UNIT-5(Concurrency Control Techniques)

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

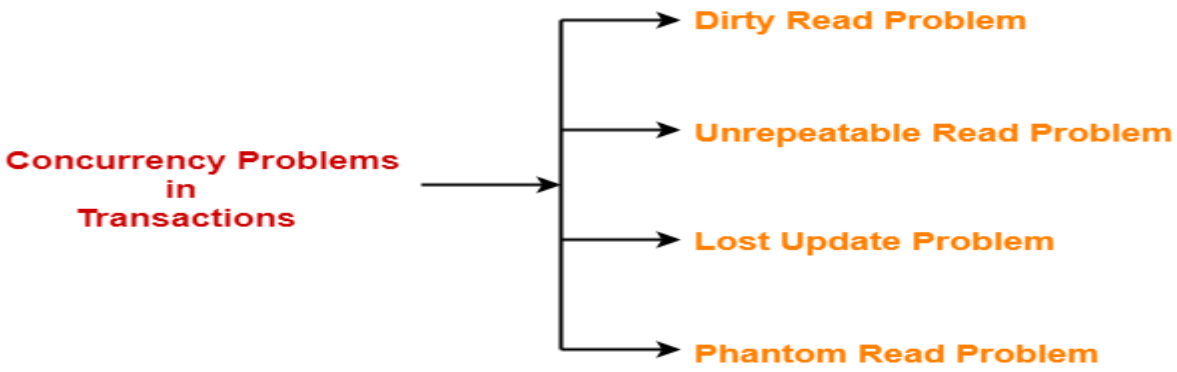
Concurrent Execution in DBMS:

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Concurrency control problems in DBMS-

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. Such problems are called as **concurrency problems**.

The concurrency problems are-



1. Dirty Read Problem
2. Unrepeatable Read Problem
3. Lost Update Problem
4. Phantom Read Problem

1. Dirty Read Problem-

Reading the data written by an uncommitted transaction is called as dirty read.

This read is called as dirty read because-

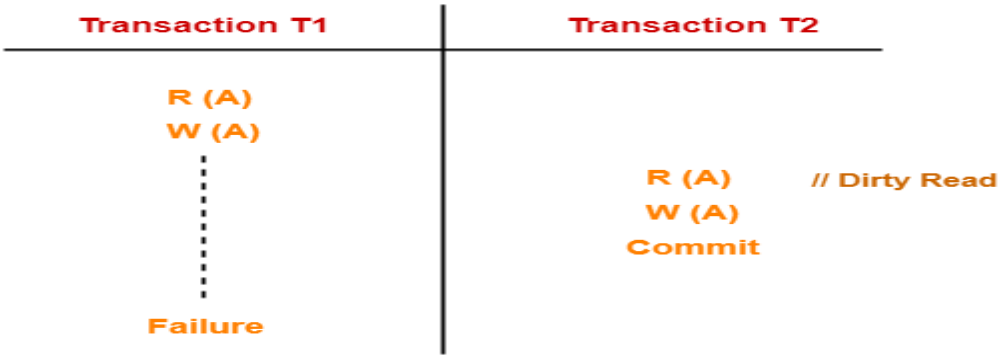
- There is always a chance that the uncommitted transaction might roll back later.
- Thus, uncommitted transaction might make other transactions read a value that does not even exist.
- This leads to inconsistency of the database.

Edit with WPS Office

NOTE-

- Dirty read does not lead to inconsistency always.
- It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.

Example-



Here,

1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back.

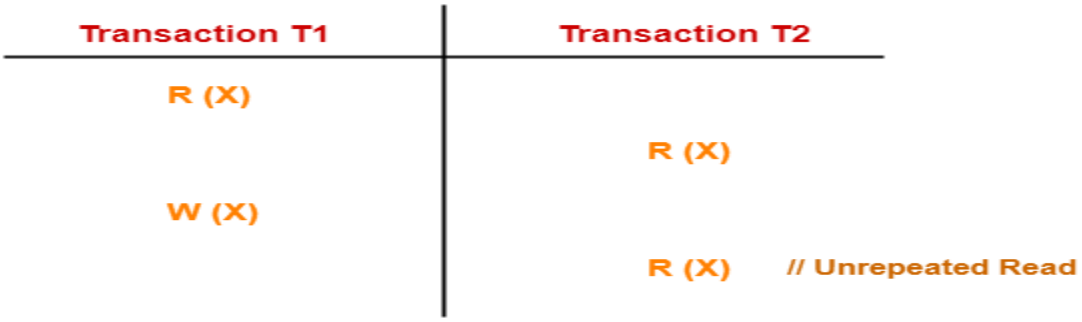
In this example,

- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

2. Unrepeatable Read Problem-

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

Example-



Here,

1. T1 reads the value of X (= 10 say).
2. T2 reads the value of X (= 10).
3. T1 updates the value of X (from 10 to 15 say) in the buffer.
4. T2 again reads the value of X (but = 15).

Edit with WPS Office

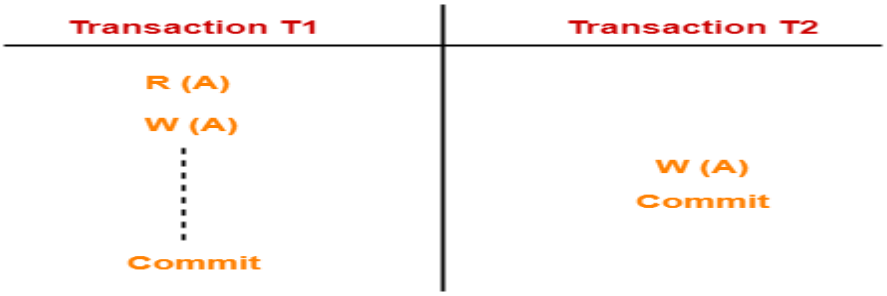
In this example,

- T2 gets to read a different value of X in its second reading.
- T2 wonders how the value of X got changed because according to it, it is running in isolation.

3. Lost Update Problem-

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.

Example-



Here,

1. T1 reads the value of A (= 10 say).
2. T1 updates the value to A (= 15 say) in the buffer.
3. T2 does blind write A = 25 (write without read) in the buffer.
4. T2 commits.
5. When T1 commits, it writes A = 25 in the database.

In this example,

- T1 writes the over written value of A in the database.
- Thus, update from T1 gets lost.

NOTE-

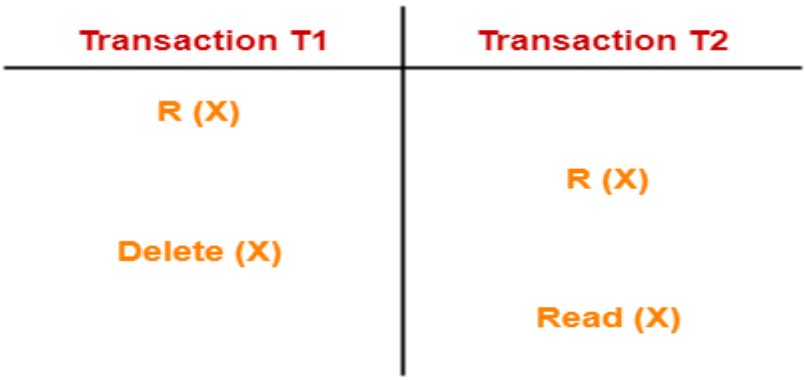
- This problem occurs whenever there is a write-write conflict.
- In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

4. Phantom Read Problem-

This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it finds that the variable does not exist.

Example-

Edit with WPS Office



Here,

1. T1 reads X.
2. T2 reads X.
3. T1 deletes X.
4. T2 tries reading X but does not find it.

In this example,

- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X because according to it, it is running in isolation.

Avoiding Concurrency Problems-

- To ensure consistency of the database, it is very important to prevent the occurrence of above problems.
- **Concurrency Control Protocols** help to prevent the occurrence of above problems and maintain the consistency of the database.

Note: Time-Stamp protocol

Two-phase Locking protocol(2PL)

Validation based Protocol

Refer in unit-5 PDF Hand written notes

Multi Version Schemes or Multiversion Concurrency Control Techniques:

These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written); they are known as **multiversion concurrency control** because several versions (values) of an item are kept by the system. When a transaction requests to read an item, the *appropriate* version is chosen to maintain
the serializability of the currently executing schedule. One reason for keeping multiple versions is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new*

Edit with WPS Office

version and the old
version(s) of the item is retained. An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. In some cases, older versions can be kept in a temporary store. It is also possible that older versions may have to be maintained anyway—for example, for recovery purposes. Some database applications may require older versions to be kept to maintain a history of the changes of
data item values. which keeps track of all changes and the times at which they occurred. In such cases, there
is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Multiversion Technique Based on Timestamp Ordering

In this method, several versions *X*1, *X*2, … , *Xk* of each data item *X* are maintained. For *each version*, the value of version *Xi* and the following two timestamps associated with version *Xi* are kept:
1. **read_TS(*Xi*).** The **read timestamp** of *Xi* is the largest of all the timestamps of transactions that have successfully read version *Xi*.
2. **write_TS(*Xi*).** The **write timestamp** of *Xi* is the timestamp of the transaction that wrote the value of version *Xi*.

Whenever a transaction *T* is allowed to execute a write_item(*X*) operation, a new version *Xk*+1 of item *X* is created, with both the write_TS(*Xk*+1) and the read_TS(*Xk*+1) set to TS(*T*). Correspondingly, when a transaction *T* is allowed to read the value of version *Xi*, the value of read_TS(*Xi*) is set to the larger of the current read_TS(*Xi*) and TS(*T*). To ensure serializability, the following rules are used:
1.If transaction *T* issues a write_item(*X*) operation, and version *i* of *X* has the highest write_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*), and read_TS(*Xi*) > TS(*T*), then abort and roll back transaction *T*; otherwise, create a new version *Xj* of *X* with read_TS(*Xj*) = write_TS(*Xj*) = TS(*T*).

2. If transaction *T* issues a read_item(*X*) operation, find the version *i* of *X* that has the highest write_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*); then return the value of *Xi* to transaction *T,* and set the value of read_TS(*Xi*) to the larger of TS(*T*) and the current read_TS(*Xi*).

As we can see in case 2, a read_item(*X*) *is always successful*, since it finds the appropriate version *Xi* to read based on the write_TS of the various existing versions of *X*.

In case 1, however, transaction *T* may be aborted and rolled back. This happens if *T* attempts to write a version of *X* that should have been read by another transaction *T'* whose timestamp is read_TS(*Xi*); however, *T'* has already read version *Xi*, which was written by the transaction with timestamp equal to write_TS(*Xi*). If this conflict occurs, *T* is rolled back; otherwise, a new version of *X*, written by transaction *T*, is created. if *T* is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction *T* should not be allowed to commit until after all the transactions that have written some version that *T* has read have committed.

Multiple Granularity

Granularity – It is the size of the data item allowed to lock. Now *Multiple Granularity* means hierarchically breaking up the database into blocks that can be locked and can be tracked needs what needs to lock and in what fashion. Such a hierarchy can be represented graphically as a tree.

For example, consider the tree, which consists of four levels of nodes. The highest level represents the entire database. Below it is nodes of type **area**; the database consists of exactly these areas. The area has children nodes which are called files. Every area has those files that are its child nodes. No file can span more than one area.

Edit with WPS Office

Finally, each file has child nodes called records. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file. Hence, the levels starting from the top level are:

- database
- area
- file
- record

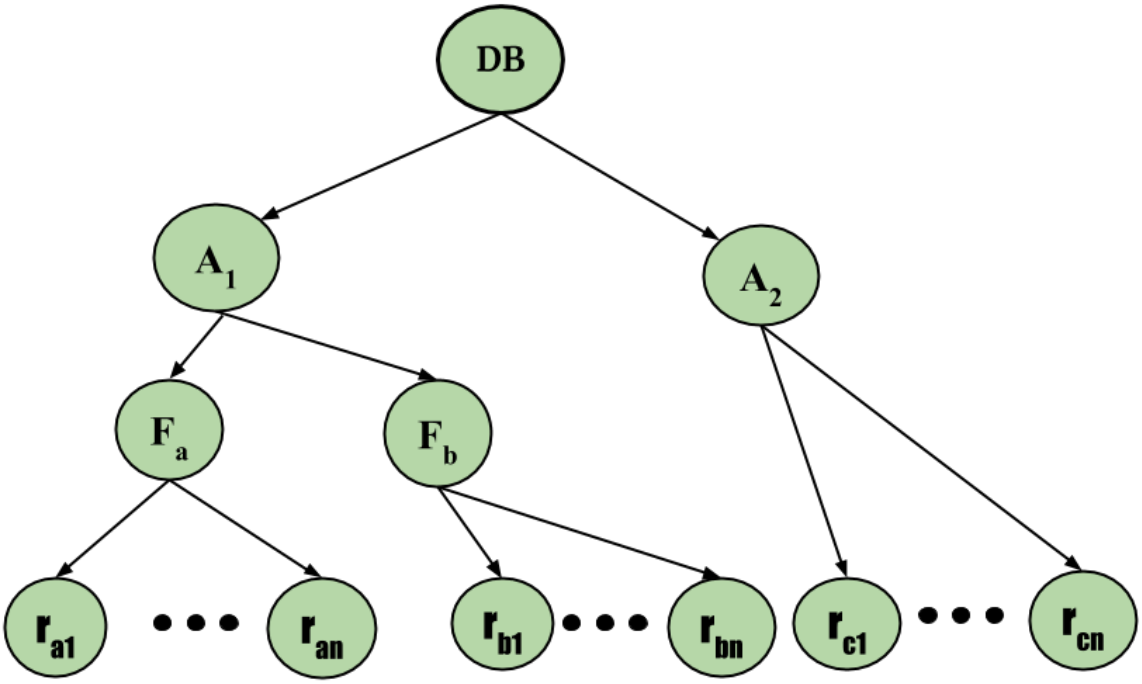


Figure – Multi Granularity tree Hierarchy

Consider the above diagram for the example given, each node in the tree can be locked individually. As in the 2-phase locking protocol, it shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also implicitly locks all the descendants of that node in the same lock mode. For example, if transaction T_i gets an explicit lock on file F_c in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of F_c explicitly.

Now, with locks on files and records made simple, how does the system determine if the root node can be locked? One possibility is for it to search the entire tree but the solution nullifies the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new lock mode, called *Intention lock mode*.

Intention Mode Lock –
In addition to **S** and **X** lock modes, there are three additional lock modes with multiple granularities:

- **Intention-Shared (IS):** explicit locking at a lower level of the tree but only with shared locks.
- **Intention-Exclusive (IX):** explicit locking at a lower level with exclusive or shared locks.
- **Shared & Intention-Exclusive (SIX):** the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive mode locks.

Edit with WPS Office

The compatibility matrix for these lock modes are described below:

	IS	IX	S	SIX	X
IS	✔	✔	✔	✔	✗
IX	✔	✔	✗	✗	✗
S	✔	✗	✔	✗	✗
SIX	✔	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

IS : Intention Shared
IX : Intention Exclusive
S : Shared
X : Exclusive
SIX : Shared & Intention Exclusive

Figure – Multi Granularity tree Hierarchy

The multiple-granularity locking protocol uses the intention lock modes to ensure serializability. It requires that a transaction T_i that attempts to lock a node must follow these protocols:

1. Transaction T_i must follow the lock-compatibility matrix.
2. Transaction T_i must lock the root of the tree first, and it can lock it in any mode.
3. Transaction T_i can lock a node in S or IS mode only if T_i currently has the parent of the node-locked in either IX or IS mode.
4. Transaction T_i can lock a node in X, SIX, or IX mode only if T_i currently has the parent of the node-locked in either IX or SIX modes.
5. Transaction T_i can lock a node only if T_i has not previously unlocked any node (i.e., T_i is two-phase).
6. Transaction T_i can unlock a node only if T_i currently has none of the children of the node-locked.

Observe that the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf to-root) order.
As an illustration of the protocol, consider the tree given above and the transactions:

- Say transaction T_1 reads record R_{a2} in file F_a. Then, T_2 needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock R_{a2} in S mode.
- Say transaction T_2 modifies record R_{a9} in file F_a . Then, T_2 needs to lock the database, area A_1, and file F_a (and in that order) in IX mode, and at last to lock R_{a9} in X mode.
- Say transaction T_3 reads all the records in file F_a. Then, T_3 needs to lock the database and area A_1 (and in that order) in IS mode, and at last to lock F_a in S mode.

Edit with WPS Office

- Say transaction T_4 reads the entire database. It can do so after locking the database in S mode.

Note that transactions T_1, T_3, and T_4 *can access the database concurrently*. Transaction T_2 can execute concurrently with T_1, but not with either T_3 or T_4.
This protocol enhances *concurrency and reduces lock overhead*. Deadlock is still possible in the multiple-granularity protocol, as it is in the two-phase locking protocol. These can be eliminated by using certain deadlock elimination techniques.

Edit with WPS Office