Module 5

Transaction and System Concepts

20.2.1 Transaction states and Additional Operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts. Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

- **BEGIN_TRANSACTION:** This marks the beginning of transaction execution.
- **READ or WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSACTION:** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.
- **COMMIT_TRANSACTION:** This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **ROLLBACK (or ABORT):** This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction that may have applied to the database must be undone.

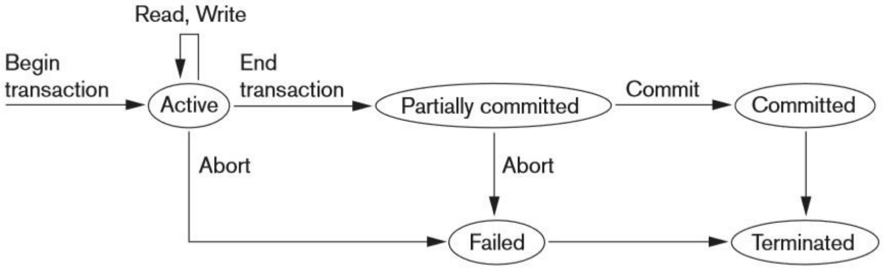


Fig: State transition diagram illustrating the states for transaction execution

The above figure shows a state transition diagram that illustrates how a transaction moves through its execution states.

- A transaction goes into an active state immediately after it starts execution, where it can execute its **READ** and **WRITE** operations.
- When the transaction ends, it moves to the partially committed state. At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not.

- Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.
- If these checks are successful, the transaction is said to have reached its commit point and enters the committed state.
- When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.
- However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its **WRITE** operations on the database. The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. **Failed or aborted transactions may be restarted later**—either automatically or after being resubmitted by the user—as brand new transactions.

20.2.2 The System Log

- To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures.
- **The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.**
- Typically, one or more main memory buffers, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the log main memory buffer.
- When the log buffer is filled, or when certain other conditions occur, the log buffer is appended to the end of the log file on disk. In addition, the log file from disk is periodically backed up to archival storage to guard against catastrophic failures.
- The following are the types of entries—**called log records**—that are written to the log file and the corresponding action for each log record. In these entries, **T refers to a unique transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:
    - **[start_transaction, T]:** Indicates that transaction T has started execution.
    - **[write_item, T, X, old_value, new_value]:** Indicates that transaction T has changed the value of database item X from old_value to new_value.
    - **[read_item, T, X]:** Indicates that transaction T has read the value of database item X.
    - **[commit, T]:** Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - **[abort, T]:** Indicates that transaction T has been aborted.

20.2.3 Commit Point of a Transaction

- A transaction T reaches its **commit point** when all its operations that access the database have been **executed successfully** and the effect of all the transaction operations on the database have been recorded in the log.
- The transaction then writes a **commit record [commit, T]** into the log. If a system failure occurs,

we can search back in the log for all transactions T that have written a **[start_transaction, T]** record into the log but have not written their **[commit, T]** record yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

- Transactions that have written their commit record in the log must also have recorded all their **WRITE** operations in the log, so their effect on the database can be **redone** from the log records.
- The log file must be kept on disk. Updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk.
- At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process if the contents of main memory are lost. Hence, before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. **This process is called force-writing the log buffer to disk before committing a transaction.**

Desirable Properties of Transaction:

Transactions should possess several properties, often called **the ACID properties**, they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

- **Atomicity:** A transaction is an atomic unit of processing, it should either be performed in its entirety or not performed at all.
  - ➢ It requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.
  - ➢ If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.
  - ➢ On the other hand, write operations of a committed transaction must be eventually written to disk.
- **Consistency preservation:** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state toanother.
  - ➢ It is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints.
  - ➢ A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the complete execution of the transaction, assuming that no interference with other transactions occurs.
- **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
  - ➢ It is enforced by the concurrency control subsystem of the DBMS.
  - ➢ If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the

temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems.

- **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.
  - ➢ It is the responsibility of the recovery subsystem of the DBMS.
- **Levels of Isolation:** There have been attempts to define the level of isolation of a transaction.
  - ➢ A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions.
  - ➢ Level 1 isolation has no lost updates.
  - ➢ Level 2 isolation has no lost updates and no dirty reads.
  - ➢ Level 3 isolation (also called true isolation) has repeatable reads.
  - ➢ Another type of isolation is called **snapshot isolation**, and several practical concurrency control methods are based on this.

20.3 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule (or history).**

20.3.1 Schedules of Transactions

- A **schedule** (or **history**) *S* of *n* transactions T_1, T_2, ..., T_n is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule *S*. However, for each transaction T_i that participates in the schedule *S*, the operations of T_i in *S* must appear in the same order in which they occur in T_i. The order of operations in *S* is considered to be a *total ordering*,

  meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.
- The purpose of recovery and concurrency control, we are mainly interested in the read_item and write_item operations of the transactions, as well as the commit and abort operations. A shorthand notation for describing a schedule uses the symbols *b*, *r*, *w*, *e*, *c*, and *a* for the operations begin_transaction, read_item,write_item, end_transaction, commit, and abort, respectively, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule.
- **example**, the schedule in which we shall call S_a, can be written as follows in this notation:

  S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

  Similarly, the schedule for fig. which we call S_b, can be written as follows, if we assume that transaction T1 aborted after its read_item(Y) operation:

  S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;

Conflicting Operations in a Schedule:

- Two operations in a schedule are said to conflict if they satisfy all three of the following conditions: (1) they belong to different transactions; (2) they access the same item X; and (3) at least one of the operations is a write_item(X).
- Intuitively, two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations r1(X); w2(X) to w2(X); r1(X), then the value of X that is read by transaction T1 changes, because in the second ordering the value of X is read by r1(X) after it is changed by w2(X), whereas in the first ordering the value is read before it is changed. This is called a **read-write conflict**. The other type is called a **write-write conflict**.

20.4.2 Characterizing Schedules Based on Recoverability

It is easy to recover from transaction and system failures. In some cases, it is even not possible to recover correctly after a failure. it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple.

once a transaction T is committed, it should never be necessary to roll back T. The schedules that theoretically meet this criterion are called **recoverable schedules**. A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS.

A recovery algorithm can be devised for any recoverable schedule. The (partial) schedules Sa and Sb from the preceding section are both recoverable. Consider the schedule Sa’ given below, which is the same as schedule Sa except that two commit operations have been added to Sa: Sa’: r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;

Sa’ is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory.

consider the two (partial) schedules Sc and Sd that follow: Sc: r1(X);

w1(X); r2(X); r1(Y); w2(X); c2; a1;

Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;

Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;

Sc is not recoverable because T2 reads item X from T1, but T2 commits before T1 commits. The problem occurs if T1 aborts after the c2 operation in Sc; then the value of X that T2 read is no longer valid and T2 must be aborted after it is committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c2 operation in Sc must be postponed until after T1 commits, as shown in Sd. If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of X it read is no longer valid. In Se, aborting T2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule Sc.

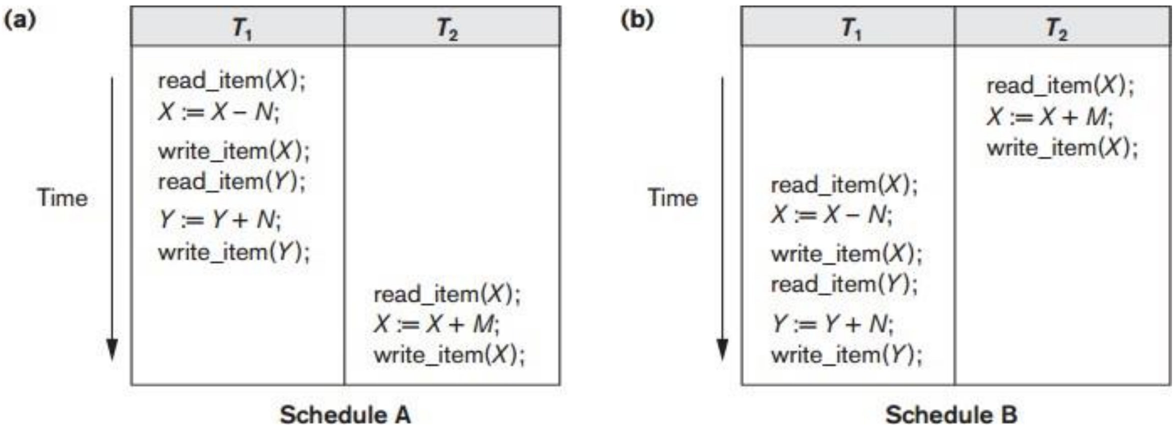
In **Cascadless Schedules** If every Xn in the scheduled reads only items that written by committed Xn. **strict schedule**, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted). For example, consider schedule Sf:

Sf: w1(X, 5); w2(X, 8); a1;

Notes:-Any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable.

20.4 Characterizing Schedules Based on Serializability

- The Schedules that are always considered to be correct when concurrent transactions are executing. Such Schedules are known as ***serializable schedules.***
- If no interleaving of operations are permitted there are only two possible arrangements for executing transactions T1 and T2:
  - Execute (in sequence) all the operations of transaction T1, followed by all the operations of transaction T2.
  - Execute (in sequence) all the operations of transaction T2, followed by all the operations of transaction T1.
- If interleaving of operations is allowed there will be many possible schedules.
- The concept of serializability of schedules is used to identify which schedules are correct.


(a)
Time
(b)
Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

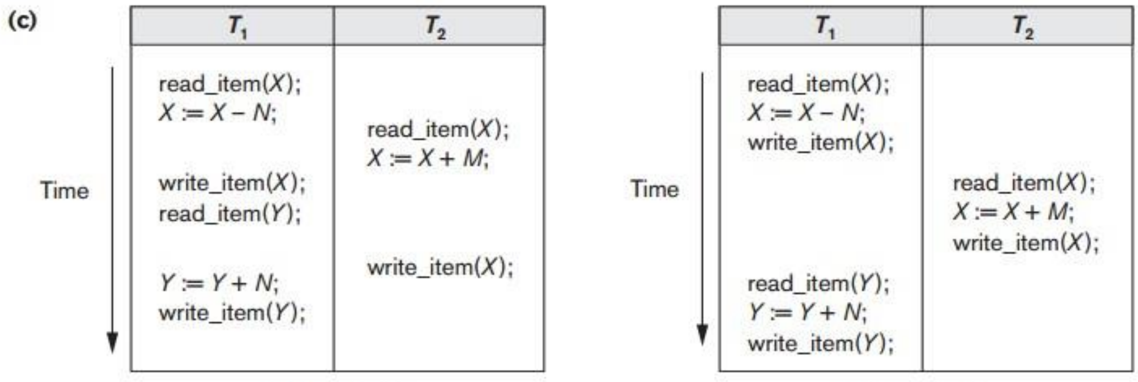
Schedule A

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

Fig: Example of serial and non serial schedules involving transactions T1 and T2.

- Serial schedule A: T1 followed by T2.
- Serial schedule B: T2 followed by T1.



(c)

Time	T_1	T_2
↓	read_item(X); $X := X - N$;	
		read_item(X); $X := X + M$;

	write_item(X);
	read_item(Y);	

		write_item(X);
	$Y := Y + N$; write_item(Y);	

Schedule C

Time	T_1	T_2

↓	read_item(X);
	$X := X - N$;
	write_item(X);	

		read_item(X);
		$X := X + M$;
		write_item(X);

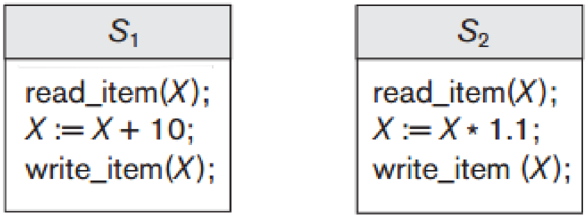
	read_item(Y);
	$Y := Y + N$;
	write_item(Y);	

Schedule D

Fig: Two non-serial schedules C and D with interleaving of operations.

20.4.1 Serial, Non-serial, and Conflict-Serializable Schedules

- A schedule S is **serial**, if for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise the schedule is called **non-serial**.
- In a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule.
- The drawback of serial schedules is that they limit concurrency of interleaving of operations.
- Two schedules are called **result equivalent** if they produce the same final state of the database.



S_1

read_item(X);
$X := X + 10$;
write_item(X);

S_2

read_item(X);
$X := X * 1.1$;
write_item (X);

Fig: Two schedules that are result equivalent for the initial value X=100, but are not result equivalent in general.

- Two definitions of equivalence of schedules are generally used: *conflict equivalence* and *view equivalence*.
    - o Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item.
- Two schedules are said to be **conflict equivalent** if the order of any two *conflicting operations* is the same in both schedules.
- A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions.
- Using conflict equivalence, we define a schedule *S* to be **conflict serializable** if it is conflict-

equivalent to some serial schedule *S*. In such a case, we can reorder the *non- conflicting* operations in *S* until we form the equivalent serial schedule *S*.

Consider the following schedule for a set of three transactions.

$w_1(A), w_2(A), w_2(B), w_1(B), w_3(B)$

We cannot perform reordering on this: The first two operations are both on *A* and at least one is a write; the second and third operations are by the same transaction; the third and fourth are both on *B* at at least one is a write; and so are the fourth and fifth. So this schedule is not conflict-equivalent to anything else — and certainly not any serial schedules.

However, since nobody ever reads the values written by the $w1(A)$, $w2(B)$, and $w1(B)$ operations, the schedule has the same outcome as the serial schedule:

$w1(A), w1(B), w2(A), w2(B), w3(B)$

20.4.2 Testing for Conflict Serializability of a Schedule

Using the definition of conflict-serializability to show that a schedule is conflict- serializable is quite difficult. There's a much more efficient algorithm:

The algorithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph** $G = (N, E)$that consists of a set of nodes $N = \{T1, T2, ..., Tn\}$ and a set of directed edges $E = \{e1, e2, ..., em\}$. There is one node in the graph for each transaction T*i* in the schedule. Each edge e*i* in the graph is of the form (Tj → Tk), $1 \leq j \leq n$, $1 \leq k$ f n, where *Tj* is the **starting node** of *ei* and *Tk* is the **ending node** of e*i*. Such an edge from node *Tj* to node *Tk* is created by the algorithm if one of the operations in *Tj* appears in the schedule before some *conflicting operation* in *Tk*.

Algorithm: Testing Conflict Serializability of a Schedule *S*

1. For each transaction *Ti* participating in schedule *S,* create a node labeled *Ti* in the precedence graph.
2. For each case in *S* where *Tj* executes a read_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti* → *Tj*) in the precedence graph.
3. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a read_item(*X*), create an edge (*Ti* → *Tj*) in the precedence graph.
4. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti* → *Tj*) in the precedence graph.
5. The schedule *S* is serializable if and only if the precedeence graph has no cycles.

As an example, consider the following schedule:

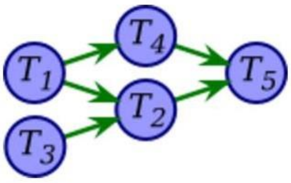
$w1(A), r2(A), w1(B), w3(C), r2(C), r4(B), w2(D), w4(E), r5(D), w5(E)$

Step1. We start with an empty graph with five vertices labeled T_1, T_2, T_3, T_4, T_5.

Step 2 and 3.

$w1(A)$: A is subsequently read by $T2$, so add edge $T1$ → $T2$
$r2(A)$: no subsequent writes to A, so no new edges
$w1(B)$: B is subsequently read by $T4$, so add edge $T1$ → $T4$
$w3(C)$: C is subsequently read by $T2$, so add edge $T3$ → $T2$
$r2(C)$: no subsequent writes to C, so no new edges
$r4(B)$: no subsequent writes to B, so no new edges
$w2(D)$: C is subsequently read by $T2$, so add edge $T3$ → $T2$
$w4(E)$: E is subsequently written by $T5$, so add edge $T4$ → $T5$
$r5(D)$: no subsequent writes to D, so no new edges
$w5(E)$: no subsequent operations on E, so no new edges

Step 4: Creating precedence graph.



Step 5.This graph has no cycles, so the original schedule must be serializable.

Moreover, since one way to topologically sort the graph is $T3$–$T1$–$T4$–$T2$–$T5$, one serial schedule that is conflict-equivalent is

$w3(C)$, $w1(A)$, $w1(B)$, $r4(B)$, $w4(E)$, $r2(A)$, $r2(C)$, $w2(D)$, $r5(D)$, $w5(E)$

20.4.3 How Serializibilty is used for Concurrency Control

The approach taken in most commercial DBMS is to design protocols that if followed by every individual transaction or if enforced by a DBMS concurrency control subsystem will ensure serializibility of all schedules in which the transaction participate.

When transaction are submitted continuously the system finds difficulty to determine when to start or end transaction. Serializibility theory can be adapted to deal with this problem by considering only the committed projection of schedules committed projection C(s) of a schedule S includes only the operations in S that belong to committed transactions.

Concurrency protocols:

1. Two phase locking
2. Time stamp ordering
3. Multi-version protocol
4. Optimistic protocols

Two phase locking: Locking the data items to prevent concurrent terms from interfering with one another and enforcing serializibility.

Time Stamp ordering: Where transaction is assigned a unique timestamp and ensures that any conflicting operations are executed in order of the transaction time stamp.

Multi-version protocol: Maintaining multiple versions of data items.

Optimistic protocol: Check possible serializibility violations after the transactions terminate but before they are permitted to commit.

20.4.4 View Equivalence and view Serializibility

Two schedules S and S′ are said to be view equivalent of the following three conditions holds.

1. The same set of transactions participate in S and S′, Sand S′ include the same operations of those transactions.
2. For any operations $r_1(x)$ of T_i in S , if the value of X read by the operation has taken written by an operation $w_j(x)$ of T_j the same condition must hold for the value of X read by operation $r_i(x)$ of T_i inS′.
3. If the operation $w_k(y)$ of T_k is the last operation to write item Y in S , then $w_k(y)$ of T_k must also be the last operation to write item Y in S′.

A schedule S is said to be **view serializable** if its view equivalent to serial schedule.

The definitions of conflict serializibility and view serializibilty are similar if a condition known as the constrained write assumption holds on all transactions in the schedule.

The definition of view serializibilty is less restrictive than that of conflict serializibilty under the unconstrained write assumption where the value written by an operation $w_i(x)$ in T_i can be independent of its old value from database.

Example: *Sg* of three transactions *T*1: *r*1(*X*); *w*1(*X*); *T*2: *w*2(*X*); and *T*3:*w*3(*X*): *Sg*: *r*1(*X*); *w*2(*X*); *w*1(*X*); *w*3(*X*); *c*1; *c*2; *c*3;
In *Sg* the operations *w*2(*X*) and *w*3(*X*) are blind writes, since *T*2 and *T*3 do not read the value of *X*. The schedule *Sg* is view serializable, since it is view equivalent to the serial schedule *T*1, *T*2, *T*3. However, *Sg* is not conflict serializable, since it is not conflict equivalent to any serial schedule.

20.5.5 Other Types of Equivalence of Schedules

Some applications can produce schedules that are correct by satisfying conditions less stringent than either conflict serializibilty or view serializibilty.

An example is the type of transactions known as debit-credit transactions, for example those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. They update the values of X by adding or subtracting X. Consider the following transactions each of which may be used to transfer an amount of money between two bank accounts:

*T*1: *r*1(*X*); *X* := *X* − 10; *w*1(*X*); *r*1(*Y*); *Y* := *Y* + 10; *w*1(*Y*);

*T*2: *r*2(*Y*); *Y* := *Y* − 20; *w*2(*Y*); *r*2(*X*); *X* := *X* + 20; *w*2(*X*);

Consider the following non-serializable schedule *Sh* for the two transactions:

Sh: *r*1(*X*); *w*1(*X*); *r*2(*Y*); *w*2(*Y*); *r*1(*Y*); *w*1(*Y*); *r*2(*X*); *w*2(*X*);

With the additional knowledge, or **semantics**, that the operations between each

ri(*I*)and *wi*(*I*) are commutative, we know that the order of executing the sequences consisting of (read, update, write) is not important as long as each (read, update, write) sequence by a particular transaction *Ti* on a particular item *I* is not interrupted by conflicting operations. Hence, the schedule *Sh* is considered to be correct even though it is not serializable.

Introduction to Protocols for Concurrency Control in Databases

21.1 Two-Phase Locking Techniques for Concurrency Control:

The main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

21.1.1 Types of Locks and System Lock Tables: Locks are of two kinds −

- **Binary Locks** −
- **Shared/exclusive**

❖ <u>**Binary Locks**:</u> A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item X as lock(X).

```
lock_item(X):
B:
 if LOCK(X) = 0 (*item is unlocked*)
     then LOCK(X) ←1 (*lock the item*) else
     begin
     wait (until LOCK(X) = 0
          and the lock manager wakes up the transaction); go to B
     end;

unlock_item(X):
     LOCK(X) ← 0; (* unlock the item *)
     if any transactions are waiting then wakeup one of the waiting transactions;
```

- Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a lock_item(X) operation.
  If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the

transaction locks the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) back to 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item. When we use the binary locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

These rules can be enforced by the lock manager module of the DBMS.

- **Shared/Exclusive (or Read/Write) Locks:** This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is beingchanged.

read_lock(X):

```
B:
if LOCK(X) = “unlocked”
    then begin LOCK(X) ← “read-locked”;
    no_of_reads(X) ← 1
    end
else if LOCK(X) = “read-locked”
     then no_of_reads(X) ← no_of_reads(X) + 1 else
begin
     wait (until LOCK(X) = “unlocked”
     and the lock manager wakes up the transaction); go to B
    end;
```

write_lock(X):

```
 B:
if LOCK(X) = “unlocked”
      then LOCK(X) ← “write-locked” else
begin
     wait (until LOCK(X) = “unlocked”
        and the lock manager wakes up the transaction); go to B
      end;
```

unlock (X):

```
if LOCK(X) = “write-locked”
        then begin LOCK(X) ← “unlocked”;
        wakeup one of the waiting transactions, if any end
 else it LOCK(X) = “read-locked” then
        begin
                no_of_reads(X) ← no_of_reads(X) −1; if
                 no_of_reads(X) = 0
                        then begin LOCK(X) = “unlocked”;
                                    wakeup one of the waiting transactions, if any end
                   end;
```

Above algorithm shows Locking and unlocking operations for two mode (read/write, or shared/exclusive) locks.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

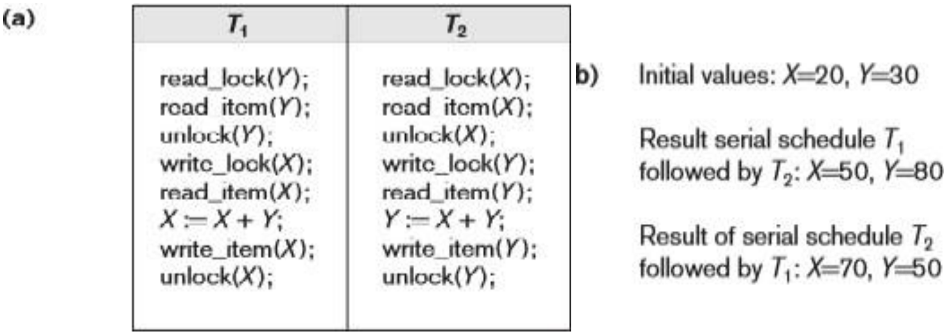
1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
4. A transaction T will not issue a read_lock (X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed for downgrading of locks, as we discuss shortly.
5. A transaction T will not issue a write_lock (X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed for upgrading of locks, as we discuss shortly.
6. A transaction T will not issue an unlock (X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

❖ **Conversion (Upgrading, Downgrading) of Locks:** It is desirable to relax conditions 4 and 5 in the preceding list in order to allow lock conversion; that is, a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.

21.1.2 Guaranteeing Serializability by Two-Phase Locking:

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases: an **expanding or growing (first) phase**, during which new locks on items can be acquired but none can be released; and **a shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

(a)



T_1	T_2

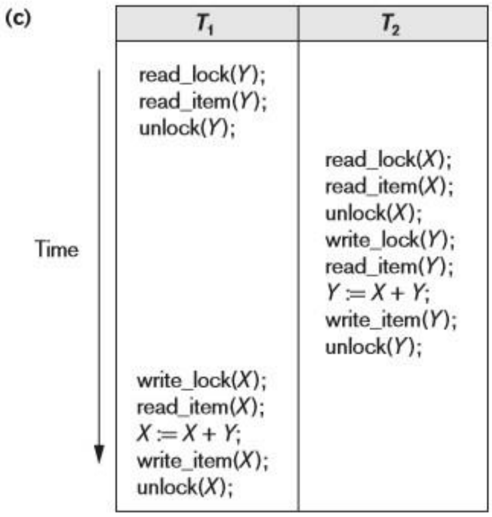
read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)



T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

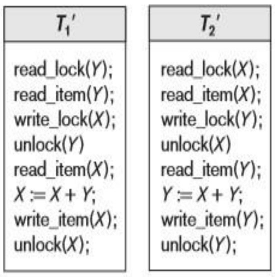
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Figure 21.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

Transactions T1 and T2 in Figure (a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2. If we enforce two-phase locking, the transactions can be
rewritten as T1′ and T2′, as shown in Figure (d). Now, the schedule shown in Figure (c) is not permitted for T1′ and T2′ (with their modified order of locking and unlocking operations) under the rules of locking described in Section 21.1.1 because T1′ will issue its write_lock(X) before it unlocks item Y; consequently, when T2′ issues its read_lock(X), it is forced to wait until T1′ releases the lock by issuing an unlock (X) in the schedule. However, this can lead to deadlock.

(d)



T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.

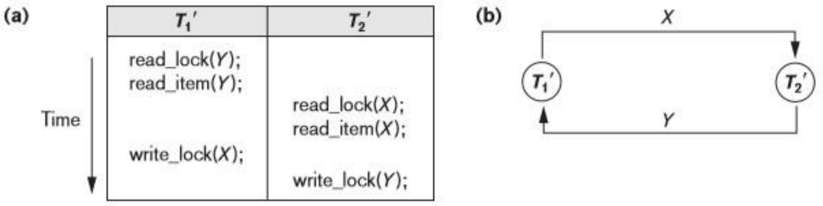
❖ **Basic, Conservative, Strict, and Rigorous Two-Phase Locking:**

- There are a number of variations of two-phase locking (2PL).
- The technique just described is known as **basic 2PL**.
- A variation known as **conservative 2PL** (**or static 2PL)** requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set.
- In practice, the most popular variation of 2PL is strict 2PL, which guarantees strict schedules.
- In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts.
- Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.
- Strict 2PL is not deadlock-free.
- A more restrictive variation of strict 2PL is rigorous 2PL, which also guarantees strict schedules.
- In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

❖ **Dealing with Deadlock and Starvation:**

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T′ in the set.
- Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- But because the other transaction is also waiting, it will never release the lock.

A simple example is shown in Figure 21.5(a), where the two transactions T1′ and T2′ are deadlocked in a partial schedule; T1′ is in the waiting queue for X, which is locked by T2′, whereas T2′ is in the waiting queue for Y, which is locked by T1′. Meanwhile, neither T1′ nor T2′ nor any other transaction can access items X and Y.



T_1'	T_2'

read_lock(Y);
read_item(Y);	

	read_lock(X);
	read_item(X);

write_lock(X);	
	write_lock(Y);

- Above Figure Illustrating the deadlock problem. (a) A partial schedule of T1′ and T2′ that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

- **Deadlock Prevention Protocols:**

One way to prevent deadlock is to use a deadlock prevention protocol.

- One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock all the items it needs in advance (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously, this solution further limits concurrency.
- A second protocol, which also limits concurrency, involves ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

- Two schemes that prevent deadlock are called wait-die and wound-wait.
  - **Wait-die:** In wait-die, an older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
  - **Wound-wait:** The wound-wait approach does the opposite: A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.

- Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms.
  - **No waiting algorithm:** In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly.
  - **Cautious waiting algorithm:** The cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts.

- **Deadlock Detection:**
  - An alternative approach to dealing with deadlock is deadlock detection, where the system checks if a state of deadlock actually exists.
  - This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time.
  - This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light.
  - On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is heavy, it may be advantageous to use a deadlock prevention scheme.

- **Timeouts:**
  - Another simple scheme to deal with deadlock is the use of timeouts.
  - This method is practical because of its low overhead and simplicity.
  - In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists.

- **Starvation:**
  - Another problem that may occur when we use locking is starvation, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
  - This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others.
  - One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue; transactions are enabled to lock an item in the order in which they originally requested the lock.
  - Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.
  - Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
  - The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem.
  - The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

21.2 Concurrency Control Based on Timestamp Ordering

21.2.1 Timestamp

- A timestamp is a unique identifier created by the DBMS to identify a transaction. Timestamp ordering do not use locks; hence, deadlocks cannot occur.
- Whenever a transaction begins, it receives a timestamp. This timestamp indicates the order in which the transaction must occur, relative to the other transactions.
- So, given two transactions that affect the same object, the operation of the transaction with the earlier timestamp must execute before the operation of the transaction with the later timestamp.
- However, if the operation of the wrong transaction is executed first, then it is aborted and the transaction must be restarted.
- Every object in the database has a **read timestamp**, which is updated whenever the object's data is read, and a **write timestamp**, which is updated whenever the object's data is changed.

Generation of time stamp

Timestamps can be generated in several ways.

1. To use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, … in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.
2. Another way is to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

21.2.2 Time Stamp Ordering Algorithm

- The idea for this scheme is to order the transactions based on their timestamps.
- A schedule in which the transactions participate is serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. This is called **timestamp ordering(TO)**.

  The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must not violate the serializability order.
- To do this, the algorithm associates with each database item X to timestamp (TS) VALUES:

  1. Read –TS (X): The read timestamp of item X; this is the largest time stamp among all the timestamps of transactions that have successfully read item – X i.e. Read – TS (X) = TS (T). Where T is the youngest transaction that has read X successfully.

  2. Write – TS (X): The write timestamp of item X; this is the largest of all the timestamps of transactions that have successfully written item X – i.e., write –TS (X) = TS(T), where T is the youngest transaction that has written X successfully.

Basic Timestamp Ordering (TO)

- Wherever some transaction T tries to issue a read- item (X) or write-item (X) operation, the basic TO (Time Ordering) compares the timestamp of T with read – TS (X) and write – TS (X) to ensure that the timestamp order of transaction is not violate.
- If this order is violated, then transaction is aborted resubmitted to the system as a new transaction with a new timestamp.
- If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not are not guaranteed to be recoverable.
- The concurrency control algorithm must check whether conflicting operations violate the time stamp ordering in the following two cases:

1. When a transaction T issues a write_item(X) operation, the following check is performed:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).
2. When a transaction T issues a read_item(X) operation, the following check is performed:
   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of item X before T had a chance to read X.
   b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Strict Timestamp Ordering (TO)

- A variation of basic TO is called Strict TO which ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable.

- In this variation, a transaction T that issues a read_item(X) or write_item(X) such that TS(T)>write_TS(X) has its read or write operation delayed until the transaction T1 that wrote the value of X (hence TS(T1) =write_TS(X)) has committed or aborted.

- To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T1 until T1 is either committed or aborted. This algorithm does not cause deadlock, since T waits T1 only if TS(T)>TS(T1).

Thomas Write Rule

A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability but it rejects some write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X)>TS(T) (read timestamp is greater than timestamp transaction), then abort and rollback T and reject the operation.
2. If Write_TS(X) > TS(T) (write timestamp is greater than timestamp transaction), then do not execute the write operation but continue processing. Because some transaction with a timestamp greater than TS(T) would have already written the value of X.
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

21.4 Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

In all concurrency control techniques we have discussed so far, a certain degree of checking is done before a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked.

In optimistic concurrency control techniques, also known as validation or certification techniques, no checking is done while the transaction is executing.

21.4.1 Validation-Based (Optimistic) Concurrency Control

In this schema, updates in the transaction are not applied directly to the database items on the disk until the transaction reaches its end and is validated. During transaction execution, all updates are applied to local copies of the data items that are kept for the transaction. At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise the transaction is aborted and restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase**. A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase**. Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase**. If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached.

21.5 Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

- ■ A database record
- ■ A field value of a database record
- ■ A disk block
- ■ A whole file
- ■ The whole database

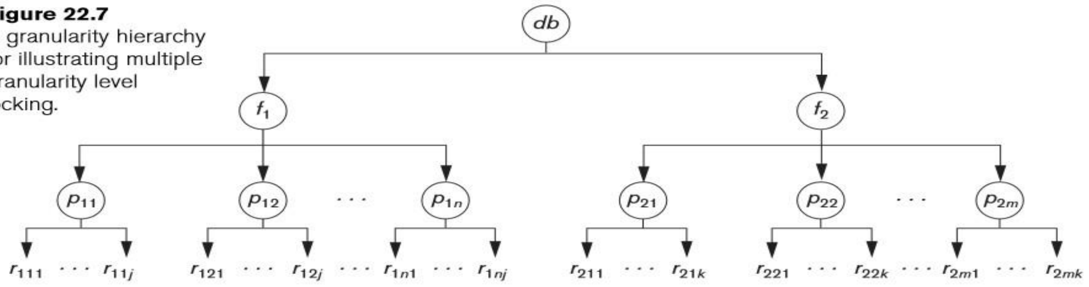
21.5.1 Granularity Level Considerations for Locking

- The size of data items is often called the data item granularity. Fine granularity => Small item sizes

  Coarse granularity => Large item sizes
- Larger the data item size is, the lower the degree of concurrency permitted. For example:
  - If the data item size is a disk block, a transaction T that needs to lock a record B must lock the whole disk block X that contains B because a lock is associated with the whole data item (block).
  - If another transaction S wants to lock a different record C that happens to reside in the same block X in a conflicting lock mode, it is forced to wait.
  - If the data item size was a single record, transaction S would be able to proceed, because it would be locking a different data item (record).
- Smaller the data item size is, the more the number of items in the database.
  - More lock and unlock operations will be performed, causing a higher overhead.
  - Hence, more storage space will be required for the lock table.
- The best item size depends on the *types of transactions* involved.

21.5.2 Multiple Granularity Level Locking

Figure 22.7
A granularity hierarchy for illustrating multiple granularity level locking.



The above figure shows a simple granularity hierarchy with a database containing two files (f1, f2), each file containing several disk pages and each page containing several records.

This can be used to illustrate a **multiple granularity level** 2PL protocol, where a lock can be requested at any level. However, additional types of locks will be needed to support such a protocol efficiently.

- Suppose transaction T1 wants to update all the records in file f1,and T1 requests and is granted an exclusive lock for f1.Then all of f1's pages (p11 through p1n)—and the records contained on those pages—are locked in exclusive mode.
- This is beneficial for T1 because setting a single file-level lock is more efficient than setting n page-level locks or having to lock each individual record.
- Now suppose another transaction T2 only wants to read record r1nj from page p1n of file f1; then T2 would request a shared record-level lock on r1nj.
- However, the database system (that is, the transaction manager or more specifically the lock manager) must verify the compatibility of the requested lock with already held locks.
- One way to verify this is to traverse the tree from the leaf r1nj to p1n to f1 to db. If at any time a conflicting lock is held on any of those items, then the lock request for r1nj is denied and T2 is blocked and must wait.
- This traversal would be fairly efficient.
- If transaction T2's request came before transaction T1's request, the shared record lock is granted to T2 for r1nj, but when T1's file-level lock is requested, it is quite difficult for the lock manager to check all nodes (pages and records) that are descendants of node f1 for a lock conflict.
- This would be very inefficient and would defeat the purpose of having multiple granularity level locks.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The 3 types are:

1. *Intention-shared* (IS) indicates that one or more shared locks will be requested on some descendant node(s).

2. *Intention-exclusive* (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).

3. *Shared-intention-exclusive* (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The multiple granularity locking (MGL) protocol consists of the following rules:

1. The lock compatibility (based on Figure 22.8) must be adhered to.

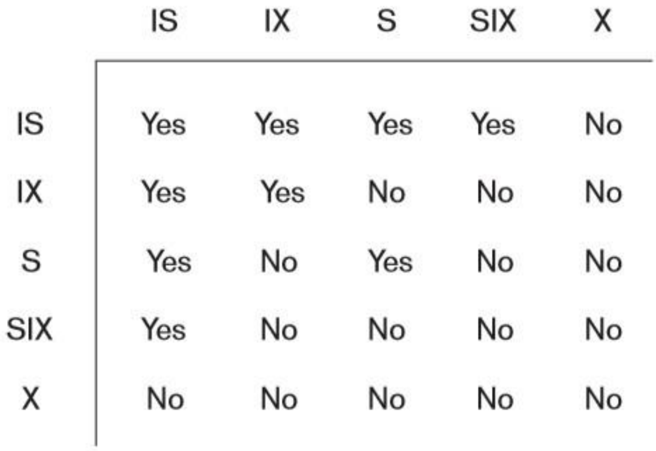
2. The root of the tree must be locked first, in any mode.

3. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.

4. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node Nis already locked by transaction Tin either IX or SIX mode.

5. A transaction T can lock a node only if it has not unlocked any node (to enforce the2PL protocol).

6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.



	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 22.8
Lock compatibility matrix for multiple granularity locking.

Rule 1 simply states that conflicting locks cannot be granted.

Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes.

Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable schedules.

To illustrate the MGL protocol with the database hierarchy in Figure 22.7, consider the following three transactions:

1. T1 wants to update record r111 and record r211.

2. T2 wants to update all records on page p12.

3. T3 wants to read record r11j and the entire f2 file.

Introduction to Database Recovery Protocols

22.1 Recovery Concepts

22.1.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force:

- When in-place updating is used, it is necessary to use a log for recovery.
- In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the before image (BFIM) is overwritten with the after image(AFIM) in the database on disk.
- This process is generally known as **write-ahead logging** and is necessary so we can UNDO the operation if this is required during recovery.
- **A REDO-type** log entry includes the new value (AFIM) of the item written by the operation since this is needed to redo the effect of the operation from the log (by setting the item value in the database on disk to its AFIM).
- **The UNDO-type** log entries include the old value (BFIM) of the item since this is needed to undo the effect of the operation from the log (by setting the item value in the database back to its BFIM).
- With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update must first be written to disk before the data block itself can be written back to disk from its main memory buffer.

Standard DBMS recovery terminology includes the terms steal/no-steal and force/no-force, which specify the rules that govern when a page from the database cache can be written to disk:

- If a cache buffer page updated by a transaction cannot be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin- unpin bit will be set to 1 (pin) to indicate that a cache buffer cannot be written back to disk.
- On the other hand, if the recovery protocol allows writing an updated buffer before the transaction commits, it is called **steal**.
- The no-steal rule means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.

- If all pages updated by a transaction are immediately written to disk before the transaction commits, the recovery approach is called a **force approach**. Otherwise, it is called **no-force**.
- The force rule means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.
- The advantage of steal is that it avoids the need for a very large buffer space to store all updated pages in memory.
- The advantage of no-force is that an updated page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to write that

page multiple times to disk and possibly having to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.

22.1.4 <u>Checkpoints in the System Log and Fuzzy Checkpointing</u>:

- Another type of entry in the log is called a checkpoint. A [checkpoint, list of active transactions] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified.
- The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time.

- A checkpoint consists of the following actions:

  1. Suspend execution of transactions temporarily.
  2. Force-write all main memory buffers that have been modified to disk.
  3. Write a [checkpoint] record to the log, and force-write the log to disk.
  4. Resume executing transactions.

- As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction.
- **fuzzy checkpointing**: In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, …] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid.

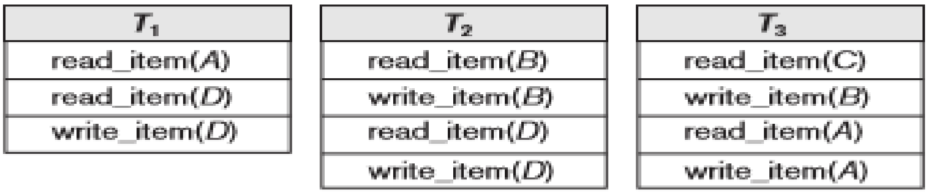
22.1.5 Transaction Rollback and Cascading Rollback

1. If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to roll back the transaction.
2. If any data item values have been changed by the transaction and written to the database on disk, they must be restored to their previous values (BFIMs).
3. The undo-type log entries are used to restore the old values of data items that must be rolled back.
4. If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back.
5. Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S must also be rolled back. This phenomenon is called **cascading rollback**, and it can occur when the recovery protocol ensures **recoverable schedules** but does not ensure **cascadeless schedules.**
6. Cascading rollback can be complex and time-consuming. So almost all recovery mechanisms are designed then cascading rollback is never required.
7. The below figure shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown fig(a) and fig (b), the system log at the point

of a system crash for a particular execution schedule of these transactions.

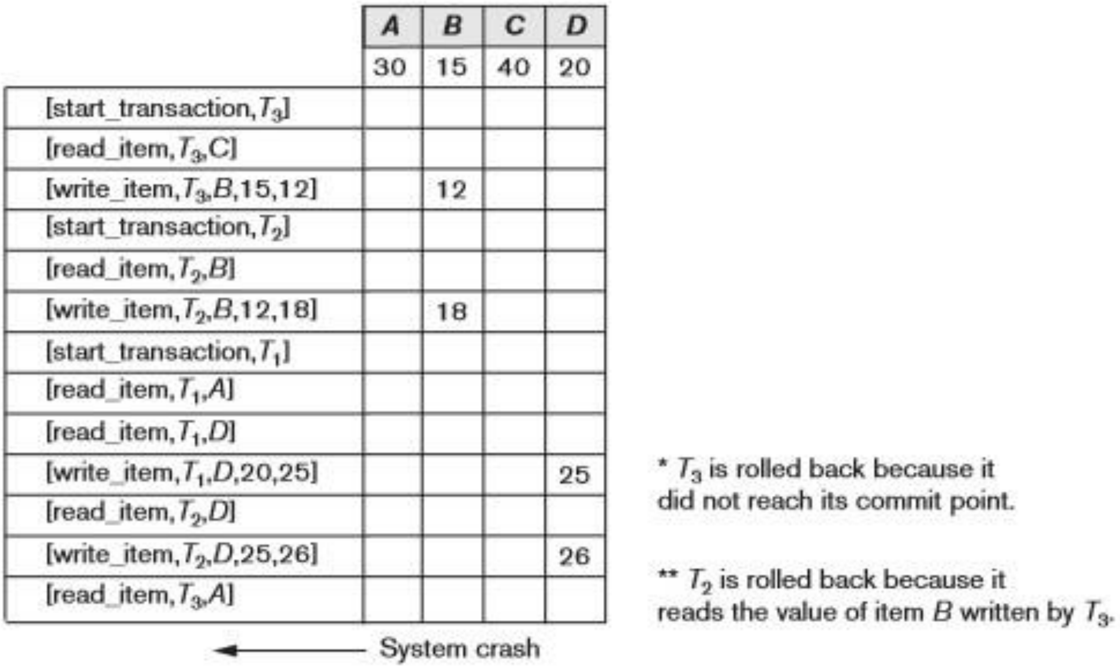
8. The values of data items A, B, C, and D, which are used by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are A = 30, B = 15, C = 40, and D = 20. At the point of system failure, transaction T3 has not reached its conclusion and must be rolled back.
9. The WRITE operations of T3, marked by a single * in fig(b), are the T3 operations that are undone during transaction rollback. The fig(c) graphically shows the operations of the different transactions along the time axis.

(a) The read and write operations of three transactions



T_1	T_2	T_3
read_item(A)	read_item(B)	read_item(C)
read_item(D)	write_item(B)	write_item(B)
write_item(D)	read_item(D)	read_item(A)
	write_item(D)	write_item(A)

(b) System log at point crashes



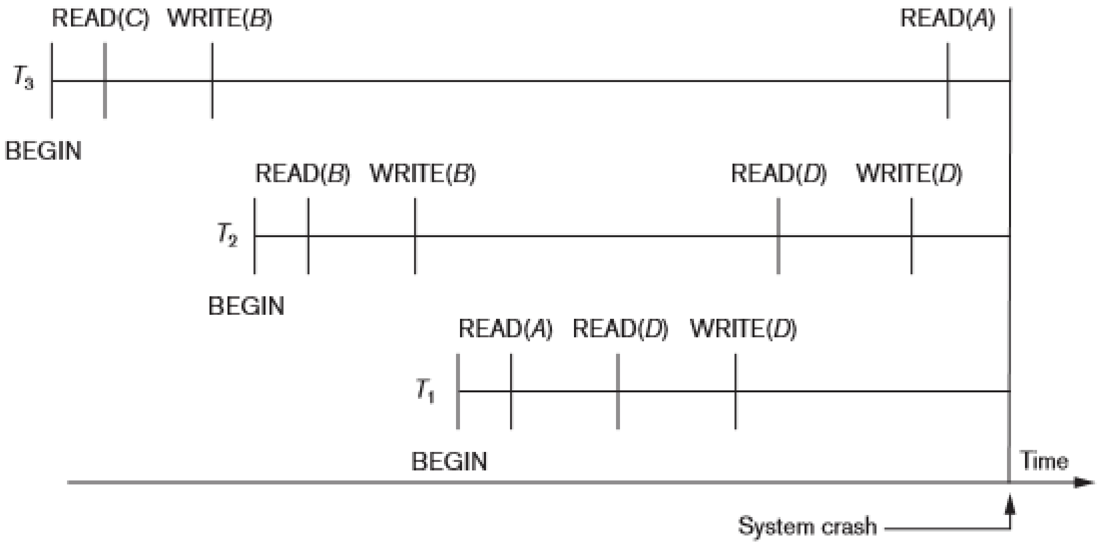
	A	B	C	D
	30	15	40	20
[start_transaction,T_3]				
[read_item,T_3,C]				
[write_item,T_3,B,15,12]		12		
[start_transaction,T_2]				
[read_item,T_2,B]				
[write_item,T_2,B,12,18]		18		
[start_transaction,T_1]				
[read_item,T_1,A]				
[read_item,T_1,D]				
[write_item,T_1,D,20,25]				25
[read_item,T_2,D]				
[write_item,T_2,D,25,26]				26
[read_item,T_3,A]				

← System crash

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3.

(c) Operations before the crash.



We must now check for cascading rollback. From fig(c), we see that transaction T2 reads the value of item B that was written by transaction T3; this can also be determined by examining the log. Because T3 is rolled back, T2 must now be rolled back, too. The WRITE operations of T2, marked by ** in the log, are the ones that are undone. **Note:**

- Only write_item operations need to be undone during transaction rollback.
- read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.
- In practice, cascading rollback of transactions is never required because practical recovery methods guarantee cascadeless or strict schedules.
- Hence, there is also no need to record any read_item operations in the log because these are needed only for determining cascading rollback.

22.1.6 Transaction Actions That Do Not Affect the Database

1. In general, a transaction will have actions that do not affect the database, such as generating and printing messages or reports from information retrieved from the database.
2. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete.
3. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action that is based on these reports and that affects the database.
4. Hence, such reports should be generated only after the transaction reaches its commit point.
5. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

22.2 NO-UNDO/REDO Recovery Based on Deferred Update

- During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force written to disk, the updates are recorded in the database.
- If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way.
- Therefore, only REDO type log entries are needed in the log.
- We can state a typical deferred update protocol as follows:
    1. A transaction cannot change the database on disk until it reaches its commit point; hence all buffers that have been changed by the transaction must be pinned until the transaction commits
    2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log and the log buffer is force-written to disk.
- Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations.
- REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk.
- For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated.

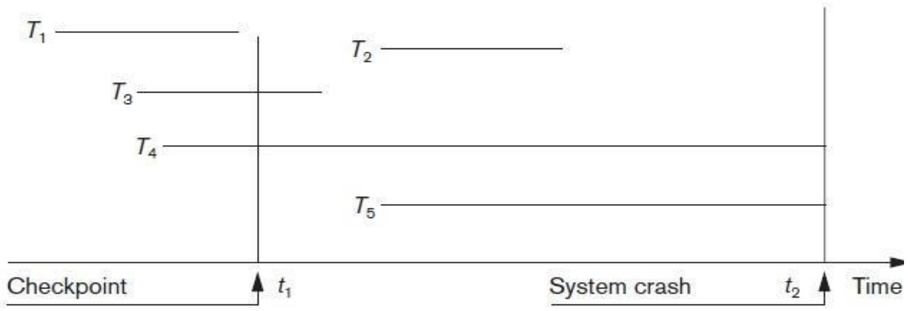
Recovery Algorithm RDU_M (Recovery using Deferred Update in a Multiuser environment) :

Procedure RDU_M (NO-UNDO/REDO with checkpoints). Use two lists of transactions maintained by the system: the committed transactions *T* since the last checkpoint (**commit list**), and the active transactions *T*′ (**active list**). REDO all the WRITE operations of the committed transactions from the log, in the order in which they were written into the log.
The transactions that are active and did not commit are effectively cancelled and must be resubmitted.

The REDO procedure is defined as follows:

Procedure REDO (WRITE_OP). Redoing a write_item operation WRITE_OP consists of examining its log entry [write_item, *T*, *X*, new_value] and setting the value of item *X* in the database to new_value, which is the after image (AFIM).

The below figure illustrates a timeline for a possible schedule of executing transactions. When the checkpoint was taken at time *t*1, transaction *T*1 had committed, whereas transactions *T*3 and *T*4 had not. Before the system crash at time *t*2, *T*3 and *T*2 were committed but not *T*4 and *T*5. According to the RDU_M method, there is no need to redo the write_item operations of transaction *T*1—or any transactions committed before the last checkpoint time *t*1. The write_item operations of *T*2 and *T*3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions *T*4 and *T*5 are ignored: They are effectively canceled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol (no- steal policy).



Drawbacks

1. The drawbacks of this method here is that it limits the concurrent execution of transactions because *all* write-locked items remain locked until the transaction reaches its commit point.
2. It require excessive buffer space to hold all updated items until the transactions commit.

Benefits:

The method's main benefit is that transaction operations never need to be undone, for two reasons:

1. A transaction does not record any changes in the database on disk until after it reaches its commit point. Hence, a transaction is never rolled back because of failure during transaction execution.
2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.