Chapter 6: Transaction

Collections of operations that form a single logical unit of work are called ***Transactions***. It is a unit of program execution that accesses and probably updates various data items.

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. 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**.

ACID Property:

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

1) ***Atomicity:*** Either all operations of the transaction are reflected properly in the database, or none are.
2) ***Consistency:*** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
3) ***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.
4) ***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.

Explanation of ACID Property:

To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. For the time being, we assume that the database permanently resides on disk, but that some portion of it is temporarily residing in main memory.
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.

A. ***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.

   Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the **transaction-management component.**

B. ***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. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

C. ***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.

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**.

D. ***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.

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. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control component**.

TRANSACTION STATE

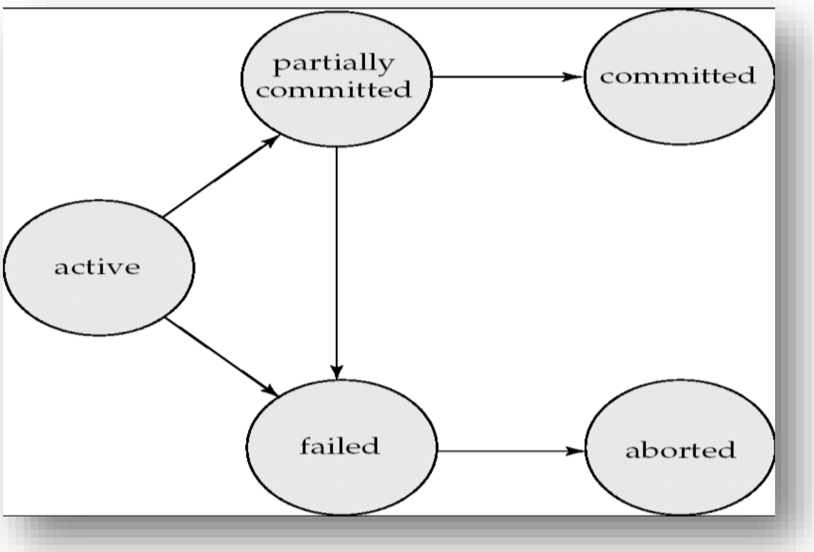
In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state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

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**.

We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

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

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.



State diagram of a transaction.

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.

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.

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, 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 and average response 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 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.***
- ***Transaction T2 transfers 10 percent of the balance from account A to account B.***

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

$T1$	$T2$
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)

Schedule 1
(*A serial schedule in which T1 is followed by T2*)
The final values of accounts *A* and *B* are $855 and $2145 respectively

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 T2 is followed by T1*)
The final values of accounts *A* and *B* are $850 and $2150 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.

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. Thus, for a set of *n* transactions, there exist *n*! different valid serial schedules. 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. Thus, the number of possible schedules for a set of *n* transactions is much larger than *n*!.

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. That is, the schedule should, in some sense, be equivalent to a serial schedule

Serializability:

The following figure shows scheduling components:-

Schedule: 3A and 3B is a concurrent schedule, equivalent to schedule 1.

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 3A

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

Schedule 3B

There are two different forms of schedule equivalence; they lead to the notion of ***Conflict Serializability*** and ***View Serializability***.

Conflict Serializability

Let us consider a schedule *S* in which there are two consecutive instructions *Ii* and *Ij*, of transactions *Ti* and *Tj*, respectively ($i \neq j$). 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 considered:

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*.

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.

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 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.*

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.

We continue to swap no conflicting instructions using the 1st Schedule given below:

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

STEP 1:
Swap the Read(B) instruction of T1 with the Write(A) instruction of T2.

T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	
	Write(A)
Write(B)	
	Read(B)
	Write(B)

STEP 2:
Swap the Read(B) instruction of T1 with the Read(A) instruction of T2.

T1	T2
Read(A)	
Write(A)	
Read(B)	
	Read(A)
	Write(A)
Write(B)	
	Read(B)
	Write(B)

STEP 3:
Swap the Write(B) instruction of T1 with the Write(A) instruction of T2.

T1	T2
Read(A)	
Write(A)	
Read(B)	
	Read(A)
Write(B)	
	Write(A)
	Read(B)
	Write(B)

STEP 4:
Swap the Write(B) instruction of T1 with the Read(A) instruction of T2.

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

The final result of these swaps, schedule S' is a serial schedule. Thus, we have shown that schedule S 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**. 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.

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

Figure: Schedule 4: *This schedule is not conflict serializability, since it is not conflict equivalent to the either serial schedules < T_3, T_4 > or <T_4, T_3 >. It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.*

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: Schedule 5: *This schedule is not conflict equivalent to the serial schedule <T_1, T_5> because write(B) of T_5 conflicts with read(B) of T_1. However the final values of account A and B are the same in both cases.*

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. 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.

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

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

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

1) *Schedule 1 is not view equivalent to schedule 2.*
2) *Schedule 1 is view equivalent to Schedule 3.*

A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

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

Blind Write

N.B.: ***Every Conflict serializable is also view serializable, but there are view-serializable schedules that are not conflict serializable. The above schedule is not conflict serializable but has a blind write so it belongs to view serializable.***

Recoverability

In this section address the 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, 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.

Recoverable Schedules

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

Consider the above schedule, in which *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.

The 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 requires that all schedules be *recoverable*. 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* .

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*. As an illustration, consider the partial schedule describe below

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**.

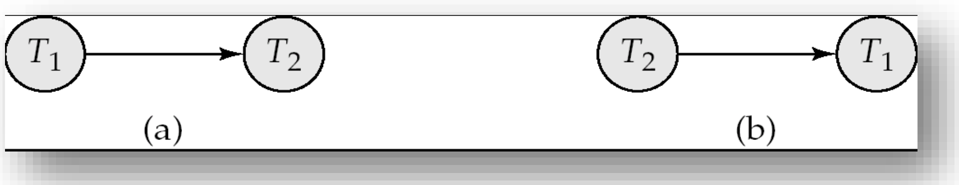
Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless schedule** is one where, for each pair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

Testing for Serializability

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*).

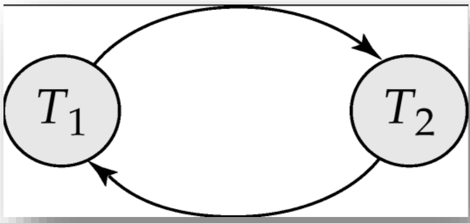
If an edge *Ti* → *Tj* exists in the precedence graph, then, in any serial schedule *S′* equivalent to *S*, *Ti* must appear before *Tj*.



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

The precedence graph for schedule given below is as follow:

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)



It contains the edge $T1 \rightarrow T2$, because $T1$ executes read(A) before $T2$ executes write(A). It also contains the edge $T2 \rightarrow T1$, because $T2$ executes read(B) before $T1$ 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.