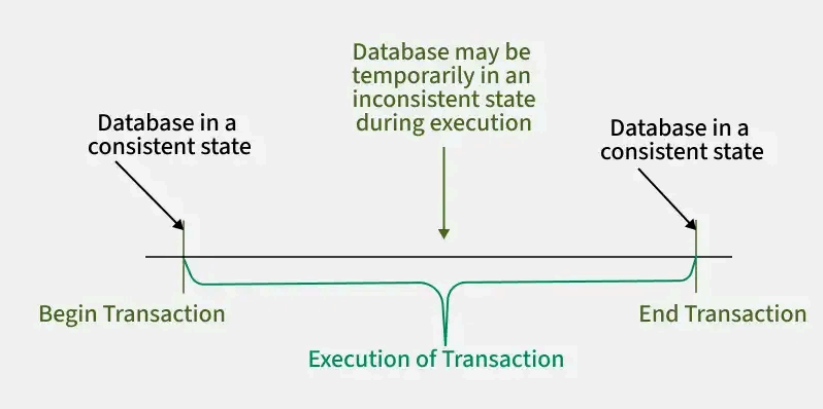
Transaction in DBMS

Last Updated : 12 Mar, 2025

In a Database Management System (DBMS), a transaction is a sequence of operations performed as a single logical unit of work. These operations may involve reading, writing, updating, or deleting data in the database. A transaction is considered complete only if all its operations are successfully executed, Otherwise the transaction must be **rolled back**, ensuring the database remains in a consistent state.

What does a Transaction mean in DBMS?

A transaction refers to a sequence of one or more operations (such as read, write, update, or delete) performed on the database as a single logical unit of work. A transaction ensures that either all the operations are successfully executed (committed) or none of them take effect (rolled back). Transactions are designed to maintain the integrity, consistency and reliability of the database, even in the case of system failures or concurrent access.



Transaction

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

Example:

Let's consider an **online banking application**:

- **Transaction**: When a user performs a **money transfer**, several operations occur, such as:
    - **Reading** the account balance of the sender.
    - **Writing** the deducted amount from the sender's account.
    - **Writing** the added amount to the recipient's account.

In a **transaction**, all these steps should either complete successfully or, if any error occurs, the database should **rollback** to its previous state, ensuring no partial data is written to the system.

Facts about Database Transactions

- A transaction is a program unit whose execution may or may not change the contents of a database.
- The transaction is executed as a single unit.
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- A successful transaction can change the database from one CONSISTENT STATE to another.
- DBMS transactions must be atomic, consistent, isolated and durable.
- If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

Operations of Transaction

A user can make different types of requests to access and modify the contents of a database. So, we have different types of operations relating to a transaction. They are discussed as follows:

1) Read(X)

A read operation is used to read the value of a particular database element **X** and stores it in a temporary buffer in the main memory for further actions such as displaying that value.

Example:
For a **banking system**, when a user checks their balance, a **Read** operation is performed on their **account balance**:

```
SELECT balance FROM accounts WHERE account_id = 'A123';
```

This updates the balance of the user's account after withdrawal

2) Write(X)

A write operation is used to write the value to the database from the buffer in the main memory. For a write operation to be performed, first a read operation is performed to bring its value in buffer, and then some changes are made to it, e.g. some set of arithmetic operations are performed on it according to the user's request, then to store the modified value back in the database, a write operation is performed.

Example:
For the **banking system**, if a user withdraws money, a **Write** operation is performed after the balance is updated:

```
UPDATE accounts SET balance = balance - 100 WHERE
account_id = 'A123';
```

This updates the balance of the user’s account after withdrawal.

3) Commit

This operation in transactions is used to maintain integrity in the database. Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed. This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction.

Example:

After a successful money transfer in a banking system, a **Commit** operation finalizes the transaction:

```
COMMIT;
```

Once the transaction is committed, the changes to the database are permanent, and the transaction is considered **successful**.

4) Rollback

If an error occurs, the **Rollback** operation undoes all the changes made by the transaction, reverting the database to its last consistent state. In simple words, it can be said that a rollback operation does undo the operations of transactions that were performed before its interruption to achieve a safe state of the database and avoid any kind of ambiguity or inconsistency.

Example:
Suppose during the money transfer process, the system encounters an issue, like insufficient funds in the sender's account. In that case, the transaction is rolled back:

```
ROLLBACK;
```

This will undo all the operations performed so far and ensure that the database remains consistent.

Transaction Schedules

When multiple transaction requests are made at the same time, we need to decide their order of execution. Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions.

There are broadly two types of transaction schedules discussed as follows:

i) Serial Schedule

In this kind of schedule, when multiple transactions are to be executed, they are executed serially, i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed. This ensures consistency in the database as transactions do not execute simultaneously.

But, it increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time. To improve the throughput of the system, another kind of schedule are used which has some more strict rules which help the database to remain consistent even when transactions execute simultaneously.

Example:
In a serial schedule, the first transaction completes before the second transaction starts:

1. Transaction 1: Read balance → Update balance → Commit
2. Transaction 2: Read balance → Update balance → Commit

ii) Non-Serial Schedule

To reduce the waiting time of transactions in the waiting queue and improve the system efficiency, we use non-serial schedules which allow multiple transactions to start before a transaction is completely executed. This may sometimes result in inconsistency and errors in database operation.

So, these errors are handled with specific algorithms to maintain the consistency of the database and improve **CPU** throughput as well. Non-serial schedules are also sometimes referred to as parallel schedules, as transactions execute in parallel in these kinds of schedules.

Example:
Transaction 1 and Transaction 2 can be executed in parallel:

- Transaction 1: Read balance → Update balance
- Transaction 2: Read balance → Update balance

Without proper isolation mechanisms, this may cause inconsistencies.

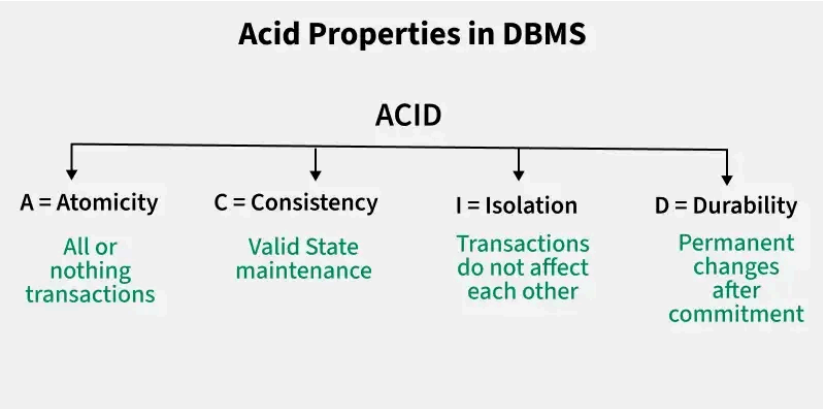
Serializable

Serializability in DBMS is the property of a non-serial schedule that determines whether it would maintain the database consistency or not. The non-serial schedule which ensures that the database would be consistent after the transactions are executed in the order determined by that schedule is said to be Serializable Schedules.

The serial schedules always maintain database consistency as a transaction starts only when the execution of the other transaction has been completed under it. Thus, serial schedules are always serializable. A transaction is a series of operations, so various states occur in its completion journey.

Properties of Transaction

- As transactions deal with accessing and modifying the contents of the database, they must have some basic properties which help maintain the consistency and integrity of the database before and after the transaction. Transactions follow 4 properties, namely, Atomicity, Consistency, Isolation, and Durability.
- Generally, these are referred to as **ACID** properties of transactions in DBMS. ACID is the acronym used for transaction properties. A brief description of each property of the transaction is as follows.



ACID Properties

1) Atomicity

This property ensures that either all operations of a transaction are executed or it is aborted. In any case, a transaction can never be completed partially. Each transaction is treated as a single unit (like an atom).

Atomicity is achieved through commit and rollback operations, i.e. changes are made to the database only if all operations related to a transaction are completed, and if it gets interrupted, any changes made are rolled back using rollback operation to bring the database to its last saved state.

Example:
If a user is transferring money from one account to another, both the **debit** and **credit** operations must be successful, or none should happen. If any step fails, the transaction will be rolled back entirely.

2) Consistency

- This property of a transaction keeps the database consistent before and after a transaction is completed.
- Execution of any transaction must ensure that after its execution, the database is either in its prior stable state or a new stable state.
- In other words, the result of a transaction should be the transformation of a database from one consistent state to another consistent state.
- Consistency, here means, that the changes made in the database are a result of logical operations only which the user desired to perform and there is not any ambiguity.

Example:
If an account balance before the transaction is 1000, after a successful transaction of 200, the new balance should be 800, ensuring the database stays in a consistent state.

3) Isolation

This property states that two transactions must not interfere with each other, i.e. if some data is used by a transaction for its execution, then any other transaction can not concurrently access that data until the first transaction has completed. It ensures that the integrity of the database is maintained and we don't get any ambiguous values. Thus, any two transactions are isolated from each other.

Example:

- **Transaction 1**: Withdraw $100 from account A.
- **Transaction 2**: Withdraw $200 from account A.

Both transactions should execute as if they are isolated from each other, and their changes should not conflict.

iv) Durability

- This property ensures that the changes made to the database after a transaction is completely executed, are durable.
- It indicates that permanent changes are made by the successful execution of a transaction.
- In the event of any system failures or crashes, the consistent state achieved after the completion of a transaction remains intact. The recovery subsystem of DBMS is responsible for enforcing this property.

Example:

After transferring money between two accounts, if the system crashes, the changes made by the transaction should still be saved when the system restarts.

Conclusion

Transactions in DBMS are pivotal in maintaining the integrity and consistency of the database through a series of well-defined operations and states. From the initial execution of operations to handling errors and ensuring consistency, transactions must adhere to the ACID properties—Atomicity, Consistency, Isolation, and Durability. These properties guarantee that transactions are processed reliably and that

Transaction States in DBMS

Last Updated : 14 Jan, 2025

Transaction in DBMS is a set of logically related operations executed as a single unit. These logic are followed to perform modification on data while maintaining integrity and consistency. Transactions are performed in a way that concurrent actions from different users don't malfunction the database. Transfer of money from one account to another in a bank management system is the best example of Transaction.

A transaction goes through several states during its lifetime. These states indicate the current status of the transaction and guide how it will proceed. They determine whether the transaction will be successfully completed (committed) or stopped (aborted). These states also use a transaction log to keep track of the process.

What is a Transaction State?

A transaction is a set of operations or tasks performed to complete a logical process, which may or may not change the data in a database. To handle different situations, like system failures, a transaction is divided into different states.

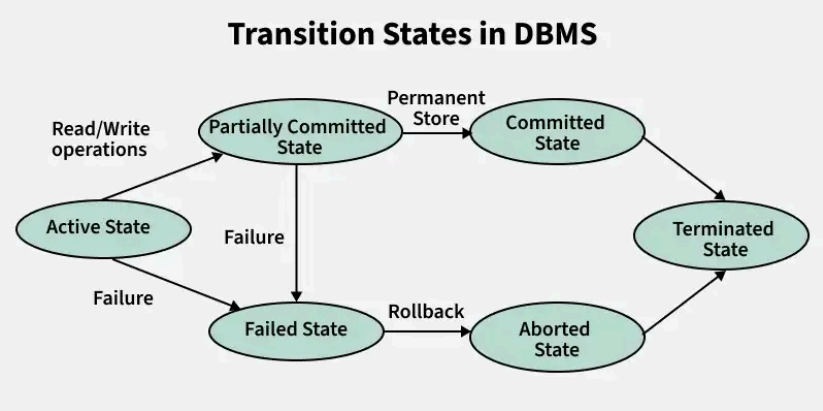
A transaction state refers to the current phase or condition of a transaction during its execution in a database. It represents the progress of the transaction and determines whether it will successfully complete (commit) or fail (abort).

A transaction involves two main operations:

1. **Read Operation**: Reads data from the database, stores it temporarily in memory (buffer), and uses it as needed.
2. **Write Operation**: Updates the database with the changed data using the buffer.

From the start of executing instructions to the end, these operations are treated as a single transaction. This ensures the database remains consistent and reliable throughout the process.

Different Types of Transaction States in DBMS



Transaction States

These are different types of Transaction States :

1. Active State – This is the first stage of a transaction, when the transaction's instructions are being executed.

- It is the first stage of any transaction when it has begun to execute. The execution of the transaction takes place in this state.
- Operations such as insertion, deletion, or updation are performed during this state.
- During this state, the data records are under manipulation and they are not saved to the database, rather they remain somewhere in a buffer in the main memory.

2. Partially Committed –

- The transaction has finished its final operation, but the changes are still not saved to the database.
- After completing all read and write operations, the modifications are initially stored in main memory or a local buffer. If the changes are made permanent on the DataBase then the state will change to "committed state" and in case of failure it will go to the "failed state".

3. Failed State –If any of the transaction-related operations cause an error during the active or partially committed state, further execution of the transaction is stopped and it is brought into a failed state. Here, the database recovery system makes sure that the database is in a consistent state.

5. Aborted State- If a transaction reaches the failed state due to a failed check, the database recovery system will attempt to restore it to a consistent state. If recovery is not possible, the transaction is either rolled back or cancelled to ensure the database remains consistent.

In the aborted state, the DBMS recovery system performs one of two actions:

- **Kill the transaction**: The system terminates the transaction to prevent it from affecting other operations.
- **Restart the transaction**: After making necessary adjustments, the system reverts the transaction to an active state and attempts to continue its execution.

6. Commuted- This state of transaction is achieved when all the transaction-related operations have been executed successfully along with the Commit operation, i.e. data is saved into the database after the required manipulations in this state. This marks the successful completion of a transaction.

7. Terminated State – If there isn't any roll-back or the transaction comes from the "committed state", then the system is consistent and ready for new transaction and the old transaction is terminated.

Example of Transaction States

Imagine a bank transaction where a user wants to transfer $500 from **Account A** to **Account B**.

Transaction States:

1. **Active State**:

   The transaction begins. It reads the balance of Account A and

checks if it has enough funds.
   - Example: Read balance of Account A = $1000.
2. **Partially Committed State**:
   The transaction performs all its operations but hasn't yet saved (committed) the changes to the database.
   - Example: Deduct $500 from Account A's balance ($1000 – $500 = $500) and temporarily update Account B's balance (add $500).

3. **Committed State**:
   The transaction successfully completes, and the changes are saved permanently in the database.
   - Example: Account A's new balance = $500; Account B's new balance = $1500. Changes are written to the database.

4. **Failed State**:
   If something goes wrong during the transaction (e.g., power failure, system crash), the transaction moves to this state.
   - Example: System crashes after deducting $500 from Account A but before adding it to Account B.

5. **Aborted State**:
   The failed transaction is rolled back, and the database is restored to its original state.
   - Example: Account A's balance is restored to $1000, and no changes are made to Account B.

These states ensure that either the transaction completes successfully (committed) or the database is restored to its original state (aborted), maintaining consistency and preventing errors.

Conclusion

In conclusion, transaction is the collection of low level tasks in DBMS. While a transaction may seem to be a two step process to the user, it involved many sub-process often referred as transaction states in data terminology. There are 6 states, a transaction may undergo. These state are responsible for successful completion of transaction. The

transaction execution ensures the ACID property of DBMS. One needs to understand the logic of transaction and its working for mastering DBMS.

Next Article

Difference between CD-R and CD-RW

Similar Reads

Transaction in DBMS

In a Database Management System (DBMS), a transaction is a sequence of operations performed as a single logical unit of work. These...

15+ min read

Concurrency problems in DBMS Transactions

Concurrency control is an essential aspect of database management systems (DBMS) that ensures transactions can execute concurrently...

15+ min read

Transaction Control in DBMS

The transaction is a single logical unit that accesses and modifies the contents of the database. Transactions access data using read and writ...

15+ min read

Transaction Isolation Levels in DBMS

The levels of transaction isolation in DBMS determine how the concurrently running transactions behave and, therefore, ensure data...

15+ min read

Serializability in DBMS

Concurrency Control in DBMS

Last Updated : 29 Jan, 2025

In a database management system (DBMS), allowing transactions to run concurrently has significant advantages, such as better system resource utilization and higher throughput. However, it is crucial that these transactions do not conflict with each other. The ultimate goal is to ensure that the database remains consistent and accurate. For instance, if two users try to book the last available seat on a flight at the same time, the system must ensure that only one booking succeeds.Concurrency control is a critical mechanism in DBMS that ensures the consistency and integrity of data when multiple operations are performed at the same time.

- Concurrency control is a concept in Database Management Systems (DBMS) that ensures multiple transactions can simultaneously access or modify data without causing errors or inconsistencies. It provides mechanisms to handle the concurrent execution in a way that maintains ACID properties.
- By implementing concurrency control, a DBMS allows transactions to execute concurrently while avoiding issues such as deadlocks, race conditions, and conflicts between operations.
- The main goal of concurrency control is to ensure that simultaneous transactions do not lead to data conflicts or violate the consistency of the database. The concept of serializability is often used to achieve this goal.

In this article, we will explore the various concurrency control techniques in DBMS, understand their importance, and learn how they enable reliable and efficient database operations.

Concurrent Execution and Related Challenges in DBMS

In a multi-user system, several users can access and work on the same database at the same time. This is known as concurrent execution, where the database is used simultaneously by different users for various operations. For instance, one user might be updating data while another is retrieving it.

When multiple transactions are performed on the database simultaneously, it is important that these operations are executed in an interleaved manner. This means that the actions of one user should not interfere with or affect the actions of another. This helps in maintaining the consistency of the database. However, managing such simultaneous operations can be challenging, and certain problems may arise if not handled properly. These challenges need to be addressed to ensure smooth and error-free concurrent execution.

Concurrent Execution can lead to various challenges:

- **Dirty Reads**: One transaction reads uncommitted data from another transaction, leading to potential inconsistencies if the changes are later rolled back.
- **Lost Updates**: When two or more transactions update the same data simultaneously, one update may overwrite the other, causing data loss.
- **Inconsistent Reads**: A transaction may read the same data multiple times during its execution, and the data might change between reads due to another transaction, leading to inconsistency.

To read more about Concurrency Problems in DBMS Transactions Refer, Here.

Why is Concurrency Control Needed?

Consider the following example:

Scenario	Without Concurrency Control	With Concurrency Control
Initial Balance	$1000	$1000
Transaction A (User 1)	Withdraws $200: Reads balance as $1000, calculates new balance $800.	Withdraws $200: Reads balance as $1000, calculates new balance $800.
Transaction B (User 2)	Withdraws $300: Reads balance as $1000 (before A updates), calculates $ 700	Waits for Transaction A to finish. Once A commits, reads updated balance $800, calculates new balance $500.
Final Updates	A writes $800. Then, B reads overwrites with $700 (lost update issue).	A writes $800. Then, B reads updated balance and writes $500 (consistent result).
Final Balance	$700 (incorrect, inconsistent).	$500 (correct, consistent).

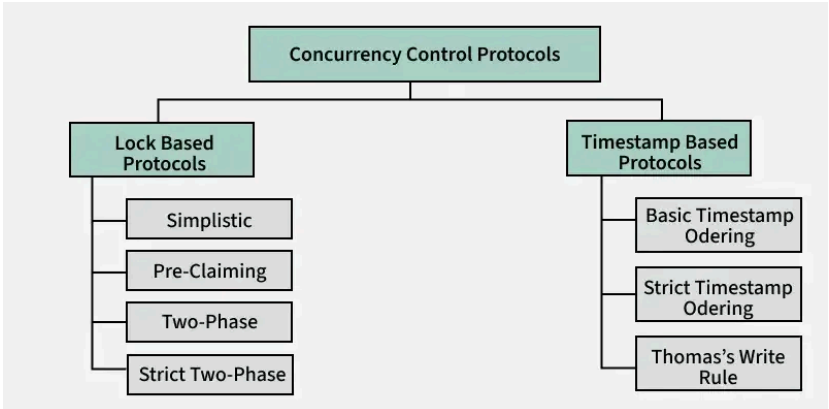
- **Without Concurrency Control**: Transactions interfere with each other, causing issues like lost updates, dirty reads or inconsistent results.

- **With Concurrency Control**: Transactions are properly managed (e.g., using locks or timestamps) to ensure they execute in a consistent, isolated manner, preserving data accuracy.

Concurrency control is critical to maintaining the accuracy and reliability of databases in multi-user environments. By preventing conflicts and inconsistencies during concurrent transactions, it ensures the database remains consistent and correct, even under high levels of simultaneous activity.

Concurrency Control Protocols

Concurrency control protocols are the set of rules which are maintained in order to solve the concurrency control problems in the database. It ensures that the concurrent transactions can execute properly while maintaining the database consistency. The concurrent execution of a transaction is provided with atomicity, consistency, isolation, durability, and serializability via the concurrency control protocols.



- Locked based concurrency control protocol

- Timestamp based concurrency control protocol

Cascadeless and Recoverable Schedules in Concurrency Control

1. Recoverable Schedules

- A **recoverable schedule** ensures that a transaction commits only if all the transactions it depends on have committed. This avoids situations where a committed transaction depends on an uncommitted transaction that later fails, leading to inconsistencies.
    - Concurrency control ensures recoverable schedules by keeping track of which transactions depend on others. It makes sure a transaction can only commit if all the transactions it relies on have already committed successfully. This prevents issues where a committed transaction depends on one that later fails.
    - Techniques like strict two-phase locking (2PL) enforce recoverability by delaying the commit of dependent transactions until the parent transactions have safely committed.

2. Cascadeless Schedules

- A **cascadeless schedule** avoids cascading rollbacks, which occur when the failure of one transaction causes multiple dependent transactions to fail.
    - Concurrency control techniques such as strict 2PL or timestamp ordering ensure cascadeless schedules by ensuring dependent transactions only access committed data.
    - By delaying read or write operations until the transaction they depend on has committed, cascading rollbacks are avoided.

To read more about different types of schedules based on Recoverability Refer, Here.

Advantages of Concurrency

In general, concurrency means that more than one transaction can work on a system. The advantages of a concurrent system are:

- **Waiting Time:** It means if a process is in a ready state but still the process does not get the system to get execute is called waiting time. So, concurrency leads to less waiting time.

- **Response Time:** The time wasted in getting the response from the CPU for the first time, is called response time. So, concurrency leads to less Response Time.

- **Resource Utilization:** The amount of Resource utilization in a particular system is called Resource Utilization. Multiple transactions can run parallel in a system. So, concurrency leads to more Resource Utilization.

- **Efficiency:** The amount of output produced in comparison to given input is called efficiency. So, Concurrency leads to more Efficiency.

Disadvantages of Concurrency

- **Overhead:** Implementing concurrency control requires additional overhead, such as acquiring and releasing locks on database objects. This overhead can lead to slower performance and increased resource consumption, particularly in systems with high levels of concurrency.

- **Deadlocks:** Deadlocks can occur when two or more transactions are waiting for each other to release resources, causing a circular dependency that can prevent any of the transactions from completing. Deadlocks can be difficult to detect and resolve, and can result in reduced throughput and increased latency.

- **Reduced concurrency:** Concurrency control can limit the number of users or applications that can access the database simultaneously.

This can lead to reduced concurrency and slower performance in systems with high levels of concurrency.

- **Complexity:** Implementing concurrency control can be complex, particularly in distributed systems or in systems with complex transactional logic. This complexity can lead to increased development and maintenance costs.

- **Inconsistency:** In some cases, concurrency control can lead to inconsistencies in the database. For example, a transaction that is rolled back may leave the database in an inconsistent state, or a long-running transaction may cause other transactions to wait for extended periods, leading to data staleness and reduced accuracy.

Conclusion

Concurrency control ensures transaction atomicity, isolation, consistency and serializability. Concurrency control issues occur when many transactions execute randomly. A dirty read happens when a transaction reads data changed by an uncommitted transaction. When two transactions update data simultaneously, the Lost Update issue occurs. Lock-based protocol prevents incorrect read/write activities. Timestamp-based protocols organize transactions by timestamp.

Next Article

ACID Properties in DBMS

Similar Reads

Lock Based Concurrency Control Protocol in DBMS

In a Database Management System (DBMS), lock-based concurrency control (BCC) is a method used to manage how multiple transactions...

15+ min read

Concurrency problems in DBMS Transactions

Last Updated : 28 Dec, 2024

Concurrency control is an essential aspect of database management systems (DBMS) that ensures transactions can execute concurrently without interfering with each other. However, concurrency control can be challenging to implement, and without it, several problems can arise, affecting the consistency of the database. In this article, we will discuss some of the concurrency problems that can occur in DBMS transactions and explore solutions to prevent them.

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in a database environment.

The five concurrency problems that can occur in the database are:

- Temporary Update Problem
- Incorrect Summary Problem
- Lost Update Problem
- Unrepeatable Read Problem
- Phantom Read Problem

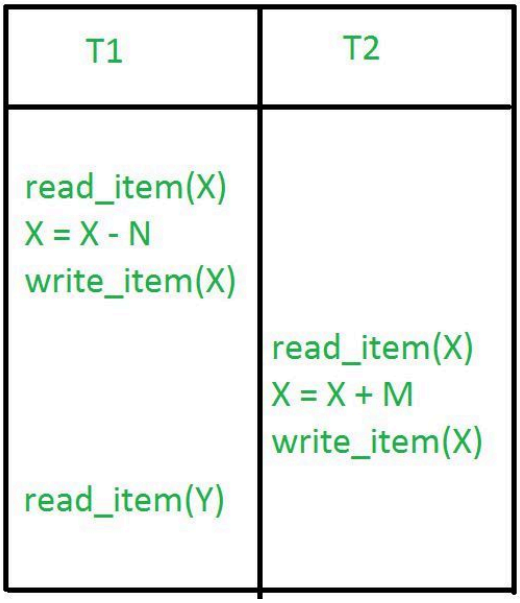
Concurrency control ensures the consistency and integrity of data in databases when multiple transactions are executed simultaneously. Understanding issues like lost updates, dirty reads, and non-repeatable reads is crucial when studying DBMS.

These are explained as following below.

Temporary Update Problem:

Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

Example:



In the above example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

Incorrect Summary Problem:

Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:

T1	T2
	sum = 0 read_item(A) sum = sum + A
read_item(X) X = X - N write_item(X)	
	read_item(X) sum = sum + X read_item(Y) sum = sum + Y
read_item(Y) Y = Y + N write_item(Y)	

In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

Lost Update Problem:

In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

T1	T2
read_item(X) X = X + N	
	X = X + 10 write_item(X)

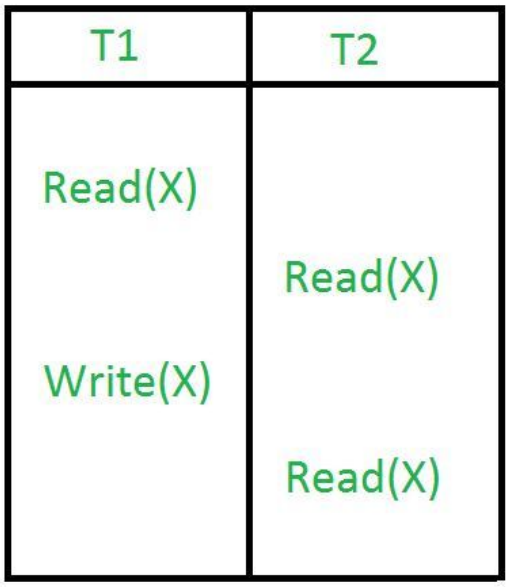
In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X *(not shown in the image above)*. Therefore, the update done by transaction 2 will be

lost. Basically, the write commit done by the **last transaction** will overwrite all previous write commits.

Unrepeatable Read Problem:

The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

Example:

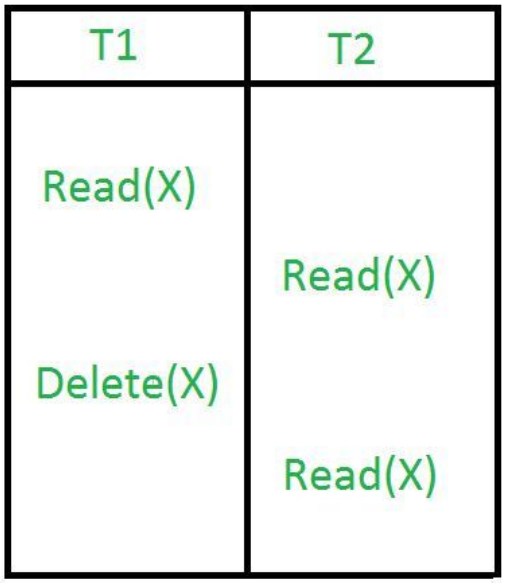


In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

Phantom Read Problem:

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

Example:



In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Solution :

To prevent concurrency problems in DBMS transactions, several concurrency control techniques can be used, including locking, timestamp ordering, and optimistic concurrency control.

Locking involves acquiring locks on the data items used by transactions, preventing other transactions from accessing the same data until the lock is released. There are different types of locks, such as shared and exclusive locks, and they can be used to prevent Dirty Read and Non-Repeatable Read.

Timestamp ordering assigns a unique timestamp to each transaction and ensures that transactions execute in timestamp order. Timestamp ordering can prevent Non-Repeatable Read and Phantom Read.

Optimistic concurrency control assumes that conflicts between transactions are rare and allows transactions to proceed without acquiring locks initially. If a conflict is detected, the transaction is rolled back, and the conflict is resolved. Optimistic concurrency control can prevent Dirty Read, Non-Repeatable Read, and Phantom Read.

In conclusion, concurrency control is crucial in DBMS transactions to ensure data consistency and prevent concurrency problems such as Dirty Read, Non-Repeatable Read, and Phantom Read. By using techniques like locking, timestamp ordering, and optimistic concurrency control, developers can build robust database systems that support concurrent access while maintaining data consistency.

Next Article

Neo4j Create Index

Similar Reads

Concurrency Control in DBMS

In a database management system (DBMS), allowing transactions to run concurrently has significant advantages, such as better system resourc...

15+ min read

Transaction Isolation Levels in DBMS

The levels of transaction isolation in DBMS determine how the concurrently running transactions behave and, therefore, ensure data...

15+ min read

Lock Based Concurrency Control Protocol in DBMS

In a Database Management System (DBMS), lock-based concurrency control (BCC) is a method used to manage how multiple transactions...

15+ min read

Conflict Serializability in DBMS

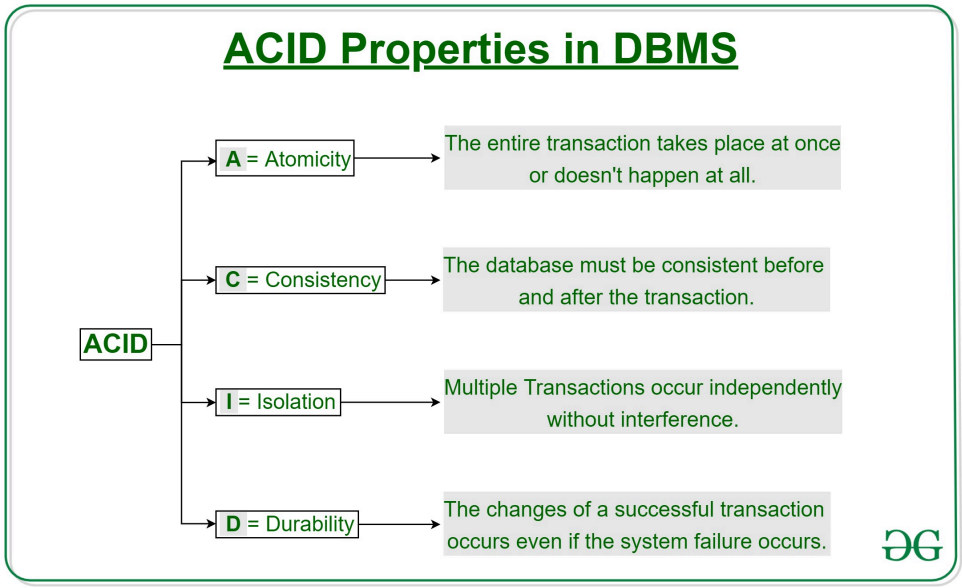
A schedule is a sequence in which operations (read, write, commit, abort) from multiple transactions are executed in a database. Serial or one by...

ACID Properties in DBMS

Last Updated : 15 Apr, 2025

In the world of **Database Management Systems** (DBMS), transactions are fundamental operations that allow us to modify and retrieve data. However, to ensure the integrity of a database, it is important that these transactions are executed in a way that maintains consistency, correctness, and reliability. This is where the **ACID properties** come into play.

ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These four key properties define how a transaction should be processed in a reliable and predictable manner, ensuring that the database remains consistent, even in cases of failures or concurrent accesses.



What Are Transactions in DBMS?

A **transaction** in DBMS refers to a sequence of operations performed as a single unit of work. These operations may involve reading or writing data to the database. To maintain data integrity, DBMS ensures that each transaction adheres to the ACID properties. Think of a transaction

like an ATM withdrawal. When we withdraw money from our account, the transaction involves several steps:

- Checking your balance.
- Deducting the money from your account.
- Adding the money to the bank's record.

For the transaction to be successful, **all steps** must be completed. If any part of this process fails (e.g., if there's a system crash), the entire transaction should fail, and no data should be altered. This ensures the database remains in a **consistent** state.

The Four ACID Properties

1. Atomicity: "All or Nothing"

Atomicity ensures that a transaction is **atomic**, it means that either the entire transaction completes fully or doesn't execute at all. There is no in-between state i.e. transactions do not occur partially. If a transaction has multiple operations, and one of them fails, the whole transaction is rolled back, leaving the database unchanged. This avoids partial updates that can lead to inconsistency.

- **Commit**: If the transaction is successful, the changes are permanently applied.
- **Abort/Rollback**: If the transaction fails, any changes made during the transaction are discarded.

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

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

If the transaction fails after completion of T1 but before completion of T2 , the database would be left in an inconsistent state. With Atomicity, if any part of the transaction fails, the entire process is rolled back to its original state, and no partial changes are made.

2. Consistency: Maintaining Valid Data States

Consistency ensures that a database remains in a valid state before and after a transaction. It guarantees that any transaction will take the database from one consistent state to another, maintaining the rules and constraints defined for the data. In simple terms, a transaction should only take the database from one **valid** state to another. If a transaction violates any database rules or constraints, it should be rejected, ensuring that only consistent data exists after the transaction.

Example: Suppose the sum of all balances in a bank system should always be constant. Before a transfer, the total balance is **$700**. After the transaction, the total balance should remain $700. If the transaction fails in the middle (like updating one account but not the other), the system should maintain its consistency by rolling back the transaction

Total before T occurs = 500 + 200 = 700 .
Total after T occurs = 400 + 300 = 700 .

X = 500 Rs	Y = 500 Rs
T	T''
Read (X) X :=X*100 Write (X) Read (Y) Y:= Y - 50 Write (Y)	Read (X) Read (Y) Z:= X + Y Write (Z)

3. Isolation: Ensuring Concurrent Transactions Don’t Interfere

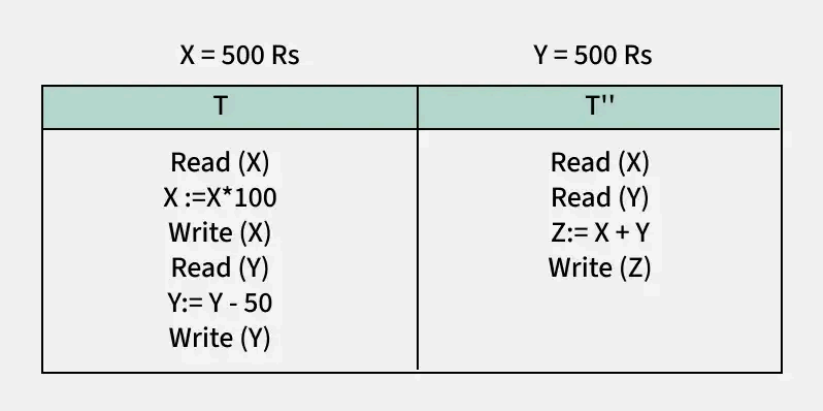
This property ensures that **multiple transactions** can occur concurrently without leading to the inconsistency of the database

state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

This property ensures that when multiple transactions run at the same time, the result will be the same as if they were run one after another in a specific order. This property prevents issues such as **dirty reads** (reading uncommitted data), **non-repeatable reads** (data changing between two reads in a transaction), and **phantom reads** (new rows appearing in a result set after the transaction starts).

Example: Let's consider two transactions:Consider two transactions **T** and **T".**

- **X = 500**, **Y = 500**



Transaction T:

- **T** wants to transfer $50 from **X** to **Y**.
- **T** reads **Y** (value: 500), deducts $50 from **X** (new **X = 450**), and adds $50 to **Y** (new **Y = 550**).

Transaction T":

- **T"** starts and reads **X** (value: 500) and **Y** (value: 500), then calculates the sum: **500 + 500 = 1000**.

But, by the time **T** finishes, **X** and **Y** have changed to **450** and **550** respectively, so the correct sum should be **450 + 550 = 1000**. **Isolation** ensures that **T"** should not see the old values of **X** and **Y** while **T** is still in progress. Both transactions should be independent, and **T"** should

only see the final state after **T** commits. This prevents inconsistent data like the incorrect sum calculated by **T”**

4. Durability: Persisting Changes

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in **non-volatile memory**. In the event of a failure, the DBMS can recover the database to the state it was in after the last committed transaction, ensuring that no data is lost.

Example: After successfully transferring money from Account A to Account B, the changes are stored on disk. Even if there is a crash immediately after the commit, the transfer details will still be intact when the system recovers, ensuring durability.

How ACID Properties Impact DBMS Design and Operation

The ACID properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

1. Data Integrity and Consistency

ACID properties safeguard the **data integrity** of a DBMS by ensuring that transactions either complete successfully or leave no trace if interrupted. They prevent **partial updates** from corrupting the data and ensure that the database transitions only between valid states.

2. Concurrency Control

ACID properties provide a solid framework for **managing concurrent transactions**. Isolation ensures that transactions do not interfere with each other, preventing data anomalies such as lost updates, temporary inconsistency, and uncommitted data.

3. Recovery and Fault Tolerance

Durability ensures that even if a system crashes, the database can recover to a consistent state. Thanks to the **Atomicity** and **Durability** properties, if a transaction fails midway, the database remains in a consistent state.

Property	Responsibility for maintaining properties
Atomicity	Transaction Manager
Consistency	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery

Advantages of ACID Properties in DBMS

1. **Data Consistency:** ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. **Data Integrity:** It maintains the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. **Concurrency Control:** ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. **Recovery:** ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS

1. **Performance Overhead**: ACID properties can introduce performance costs, especially when enforcing isolation between transactions or ensuring atomicity.
2. **Complexity**: Maintaining ACID properties in distributed systems (like microservices or cloud environments) can be complex and may require sophisticated solutions like distributed locking or transaction coordination.
3. **Scalability Issues**: ACID properties can pose scalability challenges, particularly in systems with high transaction volumes, where traditional relational databases may struggle under load.

ACID in the Real World: Where Is It Used?

In modern applications, ensuring the **reliability and consistency** of data is crucial. ACID properties are fundamental in sectors like:

- **Banking**: Transactions involving money transfers, deposits, or withdrawals must maintain strict consistency and durability to prevent errors and fraud.
- **E-commerce**: Ensuring that inventory counts, orders, and customer details are handled correctly and consistently, even during high traffic, requires ACID compliance.
- **Healthcare**: Patient records, test results, and prescriptions must adhere to strict consistency, integrity, and security standards.

Conclusion

The **ACID properties** in DBMS provide the backbone for maintaining data consistency, integrity, and reliability in the face of transaction failures, concurrent operations, and system crashes. In today's digital world, **ACID properties** ensure that database systems can handle complex transactions securely, reliably, and efficiently, which is why they remain a cornerstone of data management systems used in a variety of critical applications.

By understanding how ACID works, developers and **system administrators** can design better systems and make informed

Serializability in DBMS

Last Updated : 06 Oct, 2023

In this article, we are going to explain the serializability concept and how this concept affects the DBMS deeply, we also understand the concept of serializability with some examples, and we will finally conclude this topic with an example of the importance of serializability. The DBMS form is the foundation of the most modern applications, and when we design the form properly, it provides high-performance and relative storage solutions to our application.

What is a serializable schedule, and what is it used for?

If a non-serial schedule can be transformed into its corresponding serial schedule, it is said to be serializable. Simply said, a non-serial schedule is referred to as a serializable schedule if it yields the same results as a serial timetable.

Non-serial Schedule

A schedule where the transactions are overlapping or switching places. As they are used to carry out actual database operations, multiple transactions are running at once. It's possible that these transactions are focusing on the same data set. Therefore, it is crucial that non-serial schedules can be serialized in order for our database to be consistent both before and after the transactions are executed.

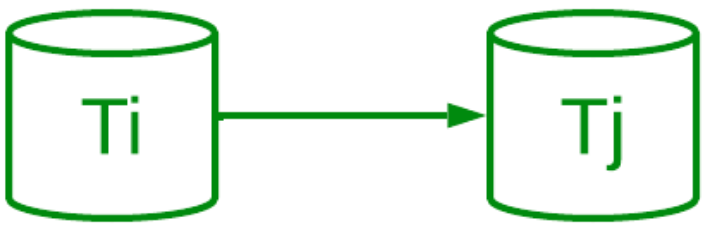
Example:

Transaction-1	Transaction-2
R(a)	
W(a)	
	R(b)
	W(b)
R(b)	
	R(a)
W(b)	
	W(a)

We can observe that Transaction-2 begins its execution before Transaction-1 is finished, and they are both working on the same data, i.e., "a" and "b", interchangeably. Where "R"-Read, "W"-Write

Serializability testing

We can utilize the Serialization Graph or Precedence Graph to examine a schedule's serializability. A schedule's full transactions are organized into a Directed Graph, what a serialization graph is.

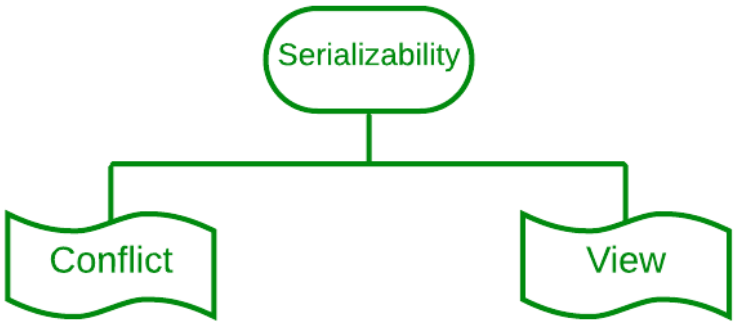


Precedence Graph

It can be described as a Graph G(V, E) with vertices V = "V1, V2, V3,..., Vn" and directed edges E = "E1, E2, E3,..., En". One of the two operations —READ or WRITE—performed by a certain transaction is contained in the collection of edges. Where Ti -> Tj, means Transaction-Ti is either performing read or write before the transaction-Tj.

Types of Serializability

There are two ways to check whether any non-serial schedule is serializable.



Types of Serializability - Conflict & View

1. Conflict serializability

Conflict serializability refers to a subset of serializability that focuses on maintaining the consistency of a database while ensuring that identical data items are executed in an order. In a DBMS each transaction has a value and all the transactions, in the database rely on this uniqueness. This uniqueness ensures that no two operations with the conflict value can occur simultaneously.

For example lets consider an order table and a customer table as two instances. Each order is associated with one customer even though a single client may place orders. However there are restrictions for achieving conflict serializability in the database. Here are a few of them.

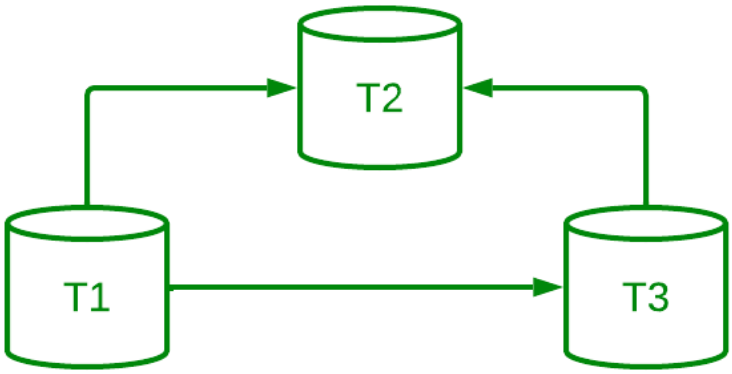
1. Different transactions should be used for the two procedures.
2. The identical data item should be present in both transactions.
3. Between the two operations, there should be at least one write operation.

Example

Three transactions—t1, t2, and t3—are active on a schedule "S" at once. Let's create a graph of precedence.

Transaction - 1 (t1)	Transaction - 2 (t2)	Transaction - 3 (t3)
R(a)		
	R(b)	
		R(b)
	W(b)	
W(a)		
		W(a)
	R(a)	
	W(a)	

It is a conflict serializable schedule as well as a serial schedule because the graph (a DAG) has no loops. We can also determine the order of transactions because it is a serial schedule.



DAG of transactions

As there is no incoming edge on Transaction 1, Transaction 1 will be executed first. T3 will run second because it only depends on T1. Due to its dependence on both T1 and T3, t2 will finally be executed.

Therefore, the serial schedule's equivalent order is: t1 --> t3 --> t2

Note: A schedule is unquestionably consistent if it is conflicting serializable. A non-conflicting serializable schedule, on the other hand, might or might not be serial. We employ the idea of View Serializability to further examine its serial behavior.

2. View Serializability

View serializability is a kind of operation in a serializable in which each transaction should provide some results, and these outcomes are the output of properly sequentially executing the data item. The view serializability, in contrast to conflict serialized, is concerned with avoiding database inconsistency. The view serializability feature of DBMS enables users to see databases in contradictory ways.

To further understand view serializability in DBMS, we need to understand the schedules S1 and S2. The two transactions T1 and T2 should be used to establish these two schedules. Each schedule must follow the three transactions in order to retain the equivalent of the transaction. These three circumstances are listed below.

1. The first prerequisite is that the same kind of transaction appears on every schedule. This requirement means that the same kind of group of transactions cannot appear in both schedules S1 and S2. The

schedules are not equal to one another if one schedule commits a transaction but it does not match the transaction of the other schedule.

2. The second requirement is that different read or write operations should not be used in either schedule. On the other hand, we say that two schedules are not similar if schedule S1 has two write operations whereas schedule S2 only has one. The number of the write operation must be the same in both schedules, however there is no issue if the number of the read operation is different.
3. The second to last requirement is that there should not be a conflict between either timetable. execution order for a single data item. Assume, for instance, that schedule S1's transaction is T1, and schedule S2's transaction is T2. The data item A is written by both the transaction T1 and the transaction T2. The schedules are not equal in this instance. However, we referred to the schedule as equivalent to one another if it had the same number of all write operations in the data item.

What is view equivalency?

Schedules (S1 and S2) must satisfy these two requirements in order to be viewed as equivalent:

1. The same piece of data must be read for the first time. For instance, if transaction t1 is reading "A" from the database in schedule S1, then t1 must also read A in schedule S2.
2. The same piece of data must be used for the final write. As an illustration, if transaction t1 updated A last in S1, it should also conduct final write in S2.
3. The middle sequence need to follow suit. As an illustration, if in S1 t1 is reading A, and t2 updates A, then in S2 t1 should read A, and t2 should update A.

View Serializability refers to the process of determining whether a schedule's views are equivalent.

Example

We have a schedule "S" with two concurrently running transactions, "t1" and "t2."

Schedule - S:

Transaction-1 (t1)	Transaction-2 (t2)
R(a)	
W(a)	
	R(a)
	W(a)
R(b)	
W(b)	
	R(b)
	W(b)

By switching between both transactions' mid-read-write operations, let's create its view equivalent schedule (S').

Schedule - S':

Transaction-1 (t1)	Transaction-2 (t2)
R(a)	

Transaction-1 (t1)	Transaction-2 (t2)
W(a)	
R(b)	
W(b)	
	R(a)
	W(a)
	R(b)
	W(b)

It is a view serializable schedule since a view similar schedule is conceivable.

Note: A conflict serializable schedule is always viewed as serializable, but vice versa is not always true.

Advantages of Serializability

1. **Execution is predictable:** In serializable, the DBMS's threads are all performed simultaneously. The DBMS doesn't include any such surprises. In DBMS, no data loss or corruption occurs and all variables are updated as intended.
2. DBMS executes each thread independently, making it much simpler to understand and troubleshoot each database thread. This can greatly simplify the debugging process. The concurrent process is therefore not a concern for us.
3. **Lower Costs:** The cost of the hardware required for the efficient operation of the database can be decreased with the aid of the

serializable property. It may also lower the price of developing the software.

4. **Increased Performance:** Since serializable executions provide developers the opportunity to optimize their code for performance, they occasionally outperform non-serializable equivalents.

For a DBMS transaction to be regarded as serializable, it must adhere to the ACID properties. In DBMS, serializability comes in a variety of forms, each having advantages and disadvantages of its own. Most of the time, choosing the best sort of serializability involves making a choice between performance and correctness.

Making the incorrect choice for serializability might result in database issues that are challenging to track down and resolve. You should now have a better knowledge of how serializability in DBMS functions and the different types that are available thanks to this guide.

Next Article

Joins in DBMS

Similar Reads

Conflict Serializability in DBMS

A schedule is a sequence in which operations (read, write, commit, abort) from multiple transactions are executed in a database. Serial or one by...

15+ min read

View Serializability in DBMS

In database systems, concurrent execution of transactions is used to improve resource utilization and system throughput. However,...

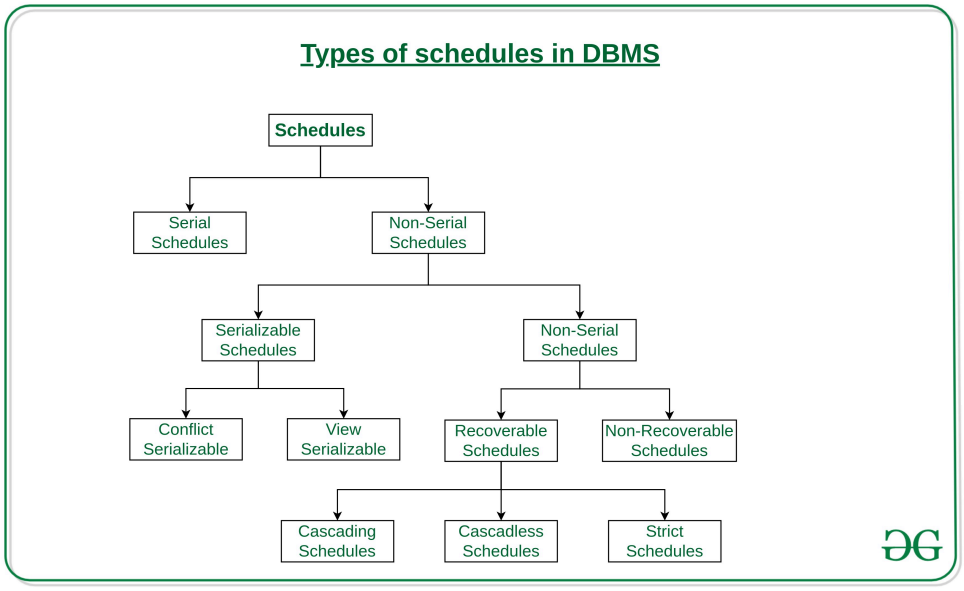
15+ min read

Recoverability in DBMS

Types of Schedules in DBMS

Last Updated : 28 Dec, 2024

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly. The basics of Transactions and Schedules is discussed in **Concurrency Control**, and **Transaction Isolation Levels in DBMS** articles. Here we will discuss various types of schedules.



1. **Serial Schedules:** Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules. **Example:** Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	

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

where R(A) denotes that a read operation is performed on some data item ‘A’ This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

2. **Non-Serial Schedule:** This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule. The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.
    1. **Serializable:** This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is

allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

1. Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:
   - They belong to different transactions
   - They operate on the same data item
   - At Least one of them is a write operation

2. View Serializable: A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

2. **Non-Serializable:** The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.
   1. Recoverable Schedule: Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i. **Example –** Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)

T_1	T_2
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct. There can be three types of recoverable schedule:

1. **Cascading Schedule:** Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

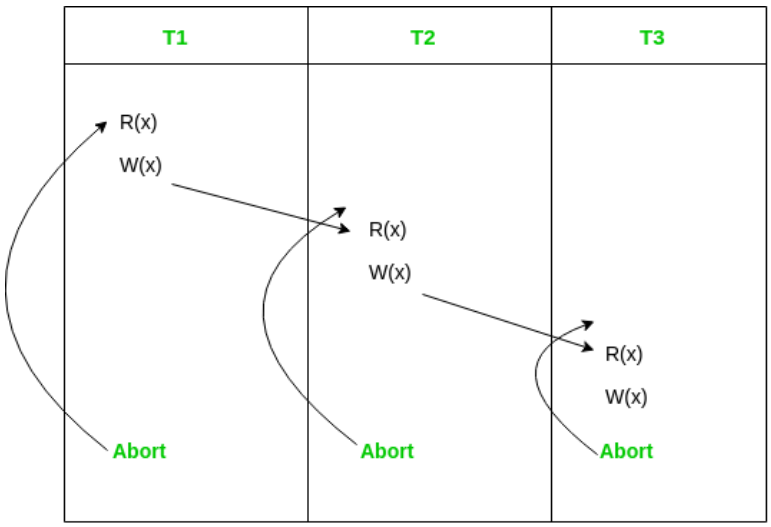


Figure - Cascading Abort

2. Cascadeless Schedule: Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule. In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i,

then the commit of T_j must read it after the commit of T_i.
Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits. **Example:** Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	

T_1	T_2
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T_1 aborts, T_2 will have to be aborted too in order to maintain the correctness of the schedule as T_2 has already read the uncommitted value written by T_1.

3. **Strict Schedule:** A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j. In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.
Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

2. **Non-Recoverable Schedule:Example:** Consider the following schedule involving two transactions T_1 and T_2.

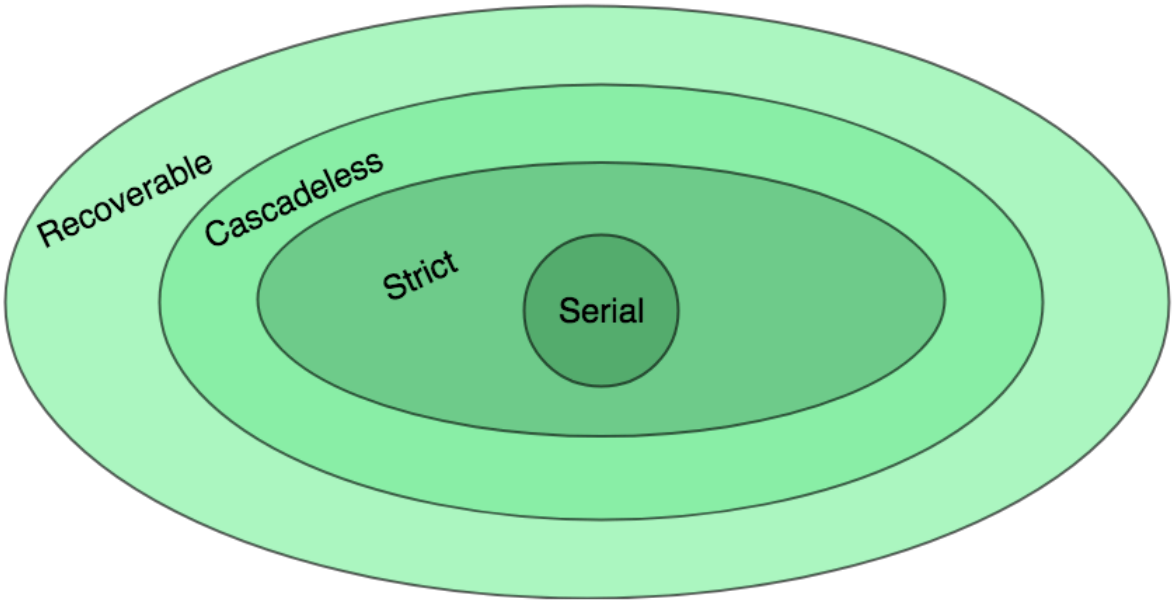
T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable** .

Note – It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:



Example:

Consider the following schedule:

```
S:R1(A), W2(A), Commit2, W1(A), W3(A), Commit3, Commit1
```

Which of the following is true? (A) The schedule is view serializable schedule and strict recoverable schedule (B) The schedule is non-serializable schedule and strict recoverable schedule (C) The schedule is non-serializable schedule and is not strict recoverable schedule. (D) The Schedule is serializable schedule and is not strict recoverable schedule

Solution:

The schedule can be re-written as:-

T_1	T_2	T_3
R(A)		
	W(A)	
	Commit	
W(A)		

T_1	T_2	T_3
		W(A)
		Commit
Commit		

First of all, it is a view serializable schedule as it has view equal serial schedule T_1 —> T_2 —> T_3 which satisfies the initial and updated reads and final write on variable A which is required for view serializability.
Now we can see there is write – write pair done by transactions T_1 followed by T_3 which is violating the above-mentioned condition of strict schedules as T_3 is supposed to do write operation only after T_1 commits which is violated in the given schedule. Hence the given schedule is serializable but not strict recoverable.
So, option (D) is correct.

Next Article

Conflict Serializability in DBMS

Similar Reads

Types of Schedules Based on Recoverability in DBMS

In a Database Management System (DBMS), multiple transactions often run at the same time, and their execution order is called a schedule. It is...

15+ min read

Conflict Serializability in DBMS

A schedule is a sequence in which operations (read, write, commit, abort) from multiple transactions are executed in a database. Serial or one by...

Lock Based Concurrency Control Protocol in DBMS

Last Updated : 23 Jan, 2025

In a Database Management System (DBMS), lock-based concurrency control (BCC) is a method used to manage how multiple transactions access the same data. This protocol ensures data consistency and integrity when multiple users interact with the database simultaneously.

This method uses locks to manage access to data, ensuring transactions don't clash and everything runs smoothly when multiple transactions happen at the same time. In this article, we'll take a closer look at how the Lock-Based Protocol works.

What is a Lock?

A lock is a variable associated with a data item that indicates whether it is currently in use or available for other operations. Locks are essential for managing access to data during concurrent transactions. When one transaction is accessing or modifying a data item, a lock ensures that other transactions cannot interfere with it, maintaining data integrity and preventing conflicts. This process, known as locking, is a widely used method to ensure smooth and consistent operation in database systems.

Lock Based Protocols

Lock-Based Protocols in DBMS ensure that a transaction cannot read or write data until it gets the necessary lock. Here's how they work:

- These protocols prevent concurrency issues by allowing only one transaction to access a specific data item at a time.
- Locks help multiple transactions work together smoothly by managing access to the database items.

- Locking is a common method used to maintain the serializability of transactions.
- A transaction must acquire a read lock or write lock on a data item before performing any read or write operations on it.

Types of Lock

1. **Shared Lock (S):** Shared Lock is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
2. **Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Read more about Types of Locks.

Rules of Locking

The basic rules for Locking are given below :

Read Lock (or) Shared Lock(S)

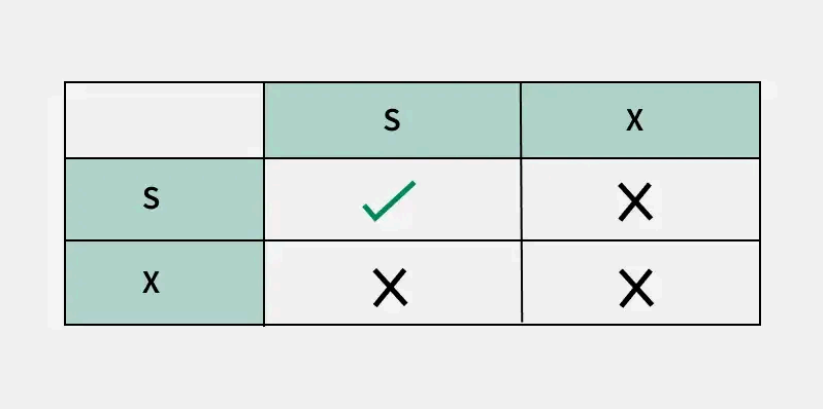
❖ If a Transaction has a Read lock on a data item, it can read the item but not update it.
❖ If a transaction has a Read lock on the data item, other transaction can obtain Read Lock on the data item but no Write Locks.
❖ So, the Read Lock is also called a Shared Lock.

Write Lock (or) Exclusive Lock (X)

❖ If a transaction has a write Lock on a data item, it can both read and update the data item.
❖ If a transaction has a write Lock on the data item, then other transactions cannot obtain either a Read lock or write lock on the data item.
❖ So, the Write Lock is also known as Exclusive Lock.

Lock Compatibility Matrix

- A transaction can acquire a lock on a data item only if the requested lock is compatible with existing locks held by other transactions.
- **Shared Locks (S):** Multiple transactions can hold shared locks on the same data item simultaneously.
- **Exclusive Lock (X):** If a transaction holds an exclusive lock on a data item, no other transaction can hold any type of lock on that item.
- If a requested lock is not compatible, the requesting transaction must wait until all incompatible locks are released by other transactions.
- Once the incompatible locks are released, the requested lock is granted.



Compatibility Matrix

Concurrency Control Protocols

Concurrency Control Protocols are the methods used to manage multiple transactions happening at the same time. They ensure that transactions are executed safely without interfering with each other, maintaining the accuracy and consistency of the database.

These protocols prevent issues like data conflicts, lost updates or inconsistent data by controlling how transactions access and modify data.

Types of Lock-Based Protocols

1. Simplistic Lock Protocol

It is the simplest method for locking data during a transaction. Simple lock-based protocols enable [illegible] to obtain a lock on the

data before inserting, deleting, or updating it. It will unlock the data item once the transaction is completed.

Example:

Consider a database with a single data item `X = 10`.

Transactions:

- **T1**: Wants to read and update `X`.
- **T2**: Wants to read `X`.

Steps:

1. T1 requests an exclusive lock on `X` to update its value. The lock is granted.
   - T1 reads `X = 10` and updates it to `X = 20`.
2. T2 requests a shared lock on `X` to read its value. Since T1 is holding an exclusive lock, T2 must wait.
3. T1 completes its operation and releases the lock.
4. T2 now gets the shared lock and reads the updated value `X = 20`.

This example shows how simplistic lock protocols handle concurrency but do not prevent problems like deadlocks or limits concurrency.

2. Pre-Claiming Lock Protocol

The Pre-Claiming Lock Protocol evaluates a transaction to identify all the data items that require locks. Before the transaction begins, it requests the database management system to grant locks on all necessary data elements. If all the requested locks are successfully acquired, the transaction proceeds. Once the transaction is completed, all locks are released. However, if any of the locks are unavailable, the transaction rolls back and waits until all required locks are granted before restarting.

Example:

Consider two transactions T1 and T2 and two data items, `X` and `Y`:

1. Transaction T1 declares th

- A write lock on X.
- A read lock on Y.

Since both locks are available, the system grants them. T1 starts execution:

- It updates X.
- It reads the value of Y.

2. While T1 is executing, Transaction T2 declares that it needs:
   - A read lock on X.

   However, since T1 already holds a write lock on X, T2's request is denied. T2 must wait until T1 completes its operations and releases the locks.
3. Once T1 finishes, it releases the locks on X and Y. The system now grants the read lock on X to T2, allowing it to proceed.

This method is simple but may lead to inefficiency in systems with a high number of transactions.

3. Two-phase locking (2PL)

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases :

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

For more detail refer the article Two-phase locking (2PL).

4. Strict Two-Phase Locking Protocol

Strict Two-Phase Locking requires that in addition to the 2-PL all Exclusive(X) locks held by the transaction be released until after the Transaction Commits.

For more details refer the article Strict Two-Phase Locking Protocol.

Problem With Simple Locking

Consider the Partial Schedule:

S.No	T_1	T_2
1	lock-X(B)	
2	read(B)	
3	B:=B-50	
4	write(B)	
5		lock-S(A)
6		read(A)
7		lock-S(B)
8	lock-X(A)	
9	.....	.....

1. Deadlock

In the given execution scenario, T1 holds an exclusive lock on B, while T2 holds a shared lock on A. At Statement 7, T2 requests a lock on B, and at Statement 8, T1 requests a lock on A. This situation creates a deadlock, as both transactions are waiting for resources held by the other, preventing either from proceeding with their execution.

2. Starvation

Starvation is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is properly designed.

Conclusion

In conclusion, lock-based concurrency control in a database management system (DBMS) uses locks to control access, avoid conflicts, and preserve the integrity of the database across multiple users. The protocol seeks to achieve a balance between concurrency and integrity by carefully controlling the acquisition and deletion of locks by operations.

Next Article

Graph Based Concurrency Control Protocol in DBMS

Similar Reads

Graph Based Concurrency Control Protocol in DBMS

In a Database Management System (DBMS), multiple transactions often run at the same time, which can lead to conflicts when they access the...

15+ min read

Concurrency Control in DBMS

In a database management system (DBMS), allowing transactions to run concurrently has significant advantages, such as better system resourc...

15+ min read

Two Phase Locking Protocol

Two Phase Locking Protocol

Last Updated : 09 Jan, 2025

The Two-Phase Locking (2PL) Protocol is an essential concept in database management systems used to maintain data consistency and ensure smooth operation when multiple transactions are happening simultaneously. It helps to prevent issues like data conflicts where two or more transactions try to access or modify the same data at the same time, potentially causing errors.

Two-Phase Locking is widely used to ensure serializability, meaning transactions occur in a sequence that maintains data accuracy. This article will explore the workings of the 2PL protocol, its types, advantages and its role in maintaining a reliable database system

Two Phase Locking

The Two-Phase Locking (2PL) Protocol is a key technique used in database management systems to manage how multiple transactions access and modify data at the same time. When many users or processes interact with a database, it's important to ensure that data remains consistent and error-free. Without proper management, issues like data conflicts or corruption can occur if two transactions try to use the same data simultaneously.

The Two-Phase Locking Protocol resolves this issue by defining clear rules for managing data locks. It divides a transaction into two phases:

1. **Growing Phase**: In this step, the transaction gathers all the locks it needs to access the required data. During this phase, it cannot release any locks.
2. **Shrinking Phase**: Once a transaction starts releasing locks, it cannot acquire any new ones. This ensures that no other transaction interferes with the ongoing process.

Types of Lock

Shared Lock (S): Shared Lock is also called a read-only lock, allows multiple transactions to access the same data item for reading at the same time. However, transactions with this lock cannot make changes to the data. A shared lock is requested using the lock-S instruction.

Exclusive Lock (X): An Exclusive Lock allows a transaction to both read and modify a data item. This lock is exclusive, meaning no other transaction can access the same data item while this lock is held. An exclusive lock is requested using the lock-X instruction.

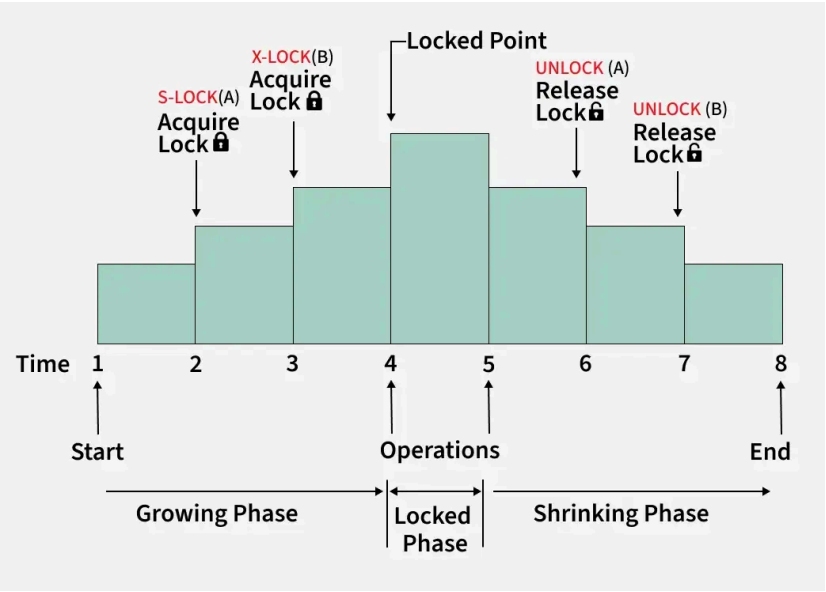
Read more about the Locks.

Lock Conversions

In the Two-Phase Locking Protocol, lock conversion means changing the type of lock on data while a transaction is happening. This process is carefully controlled to maintain consistency in the database.

1. **Upgrading a Lock**: This means changing a shared lock (S) to an exclusive lock (X). For example, if a transaction initially only needs to read data (S) but later decides it needs to update the same data, it can request an upgrade to an exclusive lock (X). However, this can only happen during the Growing Phase, where the transaction is still acquiring locks.
    - Example: A transaction reads a value (S lock) but then realizes it needs to modify the value. It upgrades to an X lock during the Growing Phase.

2. **Downgrading a Lock**: This means changing an exclusive lock (X) to a shared lock (S). For instance, if a transaction initially planned to modify data (X lock) but later decides it only needs to read it, it can downgrade the lock. However, this must happen during the Shrinking Phase, where the transaction is releasing locks.
    - Example: A transaction modifies a value (X lock) but later only needs to read the value, so it downgrades to an S lock during the Shrinking Phase.



2PL-Locking

These rules ensure that the Two-Phase Locking Protocol maintains consistency and avoids conflicts between transactions. By limiting when upgrades and downgrades can occur, the system prevents situations where multiple transactions interfere with each other's operations.

Read more about **Shared and Exclusive Locks.**

Let's see a transaction implementing 2-PL.

	T1	T2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	…….	…….
5	Unlock(A)	
6		Lock-X(C)

	T1	T2
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	…….	…….

This is a basic outline of a transaction that demonstrates how locking and unlocking work in the Two-Phase Locking Protocol (2PL).

Transaction T_1

- The growing Phase is from steps 1-3
- The shrinking Phase is from steps 5-7
- Lock Point at 3

Transaction T_2

- The growing Phase is from steps 2-6
- The shrinking Phase is from steps 8-9
- Lock Point at 6

Lock Point

The lock point in a transaction is the moment when the transaction finishes acquiring all the locks it needs. After this point, no new locks can be added, and the transaction starts releasing locks. It's a key step in the Two-Phase Locking Protocol to ensure the rules of growing and shrinking phases are followed.

Example of 2PL

Imagine a library system where multiple users can borrow or return books. Each action (like borrowing or returning) is treated as a

transaction. Here's how the Two-Phase Locking Protocol (2PL) works, including the lock point:

User A wants to:

1. Check the availability of Book X.
2. Borrow Book X if it's available.
3. Update the library's record.

Growing Phase (Locks are Acquired):

1. User A locks Book X with a shared lock (S) to check its availability.
2. After confirming the book is available, User A upgrades the lock to an exclusive lock (X) to borrow it.
3. User A locks the library's record to update the borrowing details.

Lock Point: Once User A has acquired all the necessary locks (on Book X and the library record), the transaction reaches the lock point. No more locks can be acquired after this.

Shrinking Phase (Locks are Released):

1. User A updates the record and releases the lock on the library's record.
2. User A finishes borrowing and releases the exclusive lock on Book X.

This process ensures that no other user can interfere with Book X or the library record during the transaction, maintaining data accuracy and consistency. The lock point ensures that all locks are acquired before any are released, following the 2PL rules.

Drawbacks of 2PL

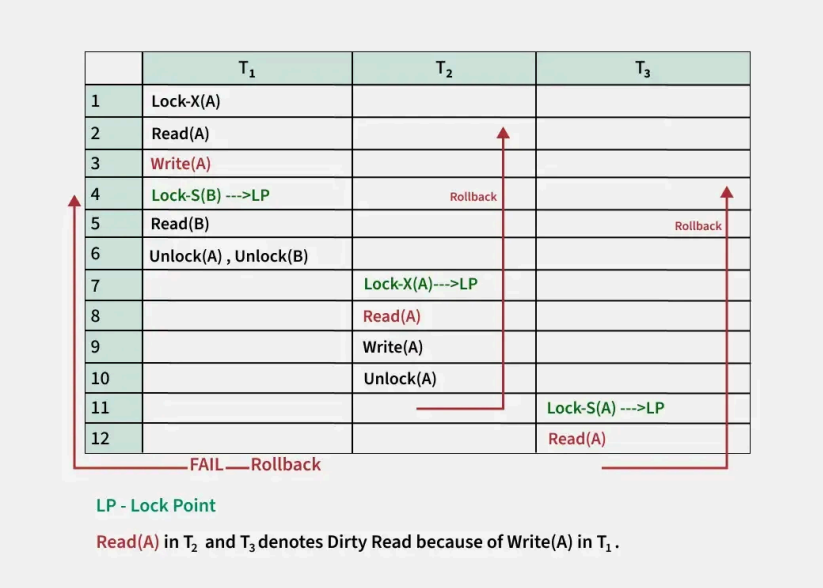
Two-phase locking (2PL) ensures that transactions are executed in the correct order by using two phases: acquiring and releasing locks. However, it has some drawbacks:

- **Deadlocks**: Transactions can get stuck waiting for each other's locks, causing them to freeze indefinitely.
- **Cascading Rollbacks**: If one transaction fails, others that depend on it might also fail leading to inefficiency and potential data issues.

- **Lock Contention**: Too many transactions competing for the same locks can slow down the system, especially when many users are working at the same time.
- **Limited Concurrency**: The strict rules of 2PL can reduce how many transactions can run at once, resulting in slower performance and longer wait times.

Cascading Rollbacks in 2-PL

Let's see the following Schedule:



Schedule

The image illustrates a transaction schedule using the Two-Phase Locking (2PL) protocol, showing the sequence of actions for three transactions T1, T2 and T3.

Key Points:

1. **Transaction T1:**
   - T1 acquires an exclusive lock (X) on data item A, performs a write operation on A and then acquires a shared lock (S) on B.
   - T1 reaches its lock point (LP) after acquiring all locks.
   - Eventually, T1 fails and a rollback is triggered, undoing its changes.

2. **Transaction T2:**

- T2 reads A after T1 writes A. This is called a **dirty read** because T1's write is not committed yet.
- When T1 rolls back, T2's operations become invalid, and it is also forced to rollback.

3. **Transaction T3:**
   - T3 reads A after T2 reads A. Since T2 depends on the uncommitted changes of T1, T3 indirectly relies on T1's changes.
   - When T1 rolls back, T3 is also forced to rollback even though it was not directly interacting with T1's operations.

Cascading Rollback Problem:

- Here, T2 and T3 both depend on uncommitted data from T1 (either directly or indirectly).
- When T1 fails and rolls back all dependent transactions (T2 and T3) must also rollback because their operations were based on invalid data.
- Cascading rollbacks waste system resources, reduce concurrency and lead to inefficiency.
- In large systems, this can significantly affect performance and make recovery more complex.

  How to avoid it?

  - **Strict 2PL**: Ensure that transactions release their locks only after they commit, preventing other transactions from accessing uncommitted changes.
  - **Rigorous 2PL**: Extend strict 2PL to hold both read and write locks until the transaction commits, offering even stronger guarantees.

Deadlock in 2-PL

Consider this simple example. We have two transactions T_1 and T_2.

Schedule: $\text{Lock-X}_1(A)$ $\text{Lock-X}_2(B)$ $\text{Lock-X}_1(B)$ $\text{Lock-X}_2(A)$

This sequence represents a locking scenario where two transactions, T1 and T2 are trying to lock two resources, A and B in a particular order. Here's what each step means:

1. **Lock-X1(A):**
   Transaction T1 acquires an exclusive lock on resource A. This means T1 has full control over A and no other transaction can use it until T1 releases the lock.
2. **Lock-X2(B):**
   Transaction T2 acquires an exclusive lock on resource B. Similarly, T2 now has full control over B and no other transaction can access B until T2 releases the lock.
3. **Lock-X1(B):**
   Transaction T1 tries to acquire an exclusive lock on resource B but T2 already holds the lock on B. So, T1 must wait for T2 to release the lock.
4. **Lock-X2(A):**
   At the same time, Transaction T2 tries to acquire an exclusive lock on resource A but T1 already holds the lock on A. So, T2 must wait for T1 to release the lock.

The above-mentioned type of 2-PL is called Basic 2PL. To sum it up, it ensures Conflict Serializability but does not prevent Cascading Rollback and Deadlock. To prevent from these issues Strict Two Phase Locking and Rigorous Two Phase Locking can be used.

Read more about Categories of Two Phase Locking.

GATE-Related Questions

1. GATE CS 2016-2 | Question 61
2. GATE CS 1999 | Question 31

Conclusion

The Two-Phase Locking (2PL) Protocol is an important method to ensure that database transactions are carried out in the correct order, keeping the data consistent and reliable. By dividing the process into

Timestamp based Concurrency Control

Last Updated : 21 Jan, 2025

Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously. This approach relies on timestamps to manage and coordinate the execution order of transactions. Refer to the timestamp of a transaction *T* as *TS(T)*.

What is Timestamp Ordering Protocol?

The Timestamp Ordering Protocol is a method used in database systems to order transactions based on their timestamps. A timestamp is a unique identifier assigned to each transaction, typically determined using the system clock or a logical counter. Transactions are executed in the ascending order of their timestamps, ensuring that older transactions get higher priority.

For example:

- If Transaction T1 enters the system first, it gets a timestamp TS(T1) = 007 (assumption).
- If Transaction T2 enters after T1, it gets a timestamp TS(T2) = 009 (assumption).

This means T1 is “older” than T2 and T1 should execute before T2 to maintain consistency.

Key Features of Timestamp Ordering Protocol:

Transaction Priority:

- Older transactions (those with smaller timestamps) are given higher priority.

- For example, if transaction T1 has a timestamp of 007 times and transaction T2 has a timestamp of 009 times, T1 will execute first as it entered the system earlier.

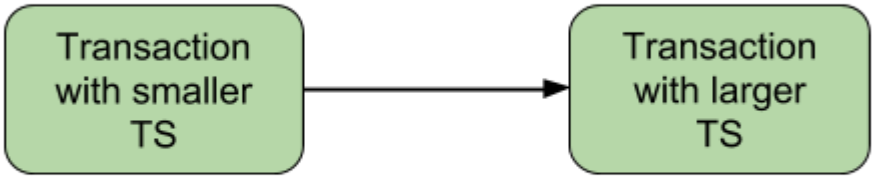
Early Conflict Management:

- Unlike lock-based protocols, which manage conflicts during execution, timestamp-based protocols start managing conflicts as soon as a transaction is created.

Ensuring Serializability:

- The protocol ensures that the schedule of transactions is serializable. This means the transactions can be executed in an order that is logically equivalent to their timestamp order.

Basic Timestamp Ordering



Precedence Graph for TS ordering

The Basic Timestamp Ordering (TO) Protocol is a method in database systems that uses timestamps to manage the order of transactions. Each transaction is assigned a unique timestamp when it enters the system ensuring that all operations follow a specific order making the schedule conflict-serializable and deadlock-free.

- Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that **$TS(T_i) < TS(T_j)$**.

- The protocol manages concurrent execution such that the timestamps determine the serializability order.

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

- Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with R_TS(X) & W_TS(X) to ensure that the Timestamp order is not violated.

This describes the Basic TO protocol in the following two cases:

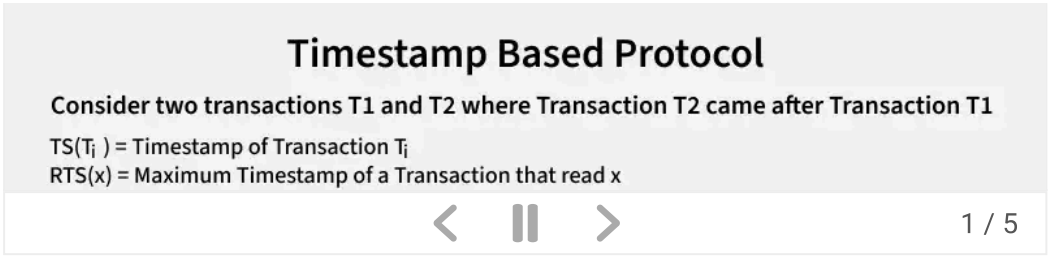
Whenever a Transaction *T* issues a W_item(X) operation, check the following conditions:

- If *R_TS(X) > TS(T)* and if *W_TS(X) > TS(T)*, then abort and rollback T and reject the operation. else,
- Execute W_item(X) operation of T and set W_TS(X) to TS(T) to the larger of TS(T) and current W_TS(X).

Whenever a Transaction *T* issues a R_item(X) operation, check the following conditions:

- If *W_TS(X) > TS(T)*, then abort and reject T and reject the operation, else
- If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever the Basic TO algorithm detects two conflicting operations that occur in an incorrect order, it rejects the latter of the two operations by aborting the Transaction that issued it.



Advantages of Basic TO Protocol

- **Conflict Serializable**: Ensures all conflicting operations follow the timestamp order.
- **Deadlock-Free**: Transactions do not wait for resources, preventing deadlocks.
- **Strict Ordering**: Operations are executed in a predefined, conflict-free order based on timestamps.

Drawbacks of Basic Timestamp Ordering (TO) Protocol

- **Cascading Rollbacks :** If a transaction is aborted, all dependent transactions must also be aborted, leading to inefficiency.
- **Starvation of Newer Transactions** : Older transactions are prioritized, which can delay or starve newer transactions.
- **High Overhead**: Maintaining and updating timestamps for every data item adds significant system overhead.
- **Inefficient for High Concurrency**: The strict ordering can reduce throughput in systems with many concurrent transactions.

Strict Timestamp Ordering

The Strict Timestamp Ordering Protocol is an enhanced version of the Basic Timestamp Ordering Protocol. It ensures a stricter control over the execution of transactions to avoid cascading rollbacks and maintain a more consistent s

Key Features

- **Strict Execution Order**: Transactions must execute in the exact order of their timestamps. Operations are delayed if executing them would violate the timestamp order, ensuring a strict schedule.

- **No Cascading Rollbacks**: To avoid cascading aborts, a transaction must delay its operations until all conflicting operations of older transactions are either committed or aborted.

- **Consistency and Serializability**: The protocol ensures conflict-serializable schedules by following strict ordering rules based on transaction timestamps.

For Read Operations (R_item(X)):

- A transaction T can read a data item X only if: W_TS(X), the timestamp of the last transaction that wrote to X, is less than or equal to TS(T), the timestamp of T and the transaction that last wrote to X has committed.

- If these conditions are not met, T's read operation is delayed until they are satisfied.

For Write Operations (W_item(X)):

- A transaction T can write to a data item X only if: R_TS(X), the timestamp of the last transaction that read X, and W_TS(X), the timestamp of the last transaction that wrote to X, are both less than or equal to TS(T) and all transactions that previously read or wrote X have committed.

- If these conditions are not met, T's write operation is delayed until all conflicting transactions are resolved.

Conclusion

Database Recovery Models

Last Updated : 20 Apr, 2023

Every database require a recovery model which signifies that what sort of backup is required or can be perform by the user to restore the data which could be lost due to any hardware failure or other issue.

There are generally three types of recovery models of database, these are explained as following below.

1. Simple Recovery :
In this model, the transaction logs get automatically removed without causing and change to the files size, because of this it is difficult to make log backups. Simple Recovery does not support backup of transaction log. It supports both full and bulk_logged backup operations.

Some operations that aren't supported by this model are : Log shipping, AlwaysOn or Mirroring and Point-in-time restore.

In this case the database is use only for testing and development. The data in this operation is static. It does not have the provision for point-to-time recovery.

2. Full Recovery :
Unlike simple recovery, it supports backups of transaction log. There will no loss of work due to damaged or lost files as this model keeps track of every operation performed on database.

It supports point-in-time for recovery for database, because of which it can recover upto an arbitrary point. When this model is use by database, the transaction logs will grow in huge number(infinitely) which will cause a problem like system crash. So to prevent it we must backup transaction log on regular basis.

This setup provides more options.

3. Bulk logged :
This model has similarity with Full Recovery Model as in both transaction logs are backup. It has high performance for bulk operations. It helps in importing bulk data quicker than other model and this keeps the transaction file size low. It did not support point-in-time recovery.

4. Differential Backup:
In addition to the backup types supported by the full and bulk-logged recovery models, the full recovery model also supports differential backups. A differential backup only backs up the data that has changed since the last full backup, which can save time and space compared to doing another full backup.

5. Log Sequence Number (LSN):
LSN is a unique identifier that is used to track every modification made to the database. It is used to ensure that all changes are recorded and to keep track of which backups need to be restored in case of a disaster.

6. Point-in-time Recovery:
This feature allows the user to restore the database to a specific point in time, using the transaction log backups. It is only supported in the Full Recovery Model and requires regular backups of the transaction log.

7. Increased Complexity:
As the recovery models become more advanced, the complexity of managing and maintaining the database also increases. This can require more expertise and resources from the database administrator.

8. Increased Storage Requirements:
Full and bulk-logged recovery models require regular backups of the transaction logs, which can quickly add up to a significant amount of storage space. This can also increase the cost of maintaining the database.

Database Recovery Techniques in DBMS

Last Updated : 30 Sep, 2024

Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS

- **Rollback/Undo Recovery Technique**
- **Commit/Redo Recovery Technique**
- **CheckPoint Recovery Technique**

Database recovery techniques ensure data integrity in case of system failures. Understanding how these techniques work is crucial for managing databases effectively. The **GATE CS Self-Paced Course** covers recovery strategies in DBMS, providing practical insights into maintaining data consistency

Rollback/Undo Recovery Technique

The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been

completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

Commit/Redo Recovery Technique

The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

Checkpoint Recovery Technique

Checkpoint Recoveryis a technique used to improve data integrity and system stability, especially in databases and distributed systems. It entails preserving the system's state at regular intervals, known as checkpoints, at which all ongoing transactions are either completed or not initiated. This saved state, which includes memory and CPU registers, is kept in stable, non-volatile storage so that it can withstand system crashes. In the event of a breakdown, the system can be restored to the most recent checkpoint, which reduces data loss and downtime. The frequency of checkpoint formation is carefully regulated to decrease system overhead while ensuring that recent data may be restored quickly.

Overall, recovery techniques are essential to ensure data consistency and availability in Database Management System, and each technique

has its own advantages and limitations that must be considered in the design of a recovery system.

Database Systems

There are both automatic and non-automatic ways for both, backing up data and recovery from any failure situations. The techniques used to recover lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect command execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred updates and immediate updates or backing up data can be used. Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur during the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

- **The log is kept on disk start_transaction(T):** This log entry records that transaction T starts the execution.

- **read_item(T, X):** This log entry records that transaction T reads the value of database item X.

- **write_item(T, X, old_value, new_value):** This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.

- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.

- **abort(T):** This records that transaction T has been aborted.

- **checkpoint:** A checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a

storage disk. Checkpoint declares a point before which the DBMS was in a consistent state, and all the transactions were committed.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it will not **abort** (terminate without completing). Once committed, the transaction is permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, the item is searched back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

- **Undoing:** If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to old-value. There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates.

- **Deferred Update:** This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm.**

- **Immediate Update:** In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in

a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**

- **Caching/Buffering:** In this one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under the control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.

- **Shadow Paging:** It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to the original are updated to refer new replacement page.

- **Backward Recovery:** The term ” **Rollback** ” and ” **UNDO** ” can also refer to backward recovery. When a backup of the data is not available and previous modifications need to be undone, this technique can be helpful. With the backward recovery method, unused modifications are removed and the database is returned to its prior condition. All adjustments made during the previous traction are reversed during the backward recovery. In other words, it reprocesses valid transactions and undoes the erroneous database updates.

- **Forward Recovery:** “ **Roll forward** “and ” **REDO** ” refers to forwarding recovery. When a database needs to be updated with all changes verified, this forward recovery technique is helpful. Some failed

transactions in this database are applied to the database to roll those modifications forward. In other words, the database is restored using preserved data and valid transactions counted by their past saves.

Backup Techniques

There are different types of Backup Techniques. Some of them are listed below.

- **Full database Backup:** In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.

- **Differential Backup:** It stores only the data changes that have occurred since the last full database backup. When some data has changed many times since the last full database backup, a differential backup stores the most recent version of the changed data. For this first, we need to restore a full database backup.

- **Transaction Log Backup:** In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transactions that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.

Conclusion

For data availability and consistency to always be ensured, database systems must be failsafe. Restoring database consistency may require employing many recovery techniques, such as rollback/undo, commit/redo, and checkpoint recovery. These solutions leverage system and transaction logs to monitor and regulate data changes. The optimal recovery approach is determined by the particular requirements and restrictions of the database system. Correct design of recovery

What is Multi-Version Concurrency Control (MVCC) in DBMS?

Last Updated : 21 Mar, 2024

Multi-Version Concurrency Control (MVCC) is a database optimization method, that makes redundant copies of records to allow for safe concurrent reading and updating of data. DBMS reads and writes are not blocked by one another while using MVCC. A technique called concurrency control keeps concurrent processes running to avoid read/write conflicts or other irregularities in a database.

Whenever it has to be updated rather than replacing the old one with the new information an MVCC database generates a newer version of the data item.

What is Multi-Version Concurrency Control (MVCC) in DBMS?

Multi-Version Concurrency Control is a technology, utilized to enhance databases by resolving concurrency problems and also data locking by preserving older database versions. When many tasks attempt to update the same piece of data simultaneously, MVCC causes a conflict and necessitates a retry from one or more of the processes.

Types of Multi-Version Concurrency Control (MVCC) in DBMS

Below are some types of Multi-Version Concurrency Control (MVCC) in DBMS

- **Timestamp-based MVCC:** The data visibility to transactions is defined by the unique timestamp assigned to each transaction that creates a new version of a record.

- **Snapshot-based MVCC:** This utilizes the database snapshot that is created at the beginning of a transaction to supply the information that is needed for the transaction.
- **History-based MVCC**: This Keeps track of every modification made to a record, making transaction rollbacks simple.
- **Hybrid MVCC: This** coordinates data flexibility and performance by combining two or more MVCC approaches.

How Does Multi-Version Concurrency Control (MVCC) in DBMS Works?

- In the database, every tuple has a version number. The tuple with the greatest version number can have a read operation done on it simultaneously.
- Only a copy of the record may be used for writing operations.
- While the copy is being updated concurrently, the user may still view the previous version.
- The version number is increased upon successful completion of the writing process.
- The upgraded version is now used for every new record operation and every time there is an update, this cycle is repeated.

Implementation of Multi-Version Concurrency Control (MVCC) in DBMS

- MVCC operates time stamps (TS) and increases transaction IDs to assure transactional consistency. MVCC manage many copies of the object, ensuring that a transaction (T) never has to wait to read a database object (P).
- A specific transaction Ti can read the most recent version of the object, which comes before the transaction's Read Timestamp RTS(Ti) since each version of object P contains both a Read Timestamp and a Write Timestamp.
- If there are other pending transactions to the same object that have an earlier Read Timestamp (RTS), then a Write cannot be completed.

- You cannot finish your checkout transaction until the person in front of you has finished theirs, much as when you are waiting in a queue at the shop.
- To reiterate, each object (P) has a Timestamp (TS). Should transaction Ti attempt to Write to an object and its Timestamp (TS) exceeds the object's current Read Timestamp, $TS(Ti) < RTS(P)$, the transaction will be cancelled and retried.
- Ti makes a new copy of object P and sets its read/write timestamp (TS) to the transaction timestamp $(TS \twoheadleftarrow TS(Ti))$.

Advantages of Multi-Version Concurrency Control (MVCC) in DBMS

Below are some advantages of Multi-Version Concurrency Control in DBMS

- **The reduced read-and-write necessity for database locks:** The database can support many read-and-write transactions without locking the entire system thanks to MVCC.
- **Increased Concurrency:** This Enables several users to use the system at once.
- **Minimize read operation delays:** By enabling concurrent read operations, MVCC helps to cut down on read times for data.
- **Accuracy and consistency:** Preserves data integrity over several concurrent transactions.

Disadvantages of Multi-Version Concurrency Control (MVCC) in DBMS

Below are some disadvantages of Multi-Version Concurrency Control in DBMS

- **Overhead:** Keeping track of many data versions might result in overhead.
- **Garbage collecting:** To get rid of outdated data versions, MVCC needs effective garbage collecting systems.

- **Increase the size of the database:** Expand the capacity of the database since MVCC generates numerous copies of the records and/or tuples.
- **Complexity:** Compared to more straightforward locking systems, MVCC usage might be more difficult.

Conclusion

In conclusion, Multiversion Concurrency Control (MVCC) in DBMS is a database optimization method, that makes redundant copies of records to allow for safe concurrent reading and updating of data. When many tasks attempt to update the same piece of data simultaneously, MVCC causes a conflict and necessitates a retry from one or more of the processes.

Next Article

What are the Strategies for Data Migration in DBMS?

Similar Reads

Storage Types in DBMS

The records in databases are stored in file formats. Physically, the data is stored in electromagnetic format on a device. The electromagnetic...

15+ min read

Multiversion Concurrency Control (MVCC) in PostgreSQL

PostgreSQL is a powerful open-source relational database management system known for its robustness and reliability. One of its key features...

4 min read

Concurrency Control in DBMS

In a database management system (DBMS), allowing transactions to run concurrently has significant advantages, such as better system resourc...

What is Optimistic Concurrency Control in DBMS

DBMS | Database | Big Data Analytics

All data items are updated at the end of the transaction, at the end, if any data item is found inconsistent with respect to the value in, then the transaction is rolled back.

Check for conflicts at the end of the transaction. No checking while the transaction is executing. Checks are all made at once, so low transaction execution overhead. Updates are not applied until end-transaction. They are applied to local copies in a transaction space.

Phases

The optimistic concurrency control has three phases, which are explained below –

Read Phase

Various data items are read and stored in temporary variables (local copies). All operations are performed in these variables without updating the database.

Validation Phase

All concurrent data items are checked to ensure serializability will not be validated if the transaction updates are actually applied to the database. Any changes in the value cause the transaction rollback. The transaction timestamps are used and the write-sets and read-sets are maintained.

To check that transaction A does not interfere with transaction B the following must hold –

- TransB completes its write phase before TransA starts the read phase.
- TransA starts its write phase after TransB completes its write phase, and the read set of TransA has no items in common with the write set of TransB.
- Both the read set and write set of TransA have no items in common with the write set of TransB and TransB completes its read before TransA completes its read Phase.

Write Phase

The transaction updates applied to the database if the validation is successful. Otherwise, updates are discarded and transactions are aborted and restarted. It does not use any locks hence deadlock free, however starvation problems of data items may occur.

Problem

S: W1(X), r2(Y), r1(Y), r2(X).

T1 -3

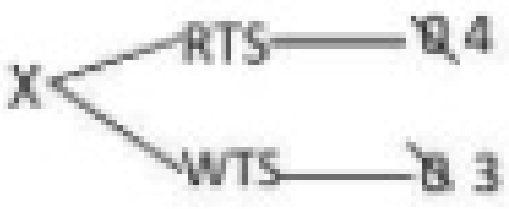
T2 – 4

Check whether timestamp ordering protocols allow schedule S.

Solution

Initially for a data-item X, RTS(X)=0, WTS(X)=0

Initially for a data-item Y, RTS(Y)=0, WTS(Y)=0



For W1(X) : TS(Ti)<RTS(X) i.e.
TS(T1)<RTS(X)
TS(T1)<WTS(X)
3<0 (FALSE)

=>goto else and perform write operation w1(X) and WTS(X)=3

For r2(Y): TS(T2)<WTS(Y)
4<0 (FALSE)

=>goto else and perform read operation r2(Y) and RTS(Y)=4

For r1(Y) :TS(T1)<WTS(Y)
3<0 (FALSE)

=>goto else and perform read operation r1(Y).

For r2(X) : TS(T2)<WTS(X)
4<3 (FALSE)

=>goto else and perform read operation r2(X) and RTS(X)=4