3.1. Concept of transaction, ACID properties, States of transaction
3.2. Concurrency control, Problems in concurrency controls
3.3. Scheduling of transactions, Serializability and testing of serilaizibity

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.

Definition :

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

A database system must ensure proper execution of transactions despite failures either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency.

In our funds-transfer example, a transaction computing the customer's total money might see the checking-account balance before it is debited by the funds transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

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**.

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

A C I D

•**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 *Ti* and *Tj* , it appears to *Ti* that either *Tj* finished execution before *Ti* started, or *Tj* started execution after *Ti* 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. These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

Transactions access data using two operations:

• **read(*X*),** which transfers the data item *X* from the database to a local buffer belonging to the transaction that executed the read operation.

• **write(*X*)**, which transfers the data item *X* from the local buffer of the transaction that executed the write back to the database.

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

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

Let us now consider each of the ACID requirements.

• **Consistency**: The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the

application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints,

• **Atomicity**: Suppose that, just before the execution of transaction *Ti* the values of accounts *A* and *B* are $1000 and $2000, respectively. Now suppose that, during the execution of transaction *Ti*, a failure occurs that prevents *Ti* from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(*A*) operation but before the write(*B*) operation. In this case, the values of accounts *A* and *B* reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction *Ti* is executed to completion, there exists a point at which the value of account *A* is $950 and the value of account *B* is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account *A* is $950, and the value of account *B* is $2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this : The database system keeps track (on disk) of the 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. Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the **transaction-management component**,

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We assume for now that a failure of the computer system may result in

loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the **recovery-management component**.

• **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state. For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to *A* and the increased total yet to be written to *B*. If a second concurrently running transaction reads *A* and *B* at this intermediate point and computes *A*+*B*, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on *A* and *B* based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order.

Transaction State :

In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. 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.

A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure. 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 transaction must be in one of the following states :

• **Active**, the initial state; the transaction stays in this state while it is executing

• **Partially committed**, after the final statement has been executed

• **Failed**, 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

The state diagram corresponding to a transaction appears in Figure 15.1. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if has either committed or aborted. 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.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state. We assume for now that failures do not result in loss of data on disk. 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:

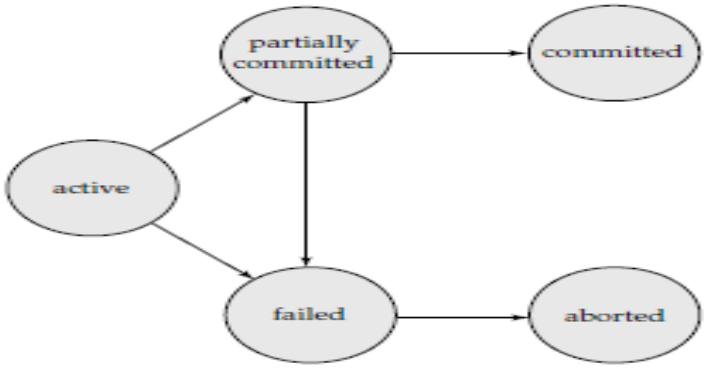


Figure 15.1 State diagram of a transaction.

1] 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.

2] 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.

We must be cautious when dealing with **observable external writes**, such as writes to a terminal or printer. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system. Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in nonvolatile storage, and to perform the actual writes only after the transaction enters the committed state. If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the users account, needs to be executed when the system is restarted.

For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data

unless we are willing to compromise transaction atomicity. Most current transaction systems ensure atomicity and, therefore, forbid this form of interaction with users.

Implementation of Atomicity and Durability :

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the **shadow copy** scheme. This scheme, which is based on making copies of the database, called ***shadow* copies**, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

If the transaction completes, it is committed as follows. First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. Figure 15.2 depicts the scheme, showing the database state before and after the update.

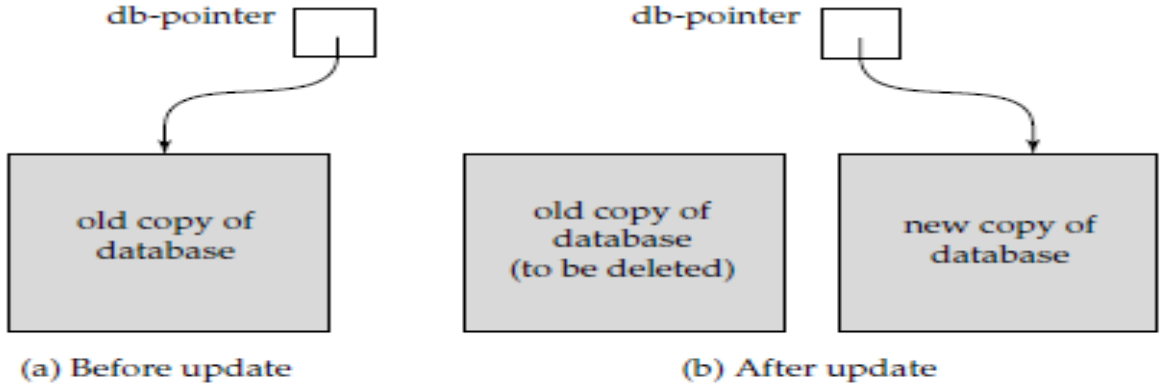


Figure 15.2 Shadow-copy technique for atomicity and durability.

The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.

We now consider how the technique handles transaction and system failures. First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected. We can abort the transaction by just deleting the new copy of the database.

Once the transaction has been committed, all the updates that it performed are in the database pointed to by db-pointer. Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

Now consider the issue of system failure. Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.

Next, suppose that the system fails after db-pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk. Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database *after* all the updates performed by the transaction.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written. If some of the bytes of the pointer were updated by the write, but others were not, the pointer is meaningless, and neither old nor new versions of the database may be found when the system restarts. Luckily, disk systems provide atomic updates to entire blocks, or at least to a disk sector. In other words, the disk system guarantees that it will update db-pointer atomically, as long as we make sure that db-pointer lies entirely in a single sector, which we can ensure by storing db-pointer at the beginning of a block.

Thus, the atomicity and durability properties of transactions are ensured by the shadow-copy implementation of the recovery-management component.

Concurrent Executions :

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. 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:

• **Improved throughput and resource utilization**.

A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• **Reduced waiting time**.

There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system. When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure consistency.

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 schemes**. 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. It is defined as

$T1$: **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 **A balance : 1000$**
B balance : 2000$

These two transactions executes one after by one (T1 followed by T2)

T1	T2

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

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

After transactions complete

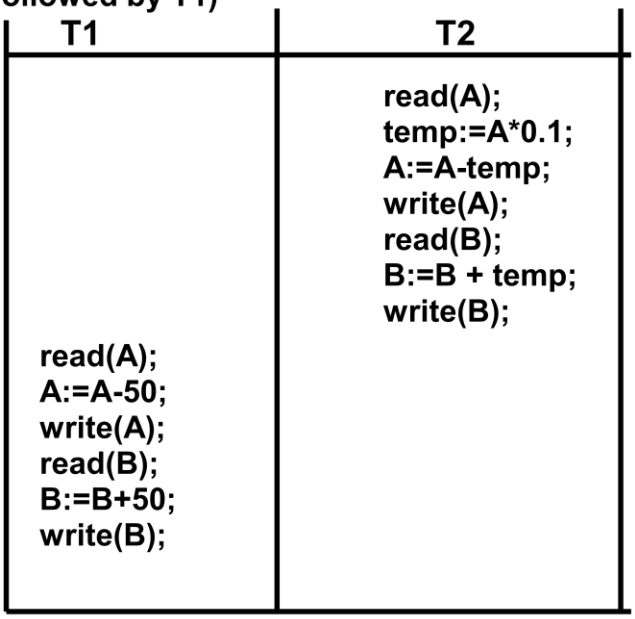
A balance : 855$

B balance : 2145$

Schedule 1

Serial Schedule

Suppose A balance : 1000$
B balance : 2000$
These two transactions executes one after by one
(T2 followed by T1)



After transactions complete

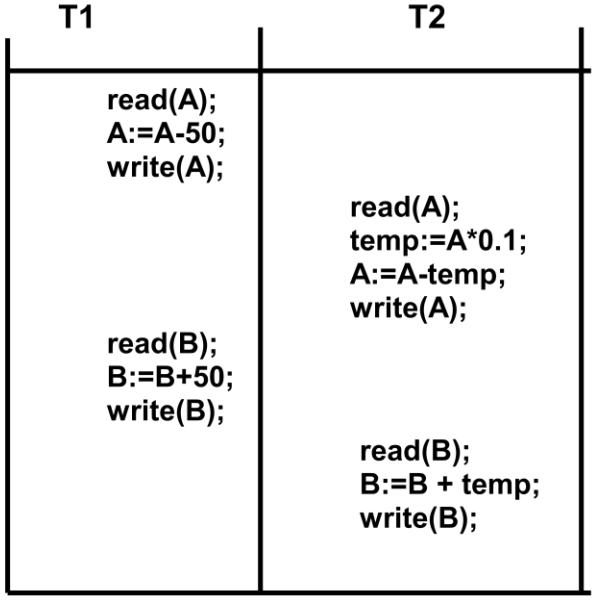
A balance : 850$

B balance : 2150$

Schedule 2

Serial Schedule

Suppose A balance : 1000$
B balance : 2000$
These two transactions executes concurrently



After transactions complete

A balance : 855 $

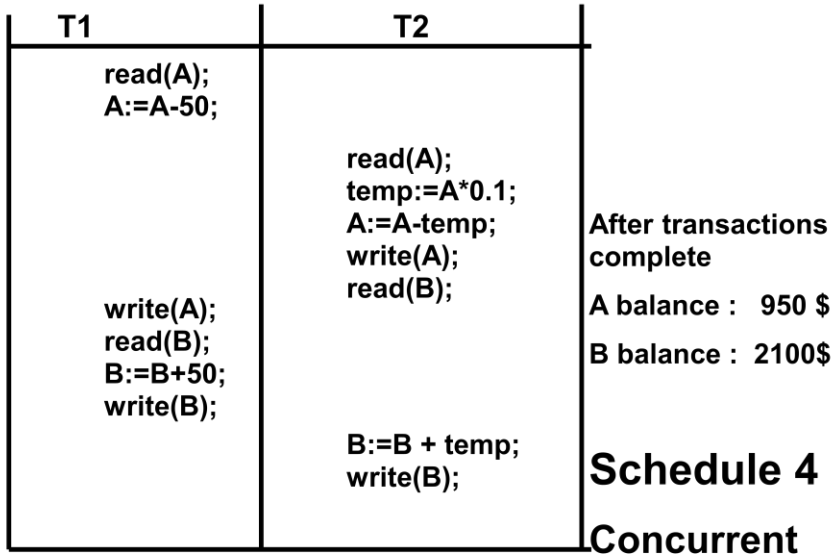
B balance : 2145$

Schedule 3

Concurrent Schedule

Sum after execution is indeed preserved

Suppose A balance : 1000$
B balance : 2000$
These two transactions executes concurrently



Sum after execution is not preserved

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, such as the one just described, are possible. 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.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution.

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.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.

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)

Figure 15.6 Schedule 4—a concurrent schedule.

For this reason, we shall not interpret the type of operations that a transaction can perform on a data item. Instead, we consider only two operations: **read and write**. We thus assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. We shall therefore usually show only read and write instructions in schedules, as we do in schedule 3 in Figure 15.7.

In this section, we discuss different forms of schedule equivalence; they lead to the notions of **conflict serializability** and **view serializability**.

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

Figure 15.7 Schedule 3—showing only the read and write instructions.

Conflict :

We say that I_i and I_j in conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write oprations.

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, 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 *Ii* and *Ij* in *S*, then the order of *Ii* and *Ij* directly affects the final value of *Q* in the database state that results from schedule *S*. Thus, only in the case where both *Ii* and *Ij* are read instructions does the relative order of their execution not matter. We say that *Ii* and *Ij* **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3, in Figure 15.7. 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 of Figure 15.7 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 15.8. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

We continue to swap non conflicting instructions :

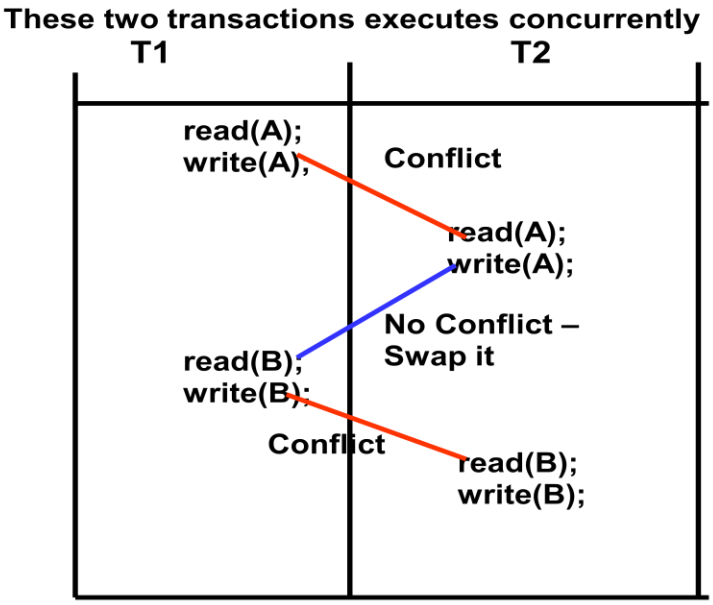
• Swap the read(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.
• Swap the write(*B*) instruction of *T*1 with the write(*A*) instruction of *T*2.
• Swap the write(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.

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

Figure 15.8 Schedule 5—schedule 3 after swapping of a pair of instructions.

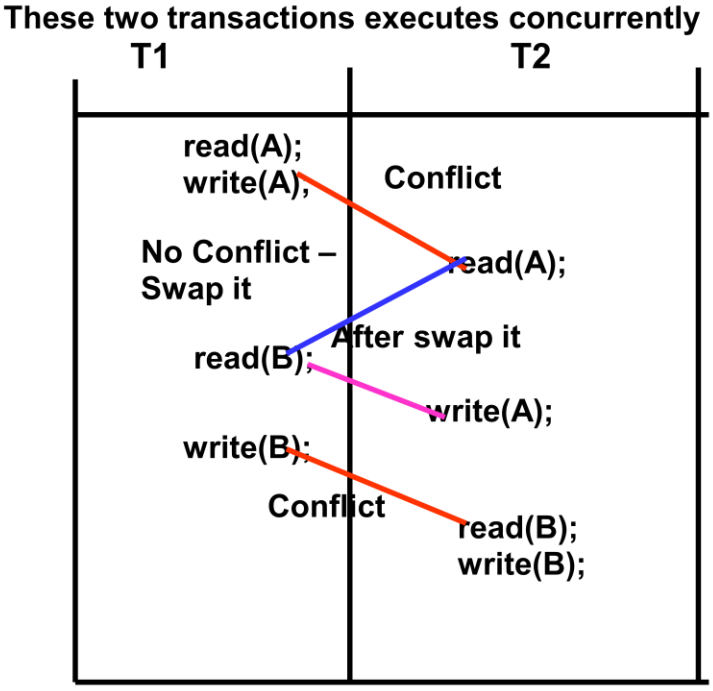
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 15.9 Schedule 6—a serial schedule that is equivalent to schedule 3.



Schedule 3

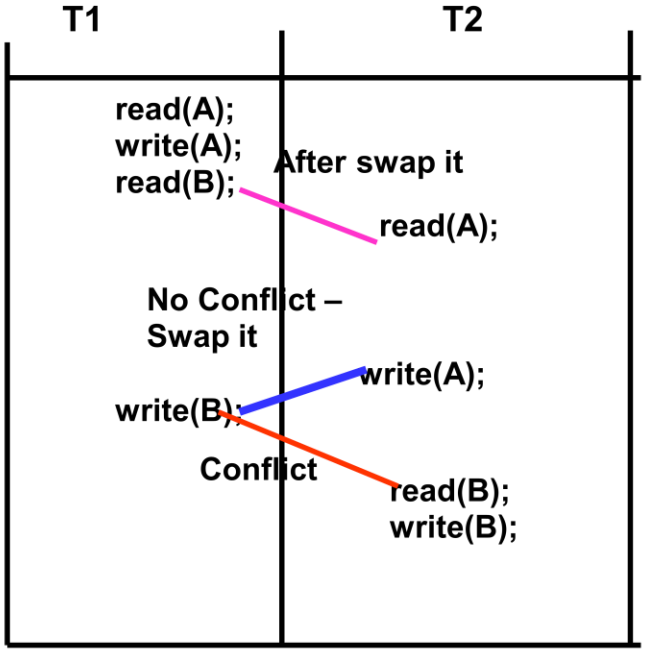
Concurrent Schedule (only read and write operations consider)



Schedule 3

Concurrent Schedule (only read and write operations consider)

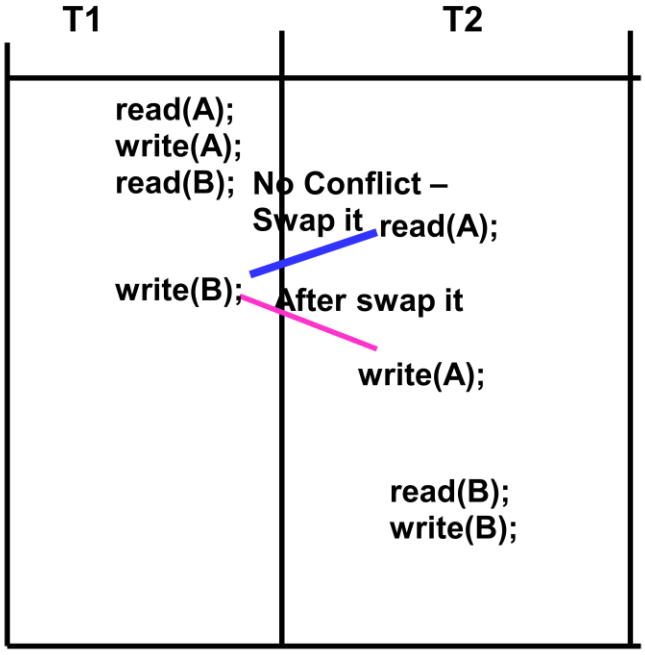
These two transactions executes concurrently



Schedule 3

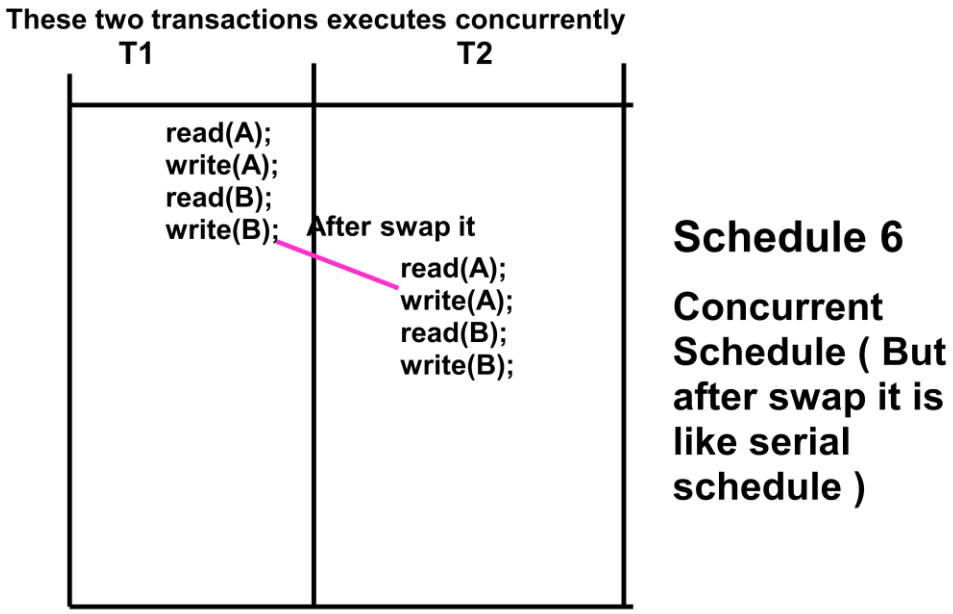
Concurrent Schedule (only read and write operations consider)

These two transactions executes concurrently



Schedule 3

Concurrent Schedule (only read and write operations consider)



Schedule 6 – serial schedule that is equivalent to schedule 3 – Concurrent schedule

The final result of these swaps, schedule 6 of Figure 15.9, is a serial schedule. 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**.

In our previous examples, schedule 1 is not conflict equivalent to schedule 2. However, schedule 1 is conflict equivalent to schedule 3, because the read(*B*) and write(*B*) instruction of *T*1 can be swapped with the read(*A*) and write(*A*) instruction of *T*2. 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. Finally, consider schedule 7 of Figure 15.10; it consists of only the significant operations (that is, the read and write) of transactions *T*3 and *T*4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <*T*3,*T*4> or the serial schedule <*T*4,*T*3>

Conflict equivalent : If a schedule S can be transformed into a schedule S' by a series of swaps of non – conflicting instructions, then we say that S and S' are **Conflict equivalent** For ex. Schdeule 3 – concurrent schedule is equivalent to

schedule 6 which is serial schedule then this type of conflict is called as **conflict serializabilty**

View Serializability :

In this section, we consider a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

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)

Figure 15.11 Schedule 8.

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. For each data item *Q*, if transaction *Ti* reads the initial value of *Q* in schedule *S*, then transaction *Ti* must, in schedule *S_*, also read the initial value of *Q*.

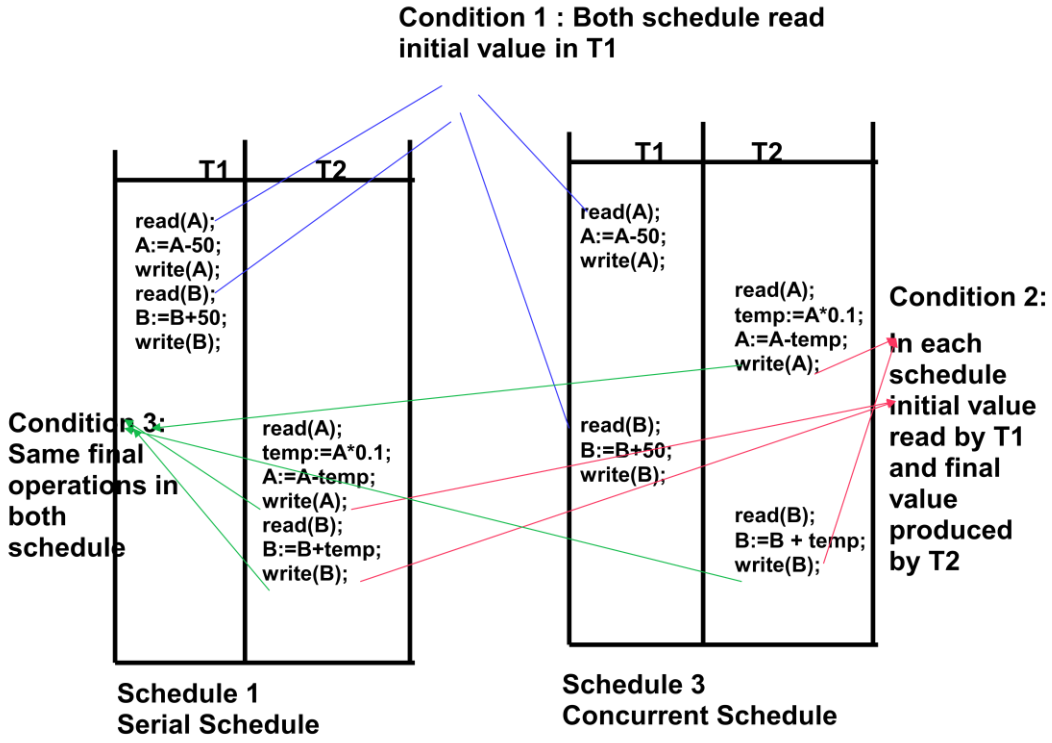
2. For each data item *Q*, if transaction *Ti* executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction *Tj* , then the read(*Q*) operation of transaction *Ti* must, in schedule *S_*, also read the value of *Q* that was produced by the same write(*Q*) operation of transaction *Tj* .

3. 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'*.

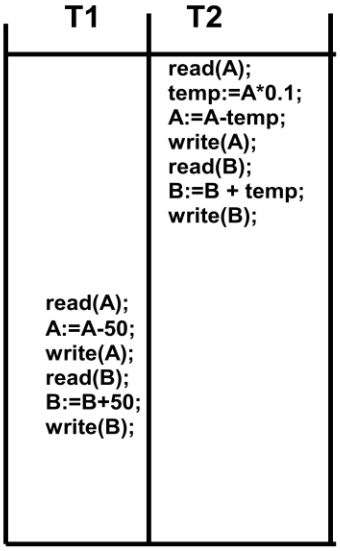
Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled

with conditions 1 and 2, ensures that both schedules result in the same final system state.

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account *A* read by transaction *T*2 was produced by *T*1, where as this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account *A* and *B* read by transaction *T*2 were produced by *T*1 in both schedules. The concept of view equivalence leads to the concept of view serializability. We say that a schedule *S* is **view serializable**

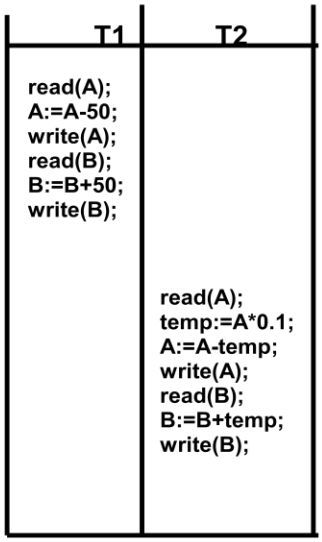


Schedule 1 is **view equivalent** to schedule 3 because the values of account A and B read by transactions T2 were produced by T1 in both schedule 1 & 3. Schedule 1 is serial schedule and schedule 3 is concurrent schedule then it is called as **view serializable equivalent.**



Schedule 2

Serial Schedule



Schedule 1
Serial Schedule

These two schedules are not.....view equivalent .