TRANSACTION PROCESSING

1. Transactions

A transaction is a unit of program execution that accesses and possibly updates various data items i.e., a sequence of operations that is regarded in a single logical operation is called a transaction. A transaction usually results from the execution of a user program written in a high-level data manipulation language or programming language (e.g. SQL, C etc) 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 the transaction.
A transaction can be modeled as a series of reads and writes.

Read (Q) – which transfers the data item Q from the database to a local buffer belonging to the transaction that executed the read operation.

Write (Q) – which transfers the data item Q from the local buffer of the transaction that executed the write back to database.

E.g.: - **Transaction Ti-** Transfers Rs.50 from account A to account B

Ti: read (A);
A=A-50;
Write (A);
Read (B);
B=B+50;
Write (B);

To ensure the integrity of the data, transactions should satisfy some properties called ACID Properties.

2. The ACID Properties

Atomicity **all-or-nothing property**
Either all operation of the transaction are reflected properly in the database or none are. Effect of an incomplete transaction should not be visible in DB
E.g.: If the system fails while executing **read (B)**, the balance of **A** will have been updated but that of **B** haven't. A Transaction is partially done, and the data is made inconsistent.

To ensure atomicity the db system keeps track (on disk) of the old values of any data on which a transaction perform a write and if the transaction does not complete its execution, the old values are restored to make it appear as though the transaction never executed. Ensuring atomicity is the responsibility of the db system specially, it is handled by a component called the **transaction manager component.**

Consistency

Database should be consistent before and after the execution of a transaction. Execution of a transaction in isolation (ie, with no other transaction executing concurrently) preserves the consistency of the db.
E.g. The consistency requirement in Ti is that the sum of A and B be unchanged by the execution of the transaction.

Ensuring consistency is the responsibility of the **application programmer** who codes the transaction. This task may be facilitated by automatic testing of **integrity constraints.**

Isolation
Each transaction is isolated from the effect of concurrent execution of other transactions. i.e., Transactions appear to be executed is serial order, even though there might be concurrent execution of other transactions.
E.g. consider the transaction Tj executing concurrently with T_i, It Tj reads A and B while executing Ti and computes A+B, it will observe an inconsistent value.

A solution to the problem of concurrently executing transactions is to execute transactions serially. i.e., one after the other. However, concurrent execution of transaction provides significant performance benefits. The isolation property ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. i.e., each transaction is unaware of other transactions executing concurrently in the system. Ensuring the isolation property is the responsibility of a component of the db system called in **concurrency controller**.

Durability.
After a transaction completes successfully, the changes it has made to db persist, even if there are system failures.

Values of A and B may be in a memory buffer after the writes have been done before they are flushed to disk. A failure of the computers system may result in loss of data is main memory, but data written to disk are never lost.

We can guarantee durability by ensuring that either (1) updates carried out by the transaction have been written to disk before the transaction completers (2) Information about the updates carried out by the transaction and written to disk is sufficient to enable the db to reconstruct the updates when the db system is restarted after the failure. i.e., transactions are logged and are redone or undone after the system restarts. Ensuring durability is the responsibility of a component of the db system called **recovery management component.**

Scanned by CamScanner
Scanned by CamScanner

3. Transaction Operations

To support transaction processing, a DBMS usually provide three transaction operations: **Start, Commit, and Abort** for database programs. These operations are used in transaction implementations besides **Read, write and other arithmetic operations.**

Start: - Program tells the DBMS that is about to begin execution of a new transaction. The DBMS will have to make certain preparation for its execution.

Commit: - A Program tells the DBMS that the transaction has terminated normally and all of its effects should be made permanent. E.g., the disk should be updated to reflect changes made to the db. Execution of a transactions commit operation guarantees that abort of the transaction is not possible afterward.

Abort: - A program tells the DBMS that the transaction has terminated abnormally and all of its effects should be obliterated. Once the changes caused prior to transaction abortion is undone, we say that the transaction is rolled back to its initial states.

4. Schedule

A schedule is a list of operations (e.g. Read, Write, Start, Abort, Commit) from a set of transactions. The order of appearance of two operations of a particular transaction T in a schedule is the same as that in T.

E.g. Consider the transactions Ti and Tj

Ti-Transfers Rs.50 from a/c A to a/c B

Ti : Read (A);
A=A-50;
Write (A);
Read (B);
B=B+50;
Write (B);
Write (B);

Tj-Transfer 10% of the balance from a/c A to account B

Tj : Read (A);
temp = A*0.1;
A=A-temp;
write (A)
read (B)
B=B+ temp;
write (B);

Serial Schedule

Each serial schedule consists of a sequence of operations from various transactions, where the operation belonging to one single transaction appear together in that schedule. For a set of n transactions (T1, T2…T9), there exist n! different valid serial schedules.

Tn

Scanned by CamScanner

Schedule 1 (Ti and Tj are executed one at a time in the order Ti followed by Tj)

Ti	Tj
Read (A) A=A-50 Write (A) Read (B) B=B+50 Write (B)	Read (A) Temp = A*0.1 Write (A) Read (B) B=B+ temp Write (B)

A + B is consistent after the execution of the schedule

Schedule 2 Serial Schedule (Tj, Ti)

Ti	Tj
Read (A) A=A-50 Write (A) Read (B) B=B+50 Write	Read (A) Temp = A*0.1 Write (A) Read (B) B=B + temp Write (B)

A + B is consistent after the execution of the schedule.

Concurrent schedule

If two transaction are running concurrently, the OS may execute one transaction for a little while then perform the context switch, execute the second transaction for some time, and then switch back to the first transaction for some time and so on. With multiple transactions, the CPU time is shared among all the transaction. In general it is not possible to predict exactly how many operations of a transaction will be executed before the CPU switches to another transaction. Thus the number of possible schedules for a set of n transaction is much larger than n!.

Scanned by CamScanner

Schedule 3 (Concurrent schedule ---Ti and Tj are executed concurrently)

Ti	Tj
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)

A + B is same before and after the execution of the schedule.

Schedule 4 (Another concurrent schedule)

Ti	Tj
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)

Inconsistent state: A+B is not preserved by the execution of the schedule

5. Concurrency controller

Concurrency controller controls the concurrent execution of transactions. If control of concurrent execution is left entirely to the OS, many possible schedules, including ones that leave the db in an inconsistent state such as schedule 4 are possible. It is the job of the db system to ensure that any schedule that gets executed will leave the db is a consistent state. The component of the db system that carries out this task is called concurrency controllers. We can ensure consistency of the db under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution. i.e., the schedule should be equivalent to a serial schedule. Concurrency controller orders the db transaction operations so that the resulting schedule is equivalent to a serial schedule.

6. Serializability.

A schedule is said to be **serialzable** if the result of executing that schedule is the same as the result of executing some serial schedule. So serializable schedule ensures consistency in concurrent execution of transactions. A **schedule S** having transactions Ti and Tj is serializable if S produces the same results as either **<Ti, Tj> or <Tj, Ti>**

Different conditions are satisfied for different kinds of schedule equivalence. They lead to the notions of **conflict serializability** and **view serializability**.

Conflict Serializability

Let Ti and Tj be two transactions with operations Ii and Ij respectively and Q be a data item in the database. If Ii and Ij refer to different data items the order of them is a schedule doesn't matter. We can swap Ii and Ij without affecting the results of any operation of the schedule. If Ii and Ij refer to the same data item Q then the order of them may matter.

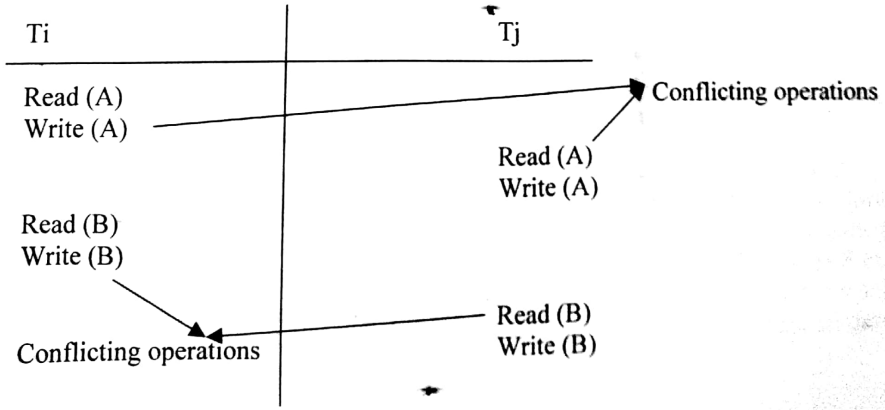
There are 4 cases to consider:

Ii = read (Q), Ij = read (Q) the order of Ii and Ij doesn't matter.
Ii = read (Q) , Ij = write (Q) the order of Ii and Ij matters
Ii = write (Q), Ij = read (Q) the order of Ii and Ij matters
Ii = write (Q), Ij = write (Q) the order of Ii and Ij matters

Two operations Ii and Ij are in conflict if they are operations by different transactions on the same item and at least one of them is a write. We can swap the order of adjacent operations in a schedule if they do not conflict with each other.

If a schedule S can be transformed to a schedule S' by a series of swaps of non-conflicting operation, S and S, are said to be conflict equivalent.

Schedule 3 (Considering only read and write operations)



Scanned by CamScanner
Scanned by CamScanner

Schedule 4 (After swapping non conflicting operation)

Ti	Tj
Read (A)	
Write (A)	
Read (B)	Read (A)
Write (B)	Write (A)
	Read (B)
	Write (B)

A schedule is Conflict Serializable if its conflict equivalent to a serial schedule.

E.g.: 1) After doing the following swapping of non conflicting operations in schedules3, the final result will be serial schedule 1

Swap Write (A) of Tj with Read (B) of Ti (We get schedule4)
Swap Read (B) of Ti with Read (A) of Tj.
Swap Write (B) of Ti with Write (A) of Tj
Swap Write (B) of Ti with Read (A) o Tj

The result is, schedule 1 (serial schedule)

Ti	Tj
Read (A)	
Write (A)	
Read(B)	
Write (B)	
	Read(A)
	Write (A)
	Read(B)
	Write (B)

Schedule 3 is conflict equivalent to schedule 1 which is a serial schedule; therefore schedule 3 is conflict serializable.

Testing serializability

(V, E)

Precedence Graphs

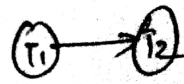
T_i – vertices, V

$T_i \rightarrow T_j$ – Edges, E

There is edge $T_i \rightarrow T_j$ only if either of following:

1. T_i executes read(Q) before T_j executes write(Q)
2. T_i " write(Q) " T_j " read(Q)
3. T_i " write(Q) " T_j " write(Q)

No cycle in the Graph - schedule is conflict serializable

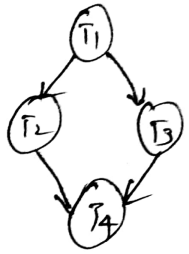


Cycle – not conflict serializable

Scanned by CamScanner

Scanned by CamScanner

Serializability Order can be obtained through topological sorting



<T1, T2, T3, T4> or

<T1, T3, T2, T4> .

E.g. 2) It is possible to have two schedules that produce the same outcome but that are not conflict equivalent.

Ti	Tj
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)

The above schedule is not conflict equivalent to either of the serial schedules <Ti, Tj> of <Tj, Ti> .But the final values of accounts A and B after the execution of either of them are the same. We can see form this example that there are less stringent definition of schedule equivalence than there are of conflict equivalence.

View Serializability

Consider two schedules S and S' where the same set of transactions participates in both schedules. S and S' are said to be view equivalent if the following three conditions are met.

1. For each data item Q, if Ti reads the initial value of Q in S then Ti, must, in S' also, reads the initial value of Q.
2. For each data item Q, if Ti executes Read (Q) in S and that value was produced by Tj (if any) then Ti must, in S', also read the value of Q that was produced b Tj.
3. For each data item Q, the transaction (if any) that perform the final Write (Q operation in S must perform the final write (Q) operation in S'.

E.g. Schedule 1 is not view equivalent to schedule 2
Schedule 1 is view equivalent to schedule 3

Scanned by CamScanner
Scanned by CamScanner

but that are

schedule S is **View Serailizable** if it is view equivalent to a serial schedule.

Eg. Schedule 3 is View Serializable since it its view equivalent to schedule which is a serial schedule.

Every conflict. Serializable schedule is View Serializable, but there are View Serializable schedules that are not Conflict Serializable.

Eg. 1) schedule 3is view serializable as well as conflict serializable.

2)

Ti	Tj	Tk
Read (Q)	Write (Q)	
Write (Q)		
		Write (Q)

This schedule is view serializable. It is view equivalent to serial schedule <Ti, Tj, Tk> But it is not conflict serializable.

Recoverability

Ti	Tj
read(A)	
write(A)	
	read (A)
write(A)	

If the schedule S consisting Ti and Tj such that Tj is reading some value produced by Ti, the schedule is recoverable only if Ti commits before Tj.

Cascadeless Rollback

Ti	Tj	Tk
read(A)		
write(A)		
// commit	read(A)	
	write(A)	
	//commit	
		read (A)

A schedule S consisting of Ti, Tj. For ensuring cascadeless rollback Ti should be committed before Tj read operation.

∴ Cascadeless Rollback schedules are Recoverable Schedules

Scanned by CamScanner

Scanned by CamScanner

CONCURRENCY CONTROL TECHNIQUES

There are two major types of concurrency control protocols:

- Lock- based protocols
- Timestamp – based protocols

1. Lock –based protocols

To ensure serializability and thus avoid violation of the isolation property, we should control access data items in a mutually exclusive manner. The idea is to allow transaction to access a data item only if it is holding a lock on it. Locks specify access mode of a data item. Two important kinds of locks are **shared and exclusive**.

Shared – If a transaction Ti has obtained a shared mode lock (denoted as S)on a data item Q, then Ti can read it but not write to it.

Exclusive - If a transaction Ti has obtained a exclusive –mode lock (denoted as X) on a data item Q, then Ti, can both read it and write to it.

Lock	Read	Write
None	No	No
Shared	Yes	No
Exclusive	Yes	Yes

Many transactions can hold shred locks on a data item. If a transaction holds an exclusive lock on a data item no other transaction can hold any lock on it.

Lock – Compatibility

Let A and B represent arbitrary lock modes. Suppose a transaction Ti requests a lock of mode A on item Q on which transaction Tj (Ti ≠ Tj) currently holds a lock of mode B. If transaction Ti can be granted a lock on Q immediately, in spite of presence of the mode B lock, then we say mode A is compatible with mode B.

Lock – compatibility matrix (between shared and exclusive locks) comp.

	S	X
S	True	False
X	False	false

An element comp (A, B) of the matrix has the value true if and only if **mode A is** compatible with mode B

Scanned by CamScanner

Two Phase Locking Protocol

Each transaction issue lock and unlock requests in two phases.

1. **Growing phase** – a transaction may obtain locks, but may not release any lock.
2. **Shrinking phase** – A transaction may release locks, but may not obtain any new locks.

Eg

	T1:Lock X(B);		T2: lock S(A)
	Read (B);		Read (A);
	B=B-50;	Lock point ⟶	Lock S(B);
	Write(B)		
Lock point ⟶	Lock X(A);		Read(B);
	Read (A);		Display (A+B)
	A=A+50;		Unlock (A);
	Write (A)		Unlock (B);
	Unlock (B)		
	Unlock (A)		

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction.

We can show that the 2PL ensures conflict serializability. Transactions can be ordered according to their lock points – this ordering is in fact a serializably ordering for the transactions.

Note: (1) 2PL doesn't ensure freedom from deadlock.
(2) cascading roll back may occur under 2PL.

Variations of 2PL

Strict 2PL- The strict 2PL requires, that in addition to locking being 2ϕ all exclusive-mode locks taken by a transaction must be held until that transactions commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the traction commits, preventing any other transaction from reading the data. (Thus we can avoid cascading rollback)

Rigorous 2PL- is 2PL with the additional requirements that all locks be held until the transaction commits. It can be easily verified that with rigorous 2PL, transactions can be serialized in the order in which they commit. Most D BMS implement either strict or rigorous two-phase locking.

Scanned by CamScanner

2PL

Example - Deadlock

lock-X(A)	
read(A)	
A=A-50	
write(A)	
	lock-X(B)
	read(B)
	B=B+50
	lock-X(A)
lock-X(B)	

Example - Cascading Rollback

Ti	Tj	Tk
lock-X(A)		
read(A)		
A=A-50		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	A=A+50	
	write(A)	
	unlock(A)	

Failure →		lock-S(A)
		read(A)
		unlock(A)

Solution - 2PL

UPGRADATION

To avoid starvation -

Ti	Tj
eg:- lock-X(A1)	
read(A1)	
lock-S(A2)	
read(A2)	
lock-S(A3)	
read(A3)	
write(A1)	
unlock(A1)	
	lock-S(A1) → starvation
	read(A1)
	unlock(A1)
unlock(A2)	
unlock(A3)	

Protocol

1. If Ti issues read(Q), system issues lock-S(Q) followed by read(Q)
2. when Ti issues write(Q), system issues upgrade(Q) if Ti is currently holding a shared lock on Q. (This upgrade(Q) will wait if any other transaction Tj currently holding a shared lock on Q)
3. else if Ti not currently holding a shared lock on Q, sys issues lock-X(Q)
4. Release locks during commit operation where downgrading also possible

eg:-

Ti	Tj
lock-S(A1)	
read(A1)	lock-S(A1)
------	read(A1)
read(A3)	unlock(A1)
upgrade(A1)	

Scanned by CamScanner
Scanned by CamScanner

2. Timestamp-Based Protocols

In time stamp based protocols the transactions are ordered in advance using timestamps. With each transaction Ti in the system, we associate a unique fixed timestamp denoted by t(Ti). The timestamp is assigned by the system before Ti starts execution. If a transaction Ti has been assigned timestamp t(Ti), and a new transactions Tj enters the system , then t(Ti) <t(Tj). Each data item we associate two time stamp values.

$t_W(Q)$- denotes the largest timestamp of any transaction that executed Write(Q) successfully.

$t_R(Q)$ – denotes the largest timestamp of any transaction that executed Read(Q) successfully.

These two timestamps are updated whenever a new Read (Q) or Write (Q) instruction is executed.

The timestamps of the transactions determine the serializability order. Thus if t(Ti)<t (Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which Ti appear before Tj.

Timestamp-ordering Protocol

The timestamp ordering protocol ensures that any conflicting Read and Write operations are enclosed in timestamp order.

The protocol operates as follows:

1. Suppose Ti issues Read (Q)
   a. If t(Ti)< $t_W(Q)$,then Ti needs to read a value of Q that was already overwritten. Hence the read operation is rejected and Ti is rolled back.
   b. If t(Ti) ≥ t_W (Q) then the read operation is ~~expected~~ accepted and tR (Q) is set to the max of $t_R(Q)$ and t(Ti)
2. Suppose Ti issues Write (Q)
   a. t(Ti)< $t_R(Q)$ then the value of Q that Ti is producing was needed previously and the system assumed that the value would never be produced. Hence the write operation is rejected and Ti is rolled back.

   b. If t(Ti) < t_W (Q) then Ti is attempting to write an ~~absolute~~ obsolete value of Q. Hence the write operation is rejected and Ti is rolled back.
   c. Other wise, the write operations is executed and tw(Q) is set to t(Ti)

The time stamp ordering protocol ensures conflict serializability. This assertion follows from the fact that conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, since no transaction ever waits.

e.g. T1: Read (A), Write (A), Read (B), Write (B)

T2: Read (B), Write (A), Read (A), Write (A)

Suppose t(T1)<t(T2) and all t0,t1, t2 and t3 are smaller than t (T1)

Schedule 1

T1	T2	t_R(A)	t_W (A)	t_R(B)	t_W (B)
		t0	t1	t2	t3
Read (A)		t(T1)			
Write (A)			t(T1)		
Read (B)				t(T1)	
Write (B)					t(T1)
	Read (B)			t(T2)	
	Write (B)				t(T2)
	Read (A)	t(T2)			
	Write (A)		t(T2)		

Schedule 2

T1	T2	t_R(A)	t_W (A)	t_R(B)	t_W (B)
		t0	t1	t2	t3
Read (A)		t(T1)			
Write (A)			t(T1)		
	Read (B)			t(T2)	
	Write(B)				t(T2)
	Read(A)	t(T2)			
	Write(A)		t(T2)		
Read(B) → abort() Time see t(T1) < tw(B), Ti is rolled back					

Scanned by CamScanner

Scanned by CamScanner

Thomas' Write Rule

This is a modified version of timestamp –ordering protocol. The protocol rules for read operations remain unchanged. The protocol rules for write operations are slightly different from eh timestamp ordering protocol.

Suppose Ti issues Write (Q)

1. If $t(Ti) < t_R(Q)$, then the value of Q that Ti is producing was previously needed, and it was assumed that the value would never be produced. Hence, the write operation is rejected and Ti is rolled back.

2. If $t(Ti) < t_W(Q)$, then Ti is attempting to write an obsolete value of Q. Hence this write operation can be ignored.

3. Otherwise, the write operation is executed and $t_W(Q)$ is set to t(Ti)

eg:

		Timestamp ordering protocol		Thomas write Rule	
T1	T2	$t_R(Q)$	$t_W(Q)$	$t_R(Q)$	$t_W(Q)$
Read (Q)		t(T1)	-	t(T1)	-
	Write (Q)	t(T1)	t(T2)	t(T1)	t(T2)
Write (Q)	→ Abort(), Ti is rolled back			t(T1)	t(T2)

The roll back of Ti is required by time stamp ordering protocol. This is unnecessary because T2 has already written Q, the value that T1 is attempting to write is one that will never need to be read.

Any transaction Ti with $t(Ti) \leq t(T2)$ that attempts a Read (Q) will be rolled back since $t(Ti) < tw(Q)$.

Any transaction Tj with $t(tj) > t(T2)$ must read the value of Q written by T2, rather that by T1.

Advatage. Ensures view serializability. The schedule in the above example is not conflict serializable and this is not possible under any of 2ϕ locking, tree protocol or the timestamp ordering protocol. Under Thomas write rule the write (Q) operation of T1 would be ignored The result is a schedule that is view equivalent to the serial schedule <T1,T2>.

Scanned by CamScanner