Database Transactions

Tushar B. Kute,
http://tusharkute.com

What is Database?

- A database is an organized collection of data, so that it can be easily accessed and managed.
- You can organize data into tables, rows, columns, and index it to make it easier to find relevant information.
- Database handlers create a database in such a way that only one set of software program provides access of data to all the users.
- The main purpose of the database is to operate a large amount of information by storing, retrieving, and managing data.

What is Database Management System?

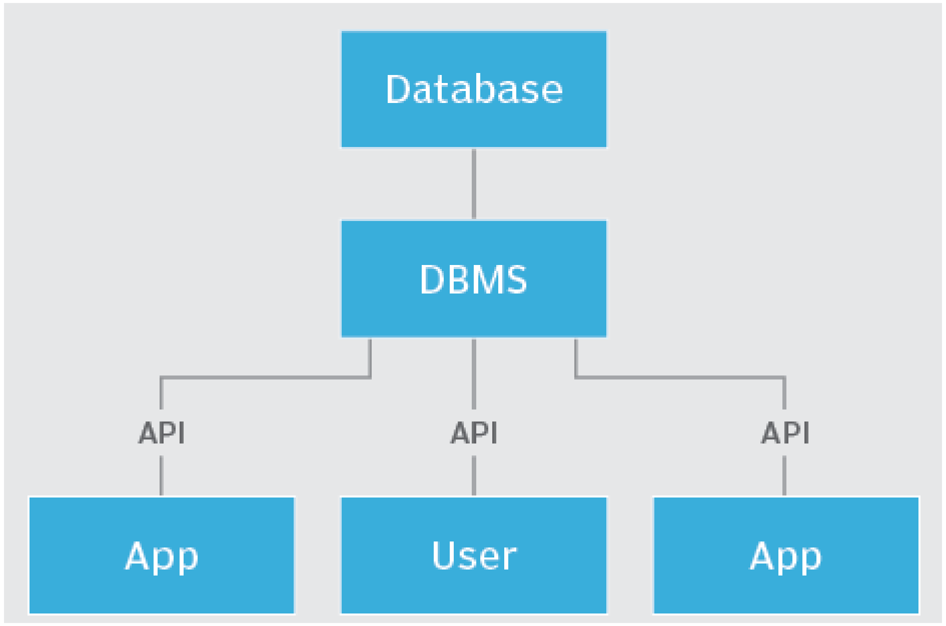
- A database management system (DBMS) is system software for creating and managing databases.
- A DBMS makes it possible for end users to create, protect, read, update and delete data in a database.
- The most prevalent type of data management platform, the DBMS essentially serves as an interface between databases and users or application programs, ensuring that data is consistently organized and remains easily accessible.

What DBMS do?

- The DBMS manages the data; the database engine allows data to be accessed, locked and modified; and the database schema defines the database's logical structure.
- These three foundational elements help provide concurrency, security, data integrity and uniform data administration procedures.
- The DBMS supports many typical database administration tasks, including change management, performance monitoring and tuning, security, and backup and recovery.
- Most database management systems are also responsible for automated rollbacks and restarts as well as logging and auditing of activity in databases and the applications that access them.

DBMS Components

- A DBMS is a sophisticated piece of system software consisting of multiple integrated components that deliver a consistent, managed environment for creating, accessing and modifying data in databases



Storage Engine

- This basic element of a DBMS is used to store data. The DBMS must interface with a file system at the operating system (OS) level to store data.
- It can use additional components to store data or interface with the actual data at the file system level.

What is SQL?

- SQL is the standard language for dealing with Relational Databases.
- SQL can be used to insert, search, update, and delete database records. SQL can do lots of other operations, including optimizing and maintenance of databases.
- SQL Full Form
  - SQL stands for Structured Query language, pronounced as "S-Q-L" or sometimes as "See-Quel"… Relational databases like MySQL Database, Oracle, MS SQL Server, Sybase, etc. use ANSI SQL.

MySQL

- MySQL is released under an open-source license. So you have nothing to pay to use it.
- MySQL is a very powerful program in its own right. It handles a large subset of the functionality of the most expensive and powerful database packages.
- MySQL uses a standard form of the well-known SQL data language.
- MySQL works on many operating systems and with many languages including Python, PHP, PERL, C, C++, JAVA, etc.
- MySQL works very quickly and works well even with large data sets.

MySQL

- MySQL works very quickly and works well even with large data sets.
- MySQL is very friendly to PHP, the most appreciated language for web development.
- MySQL supports large databases, up to 50 million rows or more in a table. The default file size limit for a table is 4GB, but you can increase this (if your operating system can handle it) to a theoretical limit of 8 million terabytes (TB).
- MySQL is customizable. The open-source GPL license allows programmers to modify the MySQL software to fit their own specific environments.

Types of SQL Statements

- Here are five types of widely used SQL queries.
  - Data Definition Language (DDL)
  - Data Manipulation Language (DML)
  - Data Control Language (DCL)
  - Transaction Control Language (TCL)
  - Data Query Language (DQL)

MySQL Clients

- MYSQL Client are programs for communicating with the server to manipulate the information in the databases that the server manages.
- Example : mysql is the command line program that acts as a text-based front end for the server.

MySQL Workbench

- MySQL Workbench is a unified visual tool for database architects, developers, and DBAs.
- MySQL Workbench provides data modeling, SQL development, and comprehensive administration tools for server configuration, user administration, backup, and much more.
- MySQL Workbench is available on Windows, Linux and Mac OS X.

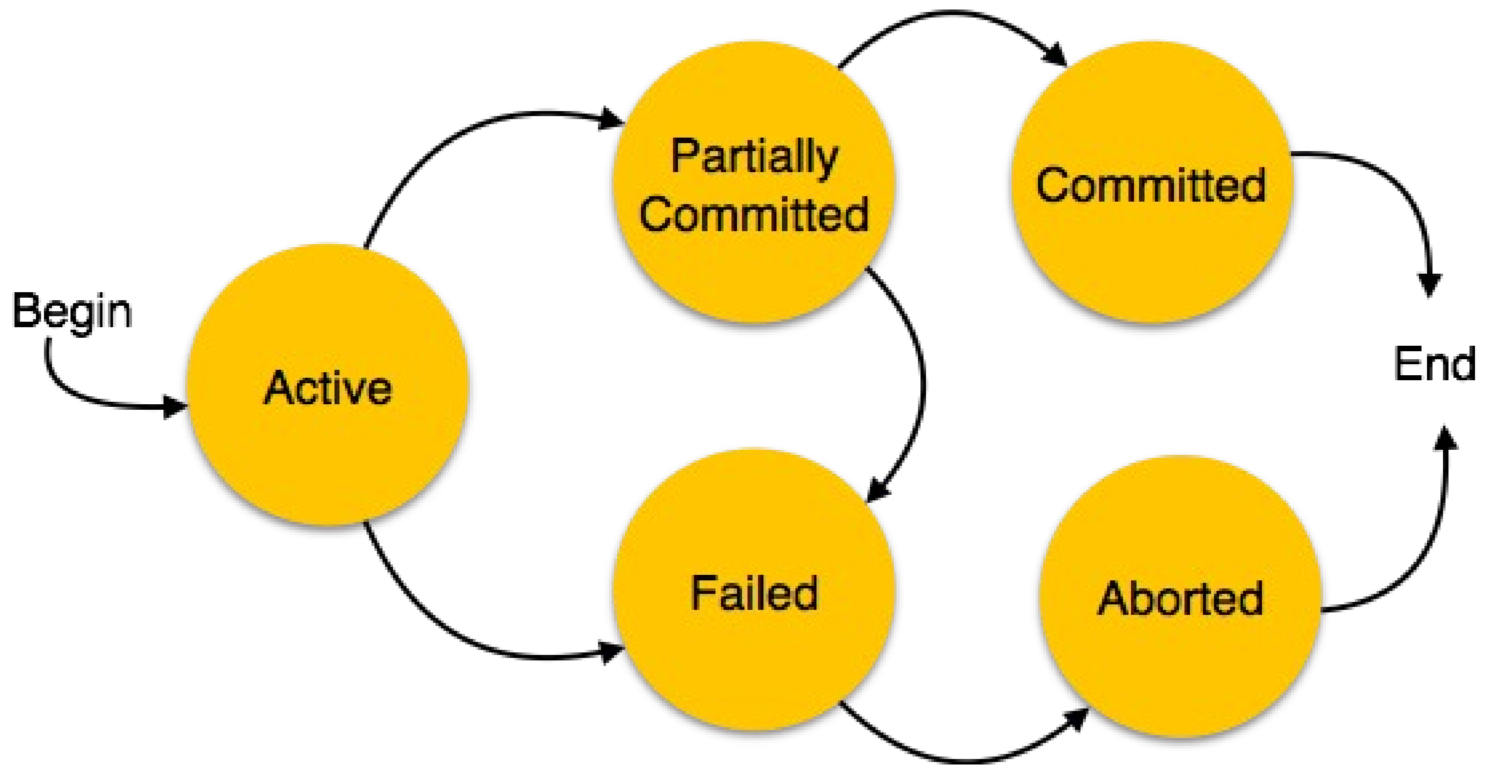
MySQL Shell

- The MySQL Shell is an interactive Javascript, Python, or SQL interface supporting development and administration for the MySQL Server and is a component of the MySQL Server.
- You can use the MySQL Shell to perform data queries and updates as well as various administration operations.

Database Transaction

- A database transaction is a sequence of operations or tasks performed on a database that are treated as a single unit.
- A transaction ensures that the database remains in a consistent and reliable state, even in the event of a system failure, crash, or interruption during the operation.
- Transactions are fundamental to database management systems (DBMS) because they ensure data integrity, consistency, and reliability.
- A transaction involves operations like INSERT, UPDATE, DELETE, or SELECT on the database.

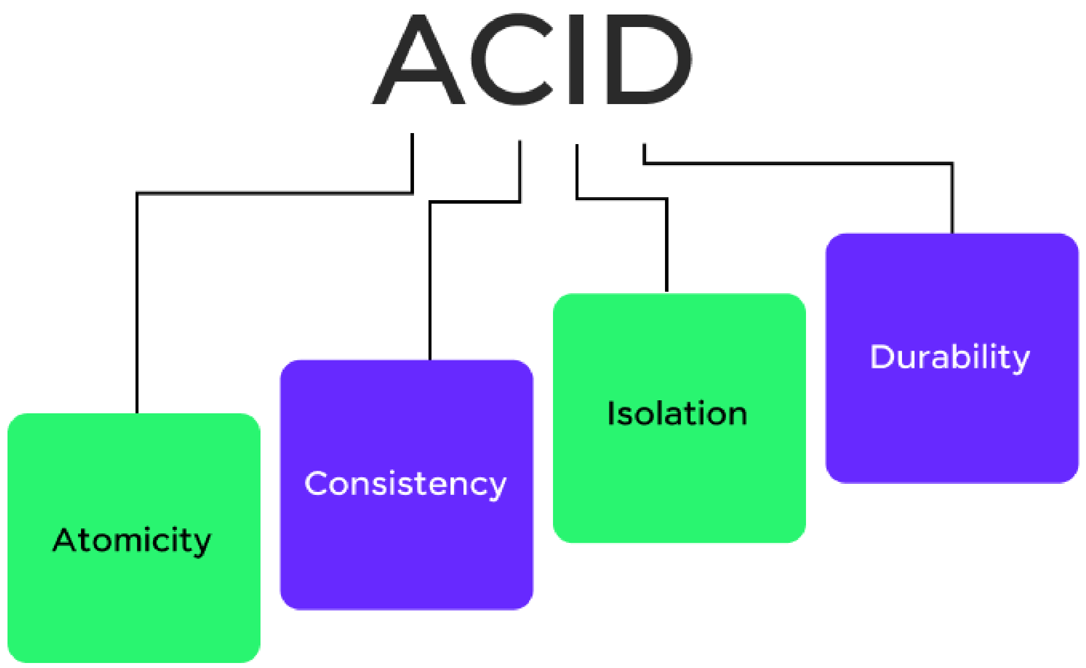
Database Transaction



Database Transaction

- A database transaction is considered to be atomic, meaning that it either completes entirely or does not happen at all.
- This is critical to maintaining the ACID properties (Atomicity, Consistency, Isolation, and Durability) of a transaction.

ACID



ACID

- Atomicity:
  - A transaction is indivisible. It either completes entirely or has no effect at all. If one part of the transaction fails, the whole transaction is rolled back.
- Consistency:
  - A transaction brings the database from one valid state to another valid state. All integrity constraints are maintained during the transaction, and the database is consistent before and after the transaction.

ACID

- Isolation:
  - Transactions are isolated from each other. Even though multiple transactions might be executed simultaneously, each transaction should not affect the others. Intermediate states of a transaction are not visible to other transactions until the transaction is committed.
- Durability:
  - Once a transaction is committed, its changes are permanent, even if there is a system failure. The data changes made by the transaction are saved to the database and are not lost.

Real World Example

- Consider a banking system where you want to transfer money from Account A to Account B. The transaction would involve two steps:
  - Subtracting money from Account A (e.g., debit).
  - Adding money to Account B (e.g., credit).
- This entire operation is a single transaction, and both operations need to be completed successfully.
- If the system fails or crashes after the debit but before the credit, the transaction should rollback, ensuring that the money is neither deducted from Account A nor added to Account B.
- If the transaction completes successfully, the changes are committed, and both accounts reflect the new balances.

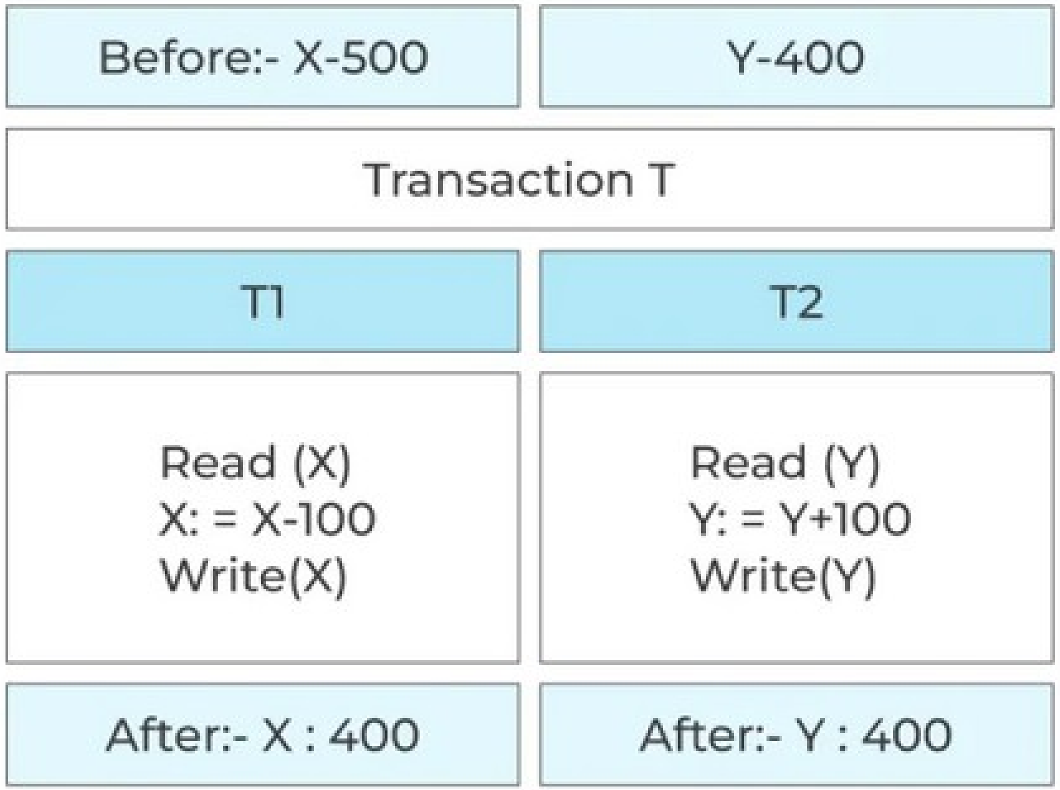
Atomicity

- Atomicity ensures that a transaction is an all-or-nothing operation.
- This means that either all the operations within the transaction are completed successfully, or if any part of the transaction fails, the whole transaction is rolled back, and the database remains unchanged.

Atomicity

- Example: Imagine you are transferring money between two bank accounts:
  - Account A has $500.
  - Account B has $300.
- You want to transfer $100 from Account A to Account B.
- Steps in the Transaction:
  - Subtract $100 from Account A.
  - Add $100 to Account B.
- If the system crashes after the first step (subtracting from Account A) but before the second step (adding to Account B), atomicity ensures that the transaction is rolled back, and Account A will not be deducted $100. Both accounts will remain in their original state, preventing an inconsistent database.

Atomicity



Before:- X-500	Y-400
Transaction T	
T1	T2
Read (X) X: = X-100 Write(X)	Read (Y) Y: = Y+100 Write(Y)
After:- X : 400	After:- Y : 400

Transaction Example

- START TRANSACTION;
- -- Operation 1: Deduct $100 from Account A
- UPDATE accounts SET balance = balance - 100 WHERE account_id = 'A';
- -- Operation 2: Add $100 to Account B
- UPDATE accounts SET balance = balance + 100 WHERE account_id = 'B';
- -- If both operations are successful, commit the transaction
- COMMIT;

Consistency

- Consistency ensures that a transaction takes the database from one valid state to another valid state, adhering to all predefined rules, constraints, and triggers.
- After the transaction, the database must remain consistent and satisfy all integrity constraints, like foreign keys, unique constraints, and more.

Consistency

- Example: Let's consider a database with a Students table and an Enrollment table.
  - Students Table: Contains student data (e.g., student_id, name, balance).
  - Enrollment Table: Contains course enrollment data (e.g., student_id, course_id).
- A student has a balance of $500, and the rule is that a student must have a balance greater than $100 to enroll in a course.
- Transaction Example:
  - Check the student’s balance: Ensure the student has enough balance to enroll.
  - Enroll the student in a course: Deduct the course fee from the student's balance.
  - Commit: If the balance is valid and the enrollment is successful, commit the transaction.
- If the student’s balance is insufficient, the transaction is rolled back to ensure the integrity of the database.

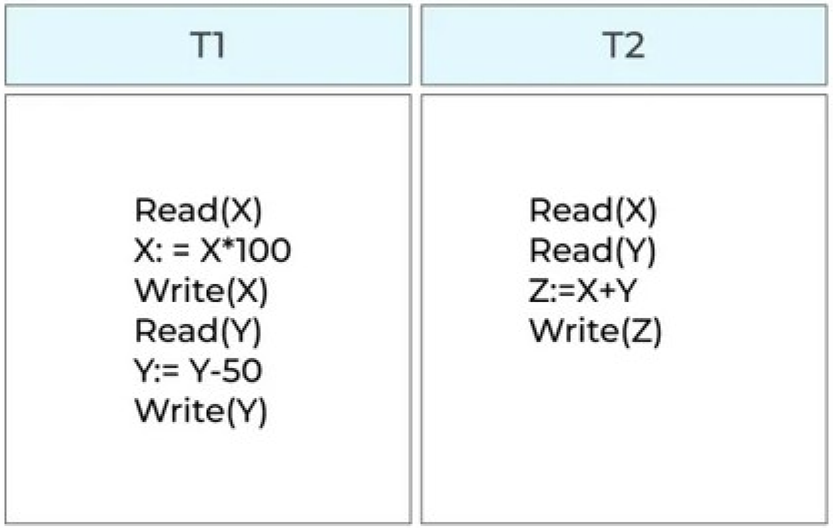
Isolation

- Isolation ensures that concurrent transactions do not interfere with each other.
- Each transaction should execute as if it is the only transaction in the system, meaning that intermediate results of one transaction are invisible to other transactions until the transaction is committed.

Isolation

- Example: Consider two transactions happening simultaneously:
  - Transaction 1: A bank account transfer is happening from Account A to Account B.
  - Transaction 2: Another user is viewing the account balance of Account A.
- Scenario: Both transactions should not see inconsistent or intermediate results of each other. For instance, Transaction 2 should not see the balance of Account A as if the money was already deducted when Transaction 1 is still in progress.
  - Transaction 1: Subtract money from Account A and add money to Account B.
  - Transaction 2: Check the balance of Account A.
- In an isolated environment, Transaction 2 should either see the balance before Transaction 1 starts or after it finishes — not an inconsistent intermediate balance.
- This is controlled by different isolation levels (Read Uncommitted, Read Committed, Repeatable Read, Serializable). The higher the isolation level, the less interference there is between transactions.

Isolation



T1	T2

Read(X)
X: = X*100
Write(X)
Read(Y)
Y:= Y-50
Write(Y)
	Read(X)
	Read(Y)
	Z:=X+Y
	Write(Z)

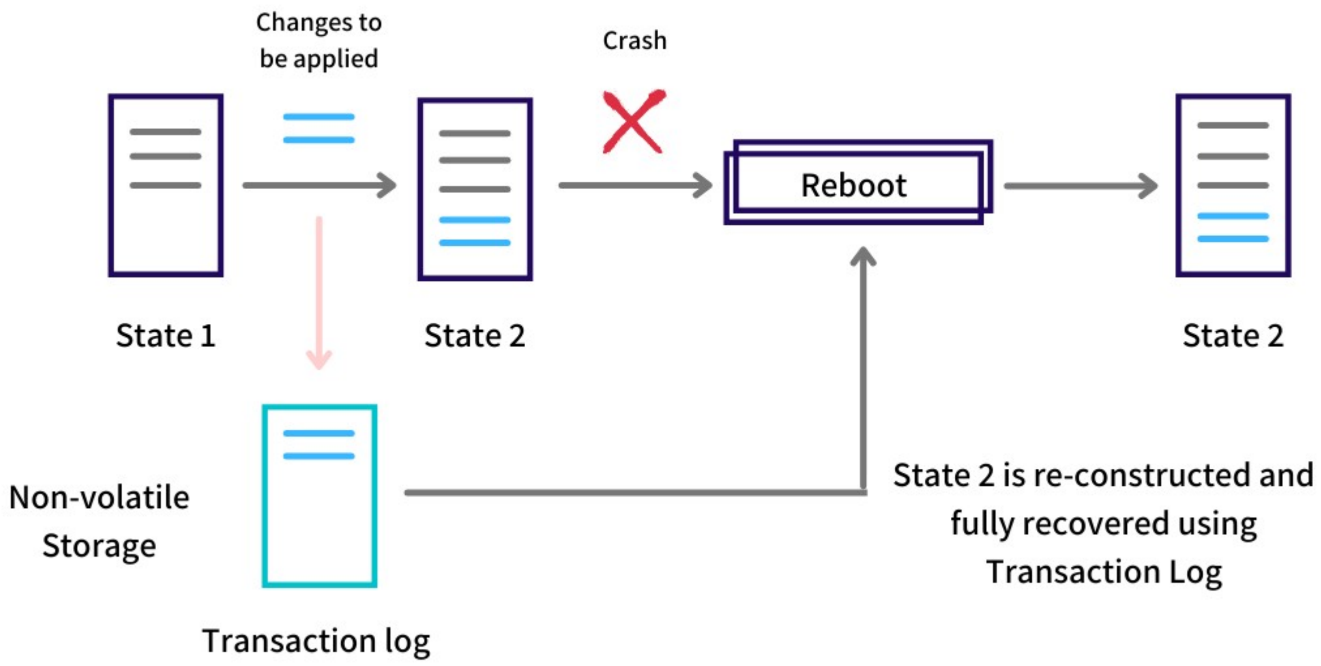
Durability

- Durability ensures that once a transaction is committed, its changes are permanent and survive system failures, such as crashes, power outages, or hardware failures.
- The changes made by the transaction are stored in the database and are guaranteed not to be lost.

Durability

- Example: Let's say a customer places an order on an e-commerce platform:
  - The platform updates the Orders table to reflect the new order and deducts the stock from the Products table.
- Scenario:
  - If the system crashes after the transaction has been committed (e.g., the order is confirmed and saved), the changes will not be lost. The Orders table will reflect the new order, and the stock in the Products table will show the updated quantity.
  - If the system crashes before the commit, the transaction will be rolled back, and no changes will be made to the database, preserving consistency.

Durability


Changes to be applied
Crash
Reboot
State 1
State 2
State 2
Non-volatile Storage
Transaction log
State 2 is re-constructed and fully recovered using Transaction Log

Summary

ACID Property	Description	Example
Atomicity	A transaction is an all-or-nothing operation. If part of the transaction fails, the whole transaction fails.	Money transfer between two bank accounts: If one part fails, no changes are applied.
Consistency	A transaction takes the database from one valid state to another valid state. The integrity constraints are maintained.	Enrolling a student in a course: The student must have sufficient balance to enroll.
Isolation	Transactions do not interfere with each other. Each transaction executes as though it's the only transaction.	Two transactions happening at the same time should not interfere with each other's data.
Durability	Once a transaction is committed, the changes are permanent, even in the event of a system failure.	After placing an order on an e-commerce platform, the order and stock changes remain even if the system crashes after committing the transaction.

Transaction Managament

- Transaction management in a Database Management System (DBMS) refers to the process of managing transactions to ensure data consistency, integrity, and reliability.
- Transactions are sequences of operations performed as a single unit of work, which is crucial in maintaining the ACID properties (Atomicity, Consistency, Isolation, and Durability).
- A transaction management system helps control how transactions are handled and provides mechanisms for concurrency control, recovery, and ensuring the ACID properties.

Transaction Managament: Components

- Transaction:
  - A transaction is a logical unit of work that contains one or more operations (e.g., INSERT, UPDATE, DELETE) on the database.
  - A transaction ensures that the database moves from one consistent state to another while maintaining the integrity of data.

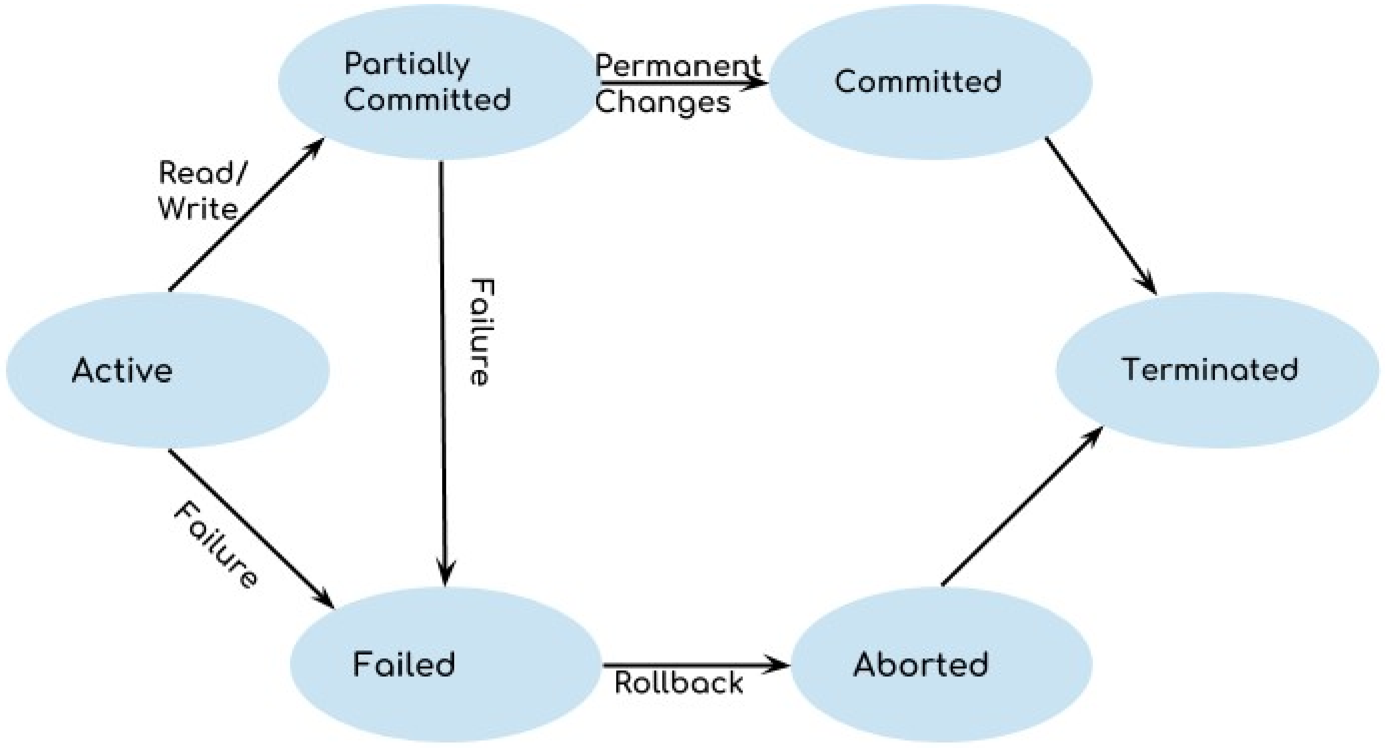
Transaction Managament: Components

- Transaction Log:
  - The transaction log is a record of all transactions and the changes they make to the database. It helps in recovery (in case of a system failure).
  - Each transaction in the log has information about what operations it performed (e.g., which records were modified, what values were updated, etc.).

Transaction Managament: Components

- Transaction States:
  - A transaction passes through several states during its lifetime:
    - Active: The transaction is being executed.
    - Partially Committed: The transaction has executed its final operation but has not yet been committed.
    - Committed: The transaction has been completed successfully, and all changes are permanent.
    - Failed: An error occurred during the transaction, and it is in an inconsistent state.
    - Aborted: The transaction has been rolled back, and the database is restored to its state before the transaction began.

Transaction Managament: Components



Transaction Control in DBMS

- Transaction Control Language (TCL) is a critical component of SQL used to manage transactions and ensure data integrity in relational databases.
- By using TCL commands, we can control how changes to the database are committed or reverted, maintaining consistency across multiple operations.

Transaction Managament: Components

- Transaction Control Commands:
  - START TRANSACTION / BEGIN TRANSACTION: Starts a new transaction.
  - COMMIT: Saves the changes made by the transaction to the database permanently.
  - ROLLBACK: Cancels the changes made by the transaction and restores the database to its previous state.
  - SAVEPOINT: Sets a point within a transaction to which you can later roll back.
  - SET TRANSACTION: Defines the properties of a transaction, such as isolation level.

Commit

- The COMMIT command is used to save all the transactions to the database that have been performed during the current transaction.
- Once a transaction is committed, it becomes permanent and cannot be undone.
- This command is typically used at the end of a series of SQL statements to ensure that all changes made during the transaction are saved.
  - Syntax:
    - COMMIT;

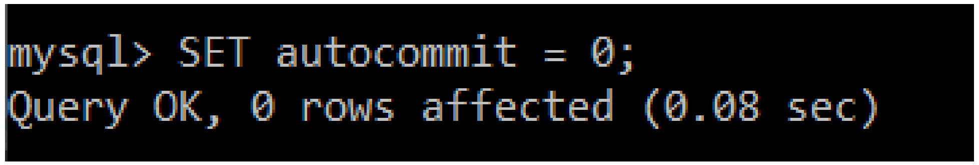
Commit

- mysql> USE school;
- mysql> CREATE TABLE t_school….
- mysql> START TRANSACTION;
- mysql> INSERT INTO t_school…….
- mysql> SELECT *FROM t_school;
- mysql> COMMIT;

Commit

- Autocommit is by default enabled in MySQL. To turn it off, we will set the value of autocommit as 0.

  mysql> SET autocommit = 0;



```
mysql> SET autocommit = 0;
Query OK, 0 rows affected (0.08 sec)
```

- MySQL, by default, commits every query the user executes. But if the user wishes to commit only the specific queries instead of committing every query, then turning off the autocommit is useful.

Rollback

- The ROLLBACK command is used to undo all the transactions that have been performed during the current transaction but have not yet been committed.
- This command is useful for reverting the database to its previous state in case an error occurs or if the changes made are not desired.
- Syntax:
  - ROLLBACK;

Savepoint

- The SAVEPOINT command is used to set a point within a transaction to which we can later roll back.
- This command allows for partial rollbacks within a transaction, providing more control over which parts of a transaction to undo.
- Syntax:
  - SAVEPOINT savepoint_name;

Rollback

- mysql> USE bank;
- mysql> CREATE TABLE customer….
- mysql> INSERT INTO customer….
- mysql> SELECT *FROM customer;
- mysql> START TRANSACTION;
- mysql> SAVEPOINT Insertion;
- mysql> DELETE FROM customer WHERE….
- mysql> SELECT *FROM customer;
- mysql> SAVEPOINT Deletion;
- mysql> ROLLBACK TO Insertion;
- mysql> SELECT *FROM customer;

Set Transaction

- The SET TRANSACTION statement is used to configure the properties of a transaction, such as isolation level or read-write behavior.
- Example:
  - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

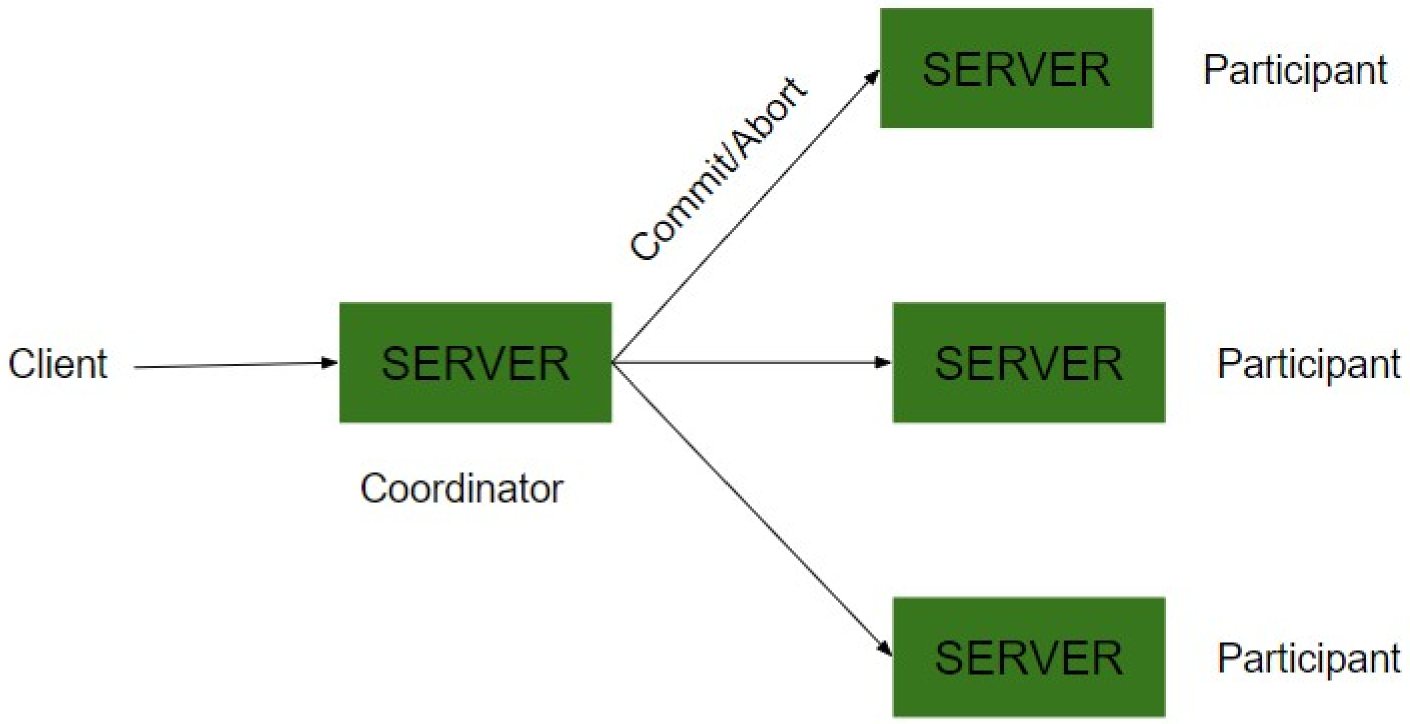
Commit Protocols

- Commit protocols in a Database Management System (DBMS) are mechanisms used to ensure that a transaction is either successfully committed or completely rolled back.
- These protocols play an essential role in managing the commit phase of transactions to ensure consistency, durability, and atomicity in a multi-user or distributed environment.
- A commit protocol defines the rules and procedures followed by the DBMS to safely complete a transaction and make its changes permanent in the database.

Commit Protocols

- There are several commit protocols used in DBMS to handle transaction commits, especially when dealing with distributed transactions.
- Some of the most common commit protocols are:
  - Immediate Commit Protocol
  - Deferred Commit Protocol
  - Two-Phase Commit (2PC) Protocol
  - Three-Phase Commit (3PC) Protocol

Commit Protocols



Immediate Commit Protocol

- In the Immediate Commit Protocol, once a transaction reaches its commit point, the DBMS immediately writes all changes made by the transaction to the database and the transaction is considered committed.
  - The transaction begins by performing operations (such as INSERT, UPDATE, DELETE) on the database.
  - When the transaction reaches its commit point, all changes are immediately written to disk.
  - *If any failure occurs during the transaction, it will be rolled back completely, and no changes will be saved to the database.*

Immediate Commit Protocol

- Advantages:
  - Simple and efficient when working with a single database instance.
  - Ensures that once the transaction is committed, all changes are immediately saved.
- Disadvantages:
  - Not ideal in a distributed database environment where multiple nodes need to coordinate the commit process.

Deferred Commit Protocol

- The Deferred Commit Protocol delays the actual writing of changes to the database until the transaction has been committed.
- Instead of immediately committing changes, the changes are kept in memory and only written to the database at the commit point.
  - A transaction performs all of its operations (e.g., inserts, updates).
  - Instead of writing the changes to the database immediately, the system defers the actual commit process.
  - When the transaction reaches the commit point, all changes are written to the database at once.
  - If any failure happens before the commit, all changes are discarded, and no changes are applied to the database.

Deferred Commit Protocol

- Advantages:
  - It reduces the number of write operations to the database.
  - Ensures that only committed transactions affect the database, minimizing the risk of partial updates.
- Disadvantages:
  - If there is a failure before the commit, changes made by the transaction are lost, and the transaction must be retried.
  - It is not suitable for high-concurrency environments where multiple transactions are simultaneously running.

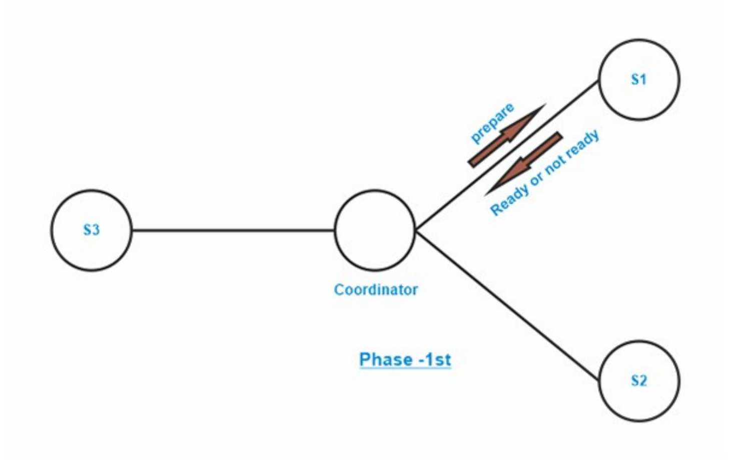
Two-Phase Commit (2PC) Protocol

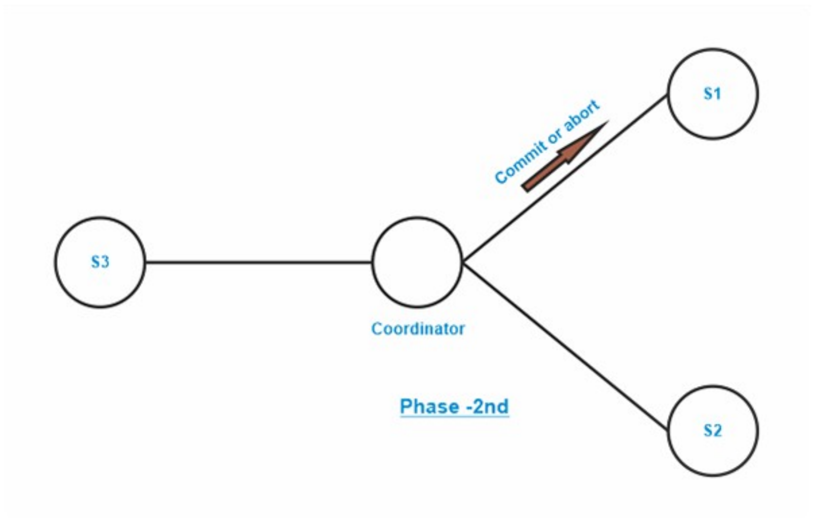
- The Two-Phase Commit (2PC) protocol is used in distributed databases to ensure that a transaction is either committed or rolled back across all participating databases in the same way.
- It ensures atomicity and consistency in distributed transactions.

Two-Phase Commit (2PC) Protocol

- The Two-Phase Commit (2PC) protocol consists of two phases:
  - Phase 1: Prepare Phase:
    - The coordinator (or transaction manager) sends a prepare message to all participants (nodes or databases) involved in the transaction.
    - Each participant responds with either a Yes (if they are ready to commit) or No (if there is any issue).
  - Phase 2: Commit or Abort Phase:
    - If all participants reply Yes, the coordinator sends a commit message, and the transaction is committed across all nodes.
    - If any participant replies No, the coordinator sends an abort message, and the transaction is rolled back on all nodes.

Two-Phase Commit (2PC) Protocol





Two-Phase Commit (2PC) Protocol

- Advantages:
  - Ensures consistency across distributed systems.
  - All participating nodes either commit the transaction or all abort, ensuring atomicity.
- Disadvantages:
  - Blocking: If the coordinator fails during the process, the transaction may be blocked because participants are waiting for the final decision (commit or abort).
  - Single point of failure: The failure of the coordinator can prevent progress.
- Example of 2PC Process:
  - Coordinator sends a Prepare request to all participants.
  - Each Participant responds with Yes or No.
  - If all participants say Yes, the Coordinator sends a Commit message. If any participant says No, the Coordinator sends an Abort message.

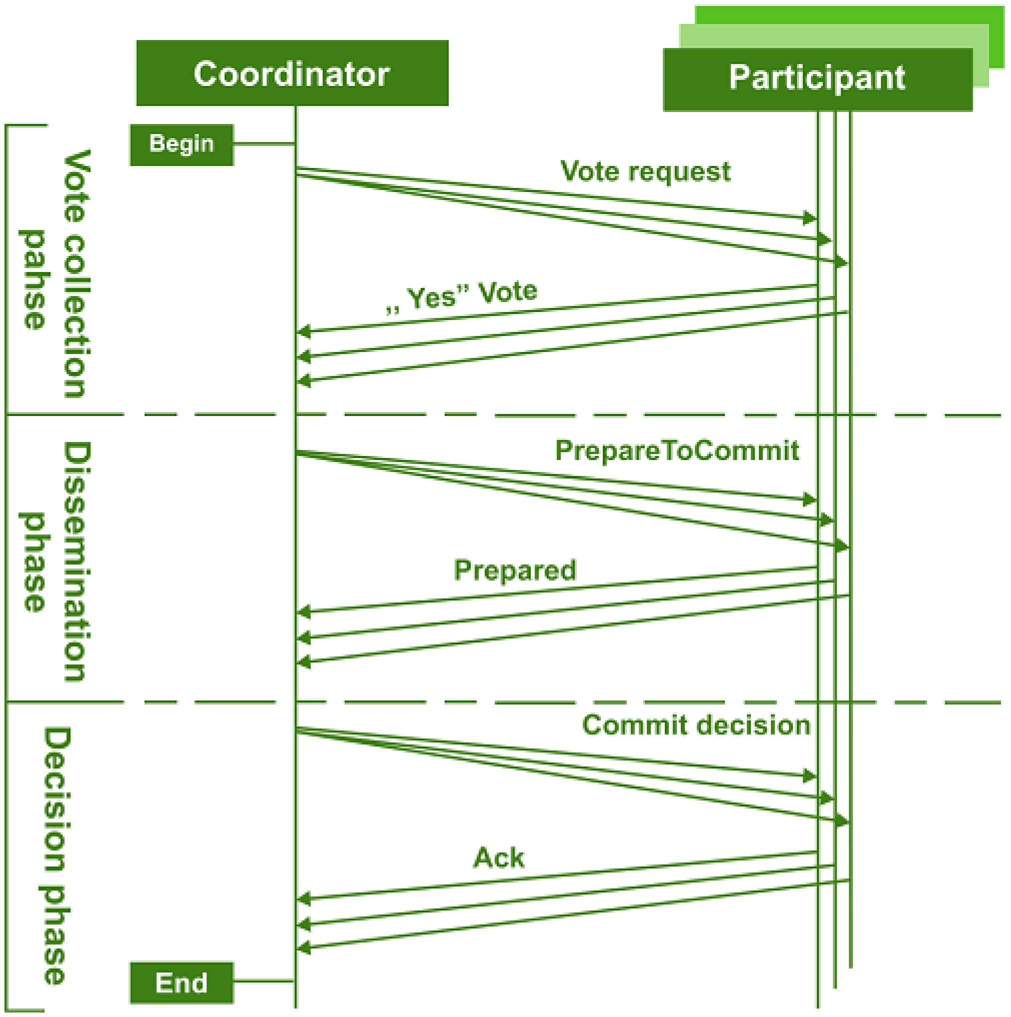
Three-Phase Commit (2PC) Protocol

- The Three-Phase Commit (3PC) protocol is an extension of the Two-Phase Commit protocol designed to overcome some of the limitations of 2PC, particularly the blocking problem.
- It ensures that a transaction is either fully committed or fully aborted, even in the case of coordinator or participant failures.

Three-Phase Commit (2PC) Protocol

- Phase 1: CanCommit Phase:
  - The coordinator sends a CanCommit message to all participants to check if they can prepare to commit.
  - The participants respond with either Yes (ready to commit) or No (unable to commit).
- Phase 2: PreCommit Phase:
  - If all participants respond Yes, the coordinator sends a PreCommit message to all participants.
  - The participants acknowledge that they are ready to commit and will write the transaction to a non-volatile storage.
- Phase 3: Commit/Abort Phase:
  - Once all participants respond with an acknowledgment, the coordinator sends the Commit message to confirm the transaction.
  - If there is any failure, the coordinator sends an Abort message, and the transaction is rolled back.

Three-Phase Commit (2PC) Protocol



Three-Phase Commit (2PC) Protocol

- Advantages:
  - Non-blocking: In case of a failure, the protocol ensures that participants can move to the next step or abort the transaction without being blocked indefinitely.
  - Fault-tolerant: The protocol can handle some failures better than 2PC by introducing an additional phase for recovery.
- Disadvantages:
  - Increased complexity in comparison to the 2PC protocol.
  - Still, in some failure conditions, the transaction may not be fully resolved (e.g., if the coordinator fails at an inappropriate time).

Three-Phase Commit (2PC) Protocol

- Example of 3PC Process:
  - Coordinator sends a CanCommit request.
  - Participants respond with Yes or No.
  - If Yes, the Coordinator sends a PreCommit message.
  - After acknowledgment, the Coordinator sends a Commit message.
  - If failure occurs, the Coordinator sends an Abort message.

Protocols: Summary

- Commit protocols in DBMS ensure that transactions are either fully committed or fully rolled back, maintaining the ACID properties of transactions.
- In a distributed system, Two-Phase Commit (2PC) is a widely used protocol for coordinating commits across multiple databases, while Three-Phase Commit (3PC) improves upon 2PC to provide fault tolerance and non-blocking behavior.
- Immediate Commit and Deferred Commit are simpler protocols used in local databases or environments where transactions are not distributed.
- The choice of commit protocol depends on the system's requirements, particularly in terms of fault tolerance, complexity, and transaction coordination.

Schedule

- A schedule is a sequence of operations (transactions) that are executed, such that the interleaving of those operations preserves the consistency of the database and ensures correctness in a multi-user environment.
- A schedule is essentially the order in which the operations (like read, write, commit, etc.) of different transactions are executed.
- Since multiple transactions may be running concurrently in a DBMS, a schedule defines how these transactions are interleaved, and whether their execution results in a valid final state of the database.

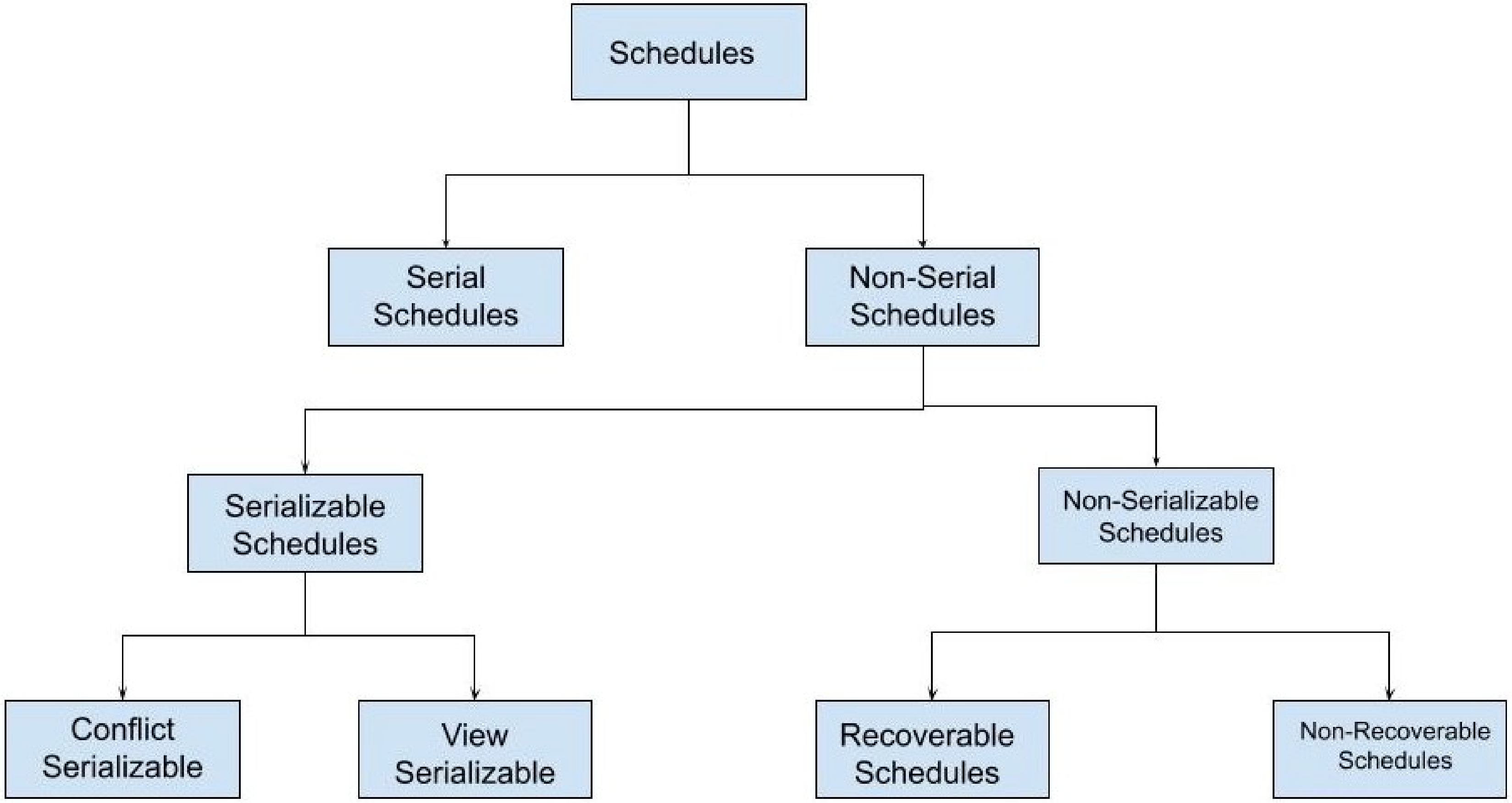
Schedule: Basic Operations

- Each transaction typically involves several operations, such as:
  - Read (R): A transaction reads a data item.
  - Write (W): A transaction writes a value to a data item.
  - Commit (C): A transaction successfully completes, and its changes are saved.
  - Abort (A): A transaction is rolled back due to an error or failure.

Schedule: Types

- There are different types of schedules based on the rules governing the order of transaction operations.
- The key types are:
  - Serial Schedule
  - Non-Serial Schedule
  - Recoverable Schedule
  - Cascadeless Schedule
  - View Serializable Schedule
  - Conflict Serializable Schedule

Schedule: Types



Serial Schedule

- A serial schedule is one in which the operations of the transactions are executed one after another, without any interleaving of operations.
- In a serial schedule, the transactions do not overlap, and the execution of each transaction is completed before the next one begins.
- Example:
  - Consider two transactions, T1 and T2:
    - T1: Read(A), Write(A)
    - T2: Read(B), Write(B)

Serial Schedule

- A serial schedule could be:
  - T1: Read(A), Write(A)
  - T2: Read(B), Write(B)
- OR:
  - T2: Read(B), Write(B)
  - T1: Read(A), Write(A)
- Since transactions are executed in isolation (one after the other), serial schedules are always consistent, but they do not take advantage of concurrent execution and may be inefficient.

Non-Serial Schedule

- A non-serial schedule is one in which the operations of multiple transactions are interleaved, i.e., the operations of one transaction are mixed with operations of another transaction.
- Non-serial schedules can increase concurrency, leading to better performance, but they require mechanisms to ensure that the schedule results in a valid final state.
- Consider the following interleaved schedule:
  - T1: Read(A), Write(A)
  - T2: Read(B), Write(B)
  - T1: Read(B), Write(B)
  - T2: Read(A), Write(A)
- This schedule is non-serial because the operations of T1 and T2 are interleaved.

Non-Serial Schedule

Non-Serial

S1

Transaction T1	Transaction T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)

Serial

S2

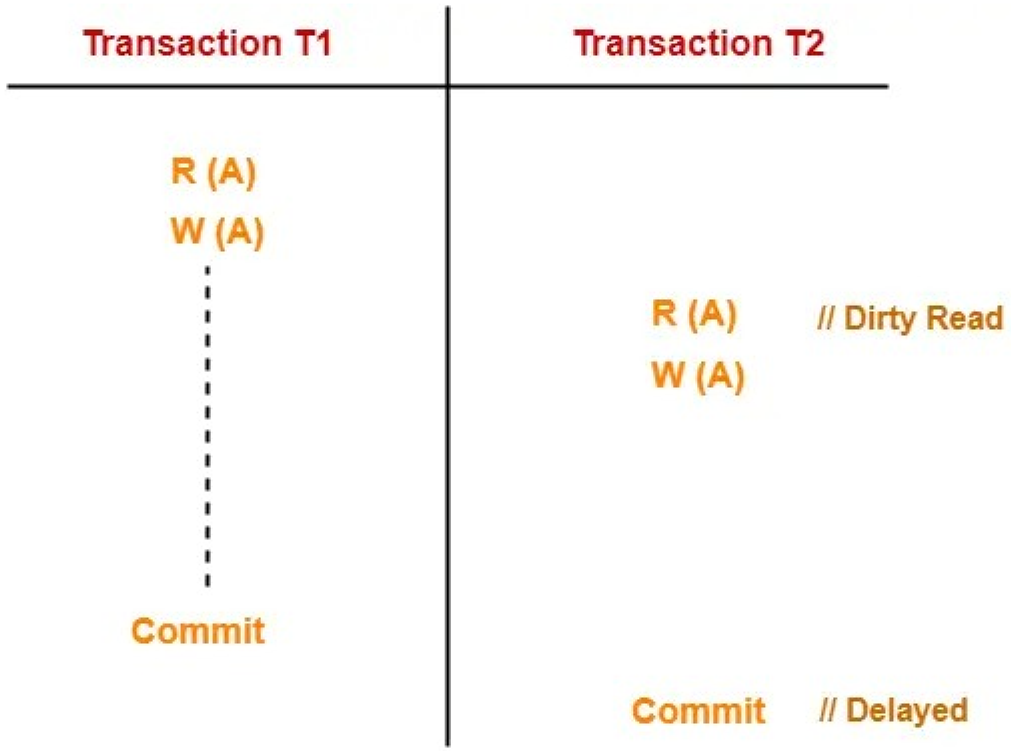
Transaction T1	Transaction T2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

S2 is the serial schedule of S1. If we can prove that they are view equivalent then we we can say that given schedule S1 is view Serializable

Recoverable Schedule

- A recoverable schedule ensures that if a transaction T2 depends on a transaction T1 (e.g., reading a value written by T1), T2 is only allowed to commit after T1 commits. This prevents situations where a transaction reads uncommitted data from another transaction that eventually aborts.
- Example:
  - If T2 reads a value written by T1, T2 must commit only if T1 commits.
  - If T1 aborts, T2 must also abort.
- A recoverable schedule prevents situations where a transaction's result might depend on another transaction that has not yet committed, avoiding the issue of dirty reads.

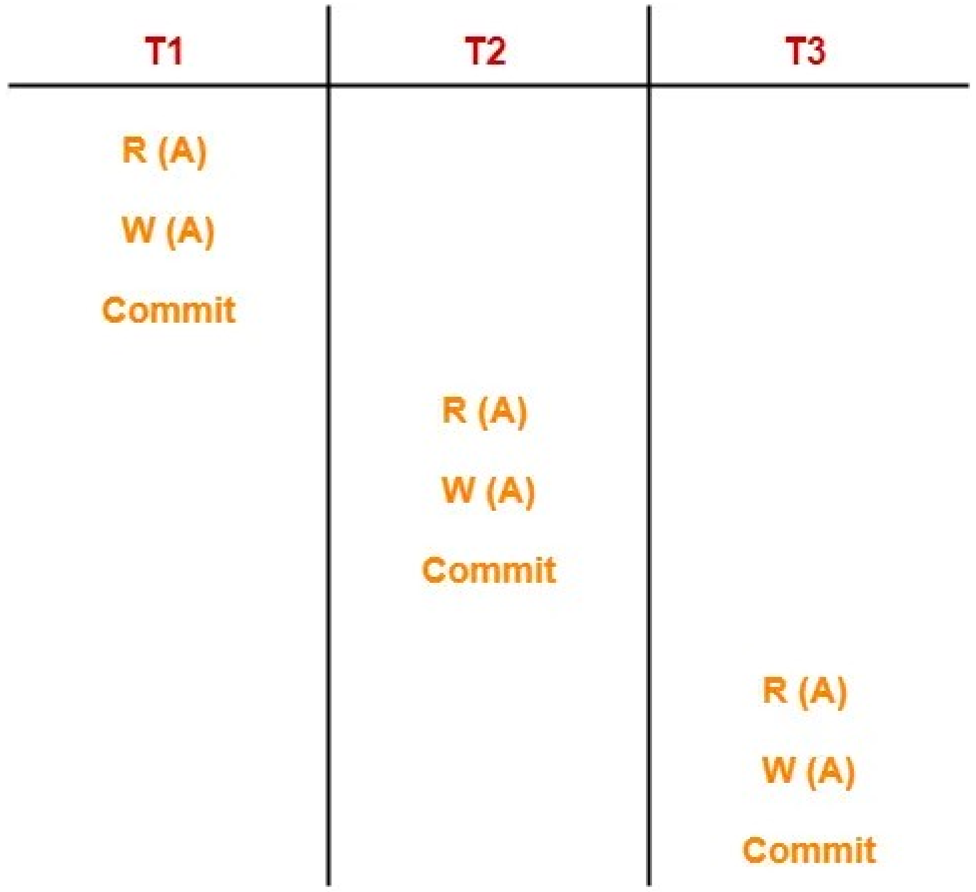
Recoverable Schedule



Cascadeless Schedule

- A cascadeless schedule is one in which a transaction never reads uncommitted data (i.e., no dirty reads). This type of schedule ensures that transactions that read values only do so after the transaction that wrote those values has committed.
  - T1 writes a value to a data item, and T2 can read that value only after T1 has committed. This avoids the problem of transactions depending on uncommitted data.
- Advantages:
  - Prevents the cascading rollback problem, where the failure of one transaction leads to the need to roll back several others.

Cascadeless Schedule



T1	T2	T3
R (A)		
W (A)		
Commit		
	R (A)	
	W (A)	
	Commit	
		R (A)
		W (A)
		Commit

View Serializable Schedule

- A schedule is view serializable if, by permuting the operations in the schedule (while respecting the operation order within individual transactions), it can be converted into a serial schedule. This is a weaker form of serializability than conflict serializability but still ensures that the final outcome of the transactions is the same as if the transactions were executed serially.
- Conditions for View Serializability:
  - The initial reads of all transactions must be the same.
  - The final writes of all transactions must be the same.
  - If a transaction reads a value written by another transaction, the final write must be consistent with the reading.

Conflict Serializable Schedule

- A conflict serializable schedule is one that can be transformed into a serial schedule by swapping non-conflicting operations.
- Two operations conflict if they belong to different transactions and access the same data item, and at least one of them is a write.

Conflict Serializable Schedule

- Consider the following non-serial schedule:
  - T1: Read(A), Write(A)
  - T2: Read(B), Write(B)
  - T1: Write(B)
  - T2: Write(A)
- *In this case, we can swap the operations of T1 and T2 to form a serial schedule, maintaining the consistency of the database.*
- Conflict serializability is a stronger condition than view serializability and ensures that the final database state remains consistent regardless of transaction interleaving.

Schedule Properties

- Serializability:
  - The schedule should be serializable, meaning that the final result of the interleaved operations is the same as if the transactions had been executed one by one (serially).
- Recoverability:
  - A schedule is recoverable if, after a transaction writes a value, no other transaction reads it before the first transaction commits.
- Cascadeless:
  - A schedule is cascadeless if transactions never read uncommitted data from other transactions, thereby avoiding cascading rollbacks.
- Conflict Serializability:
  - A schedule is conflict serializable if it can be converted into a serial schedule by swapping non-conflicting operations.

Serializability

- Schedule 1 (Conflict Serializable)
  - T1: Read(A), Write(A)
  - T2: Read(B), Write(B)
  - T1: Write(B)
  - T2: Write(A)
- This schedule is conflict serializable because you can rearrange the operations to form a serial schedule:
  - T1: Read(A), Write(A), Write(B)
  - T2: Read(B), Write(A)

Serializability

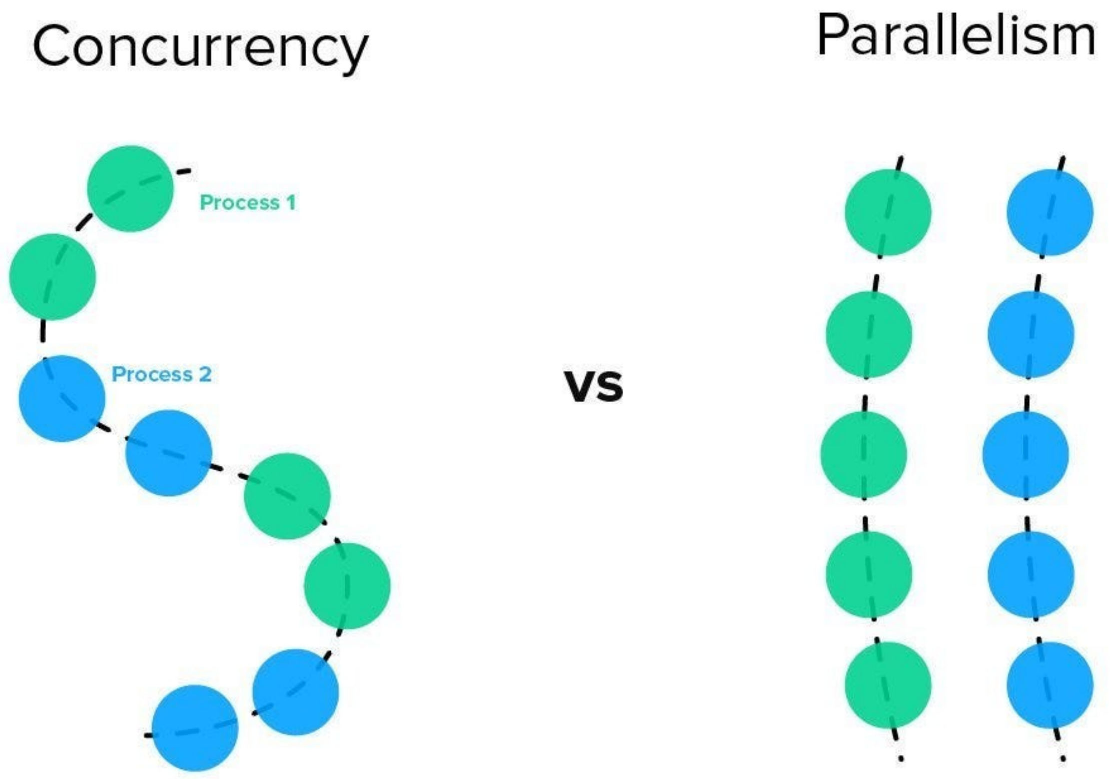
- Schedule 2 (View Serializable but not Conflict Serializable)
  - T1: Write(A)
  - T2: Read(A), Write(A)
- This schedule is view serializable but not conflict serializable.
- While the result of this schedule can be made equivalent to a serial schedule by changing the order, the conflict serializability condition is violated due to overlapping write and read operations on the same data.

Concurrency Control

Concurrency Control

- Concurrency control is a mechanism used in a Database Management System (DBMS) to ensure that the database remains consistent and accurate while handling multiple transactions simultaneously.
- In a multi-user environment, where multiple transactions can access the database at the same time, it's important to manage how these transactions interact with each other to prevent conflicts, such as lost updates, dirty reads, uncommitted data, and inconsistent reads.
- Concurrency control ensures that the ACID properties (Atomicity, Consistency, Isolation, and Durability) are maintained, particularly the Isolation property, which isolates the operations of a transaction from those of other transactions.

Concurrency Control



Concurrency Control: Key Issues

- Lost Updates:
  - When two transactions try to update the same data simultaneously, and one of the updates is overwritten, resulting in the loss of data.
- Temporary Inconsistency (Dirty Reads):
  - When one transaction reads data that is modified by another transaction that has not yet been committed, causing the first transaction to operate on inconsistent or incorrect data.

Concurrency Control: Key Issues

- Uncommitted Data:
  - A transaction reads data that has been modified but not committed by another transaction. If the second transaction is rolled back, the first transaction might be using invalid data.
- Non-repeatable Reads:
  - A transaction reads the same data multiple times, but the value changes between reads because of another transaction's update.
- Phantom Reads:
  - A transaction reads a set of data based on a query, but another transaction inserts, deletes, or updates data that affects the result of the query.

Concurrency Control: Techniques

- Lock-based Concurrency Control
- Timestamp-based Concurrency Control
- Optimistic Concurrency Control
- Multiversion Concurrency Control (MVCC)

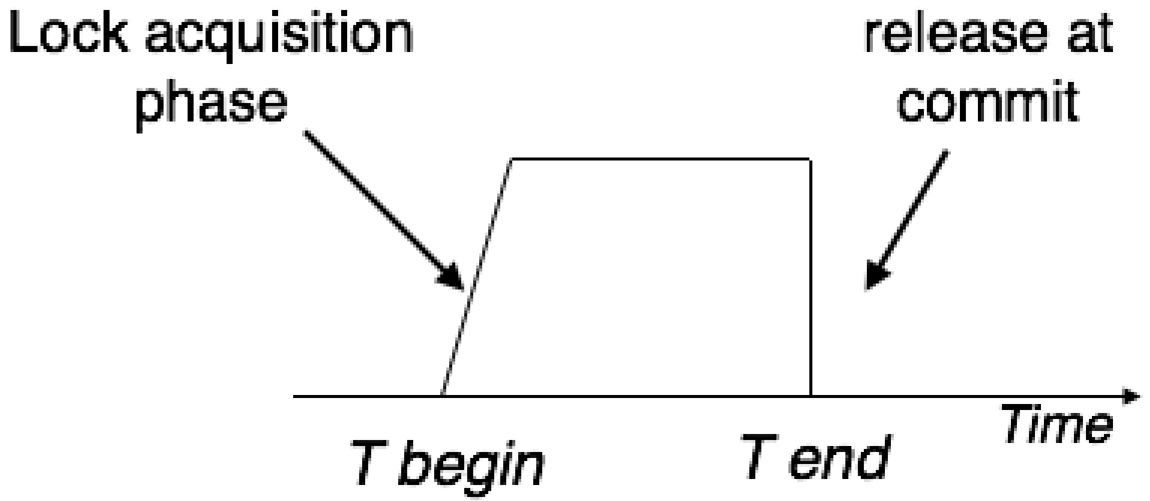
1. Lock-based Concurrency Control

- In this method, locks are used to control access to data items. When a transaction needs to read or write a data item, it acquires a lock on that item to prevent other transactions from interfering with it. There are different types of locks used in this technique:
  - Shared Lock (S-lock): A shared lock allows multiple transactions to read the same data item concurrently. However, no transaction can modify the data item while it is locked with a shared lock.
  - Exclusive Lock (X-lock): An exclusive lock is acquired when a transaction intends to write (modify) a data item. When a data item has an exclusive lock, no other transaction can read or write that item until the lock is released.

1. Lock-based Concurrency Control

- Two-Phase Locking (2PL): This protocol ensures that transactions acquire locks in two phases—growing (acquiring locks) and shrinking (releasing locks).
- Once a transaction releases a lock, it cannot acquire any more locks. This protocol guarantees serializability.
- Example:
  - Transaction 1 acquires a shared lock on a data item and reads it.
  - Transaction 2 also acquires a shared lock and reads the same data item.
  - Transaction 1 then acquires an exclusive lock and updates the data item.
  - Transaction 1 releases the lock after committing, and only then can Transaction 2 acquire an exclusive lock to update the data item.

1. Lock-based Concurrency Control



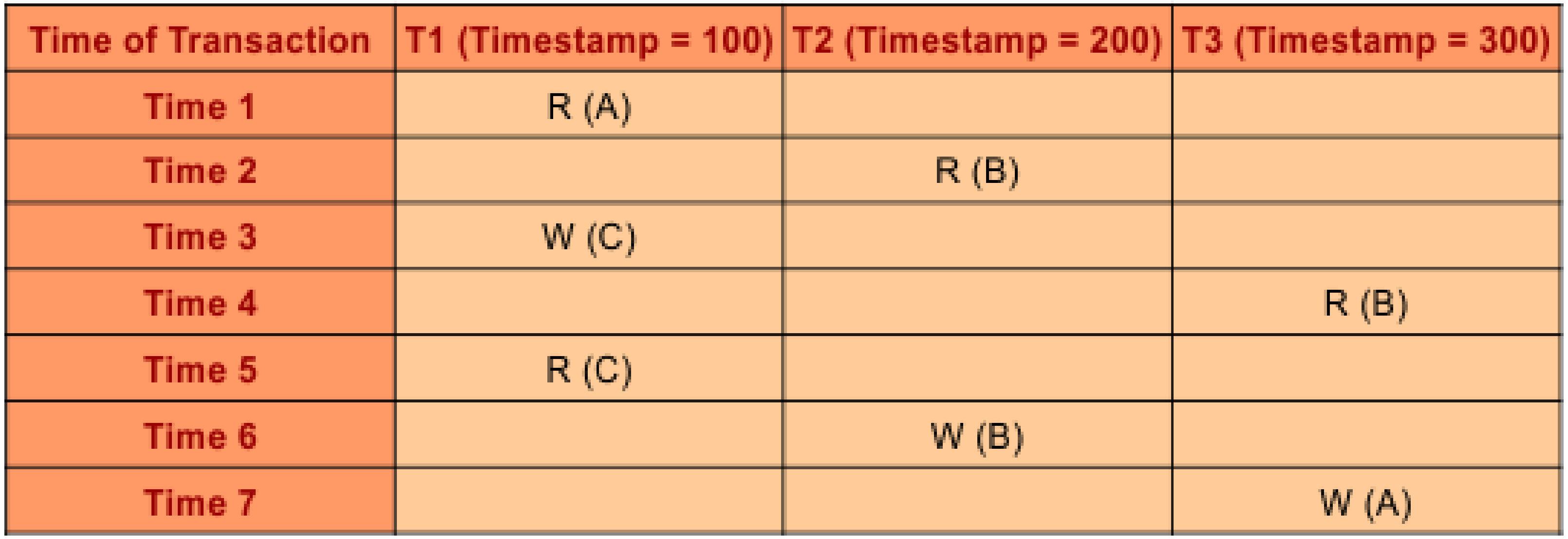
Deadlock

- A situation where two or more transactions are blocked because they each hold a lock that the other transaction needs.
- Deadlock detection and resolution mechanisms are used to prevent this.

Timestamp-based Concurrency Control

- Each transaction is assigned a unique timestamp when it starts. This timestamp is used to order transactions and ensure that they are executed in a serializable manner. The two primary operations are:
  - Transaction Ordering: If two transactions T1 and T2 conflict (access the same data item), the one with the smaller timestamp is allowed to proceed, and the one with the larger timestamp is rolled back if it conflicts with the first transaction.
  - Basic Protocol: The transaction that requests a data item checks if the timestamp of the transaction matches the read/write request. If there is a conflict (e.g., a transaction tries to write a value that has been read by another transaction), the system will determine which transaction should proceed based on the timestamps.

Timestamp-based Concurrency Control



Time of Transaction	T1 (Timestamp = 100)	T2 (Timestamp = 200)	T3 (Timestamp = 300)
Time 1	R (A)		
Time 2		R (B)	
Time 3	W (C)		
Time 4			R (B)
Time 5	R (C)		
Time 6		W (B)	
Time 7			W (A)

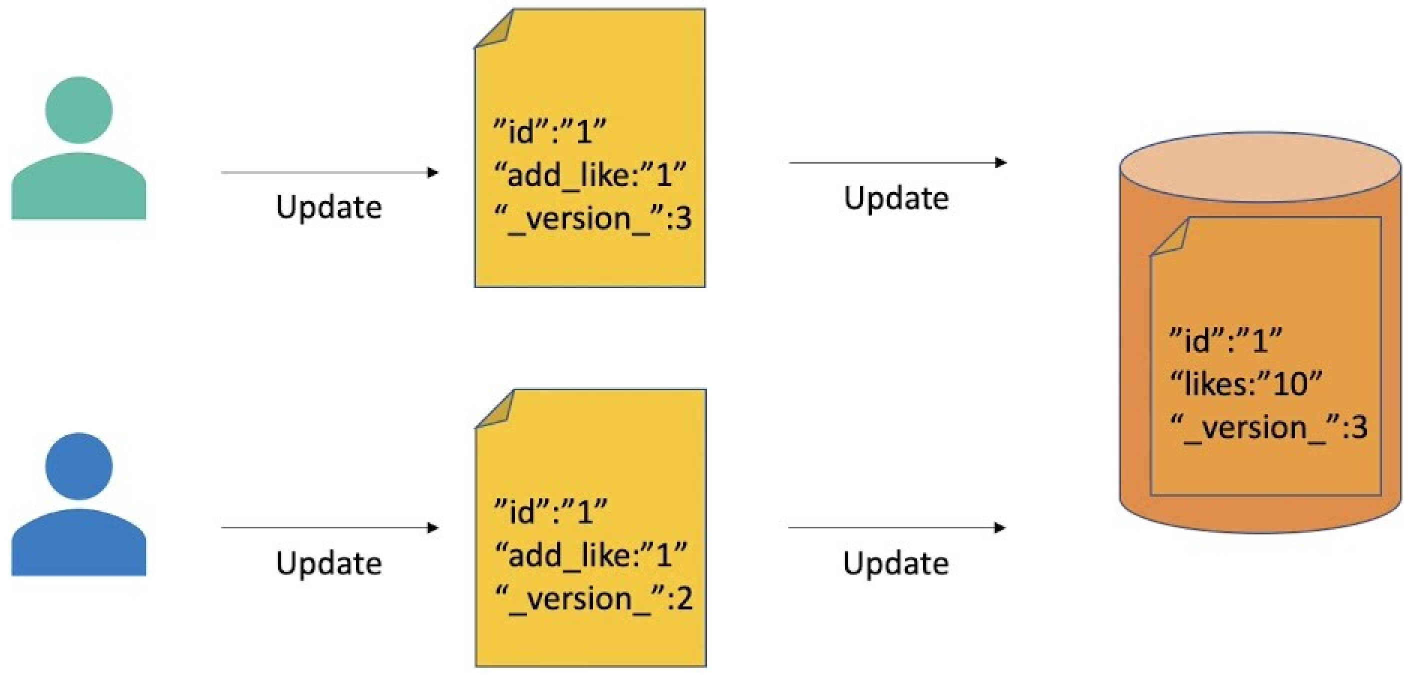
Optimistic Concurrency Control (OCC)

- Optimistic concurrency control is based on the assumption that conflicts between transactions are rare.
- The basic idea is to allow transactions to execute without locking data, and only check for conflicts before committing the transaction.
- This approach works well in situations where read-heavy workloads are common.

Optimistic Concurrency Control (OCC)

- Steps:
  - Transaction Execution: A transaction executes without acquiring any locks, and performs its operations on the data.
  - Validation Phase: Before committing, the transaction is validated to check if it conflicts with any other transaction. This is usually done by checking whether any other transaction has modified the same data that the current transaction has read or written.
  - Commit or Rollback: If the transaction passes the validation phase (i.e., no conflict), it is committed. If conflicts are found, the transaction is rolled back and retried.

Optimistic Concurrency Control (OCC)



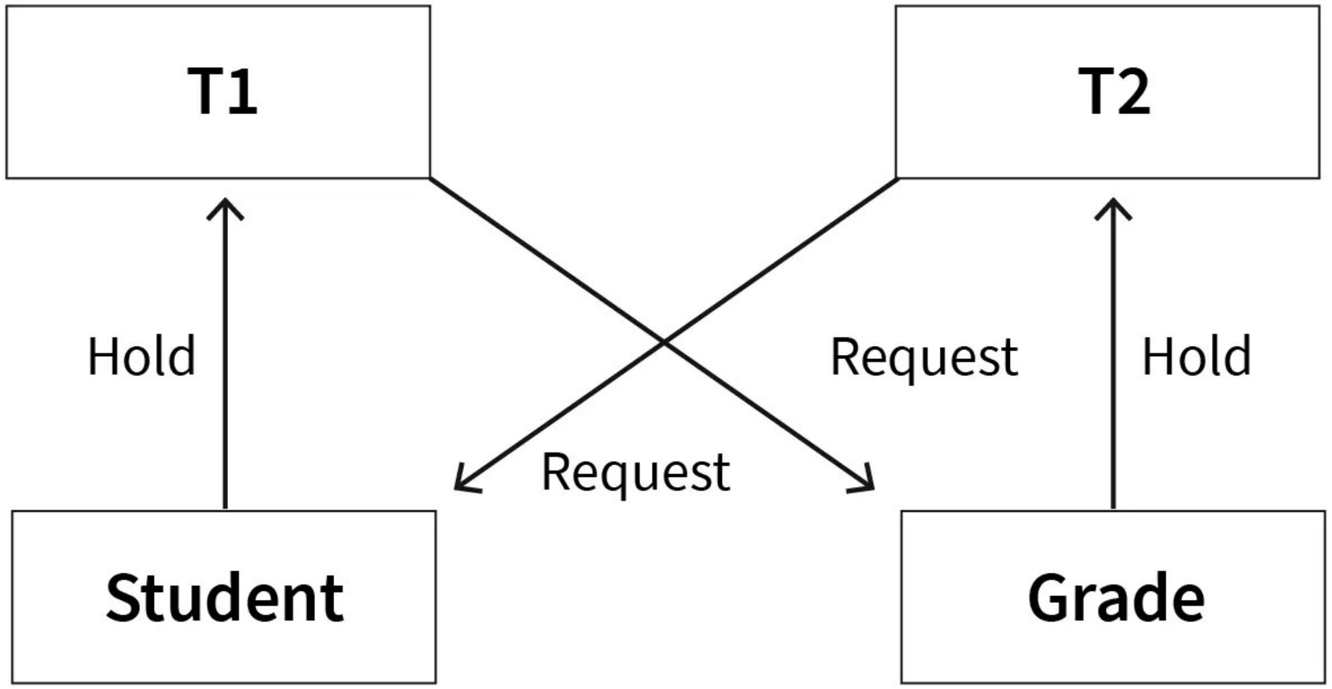
Optimistic Concurrency Control (OCC)

- Example:
  - Transaction T1 reads and writes data without locking the data.
  - Before T1 commits, the system checks whether T2 has modified the same data.
  - If T2 has made changes, T1 is rolled back. Otherwise, T1 commits successfully.

Deadlock

- Deadlock is a situation that occurs in a Database Management System (DBMS) (or any multi-process system) when two or more transactions are unable to proceed because each is waiting for the other to release a resource or a lock.
- In a deadlock, the involved transactions are blocked forever and cannot make progress.
- This results in a stale state, where no transaction can complete unless some action is taken to resolve the deadlock.

Deadlock



Deadlock : Features

- For a deadlock to occur, the following four conditions, known as the Coffman conditions, must hold true:
  - Mutual Exclusion: A resource (such as a data item) can only be used by one transaction at a time. If a transaction holds a lock on a resource, others cannot access it.
  - Hold and Wait: A transaction holding at least one resource is waiting to acquire additional resources that are currently being held by other transactions.
  - No Preemption: Resources cannot be forcibly taken from transactions holding them. A resource can only be released voluntarily by the transaction holding it.
  - Circular Wait: A set of transactions {T1, T2, ..., Tn} exists, where each transaction is waiting for a resource held by the next transaction in the set, forming a cycle.

Deadlock : Example

- Consider the following example of a deadlock scenario involving two transactions:
  - Transaction T1 locks resource A and wants resource B.
  - Transaction T2 locks resource B and wants resource A.
  - Transaction T1 is waiting for Transaction T2 to release resource B.
  - Transaction T2 is waiting for Transaction T1 to release resource A.
- Since neither transaction can proceed without the other releasing a resource, they are in a deadlock state. Both transactions are blocked indefinitely.

Deadlock Prevention

- Eliminate Circular Wait: Prevent transactions from holding one resource while waiting for others. One way is to ensure that transactions request all the resources they need at once, or none at all (atomic requests).
- Preemption: If a transaction is holding a resource and waiting for others, the DBMS may preempt (forcefully take away) the resource and assign it to another transaction. This resolves the Hold and Wait condition.
- Ordering of Resources: Transactions request resources in a predefined order (e.g., always acquire resource A before resource B). This ensures that circular wait is avoided.

Deadlock Detection

- The system uses a wait-for graph or resource allocation graph to monitor the resources and transactions.
- The wait-for graph is a directed graph where:
  - Nodes represent transactions.
  - Edges represent a transaction waiting for a resource held by another transaction.
- If a cycle is detected in this graph, a deadlock exists. Once a deadlock is detected, one of the transactions involved in the cycle is chosen to be rolled back (aborted), releasing the resources and allowing the remaining transactions to proceed.

Deadlock Detection

$$\begin{tikzcd}
\mathbf{P}_i \arrow[rr, "x"] & & \mathbf{P}_j \arrow[dl, "y"] \\
& \mathbf{P}_k \arrow[ul, "z"] &
\end{tikzcd}$$

Deadlock Recovery

- Transaction Rollback: The simplest way to resolve a deadlock is to abort *one of the transactions* involved in the deadlock cycle. Once the transaction is rolled back, it releases its locks, and the other transactions can continue.
- Resource Preemption: In some cases, the system may *preempt resources from one transaction* (even if it hasn't completed) and assign them to other transactions. The preempted transaction can then be rolled back or restarted.
- Transaction Restart: After *rolling back a transaction*, it can be restarted. However, this may incur additional overhead if the transaction has already performed significant work.

Deadlock in MySQL

- **Deadlock Detection**: InnoDB detects deadlocks using a timeout mechanism. If a transaction is waiting for a resource that is locked by another transaction, and that other transaction is also waiting for a resource that the first transaction holds, InnoDB detects a cycle (deadlock) and chooses one transaction to roll back.
- **Deadlock Handling**: When a deadlock is detected, InnoDB automatically chooses one of the transactions to rollback and returns a deadlock error (error code 1213).
- Example of a deadlock error in MySQL:
  - ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

Deadlock in MySQL

- Consider the following two transactions in MySQL:
- Transaction 1 (T1):
  - START TRANSACTION;
  - UPDATE accounts SET balance = balance - 500 WHERE account_id = 1; -- Transaction 1 locks account_id = 1
  - UPDATE accounts SET balance = balance + 500 WHERE account_id = 2; -- Transaction 1 is waiting for account_id = 2

Deadlock in MySQL

- Transaction 2 (T2):
  - START TRANSACTION;
  - UPDATE accounts SET balance = balance + 500 WHERE account_id = 2; -- Transaction 2 locks account_id = 2
  - UPDATE accounts SET balance = balance - 500 WHERE account_id = 1; -- Transaction 2 is waiting for account_id = 1

Deadlock in MySQL

- If both transactions are executing concurrently, a deadlock will occur:
  - Transaction 1 has locked account_id = 1 and is waiting for account_id = 2.
  - Transaction 2 has locked account_id = 2 and is waiting for account_id = 1.
- MySQL's InnoDB engine will detect this deadlock and roll back one of the transactions to resolve the issue.

Recovery

- Database recovery refers to the process of restoring a database to a correct state after a failure.
- In a database management system (DBMS), failures can occur for various reasons, including hardware failure, software crash, power failure, human error, or even corruption of the database.
- The primary goal of recovery is to ensure ACID properties (Atomicity, Consistency, Isolation, and Durability) are maintained even after a failure.
- Recovery mechanisms ensure that the database remains consistent and no data is lost, while also preventing the system from entering an inconsistent state.

Types of failures

- Transaction Failures:
  - When a transaction cannot be completed due to an error (e.g., an application crash or logic error).
- System Failures:
  - When the DBMS or operating system crashes, possibly resulting in lost data or incomplete transactions.
- Disk Failures:
  - When a disk or storage medium where the database is stored fails.
- Human Errors:
  - Accidental data deletion, modification, or other mistakes made by users.

Recovery Techniques

- Log-based Recovery
- Shadow Paging
- Checkpointing
- Backup and Restore

Log-Based Recovery

- Log-based recovery is the most widely used method in modern DBMS for transaction management.
- In this technique, the DBMS keeps a log of all changes made to the database, including transaction start, commit, and rollback operations.
- The log contains records of every transaction in the database, and it is stored in a sequential manner.
  - Write-Ahead Logging (WAL): In WAL, the log is written to a stable storage before any changes are applied to the actual database. This ensures that if a crash occurs, the DBMS can refer to the log to undo any incomplete transactions and redo any committed transactions.

Log-Based Recovery

- Example:
  - When a transaction updates a record, an entry is made in the log file that includes the transaction ID, the record updated, and the new value.
  - If a failure occurs, the DBMS uses the log to either undo (rollback) uncommitted changes or redo (reapply) changes of committed transactions.

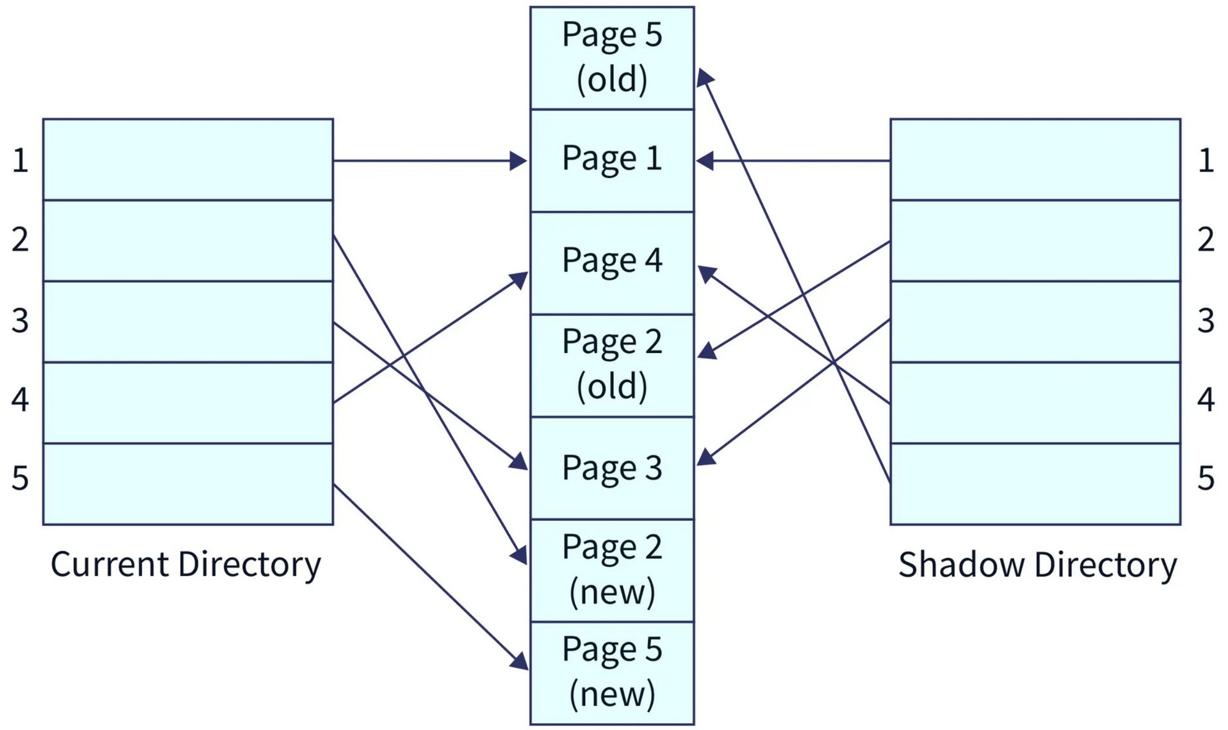
Log-Based Recovery

- Steps:
  - Before writing to disk: The transaction logs are written.
  - Transaction commit: The log is updated with a commit record.
  - Failure recovery: The system uses the log to undo or redo transactions based on whether they were completed or not.

Shadow Paging

- Shadow paging involves maintaining a copy of the database called a shadow page.
- When a transaction makes a change to the database, it modifies the current page. If a crash occurs, the shadow page remains intact and the database can be restored to the previous consistent state.
  - Shadow Pages: In shadow paging, the DBMS maintains two sets of pages: the current page and the shadow page.
  - Atomic Updates: Changes are first made to the new page, and once the transaction commits, the current page is switched with the shadow page.
  - Rollback: If a failure occurs, the shadow page (unchanged) is used to restore the database to its previous state.

Shadow Paging



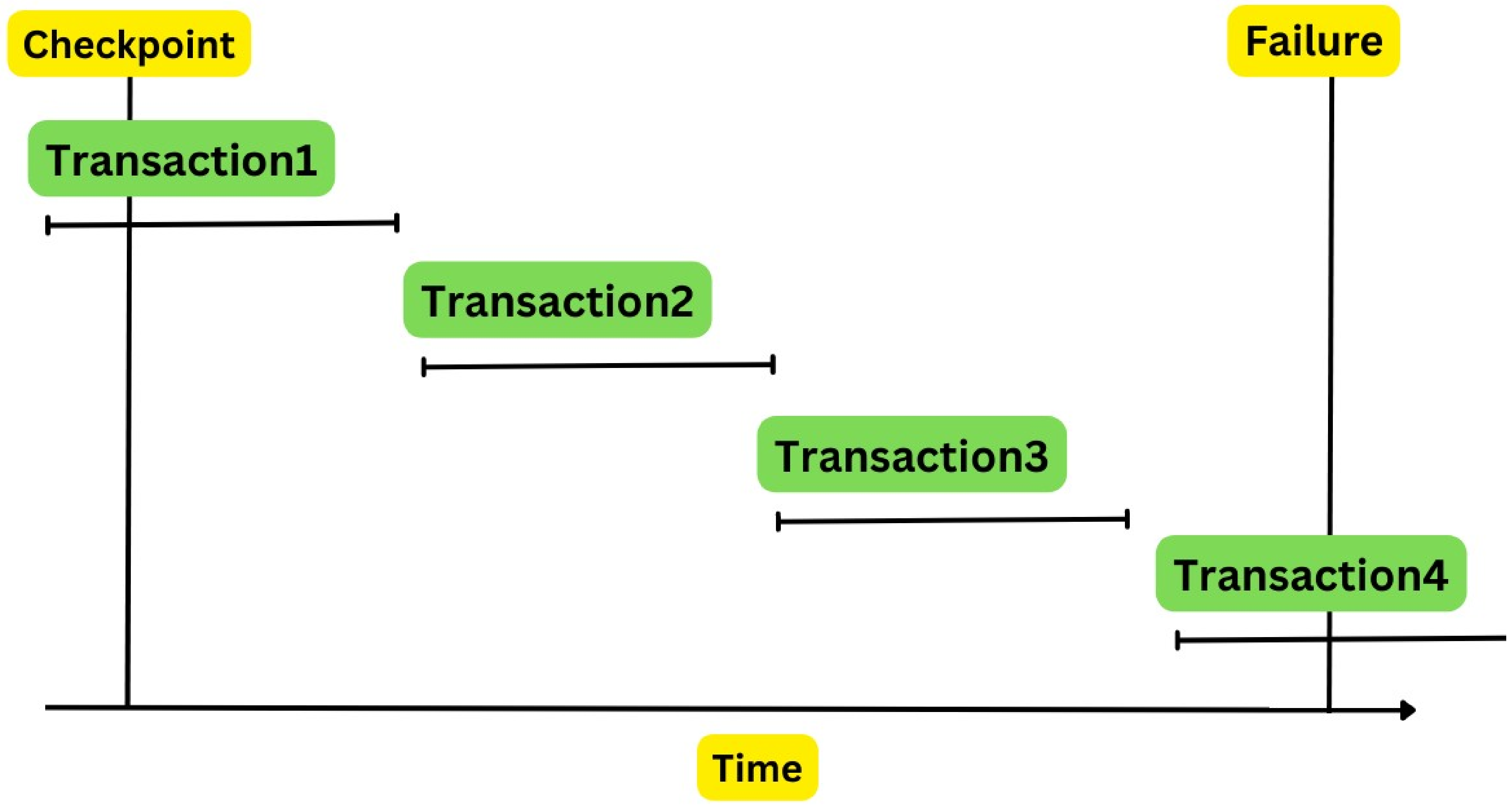
Shadow Paging

- Steps:
  - A shadow copy of the database is created.
  - When a transaction updates data, changes are made to the current page.
  - When the transaction commits, the current page replaces the shadow page.
  - If a failure happens before the transaction commits, the system rolls back to the shadow page.

Checkpointing

- Checkpointing is a technique used to minimize the amount of work required during recovery. It involves saving the current state of the database periodically by creating checkpoint records in the log.
  - Checkpoint Record:
    - A checkpoint is a point in time where the DBMS writes all the changes from memory to disk.
    - After a checkpoint is created, only transactions that start after that checkpoint need to be considered for recovery in the event of a failure.

Checkpointing



Checkpointing

- Steps:
  - Periodically, the DBMS writes a checkpoint record to the log, marking a known point in the database.
  - After the checkpoint is written, the DBMS flushes all changes in memory (buffer pool) to disk.
  - If a failure occurs after the checkpoint, the system needs to only look at the log entries after the checkpoint and perform a recovery of those transactions.

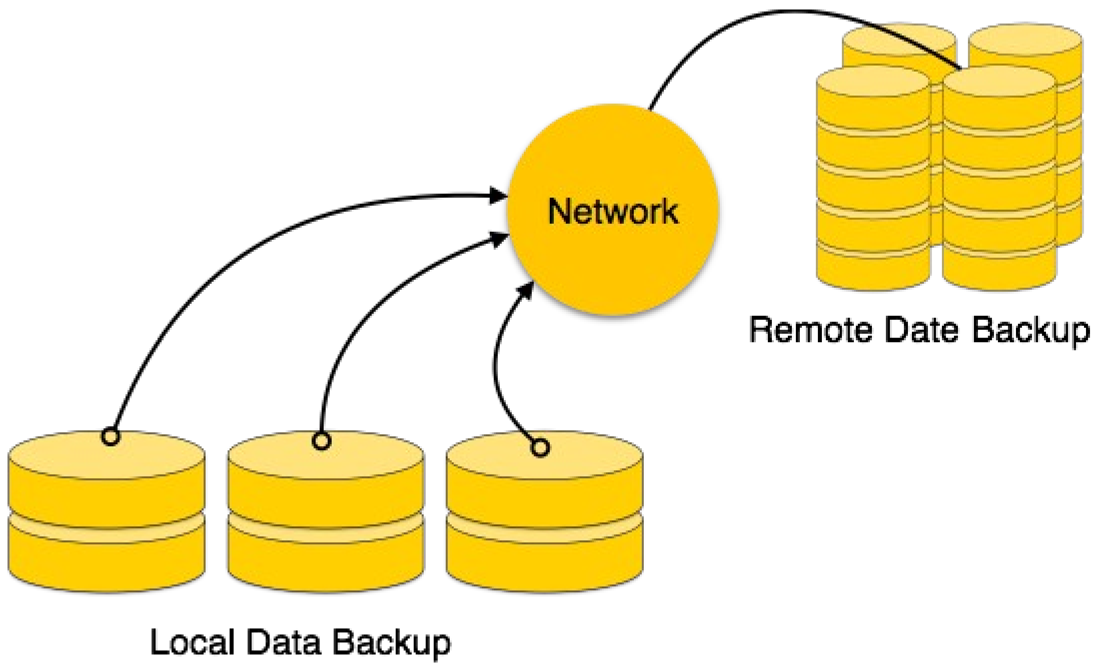
Backup and Restore

- It involves creating periodic backups of the entire database and restoring it when needed.
- In the event of a failure, the system can be restored to the point of the last backup.
  - Full Backup: A snapshot of the entire database is taken and stored on a separate medium.
  - Incremental Backup: Only changes since the last backup are stored.
  - Point-in-time Recovery: You can restore the database to a specific point in time using backup and transaction logs.

Backup and Restore

- Steps:
  - Perform regular backups of the database (daily, weekly, etc.).
  - Store backups securely (off-site or in cloud storage).
  - In case of failure, restore the most recent backup.
  - Use transaction logs or incremental backups to restore the database to its last consistent state.

Backup and Restore



Recovery in Log-based Systems

- Undo Recovery:
  - Used for aborted transactions. The system undoes changes made by a transaction that has not committed.
  - This ensures that the transaction’s partial updates are discarded, leaving the database in a consistent state.
- Example: If a transaction updates a table but fails before committing, the system will roll back any changes made by that transaction.

Recovery in Log-based Systems

- Redo Recovery:
  - Used for committed transactions. This ensures that any changes made by committed transactions are preserved even if a crash occurs before they were written to the database.
- Example: If a transaction successfully commits but the change has not been written to disk when a crash occurs, the system will redo the transaction using the log file.

Recovery in Log-based Systems

- No-Action Recovery:
  - No action is taken if a failure occurs, as the system assumes the transaction will complete successfully and commit after it restarts.
  - However, this method is not commonly used as it doesn't guarantee ACID compliance.

Conclusion

- Transaction management is essential for maintaining data integrity and consistency in DBMS.
- ACID properties (Atomicity, Consistency, Isolation, Durability) ensure reliable transactions.
- Concurrency control techniques help in handling simultaneous transaction requests.
- Recovery methods like logging, checkpointing, and backup ensure system reliability during failures.
- A well-designed transaction system enhances database performance and ensures fault tolerance.

Thank you

@mitu_skillologies @mITuSkillologies @mitu_group @mitu-skillologies @MITUSkillologies

kaggle

@mituskillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

@mituskillologies

contact@mitu.co.in

tushar@tusharkute.com