Failure, Recovery and Concurrency Control

UNIT – V

Database Management System

Reference: Database Systems by Elmasri Navathe, Pearson Publicatoins
Database Concepts, Korth, Silbertz, Sudarshan

Transaction Processing

Transaction Processing Concepts

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution, the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Desirable Properties of transaction (ACID)

- **Atomicity:** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, 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.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example

Transaction to transfer Rs. 5000 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 5000
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 5000
6. **write**(*B)*

- **Atomicity requirement** - if the transaction fails after step 3 And before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.

Contd….

- **Isolation requirement** - if between steps 3 and 6, another transaction is allowed to access partially updated database, it will see an inconsistent database.

  - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
  - However, executing multiple transactions concurrently has significant benefits.

- **Durability requirement** - once the user has been notified that the transaction has completed (i.e., the transfer of the Rs. 5000 has taken place), the updates to the database by the transaction must persist despite failures.

Transaction 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.

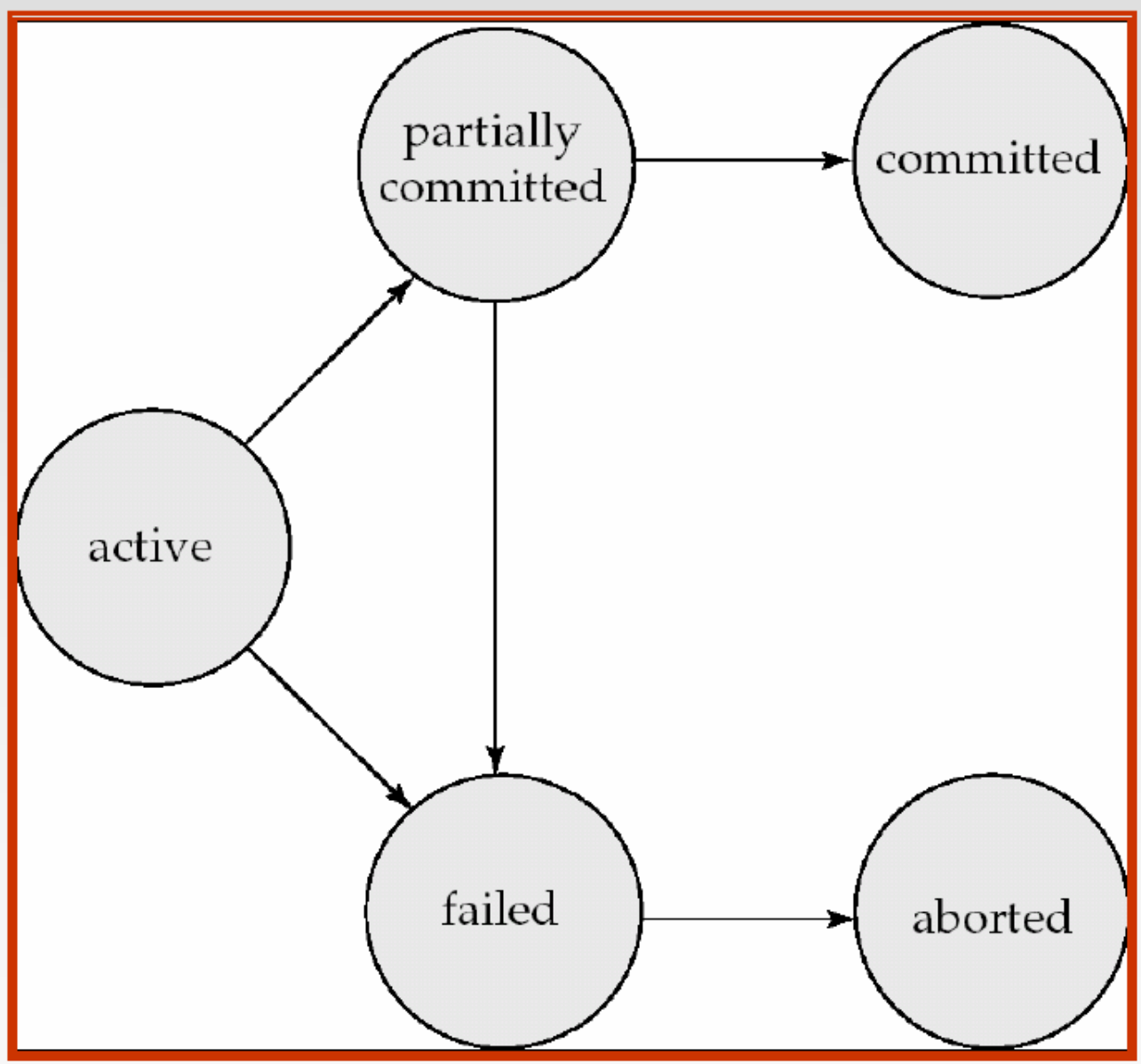
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.

  Two options after it has been aborted:

  - restart the transaction; can be done only if no internal logical error
  - kill the transaction

- **Committed** – after successful completion.

State Diagram



Schedule

- A sequence 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.

- A transaction that fails to successfully complete its execution will have an abort instructions as the last statement .

Schedule

Suppose the current values of accounts *A* and *B* are $1000 and $2000, respectively.

Suppose also that the two transactions are executed one at a time in the order *T*1 followed by *T*2. This execution sequence appears in Figure 1.0 In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of *T*1 appearing in the left column and instructions of *T*2 appearing in the right column.

The final values of accounts *A* and *B*, after the execution in Figure 1.0 takes place respectively. Thus, the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*—is preserved after the execution of both transactions.

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)

Fig:1.0

Schedule – 2

A serial schedule where *T2* 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)	

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system.

Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction.

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

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)

Figure 1.2 Schedule 3
A concurrent schedule equivalent to schedule 1.

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)

Figure 1.0 Schedule 1

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 1.2 After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order $T1$ followed by $T2$.

The sum $A + B$ is indeed 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)

Concurrency Control

One of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.

To ensure that , 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**.

The concurrency-control schemes, are all based on the serializability property. That is, all the schemes presented here ensure that schedules are serializable.

Why Concurrency Control is needed:

- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason.
  - The updated item is accessed by another transaction before it is changed back to its original value.

- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Time

T1	T2
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);	

Fig(a): The lost update problem: *Item X has an incorrect value because its update by T1 is lost*

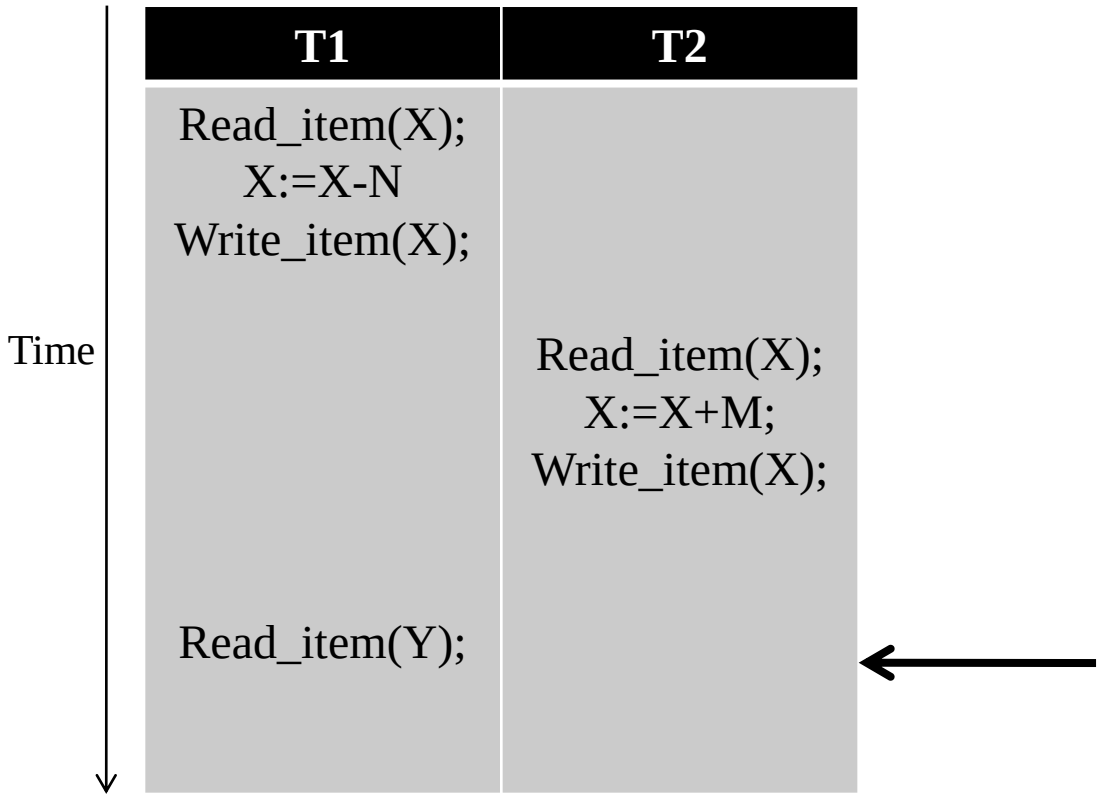


Fig (b): The temporary update problem: *Transaction T1 fails and must change the value of X back to its old value; meanwhile T2 has read the temporary incorrect value of X.*

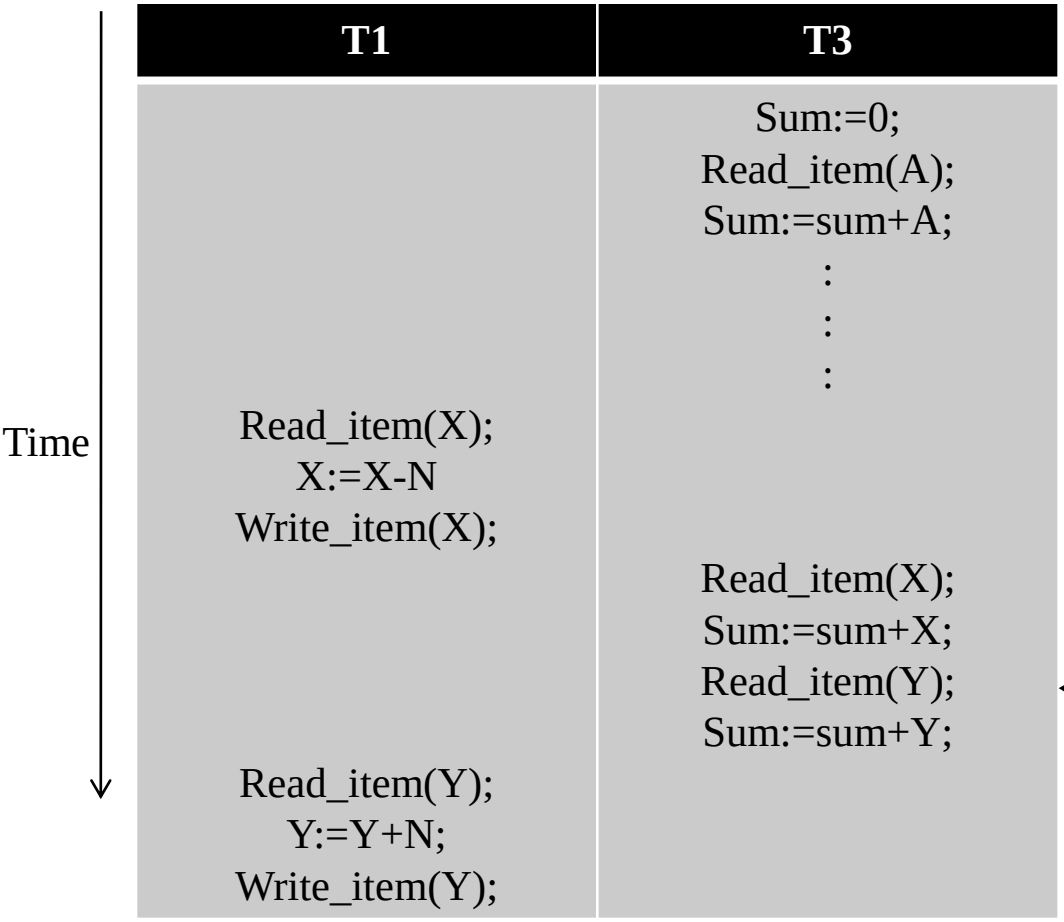


Fig (b): A wrong Summary problem: *T3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result.*

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:
  - **Conflict serializability**
  - **View 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.

Conflicting Instructions

- Instructions *Ii* and *Ij* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *Ii* and *Ij*, and at least one of these instructions wrote *Q.*
  1. *Ii* = **read**(*Q*), *Ij* = **read**(*Q*). *Ii* and *Ij* don't conflict.
  2. *Ii* = **read**(*Q*), *Ij* = **write**(*Q*). They conflict.
  3. *Ii* = **write**(*Q*), *Ij* = **read**(*Q*). They conflict
  4. *Ii* = **write**(*Q*), *Ij* = **write**(*Q*). They conflict

- Intuitively, a conflict between *Ii* and *Ij* forces a (logical) temporal order between them.
  - If *Ii* and *Ij* 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.

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.

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

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

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* .**

(*Conditions 1 and 2 ensure that each transaction reads the same values in both schedules, and therefore performs the same computation)*

View Serializability

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´*.

   As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

(*Condition 3, coupled with conditions 1 and 2 , ensures that both schedules result in the same final system state.*

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.
- Below is a schedule which is view-serializable but *not* conflict serializable.

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

Let value of A=100 and B=100

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)

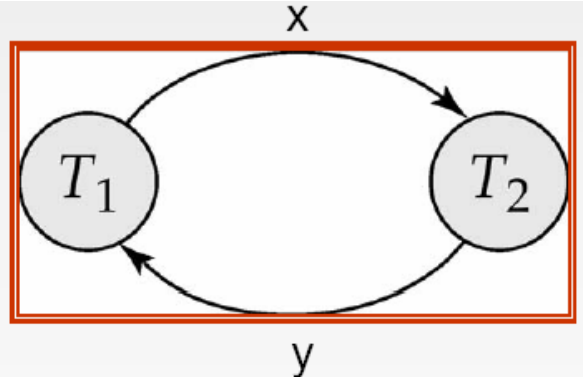
Testing of Serializability

- Consider some schedule of a set of transactions *T*1, *T*2, ..., *Tn*

- **Precedence graph** – a direct graph where the vertices are the transactions (names).

- We draw an arc from *Ti* to *Tj* if the two transaction conflict, and *Ti* accessed the data item on which the conflict arose earlier.

- We may label the arc by the item that was access



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.*

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.

Testing for Serializability

The set of edges consists of all edges *Ti* →*Tj for which* one of three conditions holds:

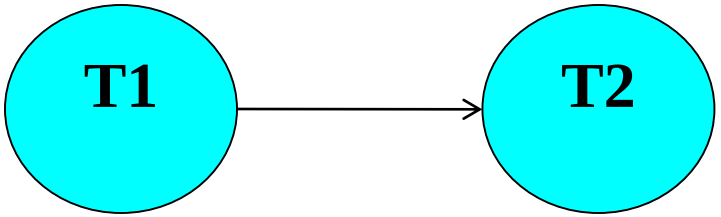
1. ***Ti executes write(Q) before Tj executes read(Q).***
2. ***Ti executes read(Q) before Tj executes write(Q).***
3. ***Ti executes write(Q) before Tj executes write(Q).***

T1	*T2*
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*)

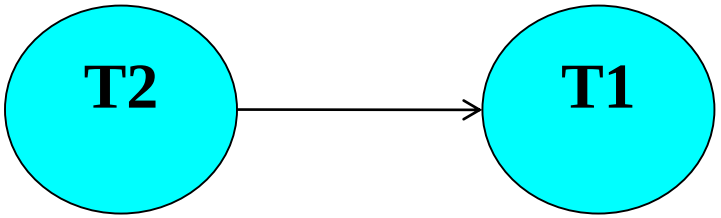
Figure : Schedule 1— a serial schedule in which *T1 is followed by T2*.

T1	*T2*
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*)

Figure : Schedule 2—a serial schedule in which *T2 is followed by T1*.



(a)



(b)

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* .

T1	T2
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*)

Figure : Schedule 4 -a concurrent schedule.

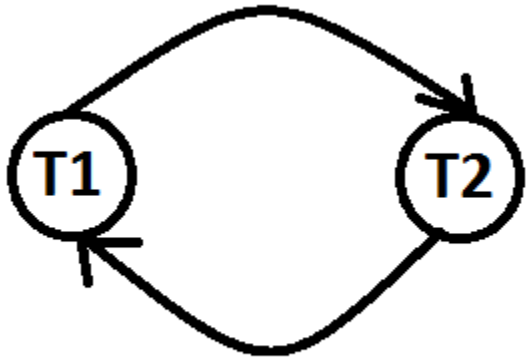


Figure 15.16 Precedence graph for schedule 4.

Why a Transaction fails

1. **A computer failure (system crash):**

   A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. **A transaction or system error:**

   Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Why a Transaction fails contd..

3. Local errors or exception conditions detected by the transaction:

Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

5. Disk failure:

Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:

This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator etc.

Recovery from a Transaction failure

If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the automacity property of the transaction.

In the system that allows concurrent execution , it is necessary also to ensure that any transaction Tj that is dependent on Ti is also aborted.

To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

Recoverable Schedules

Consider the schedule in the fig 2.0, in which T9 is a transaction that performs only one instruction : read(A)

Suppose that the system allows T9 to commit immediately after executing the read(A) instruction . Thus, T9, commits before T8 does.

Now suppose that T8 fails before it commits. Since T9 has read the value of data item A written by T8, we must abort T9 to ensure transaction atomicity.

However T9 has already committed and cannot be aborted. This situation is called Unrecoverable schedules.

T8	T9
Read(A)	
Write(A)	
	read(A)
Read(B)	

Fig:2.0

Recoverable Schedules

Schedule shown in the below Figure 2.0 with the commit happening immediately after the read(A) instruction, is an example of a ***nonrecoverable*** schedule, which should not be allowed.

Most database systems require that all schedules be recoverable.

*A **recoverable schedule** is one where, for each pair of transaction 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.*

Ti	Tj
Read(A)	
Write(A)	
	read(A)
Read(B)	

Fig:2.1

Cascade Schedules

T10	T11	T12
Read(A)		
Read(B)		
Write(A)		
	read(A)	
	write(A)	
		read(A)

Suppose that, T10 fails , T10 must be rolled back. Since T11 is dependent on T10, T11 must be rolled back. Since T12 is dependent on T11, T12 must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called ***cascading rollback.***

Cascadeless Schedules

Cascading rollback is undesirable, since it leads to the un doing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called ***cascadless*** schedules.

Formally, a cascadless 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 read operation of Tj.

It is easy to verify that every cascadless schedule is also recoverable.

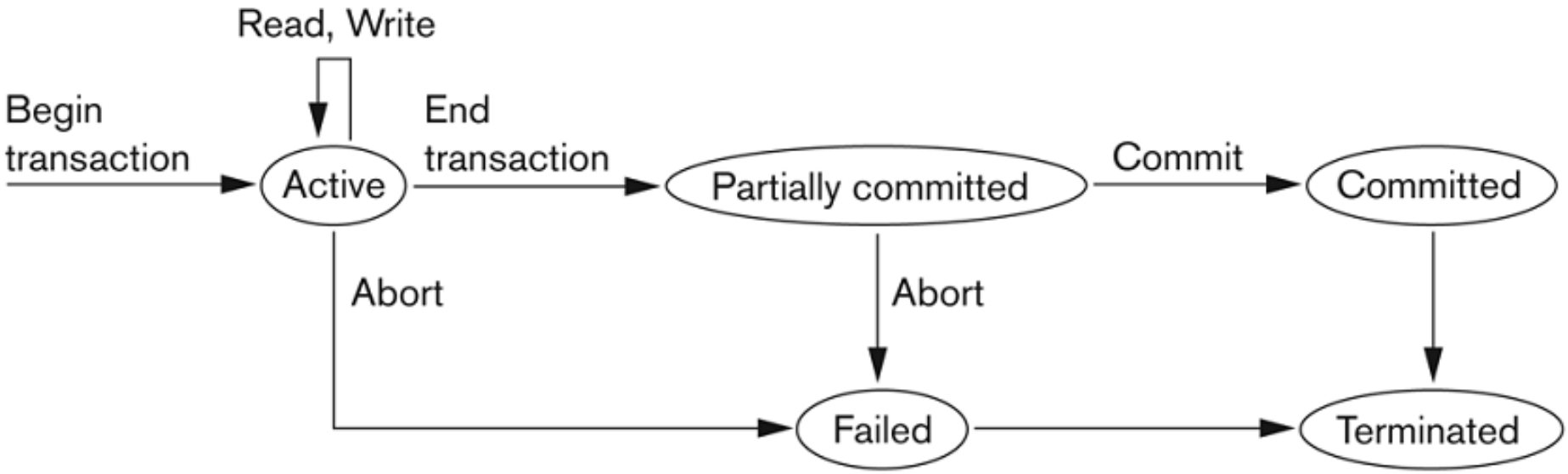
Recovery from a Transaction failure

- Recovery manager keeps 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 limit of transaction execution.
    - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

Contd…

- Recovery manager keeps track of the following operations (cont):
  - **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 may have applied to the database must be undone.
- **Recovery techniques use the following operators:**
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

State transition diagram illustrating the states for transaction execution



Log-Based Recovery

- The System Log
  - **Log** or **Journal**: The log keeps track of all transaction operations that affect the values of database items.
    - This information may be needed to permit recovery from transaction failures.
    - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
    - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

Contd…

- The System Log (cont):
  - **T** in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:
  - Types of log record:
    - [**start_transaction,T**]: Records that transaction T has started execution.
    - [**write_item,T,X,old_value,new_value**]: Records that transaction T has changed the value of database item X from old_value to new_value.
    - [r**ead_item,T,X**]: Records that transaction T has read the value of database item X.
    - [**commit,T**]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - [**abort,T**]: Records that transaction T has been aborted.

Contd….

- The System Log (cont):
  - Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log*, whereas other protocols require these entries for recovery.

Log-Based Recovery

- If the system crashes, we can recover to a consistent database state by examining the log.
  1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
  2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Precedence Graph

A serializiblity order of the transactions can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph.

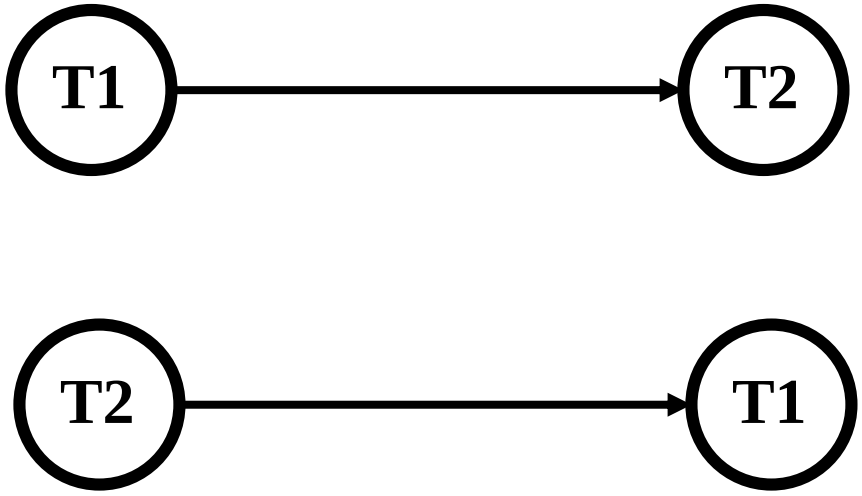
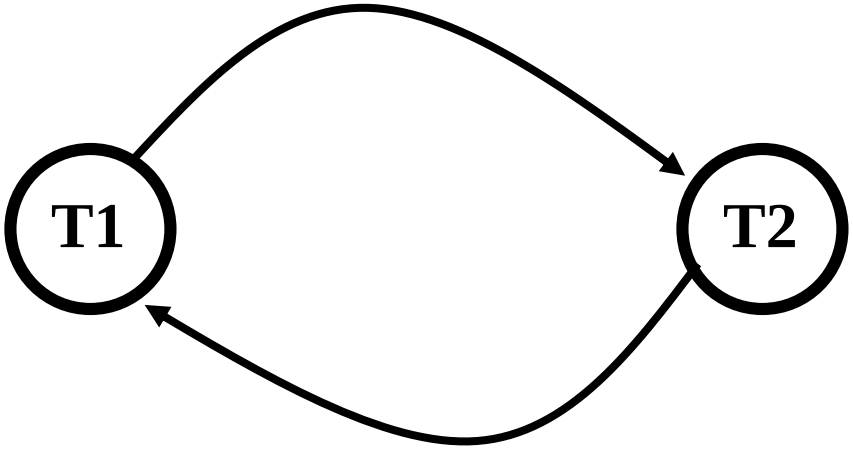


Fig: Precedence Graph

Precedence Graph for the Schedule S1



Thus to test for conflict serializiblity , we need to construct the precedence graph and

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)

Fig: Schedule S1

Checkpoints

A Checkpoint is like a snapshot of the DBMS state. By taking checkpoints, the DBMS can reduce the amount of work to be done during restart in the event of subsequent crashes.

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.

In principle, we need to search the entire log to determine this information.

There are two major difficulties with this approach:

1. The search process is time consuming.

2. Most of the transactions that, according to the algorithm, need to be redone have already written their updates into the database.

Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

Checkpoints contd..

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system.

At time passes log file may be too big to be handled at all. ***Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in storage disk.***

Checkpoint declares a point before which the DBMS was in consistent state and all the transactions were committed.

To reduce these types of overhead, checkpoints are used. The system periodically performs **checkpoints, which require** the following sequence of actions to take place:

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

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

The presence of a *<checkpoint> record in the log allows the system to streamline* its recovery procedure.

Techniques of Checkpoints

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

✓A transaction is said to be partially committed once the final action of the transaction has been executed. The deferred-modification technique assumes that transactions are executed serially.

✓When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes.

✓If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

Immediate Database Modification

The **immediate-modification technique allows database modifications to be output** to the database while the transaction is still in the active state.

Data modifications written by active transactions are called **uncommitted modifications. In the event** of a crash or a transaction failure, the system must use the old-value field of the log records to restore the modified data items to the value they had prior to the start of the transaction.

Immediate Database Modification contd….

The undo operation accomplishes this restoration.

Before a transaction *Ti starts its* execution, the system writes the record <*Ti start*> to the log.

During its execution, any write(*X) operation by Ti is preceded by the writing* of the appropriate new update record to the log.

When *Ti partially commits*, *the* system writes the record <*Ti commit*> *to the log*.

Immediate Database Modification contd….

Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage.

We therefore require that, before execution of an output(*B) operation, the log records corresponding to B be written* onto stable storage.

Transactions, Read and Write Operations, and DBMS Buffers

A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**

The model of a database that is used to explain transaction processing concepts is much simplified. A **database** is basically represented as a collection of **named data items.** The size of a data item is called its **granularity,** and it can be a field of some record in the database, or it may be a larger unit such as a record or even a whole disk block.

Transactions, Read and Write Operations, and DBMS Buffers

The basic database access operations that a transaction can include are as follows:

- **read_item(*X*):** Reads a database item named *X* into a program variable. To simplify our notation, we assume that *the program variable is also named X.*
- **write_item(*X*):** Writes the value of program variable *X* into the database item named *X*.

Executing a read_item(X) command includes the following steps:

1. Find the address of the disk block that contains item *X*.
2. Copy that disk block into a buffer in main memory
3. Copy item *X* from the buffer to the program variable named *X*.

Transactions, Read and Write Operations, and DBMS Buffers

Executing a write_item(X) command includes the following steps:

1. Find the address of the disk block that contains item *X*.
2. Copy that disk block into a buffer in main memory
3. Copy item *X* from the program variable named *X* into its correct location in the buffer.
4. Store the updated block from the buffer back to disk.

Why Concurrency Control is Needed?

To over come some of the following problems:

- The Lost Update Problem
- The Temporary Update (or Dirty Read) Problem
- The Incorrect Summary Problem.

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight.

Concurrency Control Techniques

Concurrency Control techniques are used to ensure the non-interference or isolation property of concurrency executing transactions.

Most of these techniques ensure serializability of schedules, using protocols that guarantee serializablity .

One important set of protocols employs the technique of locking data items to prevent multiple transactions from accessing the items concurrently .

Another set of concurrency control protocols use timestamps. A time stamp is a unique identifier for each transaction, generated by the system.

Another factor that affects concurrency control is the granularity of the data items-that is, what portion of the database a data item represents. An item can be as small as a single attribute value or as large as a disk block, or even a whole file or the entire database.

Concurrency Control Techniques

Locking : Whenever a transaction is being accessed by a transaction , it must not be modified by any other transactions. In order to ensure this, a transaction needs a ***lock*** on the required data item.

A Lock is a variable associated with each data item that indicates whether a read or write operation can be applied to the data item.

In addition ,it synchronizes the concurrent access of data items.

Database system mainly use two modes of locking:

- ✓ ***Exclusive Lock***
- ✓ ***Shared Locks***

Concurrency Control Techniques

Database system mainly use two modes of locking:

Exclusive Lock

It is denoted by (X) is the commonly used locking strategy that provides a transaction an exclusive control on the data item. A transaction that ***wants to read as well as write*** a data item must acquire exclusive lock on the data item. ***It is also known as update lock.***

Shared Lock

It is denoted by (S) can be acquired on a data item when a transaction wants to ***only read a data item*** and do not modify it. ***It is also known as read lock.***

Concurrency Control Techniques

Exclusive Lock-Example

If a transaction Ti has acquired an Exclusive lock on a data item, say Q, then Ti can both read and write Q.

If another transaction say Tj, requests to access Q, then Tj has to wait for Ti to release its lock on Q.

It means no transaction is allowed to access data item when it is exclusive locked by another transaction. *Lock prevents concurrent transactions from corrupting one another.*

Shared Lock -Example

If a transaction Ti has acquired a shared lock on data item Q, then Ti can read but cannot write on Q. More over any number of transactions can acquire shared lock on the same data item simultaneously without any risk of interference between tracsactions.

IMPLEMENTATION OF LOCK

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks).

Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction. The lock manager uses this data structure:

For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**.

Fig: 5.1 Lock Table

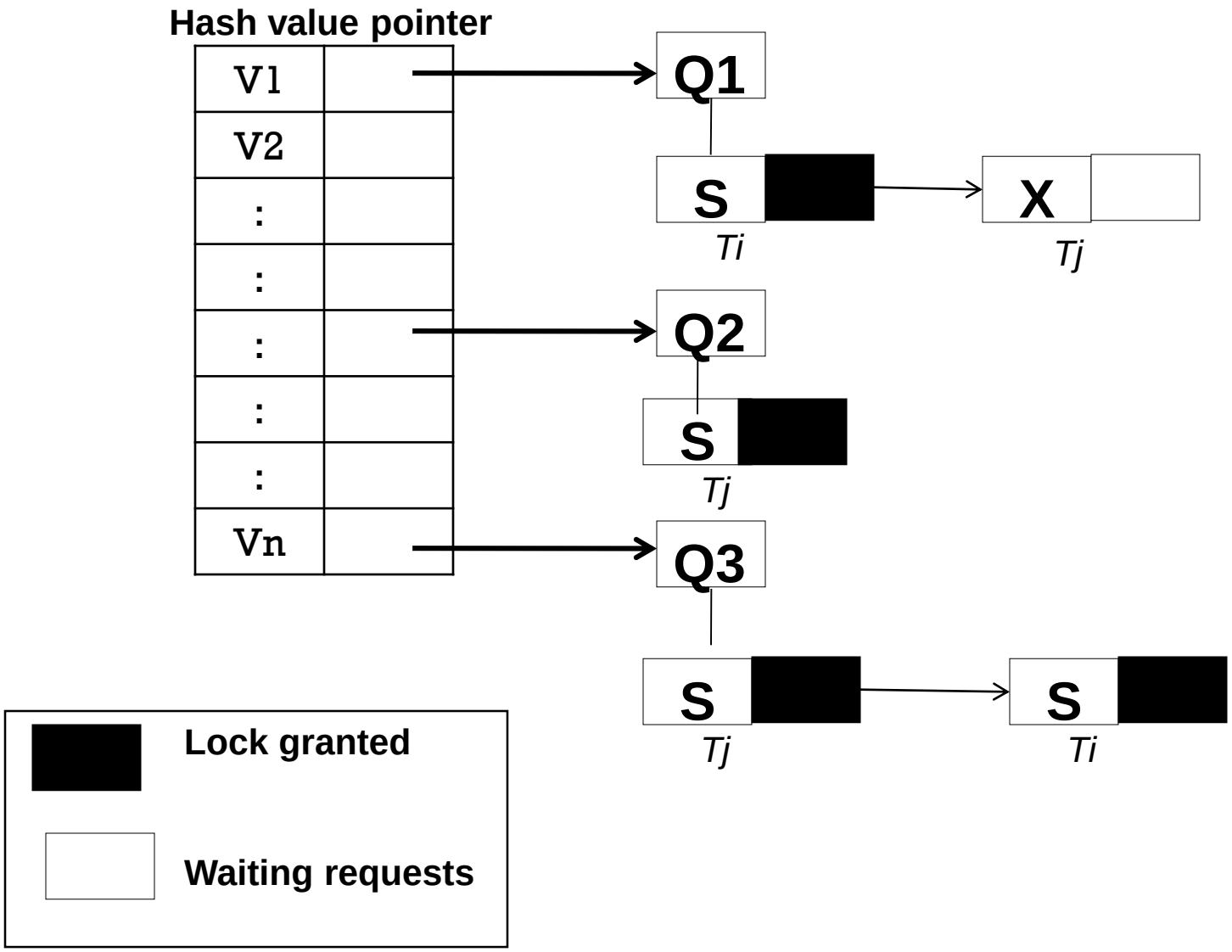


Fig5.2: Deadlock Situation

T3	T4
Lock-X(R)	
……	
	lock-S(Q)
	……
	lock-S(R)
Lock-X(Q)	

Note: Deadlock is a situation that occurs when all the transactions in a set of two or more are in a simultaneous wait state and each of them is waiting for the release of a data item held by one of the other waiting transaction in a set. None of the transaction can proceed until at least one of the waiting transaction releases lock on the data item.

Two-Phase Locking

It requires that each transaction be divided into two phases.
During First Phase, ***the transaction acquires all the locks;***

During Second Phase, ***the transaction releases all the locks .***

The phase during which locks are acquired is **growing or expanding phase. In this phase, the number of locks held by a transaction increases from zero to maximum.**

On the other hand, **a phase during which locks are released is shrinking or contracting phase. In this phase, the number of locks held by a transaction decreases from maximum to zero.**

Whenever a transaction releases a lock on a data item, it enters into the shrinking phase.

Two-Phase Locking contd….

Until all the required locks on the data items are acquired, the release of the locks must be delayed.

Thus, the two-phase locking leads to lower degree of concurrency among transactions.

The point in the schedule at which the transaction successfully acquires its last lock is called the* lock point *of the transaction.

Fig.5.3: Transactions T1 and T2 in Two-Phase Locking.

T1	T2
Lock-X(R)	
Read(R)	
R:=R-200;	
Write(R)	
Lock-X(Q)	
Read(Q)	
Q:=Q+200;	
Write(Q)	
Unlock(R)	
Unlock(Q)	
	lock-X(sum)
	sum:=0;
	lock-S(Q)
	read(Q)
	sum:=sum+Q;
	lock-S(R)
	read(R)
	write(sum)
	unlock(Q)
	unlock(R)
	unlock(sum)

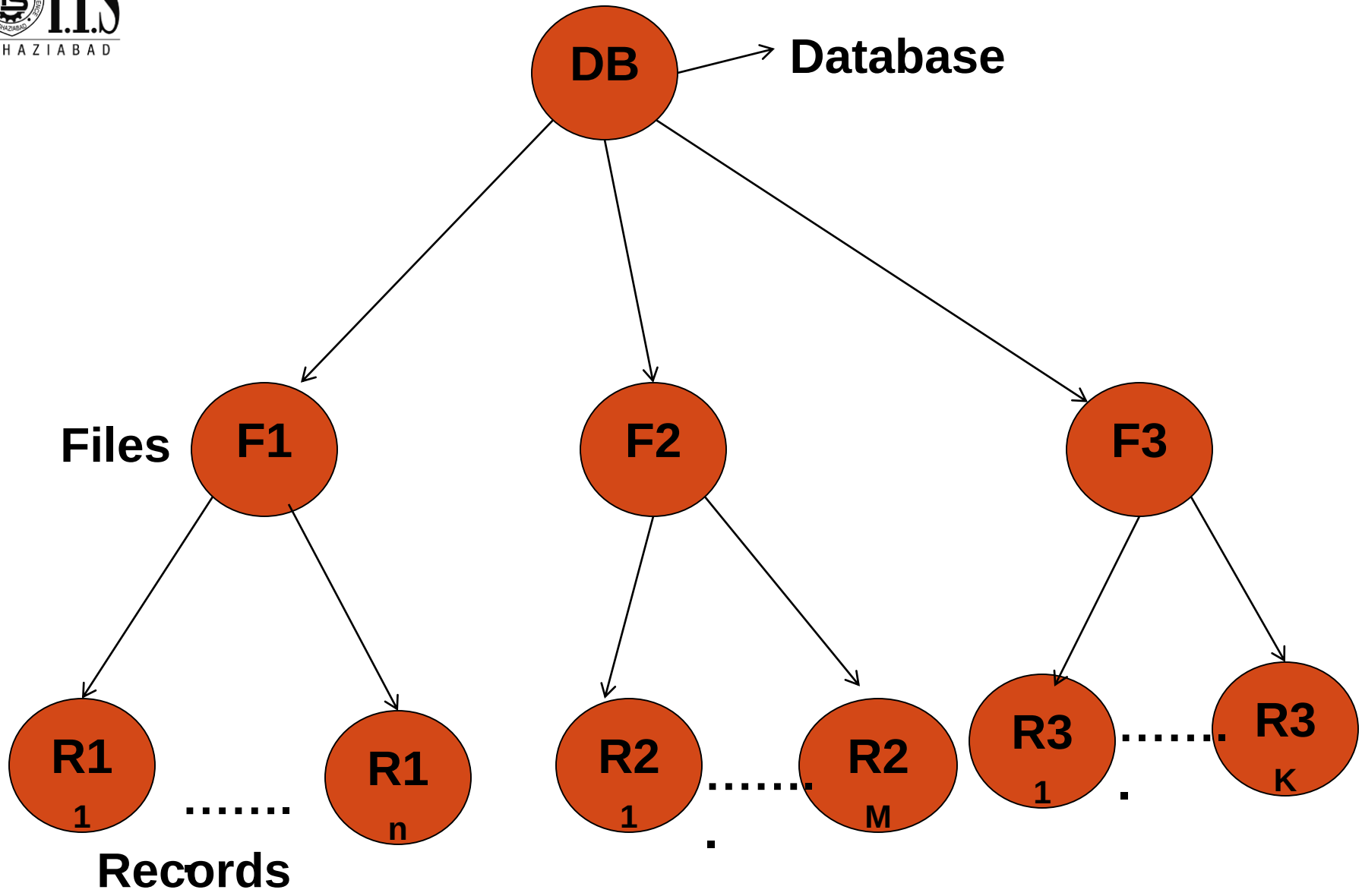


Fig.5.6: Multiple-granularity Tree

Multiple Granularity Locking

Locking granularity is the size of the data item that the lock protects. It is important for the performance of a database system.

Two terms used in Granularity are :
Coarse granularity
Fine Granularity

If the large data item is locked, it is easier for the lock manager to manage the lock.

A transaction requires lock granularity on the basis of the operation being performed.

Since different transactions have dissimilar requests, it is desirable that the database system provides a range of locking granules called multiple-granularity locking