x. E(x) should be deleted from the CT.
Remark 2: Either there has been no new conflict on x for period of time longer than θ as is the case when $x \in$ OPT, or *counter* was reset by RC(s) for as least once between t_0 to t under the condition that $(t' - t_0 \geq \theta) \wedge (counter < c_2)$, where t' is the time when the RC checked E(x). In both cases, x should be transferred to OPT.
Remark 3: Here the condition means that, having been cleared by RC, *counter* has quickly accumulated more than c_2 counts within a time period less than θ. So, x should not be transferred. See discussion later.

6.3 Discussion

In this section, we discuss the automatic controller presented above.

6.3.1 About RCs and TDs

Since a TD takes time, the frequency of TD creations is less than $1/(2\theta)$ and varies from time to time.

RCs and TDs execute concurrently. They share some variables such as entries in the CT and links that form the queue. They also execute concurrently with transactions and transfers, and share the variables such as read-sets and *Wwait*'s. Actually, we need not protect the whole TD process by a global critical section. We need only to guarantee that a process (such as a RC) has the exclusive right to access the shared variable when it is active. For example, a RC started due to lock contention on x need only hold the right to access to the data structures for x in the locking table, the entry for x in the CT, and the related links. Another RC can run concurrently with it on another data item. Of course the queue manipulation should be synchronized. It is more efficient to manage those shared variables using monitors [21]. The reason for using critical sections is only to simplify the description.

One problem is that for an OPT item x, when a RC finds that $\textit{counter} > c_1$, it will start the transfer of x to PES. However, at this moment, $t - t_0$ may be greater than θ, which suggests that the conflict rate on x is actually not so high as to justify a transfer. As seen from the description of TDs (see Fig. 6.4), when $t - t_0 \geq 2\theta$, it is highly likely that E(x) is reset by a TD. Also, the last time E(x) is checked by a RC, $t' - t_0 < \theta$ must have been the case, where t' is the time at that moment. So, $t - t_0$ is not likely to be much greater than θ. Of course, we can use more sophisticated testing. For example, if $\textit{counter}/(t - t_0) > c_1$ then transfer; otherwise set t_0 to t and *counter* to n. A similar problem arises when $x \in$ PES.

For $x \in$ OPT, we immediately reset E(x) and transfer x to PES once E(x).$\textit{counter} > c_1$. For $x \in$PES, we do not reset E(x) when $\textit{counter} \geq c_2$. Instead, we wait until $t - t_0 \geq \theta$ holds. We *can* reset E(x) immediately; but it will cause more resets than our present method, especially when conflict rates are high. For each reset, we have to manipulate the queue by relinking the entry to the end of the queue. This task, even though can be executed fast, still incurs some overhead. That is why we choose not to do so.

6.3.2 Status and Memberships

In the CT, each entry has one of the two statuses, OPT or PES. In a database, each data item belongs to OPT, PES, or neither OPT nor PES. The last case occurs only when the data item is being transferred. Clearly, we want the membership of a data item to match its status in the CT. However, temporary inconsistencies may arise due to transfers. The change in status occurs before the change in membership. The relation between the two changes should satisfy the following properties:

(6.1) once the x's status is changed, its membership will eventually be changed.

(6.2) x's membership will not be changed unless its status is changed.

To preserve above properties, the controller must behave "correctly" even when there is inconsistency.

Now let us examine these temporary inconsistencies. An inconsistency happens when a transfer process is waiting or aborted for restart. Specifically, in the transfer algorithm for Algorithm 4.2, it happens in the case T1.4, T1.5, T2.3, or T2.4. Notice that, when an RC records conflicts, it does not check the status. So, even with the presence of inconsistencies, RCs still work normally. A TD does check status. So, let us consider each case where TD is checking an entry, say E(x), while there exists a inconsistency on x. First, if E(x).*status*= OPT, the inconsistency must be caused by T2.3 or T2.4. In Case T2.3, there won't be any contention on x during the course of inconsistency. So, E(x) can be deleted. For T2.4, there is still contention on x due to conflicting locks during the inconsistency. RCs will record the conflicts and reset E(x).*counter* according to the threshold c_2. (This time the membership of x is PES.) Note that a RC, when started due to conflicting locks, never sets a status to PES. Also notice that when a TD checks E(x), there has not been any new conflict on x for at least θ. So, we can safely delete E(x). When conflict occurs again while inconsistency is still there, that strange "if" statement will allocate an entry with status OPT. So, the Property (6.1) is satisfied. However, we should point out that this strange "if" statement need not even exist, because, if θ is long enough, a restarted transfer will have already transferred x to OPT before a TD can check E(x).

Second, let us consider the situation where E(x).*status*= PES. In this case the inconsistency must have been caused by T1.4 or T1.5. In both cases, the problems here are the same. This time, $x \notin (OPT \cup PES)$. There may be still some transactions that can access it in optimistic mode. So, there could be contention on it. RCs will record the conflicts, reset E(x) according to the threshold c_1, and even fire new transfers of x to PES. These transfers are duplicates of the waiting transfer. We will discuss how to suppress those duplicate transfers in the next paragraph. When a TD checks E(x), it tries to transfer x back to OPT. This transfer will, however, violate Assumption 5.1, and is suppressed too. So, E(x) will keep the status PES and the Property (6.1) is again satisfied.

Now we discuss how to suppress transfers. Situations where suppressing transfers is necessary were already discussed in the above paragraph. Here is another situation where a transfer should be suppressed. Suppose a data item x has just been transferred to OPT. But very soon, E(x).*counter* exceeds c_1 and a RC wants to transfer it back to PES. This transfer will violate Assumption 5.1. Although we argue that this kind of situation is very rare and can be prevented almost always by carefully choosing the three parameters, we should still take it into account. To suppress transfers, we add another field in E(x). This field, say *last-t*, records the time when the last transfer of x has completed. If there is an on-going transfer, the value of *last-t* is infinite. So, when a transfer is about to be initiated, the controller first checks if $t -$ *last-t* is greater than twice the maximal transaction life-time. We want to point out that setting the field when a transfer completes is not difficult to implement. This can be done when the controller receives an acknowledgement from the transfer process, or even can be done by transfer process itself in some specific implementations.

Since transfers are only started by RCs and TDs, the Property (6.2) is always satisfied.

From the above discussions and from the algorithms for transfer, it is easy to see that our system (for Algorithm 4.2) satisfies these properties. The above discussion can be thought of as a "correctness proof" of the re-partition controller.

Chapter 7

An Implementation Proposal

In this chapter, we propose an implementation for an adaptive concurrency control system based on Algorithm 4.2. The system integrates the three functions: concurrency control, transfer, and conflict recording and transfer initiation. The system is implemented by a combination of a *concurrency manager* (CM) and a TM. The CM provides functions such as *check-member()*, lock and unlock, read-set/write-set manipulation, conflict recording, transfer initiation, and transfer. When the TM receives a *Read* or *Write* request from a transaction, it asks the CM to check the membership and to set an appropriate lock or manipulate read-/write-set depending on where the item is (OPT or PES). When the CM informs the TM that the required work is done, the TM operates on the transaction's private work space and contacts the DM if necessary. When it receives an *End* request from the transaction, the TM asks the CM to check set intersections and release locks. Upon discovering conflicts in OPT, the TM is responsible for aborting transactions. The TM is also responsible for telling the DM to reflect the transaction's modification to the database if the transaction is validated.

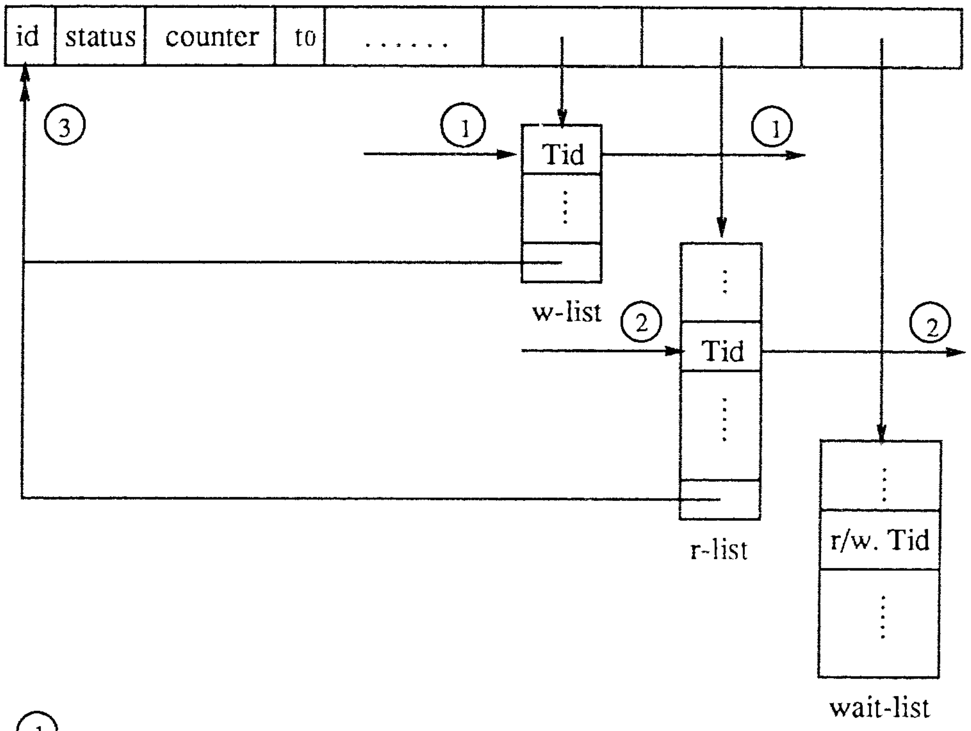
In this chapter, we focus our discussion on the CM. First, we give an overview of the CM and discuss the major data structures it maintains. Then we discuss the operational details of the CM. The discussion is organized according to the status. Finally, we comment on the proposal.

7.1 The Concurrency Control Table

In the CM, there are no separate processes for RCs, TDs, and transfers. They become a few more steps in the ordinary concurrency control activities of the CM. This reduction on the number and type of processes is made possible by a carefully designed table maintained by the CM. The table combines the functions of the locking table (for locking scheme) and the conflict table. It also facilitates the set manipulations for optimistic accesses. For such a multi-purpose object, it is hard to find a pertinent name. So, we simply call it *concurrency control table* (CCT). The CCT is organized as a hash table with the data item id as a key. Each entry in the CCT represents a data item. We use E(x) to denote the entry for x. As in the CT, for every PES item, there is a corresponding entry in the CCT, but for an OPT item which has not been accessed for a period of time, the entry corresponding to it is dropped.

An entry, say E(x), in the CCT consists of two parts: the header and the lists (see Fig. 7.1). The header contains, among other information, 1) *id*, 2) *status*, 3) *counter*, 4) t_0, and 5) three pointers, *wp*, *rp*, *wap*. *id* is the identifier of x. *counter* and t_0 are the same as those in CT, except that t_0's are not linked to form a queue for TDs. *status* now represents both the status and membership of x. This time, however, *status* has five possible values: OPT, PES, WtoP, WtoO, and PtoO. The meaning of these status is as follows:

- OPT: x is being accessed in optimistic mode.
- PES: x is being accessed in locking mode.
- WtoP: A temporary status where a transfer of x to PES is waiting at step T1.4a or T1.5a. x is now accessed in optimistic mode (see R3 and W3).



(1) Linked to/from an element in another w-list. That element contains the same Tid.

(2) Linked to/from the element in other r-list. That element contains the same Tid.

(3) Points back to the header.

Fig. 7.1 An entry in CCT

- WtoO: A temporary status where a transfer of x to OPT is waiting at step T2.3a. x cannot be accessed in this status (see R3 and W3).
- PtoO: The CM has found that x should be transferred to OPT. But currently there is contention on x. So, the transfer is deferred until there is no contention on x. x is accessed in locking mode (see T2.4).

The transitions among different statuses are shown in Fig. 7.2. The pointers *wp*, *rp*, and *wap* point to *w-list* (write list), *r-list* (read list), and *wait-list* (waiting list), respectively. However, we do not call them read list or write list because the term "read list" is used to denote another data structure. The three lists form the list part of E(x), and are discussed below.

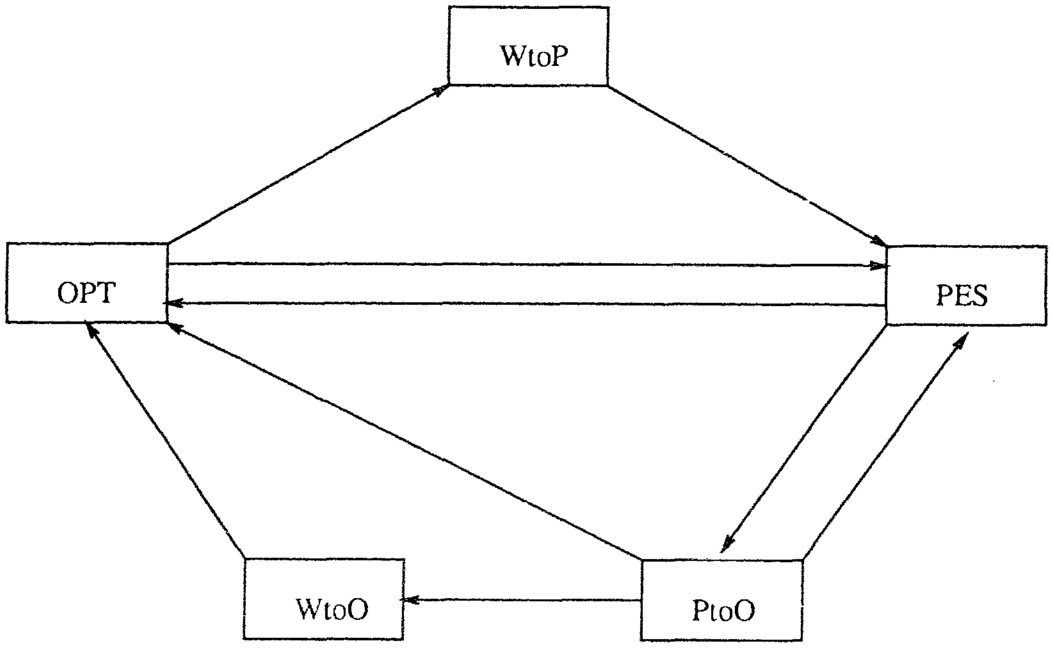


Fig. 7.2 Transitions among statuses

An element in *w-list* of E(x), when the *status* is PES, represents a write-lock on x. So, in this case, *w-list* has at most one element. The element, therefore, contains, among other information, the id of the transaction that owns the lock. All the write-locks owned by the same transaction are linked together, so that locks can be released fast (see type 1 links in Fig. 7.1). We call this list *write-lock list*. Note that a *w-list* is for a data item, but a write-lock list is for a transaction. When the *status* is OPT, an element in *w-list* indicates that x is in the write-set of the transaction whose id is stored in this element. In this case, there could be more than one element in the *w-list*. The type 1 links link all such elements with the same transaction id to form a list that represents the write-set of the transaction. At the end of *w-list*, there is a link (of type 3) pointing back to the header of E(x). The function of these links will be discussed later. When *status* is WtoP, the meaning of *w-list* is the same as the case where *status* is OPT, and when *status* is PtoO or WtoO, the meaning of *w-list* is the same as the case where *status* is PES.

The usage of *r-list* of E(x) is similar to that of *w-list*, except that all the elements for the read operations of the same transaction are linked together by the type 2 links, whether an element represents a read-lock or an element in the read-set. In this way, there is no distinction between the read-lock list and the read-set. We call this unique list *read-list*[1]. As we will see below, this unique list will not cause any problem in validation or lock operations, rather, it will simply make transfers easier. To manipulate the write-lock list, write-set, and read-list for a transaction, the CM uses pointers pointing to the heads of the corresponding lists.

The *wait-list* records all the locking requests on x. Each element in it contains the id of the transaction that submits the request and the mode of the request (read or write). To make the locking scheduler fair, we organize it as a FIFO queue. When *status* is OPT, *wait-list* must be empty. The CM takes care of the integrity constraints of the CCT. When E(x).*status* is PES, for example, *w-list* and *r-list* cannot both have elements simultaneously.

7.2 Operations of the CM

7.2.1 Operations in OPT

Read and Write Requests

[1]The motivation for separating write-set (list) from write-lock list is to perform validation quickly. The validation of a transaction involves the transaction's write-set (list), but not a write-lock.

When a read request, say, *Read(x)*, arrives from transaction T_i, the CM first checks x's membership by finding the entry E(x) in CCT and checking its *status* field. If there is no entry in CCT for x, x must be in OPT. So, the CM allocates and initializes an entry for it. Now assume that *status* is OPT. The CM simply adds an element in E(x)'s *r-list*, puts T_i's id in it, links it to the head of the read-list for T_i, and modifies the pointer to the head of the read-list.

For a write request when *status* is OPT, the process is similar.

Validation

When an *End* arrives request on behalf of T_i, the CM first, starting from the head of the read-list for T_i, deletes all the elements in the read-list from the corresponding *r-list*'s. This has the effect of releasing all the read-locks and discarding the read-set for T_i. Note that, since T_i is no longer in *Active*, its read-set is useless now. Then the CM does the validation along the list for the write-set. When the CM reaches an element in the write-set, it traces the type 3 pointer to the header of the data item being checked, say, x. It inspects the *rp* field. The null pointer means no contention on x. Otherwise, it searches the *r-list* to find all the active transactions that conflict with T_i for x and records the number of conflicts in the *counter* field. It also records the ids of the conflicting transactions for the consideration of the TM. If *counter* $> c_1$, the CM puts x with the pointer to E(x) into the list *Transf* which contains all the items that should be transferred. Then it resets the t_0 and *counter* fields. If *counter* $\leq c_1$ but $t - t_0 \geq \theta$, the CM resets t_0 and *counter* too. The reader may realize that CM is doing the tasks that a RC process was supposed to do. But unlike RC, the *status* field is not set to PES right now. It is set when the CM transfers the item. When it finishes the operations on E(x), the CM drops the element from the *w-list* and proceeds to the next element in the write-set. When the CM finishes scanning the write-set of T_i, it reports the result of validation to the TM. If a transaction is aborted, the CM will discard the read-list, write-set, and write-lock list for the transaction. The transfers will be started at the end of the critical section for the validation. At the end of the critical section, all the data items that should be transferred are in *Transf*.

The *Drop* Queue

It may happen that, when deleting an element standing for optimistic access from the *r-list* or *w-list*, the CM finds that all the three lists of E(x) are empty. In other words, there is no access to x at this moment. (The CM can detect this because, once the *r-list* or *w-list* is empty, it will trace back to the header via the type 3 link). In this case, the CM checks if *counter* $(\theta/(t - t_0)) > c_1$. If so, the CM puts the header of E(x) in a queue called *Drop*. Otherwise, it drops E(x) from the CCT immediately. The *Drop* queue has a limit, say n. When the number of the elements in *Drop* reaches n, for every element in the first half of *Drop*, the CM compares it with the entry in the CCT. If they are the same, it means that the item has not been accessed for a while. So, if the entry has status OPT, the CM drops the entry from the CCT, and if the entry has status PES, the CM transfers it to OPT. The element in *Drop* is also deleted. If they are not the same or there is no such entry in the CCT, the CM just drops the element.

This mechanism is used to replace the periodically started TDs for finding those items that still occupy entries in the CCT but have not been accessed for some time. There will be more discussion on this later.

Transfers

Transferring an item, say x, to PES can be executed very fast. Since the transfer process has a pointer to E(x), locating E(x) is trivial. The R and W sets are just the *r-list* and *w-list*, respectively. The CM checks the R and W sets. In Case T1.1, T1.2, or T1.3 (see Section 5.2), the CM can transfer x immediately. So, it sets *status* to PES and sets locks on x if appropriate. For the read operations on x, there is no need to set locks, since the read-lock list and the read-set are organized as a unique read-list. To set a write lock, we have to delete x from the write-set and then link it to the write-lock list. Because the write-set is linked in one direction, we have to start from the head of the write-set to find the element right before x. Later, we will discuss some design considerations related to this issue. In Case T1.4 or T1.5, the

CM has to wait until the contention on x is resolved. So, it sets the *status* to WtoP and proceeds to transfer the next item in *Transf.*

7.2.2 Operations in WtoP

When an item, say x, is in status WtoP, only those transactions that have previously accessed it can access it. The requests from other transactions are blocked. Because a transaction that has previously accessed x has a copy of x in its private work space, for a read request from that transaction, the TM need not ask the CM, but for a write request, the TM should have the CM put the transaction in the write-set if it is not there yet. So, when the CM receives a read request, it simply puts the request at the end of *wait-list*. When it receives a write request, the CM may put it in *w-list* or *wait-list* depending on whether or not the transaction accessed it before.

In status WtoP, unlike the description in Chapter 5, the CM need not know whether the transactions involved in the contention have finished or not; it need only to wait until one of *w-list* and *r-list* becomes empty. When *r-list* or *w-list* becomes empty, the CM traces the type 3 link back to the header of the E(x) and checks the *status*. If *status* is WtoP, the CM sets it to PES and sets locks as appropriate. Tracing back to the header along the type 3 link whenever a *r-list* or *w-list* becomes empty will incur an additional cost only when the CM discards a read-set. But this cost is somehow compensated for by that clever way to finish a transfer's waiting period. Later, we will see this is further compensated for by not having TDs and the queue of t_0's. Tracing back to the header when releasing locks and doing validation is required anyway.

7.2.3 Operations in PES

The operations in PES are more straightforward. When a read request arrives, the CM checks if the corresponding read-lock can be granted. If so, it adds an element in the *r-list* and links it to the read-list of the transaction. Otherwise, it adds an element in the *wait-list* and records the conflicts by increasing the *counter*. For a write request, the process is similar.

When there is contention, after increasing the *counter*, the CM checks for the condition $t - t_0 \geq \theta$. If the condition is satisfied, the CM further checks if *counter* $< c_2$. If not, the CM resets the t_0 and *counter* to t and 0, respectively. If yes, the CM resets t_0 and *counter*, and then sets the *status* to PtoO. A transfer will be started when there is no contention on x (see the next section for more detail).

Releasing read-locks was described in section 7.2.1. Releasing write-locks is similar. One problem is that when we release a lock, we may need to check if the locking request at the head of the *wait-list* can be granted now; if so, we set the lock. This is implemented by utilizing type 3 links. If, when deleting an element, the *w-list* (*r-list*) it belongs to becomes empty, the CM goes back to the header of E(x) via the type 3 link.

It may happen that when releasing a lock, the CM finds that all the three lists of E(x) are empty. In this case, the CM checks if *counter*$(\theta/(t - t_0)) < c_2$. If so, it sets the *status* to OPT and resets t_0 and *counter* accordingly. Otherwise, it puts the header of the entry in *Drop*.

7.2.4 Operations in PtoO and WtoO

PtoO

For the read and write requests, the CM works in almost the same way as that for PES. There is a minor difference when contention is detected. After increasing the *counter*, the CM checks if *counter* $\geq c_2$. If so, it sets the *status* back to PES and resets t_0 and *counter*. Otherwise, if $(t - t_0) \geq \theta$, it resets the t_0 and *counter*.

When a lock is released so that there is no contention on x, i.e., the *wait-list* becomes empty, the CM transfers x to OPT. This transfer is fast. *Rlock* and *Wlock* sets are just the *w-list* and *r-list*. The *Rwait* and *Wwait* must be empty at this moment. Only Case T2.2 or T2.3 is possible. In Case T2.2, all the transactions that have locks on x are active. The transfer is straightforward. In Case T2.3, the condition $T_j \in Active$ can be implemented by a risky trick. If the read-list of T_j is empty, then we bet that T_j is not in *Active*. Since a transaction usually reads something before it ever writes, when the read-list is empty, it suggests that the read-list has been deleted at the beginning of the validation-and-write phase of T_j. Because deleting the read-list is protected in a critical section, there is no risk that, when the read-list is being deleted, a transfer process checks its emptiness. If we cannot accept this assumption, the CM has to ask the TM for the information. But it is not slow though. If $T_j \notin Active$, set *status* to WtoO. t_0 and *counter* need not be reset in this case.

WtoO

Access to x is not allowed in this status. So, the CM puts all the read/write requests in *wait-list*. When the write-lock is released, the CM sets *status* to OPT and resets t_0 and *counter* accordingly. If *wait-list* is not empty, the CM puts the requests in *w-list* and *r-list* accordingly.

7.2.5 About Empty *r-list* or *w-list*

The discussion about the operations that the CM performs when a *r-list*/*w-list* becomes empty is scattered all over previous sections. Some confusion may arise and some incomplete description may exist. This section serves the purpose of clarifying and completing the issues related to it.

When one of *r-list* and *w-list* of E(x) becomes empty, the CM traces back to the header via the type 3 link, and

Case 1: *status* = OPT

```
        if all the three lists ($r$-$list$, $w$-$list$, $wait$-$list$) are empty then
          $t := gettime()$
          if $counter(\theta/(t - t_0)) > c_1$
            then put the header to $Drop$
            else delete E($x$) from the CCT
    Case 2: $status$ = WtoP
        $status$ := PES
        $t_0 := gettime()$
        $counter := 0$
        if all the three lists are empty then put the header in $Drop$
    Case 3: $status$ = PES
        if $wait$-$list$ is not empty
          then set locks for elements in $wait$-$list$, until no lock can be set
          else % At this moment, all the three lists must be empty.
              $t := gettime()$
              if $counter(\theta/(t - t_0)) < c_2$
                then $status$ := OPT
                     $t_0 := t$
                     $counter := 0$
                else put the header in $Drop$
    Case 4: $status$ = PtoO
        % At this time $wait$-$list$ must be non-empty
        set locks for elements in $wait$-$list$ until no lock can be set
        if $wait$-$list$ becomes empty after setting locks
          then start transfer of $x$ to OPT
    Case 5: $status$ = WtoO
        % this time it must be $w$-$list$ that becomes empty
        if $wait$-$list$ is empty
          then delete E($x$) from the CCT
          else $status$ = OPT
              $t_0 := gettime()$
        $counter := 0$
        put the requests in *wait-list* to *r-list* and *w-list* accordingly
```

7.3 Discussion

Now, a data item may have one of the five statuses, and transfers may take place among all the statuses rather than just between OPT and PES. A transfer, however, can be implemented as a change of status plus some relinking between the write-set and write-lock list of a transaction (if necessary). Among the five statuses, only PtoO is new. When the status of an item becomes PtoO, the conflict rate on it is going down. So, we do not expect that the item will stay in the PtoO status for a long time before it changes to OPT. Even if the conflict rate goes up again, the item cannot stay in PtoO for a long time, because the status will change back to PES. This further reduces the duration that an item is in PtoO. The statuses WtoO and WtoP are just separated from $x \notin (OPT \cup PES)$. Since our automatic re-partition controller is based on the assumption that the conflict rate on an item changes slowly, transfers will be rather infrequent. Therefore, we expect that a data item almost always has status OPT or PES.

A nice feature of our implementation is that the amount of work for validation is only related to the size of a (small) write-set. It is *independent* of the sizes of a number of (large) read-sets.

TDs' tasks are now performed by the queue *Drop* and checking when one of *r-list* and *w-list* of an entry becomes empty. So, the size of *Drop* becomes important to performance. It should be somehow related to the average number of entries in the CCT, and should be tunable. Since checking when a *r-list*/*w-list* becomes empty is needed for other jobs, only the check for a *r-list* in OPT status could be considered as the additional cost to implement TDs' tasks. In this case, since *wait-list* must be empty, it is very likely that all the three lists are empty. When all the three lists are also empty, it is more likely that the entry is deleted than its header is put in *Drop*. Since we get rid of tasks of relinking t_0's to maintain the queue, and the processes for purging *Drop* can be made less frequently than the TDs, we can achieve better performance in this way than using TDs.

Each one of the write-lock list and the write-set (list) is implemented as a single chain. This single chain makes deleting an element from the middle of the chain slow. Actually, we can use a doubly linked chain to solve this problem. However, such deletion is only needed in a transfer. As we commented before, transfers are infrequent. Besides, when a data item is write-locked or is a member of a write-set, it is very likely that the transfer is blocked due to conflict. So, setting a lock for an item in a write-set or putting a write-locked item originally in a write-set occurs less frequently than transfers. That is why we have chosen to use a single chain.

Now we discuss issues about the critical section that protects validation in Algorithm 4.2. Since concurrency control is achieved jointly by the TM and CM, a question arises as to how to implement a critical section. Here we suggest two possible approaches. We can use the TM to achieve the effect of a critical section. When validating a transaction, the TM does not accept any request from other transactions, nor does it process read/write operations of other transactions. This approach can be easily implemented, and it actually stops the database accesses by other transactions. Another approach is to use the CM to achieve a similar effect. Since there are only two purposes of this critical section, i.e., (1) preventing another transaction from being validated concurrently, and (2) protecting the elements in the write-set of the transaction from being accessed by other transactions, the CM need only do the following: (1) do not accept another *End* request, (2) prevent access to such entries in the CCT that the data items they represent are in the write-set of the transaction undergoing validation. The accesses to the other data items (and the entries representing them in the CCT) are allowed. To impelment this approach, first, the CM must remember that it is validating some transaction. Second, the CM has to check along the *wp* down to the *w-list* to see if the item is in the write-set before it grants the access, since the CM does not know whether an element is in the write-set before it has actually scanned the element through the (type 1) chain for the write-set, and also since the access requests of other transactions come from different direction (from the header) from the scanning, for an access request to a data item. This approach is more complex, but it permits more concurrency. Adding one more status to the entries in the CCT is helpful in implementing this approach.

In our implementation proposal, there are no separate processes for RCs, TDs, and transfers. This eliminates the cost for starting processes. These functions are integrated into the normal concurrency control activities and are just a few more steps beyond concurrency control. Thus, we can construct, with quite a small additional cost, an adaptive concurrency control system that takes advantage of both optimistic and locking methods.

Chapter 8

Conclusions and Future Work

The major contributions of this thesis are:

1. We have developed a data-oriented concurrency control scheme that is adaptive and that takes advantage of both optimistic and locking schemes.

2. Based on a systematic study of the optimistic scheme, we have designed several hybrid concurrency control algorithms and developed a systematic procedure to combine optimistic and locking methods.

3. To make our system adaptive, we have designed algorithms for re-partitioning a database and for recording conflicts and starting re-partitioning.

4. We have also given an implementation proposal for an adaptive concurrency control system.

5. In addition, we have extended our scheme to multiversion databases.

We expect that, when the database is properly partitioned, our hybrid schedulers will achieve higher performance than pure locking or pure optimistic schedulers. Our expectation is based not only on the fact that optimistic scheme performs better than the locking scheme when conflict rates are low. It is also based on the fact that our hybrid schedulers can, (1) avoid conflict escalation to a large extent in the part of the database with low conflicts, and (2) confine deadlocks to the high conflict part. The only additional cost at the algorithmic level is the invocation of *check-member* for every read and write operation. It leaves plenty of room (in the sense of operating cost) to adopt the other two components to constitute an adaptive concurrency control system. Actually, a non-adaptive hybrid schedulers with a fixed partition can be a promising approach. Such a fixed partition can be drawn according to the types of data items. For example, in a banking database system, certain types of accounts, such as personal saving, constitute the OPT part of the database, while some other types of accounts constitute the PES part. When a database is partitioned according to the types of data items, the *check-member* function can be implemented very efficiently.

Our transfer algorithm is efficient. Carrying out one transfer is cheap. The cost of initiating a transfer process is perhaps greater than the cost for executing the body of the process. We can also group transfers to reduce the cost for starting transfer processes.

It is always desired that the knowledge and experience of the database system administrator could have a positive impact on performance, and we expect that this positive impact can be achieved by carefully designing and managing the way the controller and the DBA interact. If successful, this would provide an opportunity for incorporating higher intelligence into concurrency control. By letting only the controller, rather than the scheduler or re-partition processor, interact with the DBA, we restrict the influence of the DBA to only the performance and not the correctness of concurrency control. The automatic re-partition controller presented in this thesis is relatively closed and at a low-intelligence level. It is based on the assumption that conflict rate changes are slow. Definitely, finding a better controller to incorporate the DBA's intelligence is a promising research direction. However, a complicated system is not desired because the cost for running a controller must be small enough, so that it will not offset the benefit gained from the hybrid scheduler.

From the implementation proposal given in Chapter 7, we can see how cleanly the functions of the re-partition processor and the automatic controller can be integrated into the activities of the scheduler. However, there is one problem that we should point out. There are two reasons why the optimistic scheme could out-perform the locking scheme. First, the optimistic scheme detects less conflicts than the locking scheme (see Section 6.1.1) and it does not block a transaction. Second, an optimistic scheduler can be cheaper to run than a locking scheduler. Read-/write-set manipulations can be decentralized. They are cheaper than locking and unlocking operations (see Section 7.3). In our implementation proposal, the second reason is simply lost. It is left as future research to implement an adaptive concurrency control system, taking advantage of decentralized read-/write-set management.

Our approach is general, since the four basic assumptions in Chapter 1 are general. In particular, our approach can be applied to relational databases. It may also find its use in newer types of databases, such as deductive and object-oriented databases, because the optimistic scheme is superior to the other two schemes when there exists long internal thinking time in transactions [4], which is often the case in deductive and object-oriented databases. In object-oriented databases, transactions are often naturally nested [19, 28]. Extending our approach to nested transactions [26] does not appear to be straightforward. However, the idea of partitioning the database may prove valuable.

In general, we can partition a database into an arbitrary number, say n, of parts. As long as it ensures that any two synchronizing sections in every transaction overlap, serializability can be guaranteed.

Our work gives rise to an interesting research topic. Previous studies only vaguely talk about "high" or "low" data contention (or conflict rates). Nobody (to our knowledge) ever tried to define "conflict rate" rigorously and to define "high" and "low" quantitively. With our attempt to partition a database according to contention frequency, and with our attempt to mechanically determine (by the repartition controller) the conflict rate of every data item and thereby determine its membership, there is an important question of defining conflict rate in a sensible way and finding exact thresholds on conflict rates to achieve a good performance in optimistic and locking methods. It is also debatable what items, highly accessed or highly contended, should be put into PES. For example, should a highly accessed item with 95% of its accesses being *Reads* be put in PES or OPT even though currently there are only read accesses?

Important work that has not been done is the performance analyses of our approach. We plan to do a simulation study to test our scheme against the locking and optimistic schemes in the near future.

It is also promising to extend our approach to distributed databases and replicated databases.

Bibliography

[1] D. Agrawal and A. El Addadi. Performance characteristics of protocols with ordered shared locks. Technical Report TRCS 90-13, University of California at Santa Barbara, 1990.

[2] D. Agrawal and A. El Addadi. Constrained shared locks for increasing concurrency in databases. Technical Report TRCS 91-20. University of California at Santa Barbara, 1991.

[3] D. Agrawal et al. Distributed multi-version optimistic concurrency control for relational databases. In *Proc. IEEE COMPCON'86*, pages 416–421, San Francisco, California, Mar 1986.

[4] R. Agrawal, M. J. Carey, and M. Livny. Models for studying concurrency control performance: Alternatives and implications. In *Proc. of ACM-SIGMOD 1985 Int'l Conf. on Management of Data*, pages 108–121, 1985.

[5] B.R. Badrinath and K. Ramamritham. Synchronizing transactions on objects. *IEEE Trans. on Computers*, C-37(5):541–547, 1988.

[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[8] B. Bhargava. Performance evaluation of the optimistic approach to distributed database systems and its comparison to locking. In *Proc. 3rd Int'l Conf. on Distributed Computing Systems*, pages 466–473, 1982.

[9] H. Boral and I. Gold. Towards a self-adapting centralized concurrency control algorithm. In *Proc. of SIGMOD 1984 Annual Meeting*, pages 18–22, 1984.

[10] J. T. Canning, P. Muthuvelraj, and J. Sieg. An adaptive concurrency control algorithm: Merging optimistic and pessimistic techniques. In *Advanced Database System Symposium'89*, pages 187–191, Kyoto, Japan, 1989.

[11] M. J. Carey and M. R. Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proc. 10th Int'l Conf. on Very Large Data Bases*, pages 107–118, 1984.

[12] S. Ceri and S. Owicki. On the use of optimistic methods for concurrency control in distributed database. In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, 1982.

[13] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McMraw-Hill Book Company, 1984.

[14] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. ACM*, 19(11):624–633, 1976.

[15] A. A. Farrag and M. Tamer Ozsu. A general concurrency control for database systems. In *AFIPS Proc. of the National Computer Conf.*, pages 567–573, July 1985.

[16] A. A. Farrag and M. Tamer Ozsu. Towards a general concurrency control algorithm for database systems. *IEEE Trans. on Software Engineering*, SE-13(10):1073–1078, 1987.

[17] J.N. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer-Verlag, 1978.

[18] J.N. Gray, R.A. Lorie, G.R. Putzulo, and I.L Traiger. Granularity of locks and degrees of consistency in a shared database. Research Report RJ1654, IBM, 1975.

[19] T. Hadzilacos and V. Hadzilacos. Transaction synchronisation in object bases. In *Proc. of the 7th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 193–200, 1988.

[20] Theo Haerder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[21] C.A.R. Hoare. Monitors: an operating system structuring concept. *Comm. ACM*, 17(10):549–557, 1974.

[22] H.T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2):213–226, 1981.

[23] G. Lausen. Concurrency control in database systems: A step towards the integration of optimistic methods and locking. In *Proc. ACM Computer Science Conference '82*, pages 64–68, 1982.

[24] D. A. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[25] R.J.T. Morris and W.S. Wong. Performance analysis of locking and optimistic concurrency control algorithms. *Performance Evaluation*, 5(2):105–118, 1985.

[26] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, 1985.

[27] U. Pradel, G. Schlageter, and R. Unland. Redesign of optimistic methods: Improving performance and applicability. In *Proc. of the 2nd Inte'l Conf. on Data Engineering*, pages 466–473, Feb. 1986.

[28] T. C. Rakow, J. Gu, and E. J. Neuhold. Serializability in object-oriented database systems. In *Proc. of 6th Data Engineering*, pages 112–120, February 1990.

[29] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract data types. *ACM Trans. on Computer Systems*, 2:223–250, 1984.

[30] R.M. Shapiro and R.E. Millstein. NSW reliability plan. Technical Report 7701-1411, Computer Associates, June 1977.

[31] R.M. Shapiro and R.E. Millstein. Reliability and fault recovery in distributed processing. In *Oceans'77 Conference Record*, volume 2, 1977.

[32] R.E. Stearns, P.M. Lewis, and D.J. Rosenkrantz. Concurrency controls for database systems. In *Proc. of 17th Symp. on Foundations of Computer Science*, pages 19–32, 1976.

[33] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.

[34] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. on Computers*, 37(12):1488–1505, dec 1988.

[35] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. on Programming Languages and Systems*, 11(2):249–283, 1989.