Unit 2 / Transaction Concepts

1. Introduction

The concept of transaction provides a mechanism for describing logical units of database processing. **Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. *Examples* of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users.

In this chapter we present the concepts that are needed in transaction processing systems. We define the concept of a transaction, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness.

A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

Many a times, 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*, booking a ticket online is a single operation from the customer's point of view; but within the database system, the same booking operation is broken into several operations. It is mandatory that all these operations occur or in case of a failure, none of the operations occur, since, it is not acceptable if the ticket is booked, but the master file consisting of total available seats are not being updated.

A transaction can, thus, be defined as a collection of operations that forms a single logical unit of work.

A database system must always ensure proper execution of transactions, irrespective of any kinds of failures that may occur. Further, it must also manage concurrent execution of transactions in such a way that it eliminates inconsistency of data within the database.

In this chapter, we focus on the basic concepts and theory that are needed to ensure the correct executions of transactions. We discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results.

2. Transaction Concepts

1

Apr. 2018 – *1M*
Define:
i. Transaction
ii. Atomicity

A transaction is a unit of program execution that accesses and possibly updates various data items.

A transaction results from the execution of a user program written in a high–level data manipulation language or programming language and is delimited by statements of the form begin transaction and end transaction.

The transaction consists of all operations executed between the begin and end of a transaction.

For example, Consider a saving account management system that is a subsystem of a Banking Enterprise. This system may have several accounts and a set of transactions that accesses and updates those accounts.

So we can consider an *example* of a transaction that transfers Rs. 500 from account A to account B. This transaction is defined as:

```
Ti:
 Begin
    Read(A);
       A:=A-500;
    Read(B);
       B:=B+500;
 Write(B);
    Commit;
 End;
```

Here a Read(X) implies reading the value of the database item X, into the local buffer area , by a transaction T_i.

A Write(X) implies writing the value of X from the local buffer to its database image, by a transaction T_i.

A Commit statement ensures that the changes made by the transaction are made permanent to the database.

Properties of Transactions

2

Oct. 2018 – *1M*
State the atomicity properties of a transaction.

Apr. 2018 – *5M*
List and explain properties of transaction.

Every transaction must follow certain properties called as ACID properties, in order to ensure the integrity of data in the database. They should be enforced by the concurrency control and recovery methods of the DBMS.

The following are the ACID properties:

ACID Properties
i. Atomicity
ii. Consistency preservation
iii. Isolation
iv. Durability or permanency

i. **Atomicity:** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all. The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity.

Example, Now suppose that the balances of the accounts of A and B are 1000/- and 2000/- respectively. Suppose a failure occurs during the execution of transaction T_i and hence T_i does not complete fully. And if the failure happens after Write(A) operation was executed, but before Write(B) is executed.

In this case, values in the accounts of A and B in the database are 500/- and 2000/- respectively. Thus we have lost 500/- as a result of failure and hence the sum (A+B) is no longer valid and hence leads the system into inconsistent state. To avoid this, the transaction should posses the isolation property.

ii. **Consistency preservation:** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data

items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

The consistency requirement here is that the sum of A and B be unchanged by the execution of transaction. Since the transaction is decreasing A's account and increasing B's account by the same account.

iii. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered by any other transactions executing concurrently.

When several transactions are interleaved in order to run concurrently, it may result in an inconsistent state of the database.

For example: At the point of time, when A's account is decremented but B's account not yet incremented, if some other concurrently running transaction reads the value of A and B and computes A+B, it will get an inconsistent value. This happens because the changes made by T_i to A was visible to other concurrently running transactions. Thus to avoid such situations, the transaction must be executed in isolation, i.e., the changes done by a transaction must not be visible to other concurrently running transactions until the later transaction completes execution successfully.

iv. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

This property guarantees that once a transaction completes successfully, all updates that is carried out on the database remains, even if there is a system failure after the transaction completes execution.

States of Transaction

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts therefore, the recovery manager of the DBMS needs to keep track of the following operations:

i. **BEGIN_TRANSACTION:** This marks the beginning of transaction execution.

ii. **READ or WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.

iii. **END_TRANSACTION:** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

iv. **COMMIT_TRANSACTION:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

v. **ROLLBACK (or ABORT):** This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be undone.

Fig. 2.1 shows a state transition diagram that illustrates how a transaction moves through its execution states.

State of transaction
i. Active state
ii. Partially committed state
iii. Committed state
iv. Failed state
v. Aborted state

i. **Active state:** A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations.

ii. **Partially committed state:** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.

iii. **Committed state:** The transaction is said to have reached its commit point and enters the *committed state*. When a transaction is committed, it has concluded its execution

successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

iv. **Failed state:** A transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

v. **Aborted state:** The **aborted** state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later either automatically or after being resubmitted by the user as brand new transactions.

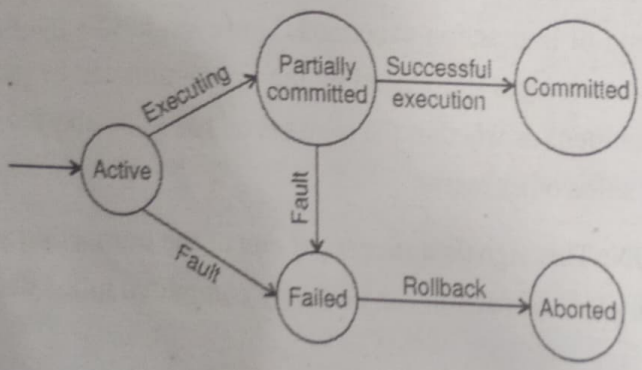

Figure 2.1: State diagram of transaction

2

Oct. 2018 – *5M*
With the help of a state transaction diagram, explain the states in which a transaction can be during its execution.
Oct. 2017 – *5M*
Explain states of transaction with diagram.

3. Concurrent Execution of Transaction and Conflicting Operations

Transaction processing systems usually allow multiple transactions to execute concurrently. When multiple transactions update data concurrently, many complications arise with respect to consistency of data. Consistency of data has to be ensured, inspite of concurrent execution of transactions.

Having transactions execute concurrently results in many advantages such as improved throughput and resource utilization, reduced waiting time, reduced average response time, etc.

The database system must control the interaction among the concurrent transactions so as to maintain the consistency of data within the database. This is done using a variety of mechanisms called *concurrency control mechanisms*.

We will now see the problems that lead to inconsistency of data, in case of concurrent execution of transactions:

Various problems are encountered when transactions run concurrently

i. **Lost update problem:** This problem occurs when two transactions that access the same database items have their operations interleaved in such a way that it makes the value of some database items incorrect. Assume that transactions, T_1 and T_2, have taken place as shown below:

T_1	T_2
read item(x)	
X := X – N	
	read item(x)
	X := X + M
write item (x)	
read item (y)	
	write item(x)
Y := Y + N	
write item(y)	

Figure 2.2

Here the value of x written by T_1 is updated by T_2, even before the transaction T_2 commits; hence resulting in incorrect value for x. Thus the update done by an active transaction T_1 is lost to another concurrently running transaction T_2.

ii. **The temporary update (or Dirty read) problem:** This problem occurs when one transaction updates a database item and this uncommitted value is accessed by another concurrently running transaction. The updated items of an active transaction is accessed by another transaction before the transaction commits.

T_1	T_2
read item(x)	
X := X – N	
	read item(y)
	X := X + M
write item (x)	
read item (y)	
	read item(x)
Y := Y + N	
write item(y)	

Figure 2.3

In the above *example*, the value written by T_1 is read by T_2, even before T_1 commits. Due to this, in case T_1 fails after writing x, then before X can be changed to it's original, its updated value has been read by T_2, thereby leading to a dirty value of x being read by T_2.

iii. **The incorrect summary problem:** If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others, after they are updated.

4. Schedules of Transaction

Schedules represent the chronological order in which the instructions from concurrently running transactions are executed in the system. 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.

2

Apr. 2018 – *1M*
What is schedule? Give types of schedule.

Oct. 2017 – *1M*
Define: Schedule.

For example: If T_1 and T_2 are two transactions running concurrently then a schedule S of $\{T_1, T_2\}$ can be given as follows in *figure 2.4.*

T_1 :	T_2 :
read(B) B := B + 10 write(B) commit	read(A) A := A * 0.1 read(B) B := B + A write(B) commit

S :	T_1	T_2
	Read (B) B : = B + 10	
		Read (A) A : = A * 0.1
	Write (B) Commit	
		Read (B) B : = B + A Write (B) Commit

Figure 2.4: A schedule S of Transactions, T_1 and T_2

A schedule S of n transactions, $T_1, T_2, \ldots, T_n$ is an ordering of the operations of the transactions, subject to the constraint that for each transaction, T_i that participates in S, the operations of T_i in S appear in the same order in which they occur in T_i. The operations of other transactions T_j can be interleaved with the operations of T_i in S.

Two operations in a schedule are said to conflict if they belong to different transactions, if they access the same item X, and if one of the two operations is a write(X). Thus the set of conflicting operations can be given as a RW (a Read-Write), a WR (a Write-Read), and a WW (a Write-Write).

For example: In schedule S of *fig. 2.4* write (B) of T_1 conflicts with read(B) of T_2. But the operations Read(B) of T_1 and Read(B) of T_2 don't conflict, because both are read operations.

4.1 Types of Schedules

Oct. 2018 – *1M*
What is a complete schedule?

i. **Complete Schedules:** A schedule S of n transactions, $T_1, T_2, \ldots, T_n$, is said to be a complete schedule if the following conditions hold:

   a. The operations in S are exactly those operations in $T_1, T_2, \ldots, T_n$ including a commit or abort operation as the last operation for each transaction in the schedule.

   b. For any pair of operations from the same transaction T_i, their order of appearance in S is the same as their order of appearance in T_i.

   c. For any two conflicting operations, one of the two must occur before the other in the schedule.

ii. **Characterizing Schedules based on Recoverability:** It is important to characterize schedules based on the recovery factor, i.e., considering types of schedules for which recovery is possible and those for which recovery is relatively simple.

   Schedules based on recoverability are as follows:

   a. ***Recoverable Schedules:*** A schedule S is said to be recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

      A transaction T is said to read from transaction T' in a schedule S if some item X is first written by T' and later read by T. In the schedule S, T' should not have aborted before T reads item X and there should be no other transactions that write X after T' writes it and before T reads it. (Unless those transactions if any have aborted before T reads X).

Fig. 2.5 shows a recoverable schedule S of Transactions, T_1 and T_2.

S :	T_1	T_2
	R(X)	
	X := X + Y	
	W(X)	
		R(X)
		R(Y)
		Y := X + Y
	R(Z)	
	commit	
		R(Z)
		commit

Figure 2.5: A recoverable schedule S (T_1 , T_2)

Note That above R (X) implies read (X) and W (X) implies write (X).

Fig. 2.6 shows a non recoverable schedules S' of T_1, T_2.

S' :	T_1	T_2
	R(X)	
	X := X + 10	
	W(X)	
		R(X)
		R(Y)
		Y := Y + X
		W(Y)
		commit
	R(Z)	
	commit	

Figure 2.6: A non recoverable schedule S' (T_1, T_2)

Here S' is non-recoverable because T_2 reads the item X written by T_1 and then T_2 commits before T_1 commits. If T_1 aborts after R(Z) operation, then the value of X read by T_2 is no longer valid and T_2 must be aborted after it had already committed, leading to a schedule that is not recoverable.

In a recoverable schedule, a committed transaction need not be rolled back. But a phenomenon called cascading rollbacks can occur.

For example: In *fig. 2.6*, in schedule S, if T_1 aborts after R(Z), then T_1 is rolled back, but since T_2 has read the value written by T_1 and is still active, T_2 also needs to be rolled back. Thus, rolling back T_1 leads to rolling back T_2 also, which leads to cascading rollbacks.

b. ***Schedules that Avoid Cascading Rollback:*** A schedule S is said to avoid cascading rollback, if every transaction in the schedule S only reads items that were written by committed transactions. Thus in this case, all items read will be committed and hence no cascading rollbacks.

Fig. 2.7 shows a schedule S_1 (T_1, T_2) that avoids cascading rollbacks.

S_1 : T_1	T_2
R(X)	
W(X)	
commit	
	R(X)
	R(Y)
	X := X + Y
	W(X)
	commit

Figure 2.7: Avoiding cascading rollbacks recoverable schedule $S_1(T_1,T_2)$

c. ***A Strict–Schedule:*** A Strict–schedule is a more restrictive schedule in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted). A strict schedule is a recoverable schedule and also avoids cascading rollbacks.

Based on the ordering of instructions across the participating transactions, in a schedule, we further have the following types of schedules as follows:

Two types of schedules
i. Serial schedule
ii. Concurrent schedule

i. **Serial schedule:** It consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule, as shown below.

T_1	T_2
read item(x)	
X := X – N	
write item (x)	
read item (y)	
write item (x)	
read item (y)	
Y := Y + N	
write item (y),commit	
	read item(x)
	X := X + M
	write item (x), commit

Figure 2.8

Similarly, we can have a serial schedule of transaction T_2 followed by transaction T_1. Thus, for a set of n transactions, there exist n! different valid serial schedules.

ii. **Concurrent schedule (Non-serial schedule):** When several transactions are executed concurrently, the corresponding schedule is called concurrent schedule. Several execution sequences are possible since the various instructions from both transactions may now be interleaved.

Examples of Schedules

Consider the banking system of several accounts and a set of transactions that accesses and updates those accounts. Let T_1 and T_2 be two accounts.

T_1: Transfers Rs.50 from account A to account B.

T_2: Transfers 10% of balance from account A to B.

```
T1:  read(A)
     A := A – 50
     write(A)
     read(B)
     B = B + 50
     write(B)
T2:  read(A)
     Temp: = A * 0.1
     A := A – temp
     write(A)
     read(B)
     B := B + temp
     write(B)
```

For T_1 and T_2, two serial transactions are possible.

i. $<T_1, T_2>$ Serial schedule

Schedule 1	
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 2	
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)	

For T_1 and T_2, number of concurrent schedules are possible but not all the transactions are consistency preserving.

Schedule 3 Concurrent (consistent) 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)	
	read(B) B := B + temp write(B)

Schedule 4 Inconsistent concurrent schedule for T_1 and T_2	
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)

4.2 Concept of Serializability

For a transaction, a serial schedule always results in a consistent database and not every concurrent schedule can result in consistent database.

But a concurrent schedule results in a consistent state if its result is equivalent to a serial schedule of that transaction. Such concurrent schedule is known as *serializable*.

A serializable schedule is defined as

Given (an interleaved execution) a concurrent schedule of n transactions;

The following conditions hold for each transaction in the set

i. All transactions are correct, i.e., if any one of the transactions is executed on a consistent database, the resulting database is also consistent.

ii. Any serial execution of the transaction is also correct and preserves the consistency of the database.

The given concurrent schedule is said to be serializable if it produces the same result as some serial schedule of the transaction.

4.3 Schedule Equivalence

There are two forms of schedule equivalence
i. Conflict
ii. View

There are two forms of schedule equivalence.

Conflict Serializability

Consider that T_1 and T_2 are two transactions and S is a schedule for T_1 and T_2. I_i and I_j are two instructions.

1

Oct. 2018 – 4M
Explain conflict serializability with suitable example.

If I_i and I_j refer to different data items, then I_p and I_j can be executed in any sequence.

But if I_i and I_j refer to the same data items, then the order of the two instructions may matter.

Here I_i and I_j can be a read or write operation only.

Hence, following four conditions are possible.

a. I_i = read (A)

I_j = read (A)

The order of I_i and I_j does not matter because both are reading the data.

b. I_i = read (A) I_j = write (A)

I_i = write (A) I_j = read (A)

Here, if read (A) is executed before write (A) then it will read the original value of A. Otherwise, it will read that value of A which is written by write (A). Hence, the order of I_i and I_j matters.

c. I_i = write (A) I_j = write (A)

Here, order of I_i and I_j does not affect either T_i or T_j. But the database is changed and it makes difference for next read.

We say that I_i and I_j conflict if they are operated by different transactions on the same data item and at least one of them is write operation.

Example

Concurrent Schedule 5	
T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Here, write (A) of T_1 conflicts with read(A) of T_2. Similarly, write (B) of T_1 conflicts with read(B) of T_2. But write (A) of T_2 does not conflict with read (B) of T_1, because both are accessing different data items.

If I_i and I_j are two consecutive instructions of schedule S, and if they don't conflict, then we can swap the order of such I_i and I_j to produce new schedule S′. We say S and S′ are equivalent. Equivalent Schedule for **Schedule 5** can be obtained by the following swap:

Swap write (A) of T_2 and read (B) of T_1

Swap read (B) of T_1 and read (A) of T_2

Swap write (B) of T_1 and write (A) of T_2

Swap write (B) of T_1 and read (A) of T_2

Schedule 6 (After Swapping)	
T_1	T_2
read(A)	
write(A)	
read (B)	
write (B)	
	read(A)
	write(A)
	read (B)
	write (B)

Thus, concurrent schedule S is transferred to serial schedule S' by a series of swaps of non-conflicting instructions, and schedules S and S' are conflict-equivalent. We say that a schedule S is conflict serializable if it is conflict-equivalent to a serial schedule. There may be any two schedules, which are not conflict equivalent but produce some result.

Schedule 7	
T_1	T_2
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)

Result of above schedule is the same as serial schedule <T_1, T_2>, but is not conflict serializable, since in above schedule write (B) of T_2 conflicts with read (B) of T_1. Thus we cannot move all instructions of T_1 before those of T_2 by swapping consecutive non–conflicting instructions.

4.4 Precedence Graph for Serializability

A Serializability schedule gives the same result as some serial schedule.

A serial schedule always gives correct result, i.e., a schedule that is serializable is always correct.

Hence, we must show that schedules generated by concurrency control scheme are serializable.

i. **Conflict Serializability:** There is an algorithm to establish the serializability of a given schedule for a set of transactions. This algorithm uses a directed graph called precedence graph, constructed from given schedule.

Precedence graph: It consists of a pair G = (V, E) where
V – Set of vertices

The set of vertices consists of all transactions participating in the schedule.

E – Set of edges.

The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of the following conditions holds:

i. T_i executes write (Q) before T_j executes read (Q)

ii. T_i executes read (Q) before T_j executes write (Q)

iii. T_i executes write (Q) before T_j executes write (Q)

A precedence graph is said to be a cyclic if there are no cycles in the graph.

Algorithm: Conflict Serializability

a. Construct a precedence graph G for given schedule S.

b. If the graph G has a cycle, schedule S is not conflict serializable.

If the graph is acyclic, then find, using the topological sort given below, a linear ordering of transactions so that if there is arc from T_i to T_j in G, T_i precedes T_j. Find a serial schedule as follows:

i. Initialize the serial schedule as empty.

ii. Find a transaction T_i, such that there are no arcs entering T_i. T_i is the next transaction in the serial schedule.

iii. Remove T_i and all edges emitting from T_i. If the remaining set is non-empty, return to (ii), otherwise the serial schedule is complete.

Examples

1.

Schedule Table 1

T_1	T_2	T_2
read(A)		
	read(B)	
$A := f_1(A)$		
		read(C)
	$B := f_2(B)$	
	write(B)	
		$C := f_2(C)$
		write(C)
write(A)		
	read(A)	
	$A := f_4(A)$	
read(C)		
	write(A)	
$C := f_5(C)$		
write(A)		
		$B := f_6(B)$
		write(B)