CONCURRENCY CONTROL

13.1 Introduction
13.2 Locks
13.3 Optimistic concurrency control
13.4 Timestamp ordering
13.5 Comparison of methods for concurrency control
13.6 Summary

This chapter concentrates on concurrency control for servers whose operations may be modelled in terms of Read and Write operations on the data items.

All of the concurrency control protocols are based on the criterion of serial equivalence and are derived from rules for conflicts between operations. Three methods are described:

- Locks are used to order transactions that access the same data items according to the order of arrival of their operations at the data items.
- Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations on data items.
- Timestamp ordering uses timestamps to order transactions that access the same data items according to their starting times.
13.1 Introduction

In general, a server executes operations on behalf of several clients whose requests are not interleaved. Atomic transactions allow clients to specify atomic sequences of operations. Transactions must be scheduled so that their effect on shared data is serial equivalent. A server can achieve serial equivalence of transactions by serializing access to the data items. This chapter discusses methods of concurrency control for transactions whose operations are addressed to a single server. Chapter 14 discusses how these methods are extended for use with transactions whose operations are addressed to several servers. Figure 12.6 shows an example of how serial equivalence can be achieved with some degree of concurrency—transactions T and U both access account B, but T completes its access before U starts accessing it.

A simple example of a serializing mechanism is the use of exclusive locks. In a strict locking scheme the server attempts to lock any data item that is about to be used by a transaction operation of a client's transaction. If a client requests access to an item that is already locked due to another client's transaction, the request is suspended and the client must wait until the item is unlocked.

Figure 13.1 illustrates the use of exclusive locks. It shows the same transactions as Figure 12.6, but with an extra column for each transaction showing the locks held, waiting, and unlocked. In this example, it is assumed that, when transactions T and U start, the data items holding the balances of the accounts A, B, and C are not yet locked.

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank$Withdraw(A, 4)</td>
<td>Bank$Withdraw(C, 3)</td>
</tr>
<tr>
<td>Bank$Deposit(B, 4)</td>
<td>Bank$Deposit(B, 3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenTransaction</td>
<td>A.Read()</td>
<td>OpenTransaction</td>
<td>C.Read()</td>
</tr>
<tr>
<td>balance := A.Read()</td>
<td>locks A</td>
<td>balance := C.Read()</td>
<td>locks C</td>
</tr>
<tr>
<td>A.Write(balance - 4)</td>
<td></td>
<td>C.Write(balance - 3)</td>
<td></td>
</tr>
<tr>
<td>balance := B.Read()</td>
<td>locks B</td>
<td>balance := B.Read()</td>
<td>waits for T's lock on B</td>
</tr>
<tr>
<td>B.Write(balance + 4)</td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>CloseTransaction</td>
<td>unlocks A, B</td>
<td>B.Write( balance + 3)</td>
<td>locks B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CloseTransaction</td>
<td>unlocks B, C</td>
</tr>
</tbody>
</table>

When transaction T is about to read account B, the server locks it for T. Subsequently, when transaction U is about to read B it is still locked for T, and transaction U waits. When transaction T is committed, B is unlocked whereupon transaction U is resumed. The use of the lock on B effectively serializes the access to B. Note that if for example, T had released the lock on B between its Read and Write operations, transaction U's Read operation on B could be interleaved between them.

Serial equivalence requires that all of a transaction's accesses to a particular data item be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock. The first phase of each transaction is a 'growing phase' during which new locks are acquired. In the second phase the locks are released (a 'shrinking phase'). This is called two-phase locking.

We saw in Section 12.3 that because transactions may abort, strict executions are needed to prevent dirty reads and premature writes. Under a strict execution regime, a transaction that needs to read or write a data item must be delayed until other transactions that wrote the same data item have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called strict two-phase locking. The presence of the locks prevents other transactions from reading or writing the data items. When a transaction commits, to ensure recoverability, the locks must be held until all the data items it updated have been written to permanent storage.

A server generally contains a large number of data items and a typical transaction accesses only a few of them and is unlikely to clash with other current transactions. The granularity with which concurrency control can be applied to data items is an important issue since the scope for concurrent access may not be limited severely if concurrency control (for example, locks) can only be applied to all the data items at once. In our banking example, if locks are applied to all customer accounts at a branch, only one bank clerk could perform an on-line banking transaction at any time – hardly an acceptable constraint!

The portion of the data items to which access must be serialized should be as small as possible, that is, just that part involved in each operation requested by transactions. In our banking example, a branch holds a set of accounts, each of which has a balance. Each banking operation affects one or more account balances – Deposit and Withdraw affect one account balance and BranchTotal affects all of them.

The description of concurrency control schemes given below does not assume any particular granularity.

We discuss concurrency control protocols that are applicable to servers whose operations can be modelled in terms of Read and Write operations on the data items. For the protocols to work correctly, it is essential that each Read and Write operation is atomic in its effects on data items. Concurrency control protocols are designed to cope with conflicts between operations in different transactions on the same data item. In this chapter, we use the notion of conflict between operations to explain the protocols. When we say that a pair of operations conflicts we mean that their combined effect depends on the order in which they are executed. The effect of an operation refers to the value of a data item set by a


13.2 Locks

The operation conflict rules in Figure 13.2 show that pairs of Read operations from different transactions on the same data item do not conflict. Therefore, a simple exclusive lock that is used for both Read and Write operations reduces concurrency more than is necessary.

It is preferable to adopt a locking scheme that controls the access to each data item so that there can be several concurrent transactions reading a data item, or a single transaction writing a data item, but not both. This is commonly referred to as a ‘many reader/single writer’ scheme. Two types of locks are used: read locks and write locks.

Before a transaction’s Read operation is performed, the server attempts to set a read lock on the data item. Before a transaction’s Write operation is performed, the server attempts to set a write lock on the data item. Whenever a server is unable to set a lock immediately it keeps the transaction (and the client) waiting until it is able to do so – it never rejects a client’s request.

As pairs of Read operations from different transactions do not conflict, an attempt to set a read lock on the data item with a read lock is always successful. All the transactions reading the same data item share its read lock – for this reason, read locks are sometimes called shared locks.

The operation conflict rules tell us that:

1. If a transaction T has already performed a Read operation on a particular data item, then a concurrent transaction U must not Write that data item until T commits or aborts.

2. If a transaction T has already performed a Write operation on a particular data item, then a concurrent transaction U must not Read or Write that data item until T commits or aborts.

To enforce (1) a request for a write lock on a data item is delayed by the presence of a read lock belonging to another transaction. To enforce (2) a request for either a read lock or a write lock on a data item is delayed by the presence of a write lock belonging to another transaction.

The Figure 13.3 shows the compatibility of read locks and write locks on any particular data item. The entries in the first column of the table show the type of lock already set – if any. The entries in the first row show the type of lock requested. The entry in each cell shows the effect on a transaction that requests the type of lock given above when the data item already has the type of lock on the left.

### Lock compatibility

<table>
<thead>
<tr>
<th>For one data item</th>
<th>Lock requested</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td>None</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>OK</td>
<td>Wait</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>Wait</td>
<td>Wait</td>
</tr>
</tbody>
</table>
Use of locks in strict two-phase locking.

1. When an operation accesses a data item within a transaction:
   a) If the data item is not already locked, the server locks it and the operation proceeds.
   b) If the data item has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   c) If the data item has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   d) If the data item has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all data items it locked for the transaction.

Inconsistent retrievals and lost updates are caused by conflicts between Read operations in one transaction and Write operations in another. Inconsistent retrievals are prevented by performing the retrieval transaction before or after the update transaction. If the retrieval transaction comes first, its read locks delay the update transaction. If it comes second, its request for read locks causes it to be delayed until the update transaction has completed.

Lost updates occur when two transactions read a value of a data item and then use it to calculate a new value. Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed. This is achieved by each transaction setting a read lock when it reads a data item and then promoting it to a write lock when it writes the same data item — when a subsequent transaction requires a read lock it will be delayed until any current transaction has completed.

A transaction with a read lock that is shared with other transactions cannot promote its read lock to a write lock because the latter would conflict with the read lock held by the other transactions. Therefore such a transaction must request a write lock and wait for the other read locks to be released.

Lock promotion refers to the conversion of a lock to a stronger lock — that is a lock that is more exclusive. The lock compatibility table shows which locks are more or less exclusive. The read lock allows other read locks, whereas the write lock does not. Neither allow other write locks. Therefore a write lock is more exclusive than a read lock. Locks may be promoted because the result is a more exclusive lock. It is not safe to demote a lock held by a transaction before it commits because the result will be more permissive than the previous one and may allow executions by other transactions that are inconsistent with serial equivalence.

The rules for the use of locks in a strict two-phase locking implementation are summarized in Figure 13.4. To ensure that these rules are adhered to, the client has access to operations for locking or unlocking items of data. Locking is performed by the server when the Read and Write operations are requested and unlocking by the Commit or Abort operations of transactional service.

The rules given in Figure 13.4 ensure strictness because the locks are held until a transaction has either committed or aborted. However, it is not necessary to hold read locks to ensure strictness. Read locks must be held until the request to commit or abort to ensure serial equivalence by using two-phase locking.

**Lock implementation** □ The granting of locks will be implemented by a separate module of the server program that we call the lock manager. We discuss the design of a lock manager for use with transactions. A lock manager is responsible for maintaining a table of locks for the data items of a server. Each entry in the table of locks includes:
- the transaction identifiers of the transactions that hold the lock (shared locks can have several holders);
- an identifier for a data item;
- a lock type;
- a condition variable.

The identifier for the data item must be something available to the Read or Write operation, generally an argument. For example, in our banking example, the identifier would be the name of an account.

In Chapter 12, it is argued that when a client needs to wait to access a shared resource, it is better for the server to suspend the client's request with a Wait operation than to tell the client to try again later. To make this possible, each client request runs in a separate server thread. When a lock cannot be granted, the thread running the request
**Figure 13.6** Deadlock with read and write locks.

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Locks</th>
<th>Transaction U</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td></td>
<td>Operations</td>
<td></td>
</tr>
<tr>
<td>balance := A.Read()</td>
<td>read locks A</td>
<td>balance := C.Read()</td>
<td>read locks C</td>
</tr>
<tr>
<td>A.Write(balance - 4)</td>
<td>write locks A</td>
<td>C.Write(balance - 3)</td>
<td>write locks C</td>
</tr>
<tr>
<td></td>
<td>***</td>
<td>balance := B.Read()</td>
<td>shares read lock on B</td>
</tr>
<tr>
<td>balance := B.Read()</td>
<td>read locks B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.Write(balance + 4)</td>
<td>waits for U</td>
<td>B.Write(balance + 3)</td>
<td>waits for T</td>
</tr>
<tr>
<td></td>
<td>***</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Waits* on the condition variable and when the server unlocks a data item, the condition variable is *Signalled*.

The lock manager provides the operations *Lock* for requesting locks and *UnLock* for releasing them (as shown in Figure 13.5). The operations on the table of locks must be atomic. The *Lock* operation uses the lock type given as argument and the lock type of entries for the same data item (belonging to other transactions), together with the lock compatibility table to decide whether there is a conflicting entry.

Note that, when several threads *wait* on the same locked item, the semantics of *wait* ensure that each transaction gets its turn. When the queue of waiting threads is headed by several requests for shared locks it is an optimization to allow them all to proceed by signalling more than once.

**Deadlocks**

The use of locks can lead to deadlock. Consider the use of locks shown in Figure 13.3. This differs from Figure 13.1 in that both transactions now read the balance of account B and share the read lock, but when T wants to write the balance of B it must wait until the other transaction unlocks it. Similarly when U wants to write to the balance of B it must wait until T unlocks the item. This is a deadlock situation – two transactions waiting and each is dependent on the other to release a lock so it can resume.

The *Deposit* and *Withdraw* operations in our banking service example can easily produce a deadlock. The reason for this is that each operation first requests a read lock on an account and then attempts to promote it to a write lock. Deadlock would be likely to arise in the banking service if these operations were to request write locks initially.

**Figure 13.7** An illustration of Violet showing the union of some diaries.

<table>
<thead>
<tr>
<th>January 1988</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 Monday</td>
</tr>
<tr>
<td>9:00-10:00</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>unavailable</td>
</tr>
<tr>
<td>13:00-14:00</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Smith</td>
</tr>
<tr>
<td>unavailable</td>
</tr>
</tbody>
</table>

*View: [Smith.qmw, Jones.qmw]*

<table>
<thead>
<tr>
<th>January 1988</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 Monday</td>
</tr>
<tr>
<td>10:00-12:00</td>
</tr>
<tr>
<td>硬件研究</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Deadlock is a particularly common situation when clients are involved in an interactive program, for a transaction in an interactive program may last for a long period of time, resulting in many data items being locked and remaining so, thus preventing other clients from using them. An interesting example of such a program is the Violet system described by Gifford [1979b] and Lampson [1981b]. The Violet system illustrated in Figure 13.7 provides a calendar or diary database that can be viewed and updated interactively by a large number of users. It allows users to view pages from other people's diaries before arranging meetings. A user who wants to make an appointment with someone views that person's diary and then writes in an appointment. Another user may have the same diary at the same time and also view a copy of the diary and subsequently attempt to add an appointment for the same date. We can regard one day in a diary as an item of data and viewing it will result in a read lock on it. Adding an entry will require altering the read lock to a write lock. It is permissible for two users to view the same object at the same time and this is implemented by sharing the read lock on the item. However, neither user will be able to write an appointment, as it is not permissible for either client to convert the read lock to a write lock in the presence of the other transaction's shared read lock on the same item.

Note that the locking of sub-items in structured data items can be useful. For example, a day in a diary could be structured as a set of time slots, each of which can be locked independently for updating. On the other hand, the view shown in Figure 13.7
show a wait-for graph for Figure 13.6.

**Deadlock**

Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock. A wait-for graph can be used to represent the waiting relationships between current transactions at a server. In a wait-for graph, the nodes represent transactions and the edges represent wait-for relationships between transactions – there is an edge from node T to node U when transaction T is waiting for transaction U to release a lock. See Figure 13.8 which illustrates the wait-for graph corresponding to the deadlock situation illustrated in Figure 13.6. Recall that the deadlock arose because transactions T and U both requested write locks on data item B when they already shared a read lock on B. Therefore T waits for U and U waits for T. The dependency between transactions is indirect – via a dependency on data items. The diagram on the right shows the data items held by and waited for by transactions T and U. As each transaction can wait for only one data item, the data items can be omitted from the wait-for graph – leaving the simple graph on the left.

Suppose that as in Figure 13.9, a wait-for graph contains a cycle T → U → V → T, then each transaction is waiting for the next transaction in the cycle. All of these transactions are blocked waiting for locks. None of the locks can ever be released and the transactions are deadlocked. If one of the transactions in a cycle is aborted, then its locks are released and the cycle is broken. For example, if transaction T in Figure 13.6 is aborted, it will release a lock on a data item that V is waiting for – and V will no longer be waiting for T.

Now consider a scenario in which the three transactions T, U, and V share a read lock on a data item C, transaction W holds a write lock on data item B on which transaction V is waiting to obtain a lock, as shown on the right in Figure 13.10. The transactions T and W then request write locks on data item C and a deadlock situation arises in which T waits for U and V, V waits for W and W waits for T, U, and V, as shown on the left in Figure 13.10. This shows that although each transaction can wait for only one data item at a time, it may be involved in several cycles. For example, transaction V is involved in cycles: V → W → T → V and V → T → V.

In this example, suppose that transaction V is aborted, this will release V’s lock on C and the two cycles involving V will be broken.

**Deadlock prevention**

One solution is to prevent deadlock. An apparently simple, but not very good way to overcome deadlock is to lock all of the data items used by a transaction when it starts. Such a transaction cannot run into deadlock with other transactions, but it unnecessarily restricts access to shared resources. In addition it is sometimes impossible to predict at the start of a transaction which data items will be used. This is generally the case in interactive applications, for example, in the Violet system the user would have to say in advance which weeks of which diaries would be viewed and which days would be updated. Deadlock can also be prevented by requesting locks on data items in a predefined order, but this can result in premature locking and a reduction in concurrency.

**Deadlock detection**

Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, the server must select a transaction to abort, so as to break the cycle.

The software responsible for deadlock detection can be part of the lock manager. It must hold a representation of the wait-for graph so that it can check it for cycles from time to time. Edges are added to the graph and removed from the graph by the lock manager’s Lock and Unlock operations. At the point illustrated by Figure 13.10 it will have the following information:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Waits for transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>U, V</td>
</tr>
<tr>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>W</td>
<td>T, U, V</td>
</tr>
</tbody>
</table>

An edge T → U is added whenever the lock manager blocks a request by transaction T for a lock on a data item that is already locked on behalf of transaction U. Note that when
a lock is shared, several edges may be added. An edge \( T \rightarrow U \) is deleted whenever \( T \) releases a lock that \( T \) is waiting for and allows \( T \) to proceed. See Exercise 13.7 for a more detailed discussion of the implementation of deadlock detection. If a transaction shares a lock, the lock is not released but the edges leading to a particular transaction are removed.

The presence of cycles may be checked each time an edge is added, or less frequently to avoid server overhead. When a deadlock is detected, one of the transactions in the cycle must be chosen and then be aborted. The corresponding node and the edges involving it must be removed from the wait-for graph. This will happen when the aborted transaction has its locks removed.

The choice of the transaction to abort is not simple. Some factors that may be taken into account are the age of the transaction and the number of cycles it is involved in.

**Timeouts**

Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After this time, the lock becomes vulnerable. Provided that no other transaction is competing for the item that is locked, an item with a vulnerable lock remains locked. However, if any other transaction is waiting to access the data item protected by a vulnerable lock, the lock is broken (that is, the data item is unlocked) and the waiting transaction resumes. The transaction whose lock has been broken is normally aborted.

There are many problems with the use of timeouts as a remedy for deadlocks: the worst problem is that transactions are sometimes aborted due to their locks becoming vulnerable when other transactions are waiting for them, but there is actually no deadlock. In an overloaded system, the number of transactions timing out will increase and transactions taking a long time can be penalized. In addition, it is hard to decide on an appropriate length for a timeout. In contrast, if deadlock detection is used, transactions are aborted because deadlocks have occurred and servers can make a choice as to which transaction to abort.

It is possible that the correctness of a transaction does not depend on the value of the data item whose lock is broken and, in that case, it may not be necessary to abort the transaction when a vulnerable lock’s broken. In the XDFS file server [Israel et al., 1992], the client is notified when a read lock is broken and may voluntarily unlock the item it protects, in which case the transaction may continue and complete successfully. If the client has not unlocked the item after the lock is broken, a commit would fail.

Using lock timeouts, we can resolve the deadlock in Figure 13.6 as shown in Figure 13.11 in which the read lock for \( T \) on \( B \) becomes vulnerable after its timeout period. Transaction \( U \) is waiting to alter it to a write lock. Therefore \( T \) is aborted and releases its share of the read lock on \( B \), allowing \( U \) to resume, convert its read lock to a write lock and complete the transaction.

**Increasing concurrency in locking schemes**

Even when locking rules are based on the conflicts between Read and Write operations and the granularity at which they are applied is as small as possible, there is still scope for increasing concurrency. We shall discuss two approaches that have been used.

---

**Figure 13.11 Resolution of the deadlock in Figure 13.6.**

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction U</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance := A.Read( )</td>
<td>read locks A</td>
<td></td>
<td>balance := C.Read( )</td>
<td>read locks C</td>
<td></td>
</tr>
<tr>
<td>A.Write(balance – 4)</td>
<td>write locks A</td>
<td></td>
<td>C.Write(balance – 3)</td>
<td>write locks C</td>
<td></td>
</tr>
<tr>
<td>***</td>
<td></td>
<td></td>
<td>***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>balance := B.Read( )</td>
<td>read locks B</td>
<td></td>
<td>balance := B.Read( )</td>
<td>shares read lock on B</td>
<td></td>
</tr>
<tr>
<td>***</td>
<td></td>
<td></td>
<td>***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.Write(balance + 4)</td>
<td>waits on U’s read lock on B</td>
<td></td>
<td>B.Write(balance + 3)</td>
<td>waits on T’s read lock on B</td>
<td></td>
</tr>
<tr>
<td>***</td>
<td></td>
<td></td>
<td>***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(timeout elapses)</td>
<td>T’s lock on B becomes vulnerable, unlock B, abort T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.Write(balance + 3)</td>
<td>write locks B</td>
<td></td>
<td></td>
<td>unlock B and C</td>
<td></td>
</tr>
</tbody>
</table>

In the first approach (two-version locking), the setting of exclusive locks is delayed until a transaction commits. In the second approach (hierarchic locks), mixed granularity locks are used.

**Two-version locking**

This is an optimistic scheme that allows one transaction to write tentative versions of data items while other transactions read from the committed version of the same data items. Read operations only wait if another transaction is currently commiting the same data item. This scheme allows more concurrency than read-write locking but writing transactions risk waiting or even rejection when they attempt to commit. Transactions cannot commit their write operations immediately if other uncommitted transactions have read the same data items. Therefore transactions that request to commit in such a situation are made to wait until the reading transactions have completed. Deadlock may occur when transactions are waiting to commit. Therefore transactions may need to be aborted when they are waiting to commit, to resolve deadlocks.

This variation on strict two-phase locking was proposed by Gifford for Fourier and implemented in the XDFS file server. It uses three types of locks: a read lock, a write lock and a commit lock. Before a transaction’s Read operation is performed, the server attempts to set a read lock on the data item – the attempt to set a read lock is successful unless the data item has a commit lock, in which case the transaction waits. Before a
Figure 13.12 Lock compatibility (read, write and commit locks).

<table>
<thead>
<tr>
<th>For one data item</th>
<th>Lock to be set</th>
<th>Read</th>
<th>Write</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>OK</td>
<td>OK</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>OK</td>
<td>Wait</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>Wait</td>
<td>Wait</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

When the server receives a request to commit a transaction, it attempts to convert all that transaction’s write locks to commit locks. If any of the data items have outstanding read locks, the transaction must wait until the transactions that set those locks have completed and the locks are released. The compatibility of the read, write and commit locks is shown in Figure 13.12.

There are two main differences in performance between the two-version locking scheme and an ordinary read-write locking scheme. On the one hand, Read operations in the two-version locking scheme are delayed only during the commitment of transactions rather than during the entire execution of transactions. In most cases, commitment takes only a small fraction of the time required to perform an entire transaction. On the other hand, Read operations of one transaction can cause delay in committing other transactions.

Hierarchic locks

In some servers, the granularity suitable for one operation is not appropriate for another operation. In our banking example, the majority of operations require locking at the granularity of an account. The BranchTotal operation is different— it reads the values of all the account balances and would appear to require a read lock on all of them. To reduce locking overhead it would be useful to allow locks of mixed granularity to coexist.

Gray [1978] proposed the use of a hierarchy of locks with different granularities. At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks. This economizes on the number of locks to be set. In our banking example, the branch is the parent and the accounts are children (see Figure 13.13).

Figure 13.14 Lock hierarchy for Violet.

We noted earlier that mixed granularity locks could be useful in the Violet system in which the data could be structured with the diary for a week being composed of a page for each day and the latter subdivided further into a slot for each time of day as shown in Figure 13.14. The operation to view a week would cause a read lock to be set at the top of this hierarchy whereas the operation to enter an appointment would cause a write lock to be set on a time slot. The effect of a read lock on a week would be to prevent write operations on any of the substructures, for example, the time slots for each day in that week.

In Gray’s scheme, each node in the hierarchy can be locked — giving the owner of the lock explicit access to the node and implicit access to its children. In our example, a read/write lock on the branch implicitly read/write locks all the accounts. Before a child node is granted a read/write lock, an intention to read/write lock is set on the parent node and its ancestors (if any). The intention lock is compatible with other intention locks but conflicts with read and write locks according to the usual rules. Figure 13.15 gives the compatibility table for hierarchic locks. Gray also proposed a third type of intention lock — that combines the property of a read lock with an intention to write lock.

In our banking example the BranchTotal operation requests a read lock on the branch which implicitly sets read locks on all the accounts. A Deposit operation needs to set a write lock on a balance, but first it attempts to set an intention to write lock on the branch. These rules prevent these operations from running concurrently.

Hierarchic locks have the advantage of reducing the number of locks when mixed granularity locking is required. The compatibility tables and the rules for promoting locks are more complex. The mixed granularity of locks can increase concurrency when many short transactions are combined with other transactions that take a long time.

Figure 13.15 Lock compatibility table for hierarchic locks.

<table>
<thead>
<tr>
<th>For one data item</th>
<th>Lock to be set</th>
<th>Read</th>
<th>Write</th>
<th>I-Read</th>
<th>I-Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>OK</td>
<td>Wait</td>
<td>OK</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Wait</td>
<td>Wait</td>
<td>Wait</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>I-Read</td>
<td>OK</td>
<td>Wait</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>I-Write</td>
<td>Wait</td>
<td>Wait</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
</tbody>
</table>
13.3 Optimistic concurrency control

Kung and Robinson [1981] identified a number of inherent disadvantages of locking and proposed an alternative optimistic approach to the serialization of transactions. This approach avoids these drawbacks. We can summarize the drawbacks of locking:

- Lock maintenance represents an overhead that is not present in systems that do support concurrent access to shared data. Even read-only transactions (queries) cannot possibly affect the integrity of the data, must, in general, use locks in order to guarantee that the data being read is not modified by other transactions at the same time. But locking may be necessary only in the worst case.

For example, consider two client processes that are concurrently incrementing the values of n data items. If the client programs start at the same time, and run about the same amount of time, accessing the data items in two unrelated sequences and using a separate transaction to access and increment each item, chances are that the two programs will attempt to access the same data item at the same time are just one in n on average, so locking is really needed only once every n transactions.

- The use of locks can result in deadlock. Deadlock prevention reduces concurrency severely and therefore deadlock situations must be resolved either by the use of timeouts or by deadlock detection. Neither of these is wholly satisfactory for use in interactive programs.

- To avoid cascading aborts, locks cannot be released until the end of a transaction. This may reduce significantly the potential for concurrency.

The alternative approach proposed by Kung and Robinson is 'optimistic' because it is based on the observation that, in most applications, the likelihood of two client transactions accessing the same data is low. Transactions are allowed to proceed, though there were no possibility of conflict with other transactions until the client completes its task and issues a CloseTransaction request. When a conflict arises, the transaction is generally aborted and will need to be restarted by the client.

Each transaction has the following phases:

**Read phase:** During the read phase, each transaction has a tentative version of each of the data items that it updates. The use of tentative versions allows a transaction to abort (with no effect on the data items), either during the read phase or if it fails validation due to other conflicting transactions. Read operations are performed immediately - if a tentative version for that transaction already exists, a Read operation accesses it, otherwise it accesses the most recently committed value of the data item. Write operations record the new values of the data items.

**Validation of transactions** - Validation uses the Read/Write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other overlapping transactions - that is any transactions that had not yet committed at the time this transaction started. To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase (that is, when the client issues a CloseTransaction). If the transaction is validated and completes successfully it retains this number; if it fails the validation checks and is aborted, or if the transaction is read only, the number is released for re-assignment. Transaction numbers are integers assigned in ascending sequence; the number of a transaction therefore defines its position in time - a transaction always finishes its read phase after all transactions with lower numbers, that is, a transaction with the number Tj always precedes a transaction with the number Ti if i < j. (If the transaction number were to be assigned at the beginning of the read phase, then a transaction that reached the end of the read phase before one with a lower number would have to wait until the earlier one had completed before it could be validated.)

The validation test is based on conflicts between operations in pairs of transaction Ti and Tj. For a transaction Tj to be serializable with respect to an overlapping transaction Ti, their operations must conform to the following rules:

<table>
<thead>
<tr>
<th>Ti</th>
<th>Tj</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>Ti must not read data items written by Tj</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>Tj must not read data items written by Ti</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>Ti must not read data items written by Tj and Tj must not read data items written by Ti</td>
</tr>
</tbody>
</table>

As the validation and write phases of a transaction are generally short in duration compared with the read phase, a simplification may be achieved by making the rule that only one transaction may be in the validation and write phase at one time. When no two transactions may overlap in the write phase, Rule 3 is satisfied. Note that this restriction on Write operations, together with the fact that no dirty reads may occur, produces strict executions. To prevent overlapping, the entire validation and write phases can be implemented as a critical section so that only one client at a time can execute it. In order
Figure 13.16 Validation of transactions.

To increase concurrency, part of the validation and writing may be implemented outside the critical section, but it is essential that the assignment of transaction numbers be performed sequentially. We note that at any instant, the current transaction number acts like a pseudo-clock that ticks whenever a transaction completes successfully.

The validation of a transaction must ensure that the Rules 1 and 2 are obeyed, i.e., testing for overlaps between the data items of pairs of transactions $T_i$ and $T_j$. There are two forms of validation — backward and forward [Härder 1984]. Backward validation checks the transaction undergoing validation with other previously overlapping transactions — those that entered the validation phase after it. Forward validation checks the transaction undergoing validation with other later transactions, which are still active.

**Backward validation** □ As all the Read operations of earlier overlapping transactions were performed before the validation of $T_j$ started, they cannot be affected by the Write of the current transaction (and Rule 1 is satisfied). The validation of transaction $T_j$ checks whether its read set (the data items affected by the Read operations of $T_j$) overlaps with any of the write sets of earlier overlapping transactions $T_i$ (Rule 2). If there is any overlap, the validation fails.

Let $startTn$ be the biggest transaction number assigned (to some other concurrent transaction) at the time when transaction $T_j$ started its read phase and $finishTn$ be the biggest transaction number assigned at the time when $T_j$ entered the validation phase.

The following program describes the algorithm for the validation of $T_j$:

```
Valid := TRUE;
FOR $T_i$ := $startTn + 1$ TO $finishTn$
   IF write set of $T_i$ intersects read set of $T_j$ THEN
      Valid := FALSE
   END
END
```

In Figure 13.16, the write set of transaction $T_j$ must be compared with the read sets of the transactions with identifiers $active1$ and $active2$. (Forward validation should allow for the fact that read sets of active transactions may change during validation and writing.) As the read sets of the transaction being validated are not included in the check, read only transactions always pass the validation check. As the transactions being compared with the validating transaction are still active, we have a choice of whether to abort the validating transaction or to take some alternative way of resolving the conflict.

Härder [1984] suggests several alternative strategies:

- **Deferred validation:** This is an optimistic approach to validation because there is always the chance that further conflicting active transactions may start before the validation is achieved.
- **Abort all the conflicting active transactions and commit the transaction being validated.**
• Abort the transaction being validated. This is the simplest strategy, but has the disadvantage that the future conflicting transactions may be going to abort, in which case the transaction under validation has aborted unnecessarily.

**Comparison of forward and backward validation**

We have already seen that forward validation allows flexibility in the resolution of conflicts, whereas backward validation allows only one choice — to abort the transaction being validated. In general, the read sets of transactions are much larger than the write sets. Therefore, backward validation compares a possibly large read set against the old write sets, whereas, forward validation checks a small write set against the read sets of active transactions. We see that backward validation has the overhead of storing old write sets until they are no longer needed. On the other hand, forward validation has to allow for new transactions starting during the validation process.

**Starvation**

When a transaction is aborted, it will normally be restarted by the client program. But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of data items each time it is restarted. The deprivation of a transaction from ever being able to commit is called starvation.

Occurrences of starvation are likely to be rare, but a server that uses optimistic concurrency control must ensure that a client does not have its transaction aborted repeatedly. Kung and Robinson suggest that this could be done if the server detects a transaction that has been aborted several times. They suggest that when the server detects such a transaction it should be given exclusive access by the use of a critical section protected by a semaphore.

### 13.4 Timestamp ordering

In concurrency control schemes based on timestamp ordering, each operation on a data item is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately. On the other hand, the client sees a unique timestamp value associated with the transaction. It is assigned its position in the time sequence of transactions. Using timestamps, requests to access transactions can be totally ordered according to their timestamps. The timestamp ordering:basic timestamp ordering rule is based on operation conflicts and is simple:

A transaction’s request to write a data item is valid only if that data item was read and written by earlier transactions. A transaction’s request to read a data item is valid only if that data item was last written by an earlier transaction.

This rule assumes that there is only one version of each data item and restricts access by one transaction at a time. If each transaction has its own tentative version of each item it accesses, then multiple concurrent transactions can access the same data item. The timestamp ordering rule is refined to ensure that each transaction accesses only the versions of data items that have been committed before the current transaction started.

![Figure 13.17 Transaction conflicts for timestamp ordering.](image)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Write</td>
<td>$T_j$ must not write a data item that has been read by any $T_i$ where $T_i &gt; T_j$ this requires that $T_j \geq$ the maximum read timestamp of the data item</td>
</tr>
<tr>
<td>2. Write</td>
<td>$T_j$ must not write a data item that has been written by any $T_i$ where $T_i &gt; T_j$ this requires that $T_j &gt;$ the maximum write timestamp of the committed data item</td>
</tr>
<tr>
<td>3. Read</td>
<td>$T_j$ must not read a data item that has been written by any $T_i$ where $T_i &gt; T_j$ this implies that $T_j$ cannot read if $T_j &lt;$ write timestamp of the committed version of the data item</td>
</tr>
</tbody>
</table>
Figure 13.18 Write operations and timestamps.

(a) $T_3$ Write

Before

After

Time

(b) $T_3$ Write

Before

After

Time

data item produced by transaction $T_1$
(with write timestamp $T_1$
$T_1 < T_2 < T_3 < T_4$

(c) $T_3$ Write

Before

After

Time

(d) $T_3$ Write

Before

After

Time

Transaction aborts

Key:
Committed
Tentative

Timestamp ordering Write rule: By combining Rules 1 and 2 we have the following for deciding whether to accept a Write operation requested by transaction $T_j$ on data item $D$:

\[
\text{IF } T_j \geq \text{ maximum read timestamp on } D \text{ AND} \\
\text{perform Write operation on committed version of } D \text{ WITH write timestamp} \\
\text{ELSE (\textit{* write is too late \text{*}})} \\
\text{Abort transaction } T_j
\]

END

If a tentative version with write timestamp $T_j$ already exists, the Write operation addressed to it, otherwise a new tentative version is created and given write time $T_j$. Note that any Write that 'arrives too late' is aborted - it is too late in the sense of a transaction with a later timestamp has already read or written the data item.

Figure 13.18 illustrates the action of a Write operation by transaction $T_3$ where $T_3 \geq$ maximum read timestamp on the data item (the read timestamps are shown). In cases (a) to (c) $T_3 >$ write timestamp on the committed version of the data item and a tentative version with write timestamp $T_3$ is inserted at the appropriate position into the list of tentative versions ordered by their transaction timestamps. In case (d) $T_3 <$ write timestamp on the committed version of the data item and the transaction is aborted.

Timestamp ordering Read rule: By using Rule 3 we have the following rule for deciding whether to accept immediately, to wait or to reject a Read operation requested by transaction $T_j$ on data item $D$:

\[
\text{IF } T_j > \text{ write timestamp on committed version of } D \text{ THEN} \\
\text{let } D_{\text{selected}} \text{ be the version of } D \text{ with the maximum write timestamp} \leq T_j \\
\text{IF } D_{\text{selected}} \text{ is committed THEN} \\
\text{perform Read operation on the version } D_{\text{selected}} \\
\text{ELSE} \\
\text{Wait until the transaction that made version } D_{\text{selected}} \text{ commits or aborts} \\
\text{then re-apply the Read rule} \\
\text{ELSE} \\
\text{Abort transaction } T_j
\]

END

Note:
- If the transaction $T_j$ has already written its own version of the data item, this will be used.
- A Read operation that arrives too early waits for the earlier transaction to complete. If the earlier transaction commits then $T_j$ will read from its committed version. If it aborts then $T_j$ will repeat the Read rule (and select the previous version). This rule prevents dirty reads.
- A Read operation that 'arrives too late' is aborted - it is too late in the sense that a transaction with a later timestamp has already written the data item.

The Figure 13.19 illustrates the timestamp ordering read rule. It includes four cases labelled (a) to (d), each of which illustrates the action of a Read operation by transaction $T_3$. In each case, a version whose write timestamp is less than or equal to $T_3$ is selected. If such a version exists, it is indicated with a line. In cases (a) and (b) the Read operation is directed to a committed version - in (a) it is the only version, whereas in (b) there is a tentative version belonging to a later transaction. In case (c) the Read operation is directed to a tentative version and must wait until the transaction that made it commits or aborts. In case (d) there is no suitable version to read and the transaction $T_3$ is aborted.

When a server receives a request to commit a transaction, it will always be able to do so because all the operations of transactions are checked for consistency with those of earlier transactions before being carried out. The committed versions of each data item must be created in timestamp order. Therefore a server sometimes needs to wait for earlier transactions to complete before writing all the committed versions of the data items accessed by a particular transaction but there is no need for the client to wait. In order to make a transaction recoverable after a server crash, the tentative versions of data items and the fact that the transaction has committed must be written to permanent storage before acknowledging the client's request to commit the transaction.

Note that this timestamp ordering algorithm is a strict one - it ensures strict executions of transactions (see Section 13.4). The Timestamp ordering Read rule delays the reading of a committed version if there may be an earlier transaction that has a later commit time, but it does not prevent reading an earlier committed version - this would be inconsistent with our goal of avoiding dependency cycles. Note that this algorithm requires a total order on the timestamps of transactions.
Figure 13.19 Read operations and timestamps.

a) $T_3$ Read

<table>
<thead>
<tr>
<th>Read proceeds</th>
<th>Selected</th>
<th>Time</th>
</tr>
</thead>
</table>

b) $T_3$ Read

<table>
<thead>
<tr>
<th>Read proceeds</th>
<th>Selected</th>
<th>Time</th>
</tr>
</thead>
</table>

c) $T_3$ Read

<table>
<thead>
<tr>
<th>Read waits</th>
<th>$T_2$</th>
<th>Selected</th>
<th>Time</th>
</tr>
</thead>
</table>

d) $T_3$ Read

<table>
<thead>
<tr>
<th>Transaction aborts</th>
<th>$T_4$</th>
<th>Selected</th>
<th>Time</th>
</tr>
</thead>
</table>

Key:
- Committed
- Tentative

Data item produced by transaction $T_1$ (with write timestamp $T_1$)

$T_1 < T_2 < T_3 < T_4$

Figure 13.20 Timestamps in transactions $T$ and $U$.

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
<th>Timestamps and versions of data items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RTS$</td>
<td>$S$</td>
<td>$S$</td>
</tr>
</tbody>
</table>

OpenTransaction

$bal := A.Read($)

OpenTransaction

$bal := C.Read($)

A.Write ($bal - 4$

$bal := B.Read($)

C.Write ($bal - 3$

$bal := B.Read($)

B.Write ($bal + 4$

Aborts

B.Write ($bal + 3$

Multiversion timestamp ordering

In this section we have shown how the concurrency provided by basic timestamp ordering is improved by allowing each transaction to write its own tentative versions of data items. In multiversion timestamp ordering, which was introduced by Reed [1983], the server keeps old committed versions as well as tentative versions in its list of versions of data items. This list represents the history of the values of the data item. The benefit of using multiple versions is that Read operations that arrive too late need not be rejected.

Each version has a read timestamp recording the largest timestamp of any transaction that has read from it in addition to a write timestamp. As before, whenever a Write operation is accepted, it is directed to a tentative version with the write timestamp of the transaction. Whenever a Read operation is carried out it is directed to the version with the largest write timestamp less than the transaction timestamp. If the transaction timestamp is larger than the read timestamp of the version being used, the read timestamp of the version is set to the transaction timestamp.

When a read arrives late, the server can allow it to read from an old committed version, so there is no need to abort late Read operations. In multiversion timestamp ordering, Read operations are always permitted, although they may have to Wait for earlier transactions to complete (either commit or abort), which ensures that executions are recoverable. See Exercise 13.15 for a discussion of the possibility of cascading aborts. This deals with Rule 3 in the conflict rules for timestamp ordering.
Figure 13.21 Late Write operation would invalidate a Read.

There is no conflict between Write operations of different transactions because each transaction writes its own committed version of the data items it accesses. This removes Rule 2 in the conflict rules for timestamp ordering, leaving us with:

Rule 1. T_j must not write data items that have been read by any T_i where T_i > T_j.

This rule will be broken if there is any version of the data item with read timestamp > T_j; but only if this version has write timestamp less than or equal to T_j. (This will not happen if there are no later versions.)

Multiversion timestamp ordering Write rule: As any potentially conflicting Read operation will have been directed to the most recent version of a data item, the server inspects the version, D_{maxEarlier} with the maximum write timestamp less than or equal to T_j. We have the following rule for performing a Write operation requested on data item D:

IF read timestamp of D_{maxEarlier} ≤ T_j THEN
perform Write operation on a tentative version of D with write timestamp T_j
ELSE Abort transaction T_j
END

Figure 13.21 illustrates an example where a Write is rejected. The data item already has committed versions with write timestamps T_1 and T_2. The server receives the following sequence of requests for operations on the data item:

T_3 Read; T_3 Write; T_5 Read; T_4 Write

1. T_3 requests a Read operation which puts a read timestamp T_3 on T_2's version.
2. T_3 requests a Write operation which makes a new tentative version with write timestamp T_3.
3. T_3 requests a Read operation which uses the version with write timestamp T_3 (the highest timestamp that is less than T_3).
4. T_4 requests a Write operation which is rejected because the read timestamp T_5 of the version with write timestamp T_3 is bigger than T_4. (If it were permitted, the write timestamp of the new version would be T_4. If such a version were allowed, then it would invalidate T_5's Read operation that should have used the version with timestamp T_4.)

When a transaction is aborted, all the versions that it created are deleted. When a transaction is committed, all the versions that it created are retained, but to control the use of storage space, old versions must be deleted from time to time. Although it has the overhead of storage space, multiversion timestamp ordering does allow considerable concurrency, does not suffer from deadlocks and always permits Read operations. For further information about multiversion timestamp ordering, see Bernstein et al. [1987].

13.5 Comparison of methods for concurrency control

We have described three separate methods for controlling concurrent access to shared data: strict two-phase locking, optimistic methods and timestamp ordering. All of the methods carry some overheads in the time and space they require and they all limit to some extent the potential for concurrent operation.

The timestamp ordering method is similar to two-phase locking in that both use pessimistic approaches in which the server detects conflicts between transactions as each data item is accessed. On the one hand, timestamp ordering decides the serialization order statically – when a transaction starts. On the other hand, two-phase locking decides the serialization order dynamically – according to the order in which data items are accessed. Timestamp ordering and in particular multiversion timestamp ordering is better than strict two-phase locking for read only transactions. Two-phase locking is better when the operations in transactions are predominantly updates.

Some recent work uses the observation that timestamp ordering is beneficial for transactions with predominantly Read operations and that locking is beneficial for transactions with more Writes than Reads as an argument for allowing hybrid schemes in which some transactions use timestamp ordering and others use locking for concurrency control. Readers who are interested in the use of mixed methods should read Bernstein et al. [1987].

The pessimistic methods differ in the strategy used when a conflicting access to a data item is detected. Timestamp ordering aborts the transaction immediately, whereas locking makes the transaction wait – but with a possible later penalty of aborting to avoid deadlock.

When optimistic concurrency control is used all transactions are allowed to proceed, but some are aborted when they attempt to commit, or in forward validation transactions are aborted earlier. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.
Locking has been in use for many years in database systems, but timestamp ordering has been used in the SDD-1 database system. Both methods have been used in file servers.

Several recent distributed systems for example, Argus [Liskov 1988] and Arjun [Shrivastava et al. 1991] have explored the use of semantic locks, timestamp ordering and new approaches to long transactions.

Recent work in two application areas has shown that the above concurrency control mechanisms are not always adequate. One of these areas concerns multi-user applications in which all users expect to see common views of data items being updated by any of the users. Such applications require their data to be atomic in the presence of concurrent updates and server failures; and transaction techniques appear to offer an approach to their design. However, these applications have two new requirements relating to concurrency control: (i) users require immediate notification of changes made by other users which is contrary to the idea of isolation, (ii) users need to be able to access data items before other users have completed their transactions, which may lead to the development of new types of locks that trigger actions when data items are accessed. Work in this area has suggested many schemes that relax isolation and provide notification of changes. For a review of this work see Ellis et al. [1991]. The second application area concerns what are sometimes described as advanced database applications such as co-operative CAD/CAM and software development systems. Such applications, transactions last for a long time, and users work on independent versions of data items that are checked out from a common database and checked in when the work is finished. The merging of versions requires co-operation between users. For a review of this work see Barghouti and Kaiser [1991].

13.6 Summary

Operation conflicts form a basis for the derivation of concurrency control protocols. Protocols not only must ensure serializability but also allow for recovery by using executions to avoid problems associated with transactions aborting, such as cascading aborts.

When a server receives a request for an operation in a transaction it may only execute it (i) immediately, (ii) delay it, or (iii) abort it. Strict two-phase locking uses the first two strategies, resorting to abortion only if the case of deadlock. It ensures serializability by ordering transactions according to when they access common data items. Its main drawback is that deadlocks cannot be detected.

Timestamp ordering uses all three strategies to ensure serializability by ordering transactions according to the time transactions start. This means transactions cannot suffer from deadlocks and is advantageous for read only transactions. However, transactions must be aborted when they arrive too late. Multiversion timestamp ordering is particularly effective.

Optimistic concurrency control allows transactions to proceed without any of checking until they are completed. Transactions are validated before being committed. Backward validation requires the maintenance of multiple write committed transactions whereas forward validation must validate against transactions and has the advantage that it allows alternative strategies for resolving conflicts. Starvation can occur due to repeated aborting of a transaction that fails validation in optimistic concurrency control and even in timestamp ordering.