ADVANCED DATABASE

[i]CHAPTER ONE

TRANSACTION CONCEPT

The concept of a transaction in advanced databases refers to a sequence of operations or actions performed on a database as a single logical unit. Imagine you are shopping online and adding items to your cart, entering your shipping address, and making the payment. All these actions collectively form a transaction.

Transactions are important because they ensure that the database remains consistent and reliable.

They follow the ACID properties:

1. **Atomicity**: A transaction is atomic, meaning it is treated as a single indivisible unit of work. Either all the operations within a transaction are completed successfully, or none of them are. In our online shopping example, either the entire process of adding items to the cart, entering the address, and making the payment is completed, or none of it is.

2. **Consistency**: A transaction takes the database from one consistent state to another. It means that the data in the database should satisfy all the predefined rules and constraints. For instance, if you have a rule that a customer cannot purchase more items than their available credit limit, the transaction should ensure that this rule is not violated.

Imagine you have a set of rules for playing a game. Consistency means that you always follow those rules, and everyone playing the game follows the same rules too. It's like having a fair and equal game for everyone.

In a database, consistency means that the data follows certain rules or conditions that are set for it. For example, let's say you have a database of students' grades, and the rule is that a grade can only be between 0 and 100. If someone tries to enter a grade of 150, that would violate the consistency rule because it's outside the allowed range.

Consistency ensures that the data in the database is reliable and makes sense. It helps prevent mistakes or incorrect information from being stored in the database. Just like following the rules of a game makes it fair for everyone, maintaining consistency in a database ensures fairness and accuracy in the data.

3. **Isolation**: Transactions should be isolated from each other, which means that the operations within a transaction should not interfere with other concurrent transactions. This prevents conflicts and ensures that each transaction sees a consistent view of the data. In our online shopping example, if two customers are simultaneously adding items to their carts, the system should ensure that they do not accidentally add the same item, and each customer sees their own separate cart.

4. **Durability**: Once a transaction is successfully completed, its changes are permanent and should persist even in the event of system failures. The data changes made during the transaction should be stored securely and not lost. In our online shopping example, once you've successfully made the payment, the system ensures that the transaction is permanently recorded, and your purchase is not lost even if there is a power outage or a server crash.

By adhering to these ACID properties, transactions provide reliability and integrity to database operations. They ensure that the database remains consistent and that multiple concurrent users can work on the database without interfering with each other.

INTRODUCTION TO TRANSACTION

A transaction is like a package of actions that you do all together as if they were just one thing. For example, imagine you want to buy something from a store. The transaction includes different steps like choosing the item, paying for it, and getting a receipt. All of these actions are grouped together as one transaction.

In a database, a transaction is a similar concept. It involves a set of actions that work together as a single logical unit. These actions can include reading information from the database (like checking the price of an item), writing or updating data (like adding your purchase to the store's records), or deleting data (like removing an item from inventory).

Transactions are used in many real-life scenarios. For instance, when you withdraw money from an ATM, the transaction includes checking your account balance, subtracting the withdrawn amount, and updating the balance. Similarly, when you make a credit card purchase, the transaction involves verifying your card, deducting the amount from your account, and recording the transaction details.

Other examples of transactions include making flight reservations, checking into a hotel, scanning items at a supermarket, registering for classes at school, or receiving a bill for your purchases. Each of these scenarios involves multiple steps, and all the steps together form a transaction.

The purpose of having transactions is to ensure that everything happens correctly and reliably. It's like having a safety net that guarantees that either all the steps of the transaction are completed successfully, or none of them are. This helps maintain the integrity and consistency of the database, making sure that the information stored remains accurate and reliable.

So, transactions are like bundles of actions that are treated as a single unit. They are used to make sure that operations on the database are done correctly and consistently, and they are commonly used in various real-life situations to handle different processes smoothly.

Imagine you have a set of Lego blocks, and you want to build a specific structure. To make sure everything goes smoothly, you decide to work on one block at a time. You start by picking up a block and place it in the right spot. Once it's in the right place, you consider it completed and move on to the next block. If something goes wrong with a block, you put it back where it was before you started.

In a similar way, a transaction in a database is like working with Lego blocks. It's an atomic unit of work, which means it's a single task that needs to be done all at once or not at all. Just like completing a block before moving on to the next one, a transaction needs to be completed entirely or not done at all.

To keep track of transactions, the system needs to know when they start, when they end, and whether they are successfully completed or not. It's like having a record of when you start building with the Lego blocks, when you finish, and whether the structure is successfully completed or needs to be undone.

In application programs, transactions are separated by "begin" and "end" statements. Think of them as markers that indicate the start and end of a transaction. An application program can have several transactions, and each transaction is bounded by these "begin" and "end" statements. It's like having multiple sets of Lego blocks, and each set represents a different transaction that you're working on.

These transaction boundaries help in organizing and managing the work. They ensure that each transaction is treated as a separate unit and can be tracked individually. If something goes wrong with a transaction, it can be aborted, just like undoing the work with a set of Lego blocks if you make a mistake.

So, in simpler terms, a transaction is like working with Lego blocks, where each block represents a step in the transaction. The transaction needs to be completed entirely or not done at all. Application programs use "begin" and "end" statements to mark the boundaries of transactions, allowing multiple transactions to be managed separately. Keeping track of transactions helps in recovery and ensuring that the work is done correctly and consistently.

BASIC TRANSACTION OPERATIONS

In a transaction, there are four basic operations that can be performed on a database: read, write, insert (or update), and delete. Let's define each of these operations in simpler terms:

1. **Read**: The read operation is like looking up information in a book or searching for something on the internet. It allows you to retrieve data from the database without making any changes to it. For example, you might want to read the price of a product or check the availability of an item in a store's inventory.
2. **Write**: The write operation is like taking a pen and making changes to a book or document. It allows you to modify existing data in the database. For example, you might want to update the quantity of an item in the inventory when it is sold or change the address associated with a customer's account.

   Insert (or Update): The insert operation is like adding a new page to a book or inserting a new entry into a list. It allows you to add new data to the database. For example, you might want to insert a new customer's information when they create an account or add a new product to the inventory.

   Update is similar to insert, but instead of adding entirely new data, it modifies existing data in the database. It's like editing or correcting information in a book or document. For example, you might want to update a customer's phone number or change the price of a product.
3. **Delete**: The delete operation is like tearing a page out of a book or removing an item from a list. It allows you to remove data from the database. For example, you might want to delete a customer's account when they request to close it or remove a product from the inventory that is no longer available.

These basic transaction operations — read, write, insert (or update), and delete—are the building blocks of working with a database. They allow you to retrieve, modify, add, and remove data to perform various tasks and maintain the accuracy and integrity of the database.

"Read_item(X)" command includes the following steps :-

1. **Find the address of the disk block that contains item X:** Imagine you have a bookshelf with many books, and you want to find a specific book called X. To do that, you need to know which bookshelf and which shelf on that bookshelf holds the book X. Similarly, in a database, the "Read_item(X)" command needs to find the location (address) of the disk block where the data for item X is stored.
2. **Copy the disk block into a buffer in main memory:** Think of the buffer in main memory as a temporary storage area, like a desk or table, where you can place the book you want to read. Before you can read the book, you need to take it from the bookshelf and put it on the desk. Similarly, the database system copies the disk block that contains item X from the disk storage to a buffer in main memory, if it's not already there. This makes it easier and faster to access and manipulate the data.
3. **Copy item X from the buffer to the program variable named x:** Once the disk block is in the buffer, it's like having the book you want to read on the desk. Now, you can open the book and find the specific information you're looking for, which is item X. In the database, the system copies the specific data for item X

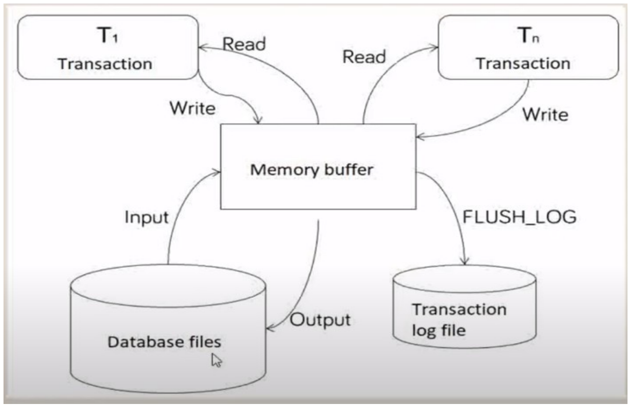
from the buffer in main memory to a program variable named x. This allows the program or application to work with the data and perform any necessary operations or calculations on it.

So, in simpler terms, the "Read_item(X)" command involves finding the location of the disk block that contains the data for item X, bringing that disk block into a temporary storage area called a buffer in main memory, and then copying the specific data for item X from the buffer into a program variable named x. This allows the program to access and work with the data effectively.

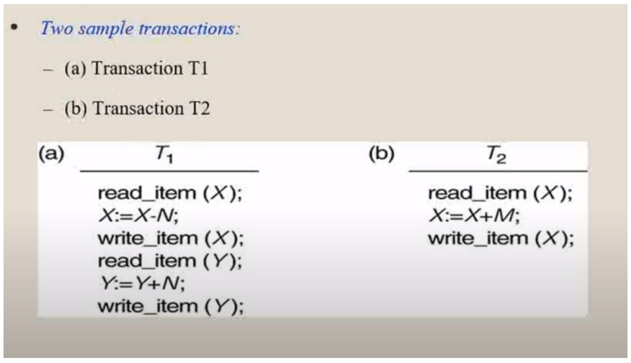
"write_item(X)" command includes the following steps :-

1. **Find the address of the disk block that contains item X:** Imagine you have a bookshelf with many books, and you want to find a specific book called X to update or make changes to it. To do that, you need to know which bookshelf and which shelf on that bookshelf holds the book X. Similarly, in a database, the "write_item(X)" command needs to find the location (address) of the disk block where the data for item X is stored.
2. **Copy the disk block into a buffer in main memory:** Think of the buffer in main memory as a temporary storage area, like a desk or table, where you can place the book you want to work on. Before you can make changes to the book, you need to take it from the bookshelf and put it on the desk. Similarly, the database system copies the disk block that contains item X from the disk storage to a buffer in main memory, if it's not already there. This makes it easier and faster to access and modify the data.
3. **Copy item X from the program variable named X into its correct location in the buffer:** Now that you have the disk block in the buffer, it's like having the book you want to work on placed on the desk. You can open the book and make changes to the specific information you're interested in, which is item X. In the database, the system copies the updated data for item X from the program variable named X and places it in the correct location within the buffer in main memory. This ensures that the changes are applied to the right place in the buffer.
4. **Store the updated block from the buffer back to disk:** Once you have made the necessary changes to the data in the buffer, it's like having the book on the desk all updated and ready to be placed back on the bookshelf. In the database, the system stores the updated disk block from the buffer back to the disk storage. This ensures that the changes you made to item X are permanently saved and stored on the disk. The updated data may be written back to the disk immediately or at a later point in time, depending on the system's implementation.

So, in simpler terms, the "write_item(X)" command involves finding the location of the disk block that contains the data for item X, bringing that disk block into a temporary storage area called a buffer in main memory, copying the updated data for item X from the program variable named X into the correct location in the buffer, and finally, storing the updated block from the buffer back to the disk storage. This ensures that the changes made to item X are correctly saved and persist in the database.



Example:- Imagine we have two transactions



- *Two sample transactions:*
  - (a) Transaction T1
  - (b) Transaction T2

```
(a)        T1                      (b)        T2
      read_item (X);                     read_item (X);
      X:=X-N;                            X:=X+M;
      write_item (X);                    write_item (X);
      read_item (Y);
      Y:=Y+N;
      write_item (Y);
```

Transaction 1:

1. read_item(x): In this step, we retrieve the current value of a variable named x from the database. It's like looking at your bank account balance to see how much money you currently have.
2. x := x - N: Here, we subtract a specific amount N from the value of x. It's like transferring or giving some money to a friend, where the value of x represents your current balance, and N represents the amount you're transferring.
3. write_item(x): This step involves updating the value of x in the database with the new result after subtracting N. It's like recording the updated balance in your bank account after transferring the money.
4. read_item(y): Now, we read the current value of another variable named y from the database. It's like checking your friend's account balance to see how much money they have.
5. y := y + N: Here, we add a specific amount N to the value of y. It's like adding the transferred amount to your friend's account, where y represents their current balance and N represents the amount received.
6. write_item(y): This step involves updating the value of y in the database with the new result after adding N. It's like recording the updated balance in your friend's account after they receive the money.

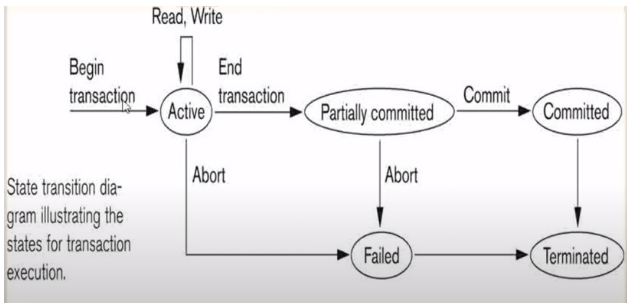
Transaction 2:

1. read_item(x): In this step, we read the current value of variable x from the database. It's like checking your own bank account balance again.
2. x := x + M: Here, we add a specific amount M to the value of x. It's like depositing or adding money to your account, where x represents your current balance and M represents the amount you're depositing.
3. write_item(x): This step involves updating the value of x in the database with the new result after adding M. It's like recording the updated balance in your bank account after making the deposit.

So, in simpler terms, transaction 1 involves reading the value of x, subtracting N from x and recording the updated value, reading the value of y, adding N to y, and recording the updated value. Transaction 2 involves reading the value of x, adding M to x, and recording the updated value.

These transactions simulate financial transactions like transferring money to a friend (transaction 1) and making a deposit to your own account (transaction 2). Each step within a transaction is performed as a single unit, ensuring consistency and integrity in the database.

TRANSACTION STATES



State transition diagram illustrating the states for transaction execution.

Active State: This is the initial state of a transaction. It indicates that the transaction has started and is currently executing its operations. It's like a person starting a task or process.

Partially Committed State: When a transaction reaches this state, it means that all the operations within the transaction have been executed successfully, and the transaction is ready to be committed. However, it has not been committed yet. It's like completing a task but not officially marking it as done.

Committed State: When a transaction is committed, it means that all its operations have been successfully executed, and the changes made by the transaction have been permanently saved in the database. It's like marking a task as completed and making it officially done.

Failed State: If a transaction encounters an error or failure during its execution, it enters the failed state. It indicates that one or more operations within the transaction were not successful. It's like encountering a roadblock or obstacle while trying to complete a task.

Aborted State: When a transaction is aborted, it means that it was unable to complete its operations successfully or encountered an error. In this state, any changes made by the transaction are discarded, and the database is rolled back to the state it was in before the transaction started. It's like undoing the work done in a task and reverting to the previous state.

Incomplete State: If a transaction is neither committed nor aborted, it is in an incomplete state. It indicates that the transaction is still in progress and has not reached a final state yet. It's like being in the middle of a task and not having completed it or encountered any errors.

Terminated State :- The "terminated" state refers to the state of a transaction after it has completed its execution, whether it has been committed or aborted. In this state, the transaction has finished its work, and its impact on the database has been finalized. It does not actively participate in any further operations or modifications.

In simpler terms, the different transaction states can be compared to the stages of completing a task. The active state is when you start working on something. The partially committed state is like completing the task but not officially marking it as done yet. The committed state is when the task is officially completed. The failed state is when you encounter an error or problem during the task. The aborted state is like undoing the work and going back to the starting point. The incomplete state is when you're still in the middle of the task and haven't finished or encountered any errors yet.

These transaction states help in managing and tracking the progress of transactions in a database, ensuring that changes are applied correctly and consistently.

DATABASE LOG RECORDS

In a database system, a log record is a mechanism used to keep a record of different actions performed by transactions. It acts as a chronological record of events, capturing the changes made to the database during transactional operations. These log records are essential for ensuring the reliability, consistency, and recoverability of the database.

Log records are typically used in conjunction with a unique transaction identifier (referred to as T in this context) that is automatically generated by the system. This transaction ID helps identify and associate each log record with a specific transaction.

There are various types of log records commonly used in database systems:

1. **[start_transaction, T]**: This log record indicates the beginning of a transaction with the unique identifier T. It serves as a marker for when a transaction starts its execution.

2. **[write_item, T, X, old_value, new_value]:** This log record represents a modification made by a transaction with identifier T to a database item X. It includes the old value and the new value of the item to track the change made by the transaction.

3. **[read_item, T, X]:** This log record signifies that a transaction with identifier T has read the value of a specific database item

X. It records the transaction's access to the item without altering its value.

4. **[commit, T]:** This log record indicates the successful completion of a transaction with identifier T. It confirms that the changes made by the transaction can be permanently recorded (committed) in the database.
5. **[abort, T]:** This log record signifies that a transaction with identifier T has been aborted or canceled. It denotes that the transaction was unable to complete successfully, and any changes made by it should be discarded.

These log records play a vital role in database management, allowing for recovery in the event of system failures, providing transactional consistency, and enabling auditing and analysis of database operations.

By capturing the sequence of events and changes made by transactions, log records help ensure data integrity, facilitate system recovery, and provide a historical trail of database activities.

DESIRABLE PROPERTIES OF TRANSACTION

Desirable properties of a transaction, often referred to as the ACID properties, are crucial for ensuring the reliability and integrity of database operations. Let's define and explain each property in simpler terms, along with examples:

Atomicity: Atomicity ensures that a transaction is treated as a single indivisible unit of work. It means that either all the operations within a transaction are completed successfully, or none of them are. It's like an "all or nothing" principle. For example, if you transfer money from one bank account to another, either the entire transfer is successful, and both accounts are updated, or no money is transferred at all.

Example :- Imagine you have two bank accounts, Account A and Account B. Account A has 5000 birr, and Account B has 8000 birr. You want to transfer 750 birr from Account A to Account B.

To perform the transfer, the transaction goes through two steps: subtracting 750 birr from Account A and adding 750 birr to Account B. However, just after subtracting the money from Account A, but before adding it to Account B, a failure occurs (like a power outage or hardware failure).

Now, the database is in an inconsistent state because the transaction was not completed. Account A lost 750 birr, but Account B did not receive it. It's like having an incomplete transaction where one account lost money, and the other account didn't receive it.

To ensure consistency, a reliable database management system (DBMS) has the capability to restore the old values. It means that the DBMS can roll back the changes made by the transaction and bring the database back to its previous state. In this case, the DBMS can restore the lost 750 birr to Account A, ensuring that the data is consistent and accurate.

In simpler terms, the example demonstrates atomicity by showing that either the entire transaction (subtracting from Account A and adding to Account B) should happen or none of it should happen. If a failure occurs in the middle, the DBMS can restore the data to maintain consistency and ensure that no money is lost in the process.

Consistency: Consistency refers to ensuring that a transaction, when executed correctly, takes the database from one valid and consistent state to another. It means that the data in the database should satisfy all the predefined rules, constraints, and integrity conditions before and after the transaction.

In the example given, before the transaction, Account A had 5000 birr, and Account B had 8000 birr. The transaction aimed to transfer 750 birr from Account A to Account B.

After the transaction is completed successfully, the new state of the database should still adhere to the integrity constraints. In this case, the updated balances are Account A with 4250 birr and Account B with 8750 birr.

The consistency property ensures that the integrity constraints are maintained. In this example, the sum of the account balances, 4250 birr (Account A) and 8750 birr (Account B), equals the initial sum of 13000 birr (5000 birr + 8000 birr). This demonstrates that the database remains consistent and satisfies the predefined rules even after the transaction.

In simpler terms, consistency means that the transaction should not violate any predefined rules or constraints. In the example, the transaction ensures that the total amount of money in the accounts remains the same (13000 birr) before and after the transaction. It confirms that the transfer of 750 birr from Account A to Account B is done correctly, maintaining the overall balance and integrity of the database.

Consistency in a database ensures that data remains reliable and accurate throughout the execution of transactions, preventing any inconsistencies or violations of predefined rules.

To ensure the consistency , the sum of the accounts before and after transaction must be the same , here 5000 + 8000 = 12,000 and 4250 + 8750 = 12,000 //

Isolation: Isolation ensures that concurrent transactions do not interfere with each other while executing simultaneously. Isolation property ensures that concurrent transactions, where multiple users access the shared database at the same time, do not interfere with each other. It means that each transaction works in isolation fashion, as if it is the only transaction running, and the data used by one transaction cannot be accessed by any other transaction until it has completed its execution.

In simpler terms, isolation ensures that transactions operate independently without interfering with each other. It's like having separate bubbles around each transaction, where they can work without any knowledge or impact of other transactions happening concurrently.

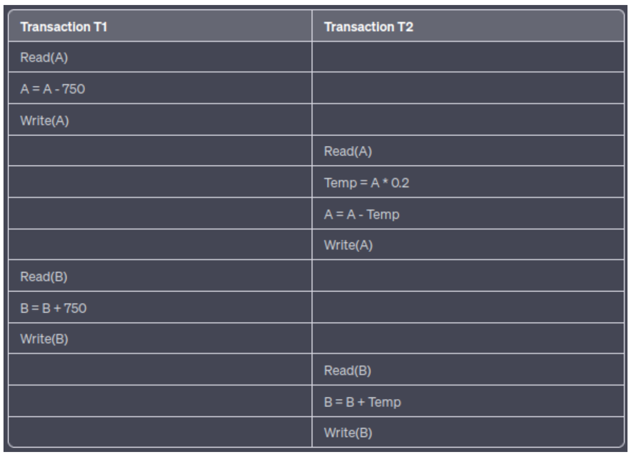
Consider a scenario where multiple users are performing transactions simultaneously on a shared database. Each transaction accesses and manipulates data, but the isolation property ensures that these transactions do not collide or interfere with each other. The changes made by one transaction should not be visible to other transactions until it has completed successfully and brought the database to a consistent state.

For example, imagine two customers making withdrawals from their bank accounts at the same time. Isolation ensures that each customer's transaction is treated independently, and the system safeguards against any issues that may arise from concurrent withdrawals. The balances and availability of funds are correctly managed for each customer, and one customer's withdrawal does not impact the other customer's transaction.

In summary, isolation property in a database ensures that concurrent transactions are isolated from one another. Each transaction operates as if it's the only one running, and changes made by a transaction are not visible to other transactions until it has successfully completed, ensuring data consistency and preventing interference or conflicts between transactions.

Example :- In this scenario, we have two concurrent transactions, T1 and T2, that involve transferring money from Account A to Account B. Transaction T1 transfers a fixed amount of $750, while Transaction T2 transfers 20% of the amount from Account A to Account B. The initial balances are Account A = $5000 and Account B = $8000.

To demonstrate the isolation property, we will show the execution of these transactions in a table. Since isolation ensures that transactions work independently, we will keep the other transaction's table empty while one transaction is executing. This helps depict the isolation between the transactions.



Transaction T1	Transaction T2
Read(A)	
A = A - 750	
Write(A)	
	Read(A)
	Temp = A * 0.2
	A = A - Temp
	Write(A)
Read(B)	
B = B + 750	
Write(B)	
	Read(B)
	B = B + Temp
	Write(B)

Description of Transactions:

Transaction T1:

1. Read(A): This step retrieves the value of Account A.
2. A = A - 750: It subtracts $750 from Account A.
3. Write(A): The updated value of Account A is written back to the database.
4. Read(B): This step retrieves the value of Account B.
5. B = B + 750: It adds $750 to Account B.
6. Write(B): The updated value of Account B is written back to the database.

Transaction T2:

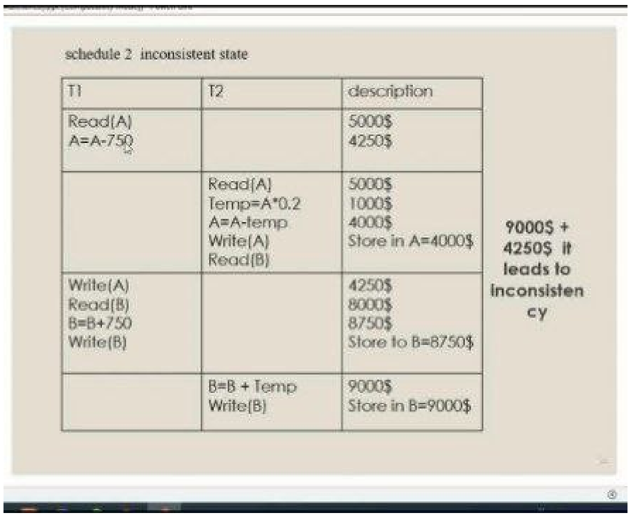
1. Read(A): This step retrieves the value of Account A.
2. Temp = A * 0.2: It calculates 20% of the value of Account A and stores it in the temporary variable Temp.
3. A = A - Temp: It subtracts the calculated Temp value from Account A.
4. Write(A): The updated value of Account A is written back to the database.
5. Read(B): This step retrieves the value of Account B.
6. B = B + Temp: It adds the value of Temp to Account B.
7. Write(B): The updated value of Account B is written back to the database.

In adherence to the isolation property, while Transaction T1 is executing, the Transaction T2 table remains empty and vice versa. This demonstrates that the transactions work independently and do not interfere with each other, ensuring isolation.

By maintaining the isolation property, the database system ensures that concurrent transactions can be executed safely and reliably without causing conflicts or inconsistencies.

Example 2 :- Let's define the scenario where two transactions, T1 and T2, do not satisfy the isolation property. Before depicting the table, let's start with a brief introduction:

In this scenario, we have two concurrent transactions, T1 and T2, that involve transferring money from Account A to Account B. However, these transactions do not adhere to the isolation property, meaning they can interfere with each other's execution. This can lead to inconsistencies and incorrect results.



schedule 2 inconsistent state

T1	T2	description

Read(A)
A=A-750
		5000\$
		4250\$

	Read(A)
	Temp=A*0.2
	A=A-temp
	Write(A)
	Read(B)
		5000\$
		1000\$
		4000\$
		Store in A=4000\$

Write(A)
Read(B)
B=B+750
Write(B)
		4250\$
		8000\$
		8750\$
		Store to B=8750\$

	B=B + Temp
	Write(B)
		9000\$
		Store in B=9000\$

9000\$ + 4250\$ it leads to inconsistency

Inconsistency leads to a data loss :- Data loss refers to the situation where information or changes made to a database are not properly recorded or stored, resulting in the loss of valuable data. It occurs when the database becomes inconsistent or experiences discrepancies, making it difficult or impossible to retrieve or recover the original or accurate information.

In the context of inconsistency, data loss can occur when transactions do not maintain the necessary integrity and fail to ensure that the database remains in a consistent state. If the transactions do not adhere to the rules, constraints, or predefined conditions, it can lead to data loss.

For example, let's consider a scenario where two transactions are attempting to update the same data simultaneously without proper isolation. If one transaction reads and modifies the data while the other transaction is in progress, it can lead to inconsistencies. In such a case, the changes made by one transaction may overwrite or disregard the changes made by the other transaction, resulting in data loss.

Data loss can have severe consequences, including incorrect or missing information, financial discrepancies, or the inability to retrieve important data. It can impact the reliability and accuracy of the database, leading to potential errors and operational issues.

To prevent data loss, it's crucial to ensure that transactions follow proper guidelines and adhere to the principles of consistency and isolation. By maintaining data integrity and avoiding inconsistencies, the risk of data loss can be minimized, ensuring that the database remains reliable and valuable for the intended purposes.

Durability: Durability ensures that once a transaction is successfully committed, its changes are permanent and will persist, even in the event of system failures or crashes. It means that the updated data is stored securely and will not be lost. For instance, when you book a flight ticket online and receive a confirmation, the durability property ensures that your ticket remains valid, even if there is a power outage or server malfunction.

In simpler terms, the ACID properties of a transaction can be explained as follows:

Atomicity: It's like an "all or nothing" rule, where a transaction is either fully completed or not done at all.
Consistency: It's like ensuring that all the data changes made by a transaction follow the predefined rules and constraints.
Isolation: It's like maintaining separate bubbles for each transaction, so they don't interfere with each other's work.

Durability: It's like guaranteeing that once a transaction is successfully completed, its changes are permanently saved and won't be lost.

These properties work together to maintain the reliability, consistency, and integrity of the database, ensuring that transactions are processed correctly and data remains accurate.

SINGLE USER AND MULTI USER USERS

Single-User System: In a single-user system, only one user interacts with the database at a time. The user has exclusive access to the database and performs transactions without interference from other users. It's like having sole ownership of the database, where only one person can make changes or access the data at any given time.

In simpler terms, a single-user system is like having exclusive control over a database. Imagine you have a personal diary that only you can write in. You can update, add, or retrieve information from the diary without anyone else accessing or modifying it.

Multi-User System: In a multi-user system, multiple users can simultaneously access and interact with the database. Each user can perform transactions independently, accessing and modifying the data concurrently. It's like having shared access to the database, where multiple people can make changes or access the data simultaneously.

In simpler terms, a multi-user system is like sharing a whiteboard with other people. Each person can write or erase information on the whiteboard at the same time, and everyone can see the changes made by others. However, to ensure data consistency and prevent conflicts, there are mechanisms in place to manage concurrent access and maintain data integrity.

In a multi-user system, the DBMS (Database Management System) handles the coordination and control of transactions to ensure that they don't interfere with each other. The DBMS manages concurrent access, enforces isolation, and ensures that transactions are executed correctly and consistently. It provides mechanisms such as locks, timestamps, and transaction scheduling algorithms to prevent conflicts and maintain data integrity.

CONCURRENCY

Concurrency refers to the ability of a system to execute multiple tasks or processes simultaneously. In the context of database systems, concurrency allows multiple transactions to be executed concurrently, enabling multiple users to access and modify the database concurrently.

Concurrency provides several benefits, such as improved system throughput, increased user responsiveness, and efficient utilization of system resources. However, it also introduces challenges, such as ensuring data integrity, managing conflicts, and preventing inconsistencies.

Now let's define interleaved processing and parallel processing, along with their advantages:

1. **Interleaved Processing:** Interleaved processing, also known as time-sharing or concurrent processing, involves executing multiple tasks or processes by interleaving their execution on a single CPU. In this approach, the CPU switches rapidly between different tasks, allocating small time slices to each task in a round-robin or priority-based manner.

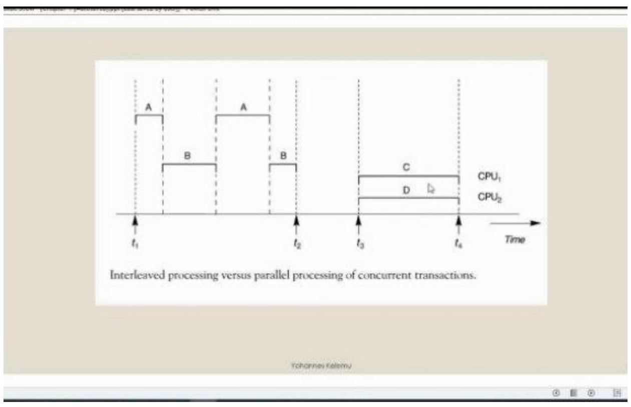
Advantages of Interleaved Processing:

- Improved responsiveness: Interleaved processing allows multiple tasks or transactions to make progress simultaneously, providing a more responsive system for users.
- Efficient resource utilization: By sharing the CPU among multiple tasks, interleaved processing maximizes the utilization of system resources, ensuring optimal performance.
- Fairness in task execution: Interleaved processing provides fair allocation of CPU time to different tasks, ensuring that each task receives a reasonable share of processing time.

2. **Parallel Processing:** Parallel processing involves the simultaneous execution of multiple tasks or processes using multiple CPUs or processor cores. In this approach, tasks are divided into smaller subtasks, and each subtask is executed on a separate CPU or core concurrently.

Advantages of Parallel Processing:

- Increased processing speed: Parallel processing allows tasks to be executed concurrently, resulting in faster processing times for large-scale computations or data-intensive operations.
- Enhanced scalability: With parallel processing, additional CPUs or cores can be added to the system, enabling scalability and accommodating increased workload or data volume.
- High-performance computing: Parallel processing is especially advantageous for tasks that can be divided into independent sub tasks, such as complex calculations, simulations, and data analytics.



Interleaved processing versus parallel processing of concurrent transactions.

Both interleaved processing and parallel processing offer advantages in different contexts. Interleaved processing is suitable for systems with a single CPU and provides improved responsiveness and resource utilization. Parallel processing, on the other hand, leverages multiple CPUs or cores to achieve higher processing speed and scalability for computationally intensive tasks.

It's important to note that the choice between interleaved processing and parallel processing depends on the system architecture, workload characteristics, and specific requirements of the application or database system.

THE WRITE OPERATION CAUSES MORE PROBLEM THAN THE READ OPERATION

The write operation can cause more problems than the read operation in concurrency control due to its potential impact on data consistency and integrity. There are a few key reasons for this:

Conflicting Updates: When multiple transactions attempt to write to the same data item concurrently, conflicts can arise. Conflicting updates occur when two or more transactions modify the same data simultaneously. This can lead to inconsistent or contradictory outcomes, making it challenging to determine the correct state of the data.

Overwriting Changes: In the case of lost updates, one transaction's changes to a data item can be overwritten by another transaction. If two transactions both read and modify the same data concurrently, one transaction's modifications may get lost when the other transaction commits its changes. This can result in data inconsistencies and the loss of valuable updates.

Rollback and Undo Operations: In some situations, a transaction may need to be rolled back or undone due to conflicts or errors during execution. When a write operation is rolled back, it needs to revert the changes it made to the data. This can be more complex and challenging than undoing a read operation, as the write operation may have altered the data in a way that affects other transactions or dependencies within the database.

Data Integrity Constraints: Write operations often involve enforcing data integrity constraints, such as primary key uniqueness, referential integrity, or other business rules. Concurrent write operations need to ensure that these constraints are maintained consistently to prevent data corruption or violation of integrity rules. Resolving conflicts and maintaining consistency while enforcing constraints can be more challenging than in read operations.

Overall, write operations can introduce more complexities and potential conflicts in concurrent environments due to their impact on data modifications and integrity constraints. Effective concurrency control mechanisms, such as locking, isolation levels, and transaction management, are employed to address these challenges and maintain data consistency and integrity in multi-user systems.

PROBLEMS OF CONCURRENT TRANSACTION

Data Inconsistency: Data inconsistency is a problem that can occur when multiple transactions are executing concurrently. It arises when the interleaved execution of transactions leads to conflicts and inconsistent outcomes. For example, if two transactions simultaneously modify the same data item, the final state of the data may be incorrect or contradictory, resulting in data inconsistency.

In simpler terms, data inconsistency means that the data becomes incorrect or contradictory when multiple transactions are executing at the same time. It can happen when transactions interfere with each other and make conflicting changes to the same data, leading to inconsistent results.

Lost Updates (Write Write Problem): Lost updates occur when multiple transactions attempt to modify the same data item simultaneously, and the changes made by one transaction are overwritten or lost due to the interleaved execution of transactions. This can lead to incorrect data or loss of valuable updates.

In simpler terms, lost updates happen when one transaction's changes to data are lost because another transaction overwrites them. It's like having two people editing the same document at the same time, and one person's changes get overwritten by the other person, resulting in lost updates.

The lost update problem occurs when the modifications made by one transaction to a data item are overwritten or lost by the update operation of another transaction before the first transaction can commit its changes. This can lead to data inconsistencies and the loss of valuable updates.

To illustrate this, let's consider a simple example using a table:

Table: Bank Account

Account Id	Balance
1	$500

Suppose we have two transactions, T1 and T2, both attempting to update the balance of Account ID 1 concurrently. The initial balance is $500.

Transaction T1:

1. Reads the current balance of Account ID 1: $500.
2. Adds $100 to the balance: $500 + $100 = $600.

Transaction T2:

1. Reads the current balance of Account ID 1: $500.
2. Adds $200 to the balance: $500 + $200 = $700.

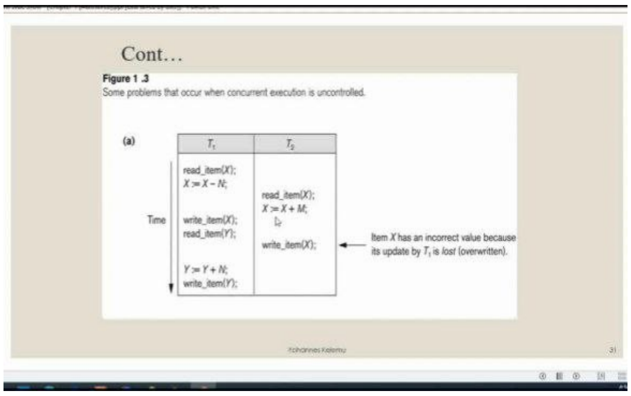
Now, let's assume the interleaved execution of these transactions:

step	Transaction T1(Balance = $500)	Transaction T2 (Balance = $500)
1	Read Balance	
2		Read Balance
3	Add $100	
4		Add $200
5	Write balance ($600)	
6		Write Balance ($700)

In this scenario, the problem arises because the update made by Transaction T1 is lost or overwritten by the update operation of Transaction T2. Transaction T1 initially reads the balance as $500, adds $100 to it, and intends to write the updated balance as $600. However, before Transaction T1 can commit its changes, Transaction T2 reads the balance as $500, adds $200 to it, and writes the updated balance as $700. As a result, the update made by Transaction T1 is lost, and the final balance is $700 instead of $600.

This demonstrates how the lost update problem can occur when concurrent transactions attempt to modify the same data, leading to the loss of valuable updates and inconsistencies in the final data state. Effective concurrency control mechanisms, such as proper locking and synchronization techniques, are needed to prevent or mitigate the lost update problem and maintain data integrity.

Example :-



Cont…

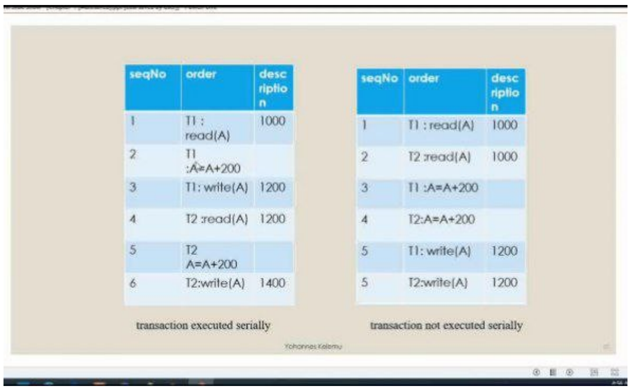
Figure 1 .3
Some problems that occur when concurrent execution is uncontrolled.

(a)

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Time
Item X has an incorrect value because its update by T_1 is *lost* (overwritten).
Yohannes Kelemu
31

Example :-



seqNo	order	description
1	T1 : read(A)	1000
2	T1 :A=A+200	
3	T1: write(A)	1200
4	T2 :read(A)	1200
5	T2 A=A+200	
6	T2:write(A)	1400

transaction executed serially

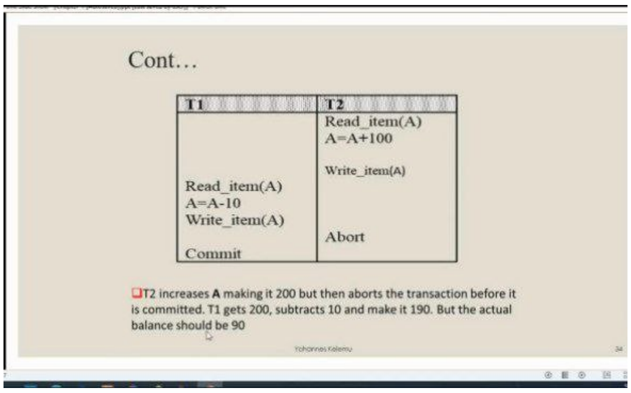
seqNo	order	description
1	T1 : read(A)	1000
2	T2 :read(A)	1000
3	T1 :A=A+200	
4	T2:A=A+200	
5	T1: write(A)	1200
5	T2:write(A)	1200

transaction not executed serially

Dirty Reads (The temporary update , inconsistent read):- A dirty read occurs when one transaction reads data that has been modified by another transaction but has not been committed yet. This can lead to reading and using incorrect or incomplete data, as the changes made by the other transaction may be rolled back or aborted later.

In simpler terms, a dirty read happens when one transaction reads data that is still in an intermediate state due to another ongoing transaction. It's like reading a draft of a document that may change later, leading to potential confusion or incorrect information.

Example :-



Cont...

T1	T2

	Read_item(A)
	A=A+100

	Write_item(A)

Read_item(A)
A=A-10
Write_item(A)	

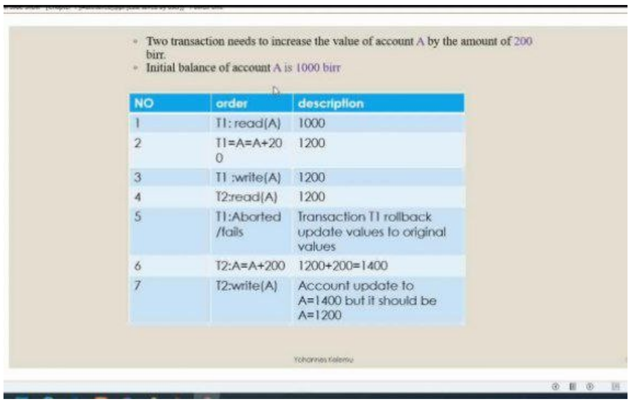
	Abort
Commit	

T2 increases **A** making it 200 but then aborts the transaction before it is committed. T1 gets 200, subtracts 10 and make it 190. But the actual balance should be 90

Yohannes Kelemu

34

Example :-



- Two transaction needs to increase the value of account A by the amount of 200 birr.
- Initial balance of account A is 1000 birr

NO	order	description
1	T1: read(A)	1000
2	T1=A=A+20 0	1200
3	T1 :write(A)	1200
4	T2:read(A)	1200
5	T1:Aborted /fails	Transaction T1 rollback update values to original values
6	T2:A=A+200	1200+200=1400
7	T2:write(A)	Account update to A=1400 but it should be A=1200

Yohannes Kelemu

The incorrect summary problem (unrepeatable read) The problem of incorrect summary occurs when one transaction is calculating an aggregate summary function, such as a total or average, on a set of records, while other concurrent transactions are updating some of these records before the first transaction is committed. As a result, the aggregate function may calculate values based on a mix of old and updated records, leading to incorrect or inconsistent summary results.

To illustrate this problem, let's consider an example:

Suppose we have a table named Sales, which tracks the sales transactions for different products:

Table: Sales

Product ID	Product Name	Quantity
1	Product A	10
2	Product B	15
3	Product C	20

Now, let's imagine two transactions, T1 and T2, operating concurrently:

Transaction T1:

1. Reads the current total quantity from the Sales table: 10 + 15 + 20 = 45.
2. Performs some other operations.

Transaction T2:

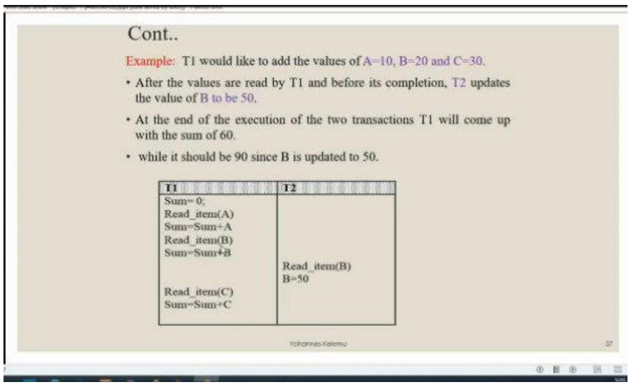
1. Updates the quantity of Product B: Quantity = 30.

Now, let's assume the interleaved execution of these transactions:

step	T1	T2
1	Read total quantity	
2		Update quantity of product B
3	Continue Execution	

In this scenario, the problem arises because Transaction T1 calculates the total quantity based on the initial values of the records before Transaction T2 updates the quantity of Product B. The result obtained by Transaction T1 is incorrect since it does not reflect the updated quantity of Product B.

Example :-



Cont..

Example: T1 would like to add the values of A=10, B=20 and C=30.

- After the values are read by T1 and before its completion, T2 updates the value of B to be 50.
- At the end of the execution of the two transactions T1 will come up with the sum of 60.
- while it should be 90 since B is updated to 50.

T1	T2
Sum= 0;	
Read_item(A)	
Sum=Sum+A	
Read_item(B)	
Sum=Sum+B	
	Read_item(B)
	B=50
Read_item(C)	
Sum=Sum+C	

Yohannes Kelemu 37

This problem occurs due to the lack of proper synchronization and isolation between the concurrent transactions. The summary function in Transaction T1 does not have a consistent view of the data, as it mixes old and updated values, leading to incorrect summary results.

To ensure the correct summary, it is essential to employ proper concurrency control mechanisms, such as transaction isolation levels and appropriate locking strategies. These mechanisms guarantee that the summary function operates on a consistent and up-to-date set of records, preventing the problem of incorrect summary and ensuring accurate results.

CHAPTER TWO

CONCURRENCY CONTROL

Concurrency control is a fundamental aspect of database management systems (DBMS) that ensures the proper execution of multiple transactions concurrently while maintaining data consistency and integrity. It involves coordinating the access and modification of shared data by multiple users or processes to prevent conflicts and inconsistencies.

In simpler terms, concurrency control is like traffic management for a busy road. Imagine a road where several cars are trying to move forward simultaneously. To prevent accidents and ensure smooth traffic flow, traffic lights, lanes, and rules are implemented. Similarly, in a DBMS, concurrency control techniques are employed to manage simultaneous access and modifications of data to avoid conflicts and maintain data integrity.

Concurrency control mechanisms in a DBMS aim to address the following challenges:

1. **Data Conflicts**: Conflicts can occur when multiple transactions try to access or modify the same data concurrently. Concurrency control ensures that conflicting operations are appropriately scheduled and executed to maintain consistency.

2. **Data Consistency**: Maintaining data consistency is crucial to ensure that the database remains in a valid state even when multiple transactions are executed concurrently. Concurrency control prevents situations where the data becomes inconsistent or violates integrity constraints.

3. **Isolation**: Isolation refers to the degree of separation between transactions. Concurrency control mechanisms ensure that each transaction appears to execute independently, as if it were the only transaction accessing the data, even though multiple transactions are running concurrently.

4. **Atomicity**: Atomicity ensures that each transaction is treated as an indivisible unit of work. Concurrency control guarantees that either all the operations within a transaction are executed successfully, or none of them are applied, preventing partial updates or incomplete transactions.

Concurrency control techniques utilize various mechanisms such as locking, timestamps, and transaction isolation levels to manage concurrent access and modifications. These mechanisms control access to shared data, resolve conflicts, enforce isolation, and ensure data integrity, thus enabling multiple transactions to execute safely and reliably.

CONCURRENCY CONTROL TECHNIQUES

Different concurrency control techniques, such as locking, timestamp, and optimistic concurrency control, using simpler terms:

1. **Locking**: Locking is a concurrency control technique that involves acquiring locks on data items to control access by multiple transactions. A lock acts like a permission slip, allowing only one transaction to read or write a data item at a time.

In simpler terms, locking is like a "keep out" sign on a door. Imagine a room that only one person can enter at a time. When someone wants to enter, they need to obtain a key (lock) to open the door. While one person has the key (lock), others have to wait until the key is released. This ensures that only one person accesses the room at any given time, preventing conflicts.

2. **Timestamp**: Timestamp-based concurrency control uses timestamps to order and control the execution of transactions. Each transaction is assigned a unique timestamp that represents its start time. The timestamps help determine the order of execution and resolve conflicts.

In simpler terms, timestamps are like a time-stamp machine that marks when each transaction starts. It ensures that transactions are executed in the right order, like people waiting in line based on the time they arrived. This helps avoid conflicts and ensures that transactions are processed in a fair and orderly manner.

3. **Optimistic Concurrency Control:** Optimistic concurrency control is a technique that assumes there will be no conflicts among concurrent transactions. It allows transactions to execute without acquiring locks initially but performs checks later to ensure that conflicts have not occurred. If conflicts are detected, the affected transactions are rolled back and retried.

In simpler terms, optimistic concurrency control is like playing a game with friends where everyone is expected to follow the rules and not interfere with each other. The game proceeds without strict restrictions or supervision. However, if any conflicts arise, such as two people trying to take the same turn, the game is reset, and they try again to ensure fairness.

These concurrency control techniques aim to prevent conflicts and ensure data consistency in multi-user systems. Locking restricts concurrent access to data items, timestamp ordering ensures transaction sequencing, and optimistic concurrency control assumes no conflicts initially but performs checks later to maintain data integrity.

Each technique has its advantages and considerations, and their selection depends on the specific requirements and characteristics of the application or database system.

LOCKING IN DETAIL

In simpler terms, the locking method is a technique used to manage multiple users or processes accessing a database at the same time. It helps maintain data integrity and prevent conflicts that may arise when multiple users try to modify the same data simultaneously.

When a user wants to read or modify data in the database, the locking method ensures that they obtain a lock on the data item they need. This lock prevents other users from accessing or modifying the same data until the first user releases the lock.

Imagine you and your friend want to update a shared document online. The locking method would be like taking turns. When you start making changes, you "lock" the document, preventing your friend from editing it at the same time. Once you finish and release the lock, your friend can then "lock" the document and make their changes.

By using locks, the locking method ensures that each user gets exclusive access to the data they need, one at a time, to avoid conflicts and maintain data consistency.

In simpler terms, the concepts of locking and unlocking in the context of transactions and data items can be explained as follows:

1. **Locking** :- When a transaction wants to access or modify a particular data item, it needs to "lock" that item. Locking means that the transaction claims exclusive ownership or control over the data item. This ensures that no other transaction can access or modify the same data item simultaneously.

Imagine you and your friend have a toy car, and you both want to play with it. To avoid conflicts, you decide to use a locking mechanism. When you want to play with the car, you "lock" it by holding onto it tightly. This prevents your friend from taking the car and playing with it until you release the lock.

2. **Unlocking** :- Once a transaction finishes using a data item, it needs to "unlock" the item. Unlocking means that the transaction releases its ownership or control over the data item, allowing other transactions to access or modify it if needed.

In the toy car scenario, when you're done playing with the car, you "unlock" it by letting go. Now, your friend can take the car and play with it because you've released the lock.

The purpose of locking and unlocking in transactions is to ensure that only one transaction can access or modify a particular data item at a time. This helps prevent conflicts and maintain the integrity and consistency of the data.

TWO LOCK MODES

In simpler terms, the shared and exclusive lock models are two types of locking mechanisms used in databases to control access to data items. Let's dive into each model in more detail:

1. **Shared Lock Model:** In the shared lock model, multiple transactions are allowed to simultaneously acquire shared locks on a data item. When a transaction holds a shared lock, it can only read the data item but cannot modify it. Other transactions can also acquire shared locks on the same data item, allowing them to read the data concurrently.

Imagine you and your friends are reading a book together. In the shared lock model, each of you can hold a shared lock on the book. This means all of you can read the book at the same time, but none of you can make changes to the text.

The shared lock model is useful when multiple transactions need to access the same data for read operations, ensuring concurrent reading without conflicts.

2. **Exclusive Lock Model:** In the exclusive lock model, only one transaction can acquire an exclusive lock on a data item at a time. When a transaction holds an exclusive lock, it has exclusive access to the data item, allowing both read and write operations. Other transactions are blocked from acquiring any type of lock on the same data item until the exclusive lock is released.

Using the book analogy again, imagine you want to write some notes in the book. In the exclusive lock model, you would acquire an exclusive lock on the book, giving you the exclusive right to make changes. This means no one else can read or modify the book until you release the lock.

The exclusive lock model ensures that only one transaction can modify a data item at a time, preventing conflicts and maintaining data integrity during write operations.

Both lock models have their uses depending on the needs of the transactions. The shared lock model allows concurrent reading but prohibits writing, while the exclusive lock model provides exclusive access for both reading and writing, ensuring data consistency during modifications.

LOCK MANAGER

In simpler terms, a lock manager is a component in a database system that is responsible for managing locks on data items. It keeps track of which transactions have locked which data items, the type of lock being held, and maintains a reference to the next data item locked.

To help in managing this information, the lock manager uses a data structure called a lock table. The lock table is like a table or a list that holds information about the locks placed on different data items.

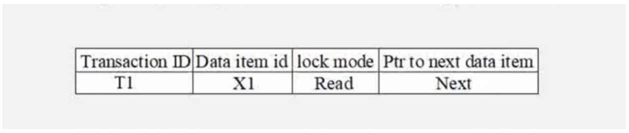
Let's break down the key components of the lock manager and the lock table:

1. **Transaction Identification:** Each transaction is assigned a unique identifier. The lock manager uses these identifiers to keep track of which transaction has locked a specific data item. It uses this information to grant or deny lock requests from other transactions.

2. **Data Item Identification**: Every data item in the database is also assigned a unique identifier. The lock manager uses these identifiers to associate locks with specific data items. It helps identify which data

items are currently locked and by which transactions.

3. **Lock Model:** The lock model refers to the type of lock being held on a data item. For example, it could be a shared lock or an exclusive lock (as explained earlier). The lock manager keeps track of the lock model for each locked data item to ensure proper access control.
4. **Pointer to the Next Locked Data Item:** The lock manager maintains a reference or a pointer to the next data item that has been locked by the same transaction. This helps in efficiently managing locks and ensuring proper order when releasing or acquiring locks.



Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

By using the lock table, the lock manager can efficiently manage and control access to data items. It keeps track of which transactions have locked which data items, the lock models being used, and maintains a reference to the next locked data item. This helps prevent conflicts, ensure data integrity, and maintain concurrency control in the database system.

A well-formed transaction follows three important rules :

1. **Locking before Reading or Writing :-** A transaction must lock a data item before it reads or writes to it. This means that before a transaction can access or modify a data item, it needs to obtain a lock on that item. Locking ensures that the transaction has exclusive access to the data item and prevents conflicts with other transactions that may also want to access or modify the same item.

Imagine you and your friend have a toy box, and you both want to play with a specific toy inside. To avoid conflicts, you decide to use locks. Before you can play with the toy, you need to "lock" it by holding onto it tightly. This ensures that your friend cannot take the toy and play with it at the same time.

2. **Not Locking an Already Locked Item :-** A transaction must not attempt to lock a data item that is already locked by another transaction. If a data item is already locked, it means that another transaction has exclusive access to it. Trying to lock it again would lead to a conflict and potential data inconsistency.

Continuing with the toy box analogy, if your friend already has the toy and has locked it by holding onto it, you cannot attempt to lock it as well. It's important to wait for your friend to release the lock before you can access the toy.

3. **Not Unlocking a Free Data Item** :- A transaction must not try to unlock a data item that is not currently locked by itself. Unlocking a free data item means releasing the lock on a data item that the transaction does not currently hold. This could lead to data inconsistencies or conflicts with other transactions.

In the toy box scenario, if you have already released the lock on the toy by letting go, it would not make sense for your friend to try to unlock it because the toy is already freely available. It's important to only unlock data items that you have previously locked.

Following these rules ensures that transactions operate in a well-formed manner, avoiding conflicts, maintaining data integrity, and preventing inconsistencies in the database system.

LOCKING BASIC RULES

In simpler terms, the basic rules of locking in a transactional system can be explained as follows:

1. **Locking an Item:** When a transaction wants to access a data item, it first requests access by issuing a "lock item(x)" operation. If the data item is already locked by another transaction, the

requesting transaction is forced to wait until the item becomes available. If the data item is currently unlocked (lock(x) = 0), the transaction sets the lock to 1 (lock(x) = 1) and is allowed to access the item.

Imagine you and your friend want to play with the same toy. To ensure fairness, you have a lock on the toy. When your friend wants to play with it, they ask for permission by saying "lock item(x)". If the toy is already locked, your friend has to wait until you finish playing. But if the toy is unlocked, your friend can set the lock to 1 and start playing with it.

2. **Unlocking an Item:** After a transaction finishes its operations on a data item, it issues an "unlock_item(x)" operation. This operation sets the lock back to 0 (lock(x) = 0), indicating that the data item is now available for other transactions to access.

Continuing with the toy example, when you finish playing with the toy, you issue an "unlock_item(x)" operation by letting go of the toy. This action sets the lock to 0, indicating that the toy is now free for your friend or other transactions to play with.

3. **Shared Lock and Exclusive Lock:** Transactions can have different types of locks on data items. A shared lock allows a transaction to read the data item but not modify it. An exclusive lock, on the other hand, gives a transaction both the ability to read and modify the data item.

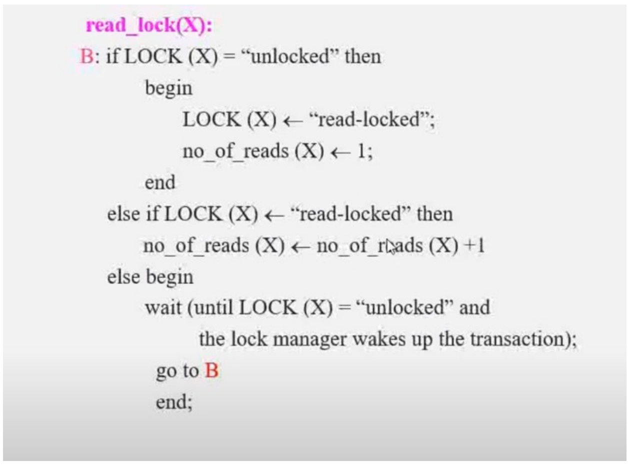
For instance, if you have a shared lock on the toy, it means you can only look at the toy and play with it, but you cannot make any changes or modifications. However, if you have an exclusive lock, you have the authority to read, modify, and even replace the toy with a different one.

4. **No Conflict in Reads:** Multiple transactions can hold shared locks on the same item simultaneously because reads do not conflict. This means that while one transaction is reading the data item, another transaction can also read the same item without causing any conflicts.

To illustrate, imagine you and your friends are reading a book together. Each of you can hold a shared lock on the book, allowing all of you to read different parts of the book simultaneously.

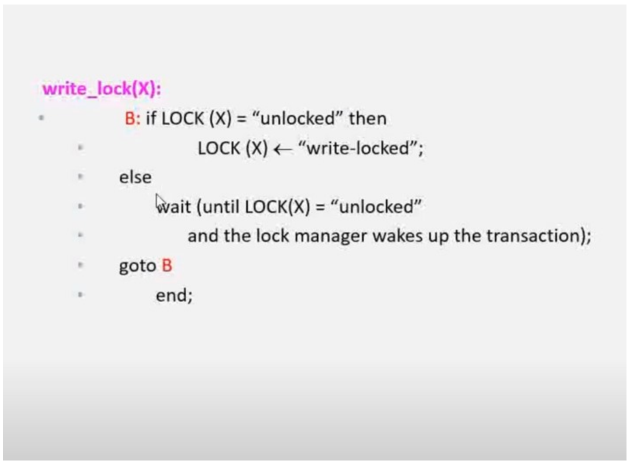
By following these locking rules, transactions can access and modify data items in a controlled and coordinated manner, preventing conflicts and maintaining data integrity in the database system.

read_lock(x) Pseudo code



```
read_lock(X):
B: if LOCK (X) = "unlocked" then
        begin
            LOCK (X) ← "read-locked";
            no_of_reads (X) ← 1;
        end
    else if LOCK (X) ← "read-locked" then
        no_of_reads (X) ← no_of_reads (X) +1
    else begin
        wait (until LOCK (X) = "unlocked" and
                the lock manager wakes up the transaction);
         go to B
         end;
```

write_lock(x) Pseudo code



```
write_lock(X):
        B: if LOCK (X) = "unlocked" then
                LOCK (X) ← "write-locked";
    else
        wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
    goto B
        end;
```

Lock Conversion

Let's break down lock conversion, lock upgrade, and lock downgrade in simpler terms:

1. **Lock Conversion**: Lock conversion refers to changing the type of lock held on a data

item by a transaction. It occurs when a transaction wants to upgrade or downgrade its lock on a data item from one type to another.

Imagine you and your friend are playing with a toy car. You have a shared lock, which means you can look at the car and play with it, but you can't make changes. However, you want to upgrade your lock to an exclusive lock, which would allow you to not only play with the car but also modify it.

To convert the lock, you would request a lock conversion operation. The system will check if the requested conversion is allowed and if there are any conflicting locks held by other transactions. If the conversion is allowed, your shared lock will be upgraded to an exclusive lock, granting you the ability to modify the toy car.

2. **Lock Upgrade:** Lock upgrade refers to changing a shared lock to an exclusive lock on a data item. It allows a transaction that initially held a shared lock to acquire exclusive access to the data item.

Continuing with the toy car example, suppose you initially had a shared lock on the car, but now you want to make modifications to it. To do so, you would request a lock upgrade. If there are no conflicting locks held by other transactions, your shared lock will be upgraded to an exclusive lock, giving you the exclusive right to modify the toy car.

3. **Lock Downgrade:** Lock downgrade, on the other hand, refers to changing an exclusive lock to a shared lock on a data item. It allows a transaction that initially held an exclusive lock to share access to the data item with other transactions.

Let's say you have an exclusive lock on the toy car, which means you have exclusive control over it. However, you realize that your friend also wants to play with the car. In this case, you can request a lock downgrade to a shared lock. If there are no conflicting locks held by other transactions, your exclusive lock will be downgraded to a shared lock, allowing both you and your friend to play with the car simultaneously.

In summary, lock conversion is the process of changing the type of lock held on a data item. Lock upgrade involves changing a shared lock to an exclusive lock, granting exclusive access. Lock downgrade, on the other hand, involves changing an exclusive lock to a shared lock, allowing shared access to the data item. These mechanisms allow transactions to adapt their lock type to accommodate their specific needs while ensuring data integrity and concurrency control in the database system.

SERIALIZABILITY

Serializability in the context of databases refers to the property of executing multiple transactions in a way that produces the same result as if the transactions were executed sequentially, one after another. In other words, it ensures that the concurrent execution of transactions does not result in any inconsistency or incorrectness in the final outcome.

To understand serializability, let's consider an example :-

Suppose there are two transactions :- T1 and T2, operating on a database. T1 transfers money from account A to account B, and T2 updates the account balance of account A.

If these transactions were executed sequentially, the outcome would be predictable and consistent. However, in a concurrent execution scenario where T1 and T2 run simultaneously, conflicts can arise. For example, T1 might read an inconsistent account balance if T2 has not yet updated it.

Serializability guarantees that the concurrent execution of transactions, even with potential conflicts, will produce a result equivalent to some sequential execution of those transactions. It ensures that the final state of the database remains consistent and valid.

To achieve serializability, concurrency control mechanisms like Two-Phase Locking (2PL) are employed. 2PL ensures that transactions acquire and release locks on data items in a consistent manner, preventing conflicts and maintaining a proper order of operations.

By enforcing serializability, database systems ensure that the execution of concurrent transactions maintains data integrity, consistency, and correctness. It allows multiple transactions to safely operate on a shared database, ensuring that the final result is equivalent to a sequential execution without any inconsistencies or conflicts.

THE SERIALIZABILITY PROBLEM

The serializability problem in read and write locks arises when there is a potential for conflicts between transactions that require both read and write access to the same data item. This can lead to inconsistencies and incorrect outcomes if not managed properly.

To understand this problem, let's consider two transactions: T1 and T2, where T1 performs a read operation (read_lock) on a data item and T2 performs a write operation (write_lock) on the same data item.

1. **Read-Write Conflict :-** The read-write conflict occurs when T1 reads a data item that T2 is currently writing to or has modified. If T1 reads the data item before T2's modifications are applied, it may obtain an inconsistent or incorrect value.

For example, let's say T1 wants to read the balance of an account while T2 is simultaneously updating that account's balance. If T1 reads the account balance before T2's update is committed, it would obtain an outdated or incorrect balance.

2. **Write-Read Conflict :-** The write-read conflict occurs when T1 writes to a data item that T2 is currently reading. This conflict can lead to T1 modifying the data item based on an outdated value read by T2, resulting in inconsistencies.

For instance, let's consider T1 updating the price of a product while T2 is reading the product's price. If T2 reads the old price before T1's update, it may perform calculations or decisions based on incorrect information.

These conflicts arise because read operations do not modify the data item and can be allowed concurrently. However, when a write operation occurs, it should have exclusive access to prevent inconsistencies.

To address the serializability problem in read and write locks, concurrency control mechanisms like Two-Phase Locking (2PL) are employed. 2PL ensures that transactions acquire appropriate locks in a consistent order, preventing read-write and write-read conflicts. This helps maintain the serializability of transactions and ensures data integrity.

By properly managing read and write locks and handling conflicts, the serializability problem can be mitigated, enabling correct and consistent execution of concurrent transactions.

TWO PHASE LOCKING TECHNIQUE

Two-Phase Locking (2PL) is a concurrency control technique used in database systems to ensure serializability and prevent conflicts between transactions. It consists of two distinct phases: the growing phase and the shrinking phase.

In simpler terms, 2PL is like a protocol or set of rules that transactions follow when accessing and modifying data. It ensures that transactions acquire and release locks in a specific order to prevent conflicts and maintain the consistency of the database.

Here's a more detailed explanation of the two phases of 2PL:

1. **Growing Phase:** During the growing phase, transactions acquire locks on data items they need before they start performing any modifications. Once a transaction acquires a lock on a data item, it holds onto that lock until it no longer needs the item.

Imagine you and your friend are playing with different toys, and you want to ensure fairness and avoid conflicts. Before you start playing with your toy, you acquire a lock on it, indicating that you are going to use it. Your friend does the same with their toy. This way, both of you have acquired locks during the growing phase, ensuring exclusive access to your respective toys.

The purpose of the growing phase is to ensure that transactions acquire all the necessary locks at the beginning and prevent conflicts later on.

2. **Shrinking Phase:** Once a transaction has finished using a data item, it releases the lock during the shrinking phase, allowing other transactions to access and modify the item.

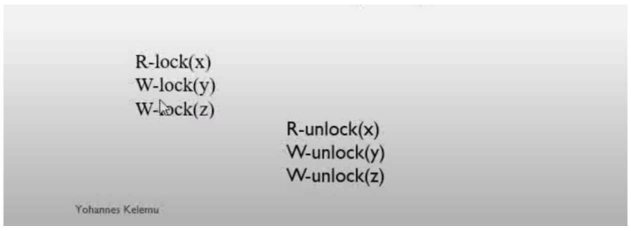
Continuing with the toy example, when you finish playing with your toy, you release the lock, indicating that you no longer need exclusive access to it. This allows your friend or any other transaction to acquire the lock and play with the toy.

The shrinking phase ensures that locks are released when they are no longer needed, allowing other transactions to access the data item and maintain concurrency.

Why do we need Two-Phase Locking?

We need Two-Phase Locking to ensure data consistency and prevent conflicts between transactions. By following the rules of 2PL, transactions acquire locks before accessing data, ensuring exclusive access to the items they need. This prevents conflicts that can arise when multiple transactions try to access or modify the same data simultaneously.

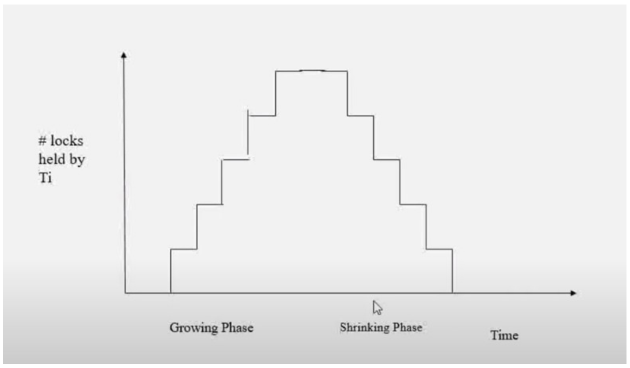
Imagine if you and your friend both tried to play with the same toy at the same time without using locks. It would lead to conflicts, confusion, and potential damage to the toy. Two-Phase Locking helps in avoiding such conflicts by enforcing a systematic approach to acquiring and releasing locks.



R-lock(x)
W-lock(y)
W-lock(z)
R-unlock(x)
W-unlock(y)
W-unlock(z)

Yohannes Kelemu

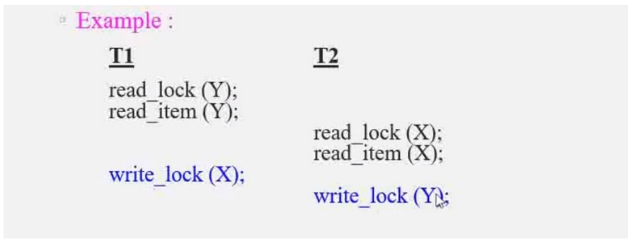
By adhering to the protocol of Two-Phase Locking, transactions can ensure that they acquire locks in a controlled manner, preventing conflicts and maintaining the integrity and consistency of the database.



Problems With Locking

Deadlock and starvation are two potential issues that can arise when using locking as a concurrency control mechanism in a database system. Let's define them in simpler terms :-

1. **Deadlock:** Deadlock occurs when two or more transactions are unable to proceed because each of them is waiting for a resource that is held by another transaction in the waiting chain. As a result, the transactions are stuck in a deadlock state, unable to make progress.



Example :

T1	T2
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
	write_lock (Y);

Imagine a scenario where two friends, Alice and Bob, each have a toy car and want to borrow the other's toy for a brief moment. Alice holds her toy car and waits for Bob to lend his car, while

Bob holds his toy car and waits for Alice to lend her car. Since both are waiting for a resource that the other is holding, they are trapped in a deadlock, unable to proceed.

In a database system, deadlock can occur when transactions hold locks on certain data items and try to acquire additional locks, but those locks are already held by other transactions in a circular manner. This can lead to a complete standstill in the system, halting progress and requiring intervention to resolve the deadlock.

2. **Starvation:** Starvation refers to a situation where a transaction is unable to execute or complete its operations because it keeps getting bypassed or delayed by other transactions. In other words, the transaction is "starved" of the necessary resources or opportunities to proceed.

Think of a scenario where there is a line of people waiting to use a single computer at a public library. If a particular person keeps getting skipped or delayed by others who are constantly joining the line, that person is experiencing starvation. They are not able to access the computer, even though they have been waiting for a long time.

In a database system, starvation can occur when a transaction repeatedly gets delayed or pushed aside by other transactions that are constantly acquiring locks on the desired data items. This can happen if there is no proper scheduling or prioritization mechanism in place.

Both deadlock and starvation are undesirable situations that can hinder the performance and efficiency of a database system. Deadlock leads to a complete standstill, while starvation results in certain transactions being delayed or unable to execute. Managing these issues requires careful design and implementation of concurrency control mechanisms, such as deadlock detection and resolution algorithms and fair scheduling policies, to ensure smooth and efficient operation of the database system.

SOLUTION TO A DEADLOCK

- **Deadlock prevention**
- **Deadlock Detection and Avoidance**
- **Lock timeouts**

DEADLOCK PREVENTION PROTOCOL

Let's define the deadlock prevention protocol, including the conservative two-phase locking and transaction time stamp, in simpler terms:

1. **Conservative Two-Phase Locking**: In the conservative two-phase locking protocol, a transaction locks all the data items it needs before it begins executing its operations. This protocol aims to prevent deadlocks by ensuring that a transaction never waits for a data item.

To understand this protocol, let's use an example: Imagine you and your friend want to borrow toys from each other. In the conservative two-phase locking approach, before you start playing with a toy, you make sure to hold all the other toys you need. This way, you won't have to wait for a toy to become available, avoiding potential deadlocks.

However, the limitation of this protocol is that it restricts concurrency. Since transactions lock all the required data items upfront, it may result in unnecessary delays if other transactions need access to some of the locked data items.

2. **Transaction Time Stamp**: In the transaction time stamp approach, each transaction is assigned a priority or time stamp when it starts. Lower priority transactions are not allowed to wait for higher priority transactions, which helps prevent deadlocks.

To understand this approach, let's imagine you and your friend want to use a computer at a library. The library assigns time stamps based on who arrived first. If you have a lower time stamp, it means you have higher priority to use the computer. Your friend, with a higher time stamp, cannot make you wait for an extended period, ensuring fairness and preventing deadlocks.

By assigning priorities to transactions based on their time stamps, the system ensures that lower priority transactions do not wait indefinitely for higher priority transactions, thereby avoiding potential deadlocks.

These deadlock prevention protocols aim to avoid deadlocks in a database system by ensuring that transactions do not get stuck waiting for resources indefinitely. While conservative two-phase locking prevents deadlocks by upfront locking, it restricts concurrency. On the other hand, the transaction time stamp approach assigns priorities to transactions, ensuring fairness and preventing lower priority transactions from waiting indefinitely for higher priority ones.

WAIT - DIE | WOUND -WAIT

Wait-die and wound-wait are two deadlock prevention strategies used in transactional systems to manage potential deadlocks. They help ensure the system avoids deadlocks by allowing transactions to either wait or be preempted based on their age or priority.

Let's define wait-die and wound-wait in simpler terms:

1. **Wait-Die:** Wait-die is a deadlock prevention strategy that follows a conservative approach. In this strategy, when a younger transaction requests a resource held by an older transaction, the younger transaction waits until the older transaction releases the resource. However, if an older transaction requests a resource held by a younger transaction, the older transaction is allowed to "die" (abort) and be restarted later.

To understand this strategy, imagine you and your friend are waiting in a queue to use a computer. If you arrive later (younger transaction) and find that your friend is already using the computer (older transaction), you would wait for your friend to finish before you can access the computer. However, if you arrived earlier (older transaction) and your friend wants to use the computer (younger transaction), you would allow your friend to "die" (abort) and restart their task later, while you continue using the computer.

Wait-die ensures that younger transactions wait for older transactions to release resources, preventing potential deadlocks while maintaining fairness. Younger transactions may experience some delays but are eventually allowed to execute.

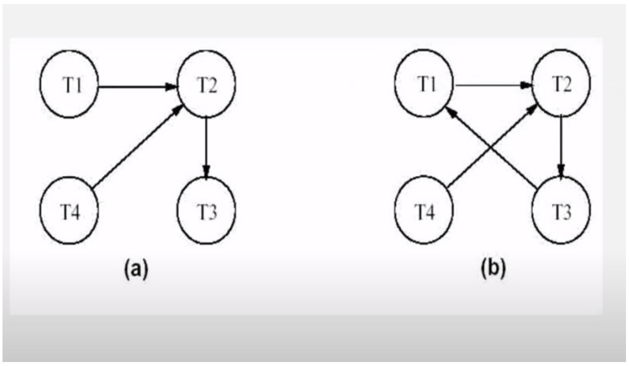
2. **Wound-Wait:** Wound-wait is another deadlock prevention strategy that follows a more aggressive approach compared to wait-die. In this strategy, when a younger transaction requests a resource held by an older transaction, the older transaction is "wounded" (aborted), allowing the younger transaction to proceed. However, if an older transaction requests a resource held by a younger transaction, the younger transaction waits until the older transaction releases the resource.

To understand this strategy, let's go back to the queue for using a computer. If you arrive later (younger transaction) and find that your friend is already using the computer (older transaction), you would wound (abort) your friend's task, allowing you to access the computer immediately. However, if you arrived earlier (older transaction) and your friend wants to use the computer (younger transaction), you would make your friend wait until you finish using the computer.

Wound-wait takes a more aggressive approach by prioritizing younger transactions over older ones. It allows younger transactions to proceed by preempting older transactions, reducing potential delays for the younger transactions.

Both wait-die and wound-wait strategies aim to prevent deadlocks by managing resource requests and ensuring fairness. These strategies determine whether a transaction should wait or be aborted based on its age or priority relative to other transactions. By employing these strategies, transactional systems can avoid deadlocks and maintain the overall integrity and concurrency of the system.

DEADLOCK DETECTION AND RESOLUTION



LOCK TIMEOUTS

lock timeouts can be considered as a mechanism for preventing deadlocks in a database system. It is a concurrency control technique that helps avoid potential deadlocks by setting a maximum duration for a transaction to wait for a lock on a data item.

In simpler terms, lock timeouts work by imposing a time limit on how long a transaction can wait to acquire a lock. If the transaction exceeds the specified time limit and is unable to acquire the lock, it is automatically aborted or rolled back, releasing any previously acquired locks.

To understand this concept, imagine you and your friend are waiting to borrow a book from a library. The library enforces a lock timeout policy, meaning if you wait for too long to borrow the book, the library staff will cancel your request and allow the book to be borrowed by someone else.

Similarly, in a database system, when a transaction requests a lock on a data item, it waits for a certain duration. If the lock is not granted within that timeframe, the transaction is terminated and treated as if it had encountered a deadlock. This helps prevent transactions from waiting indefinitely and potentially causing a deadlock situation.

By using lock timeouts, the database system ensures that transactions do not remain stuck waiting for locks indefinitely, minimizing the chances of deadlocks and allowing the system to continue executing transactions smoothly.

While lock timeouts provide a preventive measure against deadlocks, it's important to set the timeout duration carefully to avoid unnecessarily aborting transactions that could eventually acquire the required locks. Balancing the timeout duration ensures efficient concurrency while reducing the likelihood of deadlocks in the system.

TIMESTAMP BASED CONCURRENCY CONTROL

Let's delve into timestamp-based concurrency control in simpler terms:

In a database system, timestamp-based concurrency control uses timestamps to determine the order of executing transactions and resolve conflicts. Here's a more detailed explanation:

1. Timestamp Assignment: Each transaction is assigned a unique timestamp that represents its start time. This timestamp is like a "time-stamp machine" that marks when the transaction began its execution. It helps establish a chronological order for the transactions.

Imagine you and your friend enter a store, and the store manager gives you both a unique timestamp when you arrive. Your timestamp indicates the exact time you entered the store, while your friend receives a timestamp representing their arrival time.

2. Transaction Execution Order: The timestamps are used to determine the order in which transactions are executed. The database system processes transactions based on their timestamps, ensuring a systematic and ordered execution.

Continuing with the store analogy, after receiving the timestamps, you and your friend line up to make your purchases. The store serves customers based on their timestamp order, starting with the customer with the earliest timestamp. This ensures fairness and an orderly execution sequence.

3. Conflict Resolution: Timestamps also play a crucial role in resolving conflicts that may arise when multiple transactions try to access or modify the same data concurrently. Conflicts occur when there is contention for shared resources or conflicting operations.

If a conflict occurs, the transaction with the earlier timestamp is given priority and allowed to proceed, while the transaction with the later timestamp is either delayed or rolled back, depending on the conflict resolution policy.

To illustrate this, imagine you and your friend are shopping in the store, and there is only one item left of a popular product. Both of you have timestamps indicating the order of arrival. The store gives priority to the customer with the earlier timestamp, allowing them to purchase the item, while the customer with the later timestamp must wait or choose an alternative product.

By using timestamps, the database system ensures a fair and ordered execution of transactions. It helps avoid conflicts, resolves contention for resources, and maintains data consistency and integrity. Transactions are processed in a systematic manner based on their timestamps, preventing inconsistencies and ensuring that the final state of the database reflects a correct sequence of operations.

VALIDATION (OPTIMISTIC) CONCURRENCY CONTROL SCHEMES

Optimistic concurrency control is a technique used in database systems that assumes there will be no conflicts among concurrent transactions. It allows transactions to proceed without acquiring locks initially, but it performs checks later to ensure that conflicts have not occurred. If conflicts are detected during the checks, the affected transactions are rolled back and retried.

To understand this concept better, let's use an analogy of playing a game with friends:

Imagine you and your friends are playing a game together, but there is no strict supervision or restrictions on how you can play. Everyone is expected to follow the rules and not interfere with each other. You proceed with the game, assuming there won't be any conflicts.

However, if a conflict does arise, such as two people trying to take the same turn or conflicting moves, the game is reset, and everyone tries again to ensure fairness. In this case, the conflicting moves are rolled back, and the game restarts from a consistent state.

In the context of a database system, optimistic concurrency control works similarly. Transactions are allowed to execute without acquiring locks initially, assuming there won't be any conflicts. It relies on the idea that conflicts between transactions are relatively rare. After the transactions complete their operations, the system performs checks to ensure that no conflicts have occurred. If conflicts are detected during the checks, the affected transactions are rolled back and retried to maintain data integrity.

Optimistic concurrency control offers the advantage of allowing more concurrent access to data items, as transactions are not hindered by locks during execution. It aims to strike a balance between concurrency and data consistency by assuming that conflicts are infrequent. However, it requires careful checking and potential retries to handle conflicts if they do occur.

Each concurrency control technique, including locking, timestamp ordering, and optimistic concurrency control, serves the purpose of preventing conflicts and maintaining data consistency in multi-user systems. The choice of technique depends on the specific requirements and characteristics of the application or database system.

CHAPTER THREE

QUERY PROCESSING AND OPTIMIZATION

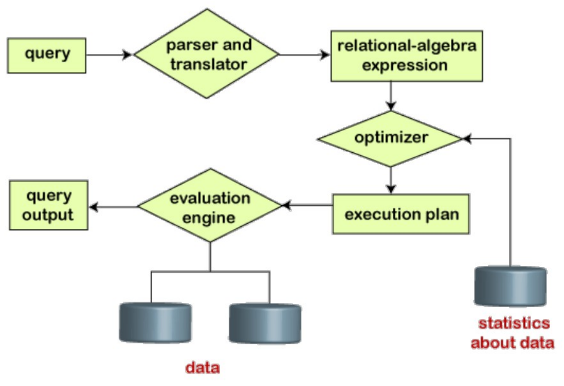
Query processing refers to the set of activities performed by a database management system (DBMS) to process and execute a user's query against a database. It involves transforming a high-level query, written in a database query language (e.g., SQL), into a series of operations that retrieve the desired data from the database.

- **Parsing and Translation:**
  - Parsing: The query is analyzed to ensure it follows the syntax and grammar rules of the database query language (e.g., SQL). The query is broken down into its constituent parts, such as keywords, table names, columns, and conditions.
  - Translation: The parsed query is then translated into an internal representation that the database system can understand and execute. This representation typically involves generating a query tree or an intermediate representation of the query structure.

- **Optimization:**
  - Optimization: The database system analyzes the query and considers different execution strategies to determine the most efficient way to retrieve the data. This involves exploring various possibilities, such as different access paths, join algorithms, and index usage, to minimize the overall cost of executing the query.
  - Cost Estimation: The system estimates the cost associated with each potential execution plan, considering factors like disk I/O, CPU utilization, and memory usage. The goal is to select the execution plan with the lowest estimated cost.

- **Evaluation:**
  - Plan Execution: The selected execution plan from the optimization phase is used to retrieve the data from the database. The system executes the query by accessing the required tables, applying any necessary joins, filtering rows based on conditions, and performing any requested aggregations or sorting operations.
  - Result Generation: As the query is executed, the system generates the result set based on the specified columns and conditions. The retrieved data is formatted into a format suitable for the user or application, such as a table or a result set object.
  - Result Delivery: The final result set is delivered to the user or application that initiated the query. This may involve transmitting the result over a network connection or storing it in a temporary location for further processing or presentation.

Overall, query processing involves parsing and translating the user's query, optimizing the query execution plan to minimize cost, and finally executing the query to fetch the requested data from the database. It is a critical component of database systems that enables efficient retrieval of information based on user queries.



Steps in query processing

PARSING AND TRANSLATION

When we talk about parsing and translation in the context of query processing, it refers to the initial stages where the user's query is transformed from a high-level language like SQL into a format that the database system can understand and execute.

Here's a simplified explanation of the concepts involved:

1. **Parsing :-** Parsing is the process of analyzing the query to ensure it follows the syntax and rules of the query language (e.g., SQL). It breaks down the query into its individual components, such as keywords, table names, columns, and conditions. This step ensures that the query is properly structured and can be understood by the database system.
2. **Translation** :- Once the query is parsed, it needs to be translated into a format that the system can work with effectively. Although SQL is convenient for humans to write and understand, it may not be directly suitable for the internal operations of the database system. So, the query is further translated into an internal representation, often using relational algebra.

Relational algebra is a mathematical language that describes operations on relational databases. It provides a formal framework for representing queries and performing operations like projection, selection, join, and aggregation. By translating the user's query into relational algebra expressions, the database system can optimize and execute the query more efficiently.

Overall, parsing and translation involve ensuring the query's syntax is correct, breaking it down into its components, and translating it into an internal representation (such as relational algebra) that the database system can work with effectively. These steps lay the foundation for further query processing activities, such as optimization and evaluation, which lead to the retrieval of the desired data from the database.

When a user executes a query, the database system goes through a process of parsing and translation to understand and process the query effectively. Here's a simpler breakdown:

1. **Parsing**: The parser checks the syntax of the query to ensure it follows the rules and structure of the query language. It verifies the names of relations (tables) in the database, the tuples (rows), and the required attribute values (columns). It creates a tree-like structure called a parse tree, representing the query's structure.
2. **Translation**: The parse tree is then translated into an internal representation, often in the form of relational algebra. Relational algebra is a mathematical language used to describe operations on relational databases. This translation involves replacing the use of views and transforming the query into a set of relational algebra expressions.

For example, let's consider the query in SQL: "SELECT emp_name FROM Employee WHERE salary > 10000;"

To make the system understand the query, it needs to be translated into relational algebra. In this case, the translation could be represented as follows:

$\sigma salary > 10000 (\pi salary (Employee))$

$\pi_{salary} (\sigma_{salary>10000}(Employee))$

This translation represents the selection operation (σ) where the salary attribute is filtered based on the condition salary > 10000, and then the projection operation (π) is applied to retrieve the emp_name attribute from the resulting set.

Once the query is translated into relational algebra, the database system can execute each operation using different algorithms. These algorithms determine the most efficient way to retrieve the requested data from the database.

Overall, query processing starts with parsing the query to check its syntax and structure. Then, the query is translated into relational algebra to create an internal representation. This internal representation allows the system to perform operations on the database efficiently and retrieve the desired data.

QUERY EXECUTION PLAN

A query execution plan is a detailed set of instructions that a database system creates to execute a user's query and retrieve the requested data. It outlines the steps and operations the system needs to follow in order to efficiently process the query.

Here's a breakdown in simpler terms:

1. **Understanding the Query:** The query execution plan starts with the database system understanding the user's query, which is written in a database query language like SQL. The system analyzes the query to determine the tables involved, the requested columns, any specified conditions, and the desired result.
2. **Query Optimization:** Once the system understands the query, it optimizes the query execution plan. This optimization process involves considering different ways to retrieve the data, evaluating possible access paths, join algorithms, and other factors to find the most efficient plan. The goal is to minimize the cost of executing the query in terms of time and resources.
3. **Plan Generation:** After the optimization phase, the database system generates the query execution plan. This plan outlines the sequence of steps and operations needed to retrieve the data and produce the result. It specifies the order of table access, join operations, filtering conditions, and any necessary aggregations or sorting.
4. **Operation Execution:** With the query execution plan in hand, the database system starts executing the plan step by step. It performs operations like table scans, index lookups, joins, filtering rows based on conditions, aggregations, and sorting. Each operation is carried out according to the instructions provided by the plan.
5. **Result Generation:** As the operations are executed, the database system generates the result set based on the specified columns, conditions, and requested aggregations. It formats the retrieved data into a suitable format, such as a table or result set object.

In simpler terms, a query execution plan is like a detailed recipe for the database system. It outlines the optimized steps and operations needed to retrieve the requested data from the database and produce the desired result. The system follows this plan to efficiently execute the query and generate the output that the user expects.

QUERY OPTIMIZATION

Query optimization refers to the process performed by a database system to generate an efficient plan for executing a user's query. The goal is to minimize the cost and time required to evaluate the query and retrieve the desired data.

Here's a breakdown in simpler terms:

1. **Understanding Query Cost:** The cost of evaluating a query can vary depending on factors like the complexity of the query, the amount of data involved, and the available resources. The database system is responsible for constructing the evaluation plan, so users do not need to worry about writing their queries in an optimized manner.
2. **Generating Efficient Evaluation Plan:** The database system uses query optimization techniques to generate an efficient

evaluation plan for the query. The system considers different possibilities, such as access paths, join algorithms, and index usage, to find the plan that minimizes the overall cost of executing the query.

3. **Cost Analysis:** To optimize a query, the query optimizer estimates the cost of each operation involved. This estimation considers factors such as memory allocations, execution costs, and other resources required by different operations. By analyzing the estimated costs, the system can choose the most efficient plan.
4. **Plan Selection:** After analyzing the costs, the query optimizer selects the evaluation plan that minimizes the overall cost of executing the query. The chosen plan outlines the sequence of operations and optimizations that will be used to retrieve the data.
5. **Query Execution:** Once the evaluation plan is selected, the database system executes the query. It follows the chosen plan and performs operations like accessing tables, joining data, applying conditions, and aggregating results.
6. **Producing Output:** As the query is executed, the system generates the output of the query, which is the desired result. The result is formatted and presented to the user in a suitable form, such as a table or result set.

In simpler terms, query optimization is the process performed by the database system to generate an efficient plan for executing a query. The system analyzes the query, estimates the cost of different operations, selects the best plan, and executes the query to produce the desired output. The aim is to minimize the time and resources required for evaluating the query.

MEASURES OF QUERY COST

The cost of a query refers to the time it takes for the query to interact with the database and return the desired result. It includes various sub-tasks and the time taken for each of these tasks. Here's a breakdown in simpler terms:

1. **Query Processing Time:** The cost of the query involves the time taken to process the query from start to finish. This includes tasks such as parsing and translating the query (making sure it follows the correct syntax and structure), optimizing the query (finding the most efficient way to execute it), evaluating the query (executing the operations and retrieving the data), and finally returning the result to the user.
2. **Execution Time:** Once the query is optimized, executing it involves accessing both primary and secondary memory based on the file organization method. The time taken to retrieve the data can vary depending on the file organization and the use of indexes. Different file organization methods and the presence of appropriate indexes can impact the efficiency of retrieving the data and affect the overall query execution time.
3. **Sub-tasks and Time:** The cost of the query considers the time taken by each sub-task involved in query processing. These sub-tasks include parsing, translation, optimization, evaluation, and result retrieval. The time spent on each task adds up to the overall cost of the query.
4. **Fraction of Seconds:** While the cost of a query is often measured in a fraction of seconds, it's important to note that it involves multiple tasks and the time taken by each of them. Even though it may seem quick, the cumulative time spent on each task contributes to the total query cost.

In simpler terms, the cost of a query is the time it takes for the query to interact with the database and return the result. It includes various tasks like parsing, translation, optimization, evaluation, and result retrieval. The execution time depends on factors such as file organization and the use of indexes. Although it may be measured in a fraction of seconds, it represents the cumulative time spent on each task involved in query processing.

ESTIMATING QUERY COST

The cost estimation of a query evaluation plan involves calculating the net estimated cost by determining the cost of each operation within the plan and combining them. The cost estimation is done based on various resources, including the number of disk accesses, CPU execution time, and communication costs in distributed or parallel database systems.

When evaluating a query, the majority of the time is spent on accessing data from memory. The cost of access time includes factors such as disk I/O time, CPU time, and network access time. Among these factors, disk I/O time is the most significant and takes the majority of the time during query processing. Other times can be disregarded compared to disk I/O time.

To calculate the disk I/O time, typically only two factors are considered: seek time and transfer time. Seek time refers to the time taken by the processor to find a single record in the disk memory. Transfer time is the time taken by the disk to return the fetched result back to the processor or user.

For example, if we need to find the student ID of a student named 'John,' the processor will fetch the relevant data from memory based on the index and file organization method. The time taken by the processor to locate John's ID on the disk is known as seek time. The time taken by the disk to transfer the fetched result back to the processor or user is called transfer time and is represented by tT.

To calculate the disk I/O cost, we consider the number of seeks (S) required to fetch a record and the number of blocks (B) that need to be returned to the user. The disk I/O cost is then calculated as:

(S * tS) + (B * tT)

In simpler terms, the disk I/O cost represents the time taken to search and find records on the disk. It considers the seek time for locating specific data and the transfer time for returning the fetched data. The disk I/O cost is a significant factor in determining the overall cost of accessing data from the disk during query processing.

CHAPTER FOUR

DATABASE RECOVERY

Database recovery refers to the process of restoring a database system to a consistent and usable state after a failure or an error. It involves recovering lost or damaged data and ensuring the database's integrity and reliability.

In simpler terms, imagine you have a digital filing cabinet where you store important information. Sometimes, accidents happen, such as power outages, hardware failures, or software errors. These accidents can cause your files to become corrupted or even disappear completely.

Database recovery is like having a backup plan for your filing cabinet. It's the process of fixing or restoring your files if something goes wrong. It ensures that your data is protected and can be retrieved in case of any mishaps. Recovery involves identifying and correcting errors, recovering lost data, and making sure everything is back in order so you can continue using your database smoothly.

PURPOSE OF DATABASE RECOVERY

The purpose of database recovery is to fix any problems caused by failures or errors and bring the database back to a state where it was consistent and reliable before the issue occurred.

Let's imagine you're using an online banking system to transfer funds from one account to another. But suddenly, the system crashes right in the middle of the transaction. As a result, the accounts involved in the transfer may have incorrect values or incomplete changes.

In this situation, database recovery comes into play. It aims to restore the database to a point before the transaction started, ensuring that the incorrect changes are undone, and the accounts are back to their previous correct values. The recovery process ensures that the transaction's properties, like the money being transferred fully or not at all (atomicity), the accounts remaining in a valid state (consistency), and the transaction being isolated from other transactions (isolation), are preserved.

By recovering the database, we make sure that any inconsistencies or errors caused by failures are corrected, and the system can continue functioning properly, providing accurate and reliable data.

TYPES OF FAILURES

Here are the types of failures in a database system explained in simpler terms:

1. **Hardware Failure:** This type of failure occurs when there is a problem with the physical components of the computer system that houses the database. It could be a malfunctioning hard drive, power supply failure, or even a complete system crash. Hardware failures can result in data loss or corruption.

2. **Software Error**: Software errors refer to bugs, glitches, or programming mistakes in the database management system (DBMS) or the applications that interact with the database. These errors can cause the database to behave unexpectedly, leading to data inconsistencies or system crashes.

3. **System Crash:** A system crash happens when the entire computer system abruptly stops working or becomes unresponsive. It can be caused by hardware failures, software errors, or external factors like power outages. A system crash can result

in the loss or corruption of data that was being processed at the time of the crash.

4. **Natural Disasters:** Natural disasters like earthquakes, floods, fires, or hurricanes can physically damage the infrastructure where the database system is stored. Such disasters can lead to the destruction of hardware, data loss, or make the system temporarily or permanently inaccessible.
5. **Human Error:** Human error refers to mistakes made by individuals interacting with the database system. It could include accidental deletion of important data, entering incorrect commands, or improperly configuring the system. Human errors can have significant consequences and may require database recovery to restore the system to a valid state.
6. **Network Failure:** Network failures occur when there is a disruption in the communication between the database server and the client systems. It can be due to issues like network outages, connectivity problems, or misconfigurations. Network failures can cause interruptions in database operations and lead to data inconsistencies.

These different types of failures can occur independently or in combination, posing risks to the integrity and availability of the database system. Understanding these failures helps in implementing appropriate measures for database recovery, such as backups, redundancy, and fault-tolerant systems, to mitigate the impact of failures and ensure the smooth functioning of the database.

TRANSACTION LOG

Transaction log, also known as a transaction log file or simply a log, is a vital component of a database management system (DBMS) that records all the changes made to a database. It serves as a chronological record of transactions, providing a detailed history of database modifications.

In simpler terms, imagine the transaction log as a journal or diary that keeps track of all the activities happening in a database. Whenever a transaction (such as an update, insertion, or deletion of data) takes place, the log captures the details of that transaction.

The transaction log contains the following information:

1. **Transaction Identification:** Each transaction is assigned a unique identifier, known as a transaction ID, which helps in identifying and tracking the specific transactions.
2. **Operation Details:** The log records the specific operations performed within a transaction, such as the changes made to data (e.g., updated values, inserted records, or deleted entries).
3. **Timestamp:** The log also includes timestamps, indicating when each operation took place. This information helps in maintaining the chronological order of the transactions.
4. **Before and After Values**: For update and deletion operations, the log stores the old value (before the change) and the new value (after the change). This allows for undoing or redoing the changes during database recovery.

The transaction log serves multiple purposes:

1. **Recovery**: The primary purpose of the transaction log is to aid in database recovery. In the event of a system failure, the log is used to restore the database to a consistent state by redoing committed transactions and undoing incomplete or uncommitted transactions.
2. **Rollback and Undo Operations:** The log enables the rollback or undoing of transactions. By analyzing the log, it is possible to revert the database to a previous state by undoing the changes made by a specific transaction or a series of transactions.

3. **Durability**: The transaction log ensures the durability property of the ACID (Atomicity, Consistency, Isolation, Durability) properties of a transaction. The log allows for the recovery of committed transactions even in the presence of failures, ensuring that changes are permanently saved and not lost.

4. **Replication and High Availability:** Transaction logs are crucial in database replication and high availability scenarios. By replicating the log to multiple database servers, changes made in one server can be applied to others, ensuring data consistency and availability.

Overall, the transaction log acts as a reliable and detailed record of database modifications, providing a mechanism for recovering data in case of failures and enabling the database system to maintain integrity and consistency.

DATA UPDATES

When it comes to data updates in a database, there are different approaches that can be used to manage the changes made to data items. Let's break down the four mentioned approaches in simpler terms:

1. **Immediate Update**: In immediate update, as soon as a data item is modified in the cache (temporary memory), the corresponding disk copy is immediately updated. It ensures that the changes are reflected on the disk right away, providing a consistent view of the data. This approach ensures that the disk always contains the most up-to-date version of the data.

2. **Deferred Update**: In deferred update, the modified data items in the cache are not immediately written to the disk. Instead, they are held in the cache until a specific condition is met. This condition can be when a transaction ends its execution or after a fixed number of transactions have completed their execution. Once the condition is met, the changes made to the data items are then written to the disk.

3. **Shadow Update:** In shadow update, the modified version of a data item does not overwrite its disk copy directly. Instead, it is written to a separate location on the disk, creating a shadow copy. This allows both the original (disk) and modified (cache) versions of the data item to coexist simultaneously. The system maintains a pointer or reference to the shadow copy, enabling efficient retrieval of the updated data when needed.

4. **In-place Update:** In in-place update, the disk version of the data item is directly overwritten by the cache version. When a modification is made to a data item in the cache, the corresponding data item on the disk is replaced with the updated value. This approach ensures that only one version of the data item exists at a time, and the disk copy is updated immediately with the latest changes.

   In in-place update, the term "in-place" refers to the fact that the modified version of a data item is directly written back to the same location on the disk where the original data item was stored. The disk version is overwritten by the cache version, effectively replacing the old value with the updated value.

Here's a simplified explanation of how in-place update works:

1. Original Data: Initially, the data item exists on the disk in its original form, and a copy is loaded into the cache for processing.

2. Modification: When a modification is made to the data item in the cache (e.g., changing the value of a field), the cache version of the data item is updated to reflect the changes.

3. Update on Disk: In an in-place update, the cache version is directly written back

to the same location on the disk where the original data item resides. This means the disk copy is replaced with the modified version.

4. One Version: After the update is complete, there is only one version of the data item, and it exists on the disk. The cache copy may still be present, but it is synchronized with the disk copy.

In-place update is a straightforward approach because it eliminates the need for additional disk space or storage locations to store multiple versions of the data item. It allows for efficient utilization of disk space and simplifies the update process. However, it's important to note that in-place updates can have implications for data recovery in the event of failures since the original data is directly overwritten.

Overall, in-place update involves directly replacing the original data item on the disk with the modified version from the cache, ensuring that only one version of the data item exists at a time.

what is the difference between immediate update and in-place update ?

The main difference between immediate update and in-place update lies in how the updates are handled and when they are written to the disk. Let's explore these two concepts in more detail:

Immediate Update:

- In immediate update, as soon as a data item is modified in the cache (temporary memory), the corresponding disk copy is immediately updated.
- The modification made in the cache is directly propagated to the disk, ensuring that the disk always contains the most up-to-date version of the data item.
- Immediate update provides immediate consistency between the cache and the disk. Once the update is performed, the changes are durable and available for future transactions to access.

In-Place Update:

- In in-place update, the disk version of the data item is overwritten by the cache version.
- When a modification is made to a data item in the cache, the cache version replaces the original data item directly at the same disk location where it was stored.
- In-place update eliminates the need for additional storage locations or versions of the data item, as there is only one version on the disk.
- This approach allows for efficient utilization of disk space and simplifies the update process by directly overwriting the original data.

To summarize, the main difference is that immediate update focuses on ensuring that the disk copy is immediately updated when a modification is made in the cache, while in-place update focuses on directly overwriting the original data item on the disk with the modified version from the cache. Both approaches aim to maintain consistency between the cache and the disk, but they differ in how and when the updates are written to the disk.

These different approaches to data updates offer flexibility and trade-offs in terms of performance, consistency, and reliability. The choice of the update method depends on factors such as the specific requirements of the application, the system's ability to handle concurrent updates, and the need for data consistency and durability.

DATABASE RECOVERY TECHNIQUES

Database recovery techniques are essential components of a database management system (DBMS) that ensure data integrity and consistency in the face of failures or errors. When a failure occurs, such as a system crash or a power outage, it can result in data loss or corruption. Database recovery techniques help bring the database back to a consistent state, restore lost or damaged data, and preserve the properties of transactions, such as atomicity, consistency, isolation, and durability (ACID). These techniques include undo and redo operations, write-ahead logging, shadow paging, checkpoints, and deferred updates. Each technique plays a specific role in recovering the database, either by reverting incomplete transactions or reapplying committed changes from transaction logs. By understanding and utilizing these recovery techniques, database administrators can ensure the reliability and availability of their databases.

1) **Undo and Redo Operations: Undo and redo operations are fundamental recovery techniques.**

Undo Operation: The undo operation is like a "do-over" button for a transaction that wasn't completed or committed. When a transaction starts but encounters an error or is interrupted, the undo operation allows us to reverse the changes made by that transaction and restore the database to its previous state.

In simpler terms, imagine you're playing a game, and you make some moves, but then realize you made a mistake. The undo operation lets you go back to a point before you made the mistake and erase the incorrect moves, effectively restoring the game to its earlier state.

In database recovery, the undo operation works similarly. The transaction log keeps track of all the changes made by a transaction. If the transaction is incomplete or uncommitted due to a failure, the undo operation uses the transaction log to identify the changes made by that transaction and applies the opposite operations to undo those changes. This way, the database is brought back to the state it was in before the transaction started.

Redo Operation: The redo operation is like replaying the actions of committed transactions during the recovery process. It ensures that any changes made by committed transactions are re-applied to the database to maintain consistency and durability.

In simpler terms, let's say you're writing a story, and you have been saving each chapter as you complete it. Unfortunately, your computer crashes, and you lose all your progress. However, you have a backup of your work up until the last saved chapter. The redo operation allows you to restore your story by replaying the actions from the backup and applying the changes you made in each chapter to get back to where you left off.

In database recovery, the redo operation works similarly. By analyzing the transaction log, the database management system (DBMS) identifies committed transactions that were not yet applied to the database due to a failure. The redo operation re-applies the changes made by those committed transactions, using the transaction log as a guide. This ensures that the database is brought to a consistent state by incorporating the changes that were previously committed but not yet applied.

When we say "reapply" in the context of database recovery, it means to perform the same changes or modifications again to the database. Reapplying changes involves executing the same operations that were previously performed by a committed transaction.

To understand it better, let's consider a simple example:

Suppose you have a database table called "Employees" with various columns like "Name," "Age," and "Salary." Let's say there was a committed transaction that increased the salary of all employees by 10%. However, due to a failure or error, these changes were not yet applied to the database.

During the recovery process, the redo operation identifies this committed transaction through the transaction log. It then re-executes the specific operation, which is to increase the salary of all employees by 10%, and applies it to the database. By doing so, the changes made by the committed transaction are reapplied to the database, ensuring that the salary increase is correctly reflected in the updated data.

In simpler terms, reapplying changes means taking the recorded modifications from the transaction log and executing them again on the database to bring it back to the state it was in before the failure occurred. It ensures that any committed changes that were not yet applied due to a failure are properly incorporated, maintaining data consistency and integrity.

Reapplying changes is an important step in database recovery as it ensures that all the committed transactions are correctly reflected in the database, preventing any inconsistencies or missing updates that may have occurred due to the failure.

Overall, the undo operation reverses incomplete or uncommitted changes, while the redo operation reapplies committed changes, both utilizing the transaction log to guide the recovery process and restore the database to a consistent state after failures or errors.

2) Write-Ahead Logging (WAL) Write-Ahead Logging (WAL) is a database recovery method that ensures data consistency and recoverability in the event of a system crash or failure. It involves recording log records before modifying the corresponding data in the database.

Here's a step-by-step breakdown of how WAL works:

1. **Log Records:** A log record is a sequential entry in a log file that captures the details of each database operation. These operations typically include insertions, updates, and deletions. Each log record contains information such as the operation type, the affected data item, and the new or old values of the data item.

2. **Logging Process:** Before any modifications are made to the database, the system first writes the corresponding log records to stable storage. Stable storage refers to a reliable, non-volatile storage medium that preserves data even in the event of a system crash or power failure. Writing the log records to stable storage ensures their durability.

3. **Data Modifications**: After the log records are written, the system proceeds with modifying the actual data in the database. This can involve updating, inserting, or deleting data items based on the user's actions or application requirements.

4. **Recovery Scenario**: In the event of a system crash or failure, the database needs to be recovered to a consistent state. The WAL method allows the system to restore the database by analyzing and applying the log records.

5. **Recovery Process**:

   a. Redo Phase: During the recovery process, the system starts with the redo phase. It scans the log from the most recent checkpoint or stable point (where all modified data was flushed to disk) to the end of the log. For each log record encountered, it re-applies the operation to the corresponding data item, bringing it up to date with the most recent changes.

   b. Undo Phase: Once the redo phase is complete, the system proceeds with the undo phase. Here, the log records are analyzed in reverse order, starting from the end of the log and moving towards the checkpoint or stable point. If necessary, the system rolls back the incomplete or uncommitted transactions, restoring the database to a consistent state before the crash occurred.

By following this WAL approach, the database system ensures that log records are written to stable storage before the corresponding data changes are made. This guarantees that the log can be used during recovery to reapply committed changes and undo uncommitted changes, thereby maintaining data consistency and recoverability.

Overall, WAL is an essential mechanism for reliable and efficient database recovery, enabling the system to recover from failures and restore the database to a consistent state based on the logged operations

3) Shadow Paging :-

Shadow Paging is a database recovery method that involves creating a shadow or temporary copy of the database during the recovery process. This shadow copy provides a consistent view of the database while the actual database is being modified. Once the modifications are complete, the system swaps the shadow copy with the original database.

Here's a step-by-step breakdown of how Shadow Paging works:

1. **Initial Snapshot:** The recovery process starts with taking an initial snapshot or copy of the entire database. This snapshot serves as the starting point for recovery and represents a consistent state of the database at that particular moment.
2. **Shadow Pages:** Shadow Paging uses a technique called "copy-on-write." Instead of modifying the original database pages directly, any modifications made during the recovery process are written to new shadow pages. These shadow pages are separate from the original pages and remain intact throughout the recovery process.
3. **Transaction Execution**: As the system executes transactions or modifies data during the recovery process, it uses the shadow pages for those modifications. The shadow pages allow the system to keep the original database intact and ensure a consistent view of the data during recovery.
4. **Page Mapping**: To keep track of the mapping between the original database pages and the corresponding shadow pages, a separate page table or mapping table is maintained. This table stores the mapping information, indicating which shadow page corresponds to each original page.
5. **Completing Modifications:** Once all the modifications are complete and the recovery process is finished, the system performs a swap operation. It replaces the original database with the shadow copy, making the modifications permanent and updating the data in the actual database.
6. **Cleaning Up:** After the swap, the shadow pages and the mapping table are no longer needed. They can be discarded or reused for future recovery processes.

By utilizing shadow paging, the database system ensures that the modifications made during recovery do not affect the original database directly. The shadow copy allows for a consistent view of the data during recovery, and the swap operation finalizes the modifications and updates the database accordingly.

Overall, Shadow Paging provides a mechanism for recovering the database without directly modifying the original data. It creates a temporary shadow copy, allows modifications to be made on the shadow copy, and then swaps it with the original database to complete the recovery process.

4) CHECKPOINTS

Checkpoints are specific points in the transaction log of a database system where the state of the database is recorded.

In simpler terms, imagine you're reading a long book, and you want to mark your progress so that if you need to stop and come back later, you know where to continue. To do this, you place a bookmark at a certain page. This way, when you want to resume reading, you can start from the bookmarked page instead of going through the entire book again.

In database systems, checkpoints serve a similar purpose. They act as bookmarks or designated points in the transaction log where the database management system (DBMS) records the state of the database. The DBMS takes a snapshot of the database at these checkpoints and saves this information, including the current state of the database and other relevant metadata.

During the recovery process, when the DBMS needs to bring the database to a consistent state after a failure or crash, it starts from the last recorded checkpoint. By doing so, it reduces the amount of transaction log that needs to be processed for recovery. The DBMS then applies the logged transactions recorded after the checkpoint, ensuring that the database is restored to its last consistent state.

Checkpoints are valuable for improving recovery efficiency and reducing the time required to bring the database to a consistent state. Instead of examining the entire transaction log, the DBMS can start from a known checkpoint and selectively process the transactions that occurred after that point. This selective processing minimizes the recovery time and resources required, making the recovery process more efficient.

In summary, checkpoints are specific points in the transaction log where the DBMS records the state of the database. During recovery, the DBMS starts from the last checkpoint, allowing it to selectively process only the transactions that occurred after that point. Checkpoints help improve recovery efficiency by reducing the amount of log to be processed and reducing the time required to bring the database to a consistent state.

6) **Deferred updates, or lazy updates (NO UNDO /REDO)**, is a strategy in database systems where the actual modifications to the database are postponed until the transaction commits.

In simpler terms, imagine you're making changes to your room's decorations. Instead of immediately gluing new decorations to the walls, you decide to write down your planned changes on a piece of paper. Then, if you change your mind or want to revert the changes, you can simply discard the paper without permanently altering the room.

In database systems, deferred updates work similarly. When a transaction makes modifications to the database, instead of immediately writing those changes directly to the database, the changes are stored temporarily in the transaction log.

This approach ensures that the modifications are not immediately applied to the database itself. The changes are recorded in the transaction log, providing a record of the pending modifications until the transaction completes and commits.

During the recovery process, the DBMS examines the transaction log and determines the fate of the recorded changes. If the transaction committed, meaning it was successful and should be applied, the changes recorded in the log are re-applied to the database. On the other hand, if the transaction rolled back or was unsuccessful, the changes in the log are discarded, ensuring that they do not affect the database.

Deferred updates offer flexibility and reliability. They allow transactions to make temporary modifications without immediately impacting the database. By deferring the actual updates until the transaction commits, the system ensures that only the successful changes are applied, and unsuccessful changes are effectively discarded.

In summary, deferred updates, or lazy updates, are a strategy where modifications to the database are postponed until the transaction commits. Instead of immediately writing changes to the database, the modifications are stored temporarily in the transaction log. During recovery, the changes recorded in the log are either re-applied to the database if the transaction committed or discarded if the transaction rolled back. This approach provides flexibility and ensures that only successful changes affect the database.

CHAPTER FIVE

PARALLEL AND DISTRIBUTED DATABASE SYSTEMS

Parallel and distributed database systems are advanced database architectures designed to handle large-scale data processing and storage across multiple computers or nodes. These systems are capable of distributing data and processing tasks to achieve high performance, scalability, fault tolerance, and availability. They are commonly used in modern data-intensive applications, such as big data analytics, cloud computing, and web-scale services.

Key Characteristics of Parallel and Distributed Database Systems:

1. **Parallelism**: These systems leverage parallel processing techniques to execute tasks concurrently across multiple processors or nodes. By dividing the workload and processing tasks in parallel, they can achieve higher performance and faster data processing.
2. **Distribution**: Data is distributed across multiple nodes in the system, allowing for efficient storage and retrieval. Distribution can be achieved through various techniques, such as partitioning, replication, or sharding.
3. **Scalability**: Parallel and distributed database systems are designed to scale horizontally by adding more nodes to the system. This allows them to handle increasing amounts of data and processing demands.
4. **Fault Tolerance**: These systems incorporate mechanisms to handle failures and ensure data integrity and availability. Techniques like data replication, redundancy, and fault detection and recovery mechanisms are employed to mitigate the impact of failures.
5. **Consistency and Concurrency Control**: Maintaining consistency and ensuring correct data access across distributed nodes is crucial. These systems use concurrency control protocols, such as locking, timestamp ordering, or optimistic concurrency control, to handle concurrent data access and ensure data consistency.

A distributed database refers to a collection of multiple databases that are spread across different computers connected through a network. A distributed database management system is a software system that handles this distributed database while ensuring that the distribution is transparent to the user.

Distributed databases are distinct from the files found on the Internet. Web pages consist of files stored on various nodes on the Internet, but they don't possess the same functionalities as a database system, such as uniform query processing and transaction management.

To differentiate a distributed database from a multiprocessor system with shared storage, certain conditions must be met:

1. The databases are connected through a computer network, with multiple sites or nodes. These sites are linked via a communication network that facilitates data and command transmission between them.
2. The connected databases are logically related, meaning that the information stored in them has some form of logical interrelation.
3. There is no requirement for all connected nodes to be identical in terms of data, hardware, or software. They can differ from one another.

The sites within a distributed database can be located physically close to each other (e.g., in the same building or adjacent buildings) and connected through a local area network (LAN). Alternatively, they can be geographically dispersed across large distances and linked via a wide area network (WAN). Different types of networks, such as wireless hubs, cables, telephone lines, or satellites, can be used individually or in combination.

The network's topology, which determines the communication paths between sites, can significantly impact performance, influencing strategies for distributed query processing and database design. However, for high-level architectural considerations, the specific network type is less important than ensuring that each site can communicate directly or indirectly with all other sites.

ADVANTAGE OF DISTRIBUTED DATABASE

Distributed databases offer several advantages for organizations. Here are some key benefits explained in simpler terms:

Improved ease and flexibility of application development: Developing and maintaining applications becomes easier when using a distributed database. The transparency of data distribution and control allows organizations with geographically distributed sites to work on applications seamlessly.

Increased reliability and availability: A distributed database system isolates faults to their specific site of origin, preventing them from affecting other databases connected to the network. If one site fails, other sites can continue to operate, ensuring data reliability and availability. Replicating data and software at multiple sites further enhances reliability. In contrast, a centralized system would become completely unavailable if a single site fails.

Improved performance: Distributed databases optimize performance by keeping data closer to where it is most frequently needed. This localization reduces contention for computing resources and minimizes access delays, especially in wide area networks. With a large database distributed across multiple sites, each site contains a smaller local database, resulting in faster queries and transactions. Additionally, executing queries in parallel across multiple sites improves overall performance.

Easier expansion: Distributed database systems offer flexibility for system expansion. Adding more data, increasing database sizes, or incorporating additional processors becomes simpler in a distributed environment. It enables organizations to scale up their system smoothly as their needs grow.

By leveraging distributed databases, organizations can enhance application development, achieve higher reliability and availability, improve performance, and facilitate system expansion. These advantages contribute to more efficient and effective data management in distributed environments.

TYPES OF DISTRIBUTED DATABASES

A homogeneous distributed database system is a network of two or more databases that use the same type of database management system (DBMS) software and can be stored on one or more machines. In simpler terms, it is a system where multiple databases, running the same software, are connected together to form a distributed network.

In a homogeneous distributed database system, data can be accessed and modified simultaneously on several databases within the network. This means that if a change is made to the data in one department, it will be automatically updated in other departments as well. This ensures consistency and synchronization of data across all the sites.

For example, let's consider a scenario where three departments within an organization are using Oracle-9i as their DBMS software. In this case, each department will have its own database, but all the databases will be connected and synchronized. If any changes are made in one department's database, those changes will be reflected in the databases of the other departments as well.

To achieve this level of cooperation and synchronization, all the sites within the distributed system must run the same type of DBMS software. Each site needs to be aware of the existence of other sites and agree to cooperate in processing user queries and exchanging information about transactions.

It's important to note that in a homogeneous distributed database system, local sites surrender some of their autonomy in terms of their right to change schemas (the structure of the database) or DBMS software. This is necessary to maintain compatibility and coordination between all the sites in the network.

Overall, a homogeneous distributed database system allows for the sharing and management of data across multiple databases, all running the same DBMS software, ensuring consistent and synchronized access to information.

2) A heterogeneous distributed database system is a network of two or more databases that use different types of database management system (DBMS) software and can be stored on one or more machines. In simpler terms, it is a system where multiple databases, running different software, are connected together to form a distributed network.

In a heterogeneous distributed database system, data can be accessed and shared among several databases within the network with the help of generic connectivity methods such as ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity). These connectivity standards enable communication and data exchange between different types of DBMS software.

For example, imagine a scenario where you have a network of databases represented by different DBMS software, such as Oracle, MySQL, and MongoDB. In this case, each site within the network can access and interact with data from other sites using ODBC or JDBC. This allows for data sharing and integration across different types of databases.

However, there are some challenges in a heterogeneous distributed database system. Unlike homogeneous systems, where all sites run the same DBMS software, in a heterogeneous system, each site may run a different DBMS, which means they may not be aware of each other. This lack of awareness can limit cooperation and coordination in transaction processing.

Another challenge arises from the differences in schemas (the structure of the database) among the different DBMS software. These schema differences can create difficulties in processing transactions that involve multiple databases. It may require additional efforts and transformations to ensure compatibility and consistency of data across the heterogeneous system.

In summary, a heterogeneous distributed database system involves connecting multiple databases running different DBMS software, allowing data sharing and accessibility through generic connectivity methods. However, the system may face challenges related to site awareness, limited cooperation in transaction processing, and schema differences among the databases.

ARCHITECTURE OF DISTRIBUTED DBMS

Distributed database management systems (DBMS) are designed to manage data spread across multiple computers or servers. They employ different architectures to handle the distribution and processing of queries. Here's a simplified explanation of the three common architectures: client-server, collaborating server, and middleware.

1. **Client-Server Architecture:-** In the client-server architecture, there are multiple clients (users or applications) and a few servers connected in a network. Clients directly connect to specific servers and access their data by sending direct queries. When a client sends a query, it is solved by the earliest available server, which then sends the result back to the client. This architecture is simple to implement and execute because it relies on a centralized server system.

2. **Collaborating Server Architecture:-** In the collaborating server architecture, a single query is executed on multiple servers. The servers work together to break down the single query into multiple smaller queries, each executed on different servers. The

results from these queries are then combined and sent back to the client. In this architecture, the database servers collaborate and support indirect queries, meaning they can query each other. This architecture allows for distributed processing of queries and can improve performance.

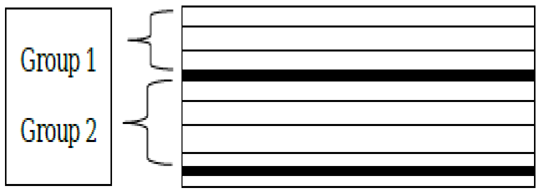
3. **Middleware Architecture:-** The middleware architecture is designed to execute a single query on multiple servers as well. However, it differs from the collaborating server architecture by having a dedicated server called middleware. This middleware server acts as a mediator between the client and multiple independent database servers. It manages queries and transactions from the client and distributes them to the appropriate servers. The middleware architecture utilizes local servers to handle local queries and transactions. Middleware software is employed to execute queries and transactions across one or more independent database servers.

In summary, the client-server architecture involves direct client-server interactions, with clients sending queries to specific servers. Collaborating server architecture allows servers to work together to process a single query. Middleware architecture uses a dedicated server to manage queries and transactions across multiple servers. These architectures enable distributed database management and enhance scalability, performance, and data availability in a networked environment.

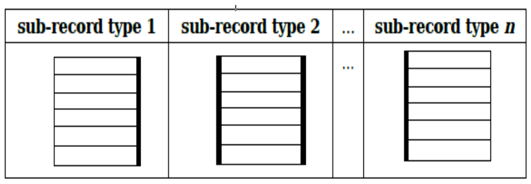
PRINCIPLE OF DATA DISTRIBUTION

In a distributed database system, there are different ways to divide the data and tables into smaller parts. This division is known as fragmentation. Two common types of fragmentation are horizontal fragmentation and vertical fragmentation.

1. **Horizontal fragmentation:** It involves dividing a table into subsets of rows or tuples. Each subset contains a portion of the original table's data. This fragmentation splits the table based on certain criteria, such as specific values in a column or ranges of values. For example, if we have an employee table, horizontal fragmentation may split it into subsets based on departments, where each subset contains employees from a particular department.



2. **Vertical fragmentation:** It involves dividing a table into subsets of columns. Each subset represents a portion of the original table's attributes or columns. This fragmentation splits the table based on specific attributes or columns. For example, if we have an employee table, vertical fragmentation may split it into subsets, such as one subset containing basic employee information (e.g., name, ID) and another subset containing additional details (e.g., address, salary).



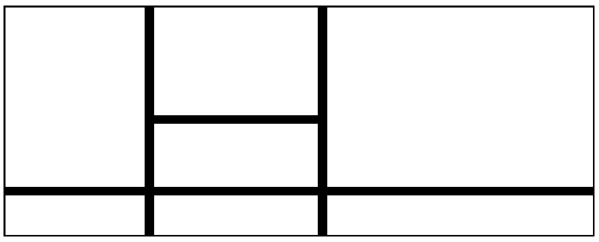
3) HYBRID FRAGMENTATION

The third principle of data distribution in a distributed database system is called hybrid fragmentation. It combines aspects of both horizontal and vertical fragmentation to achieve a more flexible and efficient data division.

In hybrid fragmentation, a table is divided into subsets of rows (horizontal fragmentation) and each subset is further divided into subsets of columns (vertical fragmentation). This allows for more granular control over which data is distributed and how it is organized.

With hybrid fragmentation, you can divide a table based on specific criteria, such as department, just like in horizontal fragmentation. However, within each subset, you can also select which columns or attributes to include, similar to vertical fragmentation.

For example, let's consider an employee table. With hybrid fragmentation, you could divide the table horizontally based on departments, creating subsets of employees for each department. Within each department subset, you can further divide the columns based on attributes, such as having one subset with basic employee information and another subset with additional details.

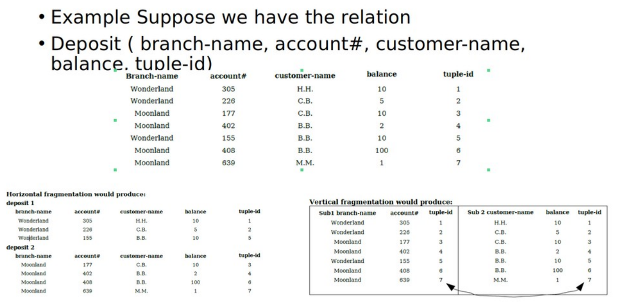


This hybrid approach provides greater flexibility in distributing and organizing data in a distributed database system. It allows for more efficient access and retrieval of data by selecting only the relevant subsets and columns for a given query or transaction.

In summary, hybrid fragmentation combines elements of both horizontal and vertical fragmentation. It allows for dividing a table into subsets of rows (horizontal) and further dividing each subset into subsets of columns (vertical). This approach provides flexibility and efficiency in distributing and organizing data in a distributed database system.

When a user submits a global query that involves the entire distributed database, it needs to be transformed into multiple fragment queries to retrieve data from the appropriate fragments. Fragmentation transparency ensures that the user is unaware of the existence of these individual fragments and can interact with the database as if it were a single entity.

Example :-



- Example Suppose we have the relation
- Deposit (branch-name, account#, customer-name, balance, tuple-id)

Branch-name	account#	customer-name	balance	tuple-id
Wonderland	305	H.H.	10	1
Wonderland	226	C.B.	5	2
Moonland	177	C.B.	10	3
Moonland	402	B.B.	2	4
Wonderland	155	B.B.	10	5
Moonland	408	B.B.	100	6
Moonland	639	M.M.	1	7

Horizontal fragmentation would produce:

deposit 1

branch-name	account#	customer-name	balance	tuple-id
Wonderland	305	H.H.	10	1
Wonderland	226	C.B.	5	2
Wonderland	155	B.B.	10	5

deposit 2

branch-name	account#	customer-name	balance	tuple-id
Moonland	177	C.B.	10	3
Moonland	402	B.B.	2	4
Moonland	408	B.B.	100	6
Moonland	639	M.M.	1	7

Vertical fragmentation would produce:

Sub1 branch-name	account#	tuple-id
Wonderland	305	1
Wonderland	226	2
Moonland	177	3
Moonland	402	4
Wonderland	155	5
Moonland	408	6
Moonland	639	7

Sub 2 customer-name	balance	tuple-id
H.H.	10	1
C.B.	5	2
C.B.	10	3
B.B.	2	4
B.B.	10	5
B.B.	100	6
M.M.	1	7

TYPES OF DATA FRAGMENTATION

Let's define the types of data fragmentation, including horizontal, vertical, and hybrid fragmentation, in simpler terms:

1. **Horizontal Data Fragmentation**: Horizontal fragmentation involves dividing a table into subsets of rows or tuples. Each subset contains a portion of the original table's data. This division is done based on specific criteria, such as values in a particular column or ranges of values. It ensures that each subset has all the columns of the original table, maintaining its completeness.

For example, if we have an employee table, horizontal fragmentation may split it into subsets based on departments, where each subset contains employees from a specific department. This helps organize the data based on different groups or categories.

Example :- In Example 1, let's say we have a database schema for students, and we want to maintain the details of all students studying Computer Science at the School of Computer Science separately. In this case, the database designer would horizontally fragment the database.

This means that the original STUDENT table would be divided into subsets or fragments based on certain criteria. Specifically, the database designer would create a new table called COMP_STD by selecting all the rows from the STUDENT table where the course is "Computer Science."

The resulting COMP_STD table would only contain the details of students enrolled in the Computer Science course. By performing this horizontal fragmentation, we ensure that the School of Computer Science has a separate table specifically dedicated to maintaining the details of Computer Science students.

In Example 2, we have an Account table with columns such as Acc_No, Balance, Branch_Name, and Type. Let's assume that there are multiple values for the Branch_Name column, such as Pune, Baroda, and Delhi.

Suppose we want to retrieve all the accounts that belong to the "Baroda" branch. In this case, the query would be written as follows:

SELECT * FROM ACCOUNT WHERE Branch_Name = "Baroda"

This query retrieves all the rows from the ACCOUNT table where the Branch_Name is "Baroda." It helps in accessing specific accounts associated with the Baroda branch.

To summarize, horizontal fragmentation involves dividing a table into subsets based on specific criteria, such as course enrollment or branch location. By horizontally fragmenting the database, we can create separate tables or subsets that hold specific data based on the defined criteria. This helps in organizing and managing data more efficiently in a distributed database system.

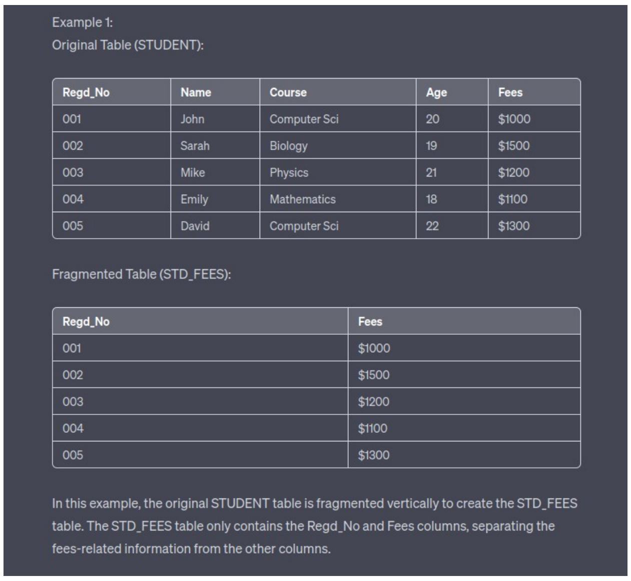
2. **Vertical Data Fragmentation:** Vertical fragmentation involves dividing a table into subsets of columns. Each subset represents a portion of the original table's attributes or columns. This division is based on specific attributes or columns of the table.

For example, if we have an employee table, vertical fragmentation may split it into subsets such as one subset containing basic employee information (e.g., name, ID) and another subset containing additional details (e.g., address, salary). This allows for more efficient storage and retrieval of data, as each subset focuses on a specific aspect of the table.

Example :-

In Example 1, let's consider a University database that keeps records of all registered students in a table called STUDENT. Now, the fees details are maintained in the accounts section. To perform vertical fragmentation, the database designer would create a new table called STD_FEES.

The STD_FEES table would be created by selecting only the required columns from the STUDENT table, specifically the Regd_No (registration number) and Fees columns. This means that the new table would contain only these two columns from the original STUDENT table. The purpose of this vertical fragmentation is to separate and store the fees-related information in a separate table, allowing for more efficient access and management of that specific data.


Example 1:
Original Table (STUDENT):

Regd_No	Name	Course	Age	Fees
001	John	Computer Sci	20	$1000
002	Sarah	Biology	19	$1500
003	Mike	Physics	21	$1200
004	Emily	Mathematics	18	$1100
005	David	Computer Sci	22	$1300

Fragmented Table (STD_FEES):

Regd_No	Fees
001	$1000
002	$1500
003	$1200
004	$1100
005	$1300

In this example, the original STUDENT table is fragmented vertically to create the STD_FEES table. The STD_FEES table only contains the Regd_No and Fees columns, separating the fees-related information from the other columns.

In Example 2, we have two fragments of vertical fragmentation. Fragmentation1 selects all the columns from the table Acc_NO, and Fragmentation2 selects all the columns from the table Balance.

Fragmentation1: SELECT * FROM Acc_NO

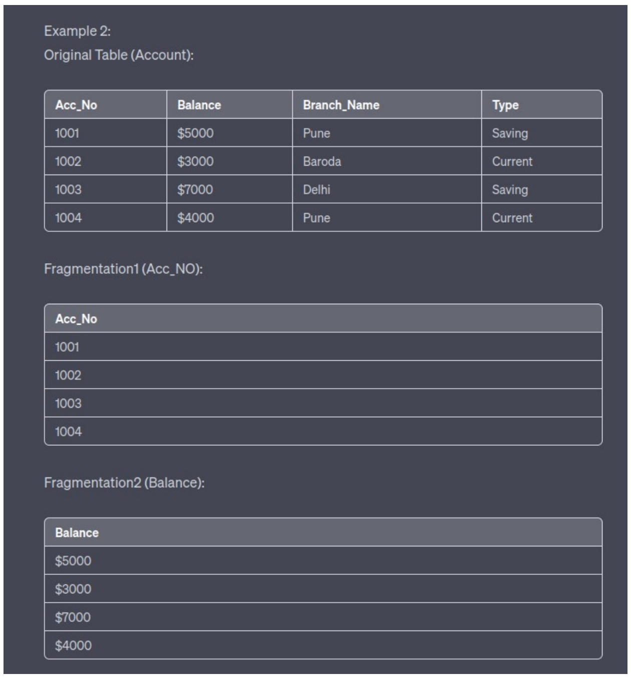
This query retrieves all the columns from the table Acc_NO. It represents a subset or fragment of the original table where only the Acc_No column is considered.

Fragmentation2: SELECT * FROM Balance

This query retrieves all the columns from the table Balance. It represents another subset or fragment of the original table where only the Balance column is considered.

In vertical fragmentation, the original table is divided into subsets or fragments based on columns. Each fragment contains a specific subset of columns from the original table. This approach helps in organizing and managing data based on different attributes or columns, allowing for more efficient storage and retrieval of specific information.

To summarize, vertical fragmentation involves dividing a table into subsets based on columns. Each subset represents a specific subset of columns from the original table. This approach enables efficient access and management of data based on specific attributes or columns, improving performance in a distributed database system.


Example 2:
Original Table (Account):

Acc_No	Balance	Branch_Name	Type
1001	$5000	Pune	Saving
1002	$3000	Baroda	Current
1003	$7000	Delhi	Saving
1004	$4000	Pune	Current

Fragmentation1 (Acc_NO):

Acc_No
1001
1002
1003
1004

Fragmentation2 (Balance):

Balance
$5000
$3000
$7000
$4000

3. **Hybrid-Fragmentation:-**Hybrid fragmentation is achieved by combining both horizontal and vertical partitioning techniques. In this approach, a table is divided horizontally into subsets of rows, and within each subset, further division is done vertically into subsets of columns.

For instance, in an employee table, hybrid fragmentation could involve horizontally dividing the table based on departments, creating subsets of employees for each department. Within each department subset, further vertical fragmentation could be applied to group columns based on specific attributes.

The hybrid fragmentation approach offers flexibility and optimization opportunities by allowing the data to be organized and distributed based on both rows and columns. It provides finer control over data distribution and can be useful in scenarios where different subsets of data need to be accessed or processed efficiently.

To summarize, horizontal fragmentation divides a table into subsets of rows, vertical fragmentation divides a table into subsets of columns, and hybrid fragmentation combines both approaches, dividing a table horizontally and further dividing each subset vertically. These fragmentation techniques help in organizing and distributing data in a distributed database system.

Example :- In the example given, let's consider a table containing employee information. We'll focus on two fragments of hybrid fragmentation:

Fragmentation1: SELECT * FROM Emp_Name WHERE Emp_Age < 40

This fragment selects all the rows from the original table where the employee's age is less than 40. It represents a subset of rows based on a specific condition.

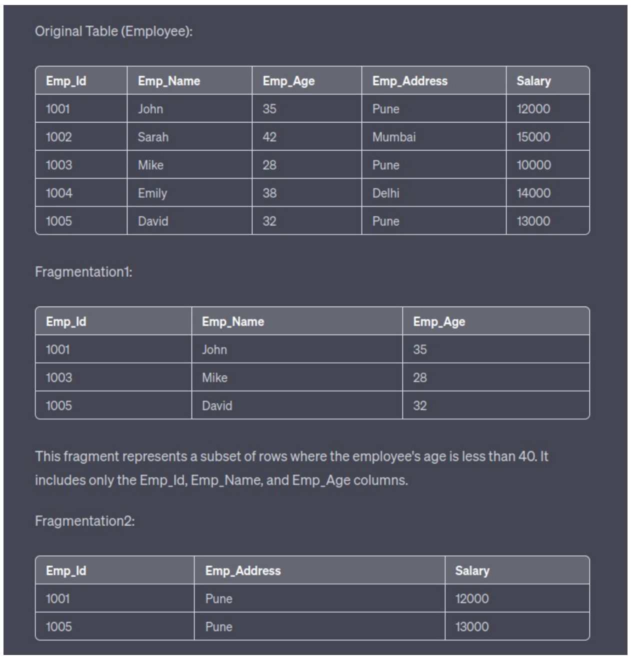
Fragmentation2: SELECT * FROM Emp_Id WHERE Emp_Address = 'Pune' AND Salary < 14000

This fragment selects all the rows from the original table where the employee's address is 'Pune' and the salary is less than 14000. It represents another subset of rows based on multiple conditions.

The purpose of hybrid fragmentation is to provide a more tailored and efficient way of organizing data based on both row subsets and column subsets simultaneously.

To reconstruct the original table from these hybrid fragments, we can use set operations like UNION and FULL OUTER JOIN. These operations help combine the fragmented subsets and merge them back into a single table.

In summary, hybrid fragmentation involves dividing a table into subsets of rows and subsets of columns simultaneously. It allows for a more flexible and customized approach to data partitioning in a distributed database system. Reconstructing the original table from hybrid fragments can be done using set operations like UNION and FULL OUTER JOIN to merge the subsets back together.



Original Table (Employee):

Emp_Id	Emp_Name	Emp_Age	Emp_Address	Salary
1001	John	35	Pune	12000
1002	Sarah	42	Mumbai	15000
1003	Mike	28	Pune	10000
1004	Emily	38	Delhi	14000
1005	David	32	Pune	13000

Fragmentation1:

Emp_Id	Emp_Name	Emp_Age
1001	John	35
1003	Mike	28
1005	David	32

This fragment represents a subset of rows where the employee's age is less than 40. It includes only the Emp_Id, Emp_Name, and Emp_Age columns.

Fragmentation2:

Emp_Id	Emp_Address	Salary
1001	Pune	12000
1005	Pune	13000

DATA REPLICATION AND ALLOCATION

In distributed database systems, data replication plays a crucial role in improving data availability and performance. Replication involves creating copies of data across multiple sites within the distributed system. This response explores the concept of data replication and allocation, ranging from full replication to no replication, and the implications they have on system reliability, performance, and transaction processing. The decision of replication and allocation strategies depends on factors such as data availability requirements, transaction patterns, and performance goals.

Data Replication and Allocation:

Data replication is the process of creating copies of data within a distributed database system. The most extreme form of replication is fully replicating the entire database at every site in the distributed system. This approach offers remarkable data availability, allowing the system to continue functioning even if some sites fail. It also improves performance for global queries since the results can be obtained locally from any site.

However, full replication can significantly impact update operations. Every logical update made to the database must be applied to each copy, resulting in increased overhead. This becomes especially challenging when there are numerous copies of the database. Concurrency control and recovery techniques become more complex and resource-intensive.

On the opposite end of the spectrum, there is no replication. Each fragment of data is stored at a single site, ensuring data uniqueness. This approach, known as non-redundant allocation, eliminates the need for synchronization among copies. However, it reduces data availability and fault tolerance.

Between these extremes, there are various degrees of partial replication. In a partially replicated database, some fragments are replicated, while others are not. The number of copies for each fragment can vary from one to the total number of sites in the distributed system. Partial replication is often employed in scenarios involving mobile workers who carry partially replicated databases on devices like laptops or PDAs, which are periodically synchronized with the central server.

The process of assigning fragments or copies of fragments to specific sites in the distributed system is referred to as data distribution or data allocation. The allocation strategy depends on factors such as performance goals, availability requirements, and transaction patterns. For example, a fully replicated database is suitable when high availability is essential and most transactions are retrieval-based. However, if specific transactions primarily access certain parts of the database at a particular site, it may be beneficial to allocate the corresponding fragments exclusively to that site. Data accessed by multiple sites can be replicated to enhance performance and reduce network latency.

Determining the optimal replication and allocation strategy for a distributed database system is a complex optimization problem. It requires considering trade-offs between data availability, system performance, resource utilization, and the specific requirements of the applications and users.

Let's define the three replication schemes (full replication, no replication, and partial replication) and the concept of query processing and optimization in simpler terms:

1. **Full Replication:** In the full replication scheme, the entire database is replicated and made available at almost every location or user in the communication network. This means that every user can access a complete copy of the database.

Advantages of full replication:

- High availability of data: Since the database is replicated everywhere, it is readily accessible, ensuring data availability.
- Faster execution of queries: Users can retrieve data quickly as they can access the nearest replica, reducing communication delays.

Disadvantages of full replication:

- Concurrency control challenges: Managing concurrent updates becomes complex when multiple replicas are involved, as conflicts need to be resolved.
- Slower update operations: Updating all replicas to maintain consistency requires more time and resources.

2. No Replication: In the no replication scheme, each fragment of the database is stored exactly at one location, and there are no additional copies or replicas.

Advantages of no replication:

- Minimized concurrency: With only one copy of each fragment, concurrency conflicts are reduced.
- Easy recovery of data: If a failure occurs, data recovery is simpler as there is only one copy to restore.

Disadvantages of no replication:

- Poor availability of data: If the server holding the fragment fails, access to that specific fragment is lost until the server is restored.

- Slower query execution: As multiple clients may need to access the same server, it can lead to increased contention and slower query processing.

3. Partial Replication: Partial replication involves replicating only selected fragments of the database, rather than replicating the entire database.

Advantages of partial replication:

- Replication based on importance: Replicas are created for specific fragments based on the importance of the data they hold.
- Flexible optimization: Allows for optimization by choosing which fragments to replicate, considering factors such as access frequency or criticality.

TRANSPARENCY :- Transparency in the context of distributed databases refers to hiding the complexities of the system's implementation from end users. A highly transparent system provides flexibility to users and application developers by minimizing their need to understand the underlying details. In a traditional centralized database, transparency relates to logical and physical data independence for application developers.

However, in the case of a distributed database scenario, where data and software are distributed across multiple sites connected by a computer network, additional types of transparency come into play. Let's consider an example of a company database that we have been discussing. The EMPLOYEE, PROJECT, and WORKS_ON tables can be divided into smaller parts and stored across different sites. This fragmentation allows for better management and potential replication of the fragments.

Transparency ensures that end users and application developers are unaware of the fragmentation and replication details. They can interact with the database as if it were a single unified entity, without needing to consider the distribution of data and the presence of replicated fragments. This transparency simplifies the development and usage of applications, making it easier to work with the distributed database system.

i