Database System

Unit 5

Transaction Processing and Concurrency Control

Transaction - Introduction

- a collection of several operations on the database appears to be a single unit from the point of view of the database user.
- For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint;
- within the database system, it consists of several operations.
- it is essential that all these operations occur,
- or in case of a failure, none occur.

- It would be unacceptable if the checking account were debited, but the savings account were not credited.
- Collections of operations that form a single logical unit of work are called transactions.
- A database system must ensure proper execution of transactions despite failures
- either the entire transaction executes, or none of it does.
- it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency.

Transaction Concept

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java)
- The transaction consists of all operations executed between the begin transaction and end transaction.

- To ensure integrity of the data,
- It is required that the database system maintain the following properties of the transactions:
- Atomicity
- Consistency
- Isolation
- Durability

- These properties are called the ACID 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.

- 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

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

Transaction State

- In the absence of failures, all transactions must complete successfully.
- a transaction may not always complete its execution successfully.
- Such a transaction is termed aborted.
- any changes that the aborted transaction made to the database must be undone.
- Once the changes caused by an aborted transaction have been undone,
- we say that the transaction has been rolled back.

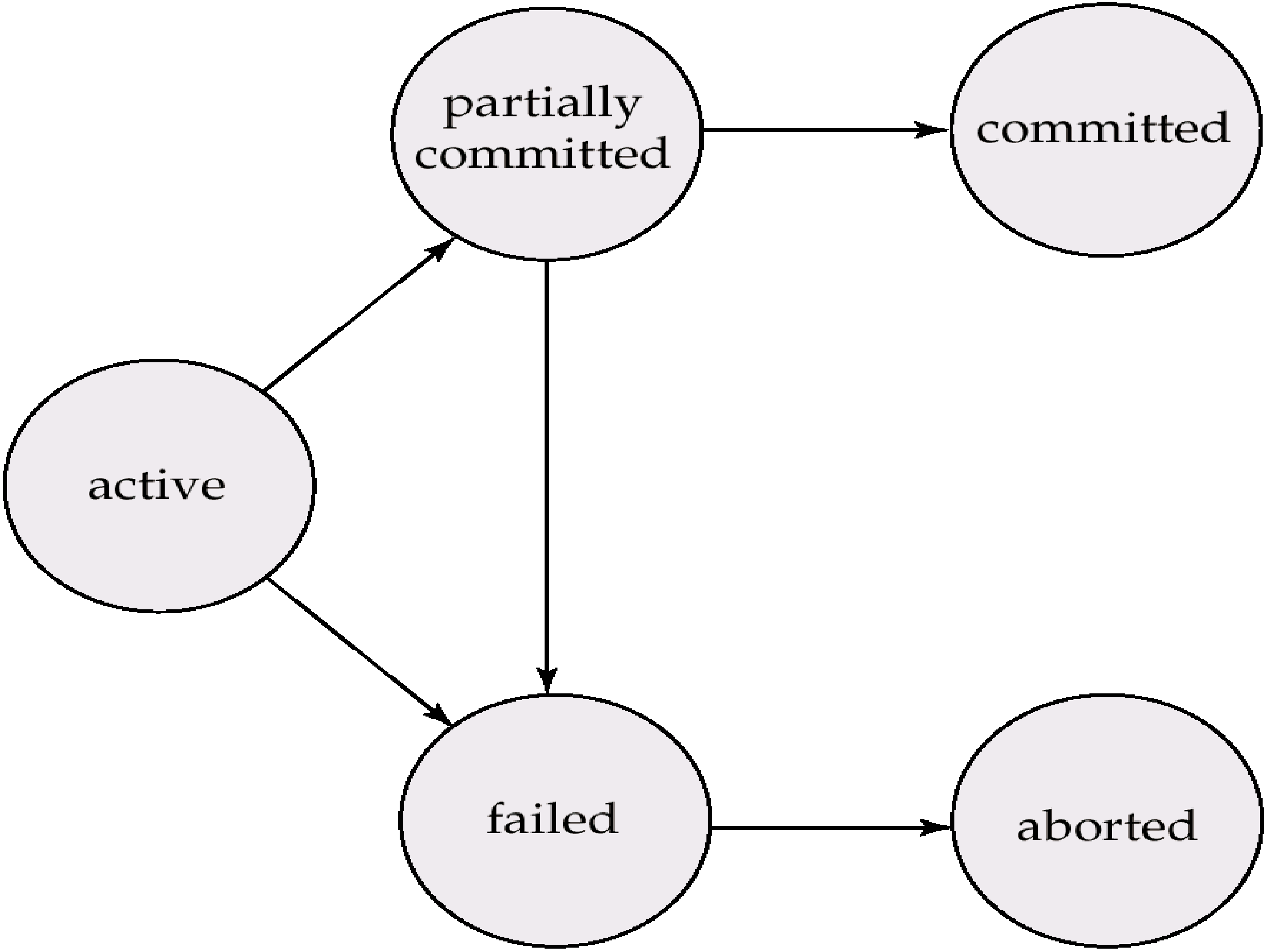
- A transaction that completes its execution successfully is said to be committed.
- A committed transaction that has performed updates
- transforms the database into a new consistent state, which must persist even if there is a system failure.
- Once a transaction has committed,we cannot undo its effects by aborting it.
- The only way to undo the effects of a committed transaction is to execute a compensating transaction.

A simple abstract transaction model

- 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

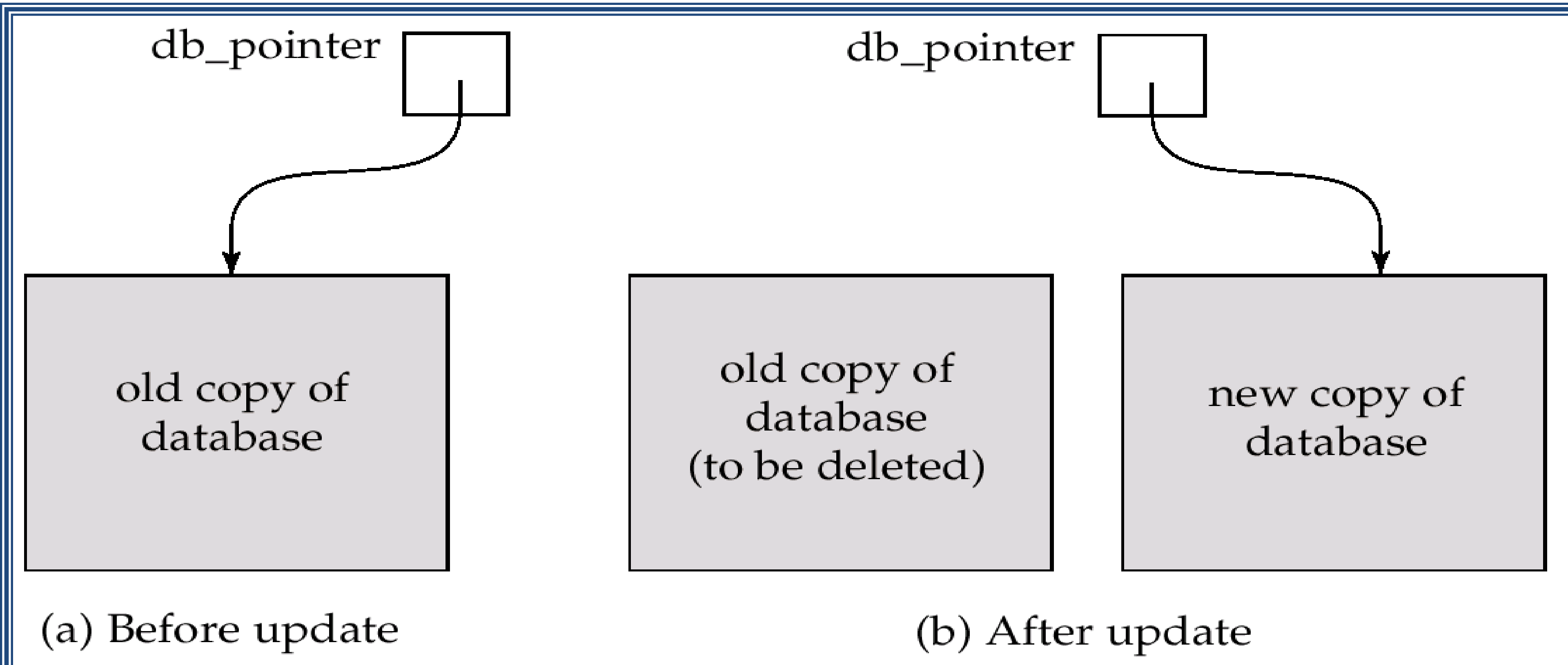
- Aborted
- after the transaction has been rolled back
- and the database has been restored to its state
  prior to the start of the transaction
- Two options after it has been aborted:
- restart the transaction –only if no internal logical error
- kill the transaction
- Committed after successful completion


partially committed
committed
active
failed
aborted

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.

The shadow-database scheme:



- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
- **increased processor and disk utilization** leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk

- **reduced average response time for transactions:** short transactions need not wait behind long ones.

Concurrent Executions

- **Concurrency control schemes –**
- mechanisms to achieve isolation,
- i.e., to control the interaction among the concurrent transactions
- in order to prevent them from destroying the consistency of the database

Schedules

- Schedules –
- sequences that indicate the chronological order
- in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction

Example- Schedules

- Let T_1 transfer \$50 from A to B,
- and T_2 transfer 10% of the balance from A to B.
- The following is a serial schedule
- in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously.
- The following schedule is not

a serial schedule,

- but it is *equivalent* to

Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In both Schedule, the sum A + B is preserved.

Example Schedules (Cont.)

- The following concurrent schedule does not preserve the value of the sum *A* + *B*.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. View serializability

Serializability

- We ignore operations other than **read** and **write** instructions,
- and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3—showing only the read and write instructions.

Conflict Serializability

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict**
- if and only if there exists some item Q accessed by both l_i and l_j,
- and at least one of these instructions wrote Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Conflict Serializability

- Ii and Ij conflict if they are operations by different transactions on the same data item,
- and at least one of these instructions is a write operation.
- a conflict between l_i and l_j forces a (logical) temporal order between them.
- If l_i and l_j are consecutive in a schedule and they do not conflict,
- their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability (Cont.)

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions,
- we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

Conflict Serializability (Cont.)

- Schedule below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
- Therefore Schedule 3 is conflict serializable.
- Schedule 1

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 2

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions.
- *S* and *S´* are **view equivalent** if the following three conditions are met:
  1. For each data item *Q,*

     if transaction T_i reads the initial value of *Q* in schedule *S,*

     then transaction T_i must, in schedule S´, also read the initial value of *Q.*

View Serializability

2. For each data item Q

if transaction Ti executes read(Q) in schedule S, and that value was produced by transaction Tj (if any), then transaction Ti must in schedule S´ also read the value of Q that was produced by transaction Tj

3. For each data item Q,

the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S´.

View Serializability (Cont.)

- The concept of view equivalence leads to the concept of view serializability.
- A schedule S is view serializable
- If it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Some schedules which are view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Some transanctions perform write(Q) operations without having performed a read(Q) operation.
- Writes of this sort are called blind writes.
- Blind writes appear in any view-serializable schedule that is not conflict serializable.

Recoverability

- **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i , the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Recoverability

- If T_8 fails and should have to abort,
- T_9 would have read (and possibly shown to the user) an inconsistent database state.
- Hence database must ensure that schedules are recoverable.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- an example of a non-recoverable schedule

- Most database system require that all schedules be recoverable.
- A recoverable schedule is one where,
- for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti,
- the commit operation of Ti appears before the commit operation of Tj.

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

•Can lead to the undoing of a significant amount of work

- **Cascadeless schedules** —
- cascading rollbacks cannot occur;
- for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i,
- the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Implementation of Isolation

- Schedules must be conflict or view serializable,
- and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules,
- but provides a poor degree of concurrency..

Testing for Serializability

- When designing concurrency control schemes,
- we must show that schedules generated by the scheme are serializable.
- To do that, we must first understand how to determine, given a particular schedule S,
- whether the schedule is serializable.

- We present a simple and efficient method for determining conflict serializability of a schedule.
- Consider a schedule S.
- We construct a directed graph, called a precedence graph, from S.
- 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 participating in the schedule.
- The set of edges consists of all edges $Ti \rightarrow Tj$ for which one of three conditions holds:

1. Ti executes write(Q) before Tj executes read(Q).
2. Ti executes read(Q) before Tj executes write(Q).
3. Ti executes write(Q) before Tj executes write(Q)

Figure 15.15 Precedence graph for (a) schedule 1 and (b) schedule 2.

- If an edge Ti →Tj exists in the precedence graph, then, in any serial schedule S` equivalent to S,
- Ti must appear before Tj.

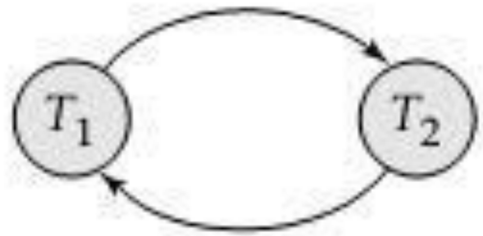


Figure 15.16 Precedence graph for schedule 4.

- The precedence graph appears in Figure.
- It contains the edge T1→T2, because T1 executes read(A) before T2 executes write(A).
- It also contains the edge T2→T1, because T2 executes read(B) before T1 executes write(B).
- If the precedence graph for S has a cycle,
- then schedule S is not conflict serializable.
- If the graph contains no cycles, then the schedule S is conflict serializable.

- A serializability order of the transactions can be obtained through topological sorting,
- which determines a linear order consistent with the partial order of the precedence graph.
- There are, several possible linear orders that can be obtained through a topological sorting.
- For example, the graph of Figure a has two acceptable linear orderings shown in Figures b and c.

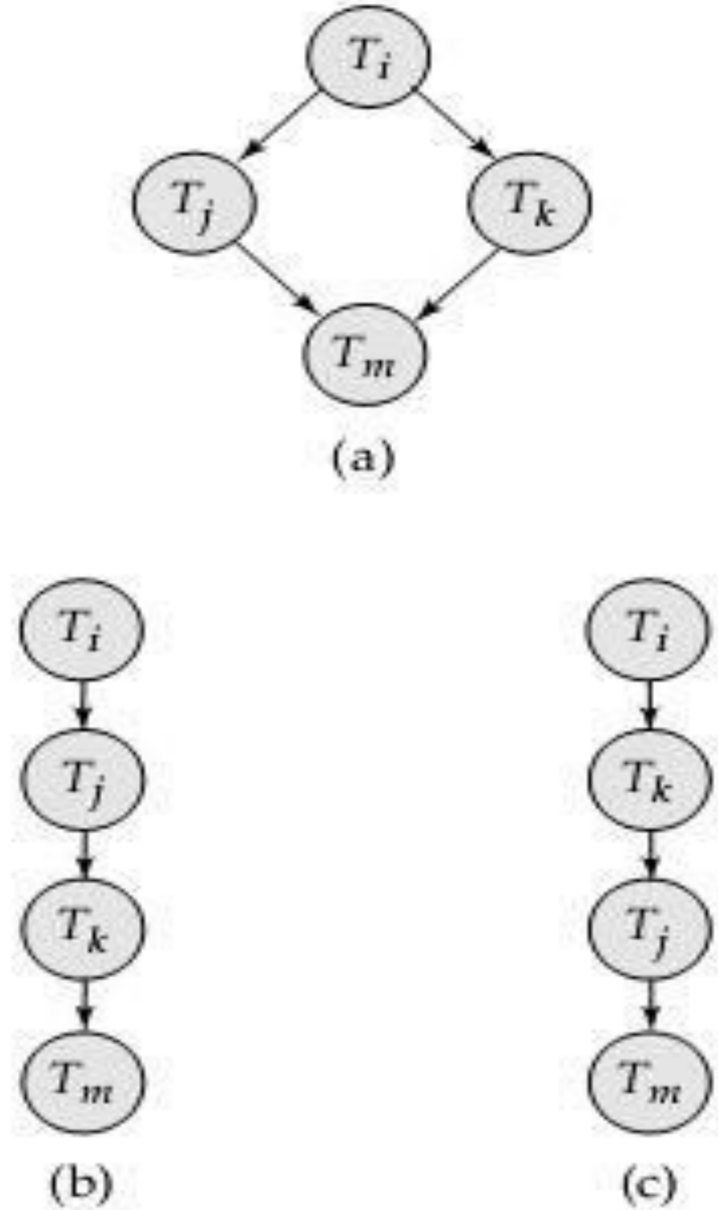


Figure 15.17 Illustration of topological sorting.

- Thus, to test for conflict serializability,
- we need to construct the precedence graph
- and to invoke a cycle-detection algorithm
- Testing for view serializability is too complicated.
- it has been shown that the problem of testing for view serializability is itself NP-complete.
- Thus, almost certainly there exists no efficient algorithm to test for view serializability.
- However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

Concurrency Control

- one of the fundamental properties of a transaction is isolation.
- When several transactions execute concurrently in the database
- Isolation property may no longer be preserved
- To ensure isolation, the system must control the interaction among the concurrent transactions;
- this control is achieved through one of a variety of mechanisms called concurrency-control schemes
- concurrency-control schemes, based on the serializability property.

Lock-Based Protocols

- One way to ensure serializability is to require that
- data items be accessed in a mutually exclusive manner;
- 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.

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
- exclusive (X) mode.
- Data item can be both read as well as written.
- X-lock is requested using **lock-X** instruction.
- shared (S) mode.
- Data item can only be read.
- S-lock is requested using **lock-S**

- Lock requests are made to concurrency-control manager.
- Transaction can proceed only after request is granted.
- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

- Example of a transaction performing locking:

T_2: **lock-S**(*A*); **read** (*A*); **unlock**(*A*); **lock-S**(*B*); **read** (*B*); **unlock**(*B*); **display**(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability —

- if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

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)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
- To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Granting of Locks

- 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 made its lock request before Ti.

- Thus, a lock request will never get blocked by a lock request that is made later.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - but may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - but may not obtain locks

- The protocol assures serializability.
- It can be proved that the transactions can be serialized in the order of their **lock points**
- (i.e. the point where a transaction acquired its final lock).

- Two-phase locking does not ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking.
- To avoid this, follow a modified protocol called **strict two-phase locking**.
- Here a transaction must hold all its exclusive locks till it commits/aborts.

- **Rigorous two-phase locking** is even stricter:
- here all locks are held till commit/abort.
- In this protocol transactions can be serialized in the order in which they commit.

Lock Conversions

- Two-phase locking with lock conversions:
  - Growing Phase / First Phase:
  - can acquire a **lock-S** on item
  - can acquire a **lock-X** on item
  - can convert a **lock-S** to a **lock-X** (**upgrade**)
  - Shrinking Phase / Second Phase:
  - can release a **lock-S**
  - can release a **lock-X**
  - can convert a **lock-X** to a **lock-S** (**downgrade**)
- This protocol assures serializability.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(D) is processed as:

if T_i has a lock on D
  then
    read(D)
  else
    begin
      if necessary wait until no other
        transaction has a **lock-X** on D
      grant T_i a **lock-S** on D;
      read(D)
    end

Automatic Acquisition of Locks

- **write**(D) is processed as:

if T_i has a **lock-X** on D
  then
    write(D)
  else
  begin
    if necessary wait until no other trans. has any lock on D,
    if T_i has a **lock-S** on D
      then
        upgrade lock on D to **lock-X**
      else
        grant T_i a **lock-X** on D
      write(D)
  end;

- All locks are released after commit or abort

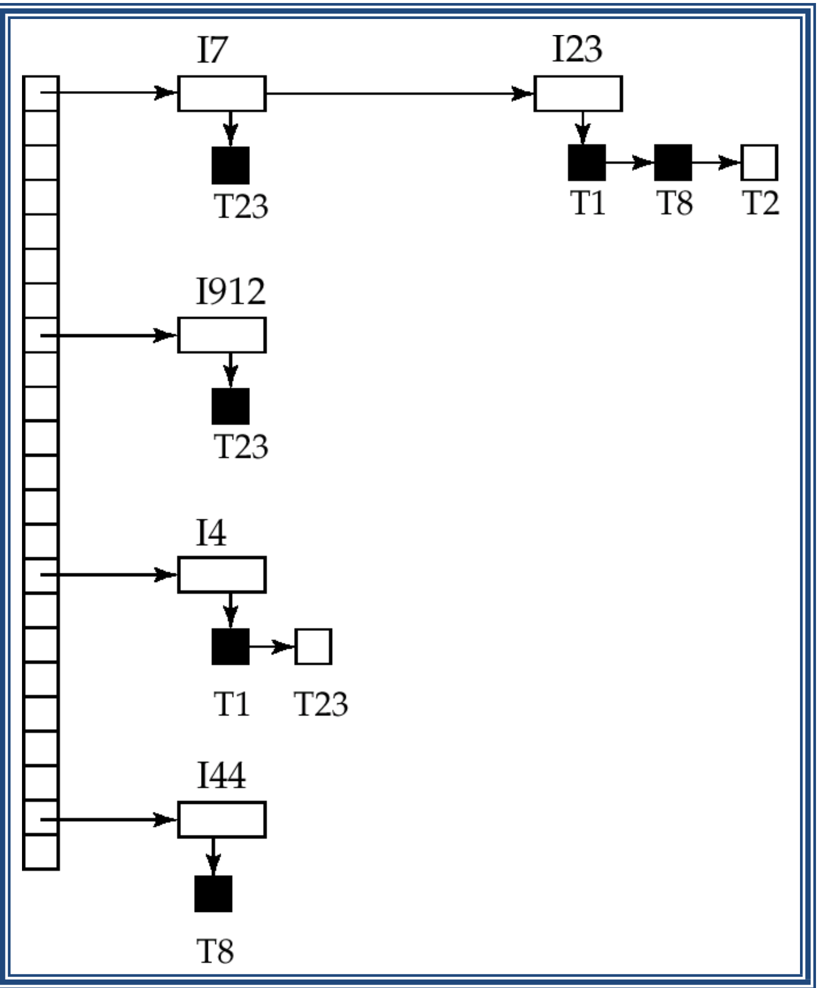
Implementation of Locking

- A **Lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered

Implementation of Locking

- The lock manager maintains a data structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

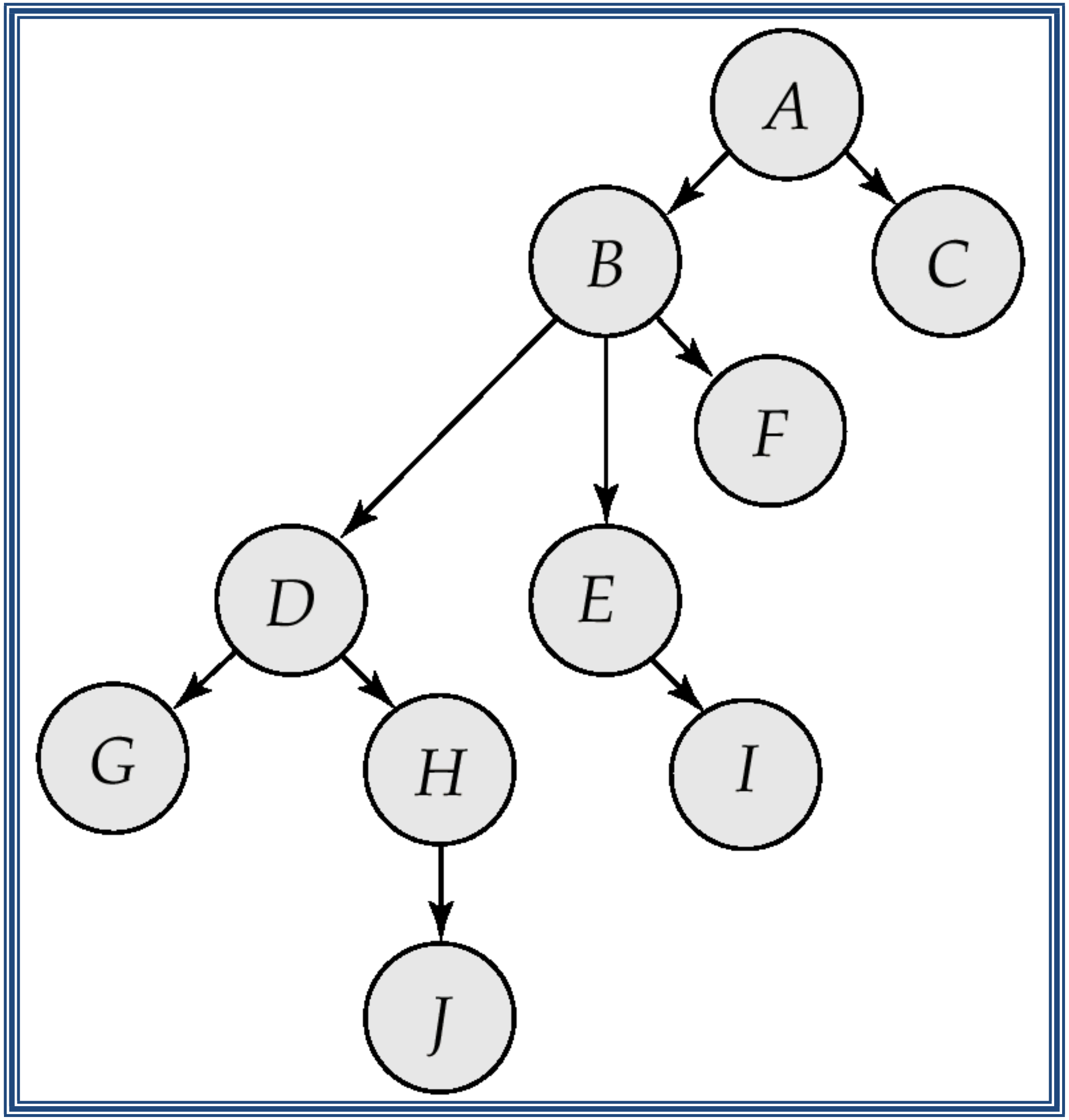
Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set
- **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

- 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

Tree Protocol



- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
  - the abort of a transaction can still lead to cascading rollbacks.

- However, in the tree-locking protocol, a transaction may have to lock data items that it does not access.
  - increased locking overhead, and additional waiting time
  - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system.
- If an old transaction T_i has time-stamp $TS(T_i)$,
- a new transaction T_j is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.

- There are two simple methods for implementing this scheme:
- 1. Use the value of the system clock as the timestamp;
- 2. Use a logical counter that is incremented after a new timestamp has been assigned;

- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
- **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
- **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

- Suppose a transaction T_i issues a **read**(Q)
- 1. If TS(T_i) $\leq$ **W**-timestamp(Q), .
- then T_i needs to read a value of Q that was already overwritten.
- Hence, the **read** operation is rejected, and T_i is rolled back.
- 2. If TS(T_i)$\geq$ **W**-timestamp(Q), then the **read** operation is executed,
- and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

- Suppose that transaction T_i issues **write**(Q).
- If TS(T_i) < R-timestamp(Q),
- then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
- Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
- Hence, this **write** operation is rejected, and T_i is rolled back.
- Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Thomas' write rule

- We now present a modification to the timestamp-ordering protocol
- that allows greater potential concurrency
- Let us consider a schedule

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

Figure 16.14 Schedule 4.

- and apply the timestamp-ordering protocol.
- SinceT16 starts beforeT17,
- we shall assume that TS(T16)<TS(T17).

- The read(Q) operation of T16 succeeds,
- as does the write(Q) operation of T17.
- When T16 attempts its write(Q)operation,
- we find that TS(T16)<W-timestamp(Q),
- since W-timestamp(Q)=TS(T17).
- Thus, the write(Q) by T16 is rejected and transaction T16 must be rolled back.

- Although the rollback of T16 is required by the timestamp-ordering protocol,
- It is unnecessary.
- Since T17 has already written Q,
- the value that T16 is attempting to write is one that will never need to be read.
- Any transaction Ti with TS(Ti)<TS(T17)
- that attempts a read(Q) will be rolled back,
- since TS(Ti) <W-timestamp(Q).
- Any transaction Tj with TS(Tj)>TS(T17) must read the value of Q written by T17, rather than the value written by T16

- This observation leads to a modified version of the timestamp-ordering protocol
- in which obsolete write operations can be ignored under certain circumstances.
- The protocol rules for read operations remain unchanged.
- The protocol rules for write operations, are slightly different from the timestamp-ordering protocol.

- The modification to the timestamp-ordering protocol, called Thomas' write rule, is this:
- Suppose that transaction Ti issues write(Q).
- 1. If TS(Ti)<R-timestamp(Q),
- then the value of Q that Ti is producing was previously needed,
- and it had been assumed that the value would never be produced.
- Hence, the system rejects the write operation and rolls Ti back.

- 2. If TS(Ti)<W-timestamp(Q),
- Then Ti is attempting to write an obsolete value

Of Q.

- Hence, this write operation can be ignored.

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

Validation-Based Protocols

- A majority of transactions are read-only transactions, the rate of conflicts among transactions may be low.
- Many of these transactions, if executed without the supervision of a concurrency-control scheme.
- A concurrency-control scheme imposes overhead of code execution and possible delay of transactions.

- A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict.

- To gain that knowledge, we need a scheme for monitoring the system.

- Each transaction Ti executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction.

1. **Read phase**. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.

2. **Validation phase**. Transaction T_i performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.

3. **Write phase**. If transaction T_i succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back T_i.

three different timestamps with transaction *Ti*:

1. **Start**(T_i), the time when T_i started its execution.
2. **Validation**(T_i), the time when T_i finished its read phase and started its validation phase.
3. **Finish**(T_i), the time when T_i finished its write phase.

- We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation (Ti).

- the value TS (Ti) = Validation (Ti) and, if TS (Tj) < TS (T k), then any produced schedule must be equivalent to a serial schedule in which transaction Tj appears before transaction Tk.

- The validation test for transaction T j requires that, for all transactions Ti with TS(T i) < TS(T j), following two conditions must hold:

1. Finish(T_i) < Start(T_j). Since T_i completes its execution before T_j started, the serializability order is indeed maintained.

2. The set of data items written by T_i does not intersect with the set of data items read by T_j, and T_i completes its write phase before T_j starts its validation phase (Start(T_j) < Finish(T_i) < Validation(T_j)). This condition ensures that

the writes of T_i and T_j do not overlap. Since the writes of T_i do not affect the read of T_j, and since T_j cannot affect the read of T_i, the serializability order is indeed maintained.

Multiple Granularity

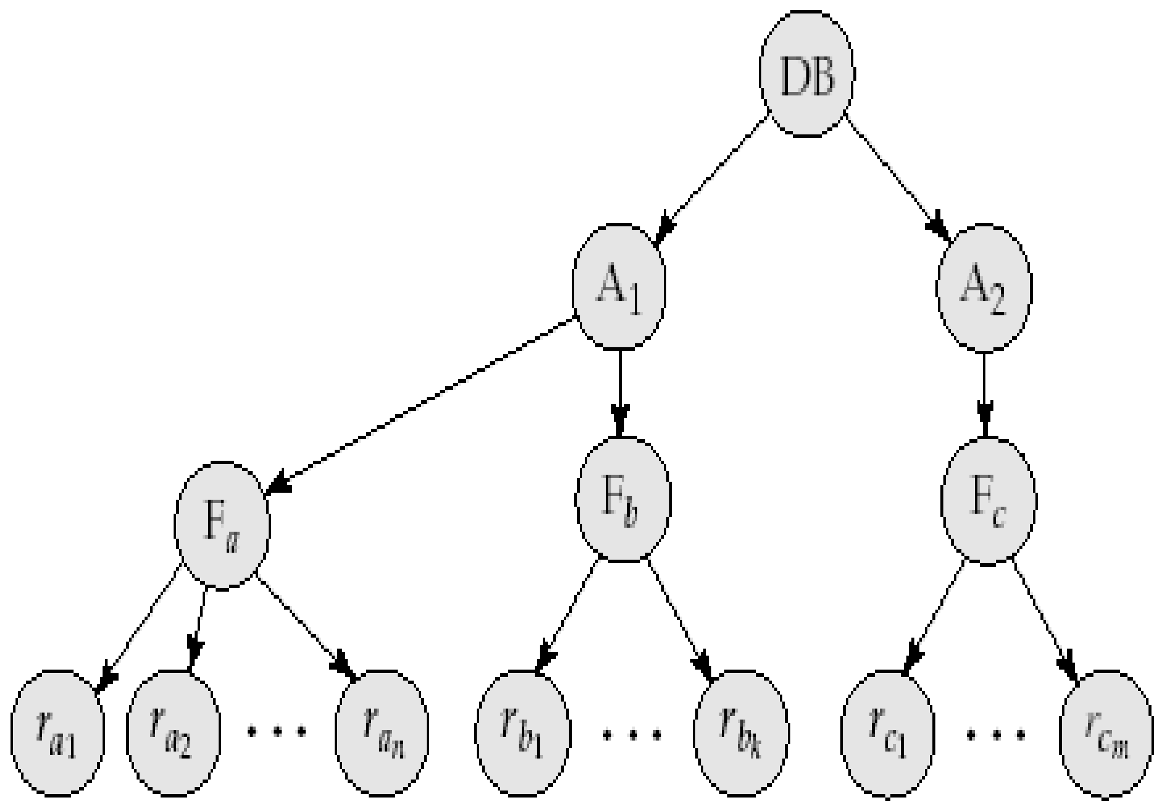
- We have used each individual data item as the unit on which synchronization is performed.

- it would be advantageous to group several data items, and to treat them as one individual synchronization unit.

- Example
- if a transaction Ti needs to access the entire database, and a locking protocol is used.
- Ti must lock each item in the database.

- executing these locks is time consuming.
- It would be better if Ti could issue a single lock request to lock the entire database.
- 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.
- We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities,

- the small granularities are nested within larger ones.
- Such a hierarchy can be represented graphically as a tree.
- A nonleaf 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 hierarchy.

- Figure consists of four levels of nodes.
- The highest level represents the entire database.
- Below it are 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.

each file has nodes of type *record*.

Each node in the tree can be locked individually.

	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

Compatibility matrix.

- There is an intention mode associated with shared mode, and there is one with exclusive mode.

- if transaction Ti gets an explicit lock on file Fc 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.

- 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.
- 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.
- 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