`

UNIT -5

Transaction:-It refers to the logical unit of work in DBMS , which comprises a set of DML statements that are to be executed automatically.

Commit:- It is the state that all transactions completed and all its updates made safe. Rollback:-

It is the state that the transaction comes back to the initial state Properties of

Transaction:

(or)

ACID (Atomicity ,Consistincy , Isolation , Durability) Properties:-

1. **Atomicity:-** Once a transaction starts execution it should be either executed fully or not at all. To meet atomicity, suppose a transaction fails during execution, then the transaction must be rolled back. The task of rolling back will be taken care by DBMS.
2. **Consistency:-** While executing a transaction, to meet consistency the values of that transaction should not be changed until the transaction is completed.
3. **Isolation :-** When a number of transactions are executing concurrently , then the isolates among the transactions i.e which transactions has started and which are going to end etc. This property is known as concurrency control, which is one of major functions of DBMS.
4. **Durability:-** All the updates of the transactions must be persist even if the system fails immediately after committing the transaction. This is the concept of durability. It is done automatically in DBMS.

DB operations:-

Read(x):- It transfers data item x from db to a local buffer of transaction that executes it.

Write(x) :- It transfers data item x from local buffer to the database Ex:

T_i transforms 50 from Account A to B T_i:

```
read(n);
A:=A-50;
Write (A);
Read(B); B:=B+50;
Write(B);
```

`

States of Transaction:

1. **ACTIVE**: A transaction enters this state when it starts execution. During entire execution it is in this state.

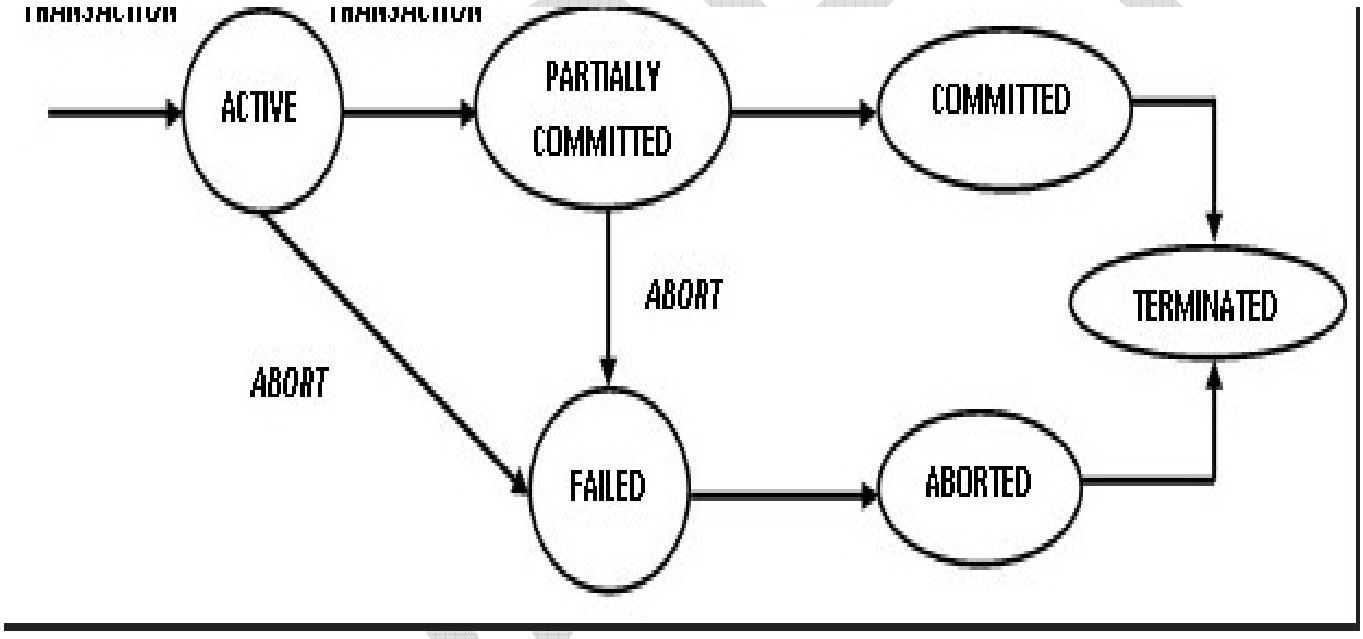
2. **PARTIALLY COMMITTED**: A transaction enters this state during the execution of last statement, but its updates are not made yet safe.

3. **COMMITTED**: A partially committed transactions enters commit state when all it's updates made safe by DBMS.

4. **FIELD**: A transaction enters this state, if it is unable to proceed in a normal manner due to internal error or due to system failure.

5. **Aborted :-** A transaction enters this state when it has been rolled back after failure
6. **Terminated:-** A transaction is said to be terminated when it committed successfully or aborted due to failure

State Diagram of Transaction Processing:



Transaction log

A **transaction log** (also **transaction journal**, **database log**, **binary log** or **audit trail**) is a history of actions executed by a database management system used to guarantee ACID properties over crashes or hardware failures. Physically, a log is a file listing changes to the database, stored in a stable storage format.

A database log record is made up of:

`

- ***Log Sequence Number* (LSN):** A unique ID for a log record. With LSNs, logs can be recovered in constant time..
- ***Prev LSN*:** A link to their last log record. This implies database logs are constructed in linked list form.
- ***Transaction ID number*:** A reference to the database transaction generating the log record.
- ***Type*:** Describes the type of database log record.
- Information about the actual changes that triggered the log record to be written.

Types of database log records:

- **Update Log Record** notes an update (change) to the database. It includes this extra information:
  - *PageID*: A reference to the Page ID of the modified page.
  - *Length and Offset*: Length in bytes and offset of the page are usually included.
  - *Before and After Images*: Includes the value of the bytes of page before and after the page change.
- **Compensation Log Record** notes the rollback of a particular change to the database. It includes this extra information:
  - *undoNextLSN*: This field contains the LSN of the next log record that is to be undone for transaction that wrote the last Update Log.
- **Commit Record** notes a decision to commit a transaction.
- **Abort Record** notes a decision to abort and hence roll back a transaction.
- **Checkpoint Record** notes that a checkpoint has been made. These are used to speed up recovery.
- **Completion Record** notes that all work has been done for this particular transaction. (It has been fully committed or aborted)

Transaction Management with SQL:

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.

Transactional control commands are only used with the **DML Commands** such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The COMMIT Command

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

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

```
COMMIT;
```

Example

`

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

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 |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

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.

The syntax for a ROLLBACK command is as follows –

`

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
|ID|NAME     |AGE|ADDRESS |SALARY |
+----+----------+-----+-----------+----------+
| 1|Ramesh  | 32|Ahmedabad| 2000.00|
| 2|Khilan  | 25|Delhi         | 1500.00|
| 3|kaushik | 23|Kota          | 2000.00|
| 4|Chaitali|25|Mumbai         | 6500.00|
| 5|Hardik  | 27|Bhopal        | 8500.00|
| 6|Komal     | 22 |MP         | 4500.00|
| 7|Muffy     | 24|Indore      |10000.00|
+----+----------+-----+-----------+----------+
```

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 | 2000.00 |
| 2|Khilan  |25|Delhi         | 1500.00|
|3|kaushik|23|Kota          | 2000.00|
|4|Chaitali|25|Mumbai         | 6500.00 |
|5|Hardik  |27|Bhopal        | 8500.00|
| 6|Komal     | 22 |MP         | 4500.00|
| 7|Muffy     |24|Indore      | 10000.00|
+----+----------+-----+-----------+----------+
```

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| 2000.00|
| 2|Khilan  | 25|Delhi         | 1500.00|
| 3|kaushik | 23|Kota          | 2000.00|
| 4|Chaitali| 25|Mumbai        | 6500.00|
| 5|Hardik  | 27|Bhopal        | 8500.00|
| 6|Komal     | 22 |MP         | 4500.00|
| 7|Muffy     | 24|Indore      |10000.00|
+----+----------+-----+-----------+----------+
```

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         | 1500.00|
| 3|kaushik | 23|Kota          | 2000.00|
| 4|Chaitali|25|Mumbai         | 6500.00|
| 5|Hardik  | 27|Bhopal        | 8500.00|
| 6|Komal     | 22 |MP         | 4500.00|
| 7|Muffy     | 24|Indore      |10000.00|
+----+----------+-----+-----------+----------+

6 rows selected.
```

Schedule :- A schedule refers to the order in which instructions of a transaction are executed by the system

Serial Schedule:- In this the transactions are executed strictly one after another . The execution of next transaction started only after completion of previous transaction.

`

Ex:-

Initial balances of accounts A=5000 and B=3000 T_1 :

Transfer 1000 from A to B

T_2 : credit 500 to account A.

Schedule s_1:-

A serial schedule : s1 (T_1 followed by T_2)

T1	T2	Data base state
read(A)		5000
A:=A-1000;		
write(A);		4000
read(B);		3000
B:=B+1000;		
write(B);		4000
		4000
	read(A);	4500
	A:=A+500;	
	write(A);	

Balances at the end of schedule s1 will be A=4500, B=4000 (as expected) Schedule s2:

A serial schedule : s2 (T_2 followed by T_1)

T1	T2	Data base state
	read(A);	5000
	A:=A+500;	
	write(A);	5500
read(A)		5500
A:=A-1000;		
write(A);		4500
read(B);		3000
B:=B+1000;		
write(B);		4000

Balances at the end of schedule s2 will be A=4500, B=4000 (as expected)

`

The schedules s_1 and s_2 execute the transactions T_1 and T_2 serially, i.e; T_1 followed by T_2 in s_1 and T_2 followed by T_1 in S_2. Execution of serial schedule will preserve in the database consistency but they will not give the parallelism among the transactions. So utilization of resources will be poor.

CONCURRENT SCHEDULERS:

When more than one transaction are taken up for the execution at a time that is known as concurrent schedule. Consider two concurrent transactions T_i and T_j. During execution when T_i requests for I/O, then it will go to wait state, the CPU is free at this time. So system will schedule another transaction T_j to CPU. T_j indicates its readiness after I/O request completed. Thus the control of CPU is multiplexed among the no. of transactions.

Ex:

Schedule S_3: A concurrent schedule: T_1 , T_2 , T_1

T_1	T_2	Database State
read(A);		5000
A:=A-1000;		
write(A);		4000
	read(A);	4000
	A:=A+500;	
	write(A);	4,500
read(B);		3000
B:=B+1000;		
write(B);		4000

Balances at the end of schedule S_3 will be A=4,500, B=4000 (as expected)

Schedule S_4:

Another concurrent schedule T_1 , T_2 , T_1

T_1	T_2	Database state
read (A)		5000
A:=A-1000;		
	read(A);	5000
	A:=A+500;	
	write(A);	5500
write(A);		4000
read(B);		3000
B:=B+1000;		
write(B);		4000

Balances at the end of schedule S_4 are A=4000, B=4000 (not expected)

The end result is not as expected since update on A performed by T_2 has been lost.

SERIAL VS CONCURRENT SCHGEDULES:

`

Serial schedules always preserve the database consistency, but they fail to maintain the parallelism among the transactions. On the other hand, if we left the execution of concurrent transactions to the system itself they fail to preserve the database consistency. So some concurrency control mechanisms should be there to maintain concurrent schedules.

EQUIVALENT SCHEDULES:

Two Schedules s and s' are said to be equivalent, if when executed independently each of the schedules transforms are the database from state s_1 to s_2.

Ex:s1 &s2 ,s1&s3, s2&s3 are equivalent.

The primary goal of concurrency is to ensure the atomicity of the execution of transactions in a multi-user database environment. Concurrency controls mechanisms attempt to interleave (parallel) READ and WRITE operations of multiple transactions so that the interleaved execution yields results that are identical to the results of a serial schedule execution.

Problems of Concurrency Control:

When concurrent transactions are executed in an uncontrolled manner, several problems can occur. The concurrency control has the following three main problems:

- Lost updates.
- Dirty read (or uncommitted data).
- Unrepeatable read (or inconsistent retrievals).

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.

Example:

Consider the situation given in figure that shows operations performed by two transactions, Transaction- A and Transaction- B with respect to time.

Transaction- A	Time	Transaction- B
-----	t0	----

`

Read X	t1	----
----	t2	Read X
Update X	t3	----
----	t4	Update X
----	t5	----

At time t1 , Transactions-A reads value of X. At time t2, Transactions-B reads value of X.
At time t3, Transactions-A writes value of X on the basis of the value seen at time t1. At time t4, Transactions-B writes value of X on the basis of the value seen at time t2.
So, update of Transactions-A is lost at time t4, because Transactions-B overwrites it without looking at its current value.
Such type of problem is referred as the Update Lost Problem, as update made by one transaction is lost here.

Dirty Read Problem (uncommitted data):

A dirty read problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value. In other words, 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.

Example:

Consider the situation given in figure:

Transaction- A	Time	Transaction- B
----	t0	----
----	t1	Update X
Read X	t2	----
----	t3	Rollback

`

----	t4	----

At time t1 , Transactions-B writes value of X. At
time t2 , Transactions-A reads value of X.
At time t3 , Transactions-B rollbacks. So, it changes the value of X back to that of prior to t1. So,
Transaction-A now has value which has never become part of the stable database.
Such type of problem is referred as the Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

Unrepeatable read (Inconsistent Retrievals) Problem:

Unrepeatable read (or inconsistent retrievals) occurs when a transaction calculates some summary (aggregate) function over a set of data while other transactions are updating the data.
The problem is that the transaction might read some data before they are changed and other data after they are changed, thereby yielding inconsistent results.
In an unrepeatable read, the transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now, if T1 rereads the record, the new value will be inconsistent with the previous value.

Example:

Consider the situation given in figure that shows two transactions operating on three accounts:

Account-1	Account-2	Account-3
Balance = 200	**Balance = 250**	**Balance = 150**

Transaction- A	Time	Transaction- B
-----	t0	----
Read Balance of Acc-1 sum <--200 Read Balance of Acc-2	t1	----
Sum <-- Sum + 250 = 450	t2	----

`

----	t3	Read Balance of Acc-3
----	t4	Update Balance of Acc-3 150 --> 150 - 50 --> 100
----	t5	Read Balance of Acc-1
----	t6	Update Balance of Acc-1 200 --> 200 + 50 --> 250
---- Read Balance of Acc-3	t7	COMMIT
Sum <-- Sum + 100 = 550	t8	----

Transaction-A is summing all balances; while, Transaction-B is transferring an amount 50 from Account-3 to Account-1.
Here, the result produced by Transaction-A is 550, which is incorrect. if this result is written in database, database will be in inconsistent state, as actual sum is 600.
Here, Transaction-A has seen an inconsistent state of database, and has performed inconsistent analysis.

The Scheduler:

DBMS Scheduler is a feature that enables database administrators and application developers to control when and where various tasks execute in the database environment. The Scheduler can help in simplifying certain management tasks by offering a set of functionality for complex scheduling needs in an organization. The basic capability of a Scheduler is the ability to schedule a job to run at a particular date and time or when a particular event occurs.

DBMS_SCHEDULER is a more sophisticated job scheduler introduced in Oracle 10g. The older job scheduler, DBMS_JOB, is still available, is easier to use in simple cases and fit some needs that DBMS_SCHEDULER does not satisfy.

Example

Although the scheduler is capable of very complicated schedules, on many occasions we just want to create a simple job with everything defined inline as shown below

```
BEGIN
  DBMS_SCHEDULER.create_job (
```

`

```
   job_name        => 'test_full_job_definition',
   job_type        => 'PLSQL_BLOCK',
   job_action      => 'BEGIN my_job_procedure; END;',
   start_date      => SYSTIMESTAMP,
   repeat_interval => 'freq=hourly; byminute=0; bysecond=0;',
   enabled         => TRUE);
END;
/
```

SERIALIZABLE SCHEDULES:

If a concurrent schedule s is logically equivalent to a serial schedule s' then s is said to be serializable schedule.

Ex: The schedule s_3 is logically equivalent to serial schedule s_1 so the schedule s_3 is called serializable schedule. Schedule s_4 is not having any equivalent serial schedule so it is not serializable schedule.

SERIALIZATION: If we left the execution of a concurrent schedule to o/s itself, we cannot predict the end results. DBMS will be in inconsistent state. DBMS will take care about the execution of concurrent transaction by using a concept called concurrency control concept. In some sense the concurrent schedule should be equivalent to a serial schedule. This process is called as serialization.

SERIALIZABILITY TYPES:

a. Conflict serializability
b. View serializability

Conflict serializability:

This is based on the concept of logical swapping of non-conflicting instructions in a schedule.

Non-Conflicting Instructions:

Let a schedule s have 2 consecutive instructions I_i and I_j belonging to two concurrent

Transactions T_i and T_j respectively such that $I_i \epsilon T_i$ and $I_j \epsilon T_j$

The two consecutive instructions are said to be non conflict if

a. The instructions are referring to the access of different data items.
   Eg : I_i may refer to the access of data item P, I_j may refer to the access of another data Q.
b. Both are referring to access of same data item(say Q) & both are only read operations.

`

Logical swapping of 2 instructions which are non-conflict makes the difference as far as database consistency is concerned.

CONFLICT INSTRUCTIONS: Two instructions I_i and I_j belonging to the two concurrent transactions T_i and T_j respectively ae called as conflict instructions if both of the instructions are referring to the same data item (Say Q) and at last one of the item is a Write (Q).

Schedule S5

T1	T2	Data base state

read(A);
write(A);
 	
read(B);
write(B);
	
	read(A);
	write(A);	

The read(B) and write(B) instructions of T1 can be swapped with read(A) and write(A) instructions off T2. The resulting schedule S6 is

Schedule S6

T1	T2

read(A);
write(A);
read(B);
write(B);
	
	read(A);
	write(A);

So, non serial schedule S3 (which is same as S5) is a conflict serializable Schedule because it is conflict equivalent to a serial schedule S6.

Non Serializable Schedule

Schedule S_7

T_1	T_2

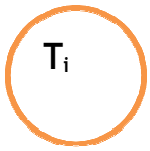
read(A);
 	
write(A);
read(B);
write(B);
	
	read(A);
	write(A);

`

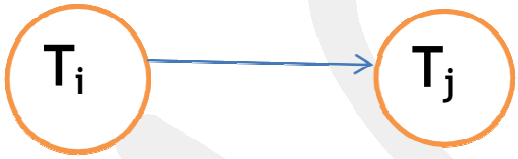
Schedule S_7 is same as the schedule taking only read & write operations. No swapping of non-conflicting instructions will result in a serial schedule. So S_4 is not a conflict serializable.

Testing of serializable of a Schedule:-

1. For a given schedule draw a precedence graph as follows.
   a) Each transaction T_i represented by a vertex.



   b) For each data item Q accessed in schedule S which is having two transactions T_i and T_j, follow the following rules.
      1. T_i executes write(Q) before T_j executes read(Q), draw edge from T_i to T_j
      2. T_i executes read(Q) before T_j executes write(Q), draw an edge from T_i to T_j
      3. T_i executes write(Q) before T_j executes write(Q), draw an edge from T_i to T_j



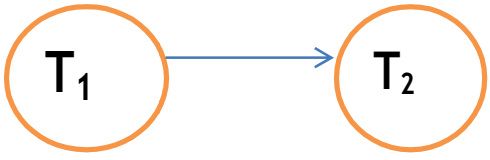
2. Test of serializability:-

If no cycle then the schedule is conflict serializable.

Ex:-

Precedence Graph for schedule S_5

1. T_1 executes read(A) before T_2 executes write(A), so draw an edge from T_1 to T_2
2. T_1 executes write(A) before T_2 executes read(A), so draw an edge from T_1 to T_2
3. T_1 executes write(A) before T_2 executes write(A), so draw an edge from T_1 to T_2

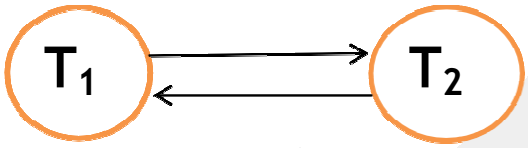


`

The precedence graph has no cycle so S_5 is conflict serializable.

Precedence Graph for S_7:-

1. T_1 executes read(A) before T_2 executes write(A), so draw an edge from T_1 to T_2
2. T_2 executes read(A) before T_2 executes write(A), so draw an edge from T_2 to T_1



The precedence graph has cycle T1→T2→T1 so S_7 is not conflict serializable.

View Serializability:

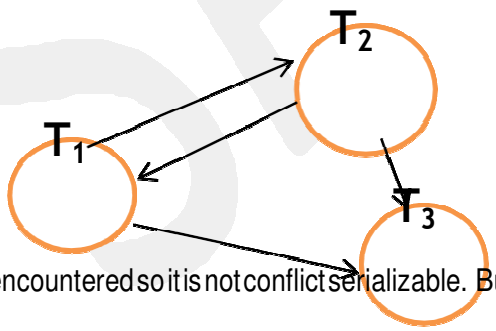
In some situations, a scheduler may not be Conflict serializable, but it may be equivalent to a serial schedule 'S'.

Schedule S_8

T_1	T_2	T_3
read(A);		
	write(A);	
write(A);		
		write(A);

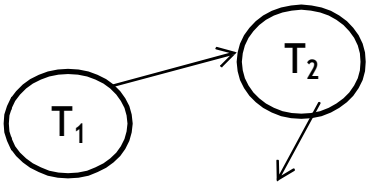
Precedence Graph:-

1. T_1 executes read(A) before T_2 executes write(A), so draw anedge from T_1 to T_2
2. T_2 executes write(A) before T_1 executes write(A), so draw anedge from $T_2$to T_1
3. T_2 writes(A) before T_3 writes(A), so draw an edge from $T_2$to T_3
4. T_1 executes write(A) before T_3 executes write(A), so draw an edge from $T_1$to T_3



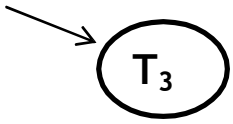
A cycle $T_1 \rightarrow T_2 \rightarrow T_1$ is encountered so it is not conflict serializable. But the result produced by S_9 below will be same as S_8.

T_1	T_2	T_3
read(A); write(A);		



`

	write(A);	write(A);



S_8 is not a conflict serializable, but it is equivalent to a serial schedule S_9. So S_8 is called as a view serializable schedule.

Concurrency Control:-

Concurrency control is one of the major functions of DBMS which is controlled by concurrency Control Manager(CCM)

Concurrency Control Protocols:-

Lock based protocol:

If a concurrent transaction T_i needs to access a shared data item Q, it will first get back on the data item in appreciate mode. Granting of locks will be managed by Concurrency Control Manager.

For accessing G T_i, proceeds as follows.

(i) T_i will request CCM to grant it lock on Q in mode M

(ii) CCM will examine whether lock on Q can be granted immediately or not.

If Q is free(no lock on 2) or is locked in mode compatible with the Requested locking mode M, then the requested locking on Q is granted to T_1 immediately, else T_i made to built till lock is granted.

(iii) T_i will execute its next instruction only after the requested lock is granted.

(iv) After T_i has finished with the accessing of Q, it will release lock on Q. So CCH will grant lock on Q for other waiting transaction.

(v) If T_i fails during execution before releasing its lock, the unreleased locks are automatically released.

Different Locking Modes:-

1. **Shared Mode:-** A Shared Mode(S-Mode) lock is requested by a transaction when it needs to access the data item in only "Read" mode. Shared mode lock can be granted to a transaction on a data item which is already locked on shared mode by some other transaction.
2. **Exclusive Mode:-** An Exclusive(X-Mode) lock is requested by a Transaction when it needs to access a data item not only "Read" mode but also in "Write" mode. One exclusive lock is granted to a transaction on a data item, no other transactions will be provided by shared or execute mode lock on that data item till it is released by that transaction.

Lock Granularity:

`

The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server. Adaptive Server supports locking at the table, page, and row level.

By locking at higher levels of granularity, the amount of work required to obtain and manage locks is reduced. If a query needs to read or update many rows in a table:

- It can acquire just one table-level lock
- It can acquire a lock for each page that contained one of the required rows
- It can acquire a lock on each row

Less overall work is required to use a table-level lock, but large-scale locks can degrade performance, by making other users wait until locks are released. Decreasing the lock size makes more of the data accessible to other users. However, finer granularity locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

Adaptive Server provides these locking schemes:

- **All pages locking**, which locks data pages and index pages
- **Data pages locking**, which locks only the data pages
- **Data rows locking**, which locks only the data rows

For each locking scheme, Adaptive Server can choose to lock the entire table for queries that acquire many page or row locks, or can lock only the affected pages or rows.

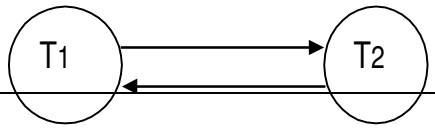
Use of locks to Achieve Serializability

Schedule L_1

T_1	T_2
read(A);	
	read(A);
write(A);	
	write(A);

Precedence Graph:

T_1 executes read(A) before T_2 executes write(A) So draw an edge from T_1 to T_2. T_2 executes read(A) before T_1 executes write(A) . So Draw an edge from T2 to T1



`

The precedence graph is having cycle T1->T2->T1 So the

above schedule L1 is not conflict serializable. By using

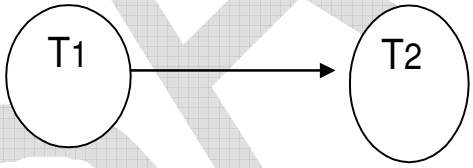
locking to L1

Schedule L2

T1	T2	<<m
Lock-x(A);		/*Grant-x(A,T1)*/
read(A);		/*T1 proceeds to execute read(A)*/
lock-x(A);		/*Because A is currently locked by T1, T2 has to wait until T1 unlocks A*/
write(A);		/* T1 proceeds to execute write(A)*/
unlock(A);		/*T1 releases A and Grant x-lock on A to T2 */
	read(A);	/*T2 now proceeds to execute read(A)*/
	write(A);	/*T2 now proceeds to execute write(A)*/
	unlock(A);	/* T2 releases A*/

Precedence graph for L2

1. T1 executes read(A) before T_2 executes write(A), so edge of T_1 to T_2.
2. T1 executes write(A) before T_2 executes read(A), so edge from T_1 to T_2.
3. T1 executes write(A) before T_2 executes write(A), so edge from T_1 to T_2.



The preceding graph of L2 having no cycle so it is conflict serializable. The use

of locks may not always a serial schedule.

Consider Schedule L3

T1	T2

`

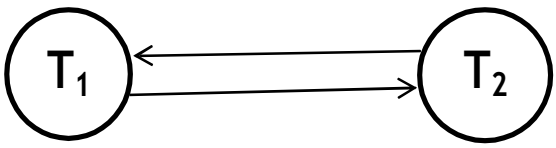
read(A);	
	read(A);
write(A);	
	write(A);
	read(B);
	write(B);
read(B);	
write(B);	

Schedule L2(After use of locking)

T1	T2	CCM
Lock-x(A);		/* Grant_x(A,T1)*/
read(A);		/* T1 proceed read(A)*/
lock-x(A);		/*since, A is locked by T1, T2 has to wait till unlocks A*/
write(A); unlock(A);		/* T1 proceed write(A)*/
		/*T1 releases A and X- lock is granted to T2 on A*/
	read(A);	/* T2 proceeds read(A)*/
	write(A);	/* T1 proceed write(A)*/
lock-X(B);	unlock(A);	/* T2 releases A*/
read(B);		/* Grant-x(B,T2)*/
		/* T2 proceeds read(B)*/
	write(B);	/* T1 proceed write(B)*/
lock-X(B);	unlock(B);	/* T2 releases B*/
read(B);		/* Grant-x(B, T1)*/
write(B);		/* T1 proceeds read(B)*/
unlock(B);		/* T1 proceed write(B)*/
		/*T1 unlocksB*/

Precedence Graph

1. T1 executes read(A) before T2 executes write(A), so edge from T1 to T2.
2. T2 executes write(B) before T1 executes read(B), so edge from T2 to T1.



`

After use of lock also L_3 is not a conflict serializable, because T_1 unlocked A prematurely. So locking needs some additional mechanism to ensure serializability.

TWO-PHASE LOCKING:

In this transaction will obtain & release locks in two distinct phases.

Phase 1: This phase begins as soon as a transaction becomes active. The transaction will only obtain locks in this phase. It will not release any locks. This phase ends as soon as a block hold by the transaction is released. It is known as grouping phase.

Phase 2: This phase begins when the 1st lock is released by the transaction. During this phase no new locks will be obtained, only locks held by the transaction will be released. It is known as shrinking stage.

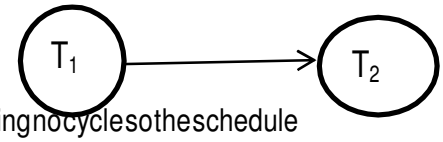
Modify locking of L_3 to Two phase locking

Schedule L_4

T_1	T_2	CCM
Lock-X(A);		/*Grant-X(A,T_1)*/
Read(A);		/*T_1 proceeds read (A)*/
	lock-x(A);	/*Because T_1 locks A, T_2 has to wait till T_1 unlock A*/
Write(A);		/*T_1 proceeds Write(A)*/
lock-x(B);		/* Grant-X(B,T_1)*/
unlock(A);		/*T_1 releases A*/
	read(A);	/*T_2 proceeds read(A) */
	write(A);	/*T_2 proceeds write(A) */
	lock-x(B); unlock(A);	/*Because B is locked by T_1 T_2 wait till T_1 releases B*/
		/*T_2 unlocks A */
read(B);		/* T_1 proceeds read(B) */
write(B);	read(B);	/* T_1 proceeds write(B) */
unlock(B);	write(B);	/* T_1 releases (B) */
	unlock(B);	/*T_2 proceeds read(B) */
		/*T_2 proceeds write(B) */
		/* T_2 unlocks B */

Precedence graph

1. T_1 reads A before T_2 writes A, so edge from T_1 to T_2 2. T_1 writes A before T_2 reads A, so edge from T_1 to T_2 3. T_1 reads B before T_2 writes B, so edge from T_1 to T_2



The precedence graph having no cycles so the schedule

`

L1 is conflict serializable.

Two phase locking results in conflict serializable schedules.

Cascade-less scheduling :-

In this type of schedule a transaction Tj is able to read the value of data item 'A' only when 'A' has been modified by Ti & Ti commits. But in the below schedule T2 accessing A before T1 commits. So it is not a cascade-less schedule.

T_1	T_2	CCM
Lock-X(A);		/* Grant-X(A,T_1)*/
Read(A);		/*T_1 proceeds read (A)*/
	lock-x(A);	/*Since T_1 locks A,T_2 has to wait */
Write(A);		/*T_1 proceeds read (A)*/
lock-x(B);		/*Grant-X(B,T_1)
unlock(A)		/*T_1 releases A*/
	read(A);	/*T_2 proceeds read(A)*/
	write(A);	/*T_1 proceeds write(A)*/
	unlock(A);	/*T_2 releases A */
read(B);		/* T_1 proceeds read(B) */
write(B);		/*T_1 proceeds write(B)*/
unlock(B);		/*T_1 unlocks(B)*/
commit		/*T_1 commits*/
	commit	/*T_2 commits*/

Strict-two-phase locking protocol :- In addition to two-phase locking protocol, it ensures that all the exclusive locks will be released only when the transaction commits. This protocol always ensures cascade-less schedule.

Schedule L_6

T_1	T_2	CCM
Lock-X(A);		/*Grant-X(A,T_1)*/
Read(A);		/*T_1 proceeds read (A)*/
	lock-x(A);	/*Since T_1 already locked A,T_2 has to wait */
Write(A);		/*T_1 proceeds write(A)*/
lock-x(B);		/* Grant-X(B,T_1)*/
unlock(A);		/* T_1 unlocking (A); */
read(B);		/* T_1 proceeds read(B)*/
write(B);		/* T_1 proceeds write(B)*/
unlock(B);		/* T_1 unlocking (B); */
commit;		/* T_1 commits */
	read(A);	/* T_2 proceeds read(A)*/
	write(A);	/* T_2 proceeds write(A)*/
	unlock(A)	/* T2unlocks(A)*/
	commit;	/*T_2 commits*/

`

So the above schedule is conflict serializable and as well as cascade-less.

Limitation of lock based algorithm :- This algorithms suffer from the problem of dead lock . Suppose there are two transactions T1 and T2 access the data items A and B. T1 access the A first and B next and T2 access B first and A next. After accessing A, T1 will wait for the accessing of B which is already locked by B and T2 wait for the access of A which is already locked by T_2. So both the transactions are in the writing state forever. This situation is known as dead lock.

Schedule L_7

T_1	T_2	CCM
lock(A)		/*Grant-x(A,T_1)*/
read(A);		/* T_1 proceeds read(A) */
	lock-x(B);	/*Grant-x(B,T_2)*/
	read(B);	/*T_1 proceeds read (B) */
write(A);		/*T_1 proceeds write (A)*/
lock-x(B);		/*As T_2 already locked B,T_1 will wait* /
	write(B);	/*T_2 proceeds write(B) */
	lock-x(A);	/* As T_1 already locked A, T_2 will wait */

In the above example T_1 and T_2 will wait forever. Thus causing dead lock condition.

Time stamp based algorithms:-

The factors of the time stamp based algorithms

1. When a transaction T_i initiated it is assigned a Time stamp $TS(T_j)$.

Execution of T_1 with time stamp $TS(T_i)$.

2. The Time stamp may be a number which could be a real time (obtained from sys) or counter value (updated by system in ascending order). Initially counter value is zero . It could be incremented by one, after assigning the current value to a new transaction. No two transactions will ever have same time stamp values. If a transaction T_j initialized later than T_i then

1. $T_s(T_j) > T_s(T_i)$

3. For each data item Q in transaction there are two time stamps read time stamp(R-T_s(Q)) and write time stamp (W.T_s(Q)). The initial values of these two are zero. They will increment as follows

->The value of R-T_s(Q) will be equal to the time stamp of the transaction T_k, which has read the data item Q successfully and the time stamp of T_k i.e $T_s(T_k)$ happens to be the largest amongst the time stamps of the transactions which have read Q successfully. So whenever a transaction T_i reads a data item Q successfully R- T_S(Q) updated as

`

$$R\text{-}T_S(Q)=Max(\ R\text{-}T_S(Q),T_S(T_i)\ )$$

->The value $W\text{-}T_S(Q)$ will be equal to the time stamp of the transaction T_k i.e $T_S(T_K)$ which has updated (write) the data item Q successfully and $T_S(T_K)$ happens to be the highest time stamp among the time stamps of all the transactions. So when a transaction T_i writes data item Q successfully $W\text{-}T_S(Q)$ is updated as

$$W\text{-}T_S(Q)=Max(W\text{-}T_S(Q),T_S(T_i)\ )$$

Time Stamp Ordering (TSO) protocol:

Whenever a transaction T_i needs to perform Read(Q) or Write(Q). It requests permission from the system.

Read(Q):-

If the transaction T_i requests (G), the system will process the request as follows

If($T_S(T_i)<W\text{-}T_S(Q)$)

/*Indicates that the transaction say T_j initiated later than T_i has already updated Q. The value of Q, which T_i wants to read has already overridden by T_j. */

Then read (Q) is rejected and T_i is rolled back else
$R\text{-}T_S(Q)=Max(R\text{-}T_S(Q),T_S(T_i)\ )$

Write(Q):-

If a transaction T_i requests write(Q)
The system will process the request as follows If(
$T_S(T_i)<R\text{-}T_S(Q)$)

/*Indicates another transaction T_j initiated later than T_i has already read the old value of Q. T_i should have performed the Write operation before than T_j read generation. */

Then Write(Q) is rejected and transaction T_i is rolled back else if (
$T_S(T_I)<=W\text{-}T_S(Q)$)
/* The transaction T_j initiated later than T_i has already updated Q. So the value of Q which T_i wants write has already modified. */
Write(Q) is rejected and transaction T_i is rolled back else
$_S(Q)=T_S(T_i)$

Advantages :- (over strict two phase backing)

`

1. It is dead lock free science no waiting of transactions to access a data item.
2. It is free of locking overheads.

Disadvantages:-

1. A transaction may be rolled back again and again and may face starvation.
2. Rolling back the transactions is also some work done by the system, so it will reduce the system performance.

Thomas Write rule:-

If $TS(T_i) < W\text{-}TS(Q)$, Indicates that a transaction T_j initiated later than T_i, has already updated Q. Means the value of which T_i wants to write has already modified by T_j. In this condition the time stamp based algorithm will rollback the transaction T_i. There is no necessary to rolling back transaction at this case. This was proposed by Thomas.

Getting ScrealizationBy using TSO protocol:-

Consider the following schedule

T_1	T_2
read(A);	
	read(A);
write(A);	
	write(A);

. The above schedule is not conflict serializable because T_2 executes read(A) before T_1 executes write(A) so we get a cycle $T_1 \rightarrow T_2 \rightarrow T_1$ in the preceding graph

Serializing the above schedule by using TSO Initial

conditions:-

1. $R\text{-}TS(A) < TS(T_1)$
2. $W\text{-}TS(A) < TS(T_1)$
3. $TS(T_1) < TS(T_2)$

assume initial values
$TS(T_1)=TS(T_2)=2$
$R\text{-}TS(A)=0, W\text{-}TS(A)=0$

T_1	T_2	comments
read(A);		$TS(T_1) > w\text{-}TS(A)$ so read(A) is performed by T_1 And $R\text{-}TS(A)=TS(T_1)$
	read(A)	$TS(T_2) > W\text{-}TS(A)$ so read(A) is performed by T_2 And $R\text{-}TS(A)=TS(T_2)$
write(A)		$TS(T_1) <= R\text{-}TS(T_1)$ so T_1 is rolled back and starts with new time stamp say $TS(T_1) > TS(T_2)$
		At this there are two possibilities

`

		(i) T_1 executes write(A) before T_2 executes write(A) (ii) T_2 executes write(A) before T_1 executes read(A)

Execution of T_1 with time stamp $TS(T_1)'$ with possibility (i).

T_1	T_2	Comments
read(A);		$T_S(T_1)'>W\text{-}T_S(A)$ so T_1 executes read(A); $R\text{-}T_S(A)=T_S(T_1)'$;
Write(A);		$T_S(T_1)'<R\text{-}T_S(A)$ and $T_S(T_1)'<=W\text{-}T_S(A)$ else T_1 executes Write(A) so $W\text{-}T_S(A)=T_S(T_1)'$
	Write(A);	$T_S(T_2)<R\text{-}T_S(A)$ so T will be rolled back and starts with new time stamp $T_S(T_2)'>T_S(T_1)'$
	read(A);	$T_S(T_2)'>W\text{-}T_S(A)$ so T_2 executes read(A) and $R\text{-}T_s(A)=T_S(T_2)$,
	Write(A);	$T_S(T_2)=R\text{-}T_s(A)$; So Write(A) execute.

Possibility(ii)

T_1	T_2	Comments
read(A);		$T_S(T_1)>W\text{-}T_S(A)$ so read(A) is preferred by T_1 and $R\text{-}T_S(A)=T_S(T_1)$
	read(A);	$T_S(T_2)>W\text{-}T_S(A)$ so T_2 executes read(A) and $R\text{-}T_S(A)=T_S(T_2)$
Write(A); rollback(T_1);		$T_S(T_1)<R\text{-}T_S(A)$; So T_1 rolled back and starts with $T_S(T_1)'>T_S(T_2)$
	Write(A);	$T_S(T_2)=R\text{-}T_S(A)$; T_2 executes write(A); $W\text{-}T_S(A)=T_S(T_2)$

`

read(A);		$T_S(T_1)$'>W-T_S(A) so T_1 executes read(A) so R-T_S(A)=$T_S(T_1)$'
Write(A);		$T_S(T_1)$'=R-T_S(A) so T_1 executes write(A)

Starvation-Free-Algorithms: (Dead Lock Free):

1. Wait-Die Algorithm:-

It operates as follows.

1. When a transaction is initiated it is assigned a time stamp as in the case of Time stamp based algorithm.
2. When a transaction T_i requests accessing of a data item Q, whatever is already locked by another transaction T_j, then the request is processed as follows.

If ($T_S(T_i)<T_S(T_j)$) then

T_i waits for T_j to finish and releases lock on Q else

T_i rolls-back and restarts with original time stamp

In this algorithm a transaction may be rolled back more than once, while attempting to access data item Q, But since when it restarts with the original time stamp of it, all the transactions older than T_i have already finished and T_i would take control. Thus there is no possibility of starvation.

2. Wound-wait algorithm:-

It operates as follows:

1. When a transaction is initiated, it is assigned a time stamp as in the case of time-stamp algorithm.

2. When a transaction T_i requests to access Q, which is already locked by another transaction T_j then the request is processed follows

If($Ts(T_i) <Ts(T_j)$)

Then T_i forces to T_j rollback , the lock on Q released i.e. Q is now available to T_i

else

Ti waits for T_j to finish and release lock on Q. In this case also there is no possibility of starvation.

Wound-wait Vs Wait-Die

`

Wound-wait is better than wait-Die since the average no. of rollback in this case will be lower.

If a transaction T_j has been forced to rollback by another transaction T_i, then T_i will be initialized with the original time stamp, now it requests the access of Q Ti, which is still holding lock on Q, then Tj will wait for Ti to finish and then access Q,

So in this case there would be one rollback.

Recovery

A failure will leave the database system in a suspect state. Recovery means restoring database (after failure) to a state that is assumed to be consistent. This can be done by using log updates maintained in log-file (non-volatile).

Consider a transaction 1000 from account 100 to 200. Begin

transaction

Update account set balance=balance-1000

Where account.no='100';

Update account set balance= balance+1000

Where account.no='200';

Commit transaction.

The above transaction involves two updates. The database will be inconsistent state during one update and other is still to execute. But at the end it is in consistent state. So a transaction transforms data base from one consistent start other consistent state. If system fails after execution of first update the database will go into inconsistent state. So the transaction should be executed fully or not at all. So during execution if a transaction fails it should be rolled back

.....

System Recovery:

There are two types of system failures

(a) **Locate failure:** Effects only that transaction during execution of which failure occurs.

Recovery: A transaction begins with successful execution of begin transaction and ends with successful execution of a commit or rollback.

A commit establishes a commit point indicates consistent state. Rollback rollbacks the database into previous commit point which is its consistent state.

`

A system does not assume that all the errors will be checked in the application programs. So system will provide automatic rollbacks for the failures that are occurred in application programs. If a transaction successfully commits, the system will garunt update the updates permanently in the database.

(b) **Global failure**: A global failure may effect all the transactions that are running during the failure.

1. System failure: This effects all the transactions that are currently executing but physically no damage to the database. This failure is also called "soft crash".

Ex: Power failure

2. Media failure: It causes damage to the database or some portion of it and also effects the transactions currently using that port of database. This failure is also called "hard crash".

Ex: Disk hard failure

Modes of database updates:

1. Immediate update: when Ti updates Q, the update is immediately reflected to database even before Ti

This case there will be requirement to undo updates of such transaction if system fail before commit.

The system maintains a log of all operations. The log contains the details of all updates , like pre and post_updates as below

<T_i , Q , old_values , new_values>

Tis means T_i update data item Q from old values to new values.

In case of roll back the system use log and log and restore pre_updated value to updated data items i.e,old_value forced to updated data items.

2. Defered update: In this case the updates performed by the transactions are not immediately reflected to data base. The updates kept in RAM in DBMS buffers. Update entry is made in log file. The system takes periodic check point. At each check points the pending updates are transferred from DBMS buffer to data base. The system also maintain the check points record into log file which contain a list of transactions that are running at the time of check points.

Recovery from system failure:

During system failure the contents of main memory i.e. data base of buffers will be lost. The state of transactions running at the time is lost. Such type of transactions should be roll back when we start the system after failure.

There may be some transactions which are already committed and unable to manage the buffer contents into data base such transaction should be redone.

To maintain these, the system takes check points.

`

Taking a check point means:

1. Physically writing the contents of data base buffers into details.

2. Writing a check point record into log fails.

3. Resume or restart the passed transactions.

UNDO and REDO:

Recovery process initiated when the system restarted after the failure .In recovery point of view there are three types of transactions

1. Transactions begin and committed before the last check point. These need no action during recovery.

2. Transactions begin either before or after the last check point committed after the check point ,prior to failure. These transactions need REDO during recovery.

3. Transaction begin either before or after the last check point but still not committed at the time of failure. These need UNDO during recovery.

Recovery procedure:

At restart time the system follows as below

1. The system makes use of the list UNDO list and REDO list. The entities of the last recent check point record are transferred to the UNDO list and initialize REDO list to empty.

2. Starting in the recent check point record , search the log files on the forward direction.

3. If "begin transaction " log entry is found for T_i then add T_i to UNDO list.

4. If "commit" log entry is found for T_i more T_i from UNDO list to REDO list.

5. If the end of log file is reached then UNDO and REDO list are final.

6. The system now works backward through the log file undoing transactions in UNDO list .This is called "backward recovery".

7. Then the system works forward redoing the transactions in the REDO list .This is called "forward recovery".

Log Based Recovery Algorithm:

PASS 1: Scans log file, starting from the failure point in the backward direction, till most recent checkpoint record.

For each transaction entry T_i checkpoint record do

`

UNDO list:=UNDO_list u{T_i}

PASS2: scans log file starting from the last checkpoint record in the forward direction till the failure point

While traversing it prepare it UNDO and REDO lists. A transaction whose begin transaction encountered will be added to UNDO list & and a transaction whose commit transaction encountered , will be added to REDO list and removed from the UNDO list.

Begin

For each record<begin_transaction T_i>_log file do

UNDO list :=undo list u {T_i}

For each record <commit T_i>_log file do Undo

list :=undo list –{T_i}

Redo list :=redo :=redo list u{T_i}

By the end of this transaction undo and redo lists are ready

PASS 3: Scan log file from the failure point in the back word direction till the last check point record, while traversing it is undoing the transactions that are in the undo list

Wait traversing log file do

Begin

for each update record <T_i ,G,old_value ,new_value>_log file If T_i_

undo list

For each record< begin-transaction T_i>c- log file do

Undo.list:=undolist_{T_i};

Pass IV:- Scans log file starting from the last check point till the failure point record occurs. While traversing it will redoing all the transactions that are in the redo list .

When traversing log.file

begin

foreachupdate record<T_i , Q, old_val,new_val>c-logfile do If T_ic-

redo _list

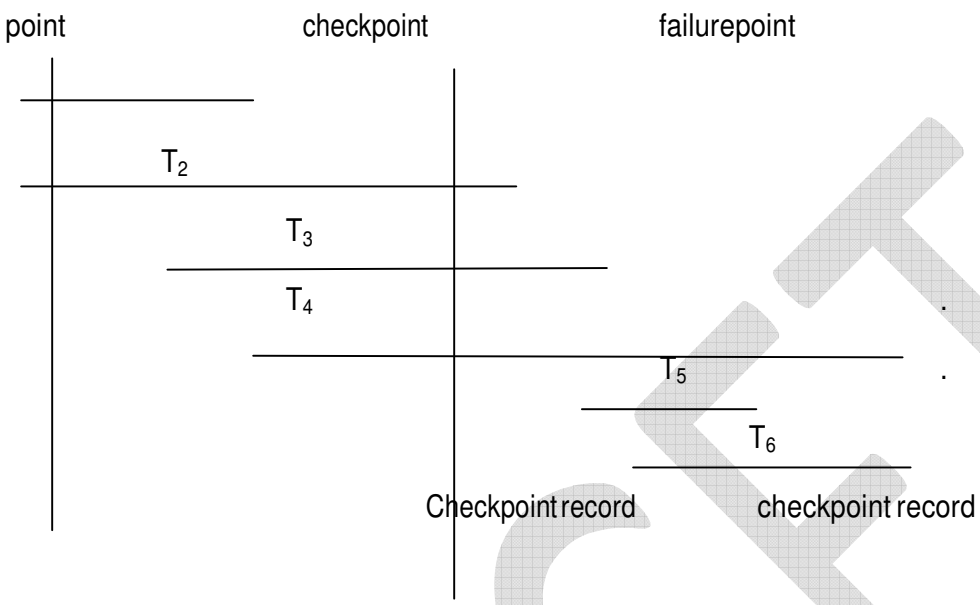
then Q:=new_val

`

For each record <commit T_i>c-logfiles do

redo_list=redo_list-{T_i};

Consider the following failure and solve it using log based recovery algorithm Check



Write-Ahead Logging (WAL) **Protocol.**

We have learnt that logs have to kept in the memory, so that when there is any failure, DB can be recovered using the log files. Whenever we are executing a transaction, there are two tasks – one to perform the transaction and update DB, another one is to update the log files. But when these log files are created - Before executing the transaction, or during the transaction or after the transaction? Which will be helpful during the crash ?

When a log is created after executing a transaction, there will not be any log information about the data before to the transaction. In addition, if a transaction fails, then there is no question of creating the log itself. Suppose there is a media failure, then how a log file can be created? We will lose all the data if we create a log file after the transaction. Hence it is of no use while recovering the data.

Suppose we created a log file first with before value of the data. Then if the system crashes while executing the transaction, then we know what its previous state / value was and we can easily revert the changes. Hence it is always a better idea to log the details into log file before the transaction is executed. In addition, it should be forced to update the log files first and then have to write the data into DB. i.e.; in ATM withdrawal, each stages of transactions should be logged into log files, and stored somewhere in the memory. Then the actual balance has to be updated in DB. This will guarantee the atomicity of the transaction even if the system fails.
This is known as Write-Ahead Logging Protocol.

ARIES(Algorithms for Recovery and Isolation Exploiting Semantics)

`

ARIES is a recovery algorithm designed to work with a no-force, steal database approach.

ARIES uses a steal/no-force approach for writing, and it is based on three concepts:

1. Write-ahead logging
2. Repeating history during redo

   ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred. Transactions that were uncommitted at the time of the crash (active transactions) are undone.

3. Logging changes during undo It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

Before describing this topic we need to elaborate some concepts:

1. Log sequence number

   It refers to a pointer used to identify the log records.

2. Dirty page table

   It refers to pages with their updated version placed in main memory and disk version of it is not updated.

   A table is maintained which is useful in reducing unnecessary redo operation.

3. Fuzzy checkpoints.

   A new type checkpoints i.e. fuzzy checkpoints has been derived that allowed to process new transactions alter the log has been updated without having to update the database.

The ARIES recovery procedure consists of three main phases:

1. **Analysis**

   The analysis phase identifies the dirty (updated) pages in the buffer (Note 6), and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined

2. **REDO**

   The REDO phase actually reapplies updates from the log to the database. Generally, the REDO operation is applied to only committed transactions. However, in ARIES, this is not the case. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. In addition, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and hence need not be reapplied. Thus only the necessary REDO operations are applied during recovery.

3. **UNDO**

   During the UNDO phase, the log is scanned backwards and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. In addition, check pointing is used. These two tables are maintained by the transaction manager and written to the log during check pointing.

Optimistic concurrency control (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.

`

OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods. However, if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly; it is commonly though that other concurrency control methods have better performance under these conditions. However, locking-based ("pessimistic") methods also can deliver poor performance because locking can drastically limit effective concurrency even when deadlocks are avoided.

More specifically, OCC transactions involve these phases:

- **Begin**: Record a timestamp marking the transaction's beginning.
- **Modify**: Read database values, and tentatively write changes.
- **Validate**: Check whether other transactions have modified data that this transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.
- **Commit/Rollback**: If there is no conflict, make all changes take effect. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible. Care must be taken to avoid a TOCTTOU bug, particularly if this phase and the previous one are not performed as a single atomic operation.

Following are the differences between Dirty Read, Non Repeatable Read and Phantom Read.

Dirty Read:-

Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.

Dirty Read Example:-

Transaction A begins.
UPDATE EMPLOYEE SET SALARY = 10000 WHERE EMP_ID= '123';

Transaction B begins.
SELECT * FROM
EMPLOYEE;
(Transaction B sees data which is updated by transaction A. But, those updates have not yet been committed.)

Non-Repeatable Read:-

`

Non Repeatable Reads happen when in a same transaction same query yields to a different result. This occurs when one transaction repeatedly retrieves the data, while a difference transactions alters the underlying data. This causes the different or non-repeatable results to be read by the first transaction.

Non-Repeatable Example:-

Transaction A begins.
SELECT * FROM EMPLOYEE WHERE EMP_ID= '123';

Transaction B begins.
UPDATE EMPLOYEE SET SALARY = 20000 WHERE EMP_ID= '123';
(Transaction B updates rows viewed by the transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.

Phantom Read:-

Phantom read occurs where in a transaction execute same query more than once, and the second transaction result set includes rows that were not visible in the first result set. This is caused by another transaction inserting new rows between the execution of the two queries. This is similar to a non-repeatable read, except that the number of rows is changed either by insertion or by deletion.

Phantom Read Example:-

Transaction A begins.
SELECT * FROM EMPLOYEE WHERE SALARY > 10000 ;

Transaction B begins.
INSERT INTO EMPLOYEE (EMP_ID, FIRST_NAME, DEPT_ID, SALARY) VALUES ('111', 'Jamie', 10, 35000);
Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.