# Chapter 8: Transaction and Concurrency Control

## Transaction Concepts

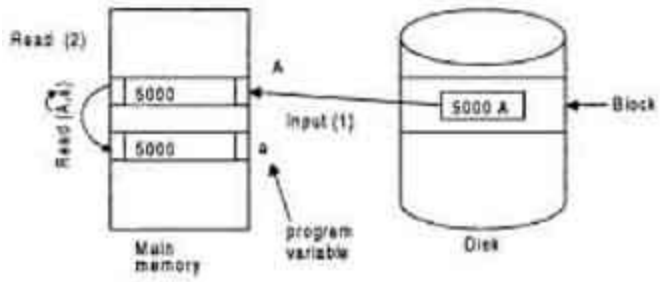
A transaction is a set of changes that must all be made together. It is a unit of program execution that access and possibly updates various data items. Moreover, it is a program unit whose execution mayor may not change the contents of a database. If the database was in consistent state before a transaction, then after execution of the transaction also, the database must be in a consistent state. For example, a transfer of money from one bank account to another requires two changes to the database both must succeed or fail together.

## Process of Transaction

The transaction is executed as a series of reads and writes of database objects, which are explained below:

### Read Operation

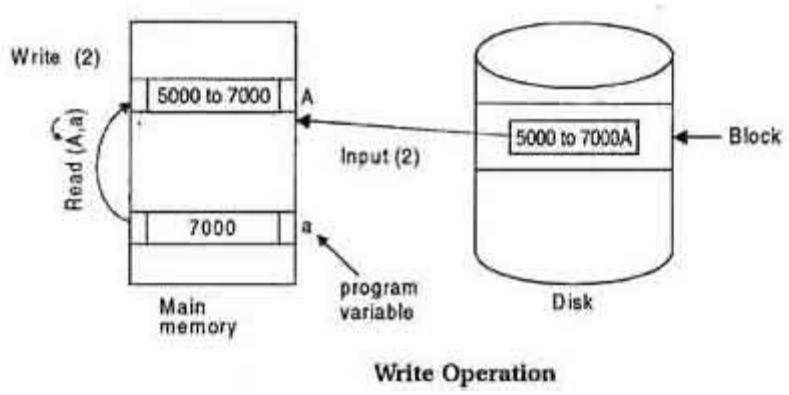
To read a database object, it is first brought into main memory from disk, and then its value is copied into a program variable as shown in figure.




Read Operation

### Write Operation

To write a database object, an in-memory copy of the object is first modified and then written to disk.



Write Operation

## Transaction Properties(ACID properties of transaction)

There are four important properties of transaction that a DBMS must ensure to maintain data in the case of concurrent access and system failures. These are:

**Atomicity: (all or nothing)**

A transaction is said to be atomic if a transaction always executes all its actions in one step or not executes any actions at all It means either all or none of the transactions operations are performed.

**Consistency: (No violation of integrity constraints)**

A transaction must preserve the *consistency* of a database after the execution. The DBMS assumes that this property holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

**Isolation: (concurrent changes invisibles)**

The transactions must behave as if they are executed in isolation. It means that if several transactions are executed concurrently the results must be same as if they were executed serially in some order. The data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

**Durability: (committed update persist)**

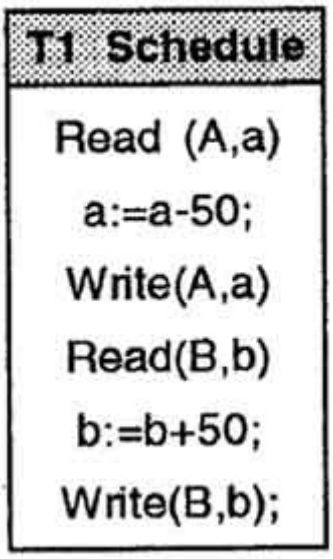
The effect of completed or committed transactions should persist even after a crash. It means once a transaction commits, the system must guarantee that the result of its operations will never be lost, in spite of subsequent failures.

The acronym ACID is sometimes used to refer above four properties of transaction that we have presented here: Atomicity, Consistency, Isolation, and Durability.

**Example**

In order to understand above properties consider the following example:

Let, Ti is a transaction that transfers Rs 50 from account A to account B. This transaction can be defined as:




**Atomicity**

Suppose that, just prior to execution of transaction Ti the values of account A and B are Rs.I000 and Rs.2000.

Now, suppose that during the execution of Ti, a power failure has occurred that prevented the Ti to complete successfully. The point of failure may be after the completion Write (A,a) and before Write(B,b). It means that the changes in A are performed but not in B. Thus the values of account A and Bare Rs.950 and Rs.2000 respectively. We have lost Rs.50 as a result 'of this failure.

Now, our database is in inconsistent state.

The reason for this inconsistent state is that our transaction is completed partially and we save the changes of uncommitted transaction. So, in order to get the consistent state, database must be restored to its original values i.e. A to Rs.I000 and B to Rs.2000, this leads to the concept of atomicity of transaction. It means that in order to maintain the consistency of database, either all or none of transaction's operations are performed.

In order to maintain atomicity of transaction, the database system keeps track of the old values of any write and if the transaction does not complete its execution, the old values are restored to make it appear as the transaction never executed.

**Consistency**

The consistency requirement here is that the sum of A and B must be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction. It can. be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

**Isolation**

If several transactions are executed concurrently (or in parallel), then each transaction must behave as if it was executed in isolation. It means that concurrent execution does not result an inconsistent state.For example, consider another transaction T2, which has to display the sum of account A and B. Then, its result should be Rs.3000.

Let's suppose that both Tl and T2 perform concurrently, their schedule is shown below:

| T1 | T2 | Status |
|---|---|---|
| Read (A,a) | | value of *A* i.e.1000 is copied to local variable a |
| a:=a-50 | | local variable *a*=950 |
| Write(A,a) | | value of local variable *a* 950 is copied to database item *A* |
| | Read(A,a) | value of database item *A* 950 is copied to *a* |
| | Read(B,b) | value of database item *B* 2000 is copied to *b* |
| | display(a+b) | 2950 is displayed as sum of accounts *A* and *B* |
| Read(B,b) | | value of data base item *B* 2000 is copied to local variable *b* |
| b:=b+50 | | local variable *b*=2050 |
| Write(B,b) | | value of local variable *b* 2050 is copied to database item *B* |

The above schedule results inconsistency of database and it shows Rs.2950 as sum of accounts A and B instead of Rs.3000. The problem occurs because second concurrently running transaction T2, reads A and B at intermediate point and computes its sum, which results inconsistent value. Isolation property demands that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

A solution to the problem of concurrently executing transaction is to execute each transaction serially 'that is one after the other. However, concurrent execution of transaction provides significant performance benefits, so other solutions are developed they allow multiple transactions to execute concurrently.

**Durability**

Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.

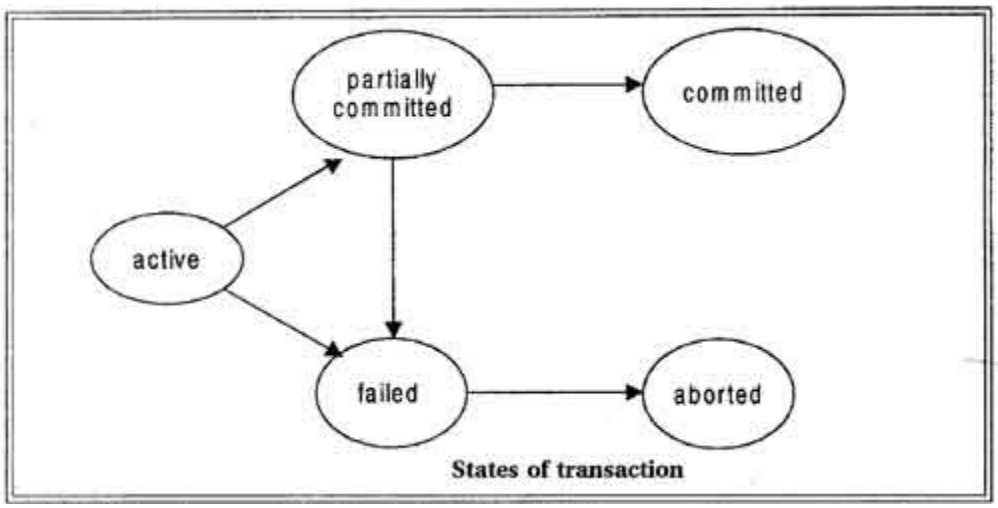
The durability property guarantees that, once a transaction completes successfully all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

## Transaction Model and State Diagram

Simple transaction model is a model of transaction how it must be. It has active, partially committed, failed, aborted, and committed states. Transaction is a several operations that can change the content of the database which is handled by a single program. Simple transaction model follows all ACID properties while doing transactions.

A transaction must be in one of the following states:

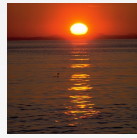
- **Active**: the initial state, the transaction stays in this state while it is executing.
- **Partially committed**: after the final statement has been executed.
- **Failed**: when the normal execution can no longer proceed.
- **Aborted**: after the transaction has been rolled back and the database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

- 

States of transaction

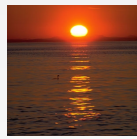
# Chapter 8(contd..): Schedules and Serializability
# Concepts of locking for concurrency control

.



# Schedules

- **Schedule** – a sequences of instructions that specify 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.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



# Schedule 1

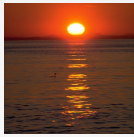
- Let $T_1$ transfer \$50 from *A* to *B*, and $T_2$ transfer 10% of the balance from *A* to *B.*
- A serial schedule in which $T_1$ is followed by $T_2$ :

| $T_1$ | $T_2$ |
|---|---|
| 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($B$) |

# Schedule 2

- A serial schedule where $T_2$ is followed by $T_1$

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



# Schedule 3

- 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 Schedules 1, 2 and 3, the sum A + B is preserved.

# Schedule 4

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

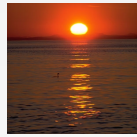
| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| write($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | $B := B + temp$ |
| | write($B$) |

# 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:

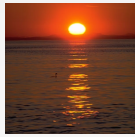
1. **conflict serializability**
2. **view serializability**

- *Simplified view of transactions*
  - We ignore operations other than **read** and **write** instructions
  - 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.



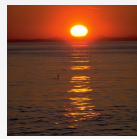
# Conflicting Instructions

- Instructions $I_i$ and $I_j$ of transactions $T_i$ and $T_j$ respectively, **conflict** if and only if there exists some item $Q$ accessed by both $I_i$ and $I_j$, and at least one of these instructions wrote $Q$.
  1. $I_i$ = **read**($Q$), $I_j$ = **read**($Q$). $I_i$ and $I_j$ don't conflict.
  2. $I_i$ = **read**($Q$), $I_j$ = **write**($Q$). They conflict.
  3. $I_i$ = **write**($Q$), $I_j$ = **read**($Q$). They conflict
  4. $I_i$ = **write**($Q$), $I_j$ = **write**($Q$). They conflict
- Intuitively, a conflict between $I_i$ and $I_j$ forces a (logical) temporal order between them.
  - If $I_i$ and $I_j$ 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

- 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.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where $T_2$ follows $T_1$, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| 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($B$) |

Schedule 6

# 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 < $T_4$, $T_3$ >.

## View Serializability

View Serializability is a process to find out that a given schedule is view serializable or not.To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule

### View Equivalent:

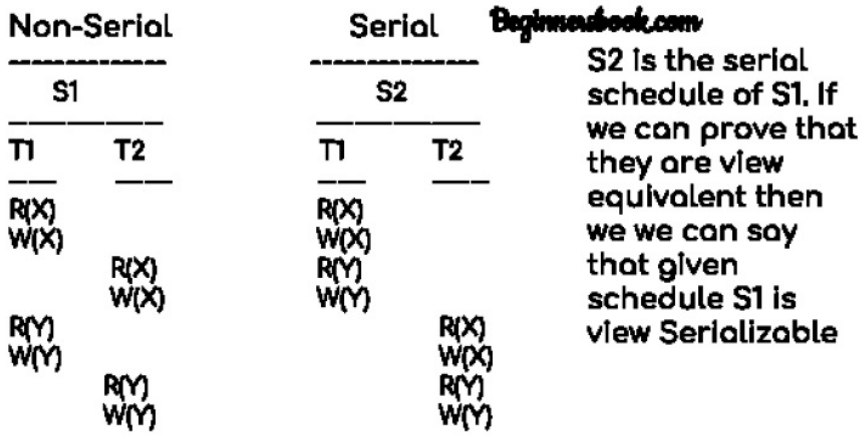
Lets learn how to check whether the two schedules are view equivalent.
Two schedules S1 and S2 are said to be view equivalent, if they satisfy all the following conditions:
1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

**Read vs Initial Read:** Don't confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read.

2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

### View Serializable Example




Lets check the three conditions of view serializability:

## Initial Read

In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

We checked for both data items X & Y and the **initial read** condition is satisfied in S1 & S2.

## Final Write

In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.

Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.

We checked for both data items X & Y and the **final write** condition is satisfied in S1 & S2.

## Update Read

In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.

In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.

The update read condition is also satisfied for both the schedules.

**Result:** Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

Note: All conflict seriable schedule are view seriable but not vice-versa.

**Example:**

| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| | Write(A) | |
| Write(A) | | |
| | | Write(A) |

**Schedule S**

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>

| T1 | T2 | T3 |
|---|---|---|
| Read(A)<br>Write(A) | Write(A) | Write(A) |

**Schedule S1**

**Step 1:** final updation on data items
In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.
**Step 2:** Initial Read
The initial read operation in S is done by T1 and in S1, it is also done by T1.
**Step 3:** Final Write
The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.Hence, view equivalent serial schedule is: T1 → T2 → T3

**Problem :** Prove whether the given schedule is View-Serializable or not?
S' : read1(A), write2(A), read3(A), write1(A), write3(A)
**Solution :** First of all we'll make a table for better understanding of given transactions of **schedule S'-**

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read(a) | | |
| | Write(a) | |
| | | read(a) |
| Write(a) | | |
| | | write(a) |

Now prove yourself

## SQL Standard Isolation Levels:

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena:

- **Dirty Read –** A Dirty read is a situation when a transaction reads data that has not yet been committed. For example, Let's say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
- **Non-Repeatable read –** Non-Repeatable read occurs when a transaction reads the same row twice and gets a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, now if transaction T1 rereads the same data, it will retrieve a different value.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

Based on these phenomena, The SQL standard defines four isolation levels:

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads, non-repeatable reads, and phantom reads. At this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions

cannot read, update or delete these rows, consequently it avoids non-repeatable read however, phantom reads are still possible.

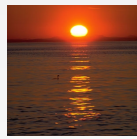
4. **Serializable –** This is the highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing, which ensures that there are no dirty reads, non-repeatable reads, or phantom reads.

The Table given below clearly depicts the relationship between isolation levels, read phenomena, and locks:

| Isolation Level | Dirty reads | Non-repeatable reads | Phantoms |
|---|---|---|---|
| **Read Uncommitted** | May occur | May occur | May occur |
| **Read Committed** | Don't occur | May occur | May occur |
| **Repeatable Read** | Don't occur | Don't occur | May occur |
| **Serializable** | Don't occur | Don't occur | Don't occur |

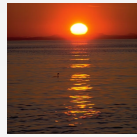
# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable 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
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.



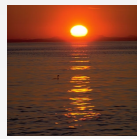
# Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  -
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.



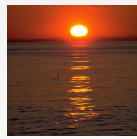
# Concurrency Control

- Lock-Based Protocols
- Graph Based protocol



# Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.



# Lock-Based Protocols (Cont.)

- 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.

# Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  $T_2$: **lock-S**(*A*);

  **read** (*A*);

  **unlock**(*A*);

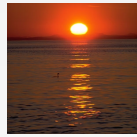
  **lock-S**(*B*);

  **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

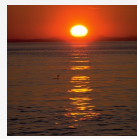
- Consider the partial schedule

| $T_3$ | $T_4$ |
|---|---|
| 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.

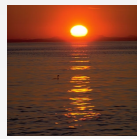
# Pitfalls of Lock-Based Protocols (Cont.)

- 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.



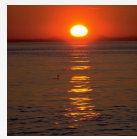
# The Two-Phase Locking Protocol

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



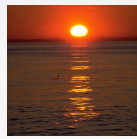
# The Two-Phase Locking Protocol (Cont.)

- 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:
  - 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)
  - 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. But still relies on the programmer to insert the various locking instructions.



## Graph-Based Protocol:

In graph-based concurrency control, transactions are represented as nodes in a graph, and the conflicts between transactions are represented as edges between the nodes. A conflict between two transactions occurs when they access the same data item, and at least one of the transactions performs a write operation. So, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

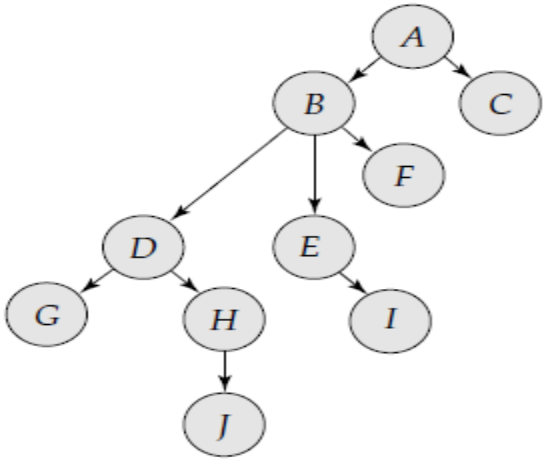



**Figure 16.11** Tree-structured database graph.

To acquire such prior knowledge, we impose a partial ordering → on the set D = {d1, d2, . . ., dh} of all data items. If di → dj , then any transaction accessing both di and dj must access di before accessing dj . This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control. The partial ordering implies that the setDmay now be viewed as a directed acyclic graph, called a database graph. The tree protocol, which is restricted to employ only exclusive locks. In the tree protocol, the only lock instruction allowed is lock-X. Each transaction Ti can lock a data item at most once, and must observe the following rules:

1. The first lock by Ti may be on any data item.

2. Subsequently, a data item Q can be locked by Ti only if the parent of Q is currently locked by Ti.

3. Data items may be unlocked at any time.

4. A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti.

All schedules that are legal under the tree protocol are conflict serializable.

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

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

One possible schedule in which these four transactions participated appears in Figure 16.12. Note that, during its execution, transaction T10 holds locks on two disjoint subtrees.
Observe that the schedule of Figure 16.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.

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

**Figure 16.12** Serializable schedule under the tree protocol.

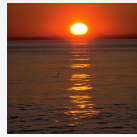
The tree protocol in Figure above does not ensure recoverability and cascadelessness.

The tree-locking protocol has an **advantage** over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

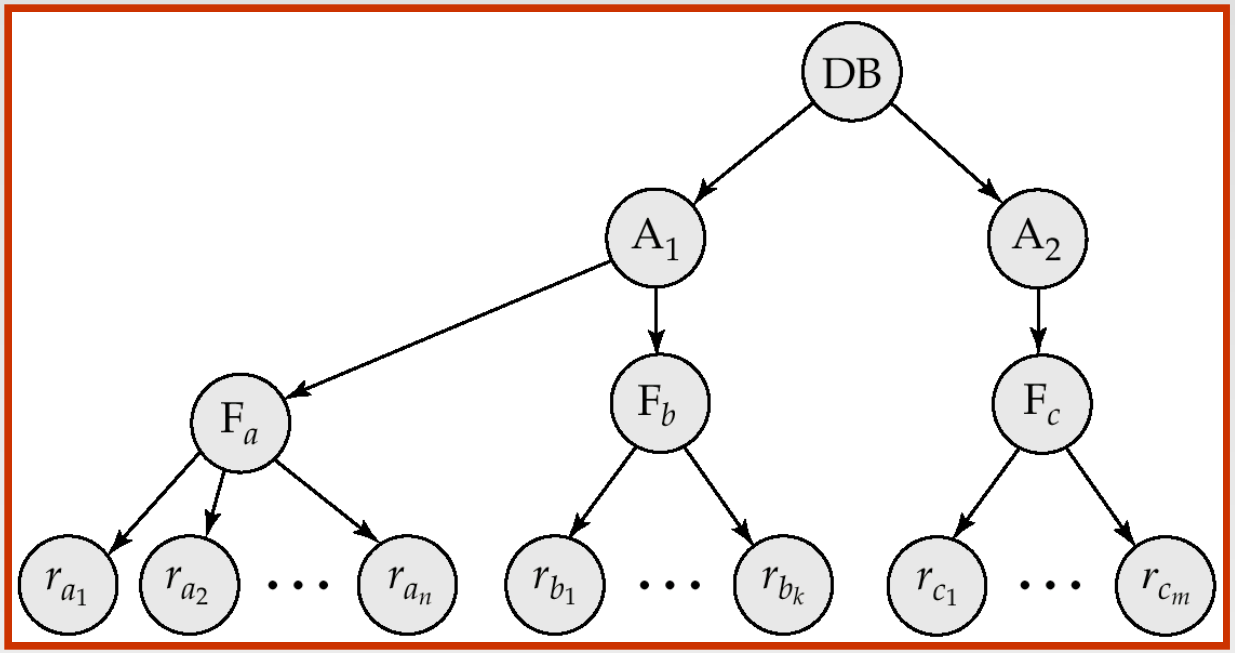
However, the protocol has the **disadvantage** that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items A and J in the database graph of Figure 16.11 must lock not only A and J, but also data items B, D, and H. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

# Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency



# Example of Granularity Hierarchy

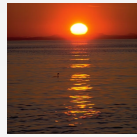



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

# Intention Lock Modes

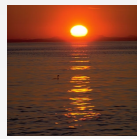
- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.



# Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

| | IS | IX | S | S IX | X |
|---|---|---|---|---|---|
| IS | ✓ | ✓ | ✓ | ✓ | × |
| IX | ✓ | ✓ | × | × | × |
| S | ✓ | × | ✓ | × | × |
| S IX | ✓ | × | × | × | × |
| X | × | × | × | × | × |



# Multiple Granularity Locking Scheme

- Transaction $T_i$ can lock a node $Q$, using the following rules:
  - The lock compatibility matrix must be observed.
  - The root of the tree must be locked first, and may be locked in any mode.
  - A node $Q$ can be locked by $T_i$ in S or IS mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or IS mode.
  - A node $Q$ can be locked by $T_i$ in X, SIX, or IX mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or SIX mode.
  - $T_i$ can lock a node only if it has not previously unlocked any node (that is, $T_i$ is two-phase).
  - $T_i$ can unlock a node $Q$ only if none of the children of $Q$ are currently locked by $T_i$.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.