Unit-4

TRANSACTION

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.

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.**

To restate the above more concisely, we require that the database system maintain the following properties of the transactions:

- **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.

A Simple Transaction Model

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
- 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

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);

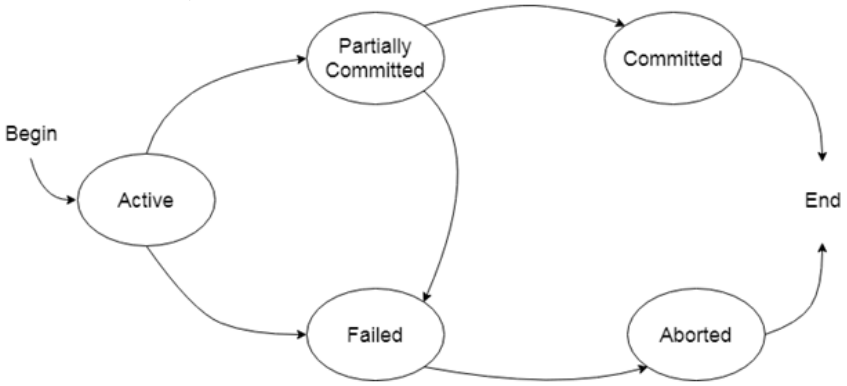
read(B);

B := B + 50;

write(B).

States of Transaction

- In a database, the transaction can 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.

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.

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.**

Consider again 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. It is defined as:

```
T1:   read(A);
      A := A − 50;
      write(A);
      read(B);
      B := B + 50;
      write(B).
```

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as:

```
T2:   read(A);
      temp := A * 0.1;
      A := A − temp;
      write(A);
```

```
read(B);
B := B + temp;
write(B)
```

Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1 followed by T2. This execution sequence appears in Figure 14.2. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Figure 14.2 takes place, 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.

T1	**T2**
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 1—a serial schedule in which T1 is followed by T2.

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure 14.3. Again, as expected, the sum A + B is preserved, and the final values of accounts A and B are $850 and $2150, respectively.

T1	T2
	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 2—a serial schedule in which T2 is followed by T1.

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. Note that we include in our schedules the commit operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence (T1 followed by T2) as schedule 1, and to the second execution sequence (T2 followed by T1) as schedule 2.

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.

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 is executed will leave the database in a consistent state. **The concurrency-control** component of the database system carries out this task.

T1	T2
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.

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.

T1	T2
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.

Schedule 4—a concurrent schedule resulting in an inconsistent state

Serializability

Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider how to determine when a schedule is serializable.

serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable.

In this section, we discuss 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:

1. I = read(Q), J = read(Q). The order of I and J does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.

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

T1	**T2**
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

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

	read(B)
	write(B)

Schedule 5—schedule 3 after swapping of a pair of instructions

3. I = write(Q), J = read(Q). The order of I and J matters for reasons similar to those of the previous case.
4. I = write(Q), J = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

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.

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

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

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

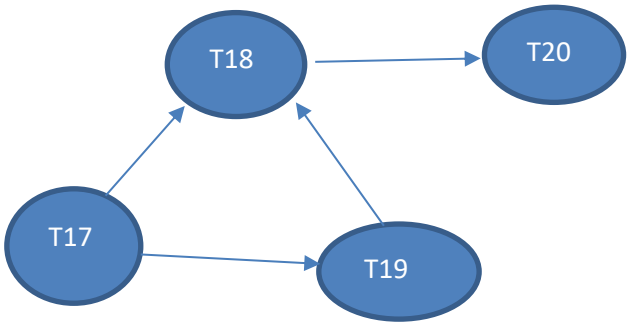
Deadlock

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions {T0, T1,..., Tn} such that T0 is waiting for a data item that T1 holds, and T1 is waiting for a data item that T2 holds, and

... , and Tn−1 is waiting for a data item that Tn holds, and Tn is waiting for a data item that T0 holds.

Deadlock Detection

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



Wait-for graph with no cycle.

When transaction Ti requests a data item currently being held by transaction Tj , then the edge Ti → Tj is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti .

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the wait-for graph

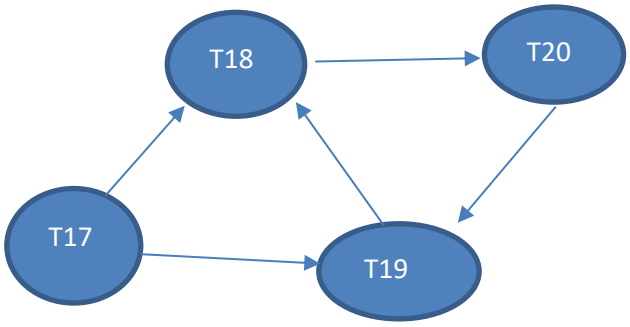
- Transaction T17 is waiting for transactions T18 and T19.
- Transaction T19 is waiting for transaction T18.
- Transaction T18 is waiting for transaction T20.

Since the graph has no cycle, the system is not in a deadlock state

Suppose now that transaction T20 is requesting an item held by T19. The edge T20 → T19 is added to the wait-for graph, resulting in the new system state in Figure 15.14. This time, the graph contains the cycle:

T18 → T20 → T19 → T18

implying that transactions T18, T19, and T20 are all deadlocked.



Wait-for graph with a cycle.