6

TRANSACTION MANAGEMENT

Unit Structure

6.1 ACID Properties

6.2 Serializability

6.3 Concurrency Control, Lock Management

6.4 Lost Update Problem

6.5 Inconsistent Read Problem

6.6 Read-Write Locks

6.7 Deadlocks Handling

6.8 Two Phase Locking protocol

Objectives:

- Will be able to explain the principle of transaction management design.
- Understand transactions and their properties (ACID) & the anomalies that occur without ACID.
- UNDERSTAND the locking protocols used to ensure Isolation & the logging techniques used to ensure Atomicity and Durability.
- Understand Recovery techniques used to recover from crashes.
- Explains the concurrency control and recovery algorithms.
- Applies transaction processing mechanisms in relational databases.

Transaction Management

A transaction is a set of logically related operations. For example, you are transferring money from your bank account to your friend's account, the set of operations would be like this: Simple Transaction Example

1. Read your account balance.
2. Deduct the amount from your balance.
3. Write the remaining balance to your account.
4. Read your friend's account balance.

5. Add the amount to his account balance.
6. Write the new updated balance to his account.

This whole set of operations can be called a transaction. Although I have shown you read, write and update operations in the above example but the transaction can have operations like read, write, insert, update, delete.

6.1 ACID PROPERTIES

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain Atomicity, Consistency, Isolation, and Durability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

Atomicity − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

Consistency − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

Durability − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

Isolation − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

6.2 SERIALIZABILITY

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

Schedule − A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

Serial Schedule − It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state.

To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

Phase 1: Prepare Phase

After each slave has locally completed its transaction, it sends a "DONE" message to the controlling site. When the controlling site has received "DONE" message from all slaves, it sends a "Prepare" message to the slaves.

The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a "Ready" message.

A slave that does not want to commit sends a "Not Ready" message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Phase 2: Commit/Abort Phase

After the controlling site has received "Ready" message from all the slaves

The controlling site sends a "Global Commit" message to the slaves.

The slaves apply the transaction and send a "Commit ACK" message to the controlling site.

When the controlling site receives "Commit ACK" message from all the slaves, it considers the transaction as committed.

After the controlling site has received the first "Not Ready" message from any slave.

The controlling site sends a "Global Abort" message to the slaves.

The slaves abort the transaction and send a "Abort ACK" message to the controlling site.

When the controlling site receives "Abort ACK" message from all the slaves, it considers the transaction as aborted.

6.3 CONCURRENCY CONTROL LOCK MANAGEMENT

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical Database, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.

DBMS Concurrency Control is used to address such conflicts, which mostly occur with a multi-user system. Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.

Potential problems of Concurrency

Here, are some issues which you will likely to face while using the DBMS Concurrency Control method:

Lost Updates occur when multiple transactions select the same row and update the row based on the value selected

Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (dirty read)

Non-Repeatable Read occurs when a second transaction is trying to access the same row several times and reads different data each time.

Incorrect Summary issue occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and

second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

Reasons for using Concurrency control method is DBMS:

To apply Isolation through mutual exclusion between conflicting transactions.

To resolve read-write and write-write conflict issues.

To preserve database consistency through constantly preserving execution obstructions.

The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.

Concurrency control helps to ensure serializability.

Example

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

1. Lock-Based Protocols
2. Two Phase Locking Protocol
3. Timestamp-Based Protocols
4. Validation-Based Protocols

Lock-based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

1. When waiting scheme for locked items is not properly managed
2. In the case of resource leak
3. The same transaction is selected as a victim repeatedly

Two Phase Locking Protocol

Two Phase Locking Protocol also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.

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

In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.

The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.

In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

Growing Phase: In this phase transaction may obtain locks but may not release any locks.

Shrinking Phase: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

In the above-given diagram, you can see that local and global deadlock detectors are searching for deadlocks and solve them with resuming transactions to their initial states.

Strict Two-Phase Locking Method

Strict-Two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It

holds all the locks until the commit point and releases all the locks at one go when the process is over.

Centralized 2PL

In Centralized 2 PL, a single site is responsible for lock management process. It has only one lock manager for the entire DBMS.

Primary copy 2PL

Primary copy 2PL mechanism, many lock managers are distributed to different sites. After that, a particular lock manager is responsible for managing the lock for a set of data items. When the primary copy has been updated, the change is propagated to the slaves.

Distributed 2PL

In this kind of two-phase locking mechanism, Lock managers are distributed to all sites. They are responsible for managing locks for data at that site. If no data is replicated, it is equivalent to primary copy 2PL. Communication costs of Distributed 2PL are quite higher than primary copy 2PL

Timestamp-based Protocols

Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

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

Validation Based Protocol

Validation based Protocol in DBMS also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.

The Validation based Protocol is performed in the following three phases:

1. Read Phase
2. Validation Phase
3. Write Phase

Read Phase

In the Read Phase, the data values from the database can be read by a transaction but the write operation or updates are only applied to the local data copies, not the actual database.

Validation Phase

In Validation Phase, the data is checked to ensure that there is no violation of serializability while applying the transaction updates to the database.

Write Phase

In the Write Phase, the updates are applied to the database if the validation is successful, else; the updates are not applied, and the transaction is rolled back.

Characteristics of Good Concurrency Protocol

An ideal concurrency control DBMS mechanism has the following objectives:

Must be resilient to site and communication failures.

It allows the parallel execution of transactions to achieve maximum concurrency.

Its storage mechanisms and computational methods should be modest to minimize overhead.

It must enforce some constraints on the structure of atomic actions of transactions.

6.4 LOST UPDATE PROBLEM

The lost update problem occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost.

The lost update problem occurs when 2 concurrent transactions try to read and update the same data. Let's understand this with the help of an example.

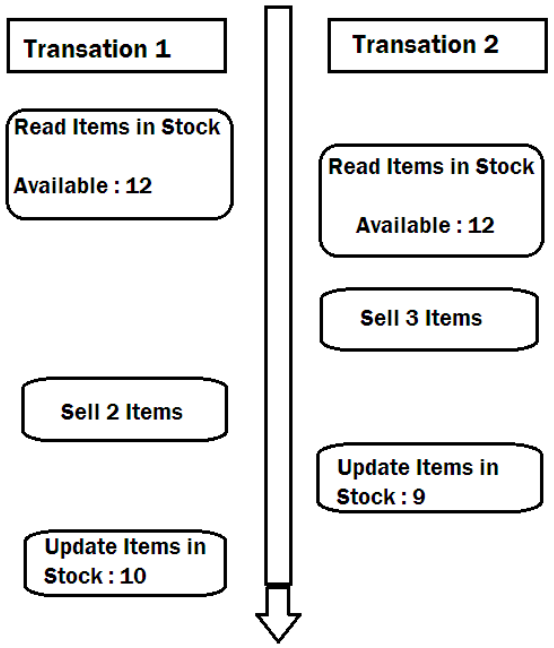
Suppose we have a table named "Product" that stores id, name, and ItemsinStock for a product. It is used as part of an online system that displays the number of items in stock for a particular product and so needs to be updated each time a sale of that product is made.

The table looks like this:

Id	Name	ItemsinStock
1	Laptops	12

Now consider a scenario where a user arrives and initiates the process of buying a laptop. This will initiate a transaction. Let's call this transaction, transaction 1.

At the same time another user logs into the system and initiates a transaction, let's call this transaction 2. Take a look at the following figure.



Transaction 1 reads the items in stock for laptops which is 12. A little later transaction 2 reads the value for ItemsinStock for laptops which will still be 12 at this point of time. Transaction 2 then sells three laptops, shortly before transaction 1 sells 2 items.

Transaction 2 will then complete its execution first and update ItemsinStock to 9 since it sold three of the 12 laptops. Transaction 1 commits itself. Since transaction 1 sold two items, it updates ItemsinStock to 10.

This is incorrect, the correct figure is 12-3-2 = 7

Working Example of Lost Update Problem

Let's us take a look at the lost update problem in action in SQL Server. As always, first, we will create a table and add some dummy data into it.

As always, be sure that you are properly backed up before playing with new code. If you're not sure, see this article on SQL Server backup.

Execute the following script on your database server.

CREATE DATABASE pos;

USE pos;

CREATE TABLE products

(Id INT PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

ItemsinStock INT NOT NULL)

INSERT into products

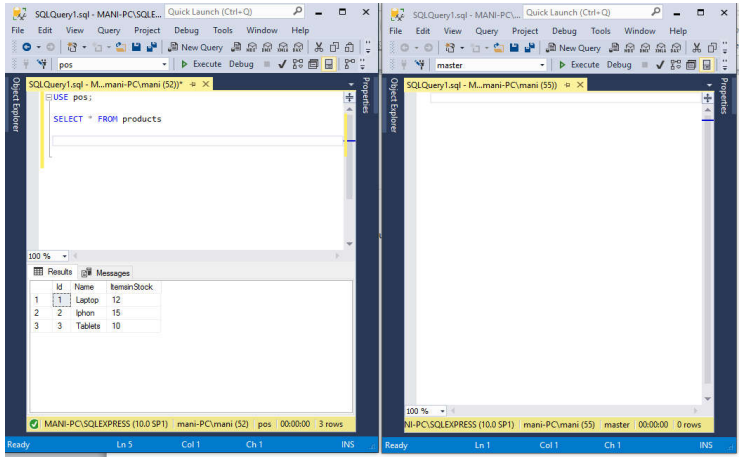
VALUES

(1, 'Laptop', 12),

(2, 'Iphon', 15),

(3, 'Tablets', 10)

Now, open two SQL server management studio instances side by side. We will run one transaction in each of these instances.



Add the following script to the first instance of SSMS.

USE pos;

-- Transaction 1

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

FROM products WHERE Id = 1

WaitFor Delay '00:00:12'

SET @ItemsInStock = @ItemsInStock - 2

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction

This is the script for transaction 1. Here we begin the transaction and declare an integer type variable "@ItemsInStock". The value of this variable is set to the value of the ItemsinStock column for the record with Id 1 from the products table. Then a delay of 12 seconds is added so that transaction 2 can complete its execution before transaction 1. After the delay, the value of @ItemsInStock variable is decremented by 2 signifying the sale of 2 products.

Finally, the value for ItemsinStock column for the record with Id 1 is updated with the value of @ItemsInStock variable. We then print the value of @ItemsInStock variable on the screen and commit the transaction.

In the second instance of SSMS, we add the script for transaction 2 which is as follows:

USE pos;

-- Transaction 2

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

FROM products WHERE Id = 1

WaitFor Delay '00:00:3'

SET @ItemsInStock = @ItemsInStock - 3

UPDATE products SET ItemsinStock = @ItemsInStock

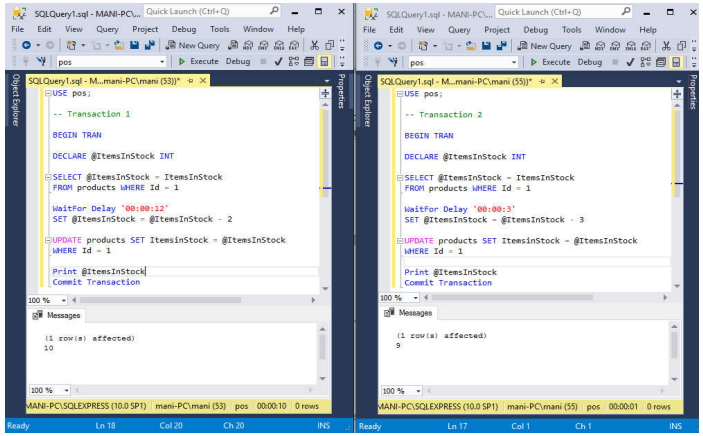
WHERE Id = 1

Print @ItemsInStock

Commit Transaction

The script for transaction 2 is similar to transaction 1. However, here in transaction 2, the delay is only for three seconds and the decrement in the value for @ItemsInStock variable is three, as it is a sale of three items.

Now, run transaction 1 and then transaction 2. You will see transaction 2 completing its execution first. And the value printed for @ItemsInStock variable will be 9. After some time transaction 1 will also complete its execution and the value printed for its @ItemsInStock variable will be 10.



Both of these values are wrong, the actual value for ItemsInStock column for the product with Id 1 should be 7.

It is important to note here that the lost update problem only occurs with read committed and read uncommitted transaction isolation levels. With all the other transaction isolation levels, this problem does not occur.

Read Repeatable Transaction Isolation Level

Let's update the isolation level for both the transactions to read repeatable and see if the lost update problem occurs. But before that, execute the following statement to update the value for ItemsInStock back to 12.

Update products SET ItemsinStock = 12

Script For Transaction 1

USE pos;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

-- Transaction 1

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

```
FROM products WHERE Id = 1

WaitFor Delay '00:00:12'

SET @ItemsInStock = @ItemsInStock - 2

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction</span>
```

Script For Transaction 2

```
<span style="font-size: 14px;">USE pos;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

-- Transaction 2

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

FROM products WHERE Id = 1

WaitFor Delay '00:00:3'

SET @ItemsInStock = @ItemsInStock - 3

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction</span>
```

Here in both the transactions, we have set the isolation level to repeatable read.

Now run transaction 1 and then immediately run transaction 2. Unlike the previous case, transaction 2 will have to wait for transaction 1 to commit itself. After that the following error occurs for transaction 2:

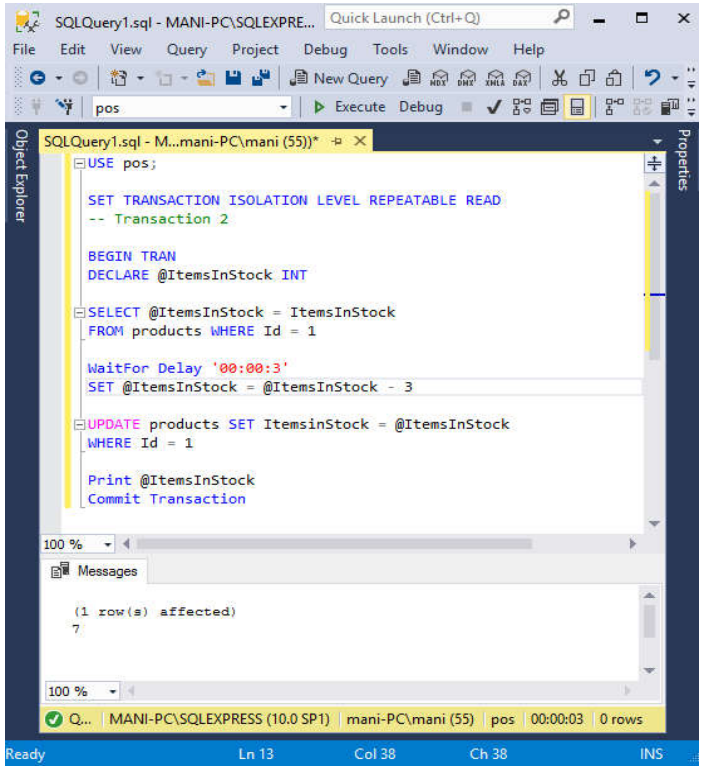
```
Msg 1205, Level 13, State 51, Line 15

Transaction (Process ID 55) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
```


This error occurs because repeatable read locks the resource which is being read or updated by transaction 1 and it creates a deadlock on the other transaction that tries to access the same resource.

The error says that transaction 2 has a deadlock on a resource with another process and that this transaction has been blocked by the deadlock. This means that the other transaction was given access to the resource while this transaction was blocked and not given access to the resource.

It also says to rerun the transaction as the resource is free now. Now, if you run transaction 2 again, you will see the correct value of items in stock i.e. 7. This is because transaction 1 had already decremented the IteminStock value by 2, transaction 2 further decrements this by 3, therefore 12 – (2+3) = 7.



6.5 INCONSISTENT READ PROBLEM

The problem is that the transaction might read some data before they are changed and other data after they are changed, this cause Inconsistent Retrievals

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 + 250 = 450	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.

6.6 READ-WRITE LOCKS

Read Locks:

1. Multiple read locks can be acquired by multiple threads at the same time.
2. When a thread has a read lock on a row/table, no thread can update/insert/delete data from that table. (Even if the thread trying to write data doesn't require a write lock.)
3. A row/table cannot have a read and a write lock at the same time.

Write Locks:

1. When a row/table has a write lock, it cannot be read by another thread if they have a read lock implemented in them but can be read by other threads if no read lock is implemented (i.e simple Select query).

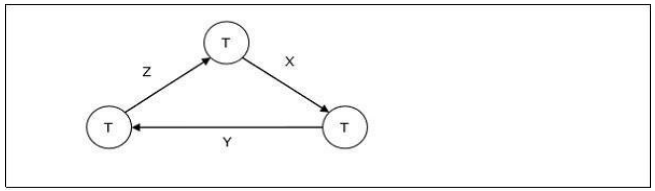
6.7 DEADLOCKS HANDLING

Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.

.



Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely −

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

Deadlock Avoidance

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely wait-die and wound-wait. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

Wait-Die − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.

Wound-Wait − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

Choose the youngest transaction.

Choose the transaction with fewest data items.

Choose the transaction that has performed least number of updates.

Choose the transaction having least restart overhead.

Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are transaction location and transaction control. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

One server may be selected as the center of control.

The center of control may travel from one server to another.

The responsibility of controlling may be shared by a number of servers.

Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.

In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.

If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

Conflict between two transactions in the same site.

Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

.**Distributed Wound-Die**

- If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
- If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.

Distributed Wait-Wait

- If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
- If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

Centralized Deadlock Detector − One site is designated as the central deadlock detector.

Hierarchical Deadlock Detector − A number of deadlock detectors are arranged in hierarchy.

Distributed Deadlock Detector − All the sites participate in detecting deadlocks and removing them.

6.8 TWO-PHASE LOCKING (2PL)

The two-phase locking protocol divides the execution phase of the transaction into three parts. In the **first part**, when the execution of the transaction starts, it seeks permission for the lock it requires. In the **second part**, the transaction acquires all the locks. The **third phase** is started as soon as the transaction releases its first lock. In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.

There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.

Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:

	T1	**T2**
0	LOCK – S(A)	
1		LOCK – S(A)
2	LOCK – X(B)	
3	--	--
4	UNLOCK(A)	
5		LOCK –X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	--	--

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

Growing phase: from step 1-3

Shrinking phase: from step 5-7

Lock point: at 3

Transaction T2:

Growing phase: from step 2-6

Shrinking phase: from step 8-9

Lock point: at 6

Two-Phase Locking (2PL) is a concurrency control method which divides the execution phase of a transaction into three parts. It ensures conflict serializable schedules. If read and write operations introduce the first unlock operation in the transaction, then it is said to be Two-Phase Locking Protocol.

This protocol can be divided into two phases,

1. In Growing Phase, a transaction obtains locks, but may not release any lock.

2. In Shrinking Phase, a transaction may release locks, but may not obtain any lock.

Two-Phase Locking does not ensure freedom from deadlocks.

Types of Two – Phase Locking Protocol

Following are the types of two – phase locking protocol:

1. Strict Two – Phase Locking Protocol
2. Rigorous Two – Phase Locking Protocol
3. Conservative Two – Phase Locking Protocol

1. Strict Two-Phase Locking Protocol

- Strict Two-Phase Locking Protocol avoids cascaded rollbacks.
- This protocol not only requires two-phase locking but also all exclusive-locks should be held until the transaction commits or aborts.
- It is not deadlock free.
- It ensures that if data is being modified by one transaction, then other transaction cannot read it until first transaction commits.
- Most of the database systems implement rigorous two – phase locking protocol.

2. Rigorous Two-Phase Locking

- Rigorous Two – Phase Locking Protocol avoids cascading rollbacks.
- This protocol requires that all the share and exclusive locks to be held until the transaction commits.

3. Conservative Two-Phase Locking Protocol

- Conservative Two – Phase Locking Protocol is also called as Static Two – Phase Locking Protocol.
- This protocol is almost free from deadlocks as all required items are listed in advanced.
- It requires locking of all data items to access before the transaction starts.

❖❖❖❖

7

DCL STATEMENTS

Unit Structure

7.1 Defining a transaction

7.2 Making Changes Permanent with COMMIT

7.3 Undoing Changes with ROLLBACK

7.4 Undoing Partial Changes with SAVEPOINT

7.5 Undoing Partial Changes with ROLLBACK

1 INTRODUCTION TO DCL

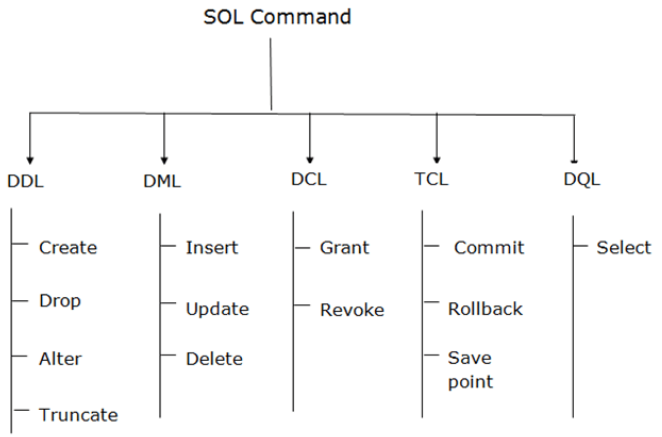
SQL Commands

SQL orders are directions. It is utilized to speak with the data set. It is additionally used to perform explicit assignments, capacities, and questions of information.

SQL can perform different undertakings like make a table, add information to tables, drop the table, change the table, set authorization for clients.

Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



data control language (DCL) is utilized to get to the put away information. It is chiefly utilized for repudiate and to allow the client the necessary admittance to an information base. In the data set, this language doesn't have the element of rollback.

- It is a part of the structured query language (SQL).
- It helps in controlling access to information stored in a database. It complements the data manipulation language (DML) and the data definition language (DDL).
- It is the simplest among three commands.
- It provides the administrators, to remove and set database permissions to desired users as needed.
- These commands are employed to grant, remove and deny permissions to users for retrieving and manipulating a database.
- Grant
- Revoke

A data control language (DCL) is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL). Data Control Language is one of the logical group in SQL Commands. SQL is the standard language for relational database management systems. SQL statements are used to perform tasks such as insert data to a database, delete or update data in a database, or retrieve data from a database.

Though database systems use SQL, they also have their own additional proprietary extensions that are usually only used on their system. For Example Microsoft SQL server uses Transact-SQL (T-SQL) which is an extension of SQL. Similarly Oracle uses PL-SQL which is their proprietary extension for them only. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

Examples of DCL commands include:

GRANT to allow specified users to perform specified tasks.

REVOKE to remove the user accessibility to database object.

The operations for which privileges may be granted to or revoked from a user or role apply to both the Data definition language (DDL) and the Data manipulation language (DML), and may include CONNECT, SELECT, INSERT, UPDATE, DELETE, EXECUTE, and USAGE.

Data control language (DCL) is used to access the stored data. It is mainly used for revoke and to grant the user the required access to a database. In the database, this language does not have the feature of rollback.

- It is a part of the structured query language (SQL).
- It helps in controlling access to information stored in a database.
- It complements the data manipulation language (DML) and the data definition language (DDL).
- It is the simplest among three commands.
- It provides the administrators, to remove and set database permissions to desired users as needed.

These commands are employed to grant, remove and deny permissions to users for retrieving and manipulating a database.

DCL stands for Data Control Language. DCL is used to control user access in a database. This command is related to the security issues.

Using DCL command, it allows or restricts the user from accessing data in database schema.

DCL commands are as follows,

1. GRANT
2. REVOKE

It is used to grant or revoke access permissions from any database user.

2. GRANT COMMAND

SQL Grant command is specifically used to provide privileges to database objects for a user. This command also allows users to grant permissions to other users too.

Syntax:

grant privilege_name on object_name

to {user_name | public | role_name}

Here privilege_name is which permission has to be granted, object_name is the name of the database object, user_name is the user to which access should be provided, the public is used to permit access to all the users.

3 REVOKE :

Revoke command withdraw user privileges on database objects if any granted. It does operations opposite to the Grant command. When a privilege is revoked from a particular user U, then the privileges granted to all other users by user U will be revoked.

Syntax:

revoke privilege_name on object_name

from {user_name | public | role_name}

Example:

grant insert,

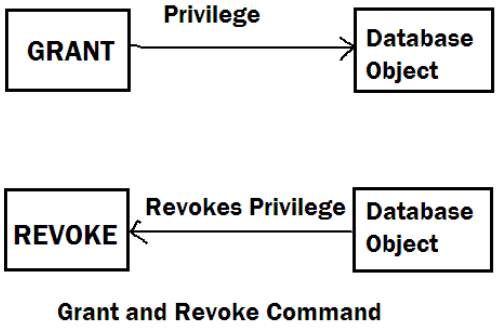
select on accounts to Ram

By the above command user ram has granted permissions on accounts database object like he can query or insert into accounts.

revoke insert,

select on accounts from Ram

By the above command user ram's permissions like query or insert on accounts database object has been removed.



Grant and Revoke Command

Difference between GRANT and REVOKE command.

GRANT	REVOKE
GRANT command allows a user to perform certain activities on the database.	**REVOKE command** disallows a user to perform certain activities.
It grants access privileges for database objects to other users.	It revokes access privileges for database objects previously granted to other users.

Example:
 	
GRANT privilege_name
ON object_name
TO
 	
{
user_name|PUBLIC|role_name
}
 	
[WITH GRANT OPTION];
	Example:
	
	REVOKE privilege_name
	ON object_name
	
	FROM
	{
	user_name|PUBLIC|role_name
	}

7.1 DEFINING A TRANSACTION

A transaction, in the context of a database, is a logical unit that is independently executed for data retrieval or updates. Experts talk about a database transaction as a "unit of work" that is achieved within a database design environment.

In relational databases, database transactions must be atomic, consistent, isolated and durable summarized as the ACID acronym. Engineers have to look at the build and use of a database system to figure out whether it supports the ACID model or not. Then, as newer kinds of database systems have emerged, the question of how to handle transactions becomes more complex.

In traditional relational database design, transactions are completed by COMMIT or ROLLBACK SQL statements, which indicate a transaction's beginning or end. The ACID acronym defines the properties of a database transaction, as follows:

Atomicity: A transaction must be fully complete, saved (committed) or completely undone (rolled back). A sale in a retail store database illustrates a scenario which explains atomicity, e.g., the sale consists of an inventory reduction and a record of incoming cash. Both either happen together or do not happen—it's all or nothing.

Consistency: The transaction must be fully compliant with the state of the database as it was prior to the transaction. In other words, the transaction cannot break the database's constraints. For example, if a database table's Phone Number column can only contain numerals, then consistency dictates that any transaction attempting to enter an alphabetical letter may not commit.

Isolation: Transaction data must not be available to other transactions until the original transaction is committed or rolled back.

Durability: Transaction data changes must be available, even in the event of database failure.

For reference, one of the easiest ways to describe a database transaction is that it is any change in a database, any "transaction" between the database components and the data fields that they contain.

However, the terminology becomes confusing, because in enterprise as a whole, people are so used to referring to financial transactions as simply "transactions." That sets up a central conflict in tech-speak versus the terminology of the average person.

A database "transaction" is any change that happens. To talk about handling financial transactions in database environments, the word "financial" should be used explicitly. Otherwise, confusion can easily crop up. Database systems will need specific features, such as PCI compliance features, in order to handle financial transactions specifically.

As databases have evolved, transaction handling systems have also evolved. A new kind of database called NoSQL is one that does not depend on the traditional relational database data relationships to operate.

While many NoSQL systems offer ACID compliance, others utilize processes like snapshot isolation or may sacrifice some consistency for other goals. Experts sometimes talk about a trade-off between consistency and availability, or similar scenarios where consistently may be treated differently by modern database environments. This type of question is changing how stakeholders look at database systems, beyond the traditional relational database paradigms.

Oracle PL/SQL transaction oriented language. Oracle transactions provide a data integrity. PL/SQL transaction is a series of SQL data manipulation statements that are work logical unit. Transaction is an atomic unit all changes either committed or rollback.

At the end of the transaction that makes database changes, Oracle makes all the changes permanent save or may be undone. If your

program fails in the middle of a transaction, Oracle detect the error and rollback the transaction and restoring the database.

You can use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION command to control the transaction.

COMMIT: COMMIT command to make changes permanent save to a database during the current transaction.

ROLLBACK: ROLLBACK command execute at the end of current transaction and undo/undone any changes made since the begin transaction.

SAVEPOINT: SAVEPOINT command save the current point with the unique name in the processing of a transaction.

AUTOCOMMIT: Set AUTOCOMMIT ON to execute COMMIT Statement automatically.

SET TRANSACTION: PL/SQL SET TRANSACTION command set the transaction properties such as read-write/read only access.

7.2 MAKING CHANGES PERMANENT WITH COMMIT

Committing a transaction means making permanent the changes performed by the SQL statements within the transaction.

Before a transaction that modifies data is committed, the following has occurred:

Oracle has generated undo information. The undo information contains the old data values changed by the SQL statements of the transaction.

Oracle has generated redo log entries in the redo log buffer of the SGA. The redo log record contains the change to the data block and the change to the rollback block. These changes may go to disk before a transaction is committed.

The changes have been made to the database buffers of the SGA. These changes may go to disk before a transaction is committed.

When a transaction is committed, the following occurs:

The internal transaction table for the associated undo tablespace records that the transaction has committed, and the corresponding unique system change number (SCN) of the transaction is assigned and recorded in the table.

The log writer process (LGWR) writes redo log entries in the SGA's redo log buffers to the redo log file. It also writes the transaction's SCN to the redo log file. This atomic event constitutes the commit of the transaction.

Oracle releases locks held on rows and tables.

Oracle marks the transaction complete.

The COMMIT statement to make changes permanent save to a database during the current transaction and visible to other users,

Commit Syntax

SQL>COMMIT [COMMENT "comment text"];

Commit comments are only supported for backward compatibility. In a future release commit comment will come to a deprecated.

Commit Example

```
SQL>BEGIN
    UPDATE emp_information SET emp_dept='Web Developer'
        WHERE emp_name='Saulin';
    COMMIT;
END;
```

7.3 UNDOING CHANGES WITH ROLLBACK

Rolling back means undoing any changes to data that have been performed by SQL statements within an uncommitted transaction. Oracle uses undo tablespaces (or rollback segments) to store old values. The redo log contains a record of changes.

Oracle lets you roll back an entire uncommitted transaction. Alternatively, you can roll back the trailing portion of an uncommitted transaction to a marker called a savepoint.

All types of rollbacks use the same procedures:

Statement-level rollback (due to statement or deadlock execution error)

Rollback to a savepoint

Rollback of a transaction due to user request

Rollback of a transaction due to abnormal process termination

Rollback of all outstanding transactions when an instance terminates abnormally

Rollback of incomplete transactions during recovery

In rolling back an entire transaction, without referencing any savepoints, the following occurs:

Oracle undoes all changes made by all the SQL statements in the transaction by using the corresponding undo tablespace.

Oracle releases all the transaction's locks of data. The transaction ends.

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

ROLLBACK Syntax

SQL>ROLLBACK [To SAVEPOINT_NAME];

ROLLBACK Example

```
SQL>DECLARE
    emp_id    emp.empno%TYPE;
BEGIN
    SAVEPOINT dup_found;
    UPDATE emp SET eno=1
          WHERE empname = 'Forbs ross'
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO dup_found;
END;
/
```

Above example statement is exception raised because eno = 1 is already so DUP_ON_INDEX exception rise and rollback to the dup_found savepoint named.

7.4 UNDOING PARTIAL CHANGESWITH SAVEPOINT

You can declare intermediate markers called savepoints within the context of a transaction. Savepoints divide a long transaction into smaller parts.

Using savepoints, you can arbitrarily mark your work at any point within a long transaction. You then have the option later of rolling back work performed before the current point in the transaction but after a declared savepoint within the transaction. For example, you

can use savepoints throughout a long complex series of updates, so if you make an error, you do not need to resubmit every statement.

Savepoints are similarly useful in application programs. If a procedure contains several functions, then you can create a savepoint before each function begins. Then, if a function fails, it is easy to return the data to its state before the function began and re-run the function with revised parameters or perform a recovery action.

After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. Other transactions that were waiting for the previously locked resources can proceed. Other transactions that want to update previously locked rows can do so.

When a transaction is rolled back to a savepoint, the following occurs:

- Oracle rolls back only the statements run after the savepoint.
- Oracle preserves the specified savepoint, but all savepoints that were established after the specified one are lost.
- Oracle releases all table and row locks acquired since that savepoint but retains all data locks acquired previous to the savepoint.
- The transaction remains active and can be continued.

Whenever a session is waiting on a transaction, a rollback to savepoint does not free row locks. To make sure a transaction does not hang if it cannot obtain a lock, use FOR UPDATE ... NOWAIT before issuing UPDATE or DELETE statements. (This refers to locks obtained before the savepoint to which has been rolled back. Row locks obtained after this savepoint are released, as the statements executed after the savepoint have been rolled back completely.)

Transaction Naming

You can name a transaction, using a simple and memorable text string. This name is a reminder of what the transaction is about. Transaction names replace commit comments for distributed transactions, with the following advantages:

It is easier to monitor long-running transactions and to resolve in-doubt distributed transactions.

You can view transaction names along with transaction IDs in applications. For example, a database administrator can view transaction names in Enterprise Manager when monitoring system activity.

Transaction names are written to the transaction auditing redo record, if compatibility is set to Oracle9i or higher.

LogMiner can use transaction names to search for a specific transaction from transaction auditing records in the redo log.

You can use transaction names to find a specific transaction in data dictionary views, such as V$TRANSACTION.

How Transactions Are Named

Name a transaction using the SET TRANSACTION ... NAME statement before you start the transaction.

When you name a transaction, you associate the transaction's name with its ID. Transaction names do not have to be unique; different transactions can have the same transaction name at the same time by the same owner. You can use any name that enables you to distinguish the transaction.

Commit Comment

In previous releases, you could associate a comment with a transaction by using a commit comment. However, a comment can be associated with a transaction only when a transaction is being committed.

Commit comments are still supported for backward compatibility. However, Oracle strongly recommends that you use transaction names. Commit comments are ignored in named transactions.

SAVEPOINT savepoint_names marks the current point in the processing of a transaction. Savepoints let you rollback part of a transaction instead of the whole transaction.

SAVEPOINT Syntax

```
SQL>SAVEPOINT SAVEPOINT_NAME;
```

SAVEPOINT Example

```
SQL>DECLARE
    emp_id   emp.empno%TYPE;
BEGIN
    SAVEPOINT dup_found;
    UPDATE emp SET eno=1
          WHERE empname = 'Forbs ross'
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
```

```
    ROLLBACK TO dup_found;
END;
```

/Autocommit

No need to execute COMMIT statement every time. You just set AUTOCOMMIT ON to execute COMMIT Statement automatically. It's automatic execute for each DML statement. set auto commit on using following statement,

AUTOCOMMIT Example

```
SQL>SET AUTOCOMMIT ON;
```

You can also set auto commit off,

```
SQL>SET AUTOCOMMIT OFF;
```

Set Transaction

SET TRANSACTION statement is use to set transaction are read-only or both read write. you can also assign transaction name.

SET TRANSACTION Syntax

```
SQL>SET TRANSACTION [ READ ONLY | READ WRITE ]
        [ NAME 'transaction_name' ];
```

Set transaction name using the SET TRANSACTION [...] NAME statement before you start the transaction.

SET TRANSACTION Example

```
SQL>SET TRANSACTION READ WRITE NAME 'tran_exp';
```

SAVEPOINT For Reverting Partial Changes

SAVEPOINT gives name and identification to the present transaction processing point. It is generally associated with a ROLLBACK statement. It enables us to revert some sections of a transaction by not touching the entire transaction.

As we apply ROLLBACK to a SAVEPOINT, all the SAVEPOINTS included following that particular SAVEPOINT gets removed [that is if we have marked three SAVEPOINTS and applied a ROLLBACK on the second SAVEPOINT, automatically the third SAVEPOINT will be deleted.]

A COMMIT or a ROLLBACK statement deletes all SAVEPOINTS. The names given to SAVEPOINT are undeclared identifiers and can be reapplied several times inside a transaction. There is a movement of SAVEPOINT from the old to the present position inside the transaction.

A ROLLBACK applied to a SAVEPOINT affects only the ongoing part of the transaction. Thus a SAVEPOINT helps to split a lengthy transaction into small sections by positioning validation points.

Syntax for transaction SAVEPOINT:

```
SAVEPOINT < save_n>;
```

Here, save_n is the name of the SAVEPOINT.

Let us again consider the TEACHERS table we have created earlier.

Code implementation of ROLLBACK WITH SAVEPOINT:

```
INSERT INTO TEACHERS VALUES (4, 'CYPRESS', 'MICHEAL');
SAVEPOINT s;
INSERT INTO TEACHERS VALUES (5, 'PYTHON', 'STEVE');
INSERT INTO TEACHERS VALUES (6, 'PYTEST', 'ARNOLD');
ROLLBACK TO s;
INSERT INTO TEACHERS VALUES (7, 'PROTRACTOR',
'FANNY');
COMMIT;
```

Next, the below query is executed:

```
SELECT * FROM TEACHERS;
```

Output of the above code should be:

CODE	SUBJECT	NAME
2	UFT	SAM
1	SELENIUM	TOP
3	JMETERE	TONK
4	CYPRESS	MICHEAL
7	PROTRACTOR	FANNY

In the above code, after ROLLBACK with SAVEPOINT s is applied, only two more rows got inserted, i.e. teachers with CODE 4 and 7, respectively. Please note teachers with code 1, 2, and 3 have been added during the table creation.

7.5 ROLLBACK TO UNDO CHANGES

If a present transaction is ended with a ROLLBACK statement, then it will undo all the modifications that are supposed to take place in the transaction.

A ROLLBACK statement has the following features as listed below:

The database is restored with its original state with a ROLLBACK statement in case we have mistakenly deleted an important row from the table.

In the event of an exception which has led to the execution failure of a SQL statement, a ROLLBACK statement enables us to jump to the starting point of the program from where we can take remedial measures.

The updates made to the database without a COMMIT statement can be revoked with a ROLLBACK statement.

Syntax for transaction ROLLBACK:

ROLLBACK;

Syntax for transaction ROLLBACK with SAVEPOINT:

ROLLBACK [TO SAVEPOINT < save_n>];

Here, the save_n is the name of the SAVEPOINT.

Let us consider the TEACHERS table we have created earlier.

Code implementation with ROLLBACK:

```
DELETE FROM TEACHERS WHERE CODE= 3;
 ROLLBACK;
```

Next, the below query is executed:

```
SELECT * FROM TEACHERS;
```

Output of the above code should be:

CODE	SUBJECT	NAME
2	UFT	SAM
1	SELENIUM	TOP
3	JMETERE	TONK

In the above code, we have executed a DELETE statement which is supposed to delete the record of the teacher with CODE equal to 3. However, because of the ROLLBACK statement, there is no impact on the database, and deletion is not done.

8

CRASH RECOVERY

Unit Structure

8.1 ARIES algorithm

8.2 The log based recovery

8.3 Recovery related structures like transaction

8.4 Dirty page table

8.5 Write-ahead log protocol

8.6 Check points

8.7 Recovery from a system crash

8.8 Redo and Undo phases.

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

Failure Classification

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

Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be −

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

System Crash

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

Examples may include operating system errors.

Disk Failure

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

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

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories −

1. **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.

2. **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Recovery and Atomicity

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

When a DBMS recovers from a crash, it should maintain the following −

1. It should check the states of all the transactions, which were being executed.
2. A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
3. It should check whether the transaction can be completed now or it needs to be rolled back.

4. No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

1. Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
2. Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

8.1 ARIES ALGORITHM

(Algorithm for Recovery and Isolation Exploiting Semantics (ARIES))

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) is based on the Write Ahead Log (WAL) protocol. Every update operation writes a log record which is one of the following :

1. Undo-only log record:

Only the before image is logged. Thus, an undo operation can be done to retrieve the old data.

2. Redo-only log record:

Only the after image is logged. Thus, a redo operation can be attempted.

3. Undo-redo log record:

Both before images and after images are logged.

In it, every log record is assigned a unique and monotonically increasing log sequence number (LSN). Every data page has a page LSN field that is set to the LSN of the log record corresponding to the last update on the page. WAL requires that the log record corresponding to an update make it to stable storage before the data page corresponding to that update is written to disk. For performance reasons, each log write is not immediately forced to disk. A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full. A transaction cannot be declared committed until the commit log record makes it to disk.

Once in a while the recovery subsystem writes a checkpoint record to the log. The checkpoint record contains the transaction table and the dirty page table. A master log record is maintained separately, in stable storage, to store the LSN of the latest checkpoint record that made it to disk. On restart, the recovery subsystem reads the master log record to find the checkpoint's LSN, reads the checkpoint record, and starts recovery from there on.

The recovery process actually consists of 3 phases:

Analysis:

The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.

Redo:

Starting at the earliest LSN, the log is read forward and each update redone.

Undo:

The log is scanned backward and updates corresponding to loser transactions are undone.

Attention reader! Don't stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the CS Theory Course at a student-friendly price and become industry ready.

8.2 LOG BASED RECOVERY

Log is nothing but a file which contains a sequence of records, each log record refers to a write operation. All the log records are recorded step by step in the log file. We can say, log files store the history of all updates activities.

Log contains start of transaction, transaction number, record number, old value, new value, end of transaction etc. For example, mini statements in bank ATMs.

If within an ongoing transaction, the system crashes, then by using log files, we can return back to the previous state as if nothing has happened to the database.

The log is kept on disk so that it is not affected by failures except disk and failures.

Example :

Different types of log records are as follows −

<Ti, Xi, V1, V2> − update log record, where Ti=transaction, Xi=data, V1=old

data, V2=new value.

<Ti, start> − Transaction Ti starts execution.

<Ti, commit> − Transaction Ti is committed.

<Ti, abort> − Transaction Ti is aborted

The log records can be written as follows −

Create a log for the given transaction T1 and T2.

T1	T2	Log
Read A	Read A	<T1, start>
A=A-2000	A=A+5000	<T1,A,5000, 3000>
Write A	Write A	<T1, B, 8000, 10000>
Read B	Read B	<T1, commit>
B=B+2000	B= B+7000	<T2, start>
Write B	Write B	<T2, A, 3000, 8000>
		<T2, B, 10000, 17000>
		<T2, commit>

Log Based Recovery

Log-based recovery provides the facility to maintain or recover data if any failure may occur in the system. Log means sequence of records or data, each transaction DBMS creates a log in some stable storage device so that we easily recover data if any failure may occur. When we perform any operation on the database at that time it will be recorded into a log file. Processing of the log file should be done before the original transaction is applied to the database. Why we use log-based recovery the main reason is that the Atomicity property of transaction states that we can either execute the whole transaction or nothing else, modification of the aborted transaction is not visible to the database, and modification of the transaction is visible so that reason we use log-based recovery system.

Syntax:

<TRX, Start>

Explanation: In the above syntax, we use TRX and start in which TRX means transaction and when a transaction in the initial state that means we write start a log.

<TRX, Name, 'First Name', 'Last Name' >

Explanation:In this syntax where TRX means transaction and name is used to First Name and Last Name. When we modify the name First Name to the Last Name then it writes a separate log file for that.

<TRX, Commits>

Explanation:In the above syntax, we use two-variable transactions as TRX and commits, when transaction execution is finished then it is written into another log file that means the end of the transaction we called commits.

Log-Based Recovery in DBMS

Log-based recovery uses the following term for execution as follows.

Transaction Identifier: It used to uniquely identify the transaction.

Data item Identifier: It is used to uniquely identify the used data in the database.

Old Value: It is the value of data before the write operation of a transaction.

New Value: It is the value of data after the write operation of a transaction.

Let's see the transaction with various log types.

First, we start the transaction by using <TRX, Start> this syntax after that we perform the write operation of the transaction that means we update the database. After the write operation, we check whether the transaction is committed or aborted.

For recovery purposes, we use the following two operations as follows.

Undo (TRX): This command is used to restore all records updated by transactions to the old value.

Redo (TRX): This command is used to set the value of all records updated by a transaction to the new value.

Transaction Modification Techniques

There are two different types we use in database medication and that are helpful in the recovery system as follows. Transaction Modification Techniques as follows:

1. Immediate Database Modification

In this type, we can modify the database while the transaction is an inactive state. Data modification done by an active transaction is called an uncommitted transaction. When a transaction is failed or we can say that a system crash at that time, the transaction uses the old transaction to bring the database into a consistent state. This execution can be completed by using the undo operation.

Example:

<TRX1 start>

<TRX1 X, 2000, 1000>

<TRX1 Y, 3000, 1050>

<TRX1 commit>

<TRX2 start>

<TRX2 Z, 800, 500>

<TRX2 commit>

Explanation: In the above example we consider the banking system, the transaction TRX1 is followed by TRX2. If a system crash or a transaction fails in this situation means during recovery we do redo transaction TRX1 and undo the transaction TRX2 because we have both TRX start and commit state in the log records. But we don't have a start and commit state for transaction TRX2 in log records. So undo transaction TRX2 done first the redo transaction TRX1 should be done.

2. Deferred Modification Technique

In this technique, it records all database operations of transactions into the log file. In this technique, we can apply all write operations of transactions on the database if the transaction is partially committed. When a transaction is partially committed at that time information in the log file is used to execute deferred writes. If the transaction fails to execute or the system crashes or the transaction ignores information from the log file. In this situation, the database uses log information to execute the transaction. After failure, the recovery system determines which transaction needs to be redone.

Example

(X) (Y)

<TRX1 start> <TRX1 start>

<TRX1 X, 850><TRX1 X , 850>

<TRX1 Y, 105><TRX1 Y, 1050>

<TRX1 commit>

<TRX2 start>

<TRX 2 Z, 500>

Explanation: If the system fails after write Y of transaction TRX1 then there is no need to redo operation because we have only <TRX1 start> in log record but don't have <TRX1 commit>. In the second transaction Y, we can do the redo operation because we have <TRX1 start> and <TRX1 commit> in log disk but at the same time, we have <TRX2 start> but don't have <TRX2 commit> as shown in the above transaction.

(Z)

<TRX1 start>

<TRX1 X , 850>

<TRX1 Y, 1050>

<TRX1 commit>

<TRX2 start>

<TRX2 Z, 500>

<TRX2 commit>

Explanation: In the above transaction, we have <TRX start> and <TRX commit> in log disk so we can redo operation during the recovery system.

Suppose we need to restore records from binary logs and by default, server creates binary logs. At that time you must know the name and current location of the binary log file, so by using the following statement we can see the file name and location as follows.

show binary logs;

Explanation: In the above statement, we use the show command to see binary logs. Illustrate the final result of the above statement by using the following snapshot.

Example show master status;

Explanation: Suppose we need to determine the current binary log file at that time we can use the above statement.

8.3 RECOVERY RELATED STRUCTURES

Structures Used for Database Recovery: Several structures of an Oracle database safeguard data against possible failures. The following sections briefly introduce each of these structures and its role in database recovery.

Database Backups

A database backup consists of operating system backups of the physical files that constitute an Oracle database. To begin database recovery from a media failure, Oracle uses file backups to restore damaged datafiles or control files.

Oracle offers several options in performing database backups; "Database Backup", for more information.

The Redo Log

The redo log, present for every Oracle database, records all changes made in an Oracle database. The redo log of a database consists of at least two redo log files that are separate from the datafiles (which actually store a database's data). As part of database recovery from an instance or media failure, Oracle applies the appropriate changes in the database's redo log to the datafiles, which updates database data to the instant that the failure occurred.

A database's redo log can be comprised of two parts: the online redo log and the archived redo log, discussed in the following sections.

The Online Redo

Log Every Oracle database has an associated online redo log. The online redo log works with the Oracle background process LGWR to immediately record all changes made through the associated instance. The online redo log consists of two or more preallocated files that are reused in a circular fashion to record ongoing database changes;

The Archived (Offline) Redo Log

Optionally, you can configure an Oracle database to archive files of the online redo log once they fill. The online redo log files that are archived are uniquely identified and make up the archived redo log. By archiving filled online redo log files, older redo log information is preserved for more extensive database recovery operations, while the pre-allocated online redo log files continue to be reused to store the most current database changes;

Rollback Segments

Rollback segments are used for a number of functions in the operation of an Oracle database. In general, the rollback segments of a database store the old values of data changed by ongoing transactions (that is, uncommitted transactions). Among other things, the information in a rollback segment is used during database recovery to "undo" any "uncommitted" changes applied from the redo log to the datafiles. Therefore, if database recovery is necessary, the data is in a consistent state after the rollback

segments are used to remove all uncommitted data from the datafiles; see "Rollback Segments".

Control Files

In general, the control file(s) of a database store the status of the physical structure of the database. Certain status information in the control file (for example, the current online redo log file, the names of the datafiles, and so on) guides Oracle during instance or media recovery

8.4 DIRTY PAGES TABLE

This table is used to represent information about dirty buffer pages during normal processing. It is also used during restart recovery.It is implemented using hashing or via the deferred- writes queue mechanism. Each entry in the table consists of 2 fields :

1. PageID and
2. RecLSN

During normal processing , when a non-dirty page is being fixed in the buffers with the intention to modify , the buffer manager records in the buffer pool (BP) dirty-pages table , as RecLSN , the current end-of-log LSN , which will be the LSN of the next log record to be written. The value of RecLSN indicates from what point in the log there may be updates. Whenever pages are written back to nonvolatile storage , the corresponding entries in the BP dirty-page table are removed. The contents of this table are included in the checkpoint record that is written during normal processing. The restart dirty-pages table is initialized from the latest checkpoint's record and is modified during the analysis of the other records during the analysis pass. The minimum RecLSN value in the table gives the starting point for the redo pass during restart recovery.

ARIES maintains two data structures and adds one more field to log record:

1. **Transaction table**: It contains all the transactions that are active at any point of time (i.e. are started but not committed/aborted). The table also stores the LSN of last log record written by the transaction in “lastLSN” field.

2. **Dirty page table**: Contains an entry for each page that has been modified but not written to disk. The table also stores the LSN of the first log record that made the associated page dirty in a field called “recoveryLSN” (also called “firstLSN”). This is the log record from which REDO need to restart for this page.

3. In addition log records are also updated to contain a field called “prevLSN” which points to previous log record for the same transaction. This creates a linked list of all log records

for a transaction. When a new log record is created, “lastLSN” from transaction table is filled into its “prevLSN” field. And the LSN of current log record becomes the “lastLSN” in transaction table. Here is updated log record table with prevLSN filled in:

4.

LSN	Prev LSN	Transaction ID	Type	Page ID
1	NIL	T1	UPDATE	P3
2	NIL	T2	UPDATE	P2
3	1	T1	COMMIT	
4	CHECKPOINT			
5	NIL	T3	UPDATE	P1
6	2	T2	UPDATE	P3
7	6	T2	COMMIT	

During checkpointing, a checkpoint log record is created. This log record contains the content of both “Transaction table” and “Dirty page table”. “Analysis” phase starts by reading last checkpoint log record to get the information about active transactions and dirty pages. Here is content of “Transaction table” and “Dirty page table” at the checkpoint stage in above table at LSN 4:

Transaction Table

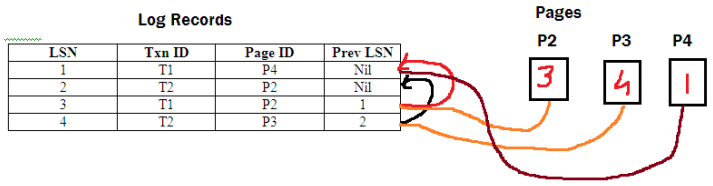
Transaction ID	Last LSN	Status
T1	3	Commit
T2	2	In Progress

Dirty Page Table

Page ID	Recovery LSN
P3	1
P2	2

This whole setup can be visualized in following picture, pay attention to LSN for P2 in the “pages” list and in dirty table (dirty page table has the first LSN, whereas the P2 page has the last LSN):

Log Records :



Transaction Table

Txn ID	Last LSN
T1	3
T2	4

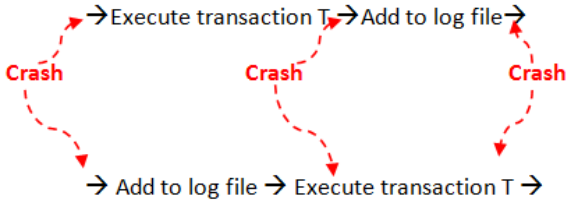
Dirty Page Table

Page ID	LSN
P1	2
P2	4
P3	1

ARIES Data Structures

8.5 WRITE-AHEAD LOG IN DBMS

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.

But in this protocol, we have I/O access twice – one for writing the log and another for writing the actual data. This is reduced by keeping the log buffer in the main memory – log files are kept in the main memory for certain pre-defined time period and then flushed into the disk. The log files are appended with data for certain period, once the buffer is full or it reaches the time limit, then it is written into the disk. This reduces the I/O time for writing the log files into the disk.

Similarly retrieving the data from the disk is also needs I/O. This can also be reduced by maintaining the data in the page cache of the main memory. That is whenever a data has to be retrieved; it will be retrieved from the disk for the first time. Then it will be kept in the page cache for the future reference. If the same data is requested again, then it will be retrieved from this page cache rather than retrieving from the disk. This reduces the time for retrieval of data. When the usage / access to this data reduce to some threshold, then it will be removed from page cache and space is made available for other data.

8.6 CHECKPOINT

1. The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.

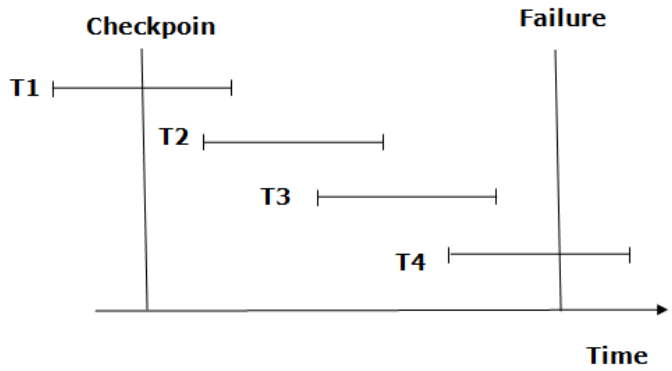
2. The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.

3. When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.

4. The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- For example: In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.

For example: Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

8.7 RECOVERY FROM SYSTEM CRASH

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

Examples may include operating system errors.

Computers crash for a variety of reasons. Random computer crashes are both frustrating and difficult for an average user to diagnose, but underneath the surface of a computer crash are five likely culprits examined below.

1: Corrupted System Registry Files

Every Windows-based PC has something called a Windows registry. The registry contains several files that are integral to the performance and operation of your computer. Over time, some of those files can become corrupted, be misplaced or get lost altogether. When that happens, the system registry becomes compromised – and frequent crashes are all-too-common symptoms. The best way to rule this possibility in or out is by running a Windows registry cleaning program. Such programs scan your Windows registry for problems then automatically make repairs. If you run a registry cleaner and the crashes persist, they are probably being caused by a different issue.

2: Disorganized Files

Windows operating systems handle file organization in a way that isn't very intuitive. Basically, they break files up and fit them into gaps in the computer's memory. As time goes by, these disorganized files can prompt frequent crashes. Luckily, a great optimization solution is built right into Windows-based PCs: the disk defragmentation utility. Although its location on a computer varies, you can generally locate it within the System and Security section inside the Control Panel. By running a defrag once every few months, you may be able to keep those pesky computer crashes at bay.

3: Malicious Software

Malicious software can take many different forms. Sometimes, it's a virus that is accidentally unleashed after opening a strange email; other times, its adware that tags along with other information that is automatically downloaded from a website. Whatever type it is, there's no question that malicious software can wreak havoc on a

computer's performance. Happily, there are many topnotch programs out there that regularly scan your computer for the presence of such problems – and that help guard against them, too. Buy one, install it and use it regularly; your crash issues may come to an end.

4: Too Little Available Memory

When you buy a new computer, it feels like there's no end to the amount of memory that it has. Of course, this isn't true at all. As never-ending as the available memory on your PC may initially seem, the fact is that it can be depleted with incredible speed. You can find out for sure by checking the information within "My Computer." If it appears that your available memory is low, you can use a PC cleanup program to remove unnecessary files; such programs remove things like temporary Internet files and other file debris that can suck away much-needed memory.

5: Overheating

If you've run through all of the preceding possibilities and continue experiencing frequent crashes, a hardware issue could be to blame. An easy one to rule out is overheating. A computer's CPU, or central processing unit, includes a fan that is designed to keep it running cool. Sometimes, the fan wears down and doesn't work as efficiently; other times, it's just not able to handle the work that your computer has to do. In either case, buying a bigger, better fan isn't very expensive. If it puts an end to your PC crashing problem, it will have been more than worth it.

8.8 REDO PHASE UNDO PHASE

REDO PHASE:-

1. Redo phase is the second phase where all the transactions that are needed to be executed again take place.
2. It executes those operations whose results are not reflected in the disk.
3. It can be done by finding the smallest LSN of all the dirty page in dirty page table that defines the log positions, & the Redo operation will start from this position
4. This position indicates that either the changes that are made earlier are I the main memory or they have already been flaunted to the disk.
5. Thus, for each change recorded in the log, the Redo phase determines whether or not the operations have been re-executed.

UNDO PHASE:-

1. In the Undo phase, all the transaction, that is listed in the active transaction set here to be undone.
2. Thus the log should be scanned background from the end & the recovery manages should Undo the necessary operations.
3. Each time an operations is undone, a compensation log recorded has been written to the log.
4. This process continues until there is no transaction left in the active transaction set.
5. After the successful competition of this phase, database can resume its normal operations.

.