Chapter – 16

16.6 Failure with loss of non volatile storage (For Understanding):

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the nonvolatile storage remains intact. Although failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure.

The basic scheme is to **dump** the entire contents of the database to stable storage periodically—say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

Approach for database Dumping:

One approach to database dumping requires that no transaction may be active during the dump procedure, and uses a procedure similar to check pointing:

1. Output all log records currently residing in main memory onto stable storage.

2. Output all buffer blocks onto the disk.

3. Copy the contents of the database to stable storage.

4. Output a log record <dump> onto the stable storage.

To recover from the loss of non volatile storage what system does?

To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the actions since the most recent dump occurred. Notice that no undo operations need to be executed.

16.7.3 Transaction Rollback with Logical Undo:

When rolling back a transaction *Ti* , the log is scanned backwards, and log records corresponding to *Ti* are processed as follows:

1. Physical log records encountered during the scan are handled as normal transaction rollback. Incomplete logical operations are undone using the physical log records generated by the operation.

2. Completed logical operations, identified by operation-end records, are rolled back differently. Whenever the system finds a log record<*Ti* , *Oj* , operation end, *U*>, it takes special actions:

> a. It rolls back the operation by using the undo information *U* in the log record. It logs the updates performed during the rollback of the operation just like updates performed when the operation was first executed. At the end of the operation rollback, instead of generating a log record <*Ti* , *Oj* , operation-end, *U*>, the database system generates a log record <*Ti* , *Oj* , operation-abort>.
>
> b. As the backward scan of the log continues, the system skips all log records of transaction *Ti* until it finds the log record<*Ti* , *Oj* , operation begin>. After it finds the operation-begin log record, it processes log records of transaction *Ti* in the normal manner again.

3. If the system finds a record <*Ti* , *Oj* , operation-abort>, it skips all preceding records (including the operation-end record for *Oj*) until it finds the record <*Ti* , *Oj* , operation-begin>.

4. As before, when the <*Ti* start> log record has been found, the transaction rollback is complete, and the system adds a record <*Ti* abort> to the log.

Chapter-14

In the case of conflict serializibility what cases we need to consider?

Let us consider a schedule *S* in which there are two consecutive instructions, *I* and *J* , of transactions *Ti* and *Tj* , respectively (*i* _= *j*). If *I* and *J* refer to different data items, then we can swap *I* and *J* without affecting the results of any instruction in the schedule. However, if *I* and *J* 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. *I* = read(*Q*), *J* = read(*Q*). The order of *I* and *J* does not matter, since the same value of *Q* is read by *Ti* and *Tj* , regardless of the order.

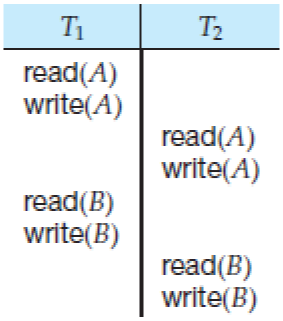
2. *I* = read(*Q*), *J* = write(*Q*). If *I* comes before *J* , then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *J* . If *J* comes before *I*, then *Ti* reads the value of *Q* that is written by *Tj*. Thus, the order of *I* and *J* matters.

3. *I* = write(*Q*), *J* = read(*Q*). The order of *I* and *J* matters for reasons similar to those of the previous case.

4. *I* = write(*Q*), *J* = 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, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(*Q*) instruction after *I* and *J* in *S*, then the order of *I* and *J* directly affects the final value of *Q* in the database state that results from schedule *S*.

Note:

To illustrate the concept of conflicting instructions, we consider schedule 3in Figure 14.6. 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.



T_1	T_2

read(*A*)
write(*A*)	

	read(*A*)
	write(*A*)

read(*B*)
write(*B*)	

	read(*B*)
	write(*B*)

Figure 14.6 Schedule 3 — showing only the read and write instructions.

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/ Show that a schedule 3 is equivalent to a serial schedule:

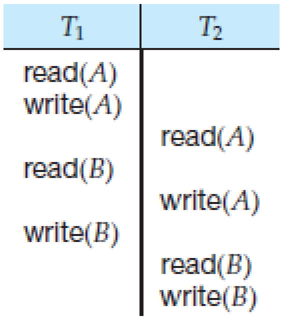
Let *I* and *J* be consecutive instructions of a schedule *S*. If *I* and *J* are instructions of different transactions and *I* and *J* do not conflict, then we can swap the order of *I* and *J* to produce a new schedule *S*`. *S* is equivalent to *S*`, since all instructions appear in the same order in both schedules except for *I* and *J* , whose order does not matter.

Since the write(*A*) instruction of *T*2 in schedule 3 of Figure 14.6 does not conflict with the read(*B*) instruction of *T*1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 14.7. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

We continue to swap non conflicting instructions:

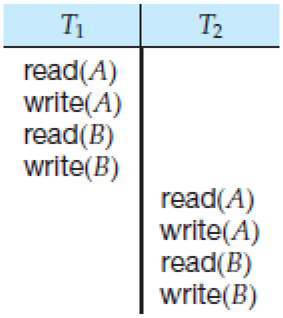
- Swap the read(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.
- Swap the write(*B*) instruction of *T*1 with the write(*A*) instruction of *T*2.
- Swap the write(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.

The final result of these swaps, schedule 6 of Figure 14.8, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the read and write instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule. 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**.



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

Figure 14.7 Schedule 5 — schedule 3 after swapping of a pair of instructions.



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

Figure 14.8 Schedule 6 — a serial schedule that is equivalent to schedule 3.

Define Conflict Serializabe:

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting operations:

Two operations are said to be conflicting if all conditions satisfy:

1. They belong to different transactions
2. They operate on the same data item
3. At Least one of them is a write operation

<u>Precedence Graph:</u>

A graph which is used to check conflict serializibility is referred to as precedence graph.

<u>Conditions of precedence graph/ Conditions for determining conflict serializibility of a schedule/Explain precedence graph with explanations:</u>

Consider a schedule *S*. We construct a directed graph, called a **precedence graph**, from *S*. This graph consists of a pair *G* = (*V*, *E*), where *V* is a set of vertices and *E* is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges *Ti* →*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*).

If an edge *Ti* → *Tj* exists in the precedence graph, then, in any serial schedule *S*` equivalent to *S*, *Ti* must appear before *Tj* . For example, the precedence graph for schedule 1 in Figure 14.10a contains the single edge *T*1 → *T*2, since all the instructions of *T*1 are executed before the first instruction of *T*2 is executed. Similarly, Figure 14.10b shows the precedence graph for schedule 2 with the single edge *T*2 →*T*1, since all the instructions of *T*2 are executed before the first instruction of *T*1 is executed. The precedence graph for schedule 4 appears in Figure 14.11. It contains the edge *T*1 →*T*2, because *T*1 executes read(*A*) before *T*2 executes write(*A*). It also contains the edge *T*2→*T*1, because *T*2 executes read(*B*) before *T*1 executes write(*B*). If the precedence graph for *S* has a cycle, then schedule *S* is not conflict serializable. If the graph contains no cycles, then the schedule *S* is conflict serializable.

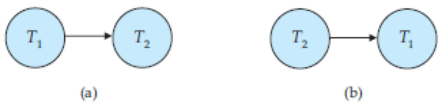


Figure 14.10 Precedence graph for (a) schedule 1 and (b) schedule 2.

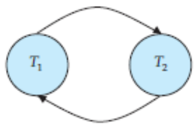
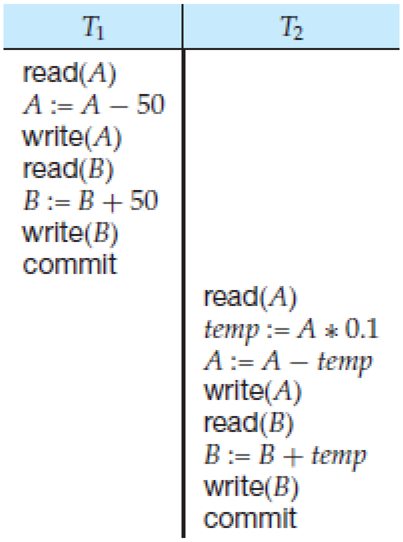
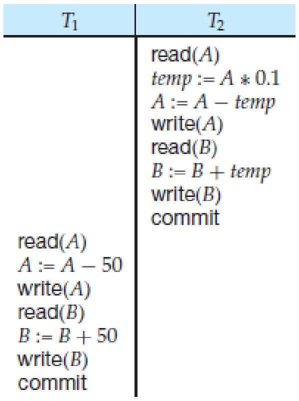


Figure 14.11 Precedence graph for schedule 4.



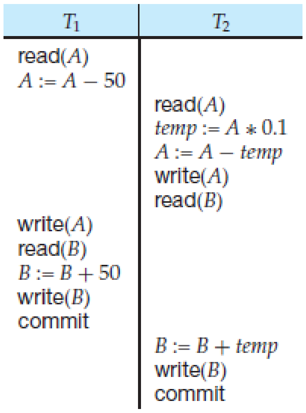
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 14.2 Schedule 1 — a serial schedule in which T_1 is followed by T_2.



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

Figure 14.3 Schedule 2 — a serial schedule in which T_2 is followed by T_1.



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)	
commit	
	$B := B + temp$
	write(B)
	commit

Figure 14.5 Schedule 4 — a concurrent schedule resulting in an inconsistent state.

Define Topological Sorting:

A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

Difference between conflict serializibility and view serializibility:

S.No.	Conflict Serializability	View Serializability
1.	Two schedules are said to be conflict equivalent if all the conflicting operations in both the schedule get executed in the same order. If a schedule is a conflict equivalent to its serial schedule then it is called Conflict Serializable Schedule.	Two schedules are said to be view equivalent if the order of initial read, final write and update operations is the same in both the schedules. If a schedule is view equivalent to its serial schedule then it is called View Serializable Schedule.
2.	If a schedule is view serializable then it may or may not be conflict serializable.	If a schedule is conflict serializable then it is also view serializable schedule.
3.	Conflict equivalence can be easily achieved by reordering the operations of two transactions therefore, Conflict Serializability is easy to achieve.	Viewequivalence is rather difficult to achieve as both transactions should perform similar actions in a similar manner. Thus, View Serializability is difficult to achieve.
4.	For a transaction T1 writing a value A that no one else reads but later some other transactions say T2 write its own value of A, W(A) cannot be placed under positions where it is never read.	If a transaction T1 writes a value A that no other transaction reads (because later some other transactions say T2 writes its own value of A) W(A) can be placed in positions of the schedule where it is never read.