UNIT-4

Transaction Systems

Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions.

Examples of such systems include airline reservations, banking, credit-card processing, stock markets and many other applications.

These systems require high availability and fast response time for hundreds of concurrent users.

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), where it is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only** transaction, otherwise, it is known as read write transaction.

A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**. A **data item** can be a *database record.* 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*.

For Example:

T_1

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

Properties of Transaction

1. Atomicity

A transaction is an atomic unit of processing, it should either be performed in its entirety or not performed at all. The basic idea behind ensuring atomicity is that database system keeps track of old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

2. Consistency

A transaction should be consistency preserving meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.
If the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

3. Isolation

A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
For every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

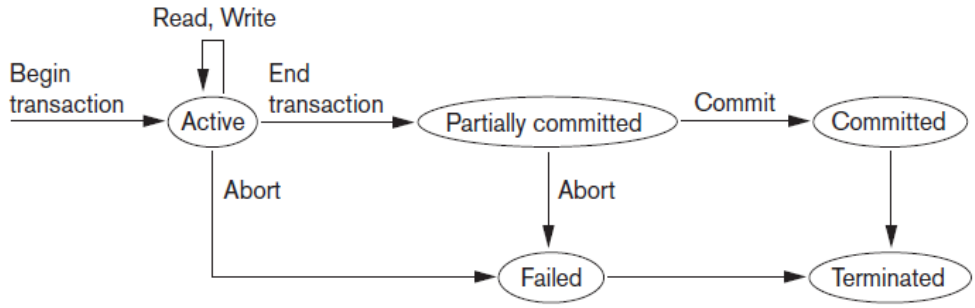
4. Durability

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure. The durability property is the responsibility of the recovery subsystem of the DBMS.

Transaction States

Figure shows a state transition diagram that illustrates how a transaction moves through its execution states.

State transition diagram illustrating the states for transaction execution.



- A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations.

- When the transaction ends, it moves to the **partially committed state**. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.

- Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state**. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

- A transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

- The **terminated state** corresponds to the transaction leaving the system.

- Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions.

Schedules (Histories) of Transactions

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

A **schedule** (or **history**) *S* of *n* transactions *T*1, *T*2, ..., *Tn* is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule *S*. However, for each transaction *Ti* that participates in the schedule *S*, the operations of *Ti* in *S* must appear in the same order in which they occur in *Ti*.

The order of operations in S is considered to be a total ordering, meaning that for any two operations in the schedule, one must occur before the other.

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

The above schedule can be written as:

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

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
(1) they belong to *different transactions*;
(2) they access the *same item X*; and
(3) *at least one* of the operations is a write_item(*X*).

For example, in schedule *Sa*, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$. However, the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; the operations $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items *X* and *Y*; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

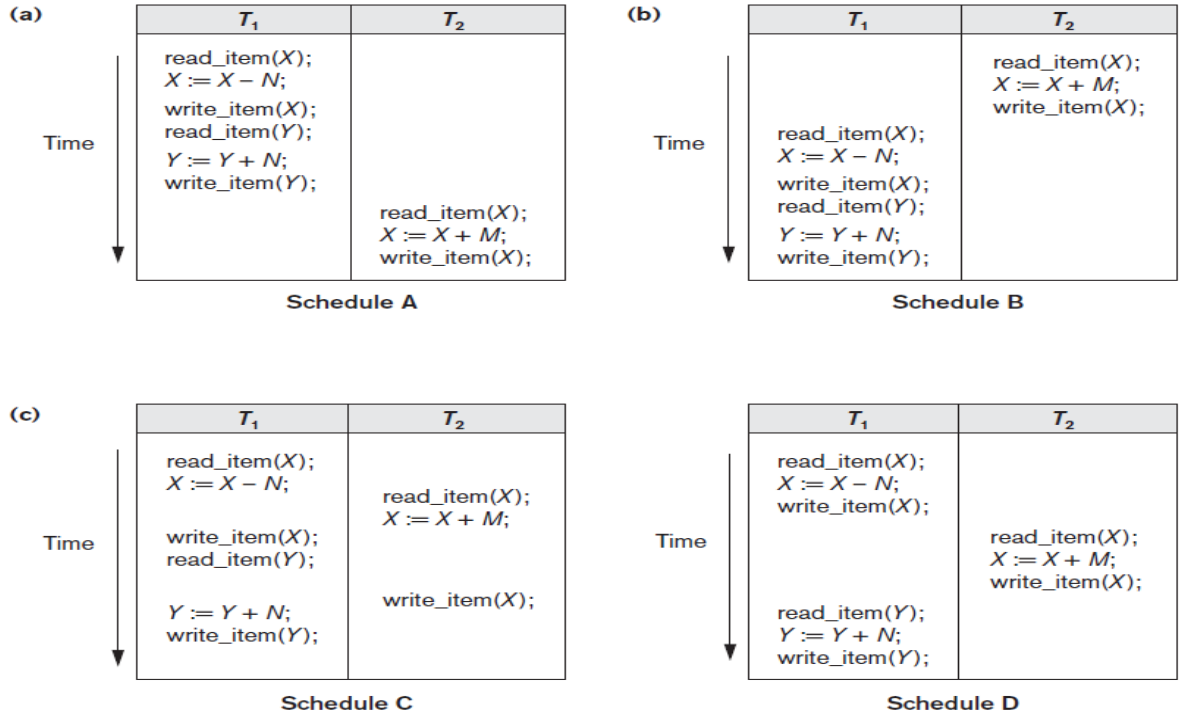
Two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T_1 changes, because in the second order the value of X is changed by $w_2(X)$ before it is read by $r_1(X)$, whereas in the first order the value is read before it is changed. This is called a read-write conflict. The other type is called a write-write conflict, and is illustrated by the case where we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$. For a write-write conflict, the last value of X will differ because in one case it is written by T_2 and in the other case by T_1.

Serial and Non-serial Schedules

A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise the schedule is called non-serial.

For Example:

Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.



Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. There are two good reasons for allowing concurrency:

1) Improved throughput and resource utilization.
2) Reduced waiting time.

When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction.

Let *T*1 and *T*2 be two transactions that transfer funds from one account to another. Transaction *T*1 transfers $50 from account *A* to account *B*. It is defined as

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction *T*2 transfers 10 percent of the balance from account *A* to account *B*. It is defined as

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Suppose the current values of accounts *A* and *B* are $1000 and $2000, respectively.
CASE1:

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 1—a serial schedule in which T_1 is followed by T_2.

The sum $A + B$—is preserved after the execution of both transactions.

CASE2:

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 2—a serial schedule in which T_2 is followed by T_1.

The sum $A + B$—is preserved after the execution of both transactions.

CASE3:

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)

Schedule 3—a concurrent schedule equivalent to schedule 1.

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.

CASE4:

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)

After the execution of this schedule, we arrive at a state where the final values of accounts *A* and *B* are $950 and $2100, respectively. This final state is an *inconsistent state*, since we have gained $50 in the process of the concurrent execution. Indeed, the sum *A* + *B* is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state. It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The **concurrency-control component** of the database system carries out this task.

Serializability

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

Conflict Serializability

Let us consider a schedule *S* in which there are two consecutive instructions *Ii* and *Ij*, of transactions *Ti* and *Tj* , respectively (*i != j*). If *Ii* and *Ij* refer to different data items, then we can swap *Ii* and *Ij* without affecting the results of any instruction in the schedule. However, if *Ii* and *Ij* refer to the same data item *Q*, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. *Ii* = read(*Q*), *Ij* = read(*Q*). The order of *Ii* and *Ij* does not matter, since the same value of *Q* is read by *Ti* and *Tj* , regardless of the order.
2. *Ii* = read(*Q*), *Ij* = write(*Q*). If *Ii* comes before *Ij*, then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *Ij*. If *Ij* comes before *Ii*, then *Ti* reads the value of *Q* that is written by *Tj*. Thus, the order of *Ii* and *Ij* matters.

3. *Ii* = write(*Q*), *Ij* = read(*Q*). The order of *Ii* and *Ij* matters for reasons similar to those of the previous case.
4. *Ii* = write(*Q*), *Ij* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either *Ti* or *Tj* . However, the value obtained by the next read(*Q*) instruction of *S* is affected.

To illustrate the concept of conflicting instructions, we consider schedule 3,

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

The write(*A*) instruction of *T*1 conflicts with the read(*A*) instruction of *T*2. However, the write(*A*) instruction of *T*2 does not conflict with the read(*B*) instruction of *T*1, because the two instructions access different data items. Let *Ii* and *Ij* be consecutive instructions of a schedule *S*. If *Ii* and *Ij* are instructions of different transactions and *Ii* and *Ij* do not conflict, then we can swap the order of *Ii* and *Ij* to produce a new schedule *S'*.

We expect *S* to be equivalent to *S'*, since all instructions appear in the same order in both schedules except for *Ii* and *Ij*, whose order does not matter.

Since the write(*A*) instruction of *T*2 in schedule 3 does not conflict with the read(*B*) instruction of *T*1, we can swap these instructions to generate an equivalent schedule, schedule 5.

Schedule 5

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

We continue to swap non-conflicting instructions:

• Swap the read(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.
• Swap the write(*B*) instruction of *T*1 with the write(*A*) instruction of *T*2.
• Swap the write(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.

The final result of these swaps, schedule 6, is a serial schedule.

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

If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of nonconflicting instructions, we say that *S* and *S'* are **conflict equivalent**.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Testing for Conflict Serializability of a Schedule

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

Algorithm. Testing Conflict Serializability of a Schedule *S*

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

If there is a cycle in the precedence graph, schedule S is not (conflict) Serializable, if there is no cycle, S is serializable.

If there is no cycle in the precedence graph, we can create an **equivalent serial schedule** *S*' that is equivalent to *S*, by ordering the transactions that participate in *S*.

A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This procedd is called **topological sorting**. There are several possible linear orders that can be obtained through a topological sort.

1. Identify the node whose indegree is zero, take this node in the execution order of transaction.

2. Now, eliminate the outgoing edges from this node, and identify the new node whose indegree is zero. If there are more than one node then next executing transaction will be any one of them and more than one serializability order exist.

3. Now, repeat step 2 till there is an edge in the graph.

For Example:

Q1: Check whether the given schedule S is conflict serializable or not-

S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$

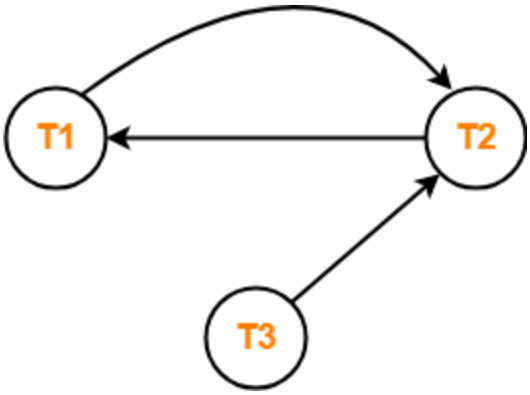
Sol: Step-01:

List all the conflicting operations and determine the dependency between the transactions-

$R_2(A)$, $W_1(A)$	$(T_2 \rightarrow T_1)$
$R_1(B)$, $W_2(B)$	$(T_1 \rightarrow T_2)$
$R_3(B)$, $W_2(B)$	$(T_3 \rightarrow T_2)$

Step-02:

Draw the precedence graph-



Clearly, there exists a cycle in the precedence graph.

Therefore, the given schedule S is not conflict serializable.

Q2: Check whether the given schedule S is conflict serializable or not. If yes, then determine all the possible serialized schedules-

T1	T2	T3	T4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

Solution-

Checking Whether S is Conflict Serializable Or Not-

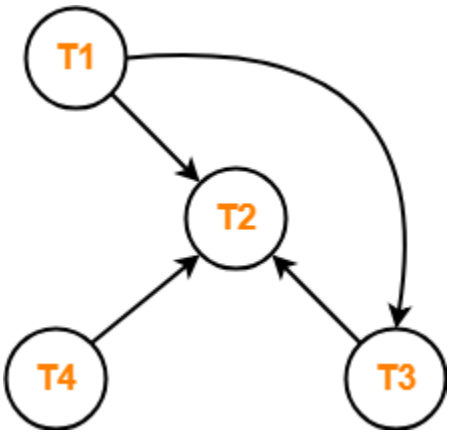
Step-01:

List all the conflicting operations and determine the dependency between the transactions-

$R_4(A)$, $W_2(A)$ $(T_4 \rightarrow T_2)$
$R_3(A)$, $W_2(A)$ $(T_3 \rightarrow T_2)$
$W_1(B)$, $R_3(B)$ $(T_1 \rightarrow T_3)$
$W_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
$R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Step-02:

Draw the precedence graph-



- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.

Finding the Serialized Schedules-

- All the possible topological orderings of the above precedence graph will be the possible serialized schedules.
- The topological orderings can be found by performing the **Topological Sort** of the above precedence graph.

After performing the topological sort, the possible serialized schedules are-

1. $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$
2. $T_1 \rightarrow T_4 \rightarrow T_3 \rightarrow T_2$
3. $T_4 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2$

View Serializability

Consider two schedules *S* and *S'*, where the same set of transactions participates in both schedules. The schedules *S* and *S'* are said to be **view equivalent** if three conditions are met:

1. Initial Read should be same in both schedules corresponding to Ti.
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. Updated read should be same in both schedules.
For each data item *Q*, if transaction *Ti* executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction *Tj* , then the read(*Q*) operation of transaction *Ti* must, in schedule *S'*, also read the value of *Q* that was produced by the same write(*Q*) operation of transaction *Tj*.

3. Final write should be same in both schedules.
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'*.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule *S* is **view serializable** if it is view equivalent to a serial schedule.

For Example: Consider two schedules S and U:

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

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

- In S both T1 & T2 are reading A and in U T1 & T2 are also reading A firstly, thus condition 1 is satisfied.

- As T1 & T2 in S are firstly reading A then in U T1 & T2 are writing A also after reading A, thus condition 2 is satisfied.
- Finally, in S T1 & T2 are writing B which is also same for U, thus condition 3 is satisfied.

Hence, serial schedule S is view equivalent to U, thus they are view serializable.

Every conflict-serializable schedule is also view serializable, but there are viewserializable schedules that are not conflict serializable.

The definitions of conflict serializability and view serializability are similar if a condition known as the **constrained write assumption** (or **no blind writes**) holds on all transactions in the schedule. This condition states that any write operation *wi*(*X*) in *Ti* is preceded by a *ri*(*X*) in *Ti* and that the value written by *wi*(*X*) in *Ti* depends only on the value of *X* read by *ri*(*X*). This assumes that computation of the new value of *X* is a function *f*(*X*) based on the old value of *X* read from the database. A **blind write** is a write operation in a transaction *T* on an item *X* that is not dependent on the value of *X*, so it is not preceded by a read of *X* in the transaction *T*.

The definition of view serializability is less restrictive than that of conflict serializability under the **unconstrained write assumption**, where the value written by an operation *wi*(*X*) in *Ti* can be independent of its old value from the database. This is possible when *blind writes* are allowed, and it is illustrated by the following schedule S_g of three transactions *T*1: *r*1(*X*); *w*1(*X*); *T*2: *w*2(*X*); and *T*3: *w*3(*X*):

S_g: *r*1(*X*); *w*2(*X*); *w*1(*X*); *w*3(*X*); *c*1; *c*2; *c*3;

Recoverability

If a transaction *Ti* fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction *Tj* that is dependent on *Ti* (that is, *Tj* has read data written by *Ti*) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

There are two types of schedules that are permitted in the system in order to recover from transaction failures.

1. Recoverable Schedules

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*.

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

*T*9 is a transaction that performs only one instruction: read(*A*). Suppose that the system allows *T*9 to commit immediately after executing the read(*A*) instruction. Thus, *T*9 commits before *T*8 does. Now suppose that *T*8 fails before it commits. Since *T*9 has read the value of data item *A* written by *T*8, we must abort *T*9 to ensure transaction atomicity. However, *T*9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of *T*8.

Above schedule, with the commit happening immediately after the read(*A*) instruction, is an example of a *nonrecoverable* schedule, which should not be allowed. Most database system require that all schedules be *recoverable*.

2. Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction *Ti*, we may have to roll back several transactions. Such situations occur if transactions have read data written by *Ti*.

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

Transaction *T*10 writes a value of *A* that is read by transaction *T*11. Transaction *T*11 writes a value of *A* that is read by transaction *T*12. Suppose that, at this point, *T*10 fails. *T*10 must be rolled back. Since *T*11 is dependent on *T*10, *T*11 must be rolled back. Since *T*12 is dependent on *T*11, *T*12 must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless 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 cascadeless schedule is also recoverable.

Log-Based Recovery

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database. There are several types of log records. An **update log record** describes a single database write. It has these fields:

• **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
• **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
• **Old value** is the value of the data item prior to the write.

• **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as:

• <*Ti* start>. Transaction *Ti* has started.
• <*Ti*, *Xj*, *V*1, *V*2>. Transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and will have value *V*2 after the write.
• <*Ti* commit>. Transaction *Ti* has committed.
• <*Ti* abort>. Transaction *Ti* has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records.

Deferred Database Modification

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

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.

The execution of transaction *Ti* proceeds as follows. Before *Ti* starts its execution, a record <*Ti* start> is written to the log. A write(*X*) operation by *Ti* results in the writing of a new record to the log. Finally, when *Ti* partially commits, a record <*Ti* commit> is written to the log.

When transaction *Ti* partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage.

Let *T*0 be a transaction that transfers $50 from account *A* to account *B*:

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let T1 be a transaction that withdraws $100 from account C:

T_1: read(C);
$C := C - 100$;
write(C).

Suppose that these transactions are executed serially, in the order *T*0 followed by *T*1, and that the values of accounts *A*, *B*, and *C* before the execution took place were $1000, $2000, and $700, respectively.

The portion of the log containing the relevant information on these two transactions appears as:

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme uses the following recovery procedure:

• redo(*Ti*) sets the value of all data items updated by transaction *Ti* to the new values.

The redo operation must be **idempotent**; that is, executing it several times must be equivalent to executing it once.

After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction *Ti* needs to be redone if and only if the log contains both the record <*Ti* start> and the record <*Ti* commit>.

For Example:

(a)	(b)	(c)
<T_0 start> <T_0, A, 950> <T_0, B, 2050>	<T_0 start> <T_0, A, 950> <T_0, B, 2050> <T_0 commit> <T_1 start> <T_1, C, 600>	<T_0 start> <T_0, A, 950> <T_0, B, 2050> <T_0 commit> <T_1 start> <T_1, C, 600> <T_1 commit>
No Redo Takes Place	Redo (T0):{A=950, B=2050}	Redo (T0): {A=950, B=2050} Redo (T1): {C=600}

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.

The portion of the log containing the relevant information concerning these two transactions appears as:

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

• **undo(*Ti*)** restores the value of all data items updated by transaction *Ti* to the old values.
• **redo(*Ti*)** sets the value of all data items updated by transaction *Ti* to the new values.

After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

• Transaction *Ti* needs to be undone if the log contains the record <*Ti* start>, but does not contain the record <*Ti* commit>.
• Transaction *Ti* needs to be redone if the log contains both the record <*Ti* start> and the record <*Ti* commit>.

For Example:

(a)	(b)	(c)
<T_0 start> <T_0, A, 1000, 950> <T_0, B, 2000, 2050>	<T_0 start> <T_0, A, 1000, 950> <T_0, B, 2000, 2050> <T_0 commit> <T_1 start> <T_1, C, 700, 600>	<T_0 start> <T_0, A, 1000, 950> <T_0, B, 2000, 2050> <T_0 commit> <T_1 start> <T_1, C, 700, 600> <T_1 commit>
Undo(T0): {B=2000, A=1000}	Undo(T1): {C=700} Redo(T0):{A=950, B=2050}	Redo(T0):{A=950, B=2050} Redo(T1):{C=600}

Checkpoints

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 our 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.

To reduce these types of overhead, we introduce checkpoints. During execution, the system maintains the log. In addition, 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. Consider a transaction *Ti* that committed prior to the checkpoint. For such a transaction, the <*Ti* commit> record appears in the log before the <checkpoint> record. Any database modifications made by *Ti* must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on *Ti*.

After a system crash has occurred, the system examines the log to find the last <checkpoint> record (this can be done by searching the log backward).

The redo and undo operations need to be applied to only transaction *Ti* and all transactions *Tj* that started executing after transaction *Ti*.

For the immediate-modification technique, the recovery operations are:
• For all transactions *Tk* in *T* that have no <*Tk* commit> record in the log, execute undo(*Tk*).
• For all transactions *Tk* in *T* such that the record <*Tk* commit> appears in the log, execute redo(*Tk*).

The undo operation does not need to be applied when the deferred-modification technique is being employed.

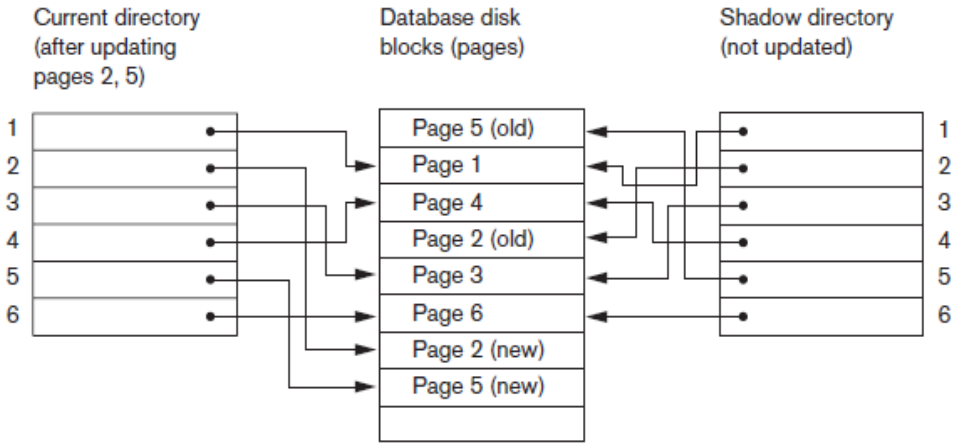
Consider the set of transactions *{T*0, *T*1, . . ., *T*100*}* executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction *T*67. Thus, only transactions *T*67, *T*68, . . ., *T*100 need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

Shadow Paging

This recovery scheme does not require the use of a log in a single-user environment. Shadow paging considers the database to be made up of a number of fixedsize disk pages (or disk blocks)—say, *n*—for recovery purposes. A **directory** with *n* entries is constructed, where the *i*th entry points to the *i*th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it.

When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never* modified. When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten.* Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.



To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory.

Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a **NO-UNDO/ NO-REDO technique** for recovery. In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique.

Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions *{T*0, *T*1, . . ., *Tn}* such that *T*0 is waiting for a data item that *T*1 holds, and *T*1 is waiting for a data item that *T*2 holds, and . . ., and *Tn*−1 is waiting for a data item

that *Tn* holds, and *Tn* is waiting for a data item that *T*0 holds. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.

Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.

The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. There are two main disadvantages to this protocol:

(1) it is often hard to predict, before the transaction begins, what data items need to be locked;
(2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction *T*2 requests a lock that transaction *T*1 holds, the lock granted to *T*1 may be **preempted** by rolling back of *T*1, and granting of the lock to *T*2. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlockprevention schemes using timestamps have been proposed:

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

For example, suppose that transactions *T*22, *T*23, and *T*24 have timestamps 5, 10, and 15, respectively. If *T*22 requests a data itemheld by *T*23, then *T*22 will wait. If *T*24 requests a data item held by *T*23, then *T*24 will be rolled back.

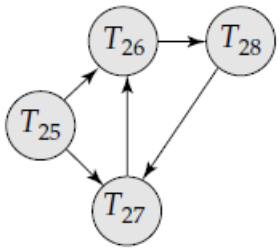
2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp larger than that of *Tj* (that is, *Ti* is younger than *Tj*). Otherwise, *Tj* is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

Returning to our example, with transactions *T*22, *T*23, and *T*24, if *T*22 requests a data item held by *T*23, then the data item will be preempted from *T*23, and *T*23 will be rolled back. If *T*24 requests a data item held by *T*23, then *T*24 will wait.

Deadlock Detection

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

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked.



19 Wait-for graph with a cycle.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. Selection of a victim. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including

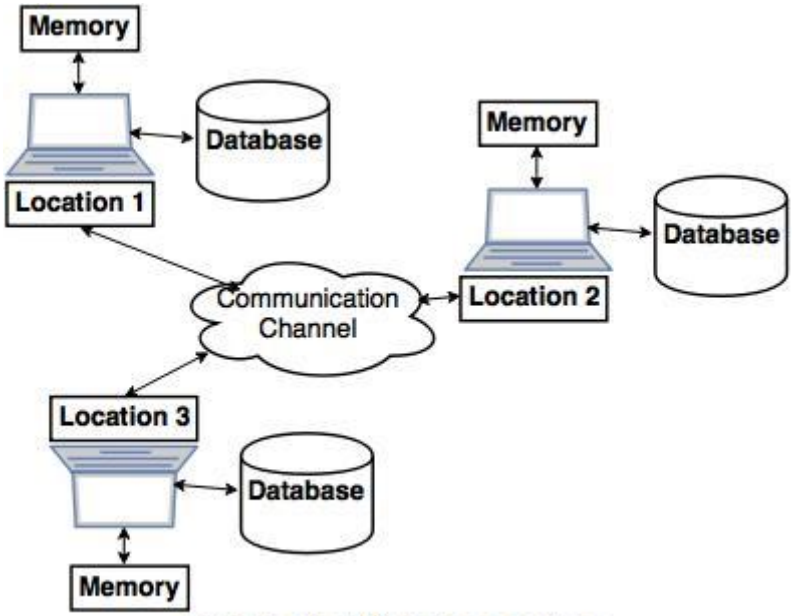
a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
b. How many data items the transaction has used.
c. How many more data items the transaction needs for it to complete.
d. How many transactions will be involved in the rollback.

2. Rollback. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions.

3. Starvation. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that transaction can be picked as a victim only a (small) finite number of times.

Distributed Databases

A distributed database system consists of a collection of sites, connected together by the use of some kind of communication network. A distributed database is fragmented into smaller data sets. Distributed database system is a collection of data with different parts under the control of separate DBMSs running on different computers. These systems do not share the common memory, but communicates among themselves.



Distributted Dtatabase system

Distributed Databases are of two types:

1. Homogeneous Distributed Database

In a homogeneous distributed database, all sites have identical database management system software, are aware of one another and agree to cooperate in processing user's requests. In such a system local sites surrender a portion of their autonomy in terms of their right to change schemas or DBMS software. That software must also cooperate with other sites in exchanging information about transactions, to make transactions processing possible across multiple sites.

2. Heterogeneous Distributed Database

In a heterogeneous distributed database, different sites may use different schemas and different DBMS software. The sites may not be aware of one another and they may provide only limited facilities for cooperation in transaction processing.

Advantages of DDBMS

1. DDBMS allows each site to store and maintain its own database, causing immediate and efficient access to data.

2. It allows to access the data stored at remote sites.

3. If one site is not working due to any reason, the system will not be down because other sites of the network can possibly continue functioning.

4. New sites can be added to the system any time.

5. If a user needs to access the data from multiple sites then the desired query can be sub-divided into sub-queries and evaluated in parallel.

Disadvantages of DDBMS

1. Complex software is required for a distributed database environment.

2. The various sites must exchange messages and perform additional calculations to ensure proper coordination among the sites.

3. If the data are not distributed properly according to their usage, or if queries are not formulated correctly, response to requests for data can be extremely slow.

Distributed Data Storage

Consider a relation *r* that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

1. Data Replication

The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. If relation *r* is replicated, a copy of relation *r* is stored in two or more sites. In the most extreme case, we have **full replication**, in which a copy is stored in every site in the system.

Advantages:
i. Availability
ii. Increased Parallelism

Disadvantages:
i. Increased Overhead on Update

2. Data Fragmentation

Data Fragmentation is a method in which different relations of a relational database system can be sub-divided and distributed among different network sites. The union of these fragments reconstructs the original relation S.

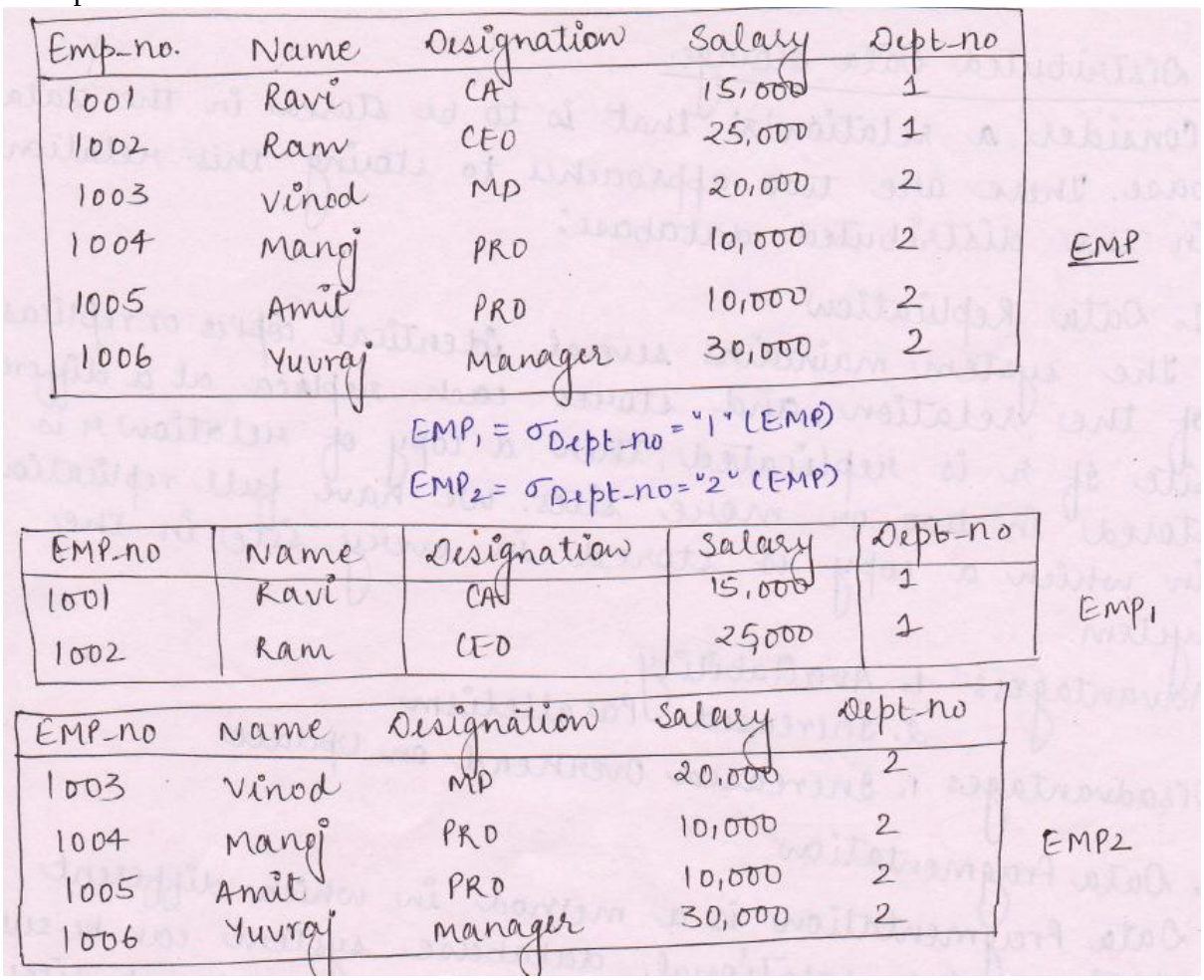
There are three types of fragmentation:

a) Horizontal Fragmentation

Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments. A relation r is partitioned into a number of subsets, r1, r2, . . . , rn. Each tuple of

relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.

Example:



EMP

Emp_no.	Name	Designation	Salary	Dept_no
1001	Ravi	CA	15,000	1
1002	Ram	CEO	25,000	1
1003	Vinod	MD	20,000	2
1004	Manoj	PRO	10,000	2
1005	Amit	PRO	10,000	2
1006	Yuvraj	Manager	30,000	2

$EMP_1 = \sigma_{Dept\_no = "1"}(EMP)$

$EMP_2 = \sigma_{Dept\_no = "2"}(EMP)$

EMP_1

EMP_no	Name	Designation	Salary	Dept_no
1001	Ravi	CA	15,000	1
1002	Ram	CEO	25000	1

EMP_2

EMP_no	Name	Designation	Salary	Dept_no
1003	Vinod	MD	20,000	2
1004	Manoj	PRO	10,000	2
1005	Amit	PRO	10,000	2
1006	Yuvraj	Manager	30,000	2

b) Vertical Fragmentation

In vertical fragmentation, attributes are divided into number of subsets. Vertical fragmentation of $r(R)$ involves the definition of several subsets of attributes $R1, R2, \ldots, Rn$ of the schema R so that

$$R = R1 \cup R2 \cup \cdots \cup Rn$$

Each fragment ri of r is defined by

$$ri = \Pi_{Ri}(r)$$

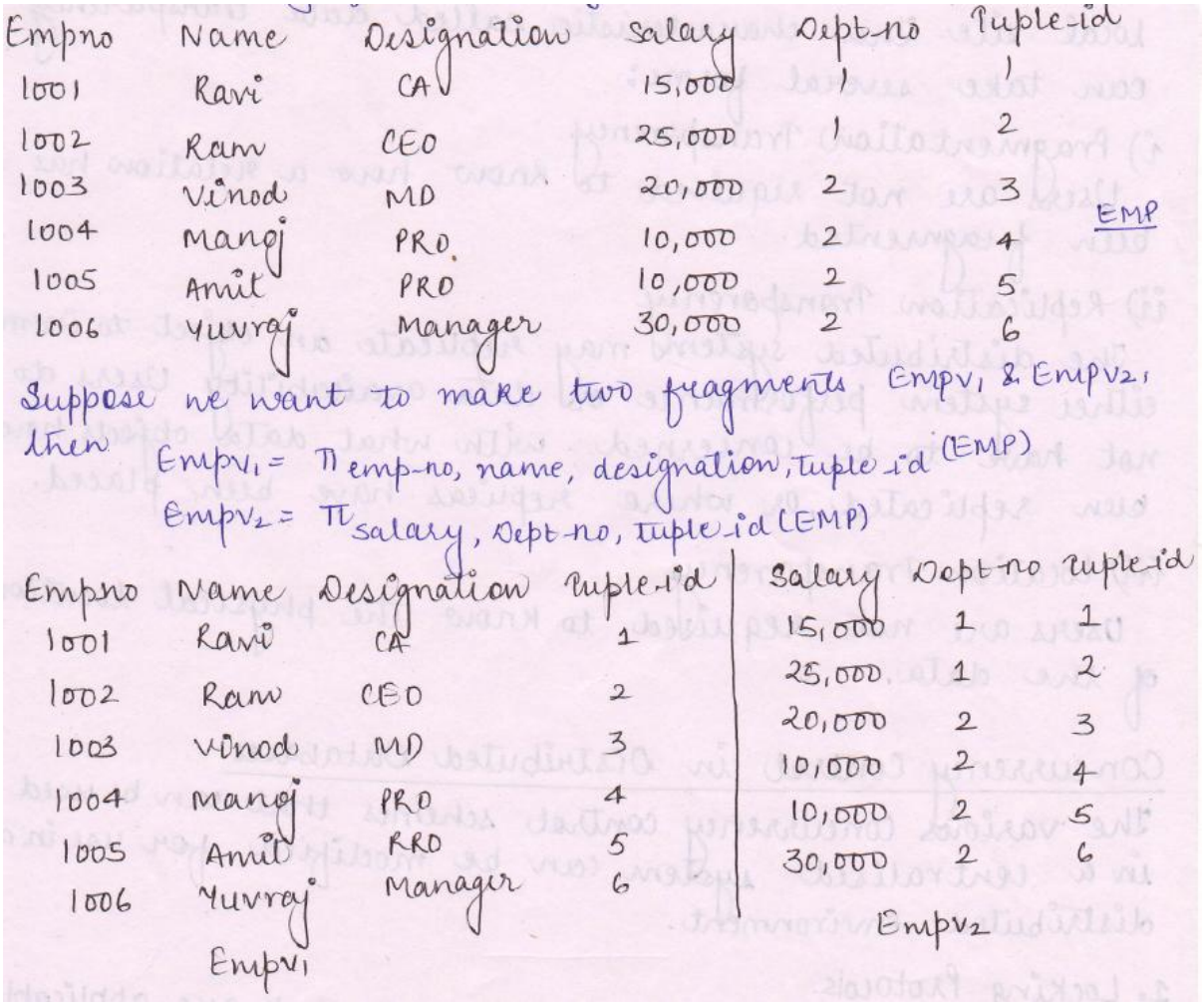
The fragmentation should be done in such a way that we can reconstruct relation r from the fragments by taking the natural join

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$$

One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each of the R_i. More generally, any superkey can be used. It is often convenient to add a special attribute, called a *tuple-id*, to the schema R. The tuple-id value of

a tuple is a unique value that distinguishes the tuple from all other tuples. The tuple-id attribute thus serves as a candidate key for the augmented schema, and is included in each of the R_is.

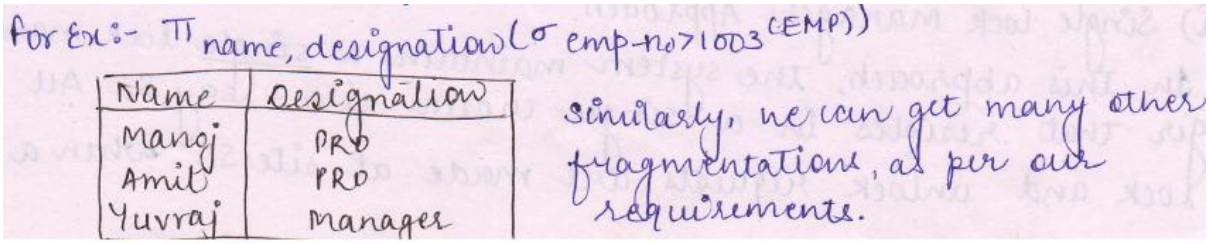
For Example:



c) Mixed Fragmentation

Horizontal (or Vertical Fragmentation) of a relation, followed by further vertical (or horizontal) fragmentation is called mixed fragmentation.

For Example:



3. Transparency

The user of a distributed database system should not be required to know either where the data are physically located or how the data can be accessed at the specific local site. This characteristic, called **data transparency**, can take several forms:

• **Fragmentation transparency**. Users are not required to know how a relation has been fragmented.

• **Replication transparency**. Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.

• **Location transparency**. Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

Concurrency Control in Distributed Databases

The various concurrency control schemes that can be used in a centralized system can be modified for use in a distributed environment.

1. Locking Protocols

Several possible schemes are presented that are applicable to an environment where data can be replicated in several sites. The existence of the *shared* and *exclusive* lock modes shall be assumed.

i) Single Lock Manager Approach

In the **single lock-manager** approach, the system maintains a *single* lock manager that resides in a *single* chosen site—say *Si*. All lock and unlock requests are made at site *Si*. When a transaction needs to lock a data item, it sends a lock request to *Si*. The lock manager determines whether the lock can be granted immediately. If the lock can be granted, the lock manager sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted. The transaction can read the data item from *any* one of the sites at which a replica of the data item resides. In the case of a write, all the sites where a replica of the data item resides must be involved in the writing.

The scheme has these advantages:

• **Simple implementation**. This scheme requires two messages for handling lock requests, and one message for handling unlock requests.

• **Simple deadlock handling**. Since all lock and unlock requests are made at one site, the deadlock-handling algorithms can be applied directly to this environment.

The disadvantages of the scheme are:

• **Bottleneck**. The site S_i becomes a bottleneck, since all requests must be processed there.

• **Vulnerability**. If the site S_i fails, the concurrency controller is lost.

ii) Distributed Lock Manager

The lock-manager function is distributed over several sites. Each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data items that are stored in that site.

This approach reduces the degree to which the coordinator is a bottleneck, but it complicates deadlock handling.

iii) Majority Protocol

The **majority protocol** works this way: If data item Q is replicated in n different sites, then a lock-request message must be sent to more than one-half of the n sites in which Q is stored. Each lock manager determines whether the lock can be granted immediately (as far as it is concerned). As before, the response is delayed until the request can be granted. The transaction does not operate on Q until it has successfully obtained a lock on a majority of the replicas of Q.

iv) Primary Copy

When a system uses data replication, we can choose one of the replicas as the **primary copy**. Thus, for each data item Q, the primary copy of Q must reside in precisely one site, which we call the **primary site** of Q.
When a transaction needs to lock a data item Q, it requests a lock at the primary site of Q. As before, the response to the request is delayed until it can be granted. Thus, the primary copy enables concurrency control for replicated data to be handled like that for unreplicated data.

v) Biased Protocol

The **biased protocol** is another approach to handling replication. The difference from the majority protocol is that requests for shared locks are given more favorable treatment than requests for exclusive locks.

• **Shared locks**.When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one site that contains a replica of Q.

• **Exclusive locks**. When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all sites that contain a replica of Q.

vi) Quorum Consensus Protocol

The **quorum consensus** protocol is a generalization of the majority protocol. The quorum consensus protocol assigns each site a nonnegative weight. It assigns read and write

operations on an item x two integers, called **read quorum** Q_r and **write quorum** Q_w, that must satisfy the following condition, where S is the total weight of all sites at which x resides:

$$Q_r + Q_w > S \text{ and } 2 * Q_w > S$$

To execute a read operation, enough replicas must be read that their total weight is ≥ Qr. To execute a write operation, enough replicas must be written so that their total weight is ≥ Qw.

2. Timestamping

It is a scheme used for generating unique timestamps, that the system uses in deciding the serialization order. There are two primary methods for generating unique timestamps, one centralized and one distributed.

In the centralized scheme, a single site distributes the timestamps. The site can use a logical counter or its own local clock for this purpose.

In the distributed scheme, each site generates a unique local timestamp by using either a logical counter or the local clock. We obtain the unique global timestamp by concatenating the unique local timestamp with the site identifier, which also must be unique.

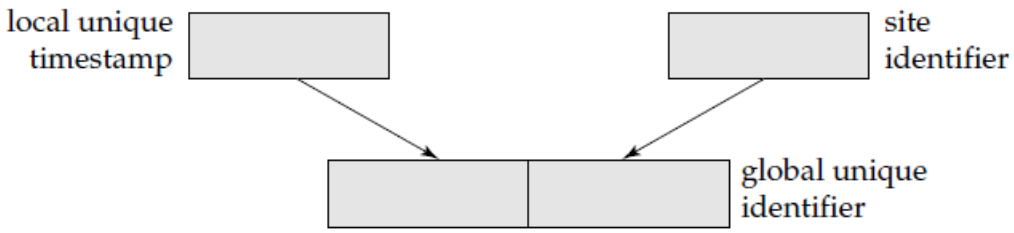


Figure 19.2 Generation of unique timestamps.

Directory Systems

A directory is a listing of information about some class of objects such as persons. Directories can be used to find information about a specific object, or in the reverse direction to find objects that meet a certain requirement.

Directory Access Protocols

Directory information can be made available through Web interfaces. **Directory access protocols** have been developed to provide a standardized way of accessing data in a directory. The most widely used among them today is the **Lightweight Directory Access Protocol** (**LDAP**).

LDAP: Lightweight Directory Access Protocol

The **X.500 directory access protocol**, defined by the International Organization for Standardization (ISO), is a standard for accessing directory information. However, the protocol is rather complex, and is not widely used. The **Lightweight Directory Access**

Protocol (**LDAP**) provides many of the X.500 features, but with less complexity, and is widely used. Directory access protocols also define a data model and access control.

LDAP Data Model

In LDAP directories store **entries**, which are similar to objects. Each entry must have a **distinguished name (DN)**, which uniquely identifies the entry. A DN is in turn made up of a sequence of **relative distinguished names (RDNs)**. The set of RDNs for a DN is defined by the schema of the directory system.

LDAP allows the definition of **object classes** with attribute names and types. Inheritance can be used in defining object classes. Moreover, entries can be specified to be of one or more object classes. It is not necessary that there be a single most-specific object class to which an entry belongs.

Entries are organized into a **directory information tree (DIT)**, according to their distinguished names. Entries at the leaf level of the tree usually represent specific objects.

Data Manipulation

Unlike SQL, LDAP does not define either a data-definition language or a data manipulation language. However, LDAP defines a network protocol for carrying out data definition and manipulation. Users of LDAP can either use an application programming interface, or use tools provided by various vendors to perform data definition and manipulation. LDAP also defines a file format called **LDAP Data Interchange Format (LDIF)** that can be used for storing and exchanging information.

The querying mechanism in LDAP is very simple, consisting of just selections and projections, without any join.

Distributed Directory Trees

Information about an organization may be split into multiple DITs, each of which stores information about some entries. The **suffix** of a DIT is a sequence of RDN=value pairs that identify what information the DIT stores; the pairs are concatenated to the rest of the distinguished name generated by traversing from the entry to the root.

A node in a DIT may contain a **referral** to another node in another DIT.