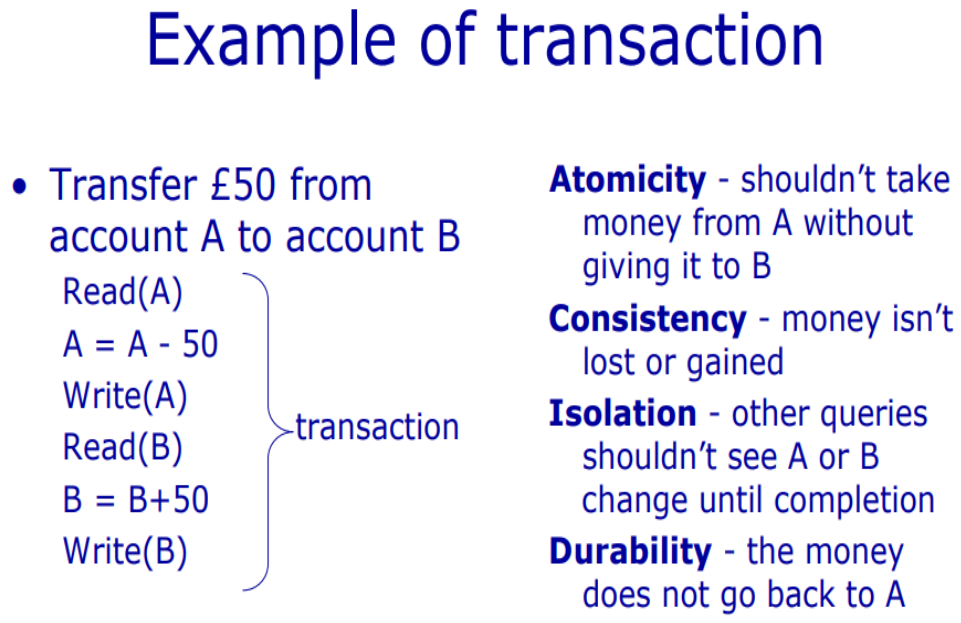
MODULE 4: Transaction

Transaction: A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program. Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

Properties of Transactions/ACID Properties

Transactions have the following four standard properties, usually referred to by the acronym ACID.

- Atomicity − ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- Consistency − ensures that the database properly changes states upon a successfully committed transaction.
- Isolation − enables transactions to operate independently of and transparent to each other.
- Durability − ensures that the result or effect of a committed transaction persists in case of a system failure.



Transaction Control

The following commands are used to control transactions.

- **COMMIT** − to save the changes.
- **ROLLBACK** − to roll back the changes.
- **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK.
- **SET TRANSACTION** − Places a name on a transaction.

Examples of Transactional Commands:

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

Consider the CUSTOMERS table having the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	Kaushik	23	Kota	2000
4	chaitali	25	Mumbai	65000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
  WHERE AGE =25;
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
3	Kaushik	23	Kota	2000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

Example

Consider the CUSTOMERS table having the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	Kaushik	23	Kota	2000
4	chaitali	25	Mumbai	65000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
  WHERE AGE =25;
SQL> ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	Kaushik	23	Kota	2000
4	chaitali	25	Mumbai	65000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	Kaushik	23	Kota	2000
4	chaitali	25	Mumbai	65000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
```

```
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone −

```
SQL> ROLLBACK TO SP2;
Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2.

```
SQL> SELECT * FROM CUSTOMERS;
```

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500
3	Kaushik	23	Kota	2000
4	chaitali	25	Mumbai	65000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

The SET TRANSACTION Command

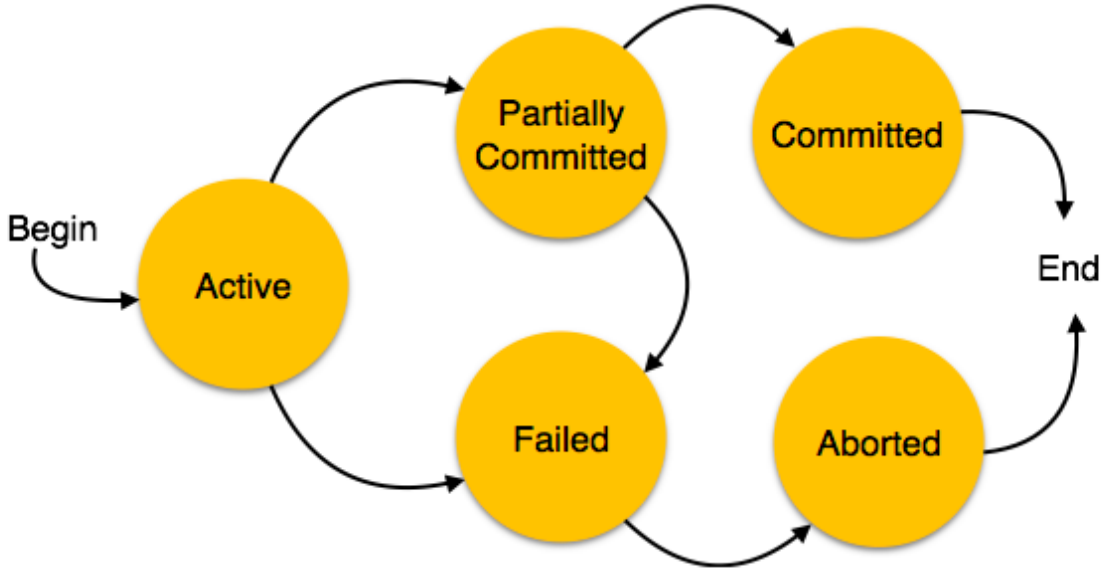
The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

The syntax for a SET TRANSACTION command is as follows.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

States of Transactions

A transaction in a database can be in one of the following states –



- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
  - Re-start the transaction
  - Kill the transaction

DBMS - Concurrency Control

- In a multiprogramming environment where multiple transactions can be executed simultaneously,
- it is highly important to control the concurrency of transactions.
- We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.
- Concurrency control protocols can be broadly divided into three categories −
  - Lock based protocols
  - Time stamp based protocols
  - Optimistic Concurrency Control

1. Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it.

Locks are of two kinds −

- **Binary Locks** −
  - A lock on a data item can be in two states; it is either locked or unlocked.
- **Shared/exclusive** −
  - This type of locking mechanism differentiates the locks based on their uses.
  - If a lock is acquired on a data item to perform a write operation, it is an exclusive lock.
  - Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state.
  - Read locks are shared because no data value is being changed.

There are four types of lock protocols available

1. Simplistic Lock Protocol

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.

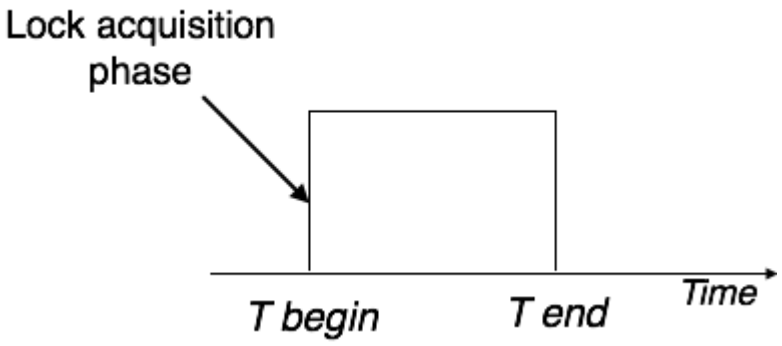
2. **Pre-claiming Lock Protocol**

   a. Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks.

   b. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand.

   c. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over.

   d. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.



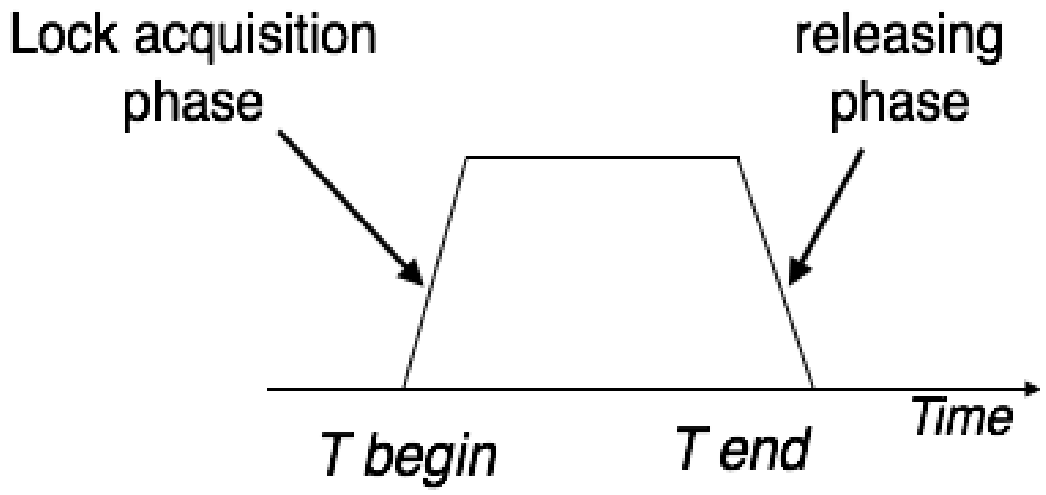
3. **Two-Phase Locking 2PL**

a. This locking protocol divides the execution phase of a transaction into three parts.

b. In the first part, when the transaction starts executing, it seeks permission for the locks it requires.

c. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock,

d. The third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

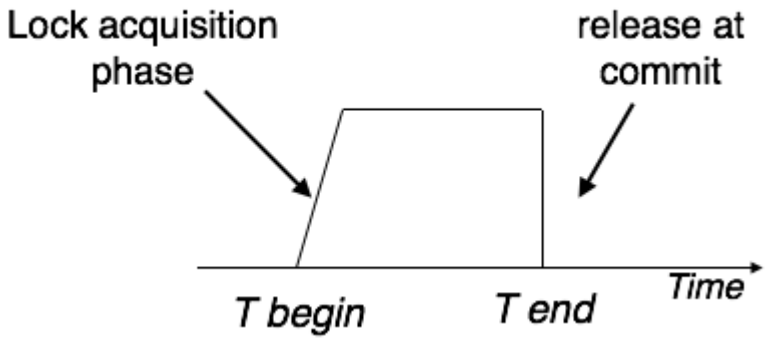


e. Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

f. To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

4. **Strict Two-Phase Locking**

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



Timestamp-based Protocols

1. The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

2. The most commonly used concurrency protocol is the timestamp based protocol.
3. This protocol uses either system time or logical counter as a timestamp.
4. Timestamp-based protocols start working as soon as a transaction is created.
5. Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction.
6. A transaction created at 0002 clock time would be older than all other transactions that come after it.
7. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

TS(TI) denotes the timestamp of the transaction Ti.

Read time stamp

R_TS(X) denotes the Read time-stamp of data-item X.

Write Time stamp

W_TS(X) denotes the Write time-stamp of data-item X.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

If W_TS(X) >TS(Ti) then the operation is rejected.

If W_TS(X) <= TS(Ti) then the operation is executed.

Timestamps of all the data items are updated.

Example on timestamp protocol

- Suppose there are there transactions T1, T2, and T3.
- T1 has entered the system at time 0010
- T2 has entered the system at 0020
- T3 has entered the system at 0030
- Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Advantages:

- Schedules are serializable just like 2PL protocols
- No waiting for the transaction, which eliminates the possibility of deadlocks!

Disadvantages:

- Starvation is possible if the same transaction is restarted and continually aborted

Locking Based Concurrency Control Protocols

Locking-based concurrency control protocols use the concept of locking data items. A lock is a variable associated with a data item that determines whether read/write operations can be performed on that data item. Generally, a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time.

Locking-based concurrency control systems can use either one-phase or two-phase locking protocols.

One-phase Locking Protocol

In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

Two-phase Locking Protocol

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprise of two phases. In the first phase, a transaction only acquires all the locks it needs and do not release any lock. This is called the expanding or the growing phase. In the second phase, the transaction releases the locks and cannot request any new locks. This is called the shrinking phase.

Intent Locks. When an SQL statement in a transaction first accesses a tablespace with a page or row level **lock**, DB2 first takes a particular type of **lock**, called an **intent lock**, at the table and tablespace level for a segmented tablespace or tablespace level for a non-segmented tablespace.

- Intent locks do not conflict with read locks, so acquiring an intent lock does not block other transactions from reading the same row.

- However, intent locks do prevent other transactions from acquiring either an intent lock or a write lock on the same row, guaranteeing that the row cannot be changed by any other transaction before an update.

Implementation of optimistic concurrency control uses multi-versioning.

- Each time that the **server reads a record** to try to update it, the **server makes a copy** of the **version number** of the record and **stores** that copy for **later referencecompares the original version number** that it read against the **version** number of the **currently committed data.**
- If the **version numbers are the same**, then no one else changed the record and the system can write the updated value.
- If the originally read value and the current value on the disk are not the same, then someone has changed the data since it was read, and the current operation is probably out-of-date.
- Thus the system discards the version of the data, aborts the transaction, and returns an error message.
- The step of checking the version numbers is called *validation*.

What is Serializability in DBMS?

- When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state.
- Serializability is a concept that helps us to check which schedules are serializable.
- A serializable schedule is the one that always leaves the database in consistent state.

What is a serializable schedule?

- **A serializable schedule always leaves the database in consistent state.**
- **A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability**

Types of Serializability:

1. **Conflict Serializability**
2. **View Serializability**

Conflict Serializability is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

A schedule is called conflict serializable if we can convert itinto a serial schedule after swapping its non-conflicting operations.

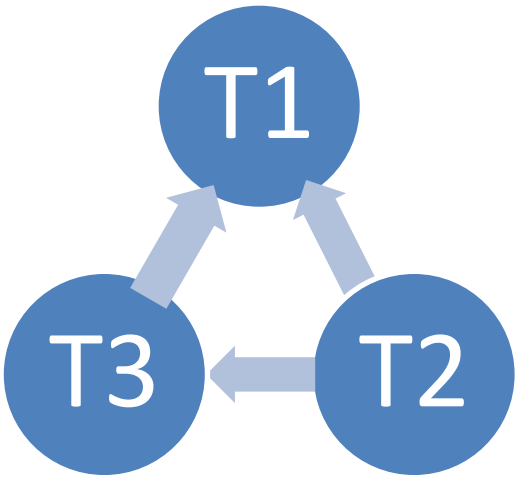
Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.
2. Both the operations are working on same data item.
3. At least one of the operation is a write operation.

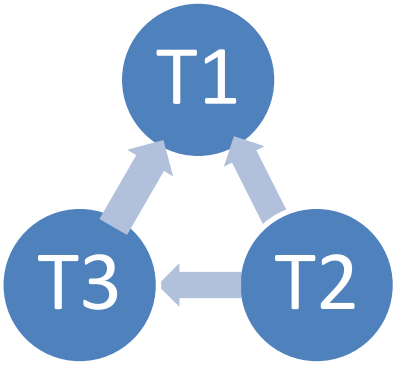
- To check the **Serializability** among transactions Use precedence graph

  Check the precedence graph is loop/cycle

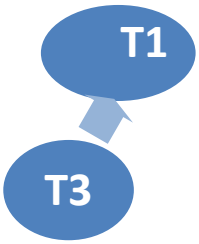


If precedence graph has no loop/cycle then it is called conflict Serializable.

- **To make Serializable check in-degree=0**



- **Check in-degree of a node of above precedence graph**
  1. **T2 has no in-degree so takeout T2**



2. T3 has no in-degree so takeout T3

For **Serializablility**
There are 6 possibilities
T1->T2->T3
T1->T3->T2
T2->T1->T3
T2->T3->T1
T3->T1->T2
T3->T2->T1

What is View Serializability?

View Serializability is a process to find out that a given **schedule** is view serializable or not.

To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule.

Deadlock:

- In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.
- For example, assume a set of transactions {T_0, T_1, T_2, ...,T_n}. T_0 needs a resource X to complete its task.
- Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0.
- Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.
- Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention

- To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute.
- The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.
- There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.
- Wait-Die Scheme
- In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then 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 − then 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 − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.
- This scheme allows the older transaction to wait but kills the younger one

Wound-Wait Scheme

- In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −
- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is $T_i$woundsT_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.
- This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.
- In both the cases, the transaction that enters the system at a later stage is aborted.

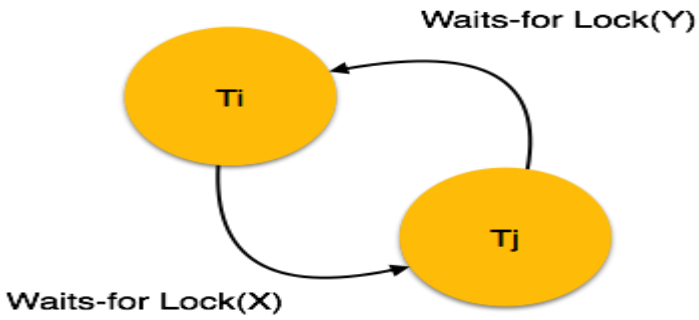
Deadlock Avoidance

- Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance.
- Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

- This is a simple method available to track if any deadlock situation may arise.
- For each transaction entering into the system, a node is created.
- When a transaction T_i requests for a lock on an item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j.
- If T_j releases item X, the edge between them is dropped and T_i locks the data item.
- The system maintains this wait-for graph for every transaction waiting for some data items held by others.

- The system keeps checking if there's any cycle in the graph.



Recovery:

- When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items.
- When a DBMS recovers from a crash, it should maintain the following − It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

Media and System Recovery

- Datafile media recovery is sometimes also called *media* recovery.
- Datafile media recovery is used to recover from a damaged (or lost) datafile, control file or spfile. The goal of datafile media recovery is to restore **database integrity**.
- A datafile that needs media recovery cannot be brought online. A database cannot be opened if any of the online datafiles need media recovery.

System Recovery:

- Any transaction that was running at the time of failure needs to be undone and restarted •
- Any transactions that committed since the last checkpoint need to be redone.