An Introduction to Database Systems

2nd Edition

Bipin C. DESAI

Concordia University
Montreal

BytePress

Limit of Liability/Disclaimer of Warranty:

The authors and the publishers have taken care to prepare this book. However, there is no warranty of the accuracy, completeness or presentation of the latest version/generation of any system discussed in this book. The reader must be aware of the fact that software systems often have multiple bugs and are not well thought out, and are usually suitable for limited situations and/or data combinations. Hence the user must be responsible for the appropriate application of any technique and use of any software or code examples.

Furthermore, there is no assurance whatsoever of the possible usefulness or commercialization of any programs, scripts and examples given in this book.

Any references given are based on their existence at the time of writing and the authors and the publishers do not endorse them or imply any usefulness of the information found therein. The reader must be aware that any web site cited may change, disappear or change their terms of service.

Published by: Electronic Publishing BytePress.com Inc.
Hardcopy - ISBN: 978-1-988392-15-8
Electronic - ISBN: 978-1-988392-08-0

10 Concurrency Management

Concurrent execution of a number of transactions implies that the operations from these transactions may be interleaved. This is not the same as serial execution of the transactions where each transaction is run to completion before the next transaction is started. Concurrent access to a database by a number of transactions requires some type of concurrency control to preserve the consistency of the database, to ensure that the modifications made by the transactions are not lost, and to guard against transaction reading data that is inconsistent. The serializability criterion is used to test whether an interleaved execution of the operations from a number of concurrent transactions is correct or not. The serializability test consists of generating a precedence graph from a interleaved execution schedule. If the precedence graph is acyclic, then the schedule is serializable, which means that the database will have the same state at the end of the schedule as some serial execution of the transactions. In this chapter, we introduce a number of concurrency control schemes.

10.1 Introduction

Larger computer systems are typically used by many users in a multi-programming mode[1]. This is a mode of operation wherein these users' programs are executed concurrently. One reason for the use of multi-programming is to exploit the different characteristics of the various programs to maximize the utilization of the equipment; thus, while one program awaits the completion of an input/output operation, the processor can be used to do the computation of another program. Another factor in the choice of multi-programming is the need to share a resource by these different programs: a database is such a shared resource. The primary objective of the database system (at least on a large main frame) is to allow many users and application programs to access data from the database in a concurrent manner.

One such shared database that is used in an on-line manner is the database for an airline reservation system which is used by many agents accessing the database from their terminals. Other such application is banking allowing customers to access and make transactions on their accounts using browser or mobile applications/

A database could also be accessed in a batch mode, exclusively, or concurrently, with the on-line access. Thus, the database for the airline reservation system, in addition to providing on-line access, could also be used by batch application programs which gather statistics and perform accounting operations.

The sharing of the database for read-only access does not cause any problem, but if one of the transactions running concurrently tries to modify some data-item, then it could lead to inconsistencies. Furthermore, if more than one transaction is allowed to simultaneously modify a data-item in the database, then it could lead to incorrect values for the data-item and an inconsistent database. Such would be the result even if each of the transactions were correct and a consistent database would remain so if each of these transactions were run one at a time. For example, suppose that two ticket agents access the

1 Multi-programming which enabled time-sharing in the late 1960s has been replaced by cloud computing in the 2010s with the loss of control on the actual systems running the software and hosting the data!

airline reservation system simultaneously to see if a seat is available on a given flight; if both agents make a reservation against the last available seat on that flight, overbooking of the flight would result.

This overbooking, may cause some problems, however due to possible cancellations, it may also work itself out of the overbooking situation at the time of the actual flight. However, in other cases, the simultaneous access of data in the modify mode could result in unacceptable results. This potential problem of leaving the database in an inconsistent state with concurrent usage requires that some kind of mutual exclusion be enforced, such that the concurrently running transactions be able to access only disjoint data for modifications.

We defined the concept of a transaction in the previous chapter on query processing as being a set of actions on the database which can be considered atomic from the point of view of the user. One method of enforcing mutual exclusion is by some type of locking mechanism which locks a shared resource (for example a data-item), used by a transaction for the duration of its usage by the transaction. The locking of a data-item by a transaction implies that the locked data-item can only be used by the transaction which locked it. The other concurrent transactions are locked out and have to wait their turn at using the data-item. However, a locking scheme must be fair. This requires that the lock manager, which is the DBMS subsystem managing the locks, must not cause some concurrent transaction to be permanently blocked from using the shared resource: this is referred to as avoiding the **starvation** or **livelock** situation. The other danger to be avoided is that of **deadlock**, wherein a number of transactions are waiting in a circular chain, each waiting for the release of resources held by the next transaction in the chain.

In other methods of concurrency control, some form of *a priori* ordering with a single or many versions of data is used. These methods are called the time-stamp ordering and the multi-version schemes. The optimistic approach, on the other hand, assumes that the data-items used by concurrent transactions are most likely be disjoint.

Concurrency and Possible Problems

We defined the concept of a transaction in the Chapter 9 where, it was stressed that a correct transaction when completed, leaves the database in a consistent state provided that the database was in a consistent state at the start of the transaction. Nevertheless, during the life of a transaction, the database could be inconsistent, although, if the inconsistencies are not accessible to other transactions, they would not cause a problem.

In the case of concurrent operations, where a number of transactions are running and using the database, we cannot make any assumptions about the order in which the statements belonging to different transactions will be executed. The order in which these statements are executed is called a **schedule**. Let us consider the two transactions given in Figure 10.1. Each transaction reads some data-item, performs some operations on the data-item which could change its value and then writes out the modified data-item.

In Figure 10.1 and in subsequent examples in this chapter, we assume that the **Read** operation, reads in the database value of the named variable to a local variable with an identical name. Any modifications by a transaction are made on this local copy. The modifications made by the transactions are indicated by the operators f_1 and f_2 in Figure 10.1. These modifications are not reflected in the database until the **Write** operation is executed, at which point the modifications in the value of the named

variable are said to be committed. In effect the **Write** operation is a signal for committing the modifications and reflecting the changes onto the physical database.

```
Transaction T1                      Transaction T2

Read(Avg_Faculty_Salary)            Read(Avg_Staff_Salary)
 Avg_Faculty_Salary :=          Avg_Faculty_Salary :=
    f1(Avg_Faculty_Salary)              f2(Avg_Faculty_Salary)
Write(Avg_Faculty_Salary)           Write(Avg_Faculty_Salary)
```

Figure 10.1 *Two Concurrent Transactions*

Since the transactions of Figure 10.1 are accessing and modifying distinct data-items, (*Avg_Faculty_Salary*, *Avg_Staff_Salary*) there is no problem in executing these transactions concurrently. In other words, regardless of the order of interleaving of the statements of these transactions, we will get a consistent database on the termination of these transactions. Figure 10.2 gives two possible schedules for executing the transactions of Figure 10.1, in an interleaved manner.

```
Schedule 1                          Schedule 2

  | Read(Avg_Faculty_Salary)          | Read(Avg_Staff_Salary)
  |                                   |
  | Avg_Faculty_Salary :=             | Avg_Staff_Salary :=
T |   f1(Avg_Faculty_Salary)        T |   f2(Avg_Staff_Salary)
i |                                 i |
m | Write(Avg_Faculty_Salary)       m | Read(Avg_Faculty_Salary)
e |                                 e |
  | Read(Avg_Staff_Salary)            | Avg_Faculty_Salary :=
  |                                   |  f1(Avg_Faculty_Salary)
  | Avg_Staff_Salary :=               |
  |   f2(Avg_Staff_Salary)            | Write(Avg_Faculty_Salary)
  |                                   |
  ▼ Write(Avg_Staff_Salary)           ▼ Write(Avg_Staff_Salary)
```

Figure 10.2 *Possible Interleaving of Concurrent Transactions of Figure 10.1*

10.1.1 Lost Update Problem

Let us consider the transactions of Figure 10.3. These transactions are accessing the same data-item *A*. Each of the transactions modifies the data-item and writes it back.

Transaction T_3	Transaction T_4
Read(*A*)	**Read**(*A*)
A := *A* +10	*A* := *A* * 1.1
Write(*A*)	**Write**(*A*)

Figure 10.3 *Two Transactions Modifying the Same Data-item*

Let us consider a number of possible inter-leavings of the execution of the statements of these transactions. These schedules are given in Figure 10.4 (a) and (b).

Schedule 1	Transaction T_3	Transaction T_4	Value of *A*
	Read(*A*)	**Read**(*A*)	200
T	*A* := *A**1.1	*A* := *A**1.1	
i	**Read**(*A*)	**Read**(*A*)	
m	*A* := *A*+10	*A* := *A* +10	
e	**Write**(*A*)	**Write**(*A*)	210
▼	**Write**(*A*)	**Write**(*A*)	220

(a)

Schedule 2	Transaction T_3	Transaction T_4	Value of *A*
	Read(*A*)	**Read**(*A*)	200
T	*A* := *A* +10	*A* := *A* +10	
i	**Read**(*A*)	**Read**(*A*)	
m	*A* := *A* * 1.1	*A* := *A* * 1.1	
e	**Write**(*A*)	**Write**(*A*)	220
▼	**Write**(*A*)	**Write**(*A*)	210

(b)

Figure 10.4 *Two Schedules for Transactions of Figure 10.3*

Starting with 200 as the initial value of *A*, let us see what the value of *A* would have been if the transactions are run without any interleaving. In other words, the transactions are run to completion, effectively without any interruptions, one at a time in a serial manner. If transaction T_3 is run first, then at the end of the transaction, the value of *A* will have changed from 200 to 210. Running transaction T_4 after the completion of T_3 will change the value of *A* from 210 to 231. Running the transactions in the order, T_4 followed by T_3, result in a final value for *A* of 230. The result obtained with neither of these two interleaved executions schedules of Figure 10.4, agrees with either of the results of executing these same transactions serially. Obviously something is wrong!

In the case of each of the schedules given in Figure 10.4, we have lost the update made by one of the transactions. In schedule 1, the update made by transaction T_3 is lost; in schedule 2, the update made by transaction T_4 is lost. Each of the schedules exhibits an example of the so-called **Lost Update** problem of the concurrent execution of a number of transactions.

It is obvious that the reason for the lost update problem is the following: even though we have been able to enforce that the changes made by one concurrent transaction not be seen by the other transactions until it commits, we have not enforced the atomicity requirement. This demands that only one transaction can modify a given data-item at a given time and other transactions should be locked out from even viewing the unmodified value (in the database) until the modifications (being made to a local copy of the data) are committed to the database.

10.1.2 The Inconsistent Read Problem

The lost update problem was caused by concurrent modifications of the same data-item. However, concurrency can also cause problems when only one transaction modifies a given set of data while that set of data is being used by other transactions.

Consider the transactions of Figure 10.5. Suppose *A* and *B* represent some data-items containing integer valued data, for example, two accounts in a bank (or quantity of some part X in two different locations etc.). Let us assume that Transaction T_5 transfers 100 units from *A* to *B*. Transaction T_6 is concurrently running and it wants to find the total of the current values of data-items *A* and *B*, (sum of balance in case *A* and *B* represent two bank accounts, or total quantity of part X in the two different locations etc.).

Transaction T_5	Transaction T_6
Read (*A*)	*Sum* := 0
A := *A* -100	**Read**(*A*)
Write (*A*)	*Sum* := *Sum* + *A*
Read(*B*)	**Read**(*B*)
B := *B* + 100	*Sum* := *Sum* + *B*
Write(*B*)	**Write**(Sum)

Figure 10.5 *Two Transactions, one modifies while the other reads*

Figure 10.6 gives a possible schedule for the concurrent execution of the transactions of Figure 10.5 with the initial value of *A* and *B* being 500 and 1000, respectively. We notice from the schedule that transaction T_6 uses the value of *A* before the transfer was made, but it uses the modified value of *B* after the transfer. The result of this is that transaction T_6 erroneously determines the total of *A* and *B* as being 1600 instead of 1500. We can also come up with another schedule of the concurrent execution of these transactions which will give a total of *A* and *B* as being 1400, and of course, other schedules where T5 and T_6 run serially would will give the correct answer.

	Schedule	Transaction T_5	Transaction T_6	Value of Database items		
				A	*B*	*Sum*
	Read(*A*)	**Read** (*A*)		500	1000	-
	Sum := 0		*Sum* := 0			0
T	**Read** (*A*)		**Read** (*A*)			
i	*A*:=*A*-100	*A* := *A* -100				
m	**Write** (*A*)	**Write** (*A*)		400		
e	*Sum*:=*Sum*+*A*		*Sum*:=*Sum*+*A*			500
	Read(*B*)	**Read**(*B*)				
	B:=*B*+100	*B* := *B* + 100				
	Write(*B*)	**Write**(*B*)			1100	
	Read(*B*)		**Read**(*B*)			
	Sum:=*Sum*+*B*		*Sum* := *Sum* + *B*			
▼	**Write**(*Sum*)					1600

Figure 10.6 *Example of Inconsistent Reads*

The reason we got an incorrect answer in the schedule of Figure 10.6 was due to the fact that transaction T_6 was using values of data-items *A* and *B* while they were being modified by transaction T_5.

Locking out transaction T_6 from these data-items individually would not have solved the problem of this inconsistent read. The problem would have been resolved in this example, only if transaction T_5 had not released the exclusive usage of the data-item A after locking data-item *B*. We will discuss this scheme, called two phase locking, in section 10.4.

10.1.3 The Phantom Phenomenon

The previous examples were deliberately made simple to illustrate the points of the lost update and the inconsistent read problems. In order to illustrate the phantom phenomenon let us consider an organization where parts are purchased and kept in stock. The parts are withdrawn from stock and used by a number of projects. In order to check the extent of loss, for example due to pilferage, we want to see if the current quantity of some part purchased and received is equal to the current sum of the quantity of that part in stock, plus the current quantities in use by various projects. Let us assume that we have record types (relations in the case of a relational database system) called INVENTORY, RECEIVED and USAGE. The fields of these records are as shown below. The record type INVENTORY keeps track of the quantity of a given part in stock at a given point in time. The record type RECEIVED contains, for a given part, the total units of that part that has been received to date. The record USAGE keeps track of the project for which a given part was used.

INVENTORY(*Part#*, *Quantity_in_Stock*)
RECEIVED(*Part#*, *Quantity_Received_to_Date*)
USAGE(*Project_No*, *Part#*, *Quantity_Used_to_Date*)

Consider transaction T_7 that will perform this auditing operation. It will, for example, proceed by locking each item in an exclusive mode before each step, as follows:

1. Lock the records of INVENTORY and for *Part#* = $Part_1$ find the *Quantity_in_Stock*.
2. Lock all existing records of USAGE and add the *Quantity_Used_to_Date* for $Part_1$ in any project that uses this part to *Quantity_in_Stock* found in step 1.
3. Lock the RECEIVED records and compare the value of the sum found with *Received_to_Date* value for $Part_1$.
4. Release all locks.

Problems will be encountered if there is another transaction, T_8, which is run to reflect the receipt of additional quantities of $Part_1$. Transaction T_8 adds this quantity to the record corresponding to $Part_1$ of the record type RECEIVED, and assigns these parts directly to be used in a new project for which a new record of the record type USAGE is created. If transaction T_8 is scheduled to run between steps 2 and 3 above, then transaction T_7 will come up with an incorrect result (T_7 will show the loss in $Part_1$).

Here we see that the locking of records did not prevent the creation of a new record, which was created after the existing records had been locked. This new record for USAGE created by transaction T_8 is a *phantom* as far as transaction T_7 is concerned. It did not exist when transaction T_7 locked the records of USAGE.

However, the problem could be prevented if the locking of a set of records also prevents the addition of such phantom records. Effectively, the locking of a record belonging to a record type must guarantee that no new record occurrences of the record type can be added until the lock is released. The other

necessary precaution for the schedule above, is to lock the record RECEIVED before releasing the lock on USAGE.

10.1.4 Semantics of Concurrent Transactions

In the case of concurrent operations, where a number of transactions are running and modifying parts of the database, we have to not only hide the changes made by a transaction from other transactions, but we also have to make sure that only one transaction has exclusive access to these data-items for at least the duration of the original transaction's usage of the data-items. This requires that an appropriate locking mechanism be used to allow exclusive access of these data-items to the transaction requiring them. In the case of the transactions of Figure 10.3, no such locking was used with the consequence that the result is not the same as the result one would have obtained had these transactions run consequently.

Now let us explain why the results obtained when we run two transactions, one after the other, need not be the same for different orderings. The modification operations performed by two transactions are not necessarily commutative. The operations $A := (A + 10) + 20$ give the same result as $A := (A + 20) + 10$ for the same initial value for A (which is assumed to be an integer valued data-item): this is so because the addition operation is commutative. Similarly, $(A * 10)* 20 = (A * 20)* 10$.

However, commuting the order of operations as illustrated by the following expressions does not always give the same result:

- *Salary* := (*Salary* + 1000) * 1.1
- *Salary* := (*Salary* * 1.1) + 1000.

In the above example, we have two transformations: in the first the *Salary* is initially modified by adding 1000 to it and then the result is augmented by 10% to give the revised *Salary*; in the second, the *Salary* is first augmented by 10% and then 1000 is added to the result which now becomes the revised *Salary*. The reasonable approach, to make sure that the intended result is obtained in all cases (i.e., to make sure that transaction T_i is completed before transaction T_j is run), would be to code the operations in a single transaction and not to divide the operations into two or more transactions. Thus, if the above set of operations on *Salary* were written as two transactions as given below, we cannot be sure as to which of the above two results would be obtained with their concurrent execution.

```
Transaction Ti                  Transaction Tj
Read Salary                     Read Salary
Salary := Salary*1.1            Salary := Salary + 1000
Write Salary                    Write Salary
```

In effect, the division of a transaction into interdependent transactions, run serially in the wrong order, would give erroneous results. Furthermore, these interdependent transactions must not be run concurrently, otherwise the concurrent execution will lead to results which could be incorrect again, and not agree with the result obtained by any serial execution of the same transactions. It is a logical error to divide a single set of operations into two or more transactions. We will assume hereafter that transactions are semantically correct.

10.2 Serializability

Let us reconsider the transactions of Figure 10.3. We assume that these transactions are independent. An execution schedule of these transactions as shown in Figure 10.7 is called a **Serial Execution**. In a serial execution, each transaction runs to completion before any statements from any other transaction is executed. In Schedule A given in Figure 10.7(a), the transaction T_3 is run to completion before transaction T_4 is executed. In Schedule *B*, transaction T_4 is run to completion before transaction T_3 is started. If the initial value of A in the database was 200, the schedule *A* would result in the value of A being changed to 231. Similarly, the schedule *B* with the same initial value of *A* would give a result of 230.

This may seem odd, but in a shared environment, the result obtained by independent transactions which modify the same data-item, always depends on the order in which these transactions are run; and any of these results is considered to be correct.

If there are two transactions and if they refer to and use distinct data-items, then the result obtained by the interleaved execution of the statements of these transactions would be the same regardless of the order in which these statements are executed (provided there are no other concurrent transactions which refer to any of these data-items). In this chapter, we assume that the concurrent transactions share some data-items and, hence, we are interested in a correct ordering of execution of the statements of these transactions.

```
    Schedule A          Transaction T3          Transaction T4

  |  Read(A)            Read(A)
T |  A := A +10         A := A +10
i |  Write(A)           Write(A)
i |  Read(A)                                   Read(A)
e |  A := A * 1.1                             A := A * 1.1
  ▼  Write(A)                                 Write(A)
```

(a)

```
    Schedule B          Transaction T3          Transaction T4

  |  Read(A)                                   Read(A)
T |  A := A * 1.1                             A := A * 1.1
i |  Write(A)                                  Write(A)
i |  Read(A)            Read(A)
e |  A := A +10         A := A +10
  ▼  Write(A)           Write(A)
```

(b)

Figure 10.7 *Two Serial Schedules*

A non-serial schedule wherein the operations from a set of concurrent transactions are interleaved is considered to be serializable if the execution of the operations in the schedule leaves the database in the same state as ***some*** serial execution of these transactions. With two transactions, we can have at most 2

distinct serial schedules, and starting with the same state of the database, each of these serial schedules, could give a different final state of the database. Starting with an initial value of 200 for *A*, the serial schedule illustrated in Figure 10.7(a) would give the final value of *A* as 231, and for the serial schedule illustrated in Figure 10.7(b) the final value of *A* would be 230. If we have n concurrent transactions, then it is possible to have n! (where, n! = n*(n-1)*(n-2)* ..*3*2*1) distinct serial schedules, and possibly that many distinct resulting modifications to the database. For a serializable schedule, all we require is that the schedule gives a result which is the same ***as any one*** of these possibly distinct results.

When n transactions are run concurrently, and in an interleaved manner, the number of possible schedules is much larger than n!. We would like to find out if a given interleaved schedule produces the same result as one of the serial schedules. If the answer is positive, then the given interleaved schedule is said to be serializable.

Definition: **Serializable schedule**

Given an interleaved execution of a set of n transactions; the following conditions hold for each transaction in the set:

- all transactions are correct in the sense that if any one of the transactions is executed by itself, on a consistent database, the resulting database will be consistent.

- any serial execution of the transactions is also correct and preserves the consistency of the database: the results obtained are correct. (This implies that the transactions are logically correct and that no two transactions are interdependent).

Then, the given interleaved execution of these transactions is said to be serializable if it produces the same result as some serial execution of the transactions.

Since a serializable schedule gives the same result as some serial schedule, and since that serial schedule is correct, then the serializable schedule is also correct. Thus, given any schedule, we can say it is correct if we can show that it is serializable.

Algorithm 10.1 given in section 10.2.2 establishes the serializability of an arbitrarily interleaved execution of a set of transactions on a database. The algorithm does not consider the nature of the computations performed by a transaction nor the exact effect of each such computational operations on the database. In effect, the algorithm ignores the semantics of the operations performed by the transactions including the commuting property of algebraic or logical computations of the transactions. We may conclude from the algorithm that a given schedule is not serializable, when in effect it is, if some of the semantics and the algebraic commutability were not ignored. However, the algorithm will never lead us to conclude that a schedule is serializable, when in effect, it does not produce the same result as some serial schedule. The computation involved in analyzing each transaction and seeing if its operations could be safely interleaved with those of other concurrent transactions is not justified by the greater degree of concurrency of the resulting 'better' serializable schedule.

In algorithm 10.1, we make the following assumptions:

- Each transaction is a modifying transaction, i.e., it would change the value of at least one database item

- For each such item *A* that a transaction modifies, it would first read the value a of the item from the database (this is the so called *read before write protocol*)
- Having read the value it would transform a to *f*(a), where *f* is some transaction dependent computation or transformation.
- It would then write this new value to the database.

Before presenting the algorithm we will present the notion of a precedence graph in the following section.

10.2.1 Precedence Graph

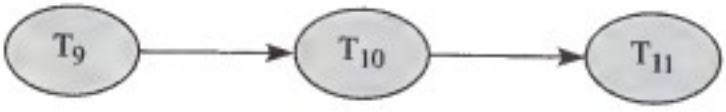
Precedence graph G (V, E) consists of a set of nodes or vertices V and a set of directed arcs or edges E. Figure 10.8 gives an example of a schedule and the corresponding precedence graph. The schedule is for three transactions T_9, T_{10} and T_{11} and the corresponding precedence graph has the vertices T_9, T_{10} and T_{11}. There is an edge from T_9 to T_{10} and another edge from T_{10} to T_{11}. If T_9, T_{10} and T_{11} represent three transactions, then the precedence graph represents the serial execution of these transactions.

In a precedence graph, a directed edge from a node T_i to a node T_j indicates one of the following conditions regarding the read and write operations in transactions T_i and T_j with respect to some database item *A*:

- T_i performs the operation **Read**(*A*) before T_j performs the operation **Write**(*A*);
- T_i performs the operation **Write**(*A*) before T_j performs the operation **Read**(*A*).

	Schedule	Transaction T_9	Transaction T_{10}	Transaction T_{11}
	Read(*A*)	**Read**(*A*)		
	$A:=f_1(A)$	$A:=f_1(A)$		
T	**Write**(*A*)	**Write**(*A*)		
i	**Read**(*A*)		**Read**(*A*)	
m	$A:=f_2(A)$		$A:=f_2(A)$	
e	**Write**(*A*)		**Write**(*A*)	
	Read(*B*)		**Read**(*B*)	
	$A:=f_3(B)$		$B:=f_3(B)$	
	Write(*B*)		**Write**(*B*)	
	Read(*B*)			**Read**(*B*)
	$A:=f_4(B)$			$B:=f_4(B)$
▼	**Write**(*B*)			**Write**(*B*)

(a)



(b)

Figure 10.8 *A Schedule and an Acyclic Precedence Graph*

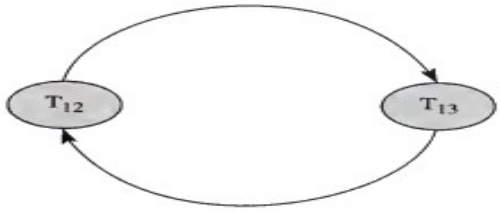
If we limit ourselves to the read-before-write protocol only, then we have to look for an edge corresponding to the second condition only. This condition involves writing of a data-item by transaction T_i prior to its being read by T_j.

In Figure 10.8(a), all the statements in transaction T_9 are executed before transaction T_{10} is started. Similarly, all the operations of T_{10} are completed before starting T_{11}. The precedence graph corresponding to the schedule of Figure 10.8(a) is given in Figure 10.8(b).

Figure 10.9(a) gives a schedule and Figure 10.9(b) gives the precedence graph for transactions T_{12} and T_{13}.

	Schedule	Transaction T_{12}	Transaction T_{13}
	Read(A)	**Read**(A)	
	$A:=f_1(A)$	$A:=f_1(A)$	
T	**Write**(A)	**Write**(A)	
i	**Read**(B)		**Read**(B)
m	$A:=f_2(B)$		$B:=f_2(B)$
e	**Write**(B)		**Write**(B)
	Read(B)	**Read**(B)	
	$A:=f_3(B)$	$B:=f_3(B)$	
	Write(B)	**Write**(B)	
	Read(A)		**Read**(A)
	$A:=f_4(A)$		$A:=f_4(A)$
▼	**Write**(A)		**Write**(A)

(a)



(b)

Figure 10.9 A *Schedule and a Cyclic Precedence Graph*

In the precedence graph of Figure 10.9(b), there is an edge from T_{12} to T_{13}, as well as an edge from T_{13} to T_{12}. The edge T_{12} to T_{13} is included because T_{12} executes a **Read** operation before T_{13} executes a **Write** operation for the database item *A*. The edge T_{13} to T_{12} is included because T_{13} executes a **Write** operation before T_{12} executes a **Read** operation for the database item *B*. We see that the precedence graph of Figure 10.9(b) has a cycle, since we can start from one of the nodes of the graph and, following the directed edges, return to the starting node.

A precedence graph is said to be acyclic if there are no cycles in the graph. The graph of Figure 10.8(b) has no cycles and, hence, it is acyclic. The graph of Figure 10.9(b) is cyclic, since it has a cycle.

The precedence graph for serializable schedule S must be acyclic, and, hence, it can be converted to a serial schedule. To test for the serializability of the arbitrary schedule S for transactions T_1,..., T_k we convert the schedule into a precedence graph, and then test the precedence graph for cycles. If no cycles are detected, then the schedule is serializable, otherwise it is not. If there are n nodes in the graph for schedule S, then the number of operations required to check if there is a cycle in the graph is proportional to n^2.

10.2.2 Serializability Algorithm: Read-before-Write Protocol

In the **read-before-write protocol** we assume that a transaction will read the data-item before it modifies it and after modifications, the modified value is written back to the database. We assume that the read operation effectively locks a data-item; this means that only one transaction at a time can be modifying it. Other concurrent transactions are, thus, locked out from accessing the data-item, at least until the write operation has been executed.

In algorithm 10.1, we give the method of testing if a schedule is serializable. Herein, we create a precedence graph and test for a cycle in the graph. If we find a cycle, then the schedule is non-serializable otherwise we find a linear ordering of the transactions.

Algorithm 10.1:
Title: Test if a Schedule is Serializable under the Read before Write Protocol
Input: Schedule S for the concurrent execution of transactions T_1, .. , T_k.
Output: A serial schedule for S if one exists.
Body:
Step 1: Create precedence graph G as follows: the transactions T_1, .. , T_k are the nodes and each edge of the graph is inserted as follows:
-For a database item X used in the schedule find an operation **Write**(X) for some transaction T_i; if there is a subsequent (earliest) operation **Read**(X) in transaction T_j, then insert an edge from T_i to T_j in the precedence graph, since T_i must be executed before T_j..
Step 2: If the graph G has a cycle (see exercise 10.6), then the schedule S is non-serializable.
If G is acyclic, then find, using topological sort, a linear ordering of the transactions so that if there is an arc from T_i to T_j in G, then T_i precedes T_j. Find a serial schedule as follows:
(a) Initialize the serial schedule as empty
(b) Find a transaction T_i, such that there are no arcs entering T_i. T_i is the next transaction in the serial schedule.
(c) Remove T_i and all edges emitting from T_i: if the remaining set is non-empty, return to (b) otherwise the serial schedule is complete.

In Examples 10.1 and 10.2, we illustrate the application of this algorithm.

Example 10.1: Consider the schedule of Figure A:

	Schedule	Transaction T_{14}	Transaction T_{15}	Transaction T_{16}
	Read (*A*)	**Read** (*A*)		
	Read (*B*)		**Read** (*B*)	
	$A:=f_1(A)$	$A:=f_1(A)$		
	Read (*C*)			**Read** (*C*)
	$B:=f_2(B)$		$B:=f_2(B)$	
T	**Write**(*B*)		**Write**(*B*)	
i	$C:=f_3(C)$			$C:=f_3(C)$
m	**Write**(*C*)			**Write**(*C*)
e	**Write**(*A*)	**Write**(*A*)		
	Read (*B*)			**Read** (*B*)
	Read (*A*)		**Read** (*A*)	
	$A:=f_4(A)$		$A:=f_4(A)$	
	Read (*C*)	**Read** (*C*)		
	Write(*A*)		**Write**(*A*)	
	$C:=f_5(C)$	$C:=f_5(C)$		
	Write(*C*)	**Write**(*C*)		
	$B:=f_6(B)$			$B:=f_6(B)$
▼	**Write**(*B*)			**Write**(*B*)

Figure A *An Execution Schedule Involving Three Transactions*

The precedence graph for the schedule of ***Figure A*** is given in ***Figure B***. The graph has three nodes corresponding to the three transactions T_{14}, T_{15}, and T_{16}. There is an arc from T_{14} to T_{15} because T_{14} writes data-item *A* before T_{15} reads it. Similarly, there is an arc from T_{15} to T_{16} because T_{15} writes data-item *B* before T_{16} reads it. Finally, there is an arc from T_{16} to T_{14} because T_{16} writes data-item *C* before T_{14} reads it. The precedence graph of Figure B has a cycle formed by the directed edges from T_{14} to T_{15}, from T_{15} to T_{16} and from T_{16} back to T_{14}. Hence, the schedule of Figure A is not serializable. We cannot execute the three transactions serially to get the same result as the given schedule.

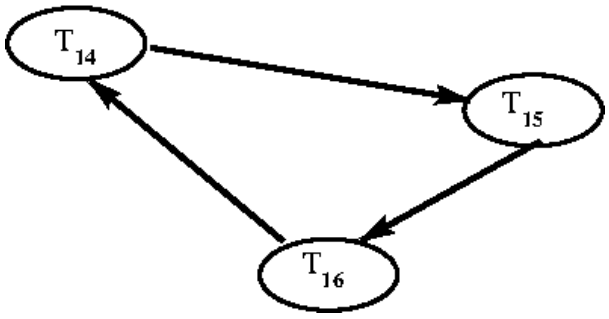


Figure B *A precedence graph with a cycle*

Example 10.2 Consider the schedule given in ***Figure C***:

The execution schedule of ***Figure C*** is serializable since the precedence graph for this schedule given in ***Figure D*** does not contain any cycles. The serial schedule is T_{17} followed by T_{18}, followed by T_{19}.

	Schedule	Transaction T_{17}	Transaction T_{18}	Transaction T_{19}
	Read (A)	**Read** (A)		
	$A:=f_1(A)$	$A:=f_1(A)$		
	Read (C)	**Read** (C)		
	Write(A)	**Write**(A)		
	$A:=f_2(C)$	$A:=f_2(C)$		
	Read (B)		**Read** (B)	
T	**Write**(C)	**Write**(C)		
i	**Read** (A)		**Read** (A)	
m	**Read** (C)			**Read** (C)
e	$B:=f_3(B)$		$B:=f_3(B)$	
	Write(B)		**Write**(B)	
	$C:=f_4(C)$			$C:=f_4(C)$
	Read (B)			**Read** (B)
	Write(C)			**Write**(C)
	$A:=f_5(A)$		$A:=f_5(A)$	
	Write(A)		**Write**(A)	
	$B:=f_6(B)$			$B:=f_6(B)$
▼	**Write**(B)			**Write**(B)

Figure C An Execution Schedule Involving Three Transactions

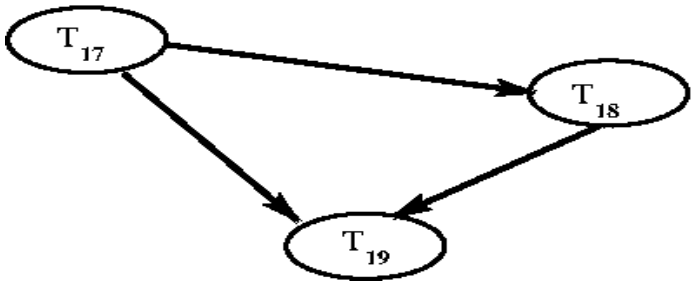


Figure D *Precedence Graph for Schedule of* ***Figure C***

10.2.2 Serializability Algorithm: Read Only, Read before Write, and Write Only Protocol

Algorithm 10.1 is for a set of transactions which follows the read-before-write protocol. However, if there are transactions which in addition to having a set of data-items which are read before re-written, that

have another set of data-items which are only read, and a further set of data-items which are only written, then some additional edges must be added to the graph. We will not treat this generalization in this text; the interested reader can refer to the bibliographic notes for further reading.

10.3 Concurrency Control

If all schedules in a concurrent environment are restricted to serializable schedules, then the result obtained will be consistent with some serial execution of the transactions, and, hence, will be considered as correct. However, using only serial schedules unnecessarily limits the degree of concurrency. Furthermore, testing for serializability of a schedule is not only computationally expensive but it is an after-the-fact technique, not practically applicable. Thus, one of the following **concurrency control** schemes is applied in a concurrent database environment to ensure that the schedules produced by concurrent transactions are serializable. The schemes we will discuss are:

Locking
Timestamp Based Order
Optimistic Scheduling
Multi-version Technique

The intent of locking is to ensure serializability by ensuring mutual exclusion in accessing data-items. In the timestamp based ordering scheme, the order of execution of the transactions is selected a priori by assigning each transaction an unique value. This value is usually based on the system clock, and is called a timestamp. The values of the timestamp of the transactions determine the sequence in which transactions contesting for a given data-item will be executed. Conflicts in the timestamp based scheme are resolved by abort and rollback. In the optimistic scheme it is recognized that the conflict between transactions, though possible, is in reality very rare, and it, thus, avoids all forms of locking. The price paid in the optimistic scheme is verifying the validity of the assumptions, that data used by a transaction has indeed not changed; and the abort and restart of a transaction for which it is ascertained that the data-items being used by it have changed between the time of reading of these values and writing of the values derived from the values read. In the multi-version technique, a data-item is never written over; each write operation creates a new version of a data-item. Many versions of a data-item exist and these represent the historical evolution of the data-item. A transaction sees the data-item of its own epoch. Conflicts are resolved by rollback of a transaction which is too late to write out all values from it's epoch. We will examine each of these concurrency control schemes in the following sections. The problem of deadlock, which is possible in some of these schemes and/or their modifications, will be discussed in section 10.8.

10.4 Locking Scheme

From the point of view of locking, a database can be considered as being made up of a set of data-items. A **lock** is a variable associated with each such data-item. Manipulating the value of a lock is called **locking**. The value of a lock variable is used in the locking scheme to control the concurrent access and manipulation of the associated data-item. Locking those items that are being used by a transaction can prevent other concurrently running transactions from using these locked items. The locking is done by a subsystem of the database management system usually called the *lock manager*.

In order to not restrict concurrency unnecessarily, at least two types of locks are defined: a data-item may be locked in a shared mode or it can be locked in an exclusive mode.

Exclusive Lock: The exclusive lock is also called an update or a write lock. The intention of this mode of locking is to provide exclusive use of the data-item to one transaction. If a transaction T locks a data-item Q in an exclusive mode, no other transaction can access Q, not even to read Q, until the lock is released by transaction T.

Shared Lock: The shared lock is also called a read lock. The intention of this mode of locking is to ensure that the data-item does not undergo any modifications while it is locked in this mode, by one or more transactions. Any number of transactions can concurrently lock and access a data-item in the shared mode, but none of these transactions can modify the data-item. A data-item locked in a shared mode cannot be locked in the exclusive mode until the shared lock is released by all transactions holding the lock. A data-item locked in the exclusive mode cannot be locked in the shared mode until the exclusive lock on the data-item is released.

Having defined these modes of locking, the protocol of sharing is as follows: each transaction, before accessing a data-item, requests that the data-item be locked in the appropriate mode. If the data-item is not locked, then the lock request is honoured by the lock manager. If the data-item is already locked, then the request may or may not be granted, depending on the mode of locking requested and the current mode in which the data-item is locked. If the mode of locking requested is shared, and if the data-item is already locked in the shared mode, then the lock request can be granted. If the data-item is locked in an exclusive mode, then the lock request cannot be granted, regardless of the mode of the request. In this case the requesting transaction has to wait till the lock is released.

	Current lock state of data-item: **unlocked**	Current lock state of data-item: **shared**	Current lock state of data-item: **exclusive**
Request for: unlock	-	yes	yes
Request for: shared	yes	yes	no
Request for: exclusive	yes	no	no

Figure 10.10 *Compatibility of Locking*

The compatibility of a lock request for a data-item with respect to its current state of locking is given in Figure 10.10. Here, we are assuming that the request for locking is made by a transaction not already holding a lock on the data-item.

If transaction T_x makes a request to lock a data-item, let us say *A*, in the shared mode, and if *A* is not locked or if it is already locked in the shared mode, then the lock request is granted. This would mean that a subsequent request from another transaction, T_y, to lock the data-item *A* in the exclusive mode would not be granted and the transaction T_y will have to wait until *A* is unlocked. While *A* is locked in the shared

mode, if transaction T_z makes a request to lock it in the shared mode, then this request can be granted. Both T_x and T_z can concurrently use the data-item *A*.

If the transaction T_x makes a request to lock the data-item *A* in the shared mode, and if *A* is locked in the exclusive mode, then the request made by transaction T_x cannot be granted. Similarly, a request by transaction T_z to lock *A* in the exclusive mode, while it is already locked in the exclusive mode, would also result in the request not being granted, and T_z would have to wait until the lock on *A* is released.

From the above we see that any lock request for a data-item can only be granted if it is compatible with the current mode of locking of the data-item. If the request is not compatible, then the requesting transaction would have to wait until such time as the mode becomes compatible.

The releasing of a lock on a data-item changes its lock status. If the data-item was locked in an exclusive mode, the release of lock request by the transaction holding the exclusive lock on the data-item would result in the data-item being unlocked. Any transaction waiting for a release of the exclusive lock would then have a chance of being granted its request for locking the data-item. If more than one transaction is waiting, then it is assumed that the lock manager would use some fair scheduling technique to choose one of these waiting transactions.

If the data-item was locked in a shared mode, the release of lock request by the transaction holding the shared lock on the data-item may not result in the data-item being unlocked. This is so because more than one transaction may be holding a shared lock on the data-item. It is only when the transaction releasing the lock was the only transaction having the shared lock that the data-item would become unlocked. The lock manager may keep a count of the number of transactions holding a shared lock on a data-item. It would increase this value by one when an additional transaction is granted a shared lock and decrease the lock associated count value by one when a transaction holding a shared lock releases the lock. The data-item would then become unlocked when the number of transactions holding a shared lock on it becomes zero. This count could be stored in appropriate data structure along with the data-item but it would be accessible only to the lock manager.

The lock manager must have a priority scheme whereby it decides whether to allow additional transactions to lock a data-item in the share mode in the following situation:

- The data-item is already locked in the share mode
- There is at least one transaction which is waiting to lock the data-item in the exclusive mode.

Allowing a higher priority to share lock requests could result in possible starvation of the transactions which are waiting for an exclusive lock.

Similarly, the lock manager has to deal with a situation where a data-item is locked in an exclusive mode and there are transactions waiting to lock the data-item in (i) the share mode and (ii) the exclusive mode.

In the following discussions we will assume that a transaction makes a request to lock data-item *A* by executing the statement **Locks**(*A*) or **Lockx**(*A*). The former is for requesting a shared lock; the latter, an exclusive lock. A lock is released by simply executing a **Unlock**(*A*) statement. We are assuming that the transactions are correct. In other words a transaction would not request a lock on a data-item for which it already holds a lock, nor would a transaction unlock a data-item if it does not hold a lock for it.

Transaction T_5	Transaction T_6
Lockx (*A*)	Sum := 0
Read (*A*)	**Locks** (*A*)
A := *A* -100	**Read**(*A*)
Write (*A*)	Sum := Sum + *A*
Unlock (*A*)	**Unlock** (*A*)
Lockx (*B*)	**Locks** (*B*)
Read(*B*)	**Read**(*B*)
B := *B* + 100	Sum := Sum + *B*
Write(*B*)	**Write**(Sum)
Unlock (*B*)	**Unlock** (*B*)

Figure 10.11 *Two Transactions of Figure 10.5 with Lock Requests*

	Schedule	Transaction T_5	Transaction T_6
	Sum := 0		*Sum* := 0
	Locks (*A*)		**Locks** (*A*)
	Read(*A*)		**Read**(*A*)
	Sum := *Sum* + *A*		*Sum* := *Sum* + *A*
T	**Unlock** (*A*)		**Unlock** (*A*)
i	**Lockx** (*A*)	**Lockx** (*A*)	
m	**Read** (*A*)	**Read** (*A*)	
e	*A* := *A* -100	*A* := *A* -100	
	Write (*A*)	**Write** (*A*)	
	Unlock (*A*)	**Unlock** (*A*)	
	Lockx (*B*)	**Lockx** (*B*)	
	Read(*B*)	**Read**(*B*)	
	B := *B* + 100	*B* := *B* + 100	
	Write(*B*)	**Write**(*B*)	
	Unlock (*B*)	**Unlock** (*B*)	
	Locks (*B*)		**Locks** (*B*)
	Read(*B*)		**Read**(*B*)
	Sum := *Sum* + *B*		*Sum* := *Sum* + *B*
	Write(*Sum*)		**Write**(*Sum*)
▼	**Unlock** (*B*)		**Unlock** (*B*)

Figure 10.12 *A Possible Schedule Causing Inconsistent Read*

A transaction may have to hold onto the lock on a data-item beyond the point in time when it last needs it to preserve consistency and avoid the inconsistent read problems discussed in section 10.1.2. We illustrate this point by re-working the example of Figure 10.5. Here each transaction request locks for the data-items *A* and *B*: Transaction T_5 in exclusive mode and transaction T_6 in shared mode. The transactions with the lock requests are given in Figure 10.11.As shown there, the transactions attempt to release the locks on the data-items as soon as possible.

Now consider Figure 10.12 which gives a possible schedule of execution of the transactions of Figure 10.11. The locking scheme did not resolve the inconsistent read problem; the reason being that transactions T_5 and T_6 are performing an operation which is made up of many steps and all these

transactions have to be executed in an atomic manner. The database is in an inconsistent state after transaction T_5 has taken 100 units from *A*, but not added it to *B*. Allowing transaction T_6 to read the values of *A* and *B* before transaction T_5 is complete, leads to the inconsistent read problem.

A possible solution to the inconsistent read problem is shown in Figure 10.13. Here, transactions T_5 and T_6 are rewritten as transactions T_{20} and T_{21}. The possible schedules of concurrent executions of these transactions are shown in Figure 10.14 and 10.15. Both of these solutions extend the period of time for which they keep some data-items locked even though the transactions no longer need these items. This extended locking forces a serialization of the two transactions and, hence, gives correct results.

```
Transaction T20                Transaction T21
                               Locks(Sum)
Lockx (A)                      Sum := 0
Read (A)                       Locks (A)
A := A -100                    Read(A)
Write (A)                      Sum := Sum + A
Lockx (B)                      Locks (B)
Unlock (A)                     Read(B)
Read(B)                        Sum := Sum + B
B := B + 100                   Write(Sum)
Write(B)                       Unlock (B)
Unlock (B)                     Unlock (A)
```

Figure 10.13 *Transactions Locking All Items Before Unlocking*

10.4.1 Two Phase Locking

The correctness of the schedules of Figure 10.14 and 10.15 of the transactions in Figure 10.13 lead us to the observation that both these solutions involve transactions, wherein the locking and unlocking operations are monotonic, in the sense that all locks are first acquired before any of the locks are released. Once a lock is released, no additional locks are requested. In other words, the release of the locks is delayed until all locks on all data-items required by the transaction have been acquired.

This method of locking is called **two phase locking**: it has two phases, a **growing phase** wherein the number of locks increase from zero to the maximum for the transaction, and a **contracting phase** wherein the number of locks held decrease from the maximum to zero. Both of these phases are monotonic: the number of locks are only increasing in the first phase, and decreasing in the second phase. Once a transaction starts releasing locks, it is not allowed to request any further locks. In this way a transaction is obliged to request all locks it may need during its life before it releases any. This leads to a possible lower degree of concurrency.

The two phase locking protocol ensures that the schedules involving transactions using this protocol will always be serializable. For instance, if S is a schedule containing the interleaved operations from a number of transactions, T_1, T_2, ... , T_k and all the transactions are using the two phase locking protocol, then schedule S is serializable. This is so because if the schedule is not serializable, then the precedence graph for S will have a cycle made up of a subset of $\{T_1, T_2, \ldots, T_k\}$. Assume the cycle consists of:

$T_a \rightarrow T_b \rightarrow T_c \rightarrow .. T_x \rightarrow T_a$

	Schedule	Transaction T_{20}	Transaction T_{21}
	Sum := 0		*Sum* := 0
	Locks (*A*)		**Locks** (*A*)
	Read(*A*)		**Read**(*A*)
	Sum := *Sum* + *A*		*Sum* := *Sum* + *A*
T	**Locks** (*B*)		**Locks** (*B*)
i	**Read**(*B*)		**Read**(*B*)
m	*Sum* := *Sum* + *B*		*Sum* := *Sum* + *B*
e	**Write**(*Sum*)		**Write**(*Sum*)
	Unlock (*B*)		**Unlock** (*B*)
	Unlock (*A*)		**Unlock** (*A*)
	Lockx (*A*)	**Lockx** (*A*)	
	Read (*A*)	**Read** (*A*)	
	A := *A* -100	*A* := *A* -100	
	Write (*A*)	**Write** (*A*)	
	Lockx (*B*)	**Lockx** (*B*)	
	Unlock (*A*)	**Unlock** (*A*)	
	Read(*B*)	**Read**(*B*)	
	B := *B* + 100	*B* := *B* + 100	
	Write(*B*)	**Write**(*B*)	
▼	**Unlock** (*B*)	**Unlock** (*B*)	

Figure 10.14 *A Possible Solution to the Inconsistent Read Problem*

	Schedule	Transaction T_{20}	Transaction T_{21}
	Lockx (*A*)	**Lockx** (*A*)	
	Read (*A*)	**Read** (*A*)	
	A := *A* -100	*A* := *A* -100	
	Write (*A*)	**Write** (*A*)	
T	**Lockx** (*B*)	**Lockx** (*B*)	
i	**Unlock** (*A*)	**Unlock** (*A*)	
m	**Read**(*B*)	**Read**(*B*)	
e	*B* := *B* + 100	*B* := *B* + 100	
	Write(*B*)	**Write**(*B*)	
	Unlock (*B*)	**Unlock** (*B*)	
	Sum := 0		*Sum* := 0
	Locks (*A*)		**Locks** (*A*)
	Read(*A*)		**Read**(*A*)
	Sum := *Sum* + *A*		*Sum* := *Sum* + *A*
	Locks (*B*)		**Locks** (*B*)
	Read(*B*)		**Read**(*B*)
	Sum := *Sum* + *B*		*Sum* := *Sum* + *B*
	Write(*Sum*)		**Write**(*Sum*)
	Unlock (*B*)		**Unlock** (*B*)
▼	**Unlock** (*A*)		**Unlock** (*A*)

Figure 10.15 *Another Solution to the Inconsistent Read Problem*

This means that a lock operation by T_b is followed by an unlock operation by T_a: a lock operation by T_c is followed by an unlock operation by T_b, and finally a lock operation by T_a is followed by an unlock operation by T_x. However, this is a contradiction of the assertion that T_a is using the two phase protocol. This leads us to the conclusion that the assumption that there was a cycle in the precedence graph is incorrect and, hence, S is serializable.

The transactions of Figure 10.13 use the two phase locking protocol, and the schedules derived from the concurrent execution of these transactions given in Figures 10.14 and 10.15 are serializable. However, the transactions of Figure 10.12 are not following the two phase locking protocol and the schedule of is not serializable.

The observant reader will notice that the danger of deadlock exists in the two phase locking protocol. We examine this problem in greater detail in section 10.8.

10.4.2 Granularity of Locking

So far we have assumed that a data-item can be locked. However, we have not defined explicitly what the data-item is. If the data-item is very large, for instance the entire database, then of course the overhead of locking is very small. The lock manager manages only one item. The drawback here is obvious: the concurrency is very low since only one transaction can run in an exclusive mode at a given point in time, even though it may need a very small portion of the database. On the other hand, if the granularity of the data-item is very small, for example a data-item could be the field of a record, then the degree of concurrency can be fairly high. Although the overhead of locking in this case can be considerable. A transaction which needs many records and fields will have to request many locks, all of which have to be managed by the lock manager. For the highest degree of flexibility, the locking scheme should allow multiple granularity of locking from a data field to entire database.

When the data-item that is locked is, for example, a record type, then to avoid the phantom read problem, locking a record type requires not only that the existing record occurrences be locked, but it also implies that non-existing records are also locked. In this manner it is possible to preclude the insertion of phantom records by other concurrent transactions.

To avoid locking too early and in situations where the transaction itself has to determine which data-items to lock, locks are requested dynamically by the transactions. This creates an additional overhead for the lock manager, which in addition to the locking overhead, has to determine if there is a situation of deadlock. The methods of handling deadlocks is discussed in section 10.8.

10.4.3 Hierarchy of Locks and Intention Mode Locking

Some data structures used in the database are structured in the form of a tree. For example the nodes of a B-tree index are hierarchically structured, and a transaction may need to lock the entire B-tree or only a portion of it, i.e., a proper sub-tree. Similarly, the database may be considered to be a hierarchy consisting of the following nodes:

- entire database
- some designated portion of the database
- a record type (or in the case of the relational database a relation)
- an occurrence of a record (a tuple)

- a field of the record (an attribute)

The nodes of the hierarchy could depend on the data model being used by the DBMS. In the case of the hierarchical model, the hierarchies represent a tree and each node of the tree can be locked. In the case of the network model, locking could be based on sets. In the hierarchy shown in Figure 10.21, we generalize the nodes to be data-model-independent. The usual practice is to limit the locking granularity to the record occurrence level.

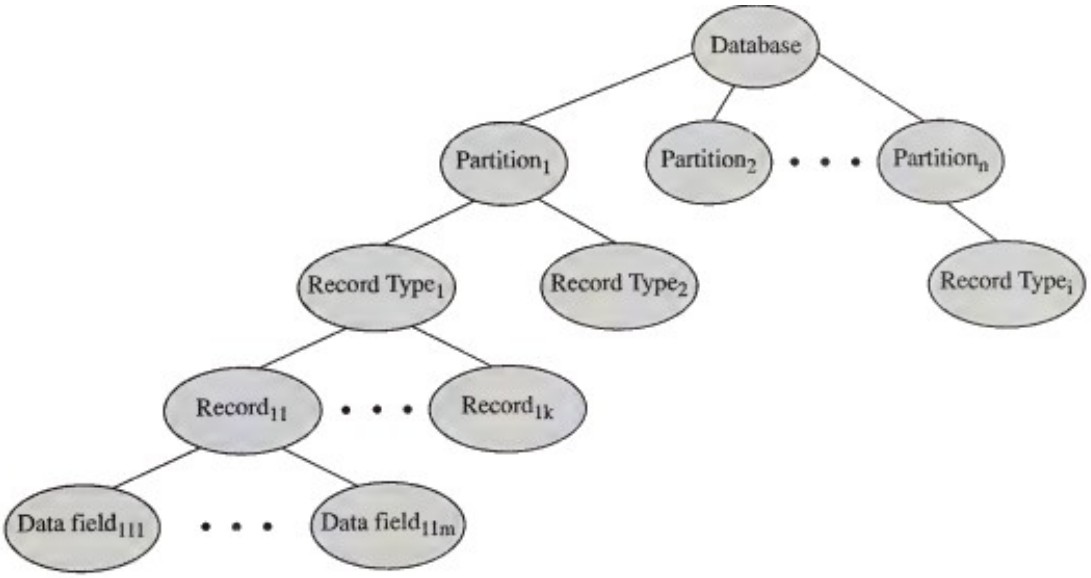


Figure 10.16 *Hierarchical Structure of the Database*

Having structured the database objects in a hierarchy, the corresponding locking scheme becomes a hierarchy; the lock manager would allow each node of the hierarchy to be locked. A hierarchy of locks provides greater flexibility and efficiency in locking. Such a scheme allows multiple granularity of locking from a data field to the entire database. The descendants of a locked node are implicitly locked in the same mode (shared or exclusive) as the node. However, if a sub-tree is locked, then it is required that the ancestor of the sub-tree not be allowed to be locked; this is so because locking an ancestor of the sub-tree implicitly locks the sub-tree. An implicit lock on a node signifies that no other transaction is allowed to lock that node (either implicitly or explicitly) in an exclusive mode (and, hence, implicitly, any of the descendants).

Hierarchical organization of the database, however, increases the overhead in determining if a request for a lock from a transaction can be accepted or not. Consider a portion of the database that is under the hierarchy specified by the node N. Suppose the transaction T_0 needs a share lock on this portion of the database; how can the lock manager know efficiently if any other transaction has locked some portion of the database rooted by node N, and if so, whether the mode is compatible with the request of transaction T_0? Checking each and every data-item under N is very inefficient.

In the case of hierarchical structured locking, a new mode of locking, called the ***intention mode*** is introduced. A transaction can lock a hierarchically structured data-item in the intention mode. The locking

of a node in this mode implies that the transaction intends to explicitly lock a lower portion of the hierarchy. In effect intention locks are placed on all ancestors of a node until the node that is to be locked explicitly is reached.

To allow a higher degree of concurrency, the intention mode of locking is refined to *intention share* and *intention exclusive* modes. The intention mode simply indicates that the transaction intends to lock the lower level in some mode. If the transaction T_a intends to lock the lower level in the share mode, then the ancestor is locked in the ***intention share mode*** to indicate that the lower level is being locked in a share mode. Other transactions can access the node and all its lower levels, including the sub-tree being accessed by T_a; no transaction however, can modify any portion of the database rooted at the node that was locked by T_a in the intention share mode. If the transaction T_a intends to lock the lower level in the exclusive or share mode, then the ancestor is locked in the ***intention exclusive mode*** to indicate that the lower level is being locked in an exclusive or share mode. Another concurrent transaction, say T_b needing to access any portion of this intention exclusively locked hierarchy in the exclusive or share mode can also lock this node in the intention exclusive mode. If T_b needs exclusive or share access to that portion of the sub-tree not being used by transaction T_a, it will place appropriate locks on them and can run concurrently with T_a. However, if T_b needs access to any portion of the sub-tree locked in the exclusive mode by T_a, then the explicit exclusive locks on these nodes will cause T_b to wait until the transaction T_a releases these explicit exclusive locks.

The intention lock locks a node to indicate that the lower level nodes are being locked either in the share or the exclusive mode, but it does no implicit locking of lower levels. Each lower level has to be locked explicitly in whichever mode required by the transaction. This adds a fairly large overhead if a transaction needs to access a sub-tree of the database and modify only a small portion of the sub-tree rooted at the intentionally locked node. The ***share and intention exclusive mode*** of locking is, thus, introduced. The share and intention exclusive mode differs from the other form of intention locking, in so far as it implicitly locks all lower level nodes, as well as the node in question. This mode allows access by other transactions to share that portion of the sub-tree not exclusively locked, and gives higher concurrency than achievable with a simple exclusive lock. The overhead of locking the root node and all nodes in the path leading to the sub-tree to be modified in the intention exclusive mode, followed by locking the sub-tree to be modified in the exclusive mode, is thus avoided. This is replaced by locking the root node in the share and intention exclusive mode (which will lock all descendents implicitly in the same mode), followed by locking the root node of the sub-tree to be modified in the exclusive mode.

We summarize, below, the possible modes in which a node of the database hierarchy could be locked and the effect of the locking on the descendants of the node.

- **S** or **Shared Lock**: The node in question *and implicitly all its descendents* are locked in the share mode; all these nodes, locked explicitly or implicitly, are accessible for read-only access. No transaction can update the node or any of its descendents when the node is locked in the shared mode.
- **X** or **Exclusive Lock**: The node in question *and implicitly all its descendents* are exclusively locked by a single transaction. No other transaction can concurrently access these nodes.
- **IS** or **Intention Share**: The node is locked in the intention share mode, which means that it or its descendents cannot be exclusively locked. The descendent of the node may be individually locked in a shared or intention shared mode. The descendents of the node that is locked in the IS mode are not locked implicitly.

- **IX** or **Intention Exclusive**: The node is locked in an intention exclusive mode. This means that the node itself cannot be exclusively locked, however, any of the descendents, if not already locked, can be locked in any of the locking modes. The descendents of the node that is locked in the IX mode are not locked implicitly.
- **SIX** or **Shared and Intention Exclusive**: The node is locked in the shared and intention exclusive mode and all the descendents are implicitly locked in the shared mode. However, any of the descendents can be explicitly locked in the exclusive, intention exclusive or shared and intention exclusive mode.

Relative Privilege of the Various Locking Modes

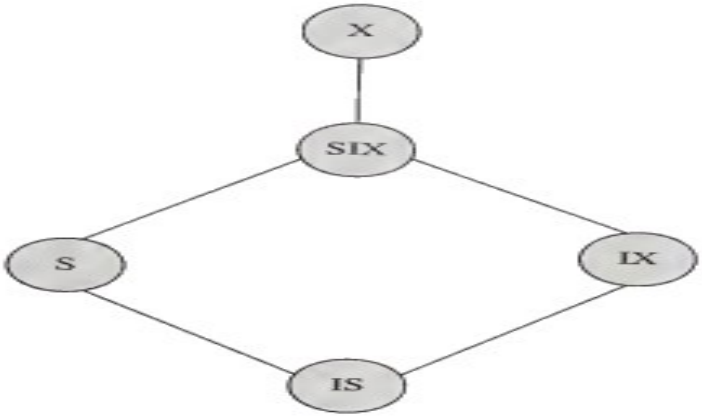


Figure 10.17 *Relative Privilege of the Locking Modes*

Figure 10.17 gives the relative privilege of the various modes of locking. The exclusive mode has the highest privilege: it locks out all other transactions from the portion of the database that is rooted at the node locked in the exclusive mode. All descendents of the node are implicitly locked in the exclusive mode. The intention share mode has the lowest privilege. The share mode is not comparable with the intention exclusive mode. Figure 10.18 gives the compatibility of the lock request, vis-a-vis the current state of locking of a node.

The advantage of the intention mode locking is that the lock manager knows that the lower level nodes of a node intentionally locked are or are being locked without having to examine all the lower level nodes. Furthermore, using the compatibility matrix shown in Figure 10.18 and discussed below, the lock manager can ascertain if a request for a lock can be granted.

Compatibility Matrix:

Considering all the modes of locking described above, the compatibility between the current mode of locking for a node and the request of another transaction for locking the node in a given mode are given below in Figure 10.18. The entry yes indicates that the request will be granted and the transaction can continue. The entry no indicates that the request cannot be granted and the requesting transaction will have to wait.

Current state of lock of the node

Request for locking

	IS	IX	S	SIX	X	unlocked
IS	yes	yes	yes	yes	no	yes
IX	yes	yes	no	no	no	yes
S	yes	no	yes	no	no	yes
SIX	yes	no	no	no	no	yes
X	no	no	no	no	no	yes
UNLOCK	yes	yes	yes	yes	yes	yes

Figure 10.18 *Access Mode Compatibility*

Locking Principle

With the above locking modes, the procedure to be followed in locking can be summarized as follows:

- A transaction is not allowed to request additional locks if it has released a lock (This is the two phase locking protocol requirement).
- The access mode compatibility matrix determines if a lock request can be granted or the requesting transaction has to wait.
- A transaction is required to request a lock in a root-to-leaf direction and a transaction is required to release locks in the opposite direction, that is, from leaf to root. Consequently, a transaction cannot unlock a node, if it currently holds a lock on one of the descendent of the node. Similarly, a transaction cannot lock a node, unless it already has a compatible lock on the ancestor of the node.
- A transaction can lock a node in the IS or the S mode only if the transaction has successfully locked the ancestors of the node in the IX or the IS mode.
- A transaction can lock a node in the IX, SIX or X mode, only if the transaction has successfully locked the ancestors of the node in the IX or the SIX mode.
- The lock manager can lock a larger portion of the database than requested by a transaction and the duration of this lock could be for a period longer than needed by the transaction.

The above locking protocol ensures serializability. Let us consider Examples 10.3 and 10.4 to illustrate the locking procedures to be followed on a database stored in a structure which is hierarchical, as shown below in Figure 10.19.

Example 10.2: To lock record occurrence R_{13} of record type 1 for retrieval only, the sequence of locking is as follows: (i) Lock Database in the IS mode, (ii) Lock Partition$_1$ in the IS mode, (iii) lock Record Type$_1$ in the IS mode, (iv) lock record R_{13} in the S mode.

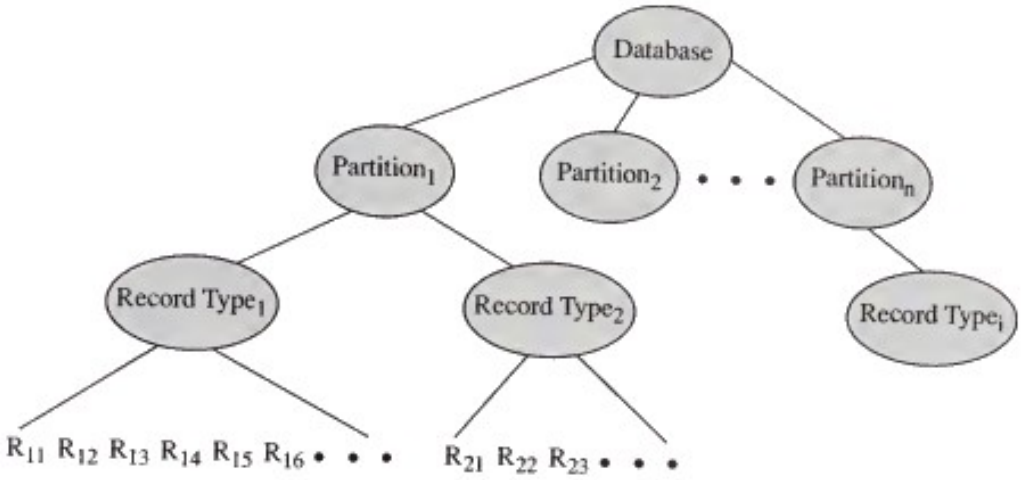


Figure 10.19 *Sample Database Storage Structure*

Example 10.3: To lock the record occurrence R_{22} of record type 2 in the exclusive mode, the sequence of locking to be used is as follows: (i) Lock the database in the IX mode, (ii) lock $Portion_1$ in the IX mode, (iii) lock $Record\ Type_2$ in the IX mode, (iv) lock record R_{22} in the X mode.

Note that if two transactions were accessing records R_{ik} and R_{il}, for $k \neq l$, in the share and exclusive modes respectively, then both these transactions can be executed concurrently if the sequence of locking for the first transaction was IS, IS, IS, and S; and that for the second transaction IX, IX, IX, and X.

10.4.4 Tree Locking Protocol

Let us assume that the storage structure of the database is in the form of a tree of data-items as shown in Figure 10.19. Then a locking protocol, called a tree locking protocol, can be defined as follows:

- All locks are exclusive locks.
- Locking a node does not automatically lock any descendent of the locked node.
- The first item locked by a transaction can be any data-item including the root node.
- Except for the first data-item locked by a transaction, a node cannot be locked by a transaction unless the transaction has already successfully locked its parent.
- No items are locked twice by a transaction: thus, releasing a lock on a data-item implies that the transaction will not attempt another lock on the data-item.

A schedule for a set of transactions, such that each transaction in the set uses the tree locking protocol, can be shown to be serializable. It must be noted that the transactions need not be two-phase and they are allowed to unlock an item before locking another item. The only requirement is that the transaction must have a lock on the parent of the node being locked and that the item was not previously locked by the transaction.

Consider the database of Figure 10.19. A transaction, for instance T_a, can start off by locking the entire database, then it proceeds to lock $Portion_1$, $Record\ Type_1$ and $Record\ Type_2$. At this point it unlocks the database and then locks record occurrences R_{11} and R_{21}, followed by unlocking $Portion_1$, and $Record\ Type_2$. Another transaction, T_b, can then proceed by first locking $Record\ Type_2$ followed by locking record occurrences R_{22}. The first transaction can now lock record occurrence R_{12}.

The advantage of the tree-locking protocol over the two-phase locking protocol is that a data-item can be released earlier by a transaction if the data-item (and of course, any of its descendants in the sub-tree rooted at the data-item) is not required by **directed acyclic graph** the transaction. In this way a greater amount of concurrency is feasible. However, since a descendant is not locked by the lock on a parent, the number of locks and associated locking overhead, including the resulting waits, is increased.

10.4.5 DAG Database Storage Structure

The use of indices to obtain direct access to the records of the database causes the hierarchical storage structure to be converted into a **directed acyclic graph** (DAG) as shown below in Figure 10.20. The locking protocol can be extended to a DAG structure; the only additional rules that need be applied is that to lock a node in the IX, SIX, or the X mode, all the parents of the node have to be locked in a compatible mode which is at least an IX mode. Thus, no other transaction can get a lock to any of the parents in the S, SIX, or the X mode. This is illustrated below:

Example 10.5: To add a record occurrence to the record type R_1, which uses an index I_1 for direct access to the records, the sequence of locking is as follows; (i) lock the database in the IX mode, (ii) lock $Portion_1$ in the IX mode, and, (iii) lock $Record\ Type_1$ and index I_1 in the X mode. With this method of locking, the phantom phenomenon is avoided at the expense of lower concurrency.

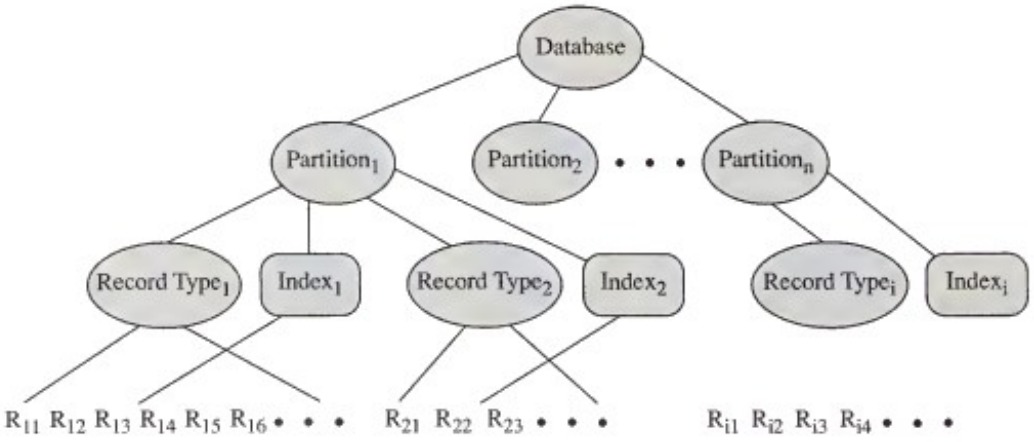


Figure 10.20 *Sample DAG Database Storage Structure*

As in the case of two-phase locking, deadlock is possible in the locking scheme using hierarchical granularity of locking. Additional details regarding references to techniques to reduce and eliminate such deadlock is cited in the bibliographic notes.

10.5 Timestamp Based Order

In the timestamp based method, a serial order is created among the concurrent transaction by assigning to each transaction an unique non-decreasing number. The usual value assigned to each transaction is the system clock value at the start of the transaction and, hence, the name -timestamp ordering. A variation to this scheme that is used in a distributed environment includes the site of a transaction appended to the system wide clock value. This value can then be used in deciding the order in which the conflict between two transactions is resolved. A transaction with a smaller timestamp value is considered to be an "older" transaction than another transaction which has a larger timestamp value.

The serializability that the system enforces, then, is the chronological order of the time-stamps of the concurrent transactions. If two transaction T_i and T_j with the timestamp values t_i and t_j respectively, such that $t_i<t_j$, are to run concurrently, then the schedule that is produced by the system is equivalent to running, first, the older transaction T_i, followed by the younger one, T_j.

The contention problem between two transactions in the timestamp ordering system is resolved by rolling back one of the conflicting transactions. The existence of conflict is determined by using the following scheme. A conflict is said to occur when an older transaction tries to read a value that is written by a younger transaction or when an older transaction tries to modify a value already read or written by, a younger transaction. Both of these attempts signify that the older transaction was "too late" in performing the required read/write operations and it could be using values from different "generations" for different data-items.

In order for the system to determine if an older transaction is processing a value already read by or written by a younger transaction, each data-item has, in addition to the value of the item, two time-stamps: a **write-timestamp** and a **read-timestamp**. The data-item X is, thus, represented by a triple X: $\{x, W_x, R_x\}$ where each component of the triple is interpreted as given below:

x, the value of the data-item X.

W_x, the write timestamp value, the largest timestamp value of any transaction that was allowed to write a value of X.

R_x, the read timestamp value, the largest timestamp value of any transaction that was allowed to read the current value X.

Now let us see how these timestamp values find their way into the data structure of a data-item and how all these values are modified. A transaction T_a with the timestamp value of t_a issues a read operation for the data-item X with the values $\{x, W_x, R_x\}$.

- This request will succeed if $t_a \geq W_x$, since the transaction T_a is younger than the transaction that last wrote (or modified) the value of X. The transaction T_a is allowed to read the value x of X and if the value t_a is larger than R_x, then t_a becomes the new value of R_x.
- This request will fail if $t_a < W_x$, i.e., the transaction T_a is an older transaction than the last transaction that wrote the value of X.

The failure of the read request is due to the fact that the older transaction was trying to read a value that had been overwritten by a younger transaction. The transaction T_a is too late to read the previous outdated value and any other values it has acquired are likely to be inconsistent with the updated value of X. It is, thus, safe to abort and rollback T_a. T_a is assigned a ***new*** timestamp and restarted.

A transaction T_a with the timestamp value of t_a issues a write operation for the data-item X with the values {x, W_x, R_x}.

- If $t_a \geq W_x$ and $t_a \geq R_x$, i.e., both the last transaction that updated the value of X and the last transaction that read the value of X are older than the transaction T_a, then T_a is allowed to write the value of X and t_a becomes the current value of W_x, the write timestamp.
- If $t_a < R_x$, then it means that a younger transaction is already using the current value of X and it would be in error if the value of X is updated. The transaction T_a is not allowed to modify the value of X. T_a is rolled back and its timestamp is reset to the current system generated timestamp value and restarted.
- If $R_x \leq t_a < W_x$, then this means that a younger transaction has already updated the value of X, and the value that T_a is writing must be based on an obsolete value of X and, hence, be obsolete. The transaction T_a is not allowed to modify the value of X: its write operation is ***ignored***.

The reason for ignoring the write operation in the last alternative is as follows. In the serial order of transaction processing, transaction T_a with the timestamp of t_a wrote the value for the data-item X. This was followed by another write operation to the same data-item by a younger transaction with a timestamp of W_x. No transaction read the data-item between the writing by T_a and the time W_x. Hence, ignoring the writing by T_a is to indicate that the value written by T_a was immediately overwritten by a younger transaction at time W_x.

Transaction T_{22}	Transaction T_{23}
Sum := 0	*Sum* := 0
Read (*A*)	**Read** (*A*)
Sum := *Sum* + *A*	*A* := *A* -100
Read (*B*)	**Write** (*A*)
Sum := *Sum* + *B*	*Sum* := *Sum* + *A*
Show(*Sum*)	**Read** (*B*)
	B := *B* + 100
	Write (*B*)
	Sum := *Sum* + *B*
	Show(*Sum*)

Figure 10.21 *Transactions for Examples 10.5, 10.6, 10.7, 10.8. 10.9, 10.10*

Let us illustrate the timestamp ordering by considering the transactions T_{22} and T_{23} given below in Figure 10.21. Each of these transactions has a local variable *Sum* and the intent is to show a user the sum of two data-items *A* and *B*. However, transaction T_{23} not only reads these values, but it also transfers 100 units from *A* to *B* and writes out to the database the modified values. Now let us suppose that $t_{23} > t_{22}$. This means that T_{23} is a transaction younger than transaction T_{22}. Also, let the data-items *A* and *B* be stored as follows: (here the W_i's and R_i's have some values assumed to be less than t_{22} and t_{23}): *A*: 400, W_a, R_a *B*: 500, W_b, R_b.

Example 10.6: Consider the transactions of Figure 10.21. In the schedule, given in Figure E both transactions $T_{22}(t_{22})$ and $T_{23}(t_{23})$ run concurrently and produce the correct result. A similar serializable schedule could have been obtained using the two phase locking protocol. See exercise 10.5)

Step	Schedule	Transaction T_{22}	Transaction T_{23}
1	*Sum* := 0	*Sum* := 0	
2	**Read** (*A*)	**Read** (*A*)	
3	*Sum* := *Sum* + *A*	*Sum* := *Sum* + *A*	
4	*Sum* := 0		*Sum* := 0
5	**Read** (*A*)		**Read** (*A*)
6	*A* := *A* -100		*A* := *A* -100
7	**Write** (*A*)		**Write** (*A*)
8	**Read** (*B*)	**Read** (*B*)	
9	*Sum* := *Sum* + *B*	*Sum* := *Sum* + *B*	
10	**Show** (*Sum*)	**Show** (*Sum*)	
11	*Sum* := *Sum* + *A*		*Sum* := *Sum* + *A*
12	**Read** (*B*)		**Read** (*B*)
13	*B* := *B* + 100		*B* := *B* + 100
14	**Write** (*B*)		**Write** (*B*)
15	*Sum* := *Sum* + *B*		*Sum* := *Sum* + *B*
16	**Show** (*Sum*)		**Show** (Sum)

Figure E *Serializable Schedule Based on Timestamp Scheme*

The steps of the schedule of Figure E, cause the following modifications to the triple for *A* and *B*:

Initially *A*: 400, W_a, R_a *B*: 500, W_b, R_b
After step 2 *A*: 400, W_a, t_{22} *B*: 500, W_b, R_b
After step 5 *A*: 400, W_a, t_{23} *B*: 500, W_b, R_b
After step 7 *A*: 300, t_{23}, t_{23} *B*: 500, W_b, R_b
After step 8 *A*: 300, t_{23}, t_{23} *B*: 500, W_b, t_{22}
After step 10 the value displayed will be 900
After step 12 *A*: 300, t_{23}, t_{23} *B*: 500, W_b, t_{23}
After step 14 *A*: 300, t_{23}, t_{23} *B*: 600, t_{23}, t_{23}
After step 14 the value displayed will be 900

In the following example we illustrate a schedule where the older transaction is rolled back.

Example 10.7: Consider the schedule, shown in Figure F: transaction T_{22} is rolled back and rerun after step 6. When it is rolled back, a new timestamp value $t_{22'}$ which would be greater than t_{23}, is

assigned to it. The sequence of changes are given below: Initially

Step	Schedule	Transaction T_{22}	Transaction T_{23}
1	*Sum* := 0	*Sum* := 0	
2	*Sum* := 0		*Sum* := 0
3	**Read** (*A*)		**Read** (*A*)
4	*A* := *A* -100		*A* := *A* -100
5	**Write** (*A*)		**Write** (*A*)
6	**Read** (*A*)	**Read** (*A*) ***causes a rollback of T_{22}.**	
7	*Sum* := *Sum* + *A*		*Sum* := *Sum* + *A*
8	**Read** (*B*)		**Read** (*B*)
9	*B* := *B* + 100		*B* := *B* + 100
10	**Write** (*B*)		**Write** (*B*)
11	*Sum* := *Sum* + *B*		*Sum* := *Sum* + *B*
12	**Show**(*Sum*)		**Show**(*Sum*)
13	*Sum* := 0	*Sum* := 0 **with a timestamp $t_{22}'(>t_{23})$**	
14	**Read** (*A*)	**Read** (*A*)	
15	*Sum* := *Sum* + *A*	*Sum* := *Sum* + *A*	
16	**Read** (*B*)	**Read** (*B*)	
17	*Sum* := *Sum* + *B*	*Sum* := *Sum* + *B*	
18	**Show**(*Sum*)	**Show**(*Sum*)	

Figure F *Serializable Schedule Produced After a Rollback*

Initially *A*: 400, W_a, R_a *B*: 500, W_b, R_b

After step 3 *A: 400, W_a, t_{23}* *B: 500, W_b, R_b* After step 5 *A*: 300, t_{23}, t_{23} *B*: 500, W_b, R_b

After step 6 *A*: 300, t_{23}, t_{23} *B*: 500, W_b, R_b* causes a rollback of T_{22} which would be reassigned a new timestamp ($t_{22'} > t_{23}$) and it would re-executed.

After step 8 *A*: 300, t_{23}, t_{23} *B*: 500, W_b, t_{22}

After step 10 *A*: 300, t_{23}, t_{23} *B*: 600, t_{23}, t_{23}

After step 12 the value displayed will be 900

After step14 *A*: 300, t_{23}, $t_{22'}$ *B*: 600, t_{23}, t_{23}

After step16 *A*: 300, t_{23}, $t_{22'}$ *B*: 600, t_{23}, $t_{22'}$

After step 18 the value displayed will be 900

Example 10.8 below illustrates a case where the write operation of a transaction could be ignored.

Example 10.8: In the example, illustrated in Figure 10.24, we have three transactions, T_{24}, T_{25}, and T_{26} with timestamp values of t_{24}, t_{25},and t_{26} respectively ($t_{24} < t_{25} < t_{26}$). Note that transactions T_{24} and T_{26} are write-only with respect to data-item *B*.

Step	Schedule	Transaction T_{24}	Transaction T_{25}	Transaction T_{26}
1	**Read** *A*	**Read** *A*		
2	*A* := *A* +1	*A* := *A* +1		
3	**Write** *A*	**Write** *A*		
4	**Read** *C*		**Read** *C*	
5	*C* := *C**3		*C* := *C**3	
6	**Read** *C*			**Read** *C*
7	**Write** *C*		**Write** *C* * causes rollback of transaction T_{25}	
8	*C* := *C**2			*C* := *C**2
9	**Write** *C*			**Write** *C*
10	*B* := 100			*B* := 100
11	**Write** *B*			**Write** *B*
12	*B* := 150	*B* := 150		
13	**Write** *B*	**Write** *B* * causes the write operation to be ignored		
14	**Read** *C*		**Read** *C*	
15	*C* := *C**3		*C* := *C**3	
16	**Write** *C*		**Write** *C*	

Figure G *Another Serializable Schedule*

Initially	*A*: 10, W_a,R_a	*B*: 50, W_b, R_b	*C*: 5, W_c, R_c
After step 1	*A*: 10, W_a,t_{24}	*B*: 50, W_b, R_b	*C*: 5, W_c, R_c
After step 3	*A*: 11, t_{24},t_{24}	*B*: 50, W_b, R_b	*C*: 5, W_c, R_c
After step 4	*A*: 11, t_{24},t_{24}	*B*: 50, W_b, R_b	*C*: 5, W_c, t_{25}
After step 5	*A*: 11, t_{24},t_{24}	*B*: 50, W_b, R_b	*C*: 5, W_c, t_{25}
After step 6	*A*: 11, t_{24},t_{24}	*B*: 50, W_b, R_b	*C*: 5, W_c, t_{26}
At step 7 transaction T_{25} with a timestamp value of t_{25} attempts to write the value of *C*: however, since the read timestamp value of *C* is t_{26} , which is greater than t_{25} transaction T_{25} would be rolled back: the transaction would be reassigned a timestamp value of, let us say $t_{25'}$(>t_{26}) and rerun at step 14.			
After step 9	*A*: 11, t_{24},t_{24}	*B*: 50, W_b, R_b	*C*: 10, t_{26}, t_{26}
After step 11	*A*: 11, t_{24},t_{24}	*B*:100, t_{26},R_b	*C*: 10, t_{26}, t_{26}
At step 13, the attempt by transaction T_{24} to write a value of *B* is ignored, since t_{24}, the timestamp of T_{24} is less than the write timestamp (t_{26}) of *B*, and it is greater than the read timestamp value (R_b) of *B*.			
After step 14	*A*: 11, t_{24},t_{24}	*B*:100, t_{26},R_b	*C*: 10, t_{26},$t_{25'}$
After step 16	*A*: 11, t_{24},t_{24}	*B*:100, t_{26},R_b	*C*: 30, $t_{25'}$,$t_{25'}$

It is obvious from the above examples that the time-stamping scheme ensures serializability without waiting but causes transactions to be rolled back. Since there is no waiting there is no possibility of a deadlock. However, when transactions are rolled back, a cascading rollback may be needed. For instance, if transaction T_{22} had written a value for a data-item, Q, before it was rolled back, then this data-item

value must be restored to its old value. If another transaction, T' had used the modified value of the data-item Q, then transaction T' has to be rolled back as well.

The cascading rollback could be avoided by disallowing the values modified by a transaction to be used until the transaction commits. This adds additional overhead and requires waiting as in the case of the locking scheme. Furthermore, the waiting can cause a deadlock!

10.6 Optimistic Scheduling

In the locking scheme, a transaction does a two-pass operation. In the first pass it locks all the data-items required by it and if all locks are successfully acquired, it goes through the second pass of actually accessing and modifying the required data-items. In the optimistic scheduling scheme, the philosophy is to be an optimist and assume that all data-items can be successfully updated at the end of a transaction, and it reads in the values for data-items without any locking. Reading is done when required and if any data-item is found to be inconsistent (with respect to the value read in) at the end of a transaction, then the transaction is rolled back. Since a DBMS normally has a rollback facility built-in for recovery operations, the optimistic approach does not require any additional components. For most transactions, which access the database for read only operations and modify disjoint sets of data-items, the optimistic scheduling scheme performs better than the two-pass locking based approach.

In the optimistic approach, each transaction is made up of three phases: the read phase, the validation phase and the write phase. The read phase is not constrained but the write phase is severely constrained, and any conflicts could cause a transaction to be aborted and rolled back. It should be noted that displaying a value of a data-item or a derived value of a set of data-items to a user is equivalent to a write operation (even though no items are modified). The optimistic technique uses a timestamp method to assign an unique identifier to each transaction, as well as for the end of the validation and the write phases. The three phases are described below.

Read Phase: This phase starts with the activation of a transaction and is considered to last until the commit. All data-items are read into local variables and any modifications that are made are only to these local copies.

Validation Phase: For data-items that were read, the DBMS will verify that the values read are still the current values of the corresponding data-items. For data-items that are modified (a deletion and an insertion can be considered as modifications), the DBMS verifies that the modifications will not cause the database to become inconsistent. Any change in the value of data-items read or any possibility of inconsistencies due to modifications causes the transaction to be rolled back.

Write Phase: If a transaction has passed the validation phase, the modifications made by the transaction are committed.

The three time-stamps associated with each transactions are the following:

- t_{si}: The start timestamp for transaction T_i. We assume that a transaction starts its read phase when it starts.
- T_{vi}: The timestamp for transaction T_i when it finishes its read phase and starts it validation phase. This will occur when the transaction completes. All writes prior to the start of the validation phase will be to local copies of database items and these local copies will not be accessible to

other concurrent transactions.

- t_{wi}: The timestamp for transaction T_i when it completes its write phase. The write phase will only start if the transaction completes the validation phase successfully. After the write phase, all modifications are reflected in the database.

A transaction, such as T_j, can complete its validation phase successfully if at least ***one*** of the following conditions is satisfied:

- For all transactions T_i, such that $t_{si} < t_{sj}$, the condition $t_{wi} < t_{sj}$ holds. This condition ensures that all older transactions must have completed their write phases before the requesting transaction began.

- For all transactions T_i, such that $t_{si} < t_{sj}$, i.e., for all older transactions, the data-items modified by T_i must be disjoint from the data-items read by the transactions T_j. Furthermore, all older transactions must complete their write phase before time t_{vj}. Here t_{vj} is the time at which transaction T_j finishes its read phase and starts its validation phase. This ensures that a younger transaction's writes are not overwritten by an older transactions writes.

- For all transactions T_i, such that $t_{si} < t_{sj}$, i.e., for all older transactions, the data-items modified must be disjoint from the data-items read or modified by the transactions T_j. Furthermore, $t_{iv} < t_{jv}$, which ensures that the older transaction, T_i, completes its read phase before T_j completes its read phase. In this way the older transaction cannot influence the read or write phase of T_j.

Example 10.9 Consider the transactions T_{22} and T_{23} of Figure 10.21 and the schedule of Figure H.

Step	Schedule	Transaction T_{22}	Transaction T_{23}
1	*Sum* := 0	*Sum* := 0	
2	*Sum* := 0		*Sum* := 0
3	**Read** (*A*)		**Read** (*A*)
4	*A* := *A* -100		*A* := *A* -100
5	**Read** (*A*)	**Read** (*A*)	
6	*Sum* := *Sum* + *A*	*Sum* := *Sum* + *A*	
7	**Write** (*A*)		**Write** (*A*)
8	**Read** (*B*)	**Read** (*B*)	
9	*Sum* := *Sum* + *B*	*Sum* := *Sum* + *B*	
10	**Show**(*Sum*)	**Show**(*Sum*)	
11	*Sum* := *Sum* + *A*		*Sum* := *Sum* + *A*
12	**Read** (*B*)		**Read** (*B*)
13	*B* := *B* + 100		*B* := *B* + 100
14	**Write** (*B*)		**Write** (*B*)
15	*Sum* := *Sum* + *B*		Sum := *Sum* + *B*
16	**Show**(*Sum*)		**Show**(*Sum*)

Figure H *Example for Optimistic Scheduling*

The initial value of *A* and *B* are as follows:

A: 400, W_a, R_a *B*: 500, W_b, R_b
The progress of the concurrent execution of transaction T_{22} and T_{23} causes the following actions:
At step 7,and 14, the write is only local and the actual write to the database would be delayed until all reads are completed.
At step 10, before the value of *Sum* is displayed, the validation phase for transaction T_{22} would find that there are no writes from older transactions outstanding and its validation will be successful and the value of *Sum* would be displayed.
At step 16, before the value of *Sum* is displayed, the validation phase for transaction T_{23} would find that there are no writes from older transactions outstanding and its validation would be successful.
Consequently the writes to *A* and *B* as well as the display of *Sum* would be completed.

Consider a schedule for a set of concurrent transactions. If each transcription in this set can complete its validation phase successfully with at least one of the above conditions, then the given schedule is serializable. Let us consider the following example to illustrate the optimistic scheduling.

As the optimistic scheme does not use locks, it is deadlock free even though starvation can still occur. This is due to the fact that a popular item, for instance an index, can be used by many transactions and each transaction could cause it to be modified as a result of to insertions or deletions. An older transaction can, thus, fail its validation phase continuously. The method of solving this problem involves resorting to some form of locking.

As the optimistic scheme does not use locks, it is deadlock free even though starvation can still occur. This is due to the fact that a popular item, for instance an index, can be used by many transactions and each transaction could cause it to be modified as a result of insertions or deletions. An older transaction can, thus, fail its validation phase continuously. The method of solving this problem involves resorting to some form of locking.

10.7 Multi-version Techniques

In the concurrency control schemes discussed so far, the arbitration that produced serializable schedules was required when one or more of the concurrent transactions using a part of the database needed to modify the data-item. Any modifications to data required that the transaction have exclusive use of the data, and other transactions would be locked out or aborted until the lock on the data-item was released.

In a database system which uses the multi-version concurrency scheme, each write of a data-item, e.g. X, is achieved by making a new copy or version (and hence the name multi-version) of the data-item X. The multi-version scheme, which is also called a time domain addressing scheme, follows the accounting principle of never overwriting a transaction. Any changes are achieved by entering compensating transactions. In this way, a history of the evolution of the value of a data-item is recorded in the database. As far as the users are concerned, their transaction running on a system with multi-versions will work in an identical manner as a single version system.

For data-item X the database could keep the multi-version in the form of a set of triple consisting, of the value, the time entered, and the time modified as shown below:

Variable:{ {value, time entered, time modified}, {...}, ... }
X:{ {x_0, t_o, t_1}, {x_1, t_1, t_2}, .., {x_n, t_n, t_p} }

Here the value of the data-item X is initially x_0 and this value is entered in the database at time t_0. At time t_1, the value is modified to x_1. The value x_n entered at time t_n is the last update made to the data-item X. Having many versions of a data-item, it is easy to know that the value of X from time t_0 to t_1 was x_0 and so on.

When a transaction needs to read a data-item, such as X, for which multiple versions exist, the DBMS selects one of the versions of the data-item. The value read by a transaction must be consistent with some serial execution of the transaction with a single version of the database. Thus, the concurrency control problem is transferred into the selection of the correct version from the multiple versions of a data-item.

With the multi-version technique, write operations can occur concurrently, since they do not overwrite each other. Furthermore, the read operation can read any version. This results in a greater flexibility in scheduling concurrent transactions and many schemes have been proposed for controlling concurrency using the multi-version approach. We discuss one such scheme based on time-stamping below. The concurrency control ensures, among other things, that no new version of a data-item is created such that it is based on a version that may have already been used to create yet another version. In this way the phenomenon of lost-update could be avoided.

In order to choose the correct version of data to be read by a given transaction, the multi-version time-stamping scheme uses the timestamp ordering of the concurrent transactions and the time parameters associated with each version of the data-items to be used by a transaction. The time-stamping of transaction was discussed earlier in section 10.5. As mentioned above, there are two time values associated with each version of a data-item X. These are the write-timestamp, W_x, and the read-timestamp, R_x. The significance of these timestamps is discussed below.

The write-timestamp of a version of a data-item is the time- stamp value of the transaction that wrote the version of the data-item. The new value of the data-item with the write-timestamp value W_x was written by a transaction with a timestamp value of W_x. Note, that here we are ignoring the time lapse from the start of the transaction to the generation of the new version. As such the timestamps are in reality pseudo-times and a non-decreasing counter can be used instead of a timestamp with similar results.

The read-timestamp of a version of a data-item is the time- stamp value of the most recent transaction that successfully read the version of the data-item. A version of the data-item with the read-timestamp of R_x was read by a transaction with a timestamp value of R_x. The read-timestamp value is the same as the time of modification of the value of the data-item, if another version of the data-item exists, otherwise it remains the most recent version of the data-item. This is so since a new version will usually not be created without first reading the current most recent version.

If a transaction, T_i, with a time stamp value of t_i writes a value, x_i, for the kth version of a data-item, X, then the kth version of X will have the value x_i; W_{xk}, the write-timestamp value, and R_{xk}, the read-timestamp value of X_k will both be initialized to t_i.

A transaction needing to read the value of the data-item X is directed to read that version of X that was the most recent version, with respect to the timestamp ordering of the transaction. We call this version the relative-most-recent version. Thus, if a transaction, T_a, with the timestamp value of t_a needs to read the value of the data-item X, it will read the version X_j such that W_{xj} is the largest write-stamp value of all versions of X which is less than, or equal to, t_a. The read-timestamp value of the version X_j of X, read by transaction T_a, is updated to t_a, if $t_a > R_{xj}$.

A transaction, T_a, wanting to modify a data-item value will first read the relative-most-recent version X_j of the data-item X as in the last paragraph. Now, when it tries to write a new value of X, one of the following actions will be performed:

- A new version of X, e.g. the version $X_{j'}$, is created and stored with the value $x_{j'}$ and with the timestamp values of $W_{xj'} = R_{xj'} = t_a$, if the current value of $R_{xj} \leq t_a$. This ensures that the transaction T_a was the most recent transaction which read the value of version X_j, and no other transaction has read the value that was the basis of updating by T_a).
- The transaction, T_a, is aborted and rolled back if the current value of $R_{xj} > t_a$; the reason being another younger transaction has read the value of the version X_j and may have used it and/or already modified it. Transaction T_a was too late and it should try to re-run to obtain the current most recent version of the value of X.

It is easy to see that the value of the write-timestamp is the same as the time of generation of a new version of the value of a data-item, and the read-timestamp value is the same as the time of modification of the value of the data-item.

A transaction, T_a, with a timestamp value of t_a, writing a new version of a data-item, X, without first reading, creates a new version of X with the write-timestamp and read-timestamp value of t_a.

It can be shown that any schedule generated according to the above requirements is serializable, and the result obtained by a set of concurrent transactions is the same as that obtained by some serial execution of the set with a single version of the data-items.

Example 10.10:

Step	Schedule	Transaction T_{22}	Transaction T_{23}
1	*Sum* := 0	*Sum* := 0	
2	*Sum* := 0		*Sum* := 0
3	**Read** (*A*)		**Read** (*A*)
4	*A* := *A* -100		*A* := *A* -100
5	**Write** (*A*)		**Write** (*A*)
6	**Read** (*A*)	**Read** (*A*)	
7	*Sum* := *Sum* + *A*	*Sum* := *Sum* + *A*	
8	**Read** (*B*)	**Read** (*B*)	
9	*Sum* := *Sum* + *B*	*Sum* := *Sum* + *B*	
10	**Show**(*Sum*)	**Show**(*Sum*)	
11	*Sum* := *Sum* + *A*		*Sum* := *Sum* + *A*
12	**Read** (*B*)		**Read** (*B*)
13	*B* := *B* + 100		*B* := *B* + 100
14	**Write** (*B*)		**Write** (*B*)
15	*Sum* := *Sum* + *B*		*Sum* := *Sum* + *B*
16	**Show**(*Sum*)		**Show**(*Sum*)

Figure I *Schedule for the Multi-version Technique*

Consider the schedule given in Figure I for two concurrent transactions T_{22} and T_{23} and suppose the multi-version technique is used for concurrency control. Assume, initially, that a single version exist for the data-items *A* and *B* with their initial values being:

A:{{ 400, W_a, R_a}} and B:{{ 500, W_b, R_b}}.

Transaction T_{22} has a time stamp value of t_{22}

Transaction T_{23} has a time stamp value of t_{23}

$t_{22} < t_{23}$, $W_a < t_{22}$, $R_a < t_{22}$, $W_b < t_{22}$, $R_b < t_{22}$

The modifications after the following steps are:

After step 3 A:{{ 400, W_a, t_{23}}} B:{{ 500, W_b, R_b}}

After step 5 A:{{ 400, W_a, t_{23}},{ 300, t_{23}, t_{23}}}
B:{{ 500, W_b, R_b}}

After step 6 A:{{ 400, W_a, t_{23}},{ 300, t_{23}, t_{23}}}
B:{{ 500, W_b, R_b}}

After step 8 A:{{ 400, W_a, t_{23}},{ 300, t_{23}, t_{23}}}
B:{{ 500, W_b,t_{22}}}

After step 10 the value shown by T_{22} is 900

After step 12 A:{{ 400, W_a, t_{23}},{ 300, t_{23}, t_{23}}}
B:{{ 500, W_b,t_{23}}}

After step 14 A:{{ 400, W_a, t_{23}},{ 300, t_{23}, t_{23}}}
B:{{ 500, W_b, t_{23}},{ 600, t_{23}, t_{23}}}

After step 16 the value shown by T_{23} is 900

Note: If the value of the timestamp t_{22} was larger than the value of the timestamp t_{23} (i.e., transaction T_{22} was younger than transaction T_{23}), then at step 5 the transaction T_{23} will be aborted and rolled back(see exercise 10.7.)

The multi-version scheme never causes a read operation to be delayed, however, the overhead of the read operation is a search for the correct version of the value of the data-item and an update of the read-timestamp of the version of the value read. This is advantageous, if the majority of database operations are reads and only one version is likely to exist for most of the data-items. The locking overhead is traded for the overhead of updating the read-timestamp. Albeit, this gets expensive when an entire file is to be processed and thousands of records are read requiring the writing of the read-timestamp for this many records!

Another drawback of the multi-version scheme is that instead of forcing transactions that modify data-items to wait, it allows them to proceed with the caveat that any transaction could be rolled back if a younger transaction read the same value as an older transaction and the older transaction was too late in modifying the value. Serializability is achieved by rollback which could result in cascading, and, hence, be quite expensive.

The deadlock problem is not possible in the timestamp based multi-version scheme though cascading rollback is possible. This problem can be avoided by disallowing other transactions to use the versions created by uncommitted transactions.

10.8 Deadlock and Its Resolution

In the concurrent mode of operation each concurrently running transaction may be allowed to, exclusively, claim one or more of a set of resources. Some of the problems with this mode of operations are that of deadlock and starvation. We can illustrate the problems of deadlock and starvation with the following examples. Here T_a, T_b, T_c, ..., T_n are a set of concurrent transactions and r_a, r_b, r_c, .., r_m are a set of shared data-items (resources). Each transaction can claim any number of these data-items exclusively.

Suppose we have a situation where transaction T_a has claimed data-item r_a and is waiting for data-item r_b. Data-item r_b, however, has been claimed by transaction T_b, which in turn is waiting for data-item r_c. This chain of transactions holding some data-items, and waiting for additional data-items continues until we come to transaction T_i, which has claimed data-item r_i, and is waiting for data-item r_a. We know that data-item r_a is held by transaction T_a! If none of these transactions are willing to release the data-items they are holding, we have a situation where none of these transactions can proceed. This is deadlock

The situation of starvation can occur if there is a transaction which is waiting for data-item r_i. However, the resource allocation method used by the system, along with the mix of transactions, is such that every time the resource, r_i, becomes available, it is assigned to some other transaction. This results in transaction T_i having to continue to wait. (Not unlike waiting for Godot.[2])

10.8.1 Deadlock Detection and Recovery

In the deadlock detection and recovery approach to handling the deadlock situation, the philosophy is to do nothing to avoid a deadlock. However, the system monitors the advance of the concurrent transactions and detects the symptoms of deadlock, namely, a chain of transactions all waiting for a resource that the next transaction in the chain has obtained in an exclusive mode.

The reason for this philosophy is that if deadlocks are rare, then the overhead of ensuring that there is no deadlock is very high, and the occasional deadlock and the recovery from it is a small price to be paid for doing nothing until a deadlock actually develops. In addition deadlock avoidance schemes avoid all potential deadlocks, even those that do not translate into an actual deadlock.

In order for the system to detect a deadlock, the system must have the following information:

- the current set of transactions,
- the current allocations of data-items to each of the transactions,
- the current set of data-items for which each of the transactions is waiting.

The system uses this information and applies an algorithm to determine if some proper subset of these transactions are in a deadlock state. If the system finds this to be the case, then it attempts to recover from the deadlock by breaking the cyclic chain of waiting transactions.

We present, below, an algorithm for deadlock detection and a method of recovery.

2 Samuel Beckett, "Waiting for Godot: A Tragicomedy in Two Acts", Grove Press (May 27 2011),ISBN-13:978-0802144423

Deadlock Detection

A deadlock is said to occur when there is a circular chain of transactions, each waiting for the release of a data-item, held by the next transaction in the chain. The algorithm to detect deadlock is based on the detection of such a circular chain in the current system, ***wait-for-graph***. The wait-for-graph is a directed graph and contains nodes and directed arcs: the nodes of the graph are active transactions. An arc of the graph is inserted between two nodes, if there is a data-item that is required by the node at the tail of the arc, which is being held by the node at the head of the arc. If there is a transaction, such as T_i, waiting for a data-item which is currently allocated and held by transaction T_j, then there is a directed arc from the node for transaction T_i, to the node for transaction T_j.

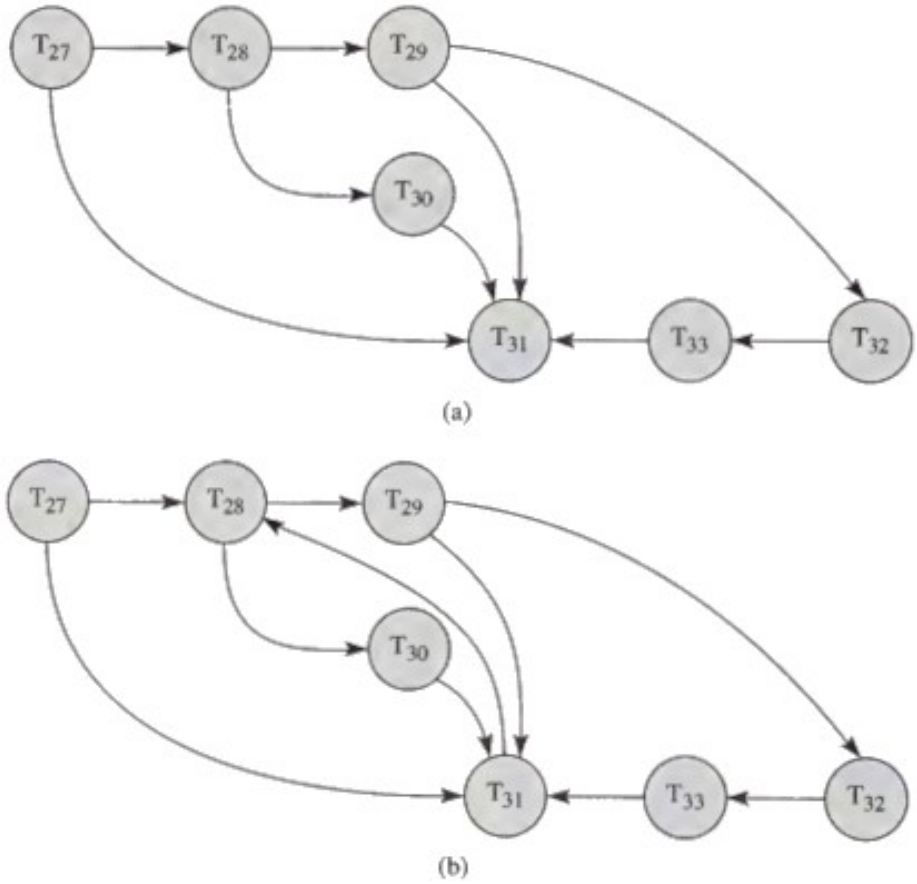


Figure 10.22 *System Wait-for-Graph showing in: (a), no cycle and, hence, no deadlock; (b) a cycle and, hence, a deadlock*

Figure 10.22 gives examples of the system wait-for-graph. In Figure 10.22(a) we have the following situation:

-the transaction T_{27} is waiting for data-items locked by transactions T_{28} and T_{31};
-the transaction T_{28} is waiting for data-items locked by transactions T_{29} and T_{30};

-the transaction T_{29} is waiting for data-items locked by transactions T_{31} and T_{32};
-the transaction T_{30} is waiting for data-items locked by transactions T_{31};
-the transaction T_{32} is waiting for data-items locked by transactions T_{33};
-the transaction T_{33} is waiting for data-items locked by transactions T_{31};

In the wait-for-graph of Figure 10.22(a), there are no cycles and, hence, the corresponding set of transactions is free from deadlock.

Figure 10.22(b) represents the state of the system after a certain period of time, when transaction T_{31} makes a request for a data-item held by transaction T_{28}. This request, assuming no previous requests depicted in the wait-for-graph of Figure 10.22(a) have been satisfied, adds the arc from the node for transaction T_{31} to the node for transaction T_{28}. Addition of this arc causes the wait-for-graph to have a number of cycles. One of these cycles is indicated by the arc from transaction T_{28} to transaction T_{30}, then, from transaction T_{30} to T_{31}, and finally from T_{31} back to T_{28}. Consequently Figure 10.32(b) represents a situation where a number of sets of transactions are deadlocked.

Since a cycle in the wait-for-graph is a necessary and sufficient condition for deadlock to exist, the deadlock detection algorithm generates the wait-for-graph at regular intervals and examines it for a chain. If the interval chosen is very small, then the deadlock detection will add considerable overhead; if the interval chosen is very large, then there is a possibility that a deadlock would not be detected for a long period. The choice of the interval is dependent on the frequency of deadlocks, and the cost of not detecting the deadlock for the chosen interval. The overhead of keeping the wait-for-graph continuously, adding arcs as requests are blocked and removing them as locks are given up would be very high.

The deadlock detection algorithm is given below. In this algorithm we use a table called Wait_for_Table. It contains columns for each of the following: transaction IDs; the data-items for which they have acquired a lock; and the data-items they are waiting for (these wait-for-items are currently locked in an incompatible mode by other transactions). The algorithm starts with the assumption that there is no deadlock. It then locates a transaction, T_s, which is waiting for a data-item. If the data-item is currently locked by the transaction T_r, then the latter is in the wait-for-graph. If T_r, in turn is waiting for a data item currently locked by a transaction T_p, then this transaction is also in the wait-for-graph. In this way finds all other transactions involved in a wait-for-graph starting with the transaction T_s. If the algorithm finally finds that there is a transaction T_q which is waiting for a data-item currently locked by T_s, .he wait-for-graph leads back to the starting transaction. Consequently the algorithm concludes that a cycle exists in the wait-for-graph and there is a potential deadlock situation.

Algorithm 10.2: Deadlock Detection
Input and Data Structure Used:
A table called Wait_for_Table as shown in Figure 10.28, contains: transaction IDs, the data-items they have acquired a lock on, and the data-items they are waiting for (these wait-for-items are currently locked in an incompatible mode by other transactions).
A boolean variable Deadlock_situation;
A first-in, first-out stack, Transaction_stack, to hold transaction IDs: this stack will contain the transactions in a deadlocked chain, if a deadlock is detected.
Output:
Whether the system is deadlocked,and if so, the transactions in the cycle
Body:
Initialize Deadlock_situation to false;
Initialize Transaction_stack to empty

```
For next transaction in table while not Deadlock_Situation
 begin
  Start_transaction := next Transaction_id;
  Push  next Transaction_id into Transaction_stack ;
      For next Data_item_waiting_for
                for transaction on top of Transaction_stack
                                        while not Deadlock_Situation
                                                and not Transaction_stack empty
       begin
                let D_next:= next Data_item_waiting_for
                find transaction which has  locked the item D_next
                 Push it's ID to Transaction_stack
                  if Start_transaction = transaction on top of stack
                          then Deadlock_situation:= true;
                end
       Pop Transaction_stack
  end
```

Transaction_Id	Data_items_locked	Data_items_waiting_for
T_{27}	*B*	*C, A*
T_{28}	*C, M*	*H, G*
T_{29}	*H*	*D, E*
T_{30}	*G*	*A*
T_{31}	*A, E*	(*C*)
T_{32}	*D, I*	*F*
T_{33}	*F*	*A*

Figure 10.23 *Wait-for-Table for Example 10.11*

Example 10.11 Consider the wait-for-table of Figure 10.23. The wait-for-graph for the transactions in this chain is given by Figure 10.22(a). It has no cycles and, hence, there are no deadlocks. However, if transaction T_{31} makes a request for data-item *C*, the wait-for-graph is converted into the one given in Figure 10.22(b). This graph has a cycle which starts at transaction T_{28}, goes through transactions T_{30}, T_{31} and back to T_{28}, and the Algorithm 10.2 detects it: there is another cycle in this graph.

An adaptive system may initially choose a fairly infrequent interval to run the deadlock detection algorithm. Every time a deadlock is detected, the deadlock detection frequency could be increased, for example, to twice the previous frequency and every time no deadlock is detected, the frequency could be reduced, for example, to half the previous frequency. Of course an upper and lower limit to the frequency would have to be established.

Recovery from Deadlock

In order to recover from deadlock, the cycles in the wait-for-graph must be broken. The common method of doing this is to rollback one or more transactions in the cycles until the system exhibits no further deadlock situation. The selection of the transactions to be rolled back is based on the following considerations:

- The progress of the transaction and the number of data-items it has used and modified. It is preferable to rollback a transaction that has just started, or has not modified any data-item, than one that has run for a considerable time and/or has modified many data-items.
- The amount of computing remaining for the transaction and the number of data-items that have yet to be accessed by the transaction. It is preferable not to rollback a transaction if it has almost run to completion and/or it needs very few additional data-items before its termination.
- The relative cost of rolling back a transaction. Notwithstanding the above considerations, it is preferable to roll back a less important or noncritical transaction than the reverse.

Once the selection of the transaction to be rolled back is done, the simplest scenario consists of rolling back the transaction to the start of the transaction, i.e., abort the transaction and restart it, *de nouveau*. If, however, additional logging is done by the system to maintain the state of all active transactions, the rollback need not be total, merely far enough to break the cycle indicating the deadlock situation. Nonetheless, the overhead of this may be excessive for many applications.

The process of deadlock recovery must also ensure that a given transaction is not continuously the one selected for rollback. If this is not avoided, the transaction will never (or at least for a period which looks like never) complete. This is starving a transaction!

10.8.2 Deadlock Avoidance

In the deadlock avoidance scheme, care is taken to ensure that a circular chain of processes holding some resources and waiting for additional ones, held by other transactions in the chain, never occurs. The two-phase locking protocol ensures serializability, but does not ensure a deadlock free situation. This is illustrated by the following example.

Example 10.12: Consider the transactions T_{34} and T_{35} given below in Figure 10.24, and the schedule of Figure 10.25. The transactions are two-phase, however, a deadlock situation exists in Figure 10.25, as transaction T_{34} waits for a data-item held by transaction T_{35}; and, later on, transaction T_{35} itself waits for a data-item held by T_{34}, which is already blocked from further progress.

Transaction T_{34}	Transaction T_{35}
Sum := 0	*Sum* := 0
Locks(*A*)	**Lockx**(*B*)
Read (*A*)	**Read** (*B*)
Sum := *Sum* + *A*	*B* := *B* + 100
Locks(*B*)	**Write** (*B*)
Read (*B*)	*Sum* := *Sum* + *B*
Sum := *Sum* + *B*	**Lockx**(*A*)
Show(*Sum*)	**Unlock**(*B*)
Unlock(*A*)	**Read** (*A*)
Unlock(*B*)	*A* := *A* -100
	Write (*A*)
	Unlock(*A*)
	Sum := *Sum* + *A*
	Show(*Sum*)

Figure 10.24 *Two Phase Transactions*

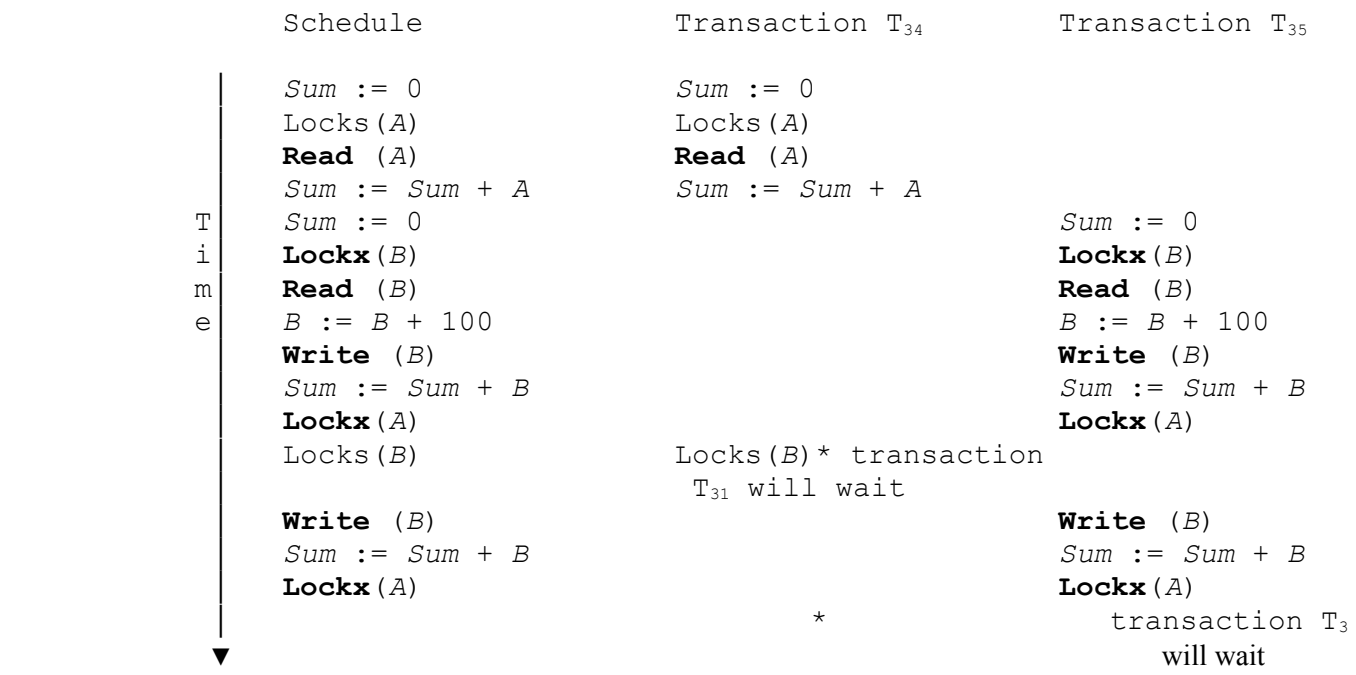


Figure 10.25 *Schedule Leading to Deadlock with Two-Phase Transactions*

One of the simplest methods of avoiding a deadlock situation is to lock all data-items at the beginning of a transaction. This will have to be done in an atomic manner, otherwise, there could be a deadlock situation again. The main disadvantage of this scheme is that the degree of concurrency is lowered considerably. A transaction typically needs a given data-item for a very short interval. Locking all data-items for the entire duration of a transaction makes these data-items inaccessible to other concurrent

transactions. This could be the case even though the transaction holding a lock on these data-items may not need them for a long time after it acquires a lock on them.

Another approach used in avoiding deadlock is assigning an order to the data-items, and requiring the transactions to request locks on the data-item in only a given order, such as only the ascending order. Thus, data-items may be ordered as having rank 1, 2, 3, etc. A transaction, T requiring data-items *A* (with a rank of say i) and *B* (with a rank of j with j>i) must first request a lock for the data-item with the lowest rank, namely *A*. When it succeeds in getting the lock for *A*, then, and only then, can it request a lock for data-item *B*. All transactions follow such protocol, even though within the body of the transaction, the data-items are not required in the same order as the ranking of the data-items for lock requests. This scheme reduces the concurrency as well, but not to the same extent as the first scheme.

Another set of approaches to deadlock avoidance is to decide whether to wait or abort and rollback a transaction, if it finds that the data-item it requests is locked in an incompatible mode by another transaction. The decision, whether to make the requesting transaction wait or abort and roll back one of the contending transactions, is controlled by their timestamp values. Aborted and rolled back transactions ***retain*** their timestamp values, and, hence, their "seniority". So, in subsequent situations of contentions they would eventually get a 'higher priority'. We examine, below, two such approaches called wait-die and wound-wait.

Wait-Die:

One solution in a case of contention for a data-item is as follows:

- if the requesting transaction is older than the transaction that holds the lock on the requested data-item, then the requesting transaction is allowed to wait, or
- if the requesting transaction is younger than the transaction that holds the lock on the requested data-item, then the requesting transaction is aborted and rolled back.

This is called the ***wait-die*** scheme of deadlock prevention.

If concurrent transactions T_{36}, T_{37}, and T_{38}; (having time-stamp values of t_{36}, t_{37} and t_{38}, respectively, with $t_{36} < t_{37} < t_{38}$) have at some instance a wait-for-graph, as shown in Figure 10.26, then transaction T_{36} would be allowed to wait, but transaction T_{38} would be aborted and rolled back.

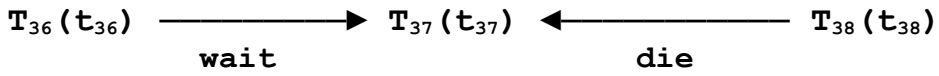


Figure 10.26 *Example for* ***Wait-die*** *Deadlock Prevention Scheme*

Wound-wait:

An opposite approach to the wait-die scheme is called ***wound-wait*** scheme. Here, the decision whether to wait or abort is as follows:

1. -f a younger transaction holds a data-item requested by an older one, then the younger transaction is the one that would be aborted and rolled back (the younger transaction is wounded by the older transaction and dies!), or

2. if a younger transaction requests a data-item held by an older transaction, the younger transaction is allowed to wait.

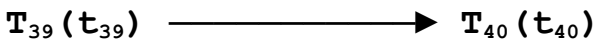


***Figure 10.27 E**xample of **Wounding** Request*

For the request shown in Figure 10.27, where transaction T_{39} has a smaller timestamp value than transaction T_{40}, the younger transaction T_{40} would be aborted and rolled back, thus freeing the data-item locked by it to be used by transaction T_{39}.

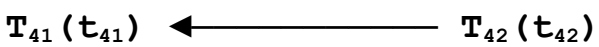


***Figure 10.28** Example of Waiting Request*

For the request shown in Figure 10.38, where transaction T_{41} has a smaller timestamp value than transaction T_{42}, the younger transaction T_{42} is allowed to wait for the completion of the older transaction T_{41}.

We observe that in neither the wait-die scheme nor the wound-wait scheme is it required to abort and rollback an older transaction. In this way the older transaction would eventually get all the data-items it needs and would run to completion. This implies that this scheme avoids the problem of starvation. However, the waiting, that may be required by an older transaction, could be significantly higher in the wait-die scheme. This is so, because the older the transaction gets, the more it has to wait for younger transactions to finish. On the other hand in the wound-wait scheme, the older a transaction gets, the greater its probability of acquiring a data-item. We explain this as follows: an older transaction would force the abortion of any younger transaction which holds data-items it needs, but would only be aborted by a transaction older than itself. However, as a transaction gets older, the number of more senior transactions would decrease!.

In the wait-die and the wound-wait schemes the first word of the scheme name indicates what the older transaction does when there is a contention for a data-item between two transactions. In the first scheme the older transaction waits for the younger transaction to finish, in the later scheme, the older transaction wounds the younger transaction which releases the data-item for the older transaction. The second component indicates what a younger transaction does when there is a contention with an older transaction. In the first scheme the younger transaction is aborted, and in the latter, the younger transaction waits.

The number of aborts and rollbacks tend to be higher in the wait-die scheme than in the wound-wait scheme. This is so, because, when a younger transaction, such as T_y, makes a requests for a data-item held by an older transaction, the younger transaction is aborted and rolled back. However, it is re-initiated with the original timestamp which it had retained. The re-initiated transaction, T_y, will make the same requests as in its last life, and it is likely that some of the data-items may still be held by older transactions. So the transaction T_y dies again, to be born again, and so on. On the other hand, in the wound-wait scheme the younger transaction, T_y, is aborted by an older transaction because the younger transaction holds a data-item needed by the older transaction. When the transaction, T_y, is re-initiated, it

will request the same data-items as in its last life. However, these data-items may still be held by the older transaction and, hence, the younger transaction merely waits.

In addition to deadlock, the problem of starvation (where one or more transactions are forced to wait forever) is also possible. For example, the situation can develop where, among the data-items required by some transaction, at least one of them is always found to be locked by another concurrent transaction.

In conclusion, we note that the disadvantage of requesting all data-items at the beginning of a transaction, and the ordered data-item request method for deadlock avoidance, is the potential lower degree of concurrency. The advantage of these schemes is that there is no deadlock detection overhead. The disadvantage of the wait-die or the wound-wait deadlock avoidance schemes is that the request for a data-item held by another transaction does not necessarily imply a deadlock. Hence, the abort, required in either of these schemes, may be entirely unnecessary.

10.9 Atomicity, Concurrency, and Recovery

The atomic property of a transaction has to be preserved under concurrent execution. The atomicity requirement, as such, is an additional constraint to the serilizability requirement which we discussed earlier. Nevertheless, concurrency and failure of a transaction are both responsible for not preserving the atomicity requirements. These two requirements force the situation known as cascading rollback, as described earlier. Consider a case where a WRITE operation modifies the database, as in the update in place scheme (see Chapter 10). Following such write operations by a transaction and the subsequent unlocking of the data-items, the updated values are accessible to other concurrent transactions. However, the first transaction may have to be aborted and rolled back. This implies that all transactions that used any data-item written by a rolled back transaction or any other data-item derived from such a data-item, also have to be undone, resulting in cascading rollback.

The method of avoiding a cascading rollback is to prevent transactions from reading a data-item modified by an uncommitted transaction. One way of doing this is to extend all locks to the point of committing a transaction, though, this reduces concurrency. Another approach requires that all writes to the database are to a log and, hence, considered as being tentative. A transaction commits after it has done all its write operations. At the time of the commit, all tentative values are reflected in the database. Any transaction that needs a tentatively written data-item has to wait for the transaction to commit. Alternatively, if a transaction is allowed to use a tentative data value, then it is marked for rollback in case the transaction that wrote the value is aborted.

The locking scheme of concurrency control can be considered to require the following steps: lock, read, and/or write, unlock, commit. The timestamp scheme, on the other hand, requires three steps as follows: read, write, and commit. The optimistic scheduling, also has three steps, these being: read, validate, and write. The two later schemes are preferable if the expected number of contentions, and the resulting number of rollbacks, is relatively low.

10.10 Summary

Concurrent access to a database by a number of transactions requires some type of concurrency control to preserve the consistency of the database; to ensure that the modifications made by the transactions are not lost; and to guard against transaction reading data that is inconsistent. A number of concurrency control schemes have been discussed in this chapter.

Concurrent execution of transactions implies that the operations from these transactions may be interleaved. This is not the same as serial execution of the transactions where each transaction is run to completion before the next transaction is started. The serializability criterion is used to test whether an interleaved execution of the operations from a number of concurrent transactions is correct or not. The serializability test consists of generating a precedence graph from a interleaved execution schedule. If the precedence graph is acyclic, then the schedule is serializable, which means that the database will have the same state at the end of the schedule as some serial execution of the transactions.

The concurrency control scheme ensures that the schedule that can be produced by a set of concurrent transactions will be serializable. One of two approaches is usually used to ensure serializability: delaying one or more contending transactions, or aborting and restarting one or more of the contending transactions.

The locking protocol uses the former approach while the time-stamp based ordering, the optimistic scheduling, and the multi-version technique of concurrency control use the latter.

In the locking protocol, a transaction, before it can access a data-item, is required to lock the data-item in an appropriate mode. It releases the lock on the data-item once it no longer needs it. In the locking scheme, the two-phase locking protocol is usually used. The principle characteristics of the two-phase locking protocol is that all locks are acquired before a transaction starts releasing any locks. This ensures serializability, however, deadlock is possible.

With a hierarchically structured storage of the database and its data-items, a different granularity of locking is implied. Thus, locking an item may imply locking all items which are its descendents. To enhance the performance of a system with hierarchically structured data, additional modes of locking are introduced. Thus, in addition to read and write locks, intention locks are required. The locking protocol is modified to require a root-to-leaf direction of lock requests, and the reverse direction of lock releases.

In the timestamp based ordering, each transaction is assigned an unique identifier, which is usually based on the system clock. This identifier is called a timestamp and the value of the time- stamp is used to schedule contending transactions. The rule used is to ensure that a transaction with a smaller timestamp, and hence, older, is effectively executed before a younger transaction, and hence, with a larger timestamp. Any variation from this rule is corrected by aborting a transaction, rolling back any modifications made by it, and restarting it again.

In the optimistic scheduling, the philosophy used is that a contention between transactions will be very unlikely and any data-item used by a transaction is not likely be used for modification by any other transaction. This assumption is valid for transactions which will, only, read the data-item. If this assumption is found to be invalid for a given transaction, then the transaction is aborted and rolled back.

In the multi-version technique, data is never written over; rather, whenever the value of a data-item is modified, a new version of the data-item is created. The result is that the history of the evolution of a

data-item is maintained. A transaction, is assigned an unique timestamp and is directed to read the appropriate version of a data-item. The write operation of a transaction, such as T, could cause a new version of the data-item to be generated. However, in case another transaction has already produced a new version of the data-item based on the version used by the transaction, T, then an attempt to write a modified value for the data-item by transaction T causes transaction T to be aborted and rolled back, and restarted as a new and younger transaction.

Deadlock is a situation which arises when data-items are locked in different order by different transactions. A deadlock situation exists when there is a circular chain of transactions, each transaction in the chain waiting for a data-item already locked by the next transaction in the chain. Deadlock situations can be either avoided or detected and recovered from. One method of avoiding deadlock is to ask for all data-items at one time. An alternative is to assign a rank to each data-item and request locks for data-items in a given order. A third technique depends on selectively aborting some transactions and allowing others to wait. The selection is based on the timestamp of the contending transactions, and the decision as to which transaction to abort and which to allow to wait is determined according to the preemptive protocol being used. The wait-die and the wound-wait are two such preemptive protocols.

Deadlock detection depends on detecting the existence of a circular chain of transactions, and then, aborting or rolling back one transaction at a time until no further deadlocks are present. The wait-for-graph is generated periodically by the system to enable it to detect a deadlock.

Key Terms

acyclic precedence graph
concurrency
concurrency control
concurrent operations
cyclic precedence graph
deadlock
deadlock avoidance
deadlock detection and recovery
exclusive lock
hierarchy of locks
inconsistent reads problem
intention exclusive mode
intention mode locking
intention share mode
live-lock
lock manager
locking
locking principle
lost update problem
multi-version technique
multi-version techniques
non-serializable schedule
optimist scheme
optimistic scheduling
phantom phenomenon
precedence graph
read phase
read-before-write protocol
schedule
serial execution
serializability
serializable schedule
share and intention exclusive mode
shared lock
starvation
storage structure - DAG
timestamp based order
timestamp scheme
tree locking protocol
two-phase locking
two-phase locking contracting phase
two-phase locking granularity of locking
two-phase locking growing phase
validation phase
wait-Die
wait-for-graph
wound-wait
write phase

Exercises

10.1 Consider two transactions as follows:

Transaction 1 :$Fac\_Salary_i := 1.1 * FacSalary_i + 1025.00$

Transaction2: $Average_{¿} = \sum_{i=1}^{N} FacSalary / N$

What precaution if any would you suggest if these were to run concurrently? Write a pseudo-code program for these transactions using an appropriate scheme to avoid undesirable results.

10.2 Consider that the adjustment of salary of the faculty members is done as follows, where Fac_Salary_i represents the salary of the i^{th} faculty member;

Transaction 1: $Fac\_Salary_i := FacSalary_i + 1025$

Transaction 2: $Fac\_Salary_i := FacSalary_i * 1.1$

What precaution if any would you suggest if these were to run concurrently? Write a pseudo-code program for these transactions using an appropriate scheme to avoid undesirable results.

10.3 Consider the schedule of Figure 10.8(a). What is the value of A and B, if $f_1(A)$ is A+ 10, $f_2(B)$ is B * 1.2, $f_3(B)$ is B =20, and $f_4(A)$ is A * 1.2? Assume the initial values of A and B are 1000, and 200 respectively.

10.4 Repeat exercise 10.3 for the schedule of Figure 10.9(b).

10.5 Consider the transactions of Figure 10.21. Rewrite the transactions using the two-phase protocol, and produce a schedule which is serializable.

10.6 Write an algorithm to find a cycle in a precedence graph. (Hint: can you use an approach similar to algorithm 10.2?)

10.7 Consider the transactions of Figure 10.21 and the schedule of Figure 10.26. What would happen at step 5, if $t_{22} > t_{23}$? Complete the schedule after step 5 and give the values for A and B after each step. Assume that the initial values are A:{ 400, Wa, Ra } B:{ 500, Wb, Rb }.

10.8 Given the following schedule of Figure 10.29, in a system where the timestamp ordering is used, suppose the transaction, T_{22}, and, T_{23}, had been assigned the timestamps t_{22} and t_{23} respectively and *Sum* is a local variable. Any value read in from the database is copied into local variables with the same names as the corresponding database items. The database items are only changed with a **Write** statement. If initially A:{ 400, W_a, R_a } B:{ 500, W_b, R_b }, indicate their values after steps 3, 5,7,8,12 and 14.

```
Step  Schedule              Transaction T22       Transaction T23

1          Sum := 0               Sum := 0
2          Sum := 0                                    Sum := 0
3          Read (A)                                    Read (A)
4          A := A -100                                 A := A -100
5          Read (A)               Read (A)
6          Sum := Sum + A         Sum := Sum + A
7          Write (A)                                   Write (A)
```

```
8          Read (B)                Read (B)
9          Sum := Sum + B          Sum := Sum + B
10         Show( Sum)              Show( Sum)
11         Sum := Sum + A                                  Sum := Sum + A
12         Read (B)                                        Read (B)
13         B := B + 100                                    B := B + 100
14         Write (B)                                       Write (B)
15         Sum := Sum + B                                  Sum := Sum + B
16         Show( Sum)                                      Show( Sum)
```

Figure 10.29 *Schedule for Exercise 10.8*

Step	Schedule	Transaction T_{24}	Transaction T_{25}	Transaction T_{26}
1	**Read** *A*	**Read** *A*		
2	*A* := f_1(*A*)	*A* := f_1(*A*)		
3	**Read** *B*			**Read** *B*
4	**Write** *A*	**Write** *A*		
5	**Read** *C*		**Read** *C*	
6	*C* := f_2(*C*)		*C* := f_2(*C*)	
7	**Read** *C*			**Read** *C*
8	**Write** *C*		**Write** *C*	
9	**Read** *B*	**Read** *B*		
10	*B* := f_3(*B*)			*B* := f_3(*B*)
11	**Write** *B*			**Write** *B*
12	*B* := f_4(*B*)	*B* := f_4(*B*)		
13	**Write** *B*	**Write** *B*		

Figure 10.30 *Schedule for Exercise 10.9*

10.9 In this example, we have three transactions, T_{24}, T_{25}, and T_{26} with timestamp values of t_{24}, t_{25},and t_{26}, respectively ($t_{24} < t_{25} < t_{26}$). The schedule for the concurrent execution of these transactions is given in Figure 10.30. Assuming that initially *A*: a, W_a,R_a, *B*: b, W_b, R_b, and *C*: c, W_c, R_c, show these values after each step ,if the timestamp ordering scheme for concurrency control is used.

10.10 Suppose we want to add a record occurrence to the record type R_1, which uses indices I_{11} and I_{12} for direct access to the records. Give the sequence of locking to perform this operation.

10.11 The algorithm 10.2 is inefficient, since some transactions are processed many times. Give a modification to the algorithm to avoid this inefficiency.

10.12 In an adaptive deadlock detection scheme, why is it necessary to choose an upper and lower limit for the frequency of running the deadlock detection algorithm?

10.13 In the concurrency control scheme based on timestamp ordering, we have assumed that the timestamp value is based on a system wide clock. Instead of using such a timestamp to determine the ordering, suppose a pseudo-random number generator was used. Show how you will modify the concept of older and younger transactions with this modification, and, hence give the modified wait-die and wound-wait protocols.

Bibliographic Notes

Gray in [Gray78] presents comprehensive operating system requirements for a database system. The transaction concept and its limitations are discussed in [Gray81].The serializability concept, the two phase locking protocol and its correctness is due to the early work by Eswaran et al [Eswa79] in connection with System R. The extension of the serializability test for read-only and write only cases are discussed in [Papa79]. The algorithm for this case is developed in [Bern79], and the text by Ullman also treats this topic. Locking scheme, multi-granularity, and, intention locking extensions are discussed in [Gray75]. Extensions to lock modes and deadlock avoidance are discussed in [Kort82] and in [Kort83].

[Reed79] presented the earliest known multi-version time-stamping algorithm. The use of a pseudo-timestamp was discussed in [Reed83] and [Svob80]. It is shown in [Bern83] that any schedule generated according to the timestamp concurrency control algorithm requirements is serializable, and the result obtained by a set of concurrent transactions is the same as obtained by some serial execution of the set of transactions with a single version of the data-items. The reader, interested in the multi-version concurrency control algorithms based on locking, is referred to [Baye80] and [Ster81]: the extension of the locking scheme and locking with timestamp ordering (combination scheme) is discussed in[Bern83]. The combination scheme was discussed in [Chan82]. The tree locking protocol, for a database whose storage is tree structured, is discussed in [Silb80] and this protocol is generalized to the read-only and write-only locks in [Kade80]. [Bern80] presents a number of different distributed database concurrency control schemes based on time-stamping.

An optimistic method for concurrency control is presented in [Kung81]. [Rose79] proposed the wait-die and wound-wait transaction re-try schemes to avoid deadlocks in a distributed database system, although, these schemes are applicable to a centralized database system as well.

The deadlock problem is surveyed in [Coff71] and [Holt72]. [Islo80] discusses the general deadlock problem and examines the problems unique to database systems, both centralized and distributed.

Bibliography

[Bass88] Bassiouni, M. A., "Single-Site and Distributed Optimistic Protocols for Concurrency Control", IEEE-SE Vol SE-14, No. 8, August 1988, pp1071-1080.
[Baye80] Bayer, H, Heller. H, Reiser, A, "Parallelism and Recovery in Database Systems", ACM TODS, Vol5-4, June 1980, pp139-156.
[Bern79] Bernstein, P. A., Shipman, D. W., Wong, W. S., "Formal Aspects of Serializability in Database Concurrency Control", IEEE-SE Vol SE-5 No. 3, May 1979, pp203-215.
[Bern80] Bernstein, P. A., Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Systems", Proc. 6th International Conf. on Very Large Data Bases, Montreal, October 1980, pp285-300.
[Bern83] Bernstein, P. A., Goodman, N., "Multiversion Concurrency Control- Theory and Algorithms", ACM TODS, Vol8-4, Dec. 1983, pp465-483.
[Caso81] Casonova, M.A., "The Concurrency Control Problem of Database Systems", Lecture Notes in Computer Science, vol. 116, Springer-Verlag, New York, 1981.
[Chan82] Chan, A, Fox. S., Lin, W.T.K., Nori, A., Ries, D. R., "The implementation of an Integrated Concurrency Control and Recovery Scheme", Proc. ACM/SIGMOD Conf. on Management of Data, Orlando, Florida, June 1982, pp184-191.
[Coff71] Coffman, E. G., Elphick, M.J., Shoshani, A. "System Deadlocks" ACM Computing Surveys,

Vol3-2, June 1971, pp67-88.
[Eswa79] Eswaran, K.P., Gray, J.N., Lorie, R. A., Traiger, I. L., "The Notion of Consistency and Predicate Locks in a Database System", CACM, Vol. 19-11, November 1979, pp624-633.
[Gray75] Gray, J. N., Lorie, R.A., Putzolu, G.R., "Granularity of Locks in a Shared Data Base", Proc. of the VLDB, 1975, pp.428-451.
[Gray79] Gray, J. N., "Notes on Data Base Operating Systems", in 'Operating Systems: An Advanced Course', Ed. R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, Berlin, 1979.
[Gray81] Gray, J. N., " The transaction Concept:Virtues and Limitations", Proc of the 7th VLDB Conference, 1981, pp144-154.
[Holt72] Holt, R. C., "Some Deadlock Properties of Computer Systems" ACM Computing Surveys, Vol4-3, September 1972, pp179-196.
[Hunt79] Hunt, H. B., Rosenkrantz, D. J., "The Complexity of Testing Predicate Locks", Proc. ACM-SIGMOD 1979 International Conference on Management of Data, May 1979, pp127-133.
[Islo80] Isloor, S. S., Marsland, T. A., "The Deadlock Problem: An Overview", Computer, Vol13-9, September, 1980, pp58-78.
[Kade80] Kadem, Z., Silberschatz, "Non-two Phase Locking Protocols with Shared and Exclusive Locks", Proc. 6th International Conf. on Very Large Data Bases, Montreal, October 1980, pp309-320.
[Kort82] Korth, H. F., "Deadlock Freedom Using Edge Locks", ACM TODS, Vol. 7-4, December 1982, pp632-652.
[Kort83] Korth, H. F., "Locking Primitives in a Database System", ACM JACM, Vol. 30-1, January 1983, pp55-79.
[Kung79] Kung, H.T., Papadimitriou, C. H., "An Optimality Theory of Concurrency Control for Databases", Proc. ACM-SIGMOD 1979 International Conference on Management of Data, May 1979, pp116-126.
[Kung81] Kung, H.T., Robinson, J. T., ""On Optimistic Methods for Concurrency Control", ACM Trans. on Database Systems, Vol 6, No. 2, June 1981, pp 213-226.
[Lync83] Lynch, Nancy A., "Multilevel Atomicity- A New Correctness Criterion for Database Concurrency Control", ACM TODS, Vol8-4, September 1983, pp484-502.
[Papa79] Papadimitriou, C. H., "The Serializability of Concurrent Database Updates" JACM, Vol26-4, October 1979, pp150-157.
[Reed79] Reed, D. P., "Naming and Synchronization in a Decentralized Computer System', MIT/LCS/TR-205, MIT Cambridge, Mass, September 1979
[Reed83] Reed, D. P., "Implementing Atomic Actions on Decentralized Data", ACM Transaction on Computer Systems, Vol1-1, pp3-23.
[Rose79] Rosenkrantz, D. J., Stearns, R. E., Lewis II, P. M., "System Level Concurrency Control For Distributed Data Base Systems", ACM TODS, Vol.3-2, March 1978, pp.178-198.
[Silb80] Silberschatz, A., Kadem, Z., "Consistency in Hierarchical Database Systems", JACM,Vol27-1, January 1980pp72-80.
[Ster81] Stern, R.E., Rosenkrantz, D. J., "Distributed Database Concurrency Controls using Before-Values", Proc. ACM/SIGMOD Conf. on Management of Data, 1981, pp74-83.
[Svob80] Svobodova Liba, "Management of Object Histories in the Swallow Repository", MIT/LCS/TR-243, MIT, Cambridge Mass, July, 1980.
[Ullm82] Ullman, J.D., 'Principles of Database Systems", Computer Science Press, 1982, Rockville, Md.

Notes