Transaction Management

- **Transaction Concept**
- **A Simple Transaction Model**
- **Serializability**

Transaction Concept

- Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user.

- **For example,**

“a transfer of funds from a checking account to a savings account”

- **is a single operation from the customer’s standpoint; within the database system,**
- **however, it consists of several operations.**

- Clearly, it is essential that all these operations occur, or that, in case of a failure, none occur.

- Collections of operations that form a single logical unit of work are called **transactions**.

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 (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in JDBC or ODBC.

- A transaction 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**.

ACID Property

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions 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.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

A Simple Transaction Model

- The data items in our simplified model contain a single data value (a number in our examples).
- Each data item is identified by a name (typically a single letter in our examples, that is, *A*, *B*, *C*, etc.).
- We shall illustrate the transaction concept using a simple bank application consisting of several accounts and a set of transactions that access and update those accounts.
- Transactions access data using two operations:
- • read(*X*) - which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.

  Eg. **read(Empno) means, the variable Empno = value in the attribute Empno**.
- • write(*X*), which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.

  Eg. write(Empno) means, the attribute Empno = value in the variable Empno

A Simple Transaction Model

- Let *Ti* be a transaction that transfers $50 from account *A* to account *B*. This transaction can be defined as:

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

Consistency: The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction.

Atomicity: Suppose A = 1000$ and B = 2000$.
If there is an error after the write(*A*) operation but before the write(*B*) then A = 950$ and B = 2000$.
the sum *A* + *B* is no longer preserved.
Now the system is in inconsistent state.

Durability: Once the execution of the transaction completes successfully, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds.

Isolation: if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.
Eg. At time when A-50 is done but not B+50, if an other transaction reads A and B.

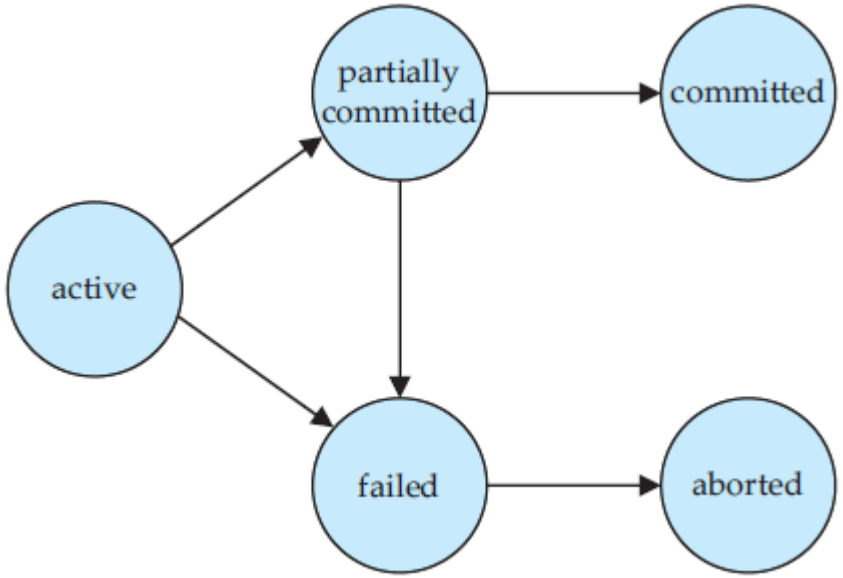
Ensuring atomicity & Durability is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system.**
Ensuring the isolation : by **concurrency-control system.**

A Simple Transaction Model

- a transaction may not always complete its execution successfully, Such a transaction is termed aborted.
- Once the changes caused by an aborted transaction have been undone, we say that the transaction has been rolled back.
- It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a log.
- A transaction that completes its execution successfully is said to be committed.
- Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction.
  - For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account.
- However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system.

A Simple Abstract Transaction Model

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



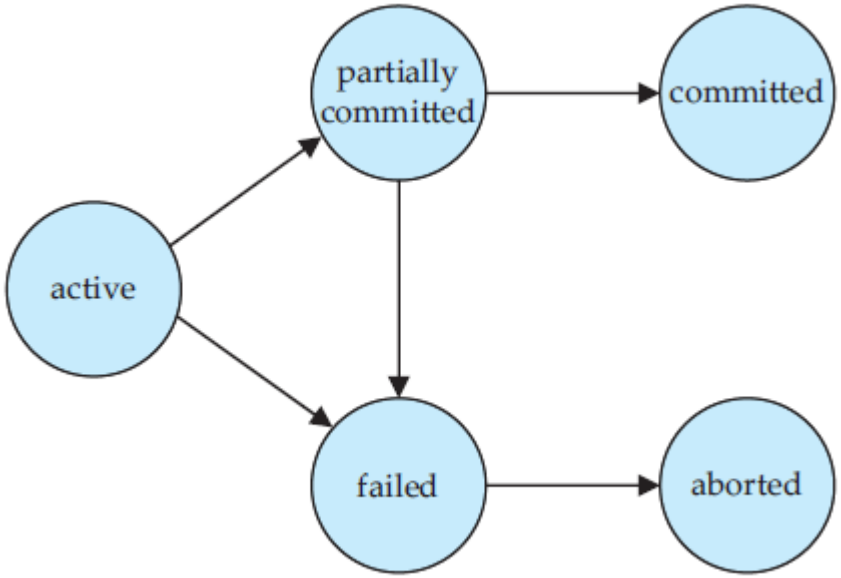
State diagram of a transaction.

A transaction starts in the active state.

When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

A transaction is said to have **terminated** if it has either committed or aborted.

A Simple Transaction Model



A transaction enters the **failed** state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be **rolled back**. Then, it enters the aborted state.

At this point, the system has two options:

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

Transaction Isolation

- Transaction-processing systems usually allow **multiple transactions to run concurrently**.

- Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier.

- 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.

- However, there are two good reasons for allowing concurrency:

(1) Improved throughput and resource utilization.

(2) Reduced waiting time.

- The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control scheme.**

A Simple Transaction Model

Consider the simplified banking system which has several accounts, and a set of transactions that access and update those accounts.

Let $T1$ and $T2$ be two transactions that transfer funds from one account to another.

Transaction $T1$ transfers \$50 from account A to account B.

Transaction $T2$ transfers 10 percent of the balance from account A to account B.

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

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

A Simple Transaction Model

- 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 $T1$ followed by $T2$. (**Schedule #1**)

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

A = 950, B = 2050

temp = 95

A = 855, B = 2145

A+B = 3000

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

The final values of accounts A and B, after the execution of this schedule are \$855 and \$2145, 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.

A Simple Transaction Model

- Similarly, if the transactions are executed one at a time in the order $T2$ followed by $T1$, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.

(Schedule #2)

A and B are \$1000 and \$2000,

T_1	T_2
	read(A)
	$temp := A * 0.1$ temp = 100
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit A = 900, B = 2100
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

A = 850, B = 2150

A+B = 3000

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

- **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.**

  For example, in transaction $T1$, the instruction write(A) must appear before the instruction read(B), in any valid schedule.

- **These schedules are serial.**
- **Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.**
- **for a set of n transactions, there exist n factorial (n!) different valid serial schedules.**

Concurrent Execution of Transactions

- When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial.

- If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.

- With multiple transactions, the CPU time is shared among all the transactions.

- Several execution sequences are possible, since the various instructions from both transactions may now be interleaved.

- In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.

- Suppose that the two transactions (T1 and T2) are executed concurrently, One possible schedule is

***A* and *B* are $1000 and $2000,**

(**Schedule #1**)

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

A = 950, B = 2050

temp = 95

A = 855, B = 2145

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

A+B = 3000

(**Schedule #3**)

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

A = 950

temp = 95

A = 855

B = 2050

B = 2145

A = 855

A+B = 3000

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 T_1 followed by T_2.

The sum $A + B$ is indeed preserved.

- Not all concurrent executions result in a correct state. To illustrate, consider the schedule.

(Schedule #3)

A and B are \$1000 and \$2000

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
A = 950	read(A)
	$temp := A * 0.1$ temp = 95
	$A := A - temp$
	write(A) A = 855
read(B)	
$B := B + 50$	
B = 2050 write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B) B = 2145
	commit A = 855 A+B = 3000

Schedule 3 — a concurrent schedule equivalent to schedule 1.

(Schedule #4)

T_1	T_2
read(A)	
Var.A = 950 $A := A - 50$	
	read(A) A = 1000
	$temp := A * 0.1$ temp = 100
	$A := A - temp$
	write(A) A = 900
	read(B) Var. B = 2000
A = 950 write(A)	
read(B)	
$B := B + 50$	
B = 2050 write(B)	
commit	
	$B := B + temp$
	write(B) B = 2100
	commit A = 950

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

A+B = 3050

$A + B$ is not preserved

This final state is an *inconsistent state*,

- 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, are possible.

- It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state.

- The **concurrency-control component** of the database system carries out this task.

- We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution.
  - That is, the schedule should, in some sense, be equivalent to a serial schedule.
  - Such schedules are called **serializable** schedules.

Serializability

- how to determine when a schedule is serializable?
  - Certainly, serial schedules are serializable,
  - but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable.
- Since transactions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.

- To simplify, we consider only the read and write operations in the schedule.

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

Schedule 3 — a concurrent schedule equivalent to schedule 1.

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

Schedule 3 — showing only the read and write instructions.

Serializability

- There are different forms of schedule equivalence, but focus on a particular form called **conflict serializability**.

- 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:

Serializability

- 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 T_i and T_j, regardless of the order.
2. I = read(Q), J = write(Q). If I comes before J, then T_i does not read the value of Q that is written by T_j in instruction J. If J comes before I, then T_i reads the value of Q that is written by T_j. 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 T_i or T_j. 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.

Thus, only in the case where both I and J are read instructions does the relative order of their execution not matter.
We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

Serializability

schedule 3

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

Schedule 5

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

Illustration

- The write(A) instruction of $T1$ conflicts with the read(A) instruction of $T2$.
- However, the write(A) instruction of $T2$ does not conflict with the read(B) instruction of $T1$, because the two instructions access different data items.
- 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 $S1$.
- $S1$ 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 does not conflict with the read(B) instruction of T_1, we can swap these instructions to generate an equivalent schedule, schedule 5.

Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

Serializability

schedule 3

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

Schedule 5

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

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Schedule 6

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

- We continue to swap nonconflicting instructions:
  - Swap the read(B) instruction of $T1$ with the read(A) instruction of $T2$.
  - Swap the write(B) instruction of $T1$ with the write(A) instruction of $T2$.
  - Swap the write(B) instruction of $T1$ with the read(A) instruction of $T2$.

Schedule 1

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

Schedule 6 is equivalent to Schedule 1.

So, Schedule 3 is serializable schedule.

Serializability

- 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**.
- Not all serial schedules are conflict equivalent to each other.
- For example, schedules 1 and 2 are not 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.

Schedule 4 is not conflict serializable

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

T1	T2	T1	T2	T1	T2	T1	T2
read(A)			read(A)		read(A)		read(A)
	read(A)	read(A)		read(A)		read(A)	
	write(A)		write(A)		write(A)		write(A)
	read(B)		read(B)	write(A)		write(A)	
write(A)		write(A)			read(B)	read(B)	
read(B)		read(B)		read(B)			read(B)
write(B)		write(B)		write(B)		write(B)	
	Write(B)		Write(B)		Write(B)		Write(B)

Serializability

- Finally, consider schedule 7, it consists of only the significant operations (that is, the read and write) of transactions $T3$ and $T4$.

- This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <$T3$,$T4$> or the serial schedule <$T4$,$T3$>.

Schedule 7.

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

Conflict Serializability – by precedence graph

- a simple and efficient method for determining conflict serializability of a schedule.
- Consider a schedule *S*.
- We construct a directed graph, called a **precedence graph**, from *S*. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges.
- The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $Ti \rightarrow Tj$ for which one of three conditions holds:

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

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).

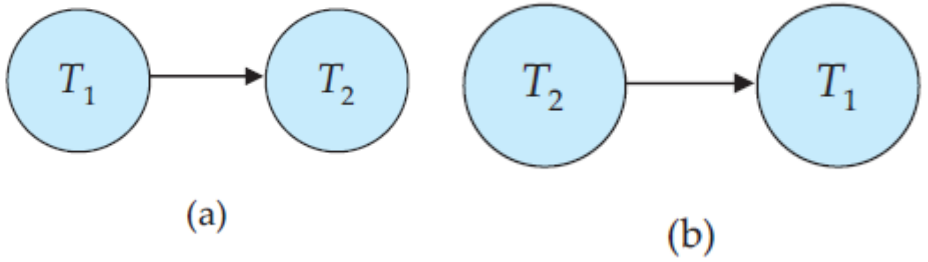
If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S equivalent to S, T_i must appear before T_j.

Conflict Serializability – by precedence graph

Example

Schedule 1

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



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

Schedule 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	

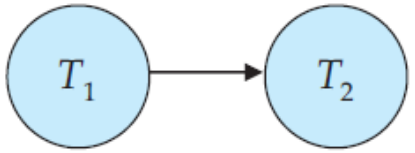
For example, the precedence graph for schedule 1 is in (a) contains the single edge $T1 \rightarrow T2$, since all the instructions of $T1$ are executed before the first instruction of $T2$ is executed.

Similarly, (b) shows the precedence graph for schedule 2 with the single edge $T2 \rightarrow T1$, since all the instructions of $T2$ are executed before the first instruction of $T1$ is executed.

Schedule 3

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

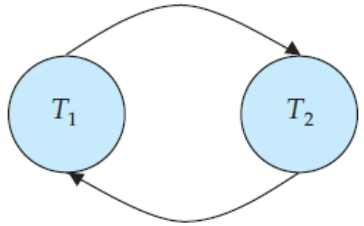
Precedence graph of Schedule 3



(a)

Schedule 4

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



Precedence graph for schedule 4.

The precedence graph for schedule 4 contains the edge $T_1 \rightarrow T_2$, because T_1 executes read(A) before T_2 executes write(A).

It also contains the edge $T_2 \rightarrow T_1$, 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**.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle

- It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.
- For example, consider transaction $T5$, which transfers \$10 from account B to account A.
- Let us define schedule 8, this not conflict equivalent to the serial schedule <$T1$,$T5$>,
- since, in schedule 8, the **write**(B) instruction of $T5$ conflicts with the **read**(B) instruction of $T1$.
- This creates an edge $T5 \rightarrow T1$ in the precedence graph. Similarly, we see that the **write**(A) instruction of $T1$ conflicts with the **read** instruction of $T5$ creating an edge $T1 \rightarrow T5$.
- This shows that the precedence graph has a cycle and that schedule 8 is not serializable.
- However, the final values of accounts A and B after the execution of either schedule 8 or the serial schedule <$T1$,$T5$> are the same—\$960 and \$2040, respectively. (Exercise)

Schedule 8.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Effect of transaction failures during concurrent execution.

- 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, the atomicity property requires that any transaction *Tj* that is dependent on *Ti* (that is, *Tj* has read data written by *Ti*) is also aborted.

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

- what schedules are acceptable from the viewpoint of recovery from transaction failure.
  - **Recoverable Schedules**
  - **Cascadeless Schedules**

- **Recoverable Schedules**
- Consider the partial schedule 9, in which $T7$ is a transaction that performs only one instruction: read(A).
- We call this a **partial schedule** because we have not included a **commit** or **abort** operation for $T6$.
- Notice that $T7$ commits immediately after executing the read(A) instruction.
- Thus, $T7$ commits while $T6$ is still in the active state.
- Now suppose that $T6$ fails before it commits. $T7$ has read the value of data item A written by $T6$. Therefore, we say that $T7$ is **dependent** on $T6$.
- Because of this, we must abort $T7$ to ensure atomicity.
- However, $T7$ has already committed and cannot be aborted.
- Thus, we have a situation where it is impossible to recover correctly from the failure of $T6$.

Schedule 9,

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

a nonrecoverable schedule.

A recoverable schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

For the example of schedule 9 to be recoverable, T_7 would have to delay committing until after T_6 commits.

- **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 many transactions have read data written by *Ti* .

- Illustration

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Schedule 10.

- Transaction *T*8 writes a value of *A* that is read by transaction *T*9. Transaction *T*9 writes a value of *A* that is read by transaction *T*10.
- Suppose that, at this point, *T*8 fails. *T*8 must be rolled back. Since *T*9 is dependent on *T*8, *T*9 must be rolled back. Since *T*10 is dependent on *T*9, *T*10 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.**

The End.//