Principles of Database Systems

Transactions

Mission

- The concept of a transaction in detail, including the properties of atomicity, durability, isolation, and other properties provided by the transaction abstraction.
- Several concurrency-control techniques that help implement the isolation property.
- The recovery management component of a database, which implements the atomicity and durability properties.

Transactions

- Collections of operations that form a single logical unit of work are called **transactions**.
- For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.
- It is important that either all actions of a transaction be executed completely, or, in case of some failure, partial effects of each incomplete transaction be undone.

Properties of the transactions

- These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.
- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

Properties of the transactions

- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions *Ti* and *Tj* , it appears to *Ti* that either *Tj* finished execution before *Ti* started or *Tj* started execution after *Ti* finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Properties of the transactions

- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction Committed

- A transaction that completes its execution successfully is said to be **committed**.
- We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

A transaction must be in one of the following states

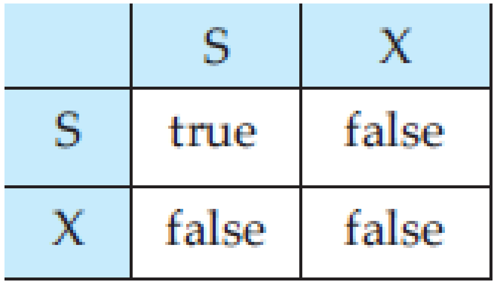
- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

Concurrency Control

- **Lock-Based Protocols**
- One way to ensure isolation 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

- **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*.



	S	X
S	true	false
X	false	false

Figure 15.1 Lock-compatibility matrix comp.

Locks

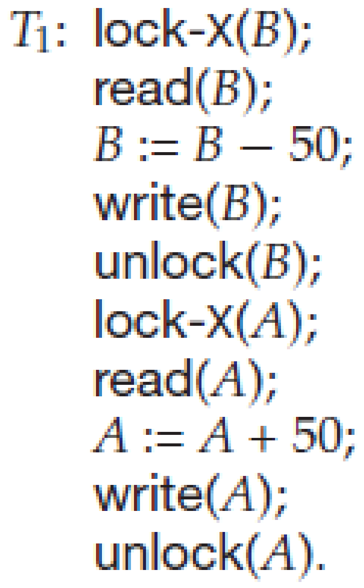
- A transaction requests a shared lock on data item *Q* by executing the lock-S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

Locks

- 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.

Example

- Let A and B be two accounts that are accessed by transactions $T1$ and $T2$. Transaction $T1$ transfers \$50 from account B to account A.

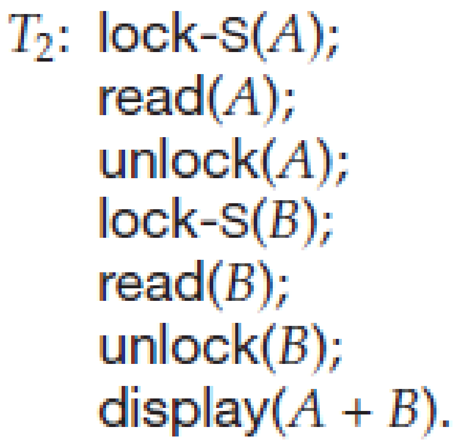


```
$T_1$: lock-X($B$);
       read($B$);
       $B := B - 50$;
       write($B$);
       unlock($B$);
       lock-X($A$);
       read($A$);
       $A := A + 50$;
       write($A$);
       unlock($A$).
```

Figure 15.2 Transaction T_1.

Example

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



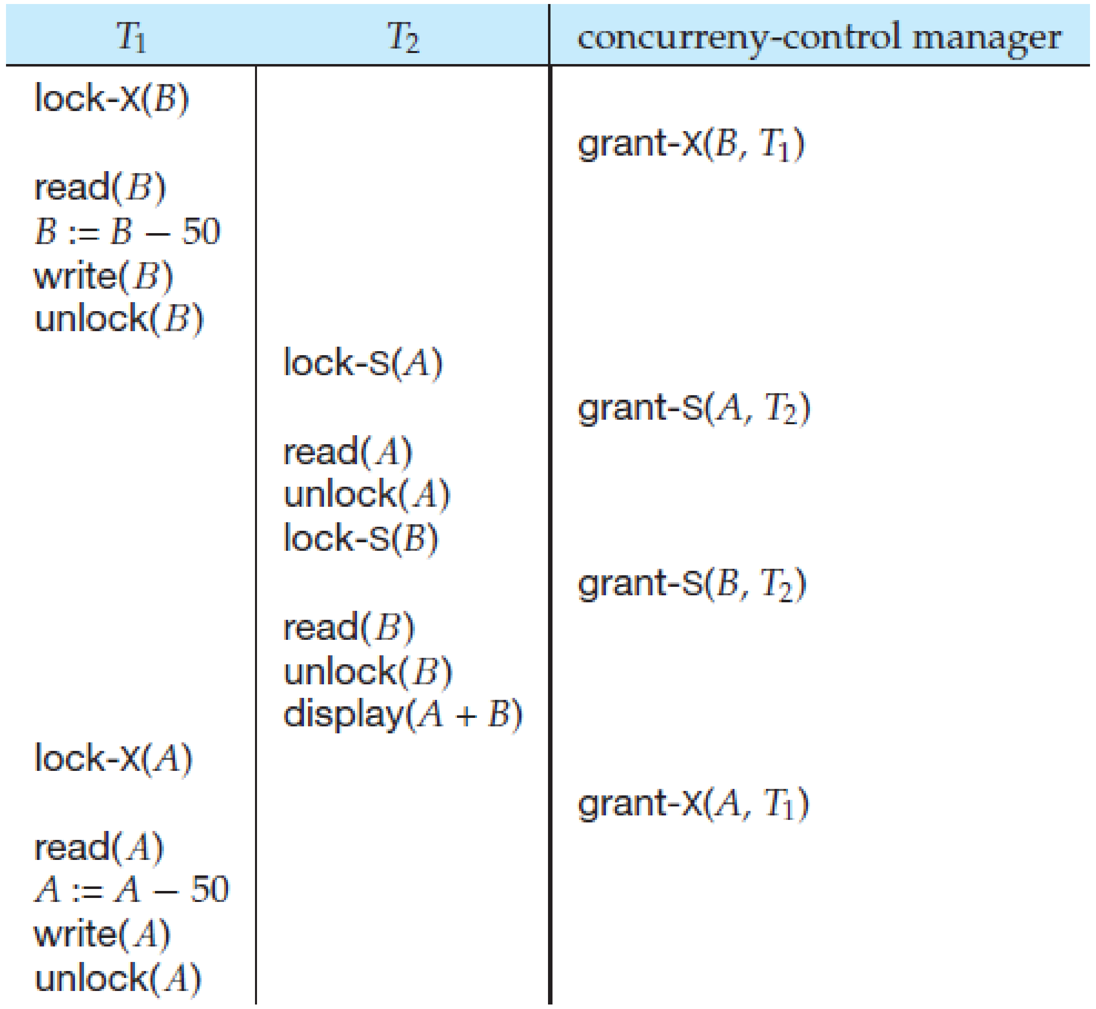
```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 15.3 Transaction T_2.

Example

- 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 $T1$, $T2$ or the order $T2$, $T1$, then transaction $T2$ will display the value \$300.
- If, however, these transactions are executed concurrently, then schedule 1, in Figure 15.4, is possible. In this case, transaction $T2$ displays \$250, which is incorrect. The reason for this mistake is that the transaction $T1$ unlocked data item B too early, as a result of which $T2$ saw an inconsistent state.

Example



T_1	T_2	concurreny-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_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Figure 15.4 Schedule 1.

Example

- Suppose now that unlocking is delayed to the end of the transaction. Transaction *T*3 corresponds to *T*1 with unlocking delayed (Figure 15.5). Transaction *T*4 corresponds to *T*2 with unlocking delayed (Figure 15.6).
- You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of $250 being displayed, is no longer possible with *T*3 and *T*4. Other schedules are possible. *T*4 will not print out an inconsistent result in any of them.

Example

```
T3: lock-X(B);
    read(B);
    $B := B - 50$;
    write(B);
    lock-X(A);
    read(A);
    $A := A + 50$;
    write(A);
    unlock(B);
    unlock(A).
```

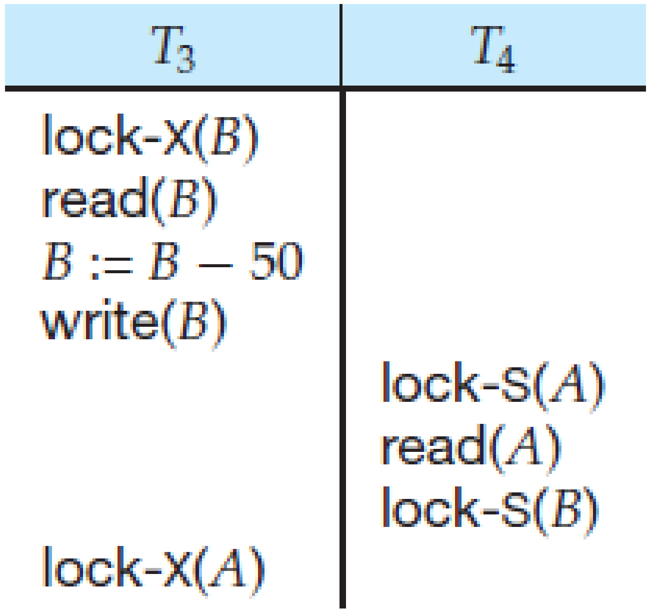
Figure 15.5 Transaction T_3 (transaction T_1 with unlocking delayed).

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display($A + B$);
    unlock(A);
    unlock(B).
```

Figure 15.6 Transaction T_4 (transaction T_2 with unlocking delayed).

Deadlock

- Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 15.7 for *T*3 and *T*4.



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 15.7 Schedule 2.

Deadlock

- 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*. 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**.

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.

Deadlock Handling

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T0, T1, \ldots, Tn\}$ such that $T0$ is waiting for a data item that $T1$ holds, and $T1$ is waiting for a data item that $T2$ holds, and . . . , and $Tn-1$ is waiting for a data item that Tn holds, and Tn is waiting for a data item that $T0$ holds. None of the transactions can make progress in such a situation.

Deadlock Handling

- The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

Deadlock Prevention

- 1. The **wait–die** scheme is a nonpreemptive technique. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp smaller than that of *Tj* (that is, *Ti* is older than *Tj*). Otherwise, *Ti* is rolled back (dies).

Deadlock Prevention

- 2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp larger than that of *Tj* (that is, *Ti* is younger than *Tj*). Otherwise, *Tj* is rolled back (*Tj* is *wounded* by *Ti*).

Deadlock Detection and Recovery

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection

- Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system.
- Each element in the set E of edges is an ordered pair $Ti \rightarrow Tj$. If $Ti \rightarrow Tj$ is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.

Wait-for graph

- When transaction Ti requests a data item currently being held by transaction Tj, then the edge $Ti \rightarrow Tj$ is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti.

Wait-for graph

- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

Wait-for graph

- Transaction $T17$ is waiting for transactions $T18$ and $T19$.
- Transaction $T19$ is waiting for transaction $T18$.
- Transaction $T18$ is waiting for transaction $T20$.

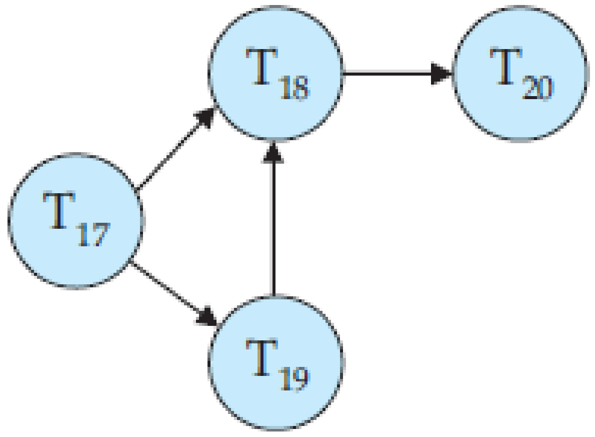


Figure 15.13 Wait-for graph with no cycle.

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.

Multiple Granularity

- 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. What is needed is a mechanism to allow the system to define multiple levels of **granularity**. This is done by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones.

Shared(S) and exclusive(X) lock modes

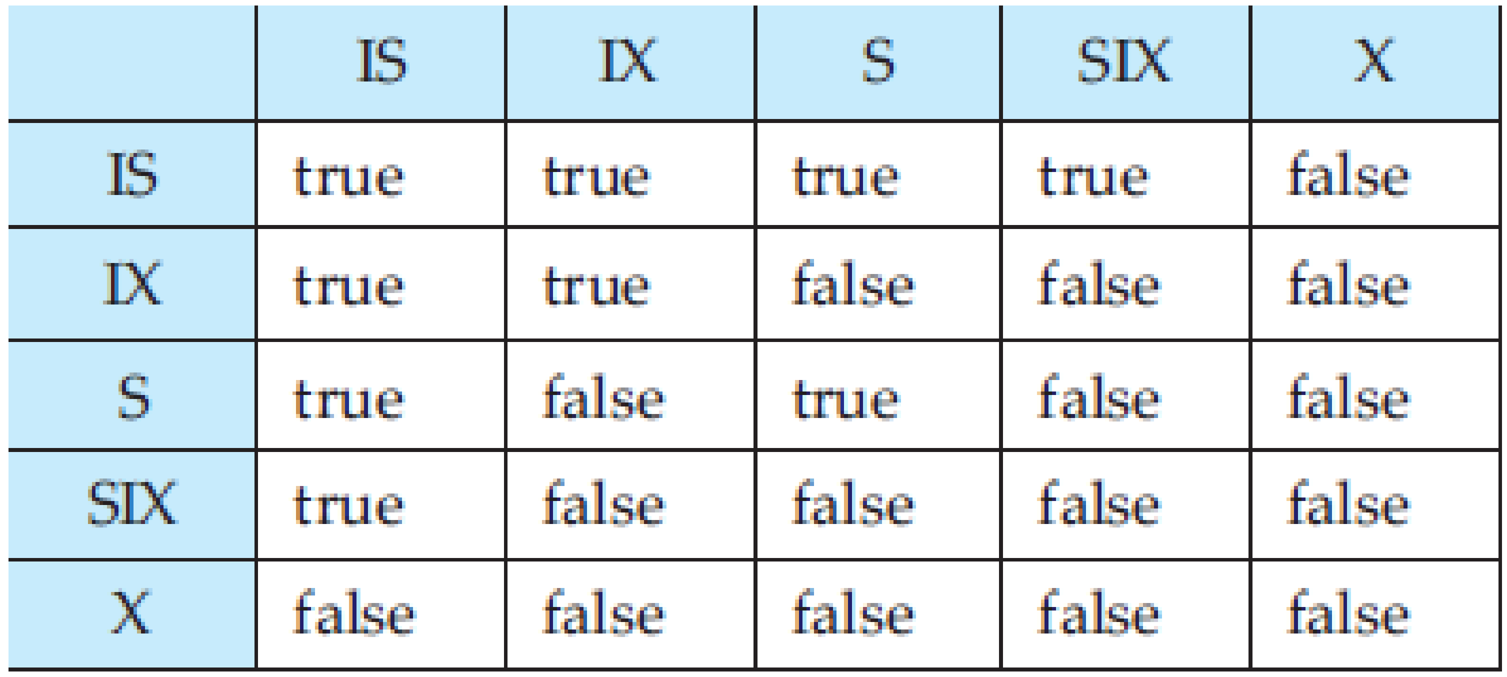
- When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode.

IS and IX 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.

SIX mode

- 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.



	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 15.16 Compatibility matrix.

Multiple-granularity locking protocol

- **1.** Transaction *Ti* must observe the lock-compatibility function of Figure 15.16.
- **2.** Transaction *Ti* must lock the root of the tree first, and can lock it in any mode.
- **3.** Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.

Multiple-granularity locking protocol

- **4.** Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
- **5.** Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).
- **6.** Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.

Recovery System

- An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.
- The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a failure.

Failure Classification

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

(1)**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.

(2)**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 reexecuted at a later time.

Failure Classification

- **2. 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.

Failure Classification

- **3. 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 DVD or tapes, are used to recover from the failure.

Storage Structure

- Storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as **volatile storage** or **nonvolatile storage**.
- And there is another class of storage, called **stable storage**.

Storage Structure

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

Storage Structure

- **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of nonvolatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage. At the current state of technology, nonvolatile storage is slower than volatile storage, particularly for random access.

Storage Structure

- **Stable storage**. Information residing in stable storage is *never* lost. Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information.

Thank you!