Concurrency Control

Introduction :-

- The term Concurrency refers to the fact that DBMSs typically allow many transactions to access the same database at the same time.
- In such a system, some kind of control mechanism is clearly needed to ensure that concurrent transactions should not interfere with each other.
- For example, Locking Mechanism, which can hold the other transaction to proceed further until the prior transaction can complete the task.

Concurrency Problems :-

- There are essentially three ways in which things can go wrong.
- Three ways, in which a transaction, through correct in itself in the sense, can nevertheless produce a wrong answer if some other transaction interferes with it in some way.
- The interfering transaction will also be correct in itself; it is the uncontrolled interleaving of operating transaction from the two individually transactions that produced the overall incorrect result.

- The three problems are:
  1. The Lost Update problem
  2. The Uncommitted Dependency problem
  3. The Inconsistent Analysis problem

1. The Lost Update Problem :-

- A Lost Update Problem occurs when two transactions that access the same database items have their operations in a way that makes the value of some database item incorrect.
- In other words, if transactions T1 and T2 both read a record and then update it, the effects of the first update will be overwritten by the second update.

<table>
<thead>
<tr>
<th>Transaction - A</th>
<th>Time</th>
<th>Transaction - B</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td></td>
<td>------</td>
</tr>
<tr>
<td>RETRIEVE - t</td>
<td>T1</td>
<td>------</td>
</tr>
<tr>
<td>------</td>
<td>T2</td>
<td>RETRIEVE - t</td>
</tr>
<tr>
<td>UPDATE - t</td>
<td>T3</td>
<td>------</td>
</tr>
<tr>
<td>------</td>
<td>T4</td>
<td>UPDATE - t</td>
</tr>
</tbody>
</table>
Consider the situation illustrated in above given table.

Transaction A retrieves some tuple t at time T1.
Transaction B retrieves the same tuple t at time T2.
Transaction A updates the tuple at time T3 base on the value seen at time T1.
Transaction B update the same tuple at time T4 base on the value seen at time T2 which is the same as that seen at time T1.
Transaction A’s updates are lost at time T4, because transaction B overwrites it without even looking at it.

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction</th>
<th>Steps</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-1</td>
<td>T1</td>
<td>READ Balance</td>
<td>7000</td>
</tr>
<tr>
<td>Time-2</td>
<td>T1</td>
<td>Balance := Balance + 2000</td>
<td></td>
</tr>
<tr>
<td>Time-3</td>
<td>T1</td>
<td>WRITE Balance</td>
<td>9000</td>
</tr>
<tr>
<td>Time-4</td>
<td>T2</td>
<td>READ Balance</td>
<td>9000</td>
</tr>
<tr>
<td>Time-5</td>
<td>T2</td>
<td>Balance := Balance – 3000</td>
<td></td>
</tr>
<tr>
<td>Time-6</td>
<td>T2</td>
<td>WRITE Balance</td>
<td>6000</td>
</tr>
</tbody>
</table>

Serial execution of these transactions under normal circumstances, producing the correct result of Balance = 6000.
Now suppose that both the transactions are executing simultaneously then the things will be the different.
Simultaneous execution of both the transactions will produce the wrong result. It means it will put the database into incorrect state.

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction</th>
<th>Steps</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-1</td>
<td>T1</td>
<td>READ Balance</td>
<td>7000</td>
</tr>
<tr>
<td>Time-2</td>
<td>T2</td>
<td>READ Balance</td>
<td>7000</td>
</tr>
<tr>
<td>Time-3</td>
<td>T1</td>
<td>Balance := Balance + 2000</td>
<td></td>
</tr>
<tr>
<td>Time-4</td>
<td>T2</td>
<td>Balance := Balance – 3000</td>
<td></td>
</tr>
<tr>
<td>Time-5</td>
<td>T1</td>
<td>WRITE Balance</td>
<td>9000</td>
</tr>
<tr>
<td>Time-6</td>
<td>T2</td>
<td>WRITE Balance</td>
<td>4000</td>
</tr>
</tbody>
</table>

First transaction T1 has not yet been committed when the second transaction T2 is executed.
Therefore the transaction T2 still operate the value 7000, and its subtraction yields 4000 in the memory.
In the meantime transaction T1 writes the value 9000 to the disk, which is immediately overwritten by transaction T2.
Thus the addition of 2000 is lost during process.
2. **Dirty READ (The Uncommitted Dependency problem) :-**

- The dirty read problem occurs when one transaction updates a database item and the transaction fails for some reason.
- The updated database item is access by another transaction before it is changed back to the original value.
- In other word, a transaction T1 updates a record, which is read by the transaction T2. Then T1 aborts and T2 now has values which have never formed part of the stable database.
- The uncommitted dependency problems arise if one transaction allowed to retrieve or update a tuple that has been updated by another transaction but not yet been committed by that transaction.
- If it has not yet been committed, there is always a possibility that it never will be committed but will be rolled back.

<table>
<thead>
<tr>
<th>Transaction - A</th>
<th>Time</th>
<th>Transaction - B</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRIEVE – t</td>
<td>T2</td>
<td>UPDATE – t</td>
</tr>
<tr>
<td></td>
<td>T3</td>
<td>ROLLBACK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction - A</th>
<th>Time</th>
<th>Transaction - B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>UPDATE – t</td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td>UPDATE – t</td>
</tr>
<tr>
<td></td>
<td>T3</td>
<td>ROLLBACK</td>
</tr>
</tbody>
</table>

- In first table transaction A sees an uncommitted update-also called an uncommitted change-at time T2.
- That updates is then undone at time T3.
- Now transaction A is therefore operating on wrong assumption that the tuple t has the value that has been seen by that transaction at time T2, whereas in fact, it has a value it had prior to time T1.
- As a result transaction A might produced an incorrect result.
- By the way, that the ROLLBACK of the transaction B might be happened due to some faults on B’s (E.g. System Crash).
- In second table, not only does transaction A become dependent on an uncommitted change at time T2, but it actually loses an updates at time T3, because ROLLBACK at time T3causes tuple t to be restored to its prior to time T1 (This is another version of Lost Update Problem).
3. **Unrepeatable READ (The Inconsistent Retrieval) :-**

- Unrepeatable Read occurs when a transaction calculates some aggregate function over a set of data while other transactions are updating the data.
- The problem is that the transaction might read the data before they are changed and other data after they are changed, thereby producing inconsistent result.

<table>
<thead>
<tr>
<th>Transaction - A</th>
<th>Time</th>
<th>Transaction - B</th>
</tr>
</thead>
</table>
| RETRIEVE ACC1: SUM = 40; | T1 | -----
| RETRIEVE ACC2: SUM = 90; | T2 | -----
| T3 | -----
| T4 | UPDATE ACC3: 30 → 20; |
| T5 | RETRIEVE ACC1: |
| T6 | UPDATE ACC1: 40 → 50; |
| T7 | COMMIT; |
| RETRIEVE ACC3: SUM=110; | T8 | INSTEAD OF 120; |

- In given example, which shows the two transactions A and B operating on account tuple:
- Transaction A is making the sum of account balances, and transaction B is transferring the amount of Rs. 10 from ACC3 to ACC1.
- The result produced by A, 110 is obviously incorrect; if A were write that result back into the database, it would leave the database in the inconsistent state.
- In effect, A has seen an inconsistent state of the database and has therefore performed an inconsistent analysis.
The difference between previous example and this example: there is no question here of A being dependent on an uncommitted change, because B has already committed all its updates before A sees ACC3.

Why the Transactions are Conflicts?

The operations that are of primary interest from a concurrency point of view are database retrieval and database updates.

In other words, we can regard a transaction as considering of a sequence of such operations only.

Let us refer to those operations as simply read and write respectively.

Then it is clear that if A and B are concurrent transactions, problems can occur if A and B want to read or write the same database object, say tuple t.

There are four possibilities for such transactions:

1. **RR**: A and B both want to read tuple t.
   - Read cannot interfere with each other, so there is no problem in this case.
2. **RW**: A reads tuple t and B wants to write t.
   - Inconsistent problem can be caused by RW Conflict.
3. **WR**: A wants to write t and B reads t.
   - Uncommitted Dependencies are caused by WR Conflict.
4. **WW**: A and B both want to write tuple t.
   - The Lost Update Problems are caused by WW Conflict.

Locking Methods for Concurrency Control

Introduction

Concurrency Problems can be solved by the means of Concurrency Control Mechanism called Locking.

It prevents access to database record by a second transaction until the first transaction has complete all of its actions.

The basic idea behind the locking mechanism is that is some transaction A needs an assurance that some object (A Database or a Part of database) it is interested,
Concurrency Control

Database Management System - 2

will not be changed in some manner while its back is turned, it acquires a lock on that object.

- The effect of acquiring a lock is to “Lock the other transaction out of the object”, and thus in particular to prevent them to change it.
- Transaction A is thus able to continue its processing in the certain knowledge that the object in question will remain in a stable state for as long as that the transaction A wishes to it.

- **Types of Locking :-**

  1. Binary Locking
  2. Exclusive and Shared Locking
  3. Two Phase Locking
  4. Three Phase Locking

1. **Binary Locking :-**

   - In binary locking, there are two state of locking namely, “Locked” (1) and “Unlocked” (0).
   - If an object of database table is locked by a transaction no other transaction can use that object.
   - A distinct lock is associated with each database item. If the value of lock in data item t is 1, item t cannot be accessed by a database operation that requires the item. If the object t is unlocked any transaction can lock the object for its use.
   - Two operations: Lock_Item and Unlock_Item.
   - A transaction request access to a data item t by first issuing a Lock_Item (t) operation.
   - If LOCK (t) = 1, the transaction is forced to wait.
   - If LOCK (t) = 0, it is set to 1 and a transaction is allowed to use data item t.
   - As rules a transaction must unlock the object after its termination.

2. **Exclusive and Shared Lock :-**

   - Exclusive Lock (X Lock) :
   - Shared Lock (S Lock):
   - X Lock and S Lock sometimes called Write Lock and Read Lock respectively.
   - If a transaction holds an exclusive (X) lock on tuple t, then a request from some distinct transaction B for a lock of either type on t cannot be immediately granted.
   - If a transaction A holds a shared (S) lock on tuple t then :
   - A request from some distinct transaction B for an X lock on t cannot be immediately be granted.
A request from some distinct transaction B for an S lock on t can and will be immediately granted (that B will allow also holding an S lock on t).

If any transaction holds an X lock on an object no other distinct transaction can get a lock (X or S) on that object.

All locks held by a transaction are released when a transaction is complete.

**Compatibility Matrix for Lock types X and S :-**

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>S</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>S</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

If transaction A holds an X lock than any other distinct transaction B cannot request for X lock on same object.

If transaction A holds an S lock than any other distinct transaction B cannot request for X lock on same object.

If transaction A holds an S lock than any other distinct transaction B can request for S lock on same object.

If transaction A holds an X lock than any other distinct transaction B cannot request for S lock on same object.

3. **Two Phase Locking (2PL) :-**

2PL is a method or a protocol to controlling concurrent processing in which all locking operation precede the first unlocking operation.

Two Phase Locking is the standard protocol used to maintain level three consistencies.

2PL defines how transactions are acquire and release locks.

The essential discipline is that after a transaction has released a lock it may not obtain any further locks.

In practice this means that transactions hold all their locks they are ready to commit.

2PL has the following two phases :

1. Growing Phase :-
   - In which a transaction acquires all the required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.

2. Shrinking Phase :-
In which transaction released all locks and can not obtain any new lock.

The above two phase locking is governed by the following rules:
- Two transactions cannot have conflict locks.
- No lock operation can precede a lock operation in the same transaction.
- No data are affected until all locks are obtained, that is, until the transaction is in its locked point.

**Data Access Protocol/Locking Protocol:**

- Data access protocol or Locking Protocol that makes use of X and S locks as just define to guarantee that concurrency problems.
  1. A transaction that wishes to retrieve a tuple must first acquire an S lock on that tuple.
  2. A transaction that wishes to update a tuple must first acquire an X lock on that tuple. If it already holds an S lock on that tuple, as it will in RETRIEVE-UPDATE sequence, then it must upgrade that S lock to X level.
- Request for locks are usually implicit—a “tuple retrieval” operation implicitly requests an S lock on the relevant tuple and a “tuple update” operation implicitly requests an X lock (or implicitly requests promotion of existing S lock to X level) on the relevant tuple.
- If a lock request from transaction B cannot be immediately granted because it conflicts with a lock already held by transaction A, B goes into a Wait State.
B will wait until the lock request can be granted, which at the earliest will not be until A’s lock is released.

4. X locks are released at the end-of-transaction (COMMIT or ROLLBACK). S locks are normally released at that time also.

<table>
<thead>
<tr>
<th>Transaction - A</th>
<th>Time</th>
<th>Transaction - B</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRIEVE ACC1:</td>
<td>T1</td>
<td>RETRIEVE ACC3:</td>
</tr>
<tr>
<td>(Acquires S Lock on ACC1) SUM =40;</td>
<td></td>
<td>(Acquires S Lock on ACC3)</td>
</tr>
<tr>
<td>RETRIEVE ACC2:</td>
<td>T2</td>
<td>UPDATE ACC3:</td>
</tr>
<tr>
<td>(Acquires S Lock on ACC2) SUM =90;</td>
<td></td>
<td>(Acquires X Lock on ACC3) 30 → 20;</td>
</tr>
<tr>
<td>RETRIEVE ACC3:</td>
<td>T3</td>
<td>RETRIEVE ACC1:</td>
</tr>
<tr>
<td>(Request S Lock on ACC3)</td>
<td></td>
<td>(Acquires S Lock on ACC1)</td>
</tr>
<tr>
<td>UPDATE ACC1:</td>
<td>T4</td>
<td>UPDATE ACC1:</td>
</tr>
<tr>
<td>(Request X Lock on ACC3)</td>
<td></td>
<td>(Request X Lock on ACC3)</td>
</tr>
<tr>
<td>Wait</td>
<td>T5</td>
<td>Wait</td>
</tr>
<tr>
<td>Wait</td>
<td>T6</td>
<td>Wait</td>
</tr>
<tr>
<td>Wait</td>
<td></td>
<td>Wait</td>
</tr>
<tr>
<td>Wait</td>
<td></td>
<td>Wait</td>
</tr>
</tbody>
</table>

**DEADLOCK :-**

- We have seen how two-phase locking protocol can be used to solve the three basic concurrency problems.
- Unfortunately, locking can introduce a problem of its own, principally the problem of **deadlock**.
- In general, deadlock is a situation in which two or more transactions are in a simultaneous wait state, each of them waiting for one of the other to release a lock before it can proceed.
- In the given figure deadlock involves two transactions, but deadlock involving three, four or more transactions are also possible.
r1 and r2 in this figure are intended to represent any lockable resources, not necessarily just database tuples.
- And the Lock Exclusive statements are intended to represent any operations that request X lock either explicitly or implicitly.

**DEADLOCK Detection and Prevention** :-

- **Dead Lock Prevention** :-
  - Dead Lock Prevention technique avoids the condition that leads the deadlocking.
  - It requires every transactions lock all data items it needs in advance.
  - If any of the items cannot be obtain, none of the items are locked.
  - A transaction requiring a new lock is aborted if there is a possibility that the deadlock can occur.
  - Thus a timeout may be used to abort the transaction that has been idle for too long.
  - This is simple but indiscriminate approach because if the transaction is aborted, all the changes made by this transaction is roll back and all the locks obtain by this transaction are released.

- **Dead Lock Detection** :-
  - DBMS periodically test the database for deadlock.
  - If deadlock occur, it is desirable that the system detect it and break it. Detecting a deadlock is detecting a cycle in the Wait-For Graph.
  - Breaking a deadlock involves choosing one of the deadlocked transaction as a victim and rolling it back, thereby releasing its locks and so allowing some other transaction to proceed.
  - Observe, incidentally, that the victim has “failed” and been rolled back through no fault of its own. Some system automatically restart such a transaction from the
beginning, on the assumption that the conditions that caused the deadlock in the first place will probably not arise again.

- **Deadlock avoidance**: -
  - In this technique, the transaction must obtain all the locks it needs before it can be executed.
  - Thus, it avoids rollback of conflicting transactions by requiring that locks be obtained in succession.
  - This is the optimal schema if the detection period is suitable.

- **Wait-For Graph**: -
  - The best way to detect a state of deadlock is for the system to construct and maintain a wait-for graph.
  - If the graph has a cycle that means deadlock is detected.
  - Transaction T1 is waiting for releasing some locks from T2 to proceed further and transaction T2 is also waiting for releasing some locks to proceed further.
  - So now both the transaction are in waiting state. It means a cycle is created in between T1 and T2 means a deadlock is created in between transactions T1 and T2.

**Timestamp Methods for Concurrency Control**

- Timestamp is a unique identifier created by the DBMS to identify the relative starting time of a transaction.
- Typically, timestamp values are assigned in the order in which the transactions are submitted to the system.
- So a timestamp can be thought of as a starting time of transaction.
- Therefore, time stamping is a method for concurrency control in which each transaction is assigned a transaction timestamp.
- *Transaction Timestamp* is a monotonically increasing number, which is often based on the system clock.
- The transactions are managed so that they appear to run in a timestamp order.
Timestamp can also be generated by incrementing a logical counter every time a new transaction starts.

The timestamp must have the two properties namely: (a) uniqueness and (b) monotonicity.

The uniqueness property assures that no equal timestamp values can exist and monotonicity assures that timestamp values always increase.

The READ and WRITE operations within the same transaction must have the same timestamp.

The DBMS executes conflicting operations in timestamp order, thereby ensuring serializability of the transactions.

If two transactions conflict, one often is stopped, rescheduled and assigned a new timestamp value.

Time stamping is a concurrency control protocol in which the fundamental goal is to order transactions globally in such a way that order transactions get priority in the event of a conflict.

The timestamp method does not require any lock.

Conflict Resolution in Timestamp:

To deal with conflicts in timestamp algorithms, some transactions involved in conflicts are made to wait and to abort other.

Following are the main strategies of conflict resolution in TS:

- **Wait-Die**: The older transaction waits for the younger if the younger is accessed first. The younger transaction is aborted (dies) and restarted if it tries to access granule after an older concurrent transaction.

- **Wound-Wait**: The older transaction prevents the younger by suspending (wounding) it if the younger transaction tries to access a granule after an older concurrent transaction.

The handling an aborted transaction is an important aspect of conflict resolution algorithm.

The aborting transaction must be restarted with new timestamp.

**Optimistic Method for Concurrency Control**

- The optimistic method of concurrency control is based on the assumption that conflicts of database operation are rare and that it is better to let transactions run to completion and only check for conflict before they commit.

- An Optimistic concurrency control method is also known as validation or certification methods.

- No checking is done while the transaction is executing. It does not require a Locking or Time stamping techniques.
Instead, a transaction is executed without any restrictions until it is committed.
In optimistic method each transaction is moved from the following phases.
- Read Phase
- Validation or Certification Phase
- Write Phase.

**Read Phase :-**

In a Read Phase the updates are prepared using private copies of a granule. In this phase, the transactions read the value of committed data from the database, execute the needed computations, and make the updates to a private copy of the database value.

All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.

It is conventional to allocate a timestamp to each transaction at the end of its Read to determine the set of transactions that must be examined by the validation procedure.

These set of transactions are those who have finished their Read Phase since the start of the transaction being verified.

**Validation Phase :-**

In a Validation or Certification Phase, the transaction is validate to ensure that the changes made will not affected the integrity and consistency of the database.

If the validation test is positive, the transaction goes to the write phase otherwise the transaction is restarted and the changes are discarded.

Thus, in this phase the list of granule is checked for conflicts.

If conflicts are detected in this phase, the transaction is aborted and restarted.

**Write Phase :-**

In Write Phase, the changes are permanently applied to the database and the updated granules are made public.

Otherwise, the updates are discarded and the transaction is restarted.

This phase is only for the Read-Write transaction not for the Read-Only transaction.

**Advantages of Optimistic Method :-**

This technique is very efficient when conflicts are rare. The occasional conflicts result in the transaction roll back.

The rollback involves only the local copy of data, the database is not involved and thus there will not be any cascading rollback.
Problems of Optimistic Methods for Concurrency Control :-

- Conflicts are expensive to deal with, since the conflicting transaction must be rolled back.
- Longer transactions are more likely to have conflicts and may be repeatedly rolled back because of conflicts with short transactions.