UNIT-IV (PART-1)

TRANSACTIONS

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

EX: transaction to transfer $50 from account A to account B:

read(*A*)
A := *A* − 50
write(*A*)
read(*B*)
B := *B* + 50
write(*B)*

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Atomicity requirement — If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

- Failure could be due to software or hardware

✓ System should ensure that updates of a partially executed transaction are not reflected in database

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction. In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints

– EX: sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

	T1	**T2**
1.	**read**(*A*)	
2.	*A* := *A* − 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*	

Isolation can be ensured trivially by running transactions **serially**

-that is, one after the other.

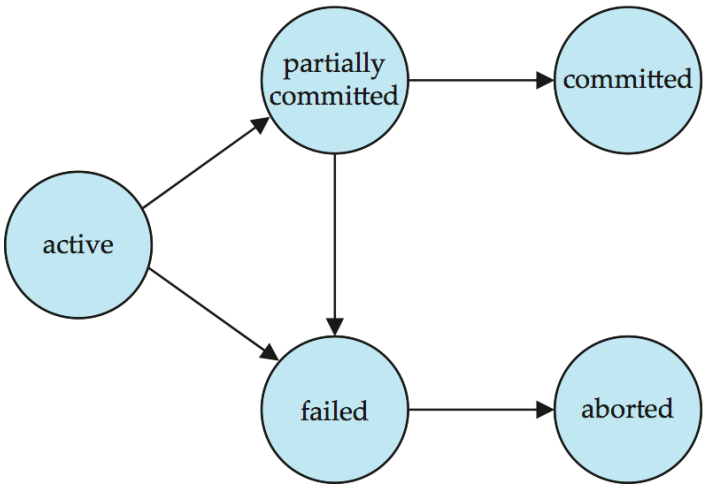
However, executing multiple transactions concurrently has significant benefits.

ACID Properties: A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

1. **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
2. **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
3. **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    a. That is, 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.
4. **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State:

1. **Active** – the initial state; the transaction stays in this state while it is executing
2. **Partially committed** – after the final statement has been executed.
3. **Failed** -- after the discovery that normal execution can no longer proceed.
4. **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    a. restart the transaction
        i. can be done only if no internal logical error
    b. kill the transaction
5. **Committed** – after successful completion.



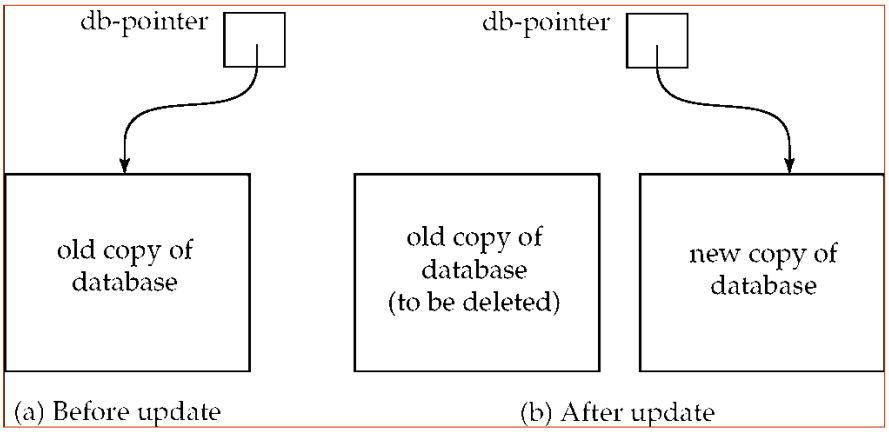
Implementation of Atomicity and Durability: The **recovery-management** component of a database system implements the support for atomicity and durability.

EX: the ***shadow-database*** scheme:

-all updates are made on a *shadow copy* of the database

db_pointer is made to point to the updated shadow copy after

- the transaction reaches partial commit and
- all updated pages have been flushed to disk.



db_pointer always points to the current consistent copy of the database.

-- In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

The shadow-database scheme:

- Assumes that only one transaction is active at a time.
- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases
    - Variant called shadow paging reduces copying of data, but is still not practical for large databases
- Does not handle concurrent transactions

Concurrent Executions: Multiple transactions are allowed to run concurrently in the system. Advantages are:

- increased processor and disk utilization, leading to better transaction *throughput*

- EX: one transaction can be using the CPU while another is reading from or writing to the disk

- reduced average response time for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – These are the mechanisms to achieve isolation. That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules: A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed. 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.

A transaction that successfully completes its execution will have commit instructions as the last statement. by default transaction assumed to execute commit instruction as its last step. A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Schedule 1: Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B.*

A serial schedule in which *T*1 is followed by *T*2:

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

Schedule 2:

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

Schedule 3:

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Let *T*1 and *T*2 be the transactions defined previously. The above schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

Note: In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4: The following concurrent schedule does not preserve the value of (*A* + *B)*.

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**
2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

Instructions *li* and *lj* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q.*

1. *li* = **read**(*Q*), *lj* = **read**(*Q*). *li* and *lj* don't conflict.
2. *li* = **read**(*Q*), *lj* = **write**(*Q*). They conflict.
3. *li* = **write**(*Q*), *lj* = **read**(*Q*). They conflict
4. *li* = **write**(*Q*), *lj* = **write**(*Q*). They conflict

Intuitively, a conflict between *li* and *lj* forces a (logical) temporal order between them. If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability: If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.

We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule. Schedule 3 can be transformed into Schedule 6, a serial schedule where *T*2 follows *T*1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable

Schedule 3

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 6

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < *T*3, *T*4 >, or the serial schedule < *T*4, *T*3 >.

View Serializability

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*

1. If in schedule S, transaction *Ti* reads the initial value of *Q*, then in schedule *S´* also transaction *Ti* must read the initial value of *Q.*
2. If in schedule S transaction *Ti* executes **read**(*Q*), and that value was produced by transaction *Tj* (if any), then in schedule *S´* also transaction *Ti* must read the value of *Q* that was produced by the same **write**(Q) operation of transaction *Tj* .
3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S´.*

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.
A schedule *S* is **view serializable** if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable. Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

The schedule below produces same outcome as the serial schedule < $T1$, $T5$ >, yet is not conflict equivalent or view equivalent to it.

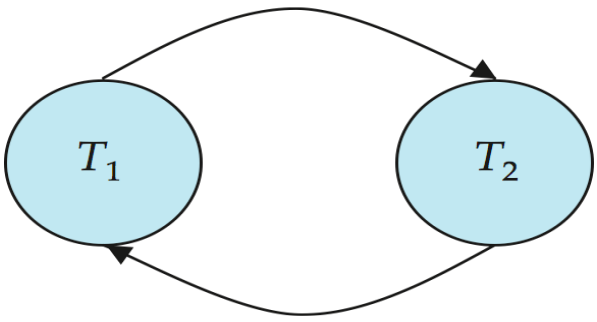
T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

Determining such equivalence requires analysis of operations other than read and write.

Testing for Serializability

- Consider some schedule of a set of transactions $T1$, $T2$, ..., Tn
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from Ti to Tj if the two transaction conflict and Ti accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Example



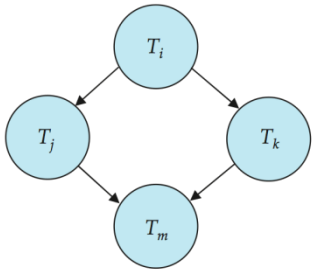
Example Schedule (Schedule A) + Precedence Graph

T1	*T2*	*T3*	*T4*	*T5*
read(Y) read(Z) read(U) read(U) write(U)	read(X) read(Y) write(Y)	 write(Z)	 read(Y) write(Y) read(Z) write(Z)	 read(V) read(W) read(W)

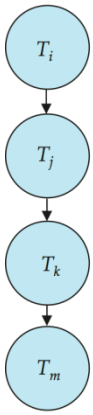
Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic.

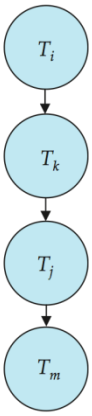
- Cycle-detection algorithms exist which take order $n2$ time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a Serializability order for Schedule A would be $T5 \rightarrow T1 \rightarrow T3 \rightarrow T2 \rightarrow T4$



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict Serializability cannot be used directly to test for view Serializability.
  - Extension to test for view Serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view Serializability can still be used.

Recoverable Schedules: Need to address the effect of transaction failures on concurrently running transactions.

Recoverable schedule — if a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti* appears before the commit operation of *Tj.*

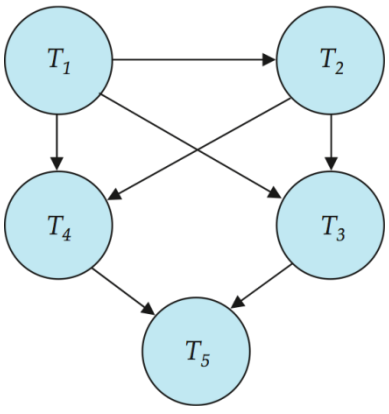
The following schedule (Schedule 11) is not recoverable if *T9* commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

If *T*8 should abort, *T*9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks: A single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		



If *T*10 fails, *T*11 and *T*12 must also be rolled back.

✓ Can lead to the undoing of a significant amount of work.

Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj*.

Every cascadeless schedule is also recoverable. It is desirable to restrict the schedules to those that are cascadeless.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for Serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for Serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- EX: a read-only transaction that wants to get an approximate total balance of all accounts
- EX: database statistics computed for query optimization can be approximate

Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lower degrees of consistency useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default

EX: Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL: In SQL, a transaction begins implicitly

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - EX: in JDBC, connection. setAutoCommit(false);