Transactions

Transaction Processing Concept:

- Transaction System
- Testing of Serializability, Serializability of Schedules, Conflict & View Serializable Schedule
- Recoverability, Recovery from Transaction, Failures, Log Based Recovery, Checkpoints,
- Deadlock Handling
- Distributed Database: Distributed Data Storage, Concurrency Control, Directory System

Transaction Concept

Definition of a Transaction:

- A transaction is a unit of program execution that accesses and possibly updates data items in a database.
- It is usually initiated by a user program written in a high-level language (e.g., SQL, COBOL, C, C++, Java).
- It is delimited by statements like begin transaction and end transaction.
- All operations between these two statements form one transaction.

ACID Properties:

Atomicity: Either all operations of the transaction are reflected properly in the database, or none are.
Consistency: Guarantees that a transaction, when executed in isolation, preserves the consistency of the database (that is, with no other transaction executing concurrently) . The database moves from one consistent state to another.

Transaction Concept….

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.

Transactions access data using two operations:
read(X): Copies the data item X from the database into the transaction's temporary memory (local buffer).
write(X): Copies the data item X from the transaction's temporary memory (local buffer) back to the database.
In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored in memory and executed on the disk later.

Transaction Concept….

Example:
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)
```

Consistency :
- Consistency ensures that a transaction preserves the correctness of the database.
- If there are two accounts, A and B, the total sum (A + B) must remain the same after a transaction (e.g., transferring money).
- Without consistency, money could be created or lost during transactions.
- If the database is consistent before the transaction, it must also be consistent after the transaction completes.
- Ensuring consistency for each transaction is mainly the responsibility of the application programmer who writes the transaction code.

Transaction Concept…

```
Ti:  read(A);
     A := A – 50;
     write(A);
     read(B);
     B := B + 50;
     write(B)
```

Atomicity :

Initial Values:
Account A = $1000, Account B = $2000. Total = $3000.
During Transaction Ti: Ti transfers $50 from A to B.
After write(A), A = $950.
Failure occurs before write(B), so B = $2000.
Problem:Total becomes $2950, meaning $50 is lost.
The database is now in an inconsistent state

- During normal execution, temporary inconsistency (A = $950, B = $2000) may occur.
- But after successful completion, the database returns to a consistent state (A = $950, B = $2050).
- To prevent partial updates from being visible. The transaction must execute all operations or none.
- The database keeps a log of old values (on disk) before performing writes. If a failure occurs, it restores the old values, making it appear as if the transaction never happened.
- Ensuring atomicity is handled by the transaction-management component of the database system.

Transaction Concept….

Durability:

- Once a transaction completes successfully, its results must permanently remain in the database.
- No system failure should erase or undo the completed changes.
- After a successful transfer and confirmation to the user, the updated account balances must not be lost, even if the system crashes immediately after.
- All updates made by a successful transaction will persist despite power failure, crash, or reboot.
- Data in main memory may be lost during a failure.
- Data written to disk is assumed to be safe and permanent.
- Either: a. All updates are written to disk before the transaction is marked complete, or b. Sufficient information (logs) is written to disk to reconstruct updates after recovery.
- The Recovery-Management Component of the database system ensures durability.

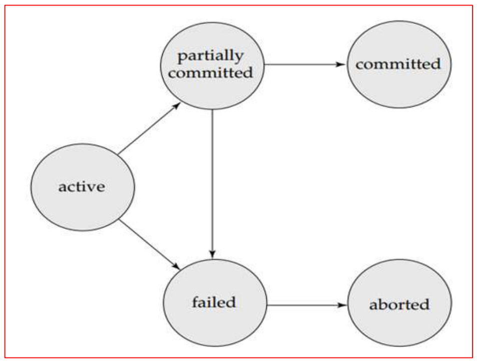
Transaction Concept….

Isolation:

- When two or more transactions run at the same time, one transaction may read data that another has partially updated, leading to inconsistent results.
- For example, during a fund transfer from A to B, if another transaction reads A and B before the transfer finishes, it may see incorrect totals.
- If that second transaction, then updates based on these inconsistent values, the database may remain inconsistent even after all transactions complete.
- One way to prevent this is to run transactions serially (one after another), but this reduces performance.
- Concurrent execution improves performance, so systems use special methods to allow safe concurrency.
- The isolation property ensures that concurrent transactions produce the same result as if they were executed one after another.
- Maintaining isolation is handled by the concurrency-control component of the database system.

Transaction Concept….

Transaction state:



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

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**.

Transaction Concept….: Transaction state:

- 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. 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.

Concurrent Executions

There are two good reasons for allowing concurrency

1. Improved throughput and resource utilization

- A transaction includes multiple steps some require I/O operations, while others require CPU processing.
- The CPU and disks can work simultaneously (in parallel).I/O operations of one transaction can occur while the CPU processes another transaction.
- Different disks can also handle separate read/write operations at the same time.
- This parallel execution of transactions increases system throughput (more transactions completed per unit time).
- It also improves resource utilization, meaning the CPU and disks remain busy doing useful work instead of staying idle.

2. Reduced waiting time

- Transactions in a system can be short or long in duration.
- If executed serially, short transactions must wait for long ones to finish, causing unpredictable delays.
- When transactions access different parts of the database, they can safely run concurrently.
- Concurrent execution allows sharing of CPU and disk resources among transactions.
- This reduces waiting time and unpredictable delays.
- It also lowers the average response time, meaning transactions complete faster on average after submission.

Transactions

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.

```
T2: read(A);
temp := A * 0.1;
A := A – temp;
write(A);
read(B);
B := B + temp;
write(B).
```

Transactions

Suppose the current values: Account A : $1000 and Account B : $2000

Schedule 1: a serial schedule in which T1 is followed by T2.

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)

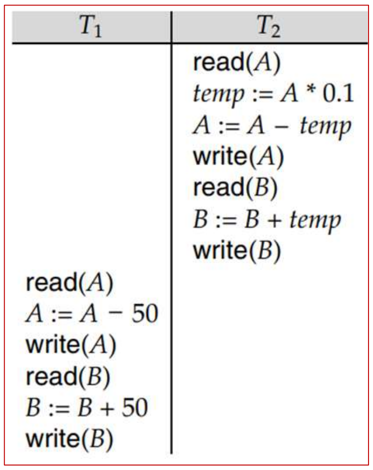
The final values of accounts *A* and *B*, after the execution of Schedule 1, are $855 and $2145

The sum of A+B is preserved: Consistent state

Transactions

Suppose the current values: Account A : \$1000 and Account B : \$2000

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



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)	

The final values of accounts *A* and *B*, after the execution of Schedule 2, are \$850 and \$2150

The sum of A+B is preserved: Consistent state

Transactions

Schedule: A transaction is a set of instructions and each instruction performs an operation on database.
When multiple transactions execute concurrently then there is a need of ordering theses sequence of operations because only one operation can be performed on a database at a time.
A schedule is a sequence of operations by a set of concurrent transactions T1, T2, …, Tn that preserves an ordering of the operations of the individual transactions. The execution sequences just described are called **schedules**.

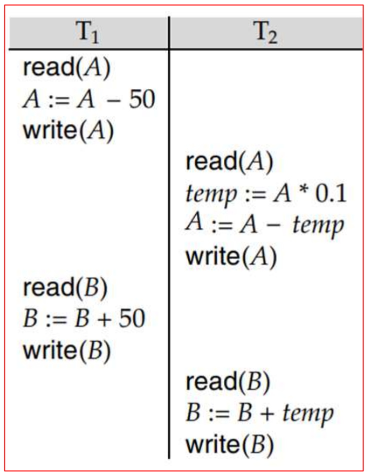
Serial Schedule: A serial schedule means the steps of each transaction are executed together, one after another.

- Instructions of a single transaction do not interleave with others.
- For n transactions, there are n! (factorial) possible serial schedules.

- Example: For 3 transactions (T1, T2, T3), there are 3! = 6 possible serial schedules.

Transactions

Schedule 3: a concurrent schedule equivalent to schedule 1



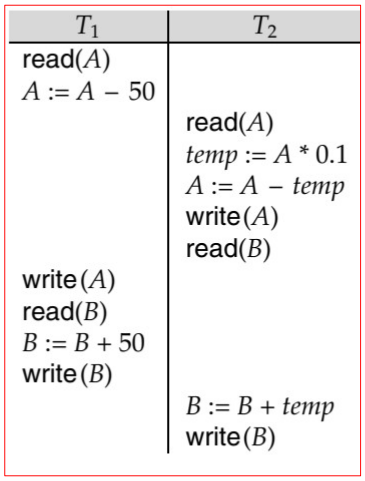
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

The final values of accounts *A* and *B*, after the execution of Schedule 3, are \$855 and \$2145 ?

The sum of A+B is preserved: Consistent state

Transactions

Schedule 4: a concurrent schedule



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)

The final values of accounts *A* and *B*, after the execution of Schedule 4, are \$950 and \$2100

This final state is an inconsistent state, since we have gained \$50 in the process of the concurrent execution.

The sum A + B is not preserved by the execution of the two transactions: Inconsistent state.

Serializability

Serializable Schedule:

Suppose S is a concurrent schedule and is logically equivalent to a serial schedule S', then S is said to be a Serializable Schedule.

This refers to a situation, wherein the concurrent execution of two transactions (say T_i and T_j) is logically equivalent to their serial execution i.e. either T_i followed by T_j or T_j followed by T_i.

Importance of read and write instructions in a schedule: *It is difficult to determine the exact instructions of a transaction and how these instructions of various transactions interact. For this reason, the only significant instructions that we shall consider read and write instructions only in further schedules*

Types of Serializability: The serializability of schedules is of two types:-

(a)Conflict Serializability

(b)View Serializability

Serializability

Conflict Serializability: Let us consider a schedule S in which there are two consecutive instructions Ii and Ij, of transactions T_i and T_j, 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:

I_i = read(Q), I_j = read(Q). The order of Ii and Ij does not matter.

I_i = read(Q), I_j = write(Q).

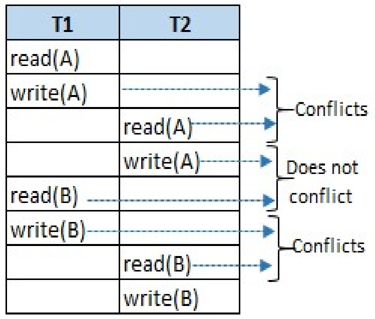
If I_i occurs before I_j, then T_i does not read the value of Q that is written by T_j in instruction I_j. If I_j occurs before I_i, then T_i reads the value of Q that is written by T_j. Thus, the order of I_i and I_j matters.

I_i = write(Q), I_j = read(Q). The order of I_i and I_j matters for reasons similar to the case-2.

I_i = write(Q), I_j = write(Q).

Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after 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.

Serializability



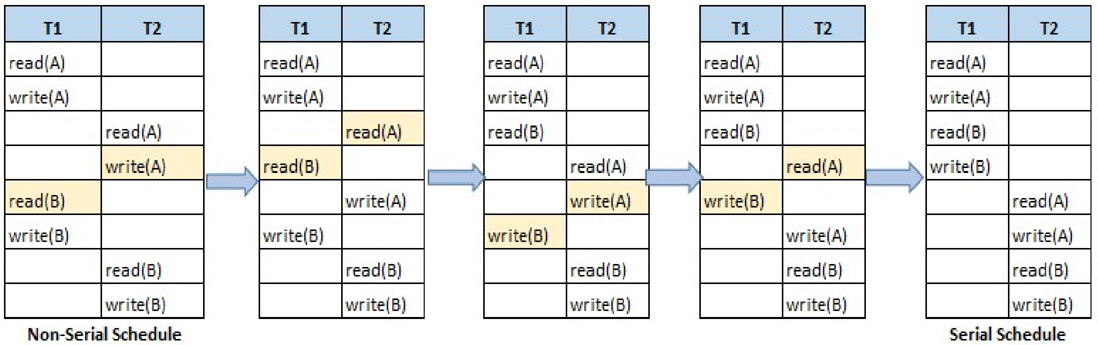
T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Conflict Serializability:

Non-conflict instructions: When both Ii and Ij are read instructions then the relative order of their execution does not matter.

Conflict instructions: 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. The figure shows the conflicting and non-conflicting instructions in two transactions T_1 and T_2.

Serializability : Conflict Serializability



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

Non-Serial Schedule

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

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

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

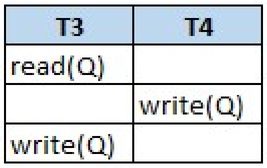
T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Serial Schedule

If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are conflict equivalent.

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.

Serializability



T3	T4
read(Q)	
	write(Q)
write(Q)	

Conflict serializable ?

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>.