Unit-3

CONCURRENCY CONTROL TECHNIQUES

Some of the main techniques used to control the concurrent execution of transaction are based on the concept of locking the data items

- **Lock**
  - A lock is a variable associated with a data item that describe the statues of the item with respect to possible operations that can be applied to it.

Types of lock

- Binary lock
- Read/write(shared / Exclusive) lock

- **Binary lock**
  - It can have two states (or) values 0 and 1.
    0 – unlocked
    1 - locked
  - Lock value is 0 then the data item can accessed when requested.
  - When the lock value is 1,the item cannot be accessed when requested.

Lock_item(x)

```
B : if lock(x) = 0 ( * item is unlocked * )
then lock(x)      1
                  else begin
        wait ( until lock(x) = 0 )
        goto  B;
end;
```

Unlock_item(x)

```
B : if lock(x)=1 ( * item is locked * )
    then lock(x)      0
    else
    printf (‘ already is unlocked ‘)
    goto  B;
    end;
```

Read / write(shared/exclusive) lock

Read_lock

- its also called shared-mode lock
- If a transaction T_i has obtain a shared-mode lock on item X, then T_i can read, but cannot write ,X.
  - Outer transactions are also allowed to read the data item but cannot write.

```
B : if lock(x) = “unlocked”  then
                  begin
                  lock(x)        “ read_locked”
                  no_of_read(x)       1
                  else if
                  lock(x) = “read_locked”
                  then
                  no_of_read(x)        no_of_read(x) +1
                  else begin
                  wait (until lock(x) = “unlocked”
                  goto  B;
                  end;
```

Write_lock

- It is also called exclusive-mode lock.
- If a transaction T_i has obtained an exclusive-mode lock on item X , then T_i can both read and write X.
- Outer transactions cannot allowed to read and write the data item.

```
B : if lock(x) = "unlocked" then
        begin
        lock(x)       "write_locked"
        else if
        lock(x) = "write_locked"
        wait ( until lock(x) = "unlocked" )
        else begin
        lock(x)="read_locked" then
        wait ( until lock(x) = "unlocked" )
    end;
```

Unlock(x)

```
        If lock(x) = "write_locked" then
        Begin
        Lock(x)      "unlocked"
        Else if
        lock(x) = "read_locked" then
        Begin
        No_of_read(x)        no_of_read(x) - 1
        If ( no_of_read(x) = 0 ) then
        Begin
        Lock(x)      "unlocked"
        end
```

TWO PHASE LOCKING PROTOCOL

This protocol requires that each transaction issue lock and unlock request in two phases

- Growing phase
- Shrinking phase

Growing phase

- During this phase new locks can be occurred but none can be released

- Shrinking phase
  - During which existing locks can be released and no new locks can be occurred

Types

- Strict two phase locking protocol
- Rigorous two phase locking protocol

Strict two phase locking protocol

- This protocol requires not only that locking be two phase, but also all exclusive locks taken by a transaction be held until that transaction commits.

Rigorous two phase locking protocol

- This protocol requires that all locks be held until all transaction commits.

Lock conversion

Upgrade

- Conversion from shared lock to exclusive lock.
- Take place in only the growing phase

Downgrade

- Conversion from exclusive lock to shared lock
- Take place in only the shrinking phase

TWO PHASE COMMIT PROTOCOL

-Two phase commit is important when the transaction is related with many resource managers.
-The system component called coordinator handles the COMMIT or ROLLBACK operation.
It has two phases
-Prepare
-Commit

- **Prepare**

  It instructs the entire resource manager to get ready to **go either way.** Then the resource manager now replies to the coordinator.
- **Commit**
  - The Coordinator receives reply from all the resource manager, if all reply was OK then it takes the decision **COMMIT.**
  - If it receives any reply as NOT OK then it takes the decision **ROLLBACK.**

SERIALIZABILITY

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**

CONFLICT SERIALIZABILITY

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

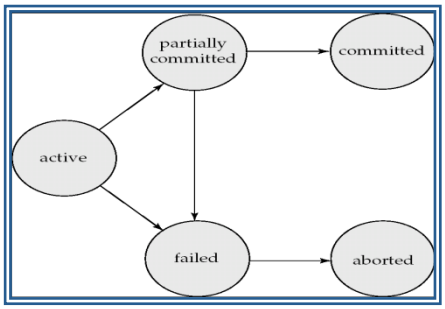
1. l_i = **read**(Q), l_j = **read**(Q).l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

- 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

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,
  - If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  - If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  - 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'.

TRANSACTION STATE AND PROPERTIES.



- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **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:
  - restart the transaction
  - kill the transaction
- **Committed** – after successful completion.

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

CONCURRENT EXECUTION IN TRANSACTION PROCESSING SYSTEM.

The transaction-processing system allows **concurrent execution** of multiple transactions to improve the system performance. In concurrent execution, the database management system controls the execution of two or more transactions in parallel; however, allows only one operation of any transaction to occur at any given time within the system. This is also known as **interleaved execution** of multiple transactions. The database system allows concurrent execution of transactions due to two reasons. First, a transaction performing read or writes operation using I/O devices may not be using the CPU at a particular point of time. Thus, while one transaction is performing I/O operations, the CPU can process another transaction. This is possible because CPU and I/O system in the computer system are capable of operating in parallel. This overlapping of I/O and CPU activities reduces the amount of time for which the disks and processors are idle and, thus, increases the **throughput** of the system (the number of transactions executed in a given amount of time).

CONCURRENCY CONTROL IS NEEDED

Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.

- lost updated problem
- Temporary updated problem
- Incorrect summery problem

Lost updated problem

T1	T2
Read(X)	
X:=X-N	
Write(X)	
Read(Y)	
Y:=Y+N	
Write(Y)	
	Read(X)
	X:=X+M
	Write(X)

Temporary updated problem

T1	T2
Read(X)	
X:=X-N	
Write(X)	
Rollback	
Read(Y)	
	Read(X)
	X:=X-M
	Write(X)

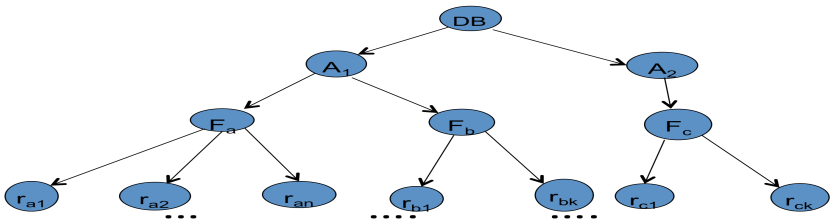
Incorrect Summery Problem

T1	T2
Read(B)	Sum:=0
B:=B-N	Read(A)
Write(B)	Sum:=sum+A
Read(X)	Read(B)
X:=X-N	Sum:=sum+B
Write(X)	.

Read(X)

Sum:=sum+X

LOCKING METHOD IN CONCURRENCY CONTROL



Intent locking

- Intent locks are put on all the ancestors of a node before that node is locked explicitly.
- If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree.

Intent shared lock (IS)

- If a node is locked in indent shared mode, explicit locking is being done at a lower level of the tree, but with only shared-mode lock
- Suppose the transaction T_1 reads record r_{a2} in file F_a. Then,T_1 needs to lock the database, area $A_{1,and}$ F_a in IS mode, and finally lock r_{a2} in S mode.

Intent exclusive lock (IX)

If a node is locked in intent locking is being done at a lower level of the tree, but with exclusive mode or shared-mode locks.

Suppose the transaction T_2 modifies record r_{a9} in file F_a. Then,T_2 needs to lock the database, area $A_{1,and}$ F_a in IX mode, and finally to lock r_{a9} in X mode

Shared Intent exclusive lock (SIX)

- If the node is locked in Shared Intent exclusive mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at lower level with exclusive mode.

Shared lock (S)

-T can tolerate concurrent readers but not concurrent updaters in R.

Exclusive lock (X)

-T cannot tolerate any concurrent access to R at all.

DEAD LOCK

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :

- Require that each transaction locks all its data items before it begins execution (predeclaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

WAIT-DIE SCHEME:

In this scheme, if a transaction request to lock a resource (data item), which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

- If $TS(T_i) < TS(T_j)$, that is T_i, which is requesting a conflicting lock, is older than T_j, T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$, that is T_i is younger than T_j, T_i dies. T_i is restarted later with random delay but with same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

WOUND-WAIT SCHEME:

In this scheme, if a transaction request to lock a resource (data item), which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

- If $TS(T_i) < TS(T_j)$, that is T_i, which is requesting a conflicting lock, is older than T_j, T_i forces T_j to be rolled back, that is T_i wounds T_j. T_j is restarted later with random delay but with same timestamp.
- If $TS(T_i) > TS(T_j)$, that is T_i is younger than T_j, T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait but when an older transaction request an item held by younger one, the older transaction forces the younger one to abort and release the item.

In both cases, transaction, which enters late in the system, is aborted.

DEAD LOCK RECOVERY ALGORITHM

The common solution is to roll back one or more transactions to break the deadlock.

Three action need to be taken

a. Selection of victim
b. Rollback
c. Starvation

Selection of victim

i. Set of deadlocked transations,must determine which transaction to roll back to break the deadlock.
ii. Consider the factor minimum cost

Rollback

- once we decided that a particular transaction must be rolled back, must determine how far this transaction should be rolled back

- Total rollback
- Partial rollback

Starvation

Ensure that a transaction can be picked as victim only a finite number of times.

DATABASE RECOVERY CONCEPTS.

Crash Recovery

Though we are living in highly technologically advanced era where hundreds of satellite monitor the earth and at every second billions of people are connected through information technology, failure is expected but not every time acceptable.

DBMS is highly complex system with hundreds of transactions being executed every second. Availability of DBMS depends on its complex architecture and underlying hardware or system software. If it fails or crashes amid transactions being executed, it is expected that the system would follow some sort of algorithm or techniques to recover from crashes or failures.

Failure Classification

To see where the problem has occurred we generalize the failure into various categories, as follows:

TRANSACTION FAILURE

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

- **Logical errors:** where a transaction cannot complete because of it has some code error or any internal error condition
- **System errors:** where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

SYSTEM CRASH

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure.

Examples may include operating system errors.

DISK FAILURE:

In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

Storage Structure

We have already described storage system here. In brief, the storage structure can be divided in various categories:

- **Volatile storage:** As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.
- **Nonvolatile storage:** These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include, hard disks, magnetic tapes, flash memory, non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it many have several transactions being executed and various files opened for them to modifying data items. As we know that transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained that is, either all operations are executed or none.

When DBMS recovers from a crash it should maintain the following:

- It should check the states of all transactions, which were being executed.

- A transaction may be in the middle of some operation; DBMS must ensure the atomicity of transaction in this case.
- It should check whether the transaction can be completed now or needs to be rolled back.
- No transactions would be allowed to left DBMS in inconsistent state.
- There are two types of techniques, which can help DBMS in recovering as well as maintaining the atomicity of transaction:
- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are are done on a volatile memory and later the actual database is updated.

Log-Based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to actual modification and stored on a stable storage media, which is failsafe.

Log based recovery works as follows:

The log file is kept on stable storage media

When a transaction enters the system and starts execution, it writes a log about it

<T, Start>

When the transaction modifies an item X, it writes logs as follows:

<T, X, V1, V2>

It reads Tn has changed the value of X, from V1 to V2.

When transaction finishes, it logs:

<T, commit>

DATABASE CAN BE MODIFIED USING TWO APPROACHES:

Deferred database modification: All logs are written on to the stable storage and database is updated when transaction commits.

Immediate database modification: Each log follows an actual database modification. That is, database is modified immediately after every operation.

Recovery with concurrent transactions

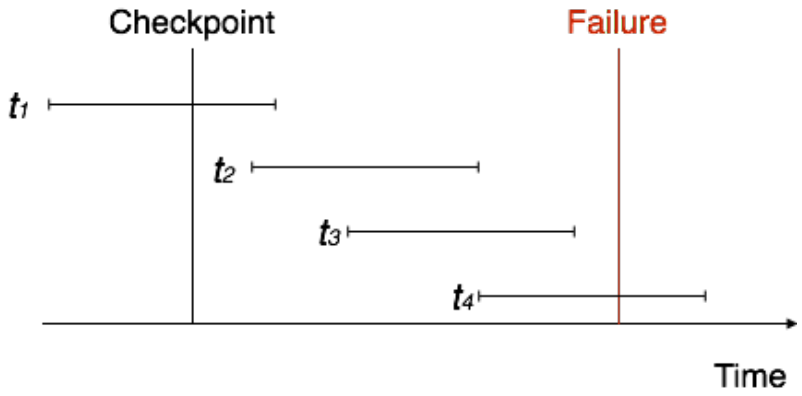
When more than one transaction is being executed in parallel, the logs are interleaved. At the time of recovery it would become hard for recovery system to backtrack all logs, and then start recovering. To ease this situation most modern DBMS use the concept of 'checkpoints'.

CHECKPOINT

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. At time passes log file may be too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in storage disk. Checkpoint declares a point before which the DBMS was in consistent state and all the transactions were committed.

RECOVERY

When system with concurrent transaction crashes and recovers, it does behave in the following manner:



The recovery system reads the logs backwards from the end to the last Checkpoint.

- It maintains two lists, undo-list and redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.
- All transactions in undo-list are then undone and their logs are removed. All transaction in redo-list, their previous logs are removed and then redone again and log saved.