Transaction Processing Concepts in DBMS

Unit – V

Session - 1

Transaction(s) in DBMS

- logically related operations executed as a **single unit**
- essential for handling user requests to access and modify database contents
- various operations and has various states in its completion journey
- specific properties that must be followed to keep the database consistent

Operations of Transaction

- **Read(X)** when a user wants to check **his/her** account's balance
  - Find the address of the disk block on the database that contains item X.
  - Copy that disk block into a buffer in main memory
  - Copy item X from the buffer to the program variable named X.
- **Write(X)** when a user requests to withdraw some money, fetch account balance and write the balance post withdrawal
  - Find the address of the disk block oh the database that contains item X
  - Copy that disk block into a buffer in main memory
  - Copy item X from the program variable to the data item X in the buffer
  - Store the updated block from the buffer back to database
- **Commit** changes made by a transaction permanently to the database
- **Rollback** bring the database to the last saved state

why concurrency control ?

- **Lost Update problem**

occurs when two transactions access the same DB item having their operations interleaved in a way that makes the value of some DB item incorrect.

- **Dirty Read problem**

occurs when transaction updates a DB item and the transactions fails for some reason. Here the updated item is accessed by another transaction before its changed to its original value due to failure

- **Incorrect Summary problem**

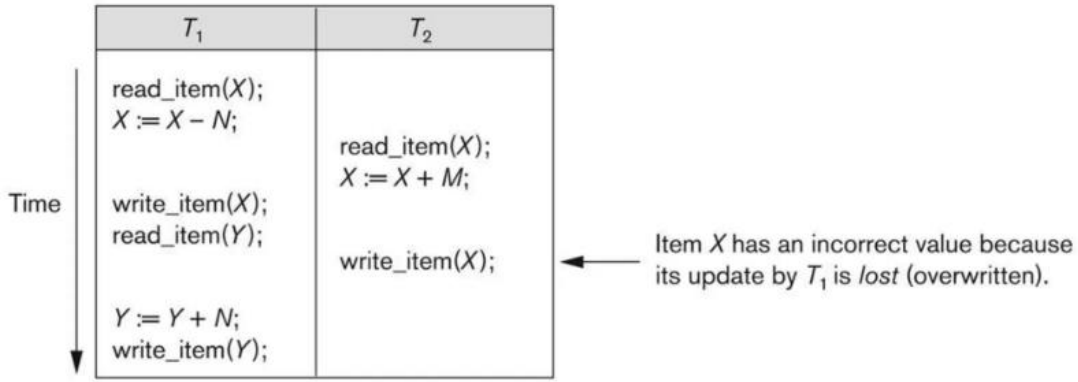
occurs when a transaction is calculating an aggregate or a summary function, Ex. Aggregating, on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and other summary functions after they are updated, thus leading to in-consistent results

- **Unrepeatable read problem**

occurs when transaction reads an item twice where item is changed by another transaction between the two reads

why concurrency control ?

- **Lost Update Problem**
  - When two transactions that update the same database items have their operations interleaved in a way that makes the value of some database item incorrect



why concurrency control ?

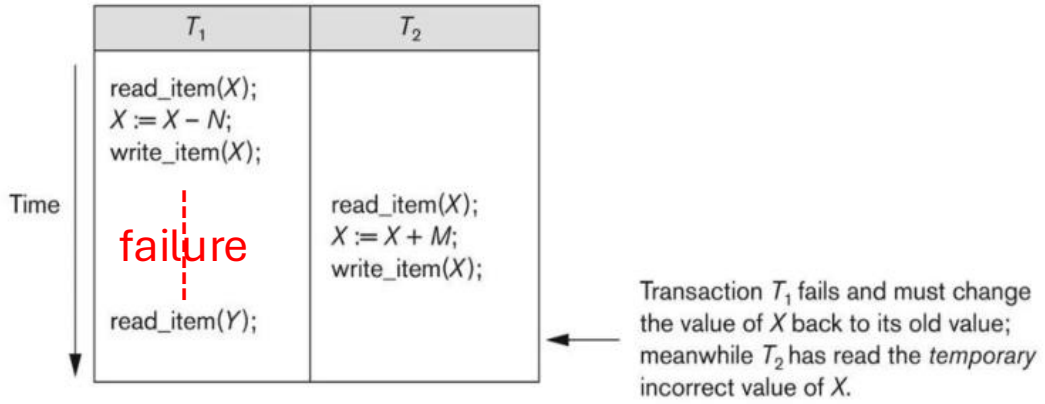
- **Lost Update problem - example**

with X = 50 and Y =50 (initial values)

T1	T2
read(x) (T1I1)	
x=x+10 (T1I2)	
	read(x) (T2I1)
	x=x+20 (T2I2)
write(x) (T1I3)	
read(y) (T1I4)	
	write(x) (T2I2)
	commit (T2I2)
y=y+10 (T1I5)	
write(y) (T1I6)	
commit (T1I7)	

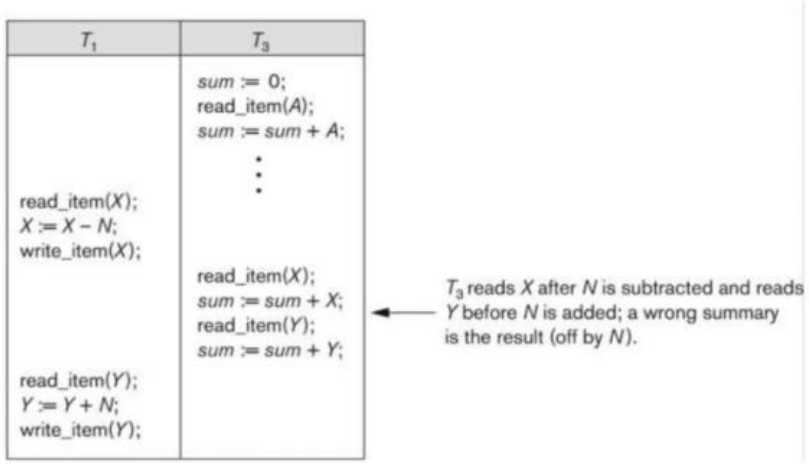
why concurrency control ?

- **Temporary Update (or Dirty Read) Problem**
  - When one transaction updates a database item and then the transaction fails for some reason
  - The updated item could be accessed by another transaction



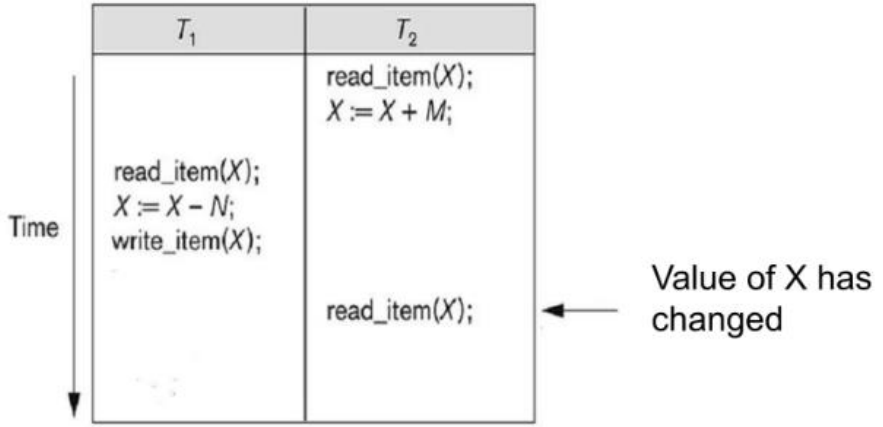
why concurrency control ?

- **Incorrect Summary Problem**
  - If a transaction is calculating an aggregate function while others are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated



why concurrency control ?

- **Nonrepeatable Read Problem**
  - If a transaction reads the same data item twice and the item is changed by another transaction between the two reads



why Transaction(s) Recovery

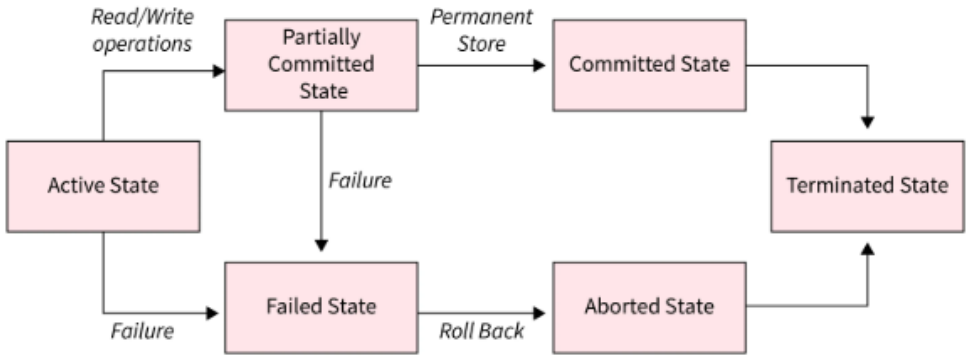
Types of failures

- **computer failure/system crash**
  -- software / hardware/main memory/network failures
- **transaction/system error**
  -- logical / runtime errors
- **exceptions detected by transaction**
  -- due to exception scenarios while execution
- **concurrency enforcement**
  -- due to concurrency control rules while execution
- **disk failures**
  -- loss of data blocks due to read/write malfunction
- **catastrophic failures**
  -- complete / sudden / electronic / unexpected breakdown such as disk crash / memory chip failure / power surge

Transaction states

Different Transaction states due to DB operations

- begin_transaction
- read / write
- end_transaction
- commit
- rollback



Properties of Transaction (ACID)

- **Atomicity (resp: transactions recovery subsystem)**

each transaction is treated as a single unit (like an atom)

a transaction can never be completed partially

transactions are completed using COMMIT or aborted using ROLLBACK

- **Consistency (resp: programmer)**

transaction keeps database consistent before and after a transaction is completed

a transaction should be the transformation of a database from one consistent state to another consistent state

changes made in the database are changes desired to perform and there is not any ambiguity

- **Isolation (resp: concurrency control subsystem)**

two transactions must not interfere with each other viz., other transaction can not concurrently access that data until the first transaction has completed

enforced by the concurrency control subsystem of DBMS

- **Durability (resp: concurrent control subsystem)**

changes made to the database after a transaction is completely executed, are durable

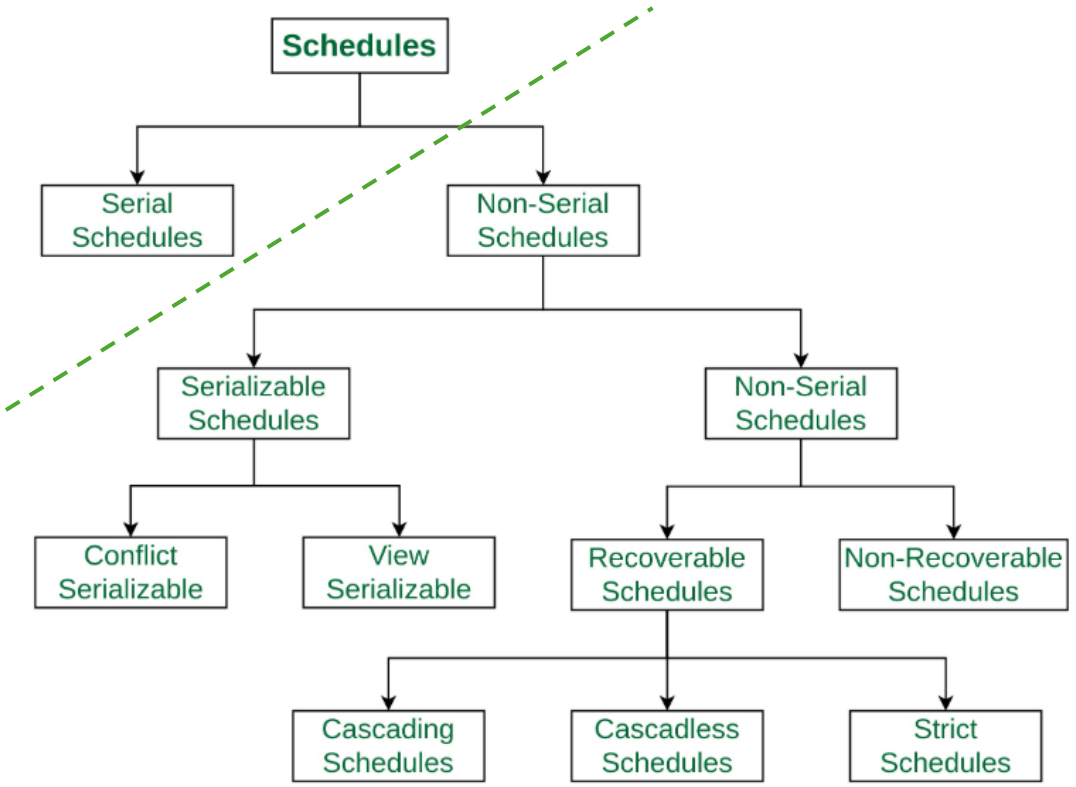
any system failures or crashes, the consistent state achieved after the completion of a transaction remains intact. The recovery subsystem of DBMS is responsible for enforcing this property

why these transaction schedule(s) ?

Characteristics of a transaction schedule will determine

- whether concurrent execution of transactions are “correct”
- whether it is “easy” to recover from transaction failures

Types of Schedules



Transaction Schedules

- **Serial Schedule**

multiple transactions are to be executed serially i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed

Pros: Consistency

Cons:

limit concurrency or interleaving of operations

while I/O wait, no switching of CPU processor to another transaction

increases the waiting time of the transactions in the queue, lowers the throughput of the system

- **Non-Serial Schedule**

Concurrent execution without giving up any correctness of data

allow multiple transactions to start before a transaction is completely executed

referred to as parallel schedules, as transactions execute in parallel

Pros: consistency of the database, handled through algorithms by the DB system and improve CPU throughput and overall system efficiency

Cons: inconsistency and errors in database operation, but handled through algorithms by the DB system

Schedules based on recoverability

S → Schedule T → Transaction

“recoverable schedules”
Schedule1:

read-1(X);
read-2(X);
write-1(X);
read-1(Y);
write-2(X);
commit-2;
write-1(Y);
commit-1;

“non-recoverable schedules”
Schedule2:

read-1(X);
write-1(X);
read-2(X);
read-1(Y);
write-2(X);
commit-2;
abort-1;

Schedules Based on Recoverability

generally, three types of schedules.

- **Recoverable/cascading schedule**

a transaction that has updated the database must commit before any other transaction reads or writes the same data

it allows for the recovery of the database to a consistent state after a transaction failure

If a transaction fails before committing, its updates must be rolled back, and any transactions that have read its uncommitted data must also be rolled back

- **cascadeless schedule**

if it does not result in a cascading rollback of transactions after a failure

a transaction that has read uncommitted data from another transaction cannot commit before that transaction commits

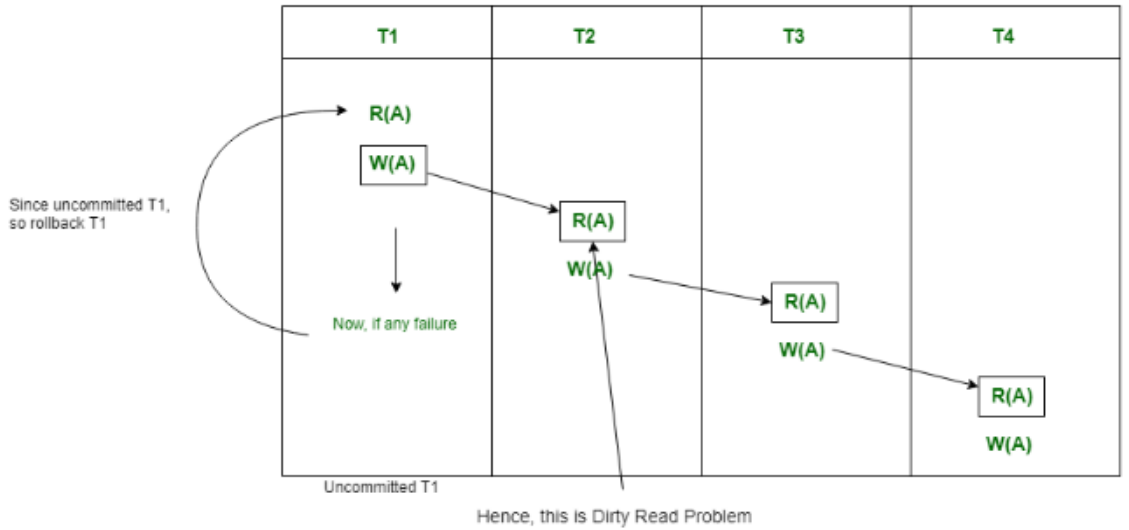
If a transaction fails before committing, its updates must be rolled back, but any transactions that have read its uncommitted data need not be rolled back

- **strict schedule**

a transaction that has read uncommitted data from another transaction cannot commit before that transaction commits, and a transaction that has updated the database must commit before any other transaction reads or writes the same data

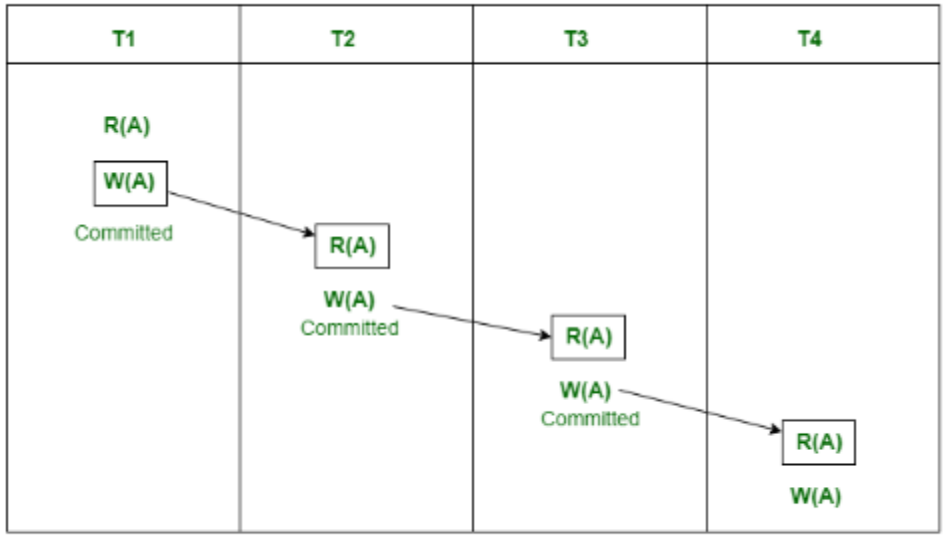
Schedules Based on Recoverability

- **cascading schedule**



Schedules Based on Recoverability

- **cascadeless schedule**



No Dirty Read problem

Schedules Based on Recoverability

- **strict schedule**

T1	T2	T3
R1(x)		
	R2(x)	
R1(z)		
		R3(x)
		R3(y)
W1(x)		
C1;		
		W3(y)
		C3;
	R2(y)	
	W2(z)	
	W2(y)	
	C2;	

Types of Serializable Schedules

Conflict serializability

checks if a schedule of transactions can be transformed into a sequence where transactions are executed one after another, without overlapping, while keeping the same results.

This type of serializability focuses on the order of conflicting operations, meaning those that can affect each other's outcomes.

View serializability

This is a bit broader.

It ensures that even if the transactions overlap, they produce the same final state as some serial execution.

This means that as long as the final view of the database is consistent with a serial order, the schedule is considered valid.

Types of Serializable Schedules

Advantages of Conflict serializability

Data Integrity: Ensures that the final state of the database is consistent, as it prevents conflicting transactions from interfering with each other.

Clear Rules: The rules for conflict serializability are straightforward and easy to understand, making it easier to implement in database systems.

Efficient Execution: Many database systems can optimize transaction execution based on conflict serializability, potentially improving performance.

Strong Isolation: Provides a strong level of isolation between transactions, which can be crucial for applications requiring high reliability.

Disadvantages of Conflict serializability

Restrictive: The strict nature of conflict serializability can lead to reduced concurrency, as it may unnecessarily block transactions that could otherwise run simultaneously.

Complexity in Management: Implementing conflict serializability may require additional mechanisms, such as locking, which can complicate transaction management.

Performance Overhead: The need to check for conflicts and maintain locks can introduce performance overhead, especially in high-load environments.

Not Always Necessary: In some applications, the strict guarantees of conflict serializability may be more than what is needed, leading to inefficiencies.

Deadlock Potential: The use of locks to enforce conflict serializability can lead to deadlocks, where two or more transactions are waiting indefinitely for each other to release resources.

Types of Serializable Schedules

Advantages of View serializability

Greater Flexibility: View serializability allows transactions to overlap, which can improve system performance and resource utilization compared to stricter methods.

Enhanced Throughput: By permitting non-conflicting transactions to run simultaneously, view serializability can enhance throughput in high-transaction environments.

Maintains Consistency: It ensures that the final state of the database is consistent with some serial execution, which is essential for data integrity.

Broader Applicability: It can be used in scenarios where the strict order of operations is not necessary, making it suitable for many real-world applications.

Simpler Management: Since it allows more overlapping operations, the management of transactions can sometimes be less complex compared to conflict serializability.

Disadvantages of View serializability

Complexity of Validation: Determining whether a schedule is view serializable can be more complex than checking for conflict serializability, requiring detailed analysis of transaction outcomes.

Potential for Inconsistency: While it aims to maintain a consistent final state, the overlapping nature of transactions can lead to challenges in ensuring that all intermediate states are valid.

Less Strong Isolation: It does not provide as strong a level of isolation as conflict serializability, which might be a concern for applications requiring high reliability.

Performance Issues: In some cases, allowing too much overlap can lead to performance bottlenecks or resource contention, particularly if transactions are not carefully managed.

Transaction support in SQL

below characteristics attributed to transaction is set using SET TRANSACTION

- **access mode**

  READ WRITE (*default unless isolation level is READ UNCOMMITTED*), READ ONLY

- **diagnostic area size**

  DIAGNOSTIC SIZE *n*

- **isolation level**

  ISOLATION LEVEL < *READ UNCOMMITTED /*
  READ COMMITTED /
  REPEATABLE READ /
  SERIALIZABLE (default) >

Transaction support in SQL

when transactions execute at a lower isolation level than SERIALAZABLE, following violations can occur in the read phenomena

- **dirty read**

  transaction T1 may read the update of a transaction T2, which has not yet committed. If T2 fails and is aborted, then T1 would have read a value that does not exist and is incorrect

- **nonrepeatable read**

  transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

- **phantoms**

  transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE-clause condition used in T1, into the table used by T1. If T1 is repeated, then T1 will see a phantom, a row that previously did not exist.

Transaction support in SQL

below depicts the relationship between isolation levels, read phenomena

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Transaction Processing Concepts in DBMS

Unit – V

End of Session - 1