UNIT 5

Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information.

• **Transaction failure -** There are two types of errors that may cause a transaction to fail:

 Logical error - The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.

 System error - The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

• **System crash** - There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.
The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level that brings the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

• **Disk failure** - A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure. To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures.

These algorithms, known as recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.

2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

Recovery Techniques –

1. **Log Based recovery technique**
2. **Shadow Paging**

Log-Based Recovery

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database. There are several types of log records. An **update log record** describes a single database write. It has these fields:
• **Transaction identifier** is the unique identifier of the transaction that performed the write operation.

• **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
• **Old value** is the value of the data item prior to the write.
• **New value** is the value that the data item will have after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.
We denote the various types of log records as:
• <*Ti* start>. Transaction *Ti* has started.
• <*Ti*, *Xj*, *V*1, *V*2>. Transaction *Ti* has performed a write on data item *Xj* . *Xj*
had value *V*1 before the write, and will have value *V*2 after the write.
• <*Ti* commit>. Transaction *Ti* has committed.
• <*Ti* abort>. Transaction *Ti* has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records. For log records to be useful for recovery from system and disk failures, the log must reside in stable storage.

Deferred Database Modification

The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits.

When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.
The execution of transaction *Ti* proceeds as follows. Before *Ti* starts its execution, a record <*Ti* start> is written to the log. A write(*X*) operation by *Ti* results in the writing of a new record to the log. Finally, when *Ti* partially commits, a record <*Ti* commit> is written to the log. When transaction *Ti* partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state. Observe that only the new value of the data item is required by the deferred modification technique. Thus, we can simplify the general update-log record structure that we saw in the previous section, by omitting the old-value field.

Let *T*0 be a transaction that transfers $50 from account *A* to account *B*:
***T*0: read(*A*);**
***A* := *A* − 50;**
write(*A*);
read(*B*);
***B* := *B* + 50;**
write(*B*).
Let *T*1 be a transaction that withdraws $100 from account *C*:
***T*1: read(*C*);**
***C* := *C* − 100;**
write(*C*).

Suppose that these transactions are executed serially, in the order *T*0 followed by *T*1, and that the values of accounts *A*, *B*, and *C* before the execution took place were $1000, $2000, and $700, respectively. The portion

of the log containing the relevant information on these two transactions appears in Figure 17.2. There are various orders in which the actual outputs can take place to both the database system and the log as a result of the execution of *T*0 and *T*1. One such order appears in Figure 17.3.

<*T*0 start>
<*T*0 , *A*, 950>
<*T*0 , *B*, 2050>
<*T*0 commit>
<*T*1 start>
<*T*1 , *C*, 600>
<*T*1 commit>

Figure 17.2 Portion of the database log corresponding to *T*0 and *T*1.

Note that the value of *A* is changed in the database only after the record <*T*0, *A*, 950> has been placed in the log. Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme uses the following recovery procedure:

• redo(*Ti*) sets the value of all data items updated by transaction *Ti* to the new values.

The set of data items updated by *Ti* and their respective new values can be found in the log.

The redo operation must be **idempotent**; that is, executing it several times must be equivalent to executing it once. This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process. After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction *Ti* needs to be redone if and only if the log contains both the record <*Ti* start> and the record <*Ti* commit>. Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.

Transactions *T*0 and *T*1 executed one after the other in the order *T*0 followed by *T*1. Figure 17.2 shows the log that result from the complete execution of *T*0 and *T*1.

***A* = 950**
***B* = 2050**
***C* = 600**
<*T*0 start>
<*T*0 , *A*, 950>
<*T*0 , *B*, 2050>
<*T*0 commit>
<*T*1 start>
<*T*1 , *C*, 600>
<*T*1 commit>

Figure 17.3 State of the log and database corresponding to *T*0 and *T*1.

<*T*0 start>	**<*T*0 start>**	**<*T*0 start>**
<*T*0 , *A*, 950>	**<*T*0 , *A*, 950>**	**<*T*0 , *A*, 950>**
<*T*0 , *B*, 2050>	**<*T*0 , *B*, 2050>**	**<*T*0 , *B*, 2050>**
	<*T*0 commit>	**<*T*0 commit>**
	<*T*1 start>	**<*T*1 start>**
	<*T*1 , *C*, 600>	**<*T*1 , *C*, 600>**
		<*T*1 commit>
(a)	**(b)**	**(c)**

Figure 17.4 The same log as that in Figure 17.3, shown at three different times.

Let us suppose that the **Log Database** system crashes before the completion of the transactions, so that we can see how the recovery technique restores the database to a consistent state. Assume that the crash occurs just after the log record for the step write(*B*) of transaction *T*0 has been written to stable storage. The log at the time of the crash appears in Figure 17.4a. When the system comes back up, no redo actions need to be taken, since no commit record appears in the log. The values of accounts *A* and *B* remain $1000 and $2000, respectively. The log records of the incomplete transaction *T*0 can be deleted from the log.

Now, let us assume the crash comes just after the log record for the step write(*C)* of transaction *T*1 has been written to stable storage. In this case, the log at the time of the crash is as in Figure 17.4b. When the system comes back up, the operation redo(*T*0) is performed, since the record <*T*0 commit> appears in the log on the disk. After this operation is executed, the values of accounts *A* and *B* are $950 and $2050, respectively. The value of account *C* remains $700. As before, the log records of the incomplete transaction *T*1 can be deleted from the log.

Finally, assume that a crash occurs just after the log record <*T*1 commit> is written to stable storage. The log at the time of this crash is as in Figure 17.4c. When the system comes back up, two commit records are in the log: one for *T*0 and one for *T*1. Therefore, the system must perform operations redo(*T*0) and redo(*T*1), in the order in which their commit records appear in the log. After the system executes these operations, the values of accounts *A*, *B*, and *C* are $950, $2050, and $600, respectively.

Finally, let us consider a case in which a second system crash occurs during recovery from the first crash. Some changes may have been made to the database as a result of the redo operations, but all changes may not have been made. When the system comes up after the second crash, recovery proceeds exactly as in the preceding examples. For each commit record <*Ti* commit> found in the log, the the system performs the operation redo(*Ti*). In other words, it restarts the recovery actions from the beginning. Since redo writes values to the database independent of the values currently in the database, the result of a successful second attempt at redo is the same as though redo had succeeded the first time.

Immediate Database Modification

The **immediate-modification technique** allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called **uncommitted modifications**. In the event of a crash or a transaction failure, the system must use the old-value field of the log records described in Section 17.4 to restore the modified data items to the value they had prior to the start of the transaction. The undo operation, described next, accomplishes this restoration. Before a transaction *Ti* starts its execution, the system writes the record <*Ti* start> to the log. During its execution, any write(*X*) operation by *Ti* is *preceded* by the writing of the appropriate new update record to the log. When *Ti* partially commits, the system writes the record <*Ti* commit> to the log.

Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage. We therefore require that, before execution of an output(*B*) operation, the log records corresponding to *B* be written onto stable storage. We shall return to this issue in Section 17.7.As an illustration, let us reconsider our simplified banking system, with transactions *T*0 and *T*1 executed one after the other in the order *T*0 followed by *T*1. The portion of the log containing the relevant information concerning these two transactions appears in Figure 17.5. Figure 17.6 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of *T*0 and *T*1.

<*T*0 start>
<*T*0 , *A*, 1000, 950>
<*T*0 , *B*, 2000, 2050>
<*T*0 commit>
<*T*1 start>

<*T*1 , *C*, 700, 600>
<*T*1 commit>
Figure 17.5 Portion of the system log corresponding to *T*0 and *T*1.

Log	**Database**
<*T*0 start>	
<*T*0 , *A*, 1000, 950>	
<*T*0 , *B*, 2000, 2050>	
	$A = 950$
	$B = 2050$
<*T*0 commit>	
<*T*1 start>	
<*T*1 , *C*, 700, 600>	
	$C = 600$
<*T*1 commit>	

Figure 17.6 State of system log and database corresponding to *T*0 and *T*1.

Notice that this order could not be obtained in the deferred-modification technique. Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:
• undo(*Ti*) restores the value of all data items updated by transaction *Ti* to the old values.
• redo(*Ti*) sets the value of all data items updated by transaction *Ti* to the new values.

The set of data items updated by *Ti* and their respective old and new values can be found in the log. The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process.
After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

• Transaction *Ti* needs to be undone if the log contains the record <*Ti* start>,
but does not contain the record <*Ti* commit>.
• Transaction *Ti* needs to be redone if the log contains both the record <*Ti* start> and the record <*Ti* commit>.

As an illustration, return to our banking example, with transaction *T*0 and *T*1 executed one after the other in the order *T*0 followed by *T*1. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure 17.7.

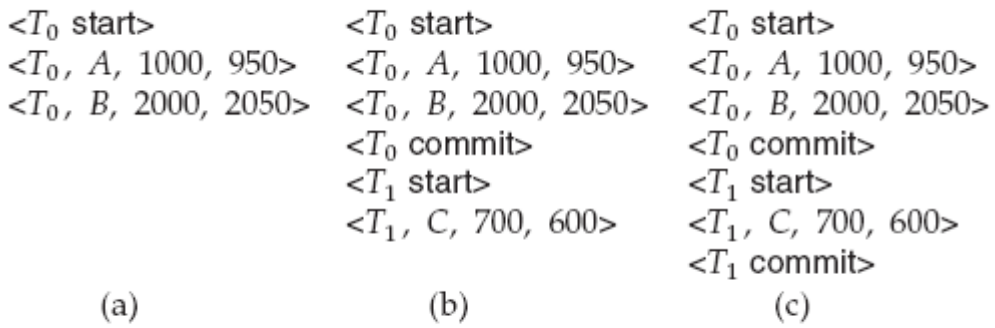

Figure 17.7 The same log, shown at three different times.

First, let us assume that the crash occurs just after the log record for the step write(*B*) of transaction *T*0 has been written to stable storage (Figure 17.7a). When the system comes back up, it finds the record <*T*0 start> in the log, but no corresponding <*T*0 commit> record. Thus, transaction *T*0 must be undone, so an undo(*T*0) is performed. As a result, the values in accounts *A* and *B* (on the disk) are restored to $1000 and $2000, respectively.

Next, let us assume that the crash comes just after the log record for the step write(*C*) of transaction *T*1 has been written to stable storage (Figure 17.7b). When the system comes back up, two recovery actions need to be taken. The operation undo(*T*1) must be performed, since the record <*T*1 start> appears in the log, but there is no record
<*T*1 commit>. The operation redo(*T*0)must be performed, since the log contains both the record <*T*0 start> and the record <*T*0 commit>. At the end of the entire recovery procedure, the values of accounts *A*, *B*, and *C* are $950, $2050, and $700, respectively. Note that the undo(*T*1) operation is performed before the redo(*T*0). In this example, the same outcome would result if the order were reversed.

Finally, let us assume that the crash occurs just after the log record <*T*1 commit> has been written to stable storage (Figure 17.7c). When the system comes back up, both *T*0 and *T*1 need to be redone, since the records <*T*0 start> and <*T*0 commit> appear in the log, as do the records <*T*1 start> and <*T*1 commit>. After the system performs the recovery procedures redo(*T*0) and redo(*T*1), the values in accounts *A*, *B*, and *C* are $950, $2050, and $600, respectively.

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce checkpoints. During execution, the system maintains the log, using one of the two techniques, i.e. deferred or immediate database modification. In addition, the system periodically performs **checkpoints**, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction *Ti* that committed prior to the checkpoint. For such a transaction, the
<*Ti* commit> record appears in the log before the <checkpoint> record. Any database modifications made by *Ti* must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on *Ti*.

After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction *Ti* that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final <checkpoint> record in the log); then it continues the search backward until it finds the next <*Ti* start> record. This record identifies a transaction *Ti*. Once the system has identified transaction *Ti*, the redo and undo operations need to be applied to only transaction *Ti* and all transactions *Tj* that started executing after transaction *Ti*. Let us denote these transactions by the set *T*. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on the modification technique being used. For the immediate-modification technique, the recovery operations are:

- For all transactions *Tk* in *T* that have no <*Tk* commit> record in the log, execute undo(*Tk*).
- For all transactions *Tk* in *T* such that the record <*Tk* commit> appears in the log, execute redo(*Tk*).

Obviously, the undo operation does not need to be applied when the deferred-modification technique is being employed. As an illustration, consider the set of transactions *{T*0, *T*1, . . ., *T*100*}* executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction *T*67. Thus, only transactions *T*67, *T*68, . . ., *T*100 need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

Shadow Paging

An alternative to log-based crash-recovery techniques is **shadow paging**. Under certain circumstances, shadow paging may require fewer disk accesses than do the log-based methods. There are, however, disadvantages to the shadow-paging approach, as we shall see, that limit its use. For example, it is hard to extend shadow paging to allow multiple transactions to execute concurrently. As before, the database is partitioned into some number of fixed-length blocks, which are referred to as **pages**. The term *page* is borrowed from operating systems, since we are using a paging scheme for memory management. Assume that there are *n* pages, numbered 1 through *n.*These pages do not need to be stored in any particular order on disk. However, there must be a way to find the *i*th page of the database for any given *i*.We use a **page table**, as in Figure 17.8, for this purpose. The page table has *n* entries—one for each database page. Each entry contains a pointer to a page on disk. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on. The example in Figure 17.8 shows that the logical order of database pages does not need to correspond to the physical order in which the pages are placed on disk.

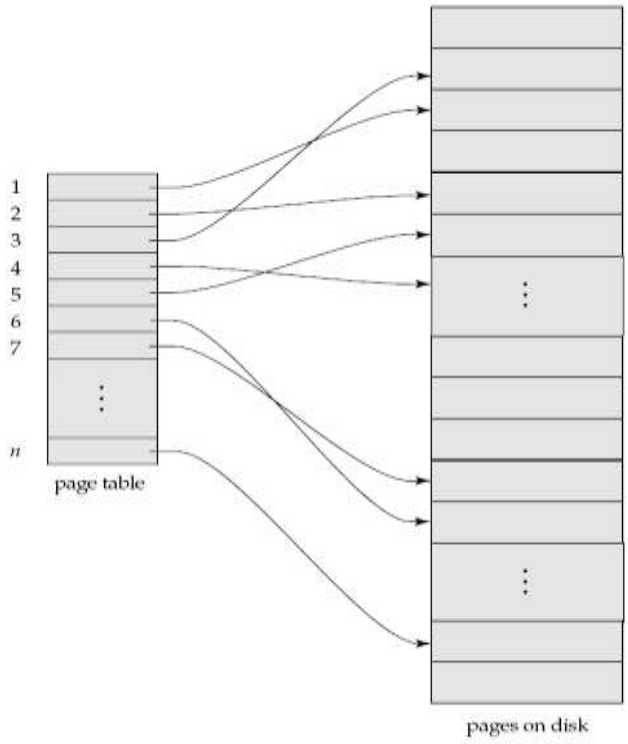

Figure 17.8 Sample page table.

The key idea behind the shadow-paging technique is to maintain *two* page tables during the life of a transaction: the **current page table** and the **shadow page table**. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk. Suppose that the transaction *Tj* performs a write(*X*) operation, and that *X* resides on the *i*th page. The system executes the write operation as follows:

1. If the *i*th page (that is, the page on which *X* resides) is not already in main memory, then the system issues input(*X*).

2. If this is the write first performed on the *i*th page by this transaction, then the system modifies the current page table as follows:

a. It finds an unused page on disk. Usually, the database system has access to a list of unused (free) pages

b. It deletes the page found in step 2a from the list of free page frames; it copies the contents of the *i*th page to the page found in step 2a.

c. It modifies the current page table so that the *i*th entry points to the page found in step 2a.

3. It assigns the value of *xj* to *X* in the buffer page.

Step 2, manipulates the current page table. Figure 17.9 shows the shadow and current page tables for a transaction performing a write to the fourth page of a database consisting of 10 pages. Intuitively, the shadow-page approach to recovery is to store the shadow page table in nonvolatile storage, so that the state of the database prior to the execution of the transaction can be recovered in the event of a crash, or transaction abort. When the transaction commits, the system writes the current page table to nonvolatile storage.

The current page table then becomes the new shadow page table, and the next transaction is allowed to begin execution. It is important that the shadow page table be stored in nonvolatile storage, since it provides the only means of locating database pages. The current page table may be kept in main memory (volatile storage).We do not care whether the current page table is lost in a crash, since the system recovers by using the shadow page table. Successful recovery requires that we find the shadow page table on disk after a crash. A simple way of finding it is to choose one fixed location in stable storage that contains the disk address of the shadow page table. When the system comes back up after a crash, it copies the shadow page table into main memory

and uses it for subsequent transaction processing. Because of our definition of the write operation, we are guaranteed that the shadow page table will point to the database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash. Thus, aborts are automatic. Unlike our log-based schemes, shadow paging needs to invoke no undo operations. To commit a transaction, we must do the following:

1. Ensure that all buffer pages in main memory that have been changed by the transaction are output to disk.
2. Output the current page table to disk. Note that we must not overwrite the shadow page table, since we may need it for recovery from a crash.
3. Output the disk address of the current page table to the fixed location in stable storage containing the address of the shadow page table. This action overwrites the address of the old shadow page table. Therefore, the current page table has become the shadow page table, and the transaction is committed.

If a crash occurs prior to the completion of step 3, we revert to the state just prior to the execution of the transaction. If the crash occurs after the completion of step 3, the effects of the transaction will be preserved; no redo operations need to be invoked. Shadow paging offers several advantages over log-based techniques. The overhead of log-record output is eliminated, and recovery from crashes is significantly.

Drawbacks to the shadow-page technique:

• **Commit overhead**. The commit of a single transaction using shadow paging requires multiple blocks to be output—the actual data blocks, the current page table, and the disk address of the current page table. Log-based schemes need to output only the log records, which, for typical small transactions, fit within one block.
The overhead of writing an entire page table can be reduced by implementing the page table as a tree structure, with page table entries at the leaves. The nodes of the tree are pages and have a high fanout, like B+-trees. The current page table's tree is initially the same as the shadow page table's tree. When a page is to be updated for the first time, the system changes the entry in the current page table to point to the copy of the page. If the leaf page containing the entry has been copied already, the system directly updates it. Otherwise, the system first copies it, and updates the copy. In turn, the parent of the copied page needs to be updated to point to the new copy, which the system does by applying the same procedure to its parent, copying it if it was not already copied. The process of copying proceeds up to the root of the tree. Changes are made only to the copied nodes, so the shadow page table's tree does not get modified. The benefit of the tree representation is that the only pages that need to be copied are the leaf pages that are updated, and all their ancestors in the tree. All the other parts of the tree are shared between the shadow and the current page table, and do not need to be copied. The reduction in copying costs can be very significant for large databases. However, several pages of the page table still need to be copied for each transaction, and the log-based schemes continue to be superior as long as most transactions update only small parts of the database.
• **Data fragmentation**. In Shadow paging causes database pages to change location when they are updated. As a result, either we lose the locality property of the pages or we must resort to more complex, higher-overhead schemes for physical storage management.
• **Garbage collection**. Each time that a transaction commits the database pages containing the old version of data changed by the transaction become inaccessible. In Figure 17.9, the page pointed to by the fourth entry of the shadow page table will become inaccessible once the transaction of that example commits. Such pages are considered **garbage**, since they are not part of free space and do not contain usable information. Garbage may be created also as a side effect of crashes. Periodically, it is necessary to find all the garbage pages, and to add them to the list of free pages. This process, called **garbage collection**, imposes additional overhead and complexity on the system. There are several standard algorithms for garbage collection.

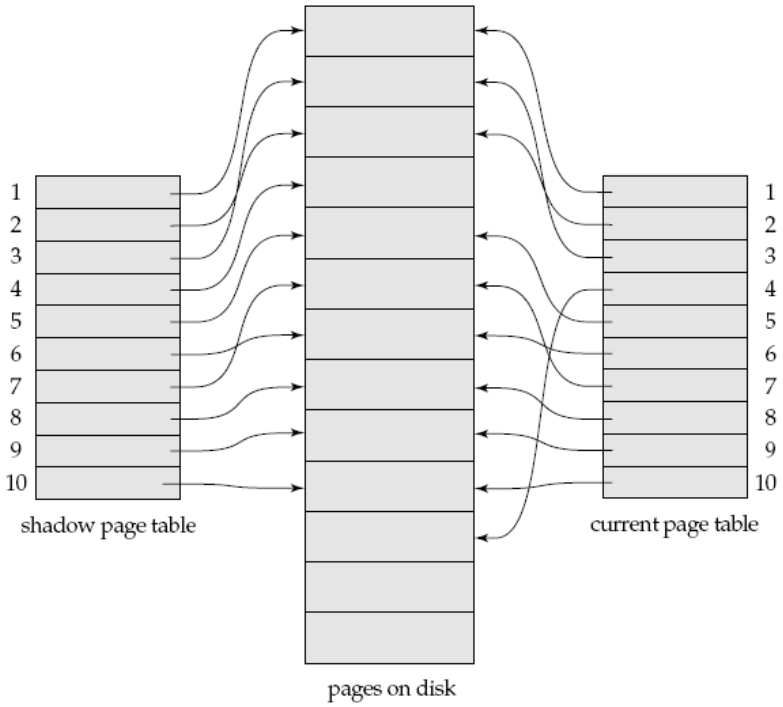

Figure 17.9 Shadow and current page tables.

In addition to the drawbacks of shadow paging just mentioned, shadow paging is more difficult than logging to adapt to systems that allow several transactions to execute concurrently. In such systems, some logging is usually required, even if shadow paging is used.

On-line backup during database updates

Or

Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**, that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called the **primary site**, and having a **remote backup** site where all the data from the primary site are replicated. The remote backup site is sometimes also called the **secondary site**. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example; we can locate it in a different state—so that a disaster at the primary does not damage the remote backup site. Figure 17.10 shows the architecture of a remote backup system.

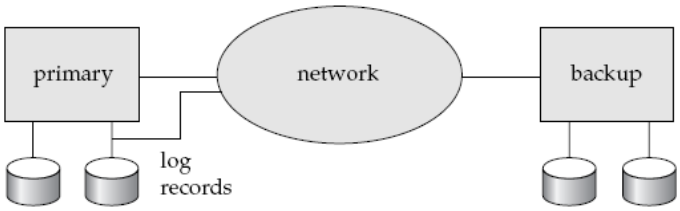

Figure 17.10 Architecture of remote backup system.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions. Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost. The performance of a remote backup system is better than the performance of a distributed system with two-phase commit.
Several issues must be addressed in designing a remote backup system:

• **Detection of failure**. As in failure-handling protocols for distributed system, it is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup. For example, in addition to the network connection, there may be a separate modem connection over a telephone line, with services provided by different telecommunication companies. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.
• **Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down. The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site.
If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.
• **Time to recover**. If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received, and can perform a checkpoint, so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result. A **hot-spare** configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.
• **Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit
a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows.

- **One-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary site. The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site, when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site.

- **Two-very-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site. The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.
- **Two-safe**. This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site. This scheme provides better availability than doe's two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost. Several commercial shared-disk systems provide a level of fault tolerance that is intermediate between centralized and remote backup systems.

Lock-Based Protocols

One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Locks –

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. ***Shared***. If a transaction *Ti* has obtained a shared-mode lock (denoted by S) on item *Q*, then *Ti* can read, but cannot write, *Q*.
2. ***Exclusive***. If a transaction *Ti* has obtained an exclusive-mode lock (denoted by X) on item *Q*, then *Ti* can both read and write *Q*.

We require that every transaction request a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction. Given a set of lock modes, we can define a compatibility function on them as follows.

Let *A* and *B* represent arbitrary lock modes. Suppose that a transaction *Ti* requests a lock of mode *A* on item *Q* on which transaction *Tj* (*Ti* _= *Tj*) currently holds a lock of mode *B*. If transaction *Ti* can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is compatible with mode *B*. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking appears in the matrix comp of Figure 16.1. An element comp(*A, B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B*.

	S	X
S	true	false
X	false	false

Figure 16.1 Lock-compatibility matrix comp.

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item *Q* by executing the lock-S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

- To access a data item, transaction *Ti* must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, *Ti* is made to wait until all incompatible locks held by other transactions have been released.

- Transaction *Ti* may unlock a data item that it had locked at some earlier point.A transaction must hold a lock on a data item as long as it accesses that item.

- For a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. Transaction *T*1 transfers $50 from account *B* to account *A* (Figure 16.2).

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 16.2 Transaction T_1.

Transaction *T*2 displays the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B* (Figure 16.3).

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 16.3 Transaction T_2.

Suppose that the values of accounts *A* and *B* are $100 and $200, respectively. If these two transactions are executed serially, either in the order *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 16.4 is possible.

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 16.4 Schedule 1.

In this case, transaction *T*2 displays $250, which is incorrect. The reason for this mistake is that the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistent state.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency-control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction.

Suppose now that unlocking is delayed to the end of the transaction. Transaction *T*3 corresponds to *T*1 with unlocking delayed (Figure 16.5). Transaction *T*4 corresponds to *T*2 with unlocking delayed (Figure 16.6).

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

Figure 16.5 Transaction T_3.

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Figure 16.6 Transaction T_4.

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 16.7 for *T*3 and *T*4. Since *T*3 is holding an exclusive-mode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 16.7 Schedule 2.

Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive-mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock. When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

If we do not use locking, or if we unlock data items as soon as possible after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. Deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back of transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

We shall require that each transaction in the system follow a set of rules, called a locking protocol, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.

Let *{T*0, *T*1, . . ., *Tn}* be a set of transactions participating in a schedule *S*. We say that *Ti* precedes *Tj* in *S*, written *Ti* → *Tj*, if there exists a data item *Q* such that *Ti* has held lock mode *A* on *Q*, and *Tj* has held lock mode *B* on *Q* later, and comp(*A*,*B*) = false. If *Ti* → *Tj* , then that precedence implies that in any equivalent serial schedule, *Ti* must appear before *Tj* .

Granting of Locks –

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction *T*2 has a shared-mode lock on a data item, and another transaction *T*1 requests an exclusive-mode lock on the data item. Clearly, *T*1 has to wait for *T*2 to release the shared-mode lock. Meanwhile, a transaction *T*3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to *T*2, so *T*3 may be granted the shared-mode lock. At this point *T*2 may release the lock, but still *T*1 has to wait for *T*3 to finish. But again, there may be a new transaction *T*4 that requests a shared-mode lock on the same data item, and is granted the lock before *T*3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but *T*1 never gets the exclusive-mode lock on the data item. The transaction *T*1 may never make progress, and is said to be starved. We can avoid starvation of transactions by granting locks in the following manner:

When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that –

1. There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.
2. There is no other transaction that is waiting for a lock on *Q*, and that made its lock request before *Ti*.

The Two-Phase Locking Protocol

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase. A transaction may obtain locks, but may not release any lock.
2. Shrinking phase. A transaction may release locks, but may not obtain any new locks. Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions *T*3 and *T*4 are two phase. On the other hand, transactions *T*1 and *T*2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction *T*3, we could move the unlock(*B*) instruction to just after the lock-X(*A*) instruction, and still retain the two-phase locking property.

We can show that the two-phase locking protocol ensures conflict serializability.
Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions.

Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions *T*3 and *T*4 are two phase, but, in schedule 2 (Figure 16.7), they are deadlocked.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure 16.7 Schedule 2. **Figure 16.8** Partial schedule under two-phase locking.

To being serializable, schedules should be cascade less. Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 16.8. Each transaction observes the two-phase locking protocol, but the failure of *T*5 after the read(A) step of *T*7 leads to cascading rollback of *T*6 and *T*7. Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase

locking protocol. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data. Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two-phase locking.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

***T*8: read(*a*1);**
read(*a*2);
. . .
read(*an*);
write(*a*1).
***T*9: read(*a*1);**
read(*a*2);
display(*a*1 + *a*2).

If we employ the two-phase locking protocol, then *T*8 must lock *a*1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that *T*8 needs an exclusive lock on *a*1 only at the end of its execution, when it writes *a*1. Thus, if *T*8 could initially lock *a*1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since *T*8 and *T*9 could access *a*1 and *a*2 simultaneously. This observation leads us to a refinement of the basic two-phase locking protocol, in which lock conversions are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by upgrade, and from exclusive to shared by downgrade. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Returning to our example, transactions *T*8 and *T*9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 16.9, where only some of the locking instructions are shown.

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Figure 16.9 Incomplete schedule with a lock conversion.

Note that a transaction attempting to upgrade a lock on an item *Q* may be forced to wait. This enforced wait occurs if *Q* is currently locked by *another* transaction in shared mode. Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and

transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascade less.
For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. In the absence of such information, two-phase locking is necessary for conflict serializability—if *Ti* is a non-two-phase transaction, it is always possible to find another transaction *Tj* that is two-phase so that there is a schedule possible for *Ti* and *Tj* that is not conflict serializable.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

A simple but widely used scheme automatically generates the appropriate lock and unlocks instructions for a transaction, on the basis of read and writes requests from the transaction:
• When a transaction *Ti* issues a read(*Q*) operation, the system issues a lock-S(*Q*) instruction followed by the read(*Q*) instruction.
• When *Ti* issues a write(*Q*) operation, the system checks to see whether *Ti* already holds a shared lock on *Q*. If it does, then the system issues an upgrade (*Q*) instruction, followed by the write(*Q*) instruction. Otherwise, the system issues a lock-X(*Q*) instruction, followed by the write(*Q*) instruction.
• All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Graph based protocol –

To acquire the knowledge about the order in which the database items will be accessed we impose a partial ordering → on the set
D = *{d*1, *d*2, . . ., *dh}* of all data items. If *di* → *dj* , then any transaction accessing both *di* and *dj* must access *di* before accessing *dj* . This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control. The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**.

In the **tree protocol**, the only lock instruction allowed is lock-X. Each transaction *Ti* can lock a data item at most once, and must observe the following rules:
1. The first lock by *Ti* may be on any data item.
2. Subsequently, a data item *Q* can be locked by *Ti* only if the parent of *Q* is currently locked by *Ti*.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by *Ti* cannot subsequently be relocked by *Ti*.
All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of Figure 16.11.

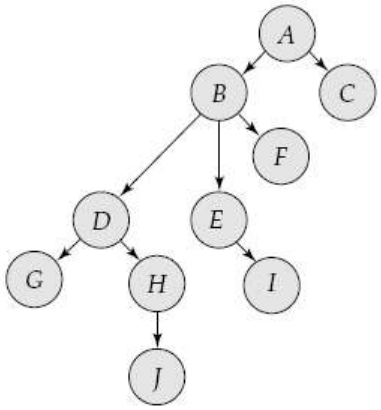

Figure 16.11 Tree-structured database graph.

The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

***T*10: lock-X(*B*); lock-X(*E*); lock-X(*D*); unlock(*B*); unlock(*E*); lock-X(*G*); unlock(*D*); unlock(*G*).**
***T*11: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).**
***T*12: lock-X(*B*); lock-X(*E*); unlock(*E*); unlock(*B*).**
***T*13: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).**

One possible schedule in which these four transactions participated appears in Figure 16.12. Note that, during its execution, transaction *T*10 holds locks on two *disjoint* sub-trees.

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Figure 16.12 Serializable schedule under the tree protocol.

Observe that the schedule of Figure 16.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock. The tree protocol in Figure 16.12 does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write we record which transaction performed the last write to the data item. Whenever a transaction *Ti* performs a read of an uncommitted data item, we record a **commit dependency** of *Ti* on the

transaction that performed the last write to the data item. Transaction *Ti* is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, *Ti* must also be aborted.
The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items *A* and *J* in the database graph of Figure 16.11 must lock not only *A* and *J*, but also data items *B*, *D*, and *H*. These additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly. For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that is not possible under the tree protocol, and vice versa.

Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

Timestamps –

With each transaction *Ti* in the system, we associate a unique fixed timestamp, denoted by TS(*Ti*). This timestamp is assigned by the database system before the transaction *Ti* starts execution. If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*). There are two simple methods for implementing this scheme:
1. Use the value of the *system clock* as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj* . To implement this scheme, we associate with each data item *Q* two timestamp values:
• **W-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
• **R-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed read(*Q*) successfully.

These timestamps are updated whenever a new read(*Q*) or write(*Q*) instruction is executed.

The Timestamp-Ordering Protocol –

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction *Ti* issues read(*Q*).
a. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten. Hence, the read operation is rejected, and *Ti* is rolled back.

b. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and Rtimestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

2. Suppose that transaction *Ti* issues write(Q).

a. If TS(Ti) < R-timestamp(Q), then the value of Q that *Ti* is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T back.

b. If TS(Ti) < W-timestamp(Q), then *Ti* is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls *Ti* back.

c. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

If a transaction *Ti* is rolled back by the concurrency-control scheme as result of issuance of either a read or writes operation, the system assigns it a new timestamp and restarts it. To illustrate this protocol, we consider transactions $T14$ and $T15$.

Transaction *T*14 displays the contents of accounts *A* and *B*:

$T14$: read(B);
read(A);
display(A + B).

Transaction *T*15 transfers $50 from account *A* to account *B*, and then displays the contents of both:

$T15$: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display(A + B).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 16.13, TS($T14$) < TS($T15$), and the schedule is possible under the timestamp protocol.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Figure 16.13 Schedule 3.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short

transactions causes repeated restarting of the long transaction. If a transaction is found to be getting restarted repeatedly, conflicting transactions need to be temporarily blocked to enable the transaction to finish. The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

• Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.
• Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits
• Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction *Ti* to commit only after the commit of any transaction that wrote a value that *Ti* read. Commit dependencies.

Multiple Granularity –

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction *Ti* needs to access the entire database, and a locking protocol is used, then *Ti* must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if *Ti* could issue a *single* lock request to lock the entire database. On the other hand, if transaction *Tj* needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

Multiple levels of **granularity** allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. A non-leaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

Granularity is a measure of the size of the components, or descriptions of components, that make up a system. Granularity is the relative size, scale, level of detail or depth of penetration that characterizes an object or activity. It is the "extent to which a larger entity is subdivided. For example, a yard broken into inches has finer granularity than a yard broken into feet

As an illustration, consider the tree of Figure 16.16, which consists of four levels of nodes.

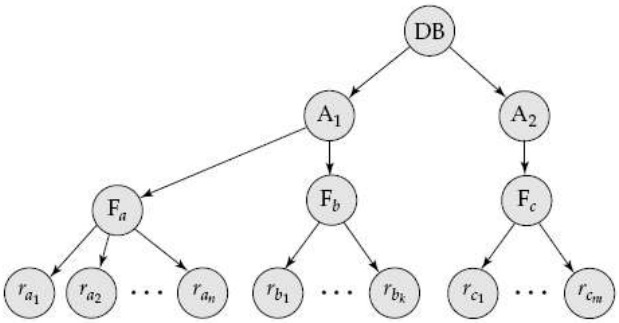

Figure 16.16 Granularity hierarchy.

The highest level represents the entire database. Below it is the nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file. Each node in the tree can be locked individually. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction *Ti* gets an **explicit lock** on file *Fc* of Figure 16.16, in exclusive

mode, then it has an **implicit lock** in exclusive mode all the records belonging to that file. It does not need to lock the individual records of *Fc* explicitly.

Suppose that transaction *Tj* wishes to lock record *rb*6 of file *Fb*. Since *Ti* has locked *Fb* explicitly, it follows that *rb*6 is also locked (implicitly). But, when *Tj* issues a lock request for *rb*6 , *rb*6 is not explicitly locked! How does the system determine whether *Tj* can lock *rb*6? *Tj* must traverse the tree from the root to record *rb*6. If any node in that path is locked in an incompatible mode, then *Tj* must be delayed. Suppose now that transaction *Tk* wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. However, *Tk* should not succeed in locking the root node, since *Ti* is currently holding a lock on part of the tree (specifically, on file *Fb*).

This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, *Q*—must traverse a path in the tree from the root to *Q*. While traversing the tree, the transaction locks the various nodes in an intention mode. There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.

Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in Figure 16.17.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 16.17 Compatibility matrix.

The **multiple-granularity locking protocol**, which ensures serializability, is this:
Each transaction *Ti* can lock a node *Q* by following these rules:

1. It must observe the lock-compatibility function of Figure 16.17.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node *Q* in S or IS mode only if it currently has the parent of *Q* locked in either IX or IS mode.
4. It can lock a node *Q* in X, SIX, or IX mode only if it currently has the parent of *Q* locked in either IX or SIX modes.
5. It can lock a node only if it has not previously unlocked any node (that is, *Ti* is two phase).
6. It can unlock a node *Q* only if it currently has none of the children of *Q* locked. Observe that the multiple-granularity protocol requires that locks be acquired in *topdown* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure 16.16 and these transactions:

• Suppose that transaction *T*18 reads record *ra*2 in file *Fa*. Then, *T*18 needs to lock the database, area *A*1, and *Fa* in IS mode (and in that order), and finally to lock *ra*2 in S mode.
• Suppose that transaction *T*19 modifies record *ra*9 in file *Fa*. Then, *T*19 needs to lock the database, area *A*1, and file *Fa* in IX mode, and finally to lock *ra*9 in X mode.
• Suppose that transaction *T*20 reads all the records in file *Fa*. Then, *T*20 needs to lock the database and area *A*1 (in that order) in IS mode, and finally to lock *Fa* in S mode.
• Suppose that transaction *T*21 reads the entire database. It can do so after locking the database in S mode.

We note that transactions *T*18, *T*20, and *T*21 can access the database concurrently. Transaction *T*19 can execute concurrently with *T*18, but not with either *T*20 or *T*21. This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

• Short transactions that access only a few data items

• Long transactions that produce reports from an entire file or set of files

There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph.

Database Integrity and Security –

Database Integrity –

It is the mechanism that is applied to ensure that the data in the database is correct and consistent. The term **semantic integrity** is sometimes used to refer to the need for maintaining database consistency in the presence of user modifications. Semantic integrity is maintained either implicitly by the data model, or is specified explicitly by appropriate constraints on the data that can be entered by the user, and this data is checked by the DBMS. Entity and referential integrity constraints are implicit in the relational data model. Set insertion and retention rules are implicit in the network model.

Database Security –

Protection of the information contained in the database against unauthorized access, modification, or destruction.

Authorization –

The culmination of the administrative policies of the organization, expressed as a set of rules that can be used to determine which user has what type of access to which portion of the database. Persons who are in charge of specifying the authorization of different portions of the database are usually called **security administrators** or **authorizers.**

Integrity constraints specifications in SQL –

Data Constraint

A constraint is a limitation that you place on the data that users can enter into a column or group of columns. A constraint is part of the table definition; you can implement constraints when you create the table or later. You can remove a constraint from the table without affecting the table or the data, and you can temporarily disable certain constraints. **. . .**

Types Of Constraints-

- I/O Constraint:
  - Primary Key Constraint
  - Foreign Key Constraint
  - Unique Key Constraint

- Business Rule Constraint
  - Check Constraint
  - Not Null Constraint

Primary Key

The primary key of a relational table uniquely identifies each record in the table. Primary keys may consist of a single attribute or multiple attributes in combination.

Syntax:

CREATE TABLE <TableName>(<ColumnName> <DataType>(<Size>) PRIMARY KEY,…….);

Ex:

- CREATE TABLE Customer
  (SID integer PRIMARY KEY,
  Last_Name varchar(30),
  First_Name varchar(30));

- ALTER TABLE Customer ADD PRIMARY KEY (SID);

Foreign Key

A foreign key is a field (or fields) that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

Syntax:

<ColumnName> <DataType>(<Size>)
REFERENCES <TableName>[(<ColumnName>)]…

Ex:

- CREATE TABLE ORDERS

(Order_ID integer primary key,
Order_Date date,
Customer_SID integer references CUSTOMER(SID),
Amount double);

- ALTER TABLE ORDERS

ADD (CONSTRAINT fk_orders1) FOREIGN KEY (customer_sid) REFERENCES CUSTOMER(SID);

Unique Key

Des:

• In relational database design, a **unique key** or **primary key** is a candidate key to uniquely identify each row in a table. A unique key or primary key comprises a single column or set of columns. No two distinct rows in a table can have the same value (or combination of values) in those columns. Depending on its design, a table may have arbitrarily many unique keys but at most one primary key.

• A unique key must uniquely identify all *possible* rows that exist in a table and not only the currently existing rows.

Syntax:

<ColumnName> <DataType>(<Size>) UNIQUE

Difference Primary / Unique Key

1) Unique key can be null but primary key cant be null.
2) Primary key can be referenced to other table as FK.
3) We can have multiple unique key in a table but PK is one and only one.
4) PK in itself is unique key.

Check Constraint

Des:

A check constraint allows you to specify a condition on each row in a table.

Note:

- A check constraint can NOT be defined on a VIEW.
- The check constraint defined on a table must refer to only columns in that table. It can not refer to columns in other tables.
- A check constraint can NOT include a SUBQUERY.

Ex:

- CREATE TABLE suppliers(supplier_idnumeric(4),supplier_namevarchar2(50),CONSTRAINT check_supplier_idCHECK (supplier_id BETWEEN 100 and 9999));
- ALTER TABLE suppliers
  add CONSTRAINT check_supplier_name
  CHECK (supplier_name IN ('IBM', 'Microsoft', 'Nvidia'));

Using The Not Null Constraint

Des:

- Use the NOT NULL keywords to require that a column receive a value during insert or update operations. If you place a NOT NULL constraint on a column (and no default value is specified), you *must* enter a value into this column when you insert a row or update that column in a row. If you do not enter a value, the database server returns an error, because no default value exists.
- The following example creates the **newitems** table. In **newitems**, the column **menucode** does not have a default value nor does it allow NULL values.

Ex: CREATE TABLE newitems (newitem_num INTEGER, menucode CHAR(3) NOT NULL, promotype INTEGER, descrip CHAR(20))

Note:

You cannot specify NULL as the explicit default value for a column if you also specify the NOT NULL constraint.