ADITYA ENGINEERING COLLEGE (A)

DBMS

TRANSACTIONS

By

Dr. S Rama Sree
Professor in CSE & Dean(Academics)
Aditya Engineering College(A)
Surampalem.

Contents

- Transaction State
- Implementation of Atomicity and Durability,
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Testing for Serializability
- Failure Classification
- Storage
- Recovery and Atomicity
- Recovery algorithm.

TRANSACTIONS

- A **transaction** is the DBMS's abstract view of a user program :

  a sequence of reads and writes.

- A transaction is a unit of program execution that accesses and possibly updates various data items.

- Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java), where it is delimited by statements (or function calls) of the form begin transaction and end transaction.

- The transaction consists of all operations executed between the begin transaction and end transaction.

- Collections of operations that form a single logical unit of work are called transactions.
- A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does.
- To ensure integrity of the data, we require that the database system maintain the following 4 properties of the transactions: **ACID Properties**

ACID PROPERTIES

- **Atomicity.** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency.** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation.** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj , it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

- Transactions access data using two operations:
  - **read(X),** which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
  - **write(X),** which transfers the data item X from the the local buffer of the transaction that executed the write, back to the database

ACID PROPERTIES

ATOMICITY:

1. Transactions should be **atomic**. Either they happen or they don't happen at all.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of T1 but before completion of T2.(say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y.

CONSISTENCY

2. Each transaction, run by itself, alone, should preserve the **consistency** of the database. The DBMS assumes that consistency holds for each transaction.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

The total amount before and after the transaction must be maintained.

Total Amount in A & B before T occurs = 500 + 200 = 700.
Total after T occurs = 400 + 300 = 700.
Therefore, database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.

ISOLATION

3. **Isolation**: Transactions are isolated from the effect of other transactions that might be executed concurrently

Let

X=500, Y=500

Consider two transactions T and T".

Suppose T has been executed till Read (Y) and then T'' starts. As a result , interleaving of operations takes place due to which T'' reads correct value of X but incorrect value of Y and sum computed by

T'': (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of transaction:

T: (X+Y = 50, 000 + 450 = 50, 450).

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

DURABILITY

4. Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

- Once the user is notified that the transaction was successful, its effects should persist even if the system crashes.

Ex:- Transaction updates a data in the database & commits, then the database will hold the modified data.

If a transaction updates a data in the database and system fails before commit, data will be updated once the system springs back to the action.

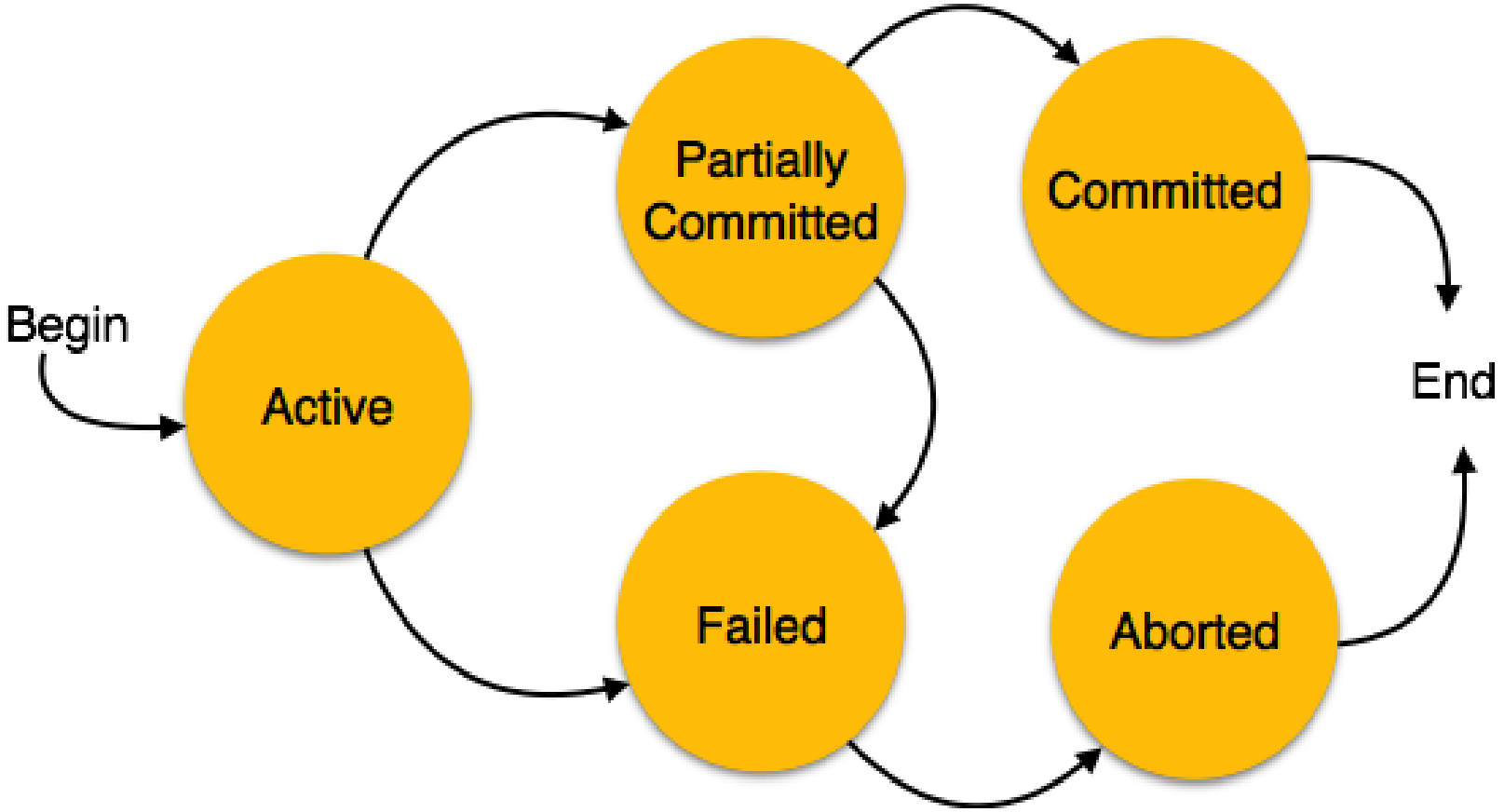
DBMS maintains a record called log that maintains all writes to the database. Log is used to ensure durability. If system crashes before the changes made by a transaction are written to disk, the log is used to remember & restore those changes when the system restarts. Component that ensures atomicity & durability called recovery manager.

TRANSACTION STATES

A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing
- **Partially committed**, after the final statement has been executed
- **Failed,** after the discovery that normal execution can no longer proceed
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- **Committed**, after successful completion.

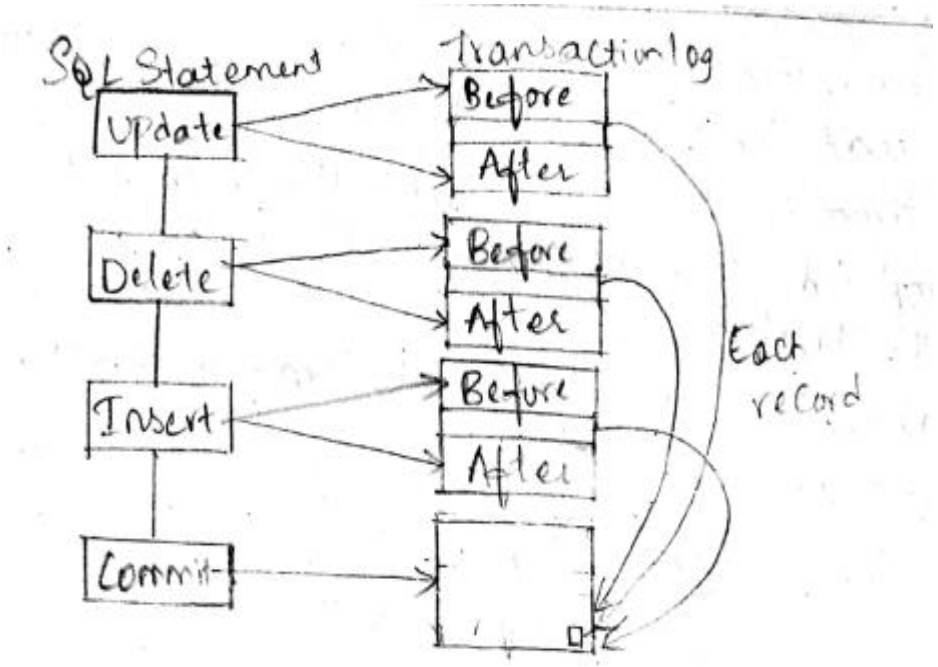
TRANSACTION STATES



- In the absence of failures, all transactions complete successfully. Any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back.**
- A transaction starts in the **active state**.
- When it finishes its final statement, it enters the **partially committed** state. At this point, the transaction has completed its execution, but it is still possible that it may have to be **aborted**, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.
- A transaction may not always complete its execution successfully. Such a transaction is termed **aborted**.
- A transaction may enter the **failed state** after the system determines that the transaction can no longer proceed with its normal execution. Such a transaction must be rolled back. Then, it enters the **aborted state**. At this point, the system has two options:
  - It can restart the transaction
  - It can kill the transaction.
- A transaction that completes its execution successfully is said to be **committed.**

TRANSACTION LOG

- It is a record that contains data before the operation & data after the operation.
- To recover from failures, system maintains a log to keep track of all transaction operations that affect values of database items.



TRANSACTION LOG

- Log is kept on disk, so it is not affected by any type of failure except for disk.

  Log is periodically backed up to archive storage(tape).

- Execute each SQL statement, its resulting pre and post state information is copied to the transaction log.

  Insert/ Update/Delete causes a before/after state

- When commit statement is finally encountered, then the data is written to the disk.

Log maintains the below records.

- T refers to a unique transaction-id that is generated automatically by the system & used to identify each transaction.
- [Start Transaction,T]: indicates transaction T has started execution.
- [Write_item,T,X, Old value, New Value]: indicates transaction T has changed the write value of database item X from old value to new value.
- [read_item, T,X]: indicates the transaction T read the value of database item X.
- [commit, T]:indicates the transaction T has completed successfully, its effect can be committed to the database.
- [abort,T]: indicates the transaction T has been aborted.

Concurrency Executions

•Transaction-processing systems usually allow multiple transactions to run concurrently.

•Allowing multiple transactions to update data concurrently causes inconsistency.

• Ensuring consistency inspite of concurrent execution of transactions requires extra work;

•It is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed.

• However, there are two good reasons for allowing concurrency
  - Improved throughput and resource utilization
  - Reduced waiting time

- A **schedule** is a list of actions (read, write, abort, commit) for a set of transactions, and the order in which they happen.
- There are 2 types of Schedules
  1) Serial Schedule
  2) Concurrent Schedule
- **Serial Schedule**: In serial schedule, the instructions belonging to one single transaction execute before the start of execution of another together in that schedule. Thus, for a set of n transactions, there exist n! different valid serial schedules.
- **Concurrent Schedule:** If the transactions are executed concurrently, the schedule is called Concurrent schedule.

Ex: Consider the sequence of execution of T1 & T2. Let A=1000 & B= 2000

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 1—a serial schedule in which T_1 is followed by T_2.

Output: After execution of T1 & T2 in order, A=855 and B=2145

Output: After execution of T2 & T1 in order, A=850 and B=2150

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 2—a serial schedule in which T_2 is followed by T_1.

These schedules are **serial**:

- When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial.
- If two transactions are running concurrently, the OS may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.
- With multiple transactions, the CPU time is shared among all the transactions. Several execution sequences are possible

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 3—a concurrent schedule equivalent to schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

15.6 Schedule 4—a concurrent schedule.

•A transaction is seen by the DBMS as a **series** (or list) of **actions** These actions are **reads** or **writes** of an object: R_T **(**O**)**, W_T **(**O**)** In addition to reading and writing, a transaction should specify **commit** or **abort** at the end: $Commit_T$, $Abort_T$.

•Assumptions:

- •Transactions only interact with each other through reads/writes
- •A database is a $fixed$ collection of $independent$ objects

SERIALIZABILITY

- The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.
- A **serializable schedule** of a set of S transactions is a schedule identical to a serial schedule of the same set of transactions.
- Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.
- There are 2 types of Serializable Schedules
  1) Conflict Serializability
  2) View Serializability
- We will consider only read &
- write instructions for schedule

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3—showing only the read and write instructions.

CONFLICT SERIALIZABILITY

- Let us consider a schedule S in which there are two consecutive instructions Ii and Ij , of transactions Ti and Tj , respectively.
- if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases :

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.
2. Ii = read(Q), Ij = write(Q). If Ii comes before Ij , then Ti does not read the value of Q that is written by Tj in instruction Ij . If Ij comes before Ii, then Ti reads the value of Q that is written by Tj . Thus, the order of Ii and Ij matters.
3. Ii = write(Q), Ij = read(Q). The order of Ii and Ij matters for reasons similar to those of the previous case.
4. Ii = write(Q), Ij = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

- Ii and Ij **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

ANOMALIES

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Concurrency can leave to an inconsistent state

Two actions in the same object conflict iF at least one is a write. Therefore,

3 types of anomalies (assume transactions T_1, T_2) exists

- **Write-Read WR conflict**: T_2 reads data previously written by T_1
- **Read-Write RW conflict**: T_2 writes data to something previously read by T_1
- **Write-Write WW conflict**: T_2 writes data to something previously written by T_1

Aditya Engineering College (A)

WR Conflict

T_2 reads data that has not been committed yet

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

This situation is called a **dirty read**.

RW Conflict

T_2 changes the value of an object already read by T_1

If T_1 tries to read it again, then it will be different Called

unrepeatable read

WW Conflict

T_2 overwrites the value of an object A, already modified by T_1, while T_1 is still in progress

T_1	T_2
W(A)	
	W(A)
	W(B)
W(B)	
Commit	
	Commit

Writes that don't read the object are called **blind writes**

VIEW SERIALIZABILITY

- View serializability is **a concept that is used to compute whether schedules are View-Serializable or not**.
- A schedule is said to be View-Serializable if it is view equivalent to a Serial Schedule (where no interleaving of transactions is possible).

RECOVERABILITY

- If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction.
- In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted.
- To achieve this surety, we need to place restrictions on the type of schedules permitted in the system. There are 2 types of schedules: recoverable schedules & nonrecoverable schedules
- A **recoverable schedule** is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the commit operation of Tj .
- Otherwise, If transaction operation is committed and cannot be aborted, such a schedule is called **nonrecoverable schedule**.

Recoverable Schedules

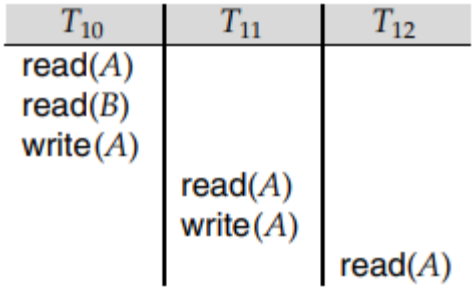
- Consider the schedule, in which T9 is a transaction that performs only one instruction: read(A).
- Suppose that the system allows T9 to commit immediately after executing the read(A) instruction. Thus, T9 commits before T8 does. Now suppose that T8 fails before it commits.
- Since T9 has read the value of data item A written by T8, we must abort T9 also to ensure transaction atomicity. However, T9 has already committed and cannot be aborted.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- Thus, we have a situation where it is impossible to recover correctly from the failure of T8.
- The Schedule, with the commit happening immediately after the read(A) instruction, is an example of a nonrecoverable schedule, which should not be allowed.
- Most database system require that all schedules be recoverable

Cascadeless Schedules

- Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti, we may have to roll back several transactions.
- The phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.
- Transaction T10 writes a value of A that is read by transaction T11. Transaction T11 writes a value of A that is read by transaction T12. Suppose that, at this point, T10 fails. T10 must be rolled back. Since T11 is dependent on T10, T11 must be rolled back. Since T12 is dependent on T11, T12 must be rolled back.



Implementation of Isolation

A concurrency control scheme- a transaction acquires a lock on the entire database before it starts and releases the lock after it has committed.

A transaction holds the lock, no other transaction is allowed to acquire the lock, all must wait for the lock to be released.

Only one transaction can execute at a time. so, serial schedules are generated.

A concurrency control scheme leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. It provides poor degree of concurrency.

Goal of concurrency control scheme is to provide a high degree of concurrency, ensuring that all schedules that can be generated are conflict or view and cascadeless.

Testing for Serializability

How to determine a schedule S is Serializable?

A simple and efficient method for determining conflict serializability of a schedule is by using a precedence graph.

A directed graph called precedence graph consists of a pair G=(V,E)

V-set of vertices, E- set of edges

Set of vertices consists of all the transactions participating in the schedule.

Set of edges consists of all edges Ti -> Tj where one of the condition holds

i)Ti executes write(Q) before Tj executes read(Q)

ii)Ti executes read(Q) before Tj executes write(Q)

iii)Ti executes write(Q) before Tj executes write(Q)

Testing for Serializability

a) Precedence Graph for Schedule-1

b) Precedence Graph for Schedule-2

c) Precedence Graph for Schedule-4

Testing for Serializability

Schedule1 contains edge T1->T2 T1 executes R(A) before T2 executes W(A)

Schedule2 contains edge T2->T1 T2 executes R(B) before T1 executes W(B)

If the precedence graph for S has a cycle then schedule S is not conflict serializable. If graph contains no cycle, then schedule S is conflict serializable.

Ex:- schedule 1 and 2 donot contain cycles.

schedule 4 contains a cycle, schedule is not conflict serializable.

Failure Classification

There are various types of failure that may occur in the system.

Simplest type of failure is one that doesn't result in the loss of information in the system.

Failures that are more difficult to deal with are those that result in loss of information. There are 3 types of failures:

1. Transaction Failure:- Two types of failures that may cause a transaction failure

Logical error:- transaction can no longer continue with its normal execution because of internal condition such as bad input, data not found, overflow, resource limit exceeded etc.

System error:- System has entered an undesirable state, as a result transaction cannot continue with its normal execution.

Failure Classification

2. System Crash:- Hardware malfunction or bug in the database software or OS that causes loss of content of volatile storage & brings transaction processing to a halt.

Assumption that hardware errors & bugs in the software bring the system to a halt, but don't corrupt the non volatile contents- **fail stop assumption**.

3. Disk failure:- A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks or tapes are used to recover from the failure. These recovery algorithms have 2 parts.

1)Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.

2)Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity & durability.

Storage Types

There 3 different types of Storage:

1. Volatile Storage:- Information residing in volatile storage doesn't survive system crashes. Ex:- Main memory & Cache Memory. Access to volatile storage is fast. It is possible to access any data item in volatile storage directly.

2. Non Volatile Storage:- Information residing in non-volatile storage Survives System Crashes

Ex:- Disk, Magnetic Tapes

Disks used for online storage.

Tapes used for archival storage. Both are subject to failure, may result in loss of information.

Non volatile is slower than volatile storage. Disk and tape devices are electromechanical, rather than based on chips, as is volatile storage.

3. Stable Storage:- Information residing in stable storage is never lost. Eg: RAID

Recovery & Atomicity

Consider the example where Transaction Ti that transfers $50 from Account A to B. Initial Values of A and B are $1000 and $2000. System crash has occurred during execution of Ti and the memory contents were lost.

Two Recovery Procedures are possible:

1)Reexecute Ti

A becoming $900 rather than $950. System enters into an inconsistent state.

2) **Donot Reexecute Ti**

Current values $950 and $2000 for A and B. System enters an inconsistent state.

When a DBMS recovers from a crash, it should maintain the following − □ It should check the states of all the transactions, which were being executed.

A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.

It should check whether the transaction can be completed now or it needs to be rolled back.

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

Crash Recovery

- What happens when the database crashes?
- Remember ACID
- We need to guarantee that committed transactions survive a system crash or a media failure
- The **recovery manager** (RM) is responsible for ensuring this
- It is one of the most difficult parts to implement .
- After the DBMS is restarted after a crash, control is given to the RM.
- The RM is also responsible to undo uncommitted transactions

Recovery Algorithm

ARIES is a recovery algorithm, conceptually simpl.

•ARIES recovers from a system crash in three passes:

1. **Analysis pass:** This pass determines which transactions to undo, which pages were dirty at the time of crash, and the log sequence number(LSN) from which the redo pass should start.
2. **Redo pass:** This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before crash.
3. **Undo pass:** This pass rolls back all transactions that were incomplete at the time of crash.

References

Text Books:

- Database Management Systems, 3/e, Raghurama Krishnan, Johannes Gehrke, TMH.
- Database System Concepts,5/e, Silberschatz, Korth, TMH.

Reference Books:

- Introduction to Database Systems, 8/e C J Date, PEA.
- Database Management System, 6/e Ramez Elmasri, Shamkant B. Navathe, PEA.
- Database Principles Fundamentals of Design Implementation and Management, Corlos Coronel, Steven Morris, Peter Robb, Cengage Learning.

Web Links:

- https://nptel.ac.in/courses/106/105/106105175/
- https://www.geeksforgeeks.org/introduction-to-nosql/
- https://www.youtube.com/watch?v=wkOD6mbXc2M
- https://beginnersbook.com/2015/05/normalization-in-dbms/

Questions???