BYJU'S EXAM PREP

Vision 2023

A Course for GATE & PSUs

Computer Science Engineering

Databases

CHAPTER 3

Transactions and Concurrency Control

CHAPTER 3 DATABASES

TRANSACTIONS AND CONCURRENCY CONTROL

1. TRANSACTION

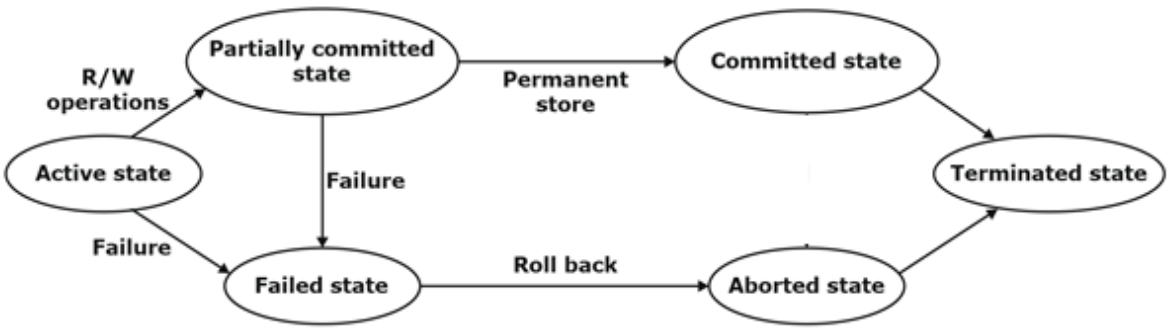
A transaction is a logical unit of work that accesses and potentially modifies a database's contents. Read and write operations are used by transactions to access data.

a. Read Operation:

- Read operation reads the data from the database and then stores it in the buffer in main memory.
- For example- **Read(X)** instruction will read the value of X from the database and will store it in the buffer in main memory.
- Steps are:
  - Find the block that contains data item X.
  - Copy the block to a buffer in the main memory.
  - Copy item X from the buffer to the program variable named X.

b. Write Operation:

- Write operation writes the updated data value back to the database from the buffer.
- For example- **Write(X)** will write the updated value of X from the buffer to the database.
- Steps are:
  - Find the address of the block which contains data item X.
  - Copy disk block into a buffer in the main memory
  - Update data item X in the main memory buffer.
  - Store the updated block from the buffer back to disk.



Transaction States in DBMS

2. ACID PROPERTIES

- To ensure the consistency of the database, certain properties are followed by all the transactions occurring in the system.
- It is important to ensure that the database remains consistent before and after the transaction.
- These properties are called as ACID Properties of a transaction.

A = Atomicity
C = Consistency
I = Isolation
D = Durability

Atomicity:

- This property ensures that a transaction should execute all the operations including commit or none of them i.e. Transactions may or may not occur.
- It is the responsibility of the Transaction Control Manager to ensure atomicity of the transactions.

Consistency:

- This means that integrity constraints must be maintained i.e. Make sure the database remains consistent before and after the transaction.
- Transaction must be logically correct, and correctness should be monitored by the user.
- It is the responsibility of DBMS and application programmer to ensure consistency of the database.

Isolation:

It is the responsibility of the concurrency control manager to ensure isolation for all the transactions.

- ❖ One transaction's execution is segregated from the execution of another, ensuring the concurrent execution of transactions results in the same system state as if the transactions were executed serially, one after the other.
- ❖ The concurrency management manager is in charge of ensuring that all transactions are isolated.i.e., one after the other.

Durability:

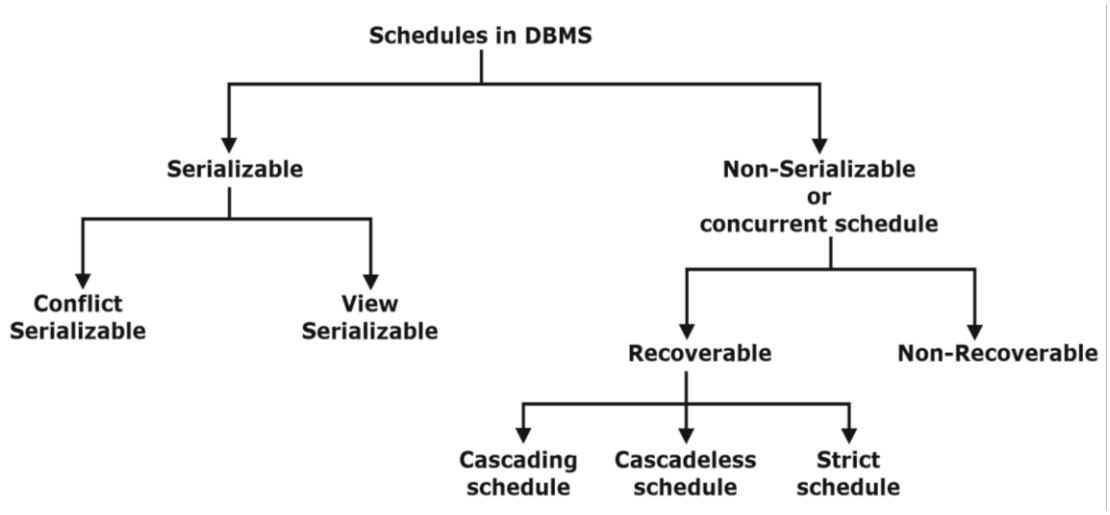
It is the duty of the recovery manager to ensure that the modifications made to the database by a successfully committed transaction remain in the database.

It also ensures that these changes continue indefinitely and are never lost even though there is a failure of some sort.

It is the responsibility of the recovery manager to ensure durability in the database.

3. SCHEDULE

Sequence that indicate the chronological order in which instructions of concurrent transactions are executed.



3.1. Strict Schedule:

- All the transactions execute serially, one after the other i.e. after commit of one transaction begins another transaction.
- Every serial schedule result always preserves integrity.
- When one transaction executes, no other transaction is allowed to execute.
- Less number of users can use the database at a time.
- The number of serial schedules possible with n transactions are n!

Example:

Transaction T1	Transaction T2
R(A)	
W(A)	
R(B)	
W(B)	
Commit	
	R(A)
	W(B)
	Commit

- Transaction T1 executes first and after completion of T1, T2 is executed
- So, this schedule is a scenario of a Serial Schedule.

3.2. Concurrent Schedule:

- Multiple transactions execute simultaneously.
- Operations of all the transactions are interleaved or mixed with each other.
- May cause inconsistency, so to maintain consistency transactions should satisfy ACID property.

Example:

Transaction T1	Transaction T2
R(A)	
W(B)	
	R(A)
R(B)	
W(B)	
Commit	
	R(B)
	Commit

This given schedule is an example of a Non-Serial Schedule since the operations of T1 and T2 are interleaved.

Conflicts in Concurrent schedule:

If all of the respective conditions are met, the two operations become incompatible:

- Both are part of different transactions.
- They have the same data item and at least one write operation in common.

a. Read-Write Conflict:

T_i	T_j
R(A)	⋮
⋮	W(A)

b. Write-Read Conflict:

T_i	T_j	
W(A)	⋮	
⋮	R(A)	→ uncommitted read

c. Write-Write Conflict:

T_i	T_j
W(A)	⋮
⋮	W(A)

Concurrency Problems in DBMS:

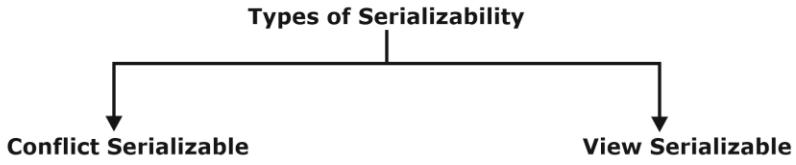
• **Dirty Read**: Reading the data written by an uncommitted transaction is called as dirty read.

Transaction T1	Transaction T2	
R(A)		
W(B)		
.	R(A)	//Dirty Read
.	W(A)	
.	Commit	
.		
.		
.		
.		
Failure		

• T2 reads the dirty value of A written by the uncommitted transaction T1.
• T1 fails in later stages and roll backs.
• Thus, the value that T2 read now stands to be incorrect.
• Therefore, the database becomes inconsistent.

4. SERIALIZABILITY

- It is the classical concurrency scheme. This ensures that a program for executing concurrent transactions is equivalent to a program that executes sequential transactions in some order.
- Some non-serial schedules can cause database inconsistency;
- Serializability is a concept that aids in determining which non-serial schedules are right and will keep the database consistent.



4.1. Conflict Serializability:

- If a schedule can turn into a serial schedule after swapping non-conflicting operations, it is called dispute serializability.
- If the schedule is conflict equal to a serial schedule, it would be conflict serializable.

Conflict serializability: Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there are some Q elements that Ii and Ij have access to, and at least one of those instructions wrote Q.

(i) I_i = read (Q), I_j = read (Q). I_i and I_j don't conflict.

(ii) I_i = read (Q), I_j = write (Q). They conflict.

(iii) I_i = write (Q), I_j = read (Q). They conflict.

(iv) I_i = write (Q), I_j = write (Q). They conflict.

teps to check Conflict Serializable Schedule:

Schedule S is a conflict serializable schedule iff precedence graph of schedule S is acyclic.

Step 1: List all the conflicting operations.

Step 2: For the precedence graph:

- Draw an edge for each conflict pair such that if X_i (V) and Y_j (V) forms a conflict pair then draw an edge from T_i to T_j, in which one of the V has to be a write operation.
- This ensures that T_i gets executed before T_j.

Step 3: Check for cycles in the graph. If there is no cycle, then conflict is serializable otherwise not conflict serializable.

S : $r_1(A)$ $w_2(A)$ $w_2(B)$ $r_3(B)$ $r_3(C)$ $w_1(C)$ Test whether S is conflict serializable or not.

Sol.

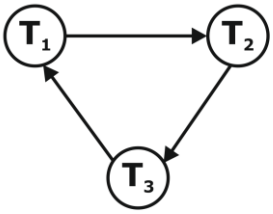
Here

r_1 (A) – w_2 (A)

w_2 (B) – r_3 (B)

r_3 (C) – w_1(C)

are the conflict pairs



Conflict Equivalent:

Two schedules are conflict equivalent if one can be transformed to another by swapping non-conflicting operations.

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. Conflict pairs must have the same precedence in both schedules.

Conflict Equal serial schedules are topological orders of precedence graph of schedule S.

Example:

S2 is the dispute complement of S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Non-serial schedule	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

Schedule S1

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

Schedule S1

Schedule S2 is a serial schedule since it completes all T1 operations before beginning any T2 operations. By swapping non-conflicting operations from S1, schedule S1 can be converted into a serial schedule.

The schedule S1 becomes: after swapping non-conflict operations

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

Since, S1 is conflict serializable.

4.2. View Serializability:

- A schedule will view serializable if it is view equivalent to a serial schedule.
- Conflict serializable which does not contain blind writes are view serializable.

View Equivalent Schedules:

Consider two schedules S1 and S2 each consisting of two transactions T1 and T2

Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them-

i. Initial readers must be same for all the data items:

For each data item X, if transaction T_i reads X from the database initially in schedule S1, then in schedule S2 also, T_i must perform the initial read of X from the database.

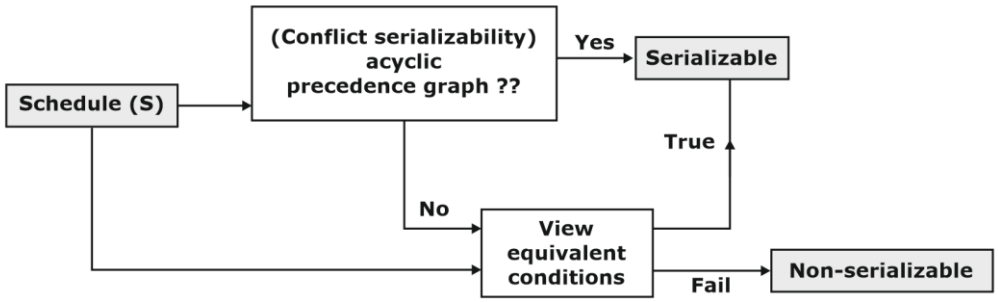
ii. Write-read sequence must be same:

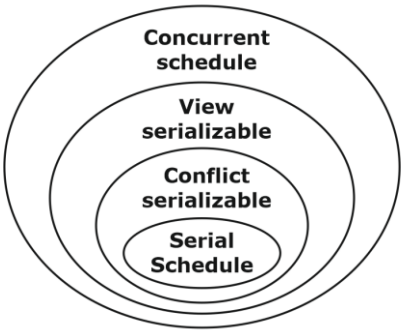
If transaction T_i reads a data item that has been updated by the transaction T_j in schedule S1, then in schedule S2 also, transaction T_i must read the same data item that has been updated by the transaction T_j

iii. Final writers must be same for all the data items:

> NOTE:
> Every conflict serializable schedule is also view serializable.
> Every view serializable schedule that is not conflict serializable has blind writes.

For each data item X, if X has been updated at last by transaction T_i in schedule S1, then in schedule S2 also, X must be updated at last by transaction T_i.

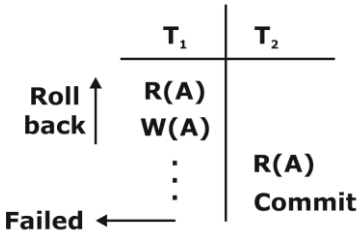




5. RECOVERABLE PROBLEMS

5.1. Irrecoverable Schedule: It is not possible to roll back after the commitment of a transaction as the initial data is nowhere.

Example:



5.2. Recoverable Schedule: A schedule is recoverable if a transaction T_j reads a data item previously written by transaction T_i, the commit operation of T_i appears before the commit operation of T_j.

Example:

T_i	T_j
R(A)	
W(A)	
⋮	R(A)
Commit	W(A)
	Commit

5.3. Cascadeless Recoverable Schedule:

The commit operation of Ti occurs before the read operation of Tj for each pair of transactions Ti and Tj in which Tj reads a data item previously written by Ti. Each cascadeless schedule can be recovered as well.

Example:

T_i	T_j
R(A)	
W(A)	
⋮	
Commit	
	R(A)
	Commit

Cascading rollback: A single transaction failure leads to a series of transaction rollbacks.

5.4. Strict Recoverable Schedule: If transaction T_i updates the data item A, any other transaction T_j not allowed to R(A) or W(A) until commit or roll back of T_i.

6. CONCURRENCY CONTROL PROTOCOLS

It's a DBMS technique for handling several operations at the same time without them colliding. When it comes to the amount of concurrency they allow and the overhead they impose, different concurrency control protocols provide different benefits.

- Lock-Based Protocols
- Two Phase Locking Protocols
- Timestamp-Based Protocols

6.1. Lock Based Protocol

A lock is a data variable that is linked to a specific data object. This lock indicates that operations on the data item are permitted. Concurrent transactions may use locks to synchronise access to database objects.

i. Binary Locks:

- A Binary lock on a data item can either be locked (1) or unlocked (0) states.
- Locked Objects cannot be used by any other transaction.
- Unlocked objects are open to all the transactions.
- Every transaction requires a lock and unlock operation for each data item that is accessed.
- Every transaction locks before use of the data item and releases the lock after performing the operation.

ii. **Shared/Exclusive:**

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

Shared lock(S):

- It is also known as a Read-only lock. In a shared lock, the data item can only be read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

Exclusive lock(X):

- In the exclusive lock, the data item can be both read as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

6.2. Two Phase Locking:

- Two-Phase locking protocol which is also called a 2PL protocol.
- In this given phase of locking protocol, the transaction should acquire a lock after it releases one of its locks.
- There are two phases of 2PL given below:

Growing phase**:** the transaction can acquire a new lock on the data object, but none can be released.

Shrinking phase: The transaction's existing locks may be released during the shrinking process, but no new locks can be obtained.

When the transaction's execution begins in the first section, it requests permission for the lock it needs.

The transaction acquires all of the locks in the second half. As soon as the transaction's first lock is released, the third step begins.

The transaction cannot claim any new locks in the third process. It just unlocks the locks that have been obtained

Problem in 2PL:

- Irrecoverability
- Deadlock
- Starvation

Strict Two- Phase Locking:

- Basic 2PL with all exclusive locks should be held until commit/rollback.
- It ensures serializability is strictly recoverable.

Problem in 2PL:

- Starvation
- Irrecoverability

6.3. Time-Stamp Protocol:

- This protocol ensures that every conflicting read and write operations are executed in timestamp order.

  Since the older transaction has a higher priority, it executes first. This protocol uses device time or a logical counter to calculate the transaction's timestamp.
- The lock-based protocol is used to control the order between conflicting pairs among transactions at the execution time. However, protocols based on timestamps begin functioning as soon as a transaction is made.

Basic Timestamp ordering protocol works as follows:

Let,

The timestamp of the transaction Ti is denoted by **TS(Ti).**

The Read time-stamp of data-item X is denoted by **R_TS(X).**

The Write time-stamp of data-item X is denoted by **W_TS(X).**

1. When a transaction Ti performs a Read (X) operation, make sure to check the following condition:

The operation is rejected if W TS(X) >TS(Ti).

If W TS(X) = TS(Ti), the procedure is carried out.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

- If TS(Ti) < R_TS(X) then the operation is rejected.
- If TS(Ti) < W_TS(X), then the Operation is rejected and Ti is rolled back; otherwise the Operation is executed.

Thomas Write Rule Stamp Ordering Protocol

1. Transaction T issues R(A) option:

(a) If WTS (A) > TS (T) then roll back T

(b) Otherwise execute successfully.

Set RTS (A) = max {TS (T), RTS (A)}

2. Transaction T issues W (A) operation:

(a) If RTS (A) > TS (T) then roll back T.

(b) If WTS (A) > TS (T) then ignore W(A) and continue execution of trans T.

If we use the Thomas write rule then some serializable schedule can be permitted that does not conflict serializable as illustrate by the schedule in a given figure:

T1	T2
R(A)	
	W(A)
	Commit
W(A)	
Commit	

In the above figure, T1's read and precedes T1's write of the same data item. This schedule does not conflict serializable.

Thomas' write rule checks that T2's write is never seen by any transaction. If we delete the write operation in transaction T2, then a conflict serializable schedule can be obtained which is shown in the figure below.

T1	T2
R(A)	Commit
W(A)	
Commit	

Strict Timestamp Ordering Protocol

A transaction T_2 that issues a R(A) or W(A) such that TS(T_2) > WTS (X) has its read write option delayed until the transaction T_1 that writes the value X has committed or rolled back.

6.4. Wait Die Protocol

- Write the transaction in ascending order of timestamp values.
- If T_1 requires a resource that is held by T_2, T_1 wait for T_2 to unlock.
- If T_2 requires a resource that is held by T_1, then roll back T_2 and restart with the same time stamp value.

6.5. Wound Wait Protocol

- Write the transaction in ascending order of TS values.
- If T_1 requires a resource that is held by T_2 then roll back T_2 and restart with same timestamp value.
- If T_2 requires a resource that is held by T_1 than T_2 wait for T_1 to unlock.
- Both wait die and wound wait protocols may have starvation.

****