UNIT-IV

TRANSACTION MANAGEMENT

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes.

Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

The Four Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties. Taking the initial letters of these four properties we collectively call them the **ACID Properties**. Here we try to describe them and explain them.

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

```
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;
```

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs 1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred.

Consistency: If we execute a particular transaction in isolation or together with other transaction, (i.e. presumably in a multi-programming environment), the transaction will yield the same expected result.

To give better performance, every database management system supports the execution of multiple transactions at the same time, using CPU Time Sharing. Concurrently executing transactions may have to deal with the problem of sharable resources, i.e. resources that multiple transactions are trying to read/write at the same time. For example, we may have a table or a record on which two transaction are trying to read or write at the same time. Careful mechanisms are created in order to prevent mismanagement of these sharable resources, so that there should not be any change in the way a transaction performs. A transaction which deposits Rs 100/- to account A must deposit the same amount whether it is acting alone or in conjunction with another transaction that may be trying to deposit or withdraw some amount at the same time.

Isolation: In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

Durability: It states that once a transaction has been complete the changes it has made should be permanent.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

Transaction States

There are the following six states in which a transaction may exist:

Active: The initial state when the transaction has just started execution.

Partially Committed: At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.

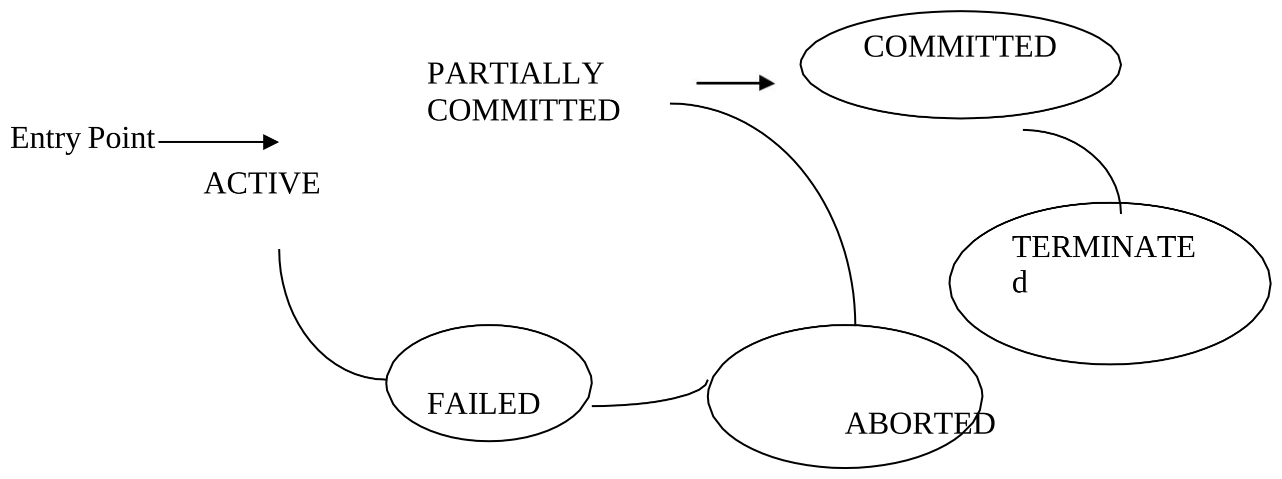
Failed: If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.

Aborted: When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.

Committed: If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.

Terminated: Either committed or aborted, the transaction finally reaches this state.

The whole process can be described using the following diagram:



Concurrent Execution

A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- **Serial:** The transactions are executed one after another, in a non-preemptive manner.
- **Concurrent:** The transactions are executed in a preemptive, time sharedmethod.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.

In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

```
    T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

    T2
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
```

```
Write C;
```

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

```
    T1                T2

Read A;
A = A – 100;
Write A;
                  Read A;
                  Temp = A * 0.1;
                  Read C;
                  C = C + Temp;
                  Write C;
Read B;
B = B + 100;
Write B;
```

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

1	T2
Read A;	
A = A – 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Rs 1000/- each, then the result of this schedule should have left Rs 900/- in A, Rs 1100/- in B and add Rs 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Rs 900/- has been updated in A. T2 reads the old value of A, which is still Rs 1000/-, and deposits Rs 100/- in C. C makes an unjust gain of Rs 10/- out of nowhere.

Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated

by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules

S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Let us consider a schedule *S* in which there are two consecutive instructions, *I* and *J* , of transactions *Ti* and *Tj* , respectively (*i* _= *j*). If *I* and *J* refer to different data items, then we can swap *I* and *J* without affecting the results of any instruction in the schedule. However, if *I* and *J* 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 = read(*Q*), *J* = read(*Q*). The order of *I* and *J* does not matter, since the same value of *Q* is read by *Ti* and *Tj* , regardless of the order.

 I = read(*Q*), *J* = write(*Q*). If *I* comes before *J* , then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *J* . If *J* comes before *I*, then *Ti* reads the value of *Q* that is written by *Tj*. Thus, the order of *I* and *J* matters.

 I = write(*Q*), *J* = read(*Q*). The order of *I* and *J* matters for reasons similar to those of the previous case.

4. *I* = write(*Q*), *J* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either *Ti* or *Tj* . 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 *I* and *J* in *S*, then the order of *I* and *J* directly affects the final value of *Q* in the database state that results from schedule *S*.

Fig: Schedule 3—showing only the read and write instructions.

We say that *I* and *J* **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3in Figure above. The write(*A*) instruction of *T*1 conflicts with the read(*A*) instruction of *T*2. However, the write(*A*) instruction of *T*2 does not conflict with the read(*B*) instruction of *T*1, because the two instructions access different data items.

Transaction Characteristics

Every transaction has three characteristics: *access mode*, *diagnostics size,* and *isolation level*. The **diagnostics size** determines the number of error conditions that can be recorded.

If the **access mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure given below. In this context, *dirty read* and *unrepeatable read* are defined as usual. **Phantom** is defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself.

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 19.3.1), and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that *T* reads only the changes made by committed transactions, and that no value read or written by *T* is changed by any other transaction until *T* is complete. However, *T* could experience the phantom phenomenon; for example, while *T* examines all

Sailors records with *rating=1*, another transaction might add a new such Sailors record, which is missed by *T*.

A REPEATABLE READ transaction uses the same locking protocol as a SERIALIZABLE transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that *T* reads only the changes made by committed transactions, and that no value written by *T* is changed by any other transaction until *T* is complete. However, a value read by *T* may well be modified by another transaction while *T* is still in progress, and *T* is, of course, exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself - a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects, and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.
For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level, or even the READ UNCOMMITTED level, because a few incorrect or missing values will not significantly affect the result if the number of sailors is large. The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

PRECEDENCE GRAPH

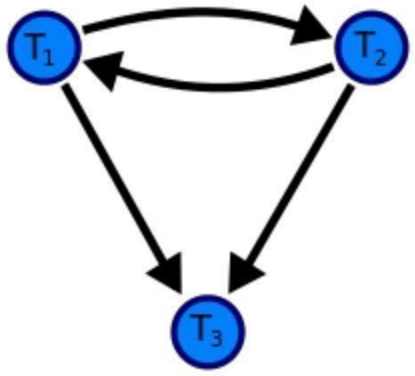
A precedence graph, also named conflict graph and serializability graph, is used in the context of concurrency control in databases.

The precedence graph for a schedule S contains:

A node for each committed transaction in S
An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.

Precedence graph example

$$D = \begin{bmatrix} T1 & T2 & T3 \\ & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \end{bmatrix}$$



A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is not Conflict serializable.

The drawing sequence for the precedence graph:-

 For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph. So the precedence graph contains T_1, T_2, T_3
 For each case in S where T_i executes a write_item(X) then T_j executes a read_item(X), create an edge (T_i --> T_j) in the precedence graph. This occurs nowhere in the above example, as there is no read after write.

3. For each case in S where T_i executes a read_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edge from T_1 to T_2.
4. For each case in S where T_i executes a write_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edges from T_2 to T_1, T_1 to T_3, and T_2 to T_3.
5. The schedule S is conflict serializable if the precedence graph has no cycles. As T_1 and T_2 constitute a cycle, then we cannot declare S as serializable or not and serializability has to be checked using other methods.

TESTING FOR CONFLICT SERIALIZABILITY

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

2 To test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm.Cycle-detection algorithms exist which takeorder 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.)

3 If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. That is, a linear order consistent with the partial order of the graph.

For example, a serializability order for the schedule (a) would be one of either (b) or (c)

4 A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph.

RECOVERABLE SCHEDULES

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

The following schedule is not recoverable if T9 commits immediately after the read(A) operation.

If T8 should abort, T9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

CASCADING ROLLBACKS

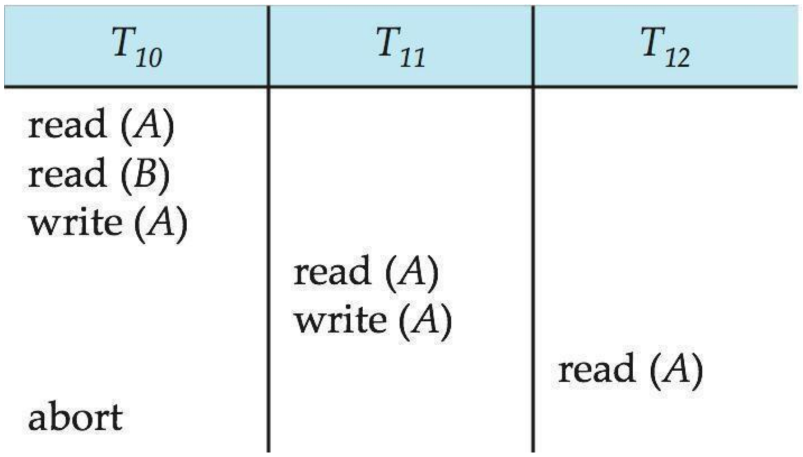
 Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

If T10 fails, T11 and T12 must also be rolled back.

- Can lead to the undoing of a significant amount of work

CASCADELESS SCHEDULES

- Cascadeless schedules — 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
- Example of a schedule that is NOT cascadeless



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

CONCURRENCY SCHEDULE

- A database must provide a mechanism that will ensure that all possible schedules are both:
- Conflict serializable.
- 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

- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur

- Testing a schedule for serializability after it has executed is a little too late!

- Tests for serializability help us understand why a concurrency control protocol is correct

- Goal – to develop concurrency control protocols that will assure serializability.

WEEK LEVELS OF CONSISTENCY

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

- E.g., a read-only transaction that wants to get an approximate total balance of all accounts

- E.g., database statistics computed for query optimization can be approximate (why?)

- Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance

LEVELS OF CONSISTENCY IN SQL

- 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

- E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

TRANSACTION DEFINITION IN SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.

- In SQL, a transaction begins implicitly.
- 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
- E.g. in JDBC, connection.setAutoCommit(false);

RECOVERY SYSTEM

Failure Classification:

- Transaction failure :
- Logical errors: transaction cannot complete due to some internal error condition
- System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash: a power failure or other hardware or software failure causes the system to crash.
- Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted as result of a system crash
- Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure: a head crash or similar disk failure destroys all or part of disk storage
- Destruction is assumed to be detectable: disk drives use checksums to detect failures

RECOVERY ALGORITHMS

- Consider transaction Ti that transfers $50 from account A to account B
- Two updates: subtract 50 from A and add 50 to B

- Transaction Ti requires updates to A and B to be output to the database.

- A failure may occur after one of these modifications have been made but before both of them are made.

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state

- Not modifying the database may result in lost updates if failure occurs just after transaction commits

- Recovery algorithms have two parts

1. Actions taken during normal transaction processing to ensure enoughinformation exists to recover from failures

2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

STORAGE STRUCTURE

- Volatile storage:

- does not survive system crashes

- examples: main memory, cache memory

- Nonvolatile storage:

- survives system crashes

- examples: disk, tape, flash memory,

non-volatile (battery backed up) RAM

- but may still fail, losing data

- Stable storage:

- a mythical form of storage that survives all failures

- approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
- copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies.

Block transfer can result in

- Successful completion
- Partial failure: destination block has incorrect information
- Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
- Execute output operation as follows (assuming two copies of each block):

1. Write the information onto the first physical block.
2. When the first write successfully completes, write the same information onto the second physical block.
3. The output is completed only after the second write successfully completes.

- Copies of a block may differ due to failure during output operation. To recover from failure:

1. First find inconsistent blocks:

1. Expensive solution: Compare the two copies of every disk block.
2. Better solution:

- Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
- Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
- Used in hardware RAID systems

2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

DATA ACCESS

- Physical blocks are those blocks residing on the disk.
- System buffer blocks are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
- input(B) transfers the physical block B to main memory.
- output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Each transaction Ti has its private work-area in which local copies of all data items accessed and updated by it are kept.
- Ti's local copy of a data item X is denoted by xi.
- BX denotes block containing X
- Transferring data items between system buffer blocks and its private work-area done by:
- read(X) assigns the value of data item X to the local variable xi.
- write(X) assigns the value of local variable xi to data item {X} in the buffer block.
- Transactions
- Must perform read(X) before accessing X for the first time (subsequent reads can be from local copy)
- The write(X) can be executed at any time before the transaction commits
- Note that output(BX) need not immediately follow write(X). System can performthe output operation when it seems fit.

Lock-Based Protocols

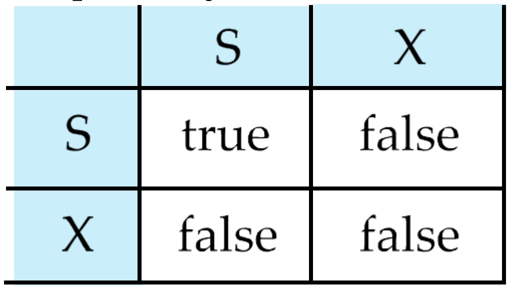
A lock is a mechanism to control concurrent access to a data item

Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as wellas written. X-lock is requested using lock-X instruction.
2. shared (S) mode. Data item can only be read. S-lockis requested using lock-S instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix



	S	X
S	true	false
X	false	false

1) A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

2) Any number of transactions can hold shared locks on an item,
but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

3) If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

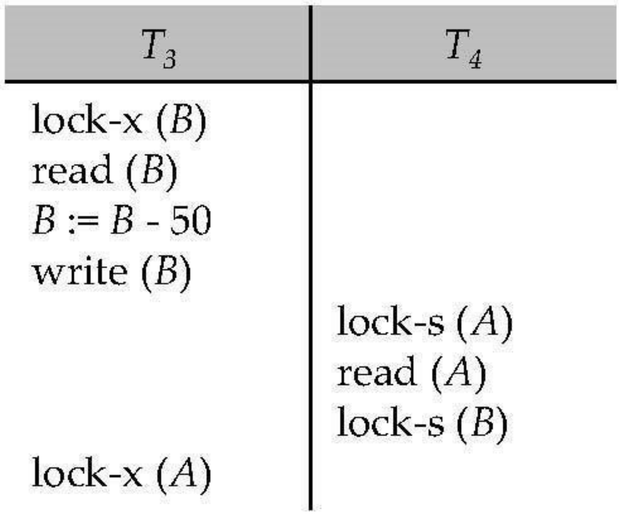
Example of a transaction performing locking:

T_2: **lock-S**(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display$(A+B)$

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Consider the partial schedule



T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.

Such a situation is called a **deadlock**.

l To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

2. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
3. **Starvation** is also possible if concurrency control manager is badly designed. For example:
   a. A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
   b. The same transaction is repeatedly rolled back due to deadlocks.
4. Concurrency control manager can be designed to prevent starvation.

THE TWO-PHASE LOCKING PROTOCOL

1. This is a protocol which ensures conflict-serializable schedules.
2. Phase 1: Growing Phase
   a.transaction may obtain locks
   b.transaction may not release locks
3. Phase 2: Shrinking Phase
   a.transaction may release locks
   b.transaction may not obtain locks
4. The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

5. Two-phase locking *does not* ensure freedom from deadlocks

6. Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all itsexclusive locks till it commits/aborts.
7. **Rigorous two-phase locking** is even stricter: here *all* locks are held tillcommit/abort.In this protocol transactions can be serialized in the order in which theycommit.
8. There can be conflict serializable schedules that cannot be obtained if two-phaselocking is used.
9. However, in the absence of extra information (e.g., ordering of access to data),two-phase locking is needed for conflict serializability in the following sense:

Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflictserializable.

TIMESTAMP-BASED PROTOCOLS

1. Each transaction is issued a timestamp when it enters the system. If an old transactionT_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
2. The protocol manages concurrent execution such that the time-stamps determinetheserializability order.
3. In order to assure such behavior, the protocol maintains for each data Q twotimestampvalues:
   a. W-timestamp(Q) is the largest time-stamp of any transaction that executedwrite(Q) successfully.
   b. R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.

4. The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.5.Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a valueof Q that wasalready overwritten.
   n Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

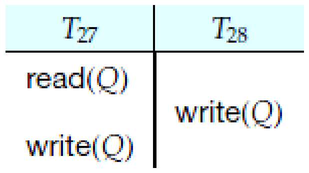
6. Suppose that transaction T_i issues **write**(Q).
   1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was neededpreviously, and the system assumed that that value would never be produced.
      n Hence, the **write** operation is rejected, and T_i is rolled back.
   2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.nHence, this **write** operation is rejected, and T_i is rolled back.
   3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Thomas' Write Rule

1. We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol i.e., Timestamp ordering Protocol . Let us consider schedule 4 of Figure below, and apply the timestamp-ordering protocol. Since *T*27 starts before *T*28, we shall assume that TS(*T*27) < TS(*T*28). The read(*Q*) operation of *T*27 succeeds, as does the write(*Q*) operation of *T*28. When *T*27 attempts its write(*Q*) operation, we find that TS(*T*27) < W-timestamp(*Q*), since Wtimestamp(*Q*) = TS(*T*28). Thus, the write(*Q*) by *T*27 is rejected and transaction *T*27 must be rolled back.

2. Although the rollback of *T*27 is required by the timestamp-ordering protocol, it is unnecessary. Since *T*28 has already written *Q*, the value that *T*27 is attempting to write is one that will never need to be read. Any transaction *Ti* with TS(*Ti*) < TS(*T*28) that attempts a read(*Q*)will be rolled back, since TS(*Ti*)<W-timestamp(*Q*).

3. Any transaction *Tj* with TS(*Tj*) > TS(*T*28) must read the value of *Q* written by *T*28, rather than the value that *T*27 is attempting to write. This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol.



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

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction *Ti* issues write(*Q*).

1. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.

2. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence,this write operation can be ignored.

3. Otherwise, the system executes the write operation and setsW-timestamp(*Q*) toTS(*Ti*).

VALIDATION-BASED PROTOCOLS

Phases in Validation-Based Protocols

1) Read phase. During this phase, the system executes transaction Ti. It readsthevalues of the various data items and stores them in variables local to Ti. It performs all writeoperations on temporary local variables, without updates of the actualdatabase.

2) Validation phase. The validation test is applied to transaction Ti. This determineswhether Ti is allowed to proceed to the write phase without causing a violation ofserializability.

If a transaction fails the validation test, the system aborts the transaction.

3) Write phase. If the validation test succeeds for transaction Ti, the temporary local variables that hold the results of any write operations performed by Ti are copied to the database.Read-only transactions omit this phase.

MODES IN VALIDATION-BASED PROTOCOLS

1. Start(Ti)
2. Validation(Ti)
3. Finish

MULTIPLE GRANULARITY.

multiple granularity locking (MGL) is a locking method used in database management systems (DBMS) and relational databases.

In MGL, locks are set on objects that contain other objects. MGL exploits the hierarchicalnature of the contains relationship. For example, a database may have files, which contain pages,which further contain records. This can be thought of as a tree of objects, where each node contains its children. A lock on such as a shared or exclusive lock locks the targeted node as wellas all of its descendants.

Multiple granularity locking is usually used with non-strict two-phase locking to guarantee serializability. The **multiple-granularity locking protocol** uses these lockmodesto ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

- Transaction *Ti* must observe the lock-compatibility function of Figure above.
- Transaction *Ti* must lock the root of the tree first, and can lock it in anymode.
- Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q*locked in either IX or IS mode.
- Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
- Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (thatis, *Ti* is two phase).
- Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.