Transaction Management

Transaction management ensures data integrity in systems that involve changes to multiple data elements. Here's a quick breakdown:

What it is:

- A process that oversees database operations to guarantee data consistency.
- Ensures all parts of a data change are completed successfully or none are applied.

Why it's important:

- Prevents partial updates or inconsistencies in databases.
- Maintains data reliability, especially in multi-user environments.

How it works:

- **ACID Properties:** Transactions uphold ACID properties for data reliability.
- **Concurrency Control:** Manages multiple users accessing data simultaneously to avoid conflicts (e.g., locking a record while updating).

Example: Imagine an ATM withdrawal. Transaction management ensures:

- Sufficient funds are available (checks balance).
- Funds are deducted (updates balance).
- Cash is dispensed (only if previous steps succeed).

Basics Concepts

Transaction:

- A single unit of work in a database system.
- Represents a series of database operations (reads, writes, etc.).
- Must succeed entirely or fail entirely (ACID properties).

Example: Transferring money between accounts. Debiting one account and crediting another must both happen or neither does.

Deadlock:

- A situation where two or more transactions are waiting for resources held by each other.

- Can lead to system hangs.

Example: Transaction A holding a lock on resource X needs resource Y held by transaction B, and vice versa.

Savepoints:

- Mark a point within a transaction where a rollback can occur.
- Useful for complex transactions where you might only need to undo a specific part.

Example: Transferring money with multiple debits and credits. A savepoint could be set before each debit to rollback only that part if needed.

Transaction States in DBMS

Transaction States describe the different stages a transaction goes through in its lifecycle within a Database Management System. It ensures data consistency during database operations. There are four main states:

1. Active State:

- Initial state of a transaction.
- Executes all its database instructions (reads, writes, etc.).
    - Example: Transferring funds between accounts (withdrawing from A, depositing to B).

2. Partially Committed State:

- All instructions executed successfully.
- Changes are in temporary storage (memory buffers) but not yet permanent on disk.
    - Example: Debited account A in the transfer example.

3. Committed State:

- Transaction completes successfully.
- Changes from temporary storage are written permanently to the database.
- Transaction is terminated.
    - Example: Credited account B and transaction is complete in the transfer example.

4. Failed State (Aborted State):

- An error occurs during execution.

- Transaction cannot be completed.
- Changes are rolled back (undone) to maintain data consistency.
- Transaction is terminated.
    - Example: System crash during the transfer, so the debit from A is reversed.

Remember: Transactions follow the ACID properties to ensure data integrity. These states ensure the "Atomicity" property, where a transaction is all-or-nothing.

ACID Properties

ACID properties are a set of guidelines that ensure data integrity and consistency in database transactions. They guarantee that a series of database operations are treated as a single unit, and the database remains in a valid state even during failures.

1. Atomicity: All operations in a transaction are treated as one. Either all succeed (committed) or none do (rolled back).

- **Example:** Imagine an ATM withdrawal. If there's not enough balance (failing operation), the transaction rolls back and no money is dispensed.

2. Consistency: A transaction transforms the database from one valid state to another.

- **Example:** Inventory management. A transaction for selling an item should reduce the stock count by 1, not leave it unchanged or with a negative value.

3. Isolation: Concurrent transactions are isolated from each other, preventing conflicts.

- **Example:** Two users trying to buy the last item. Transaction isolation ensures only one purchase goes through, avoiding data inconsistencies.

4. Durability: Once a transaction commits, the changes are permanent and survive system failures.

- **Example:** Even if the power goes out after a successful money transfer, the updated balances are persisted in the database.

Key Points to Remember:

- Transactions ensure data consistency in multi-user environments.
- ACID properties guarantee reliable data manipulation.
- Understanding transaction states helps manage successful completion or failure scenarios.

Operations in Transactions

Transactions in a DBMS are a series of logically related operations that ensure data integrity. They act like mini-programs that perform a specific task on the database.

1. Data Access Operations

- **Read (R):** Retrieves data from the database and stores it in temporary memory for processing.
    - **Example:** In a bank transaction, reading the current balance of an account before transferring funds.
- **Write (W):** Modifies existing data in the database based on the transaction's logic.
    - **Example:** Updating the account balance after a successful transfer.

2. Transaction Control Operations

- **Commit:** Makes all changes performed by the transaction permanent in the database.
    - **Example:** After successfully transferring funds, the commit operation finalizes the changes in both accounts.
- **Rollback:** Undoes all changes made by the transaction if any errors occur or the transaction is cancelled.
    - **Example:** If there are insufficient funds during a transfer, the rollback operation ensures the original balances remain unchanged.

Storage Structures

In a DBMS, data is stored using specific structures on physical storage devices. These structures impact how efficiently data can be accessed and manipulated.

Types of Storage Structures:

1. File Organization:

- **Heap Files:** Records are stored anywhere with free space. Good for inserts but slow for searching specific data. (e.g., storing emails)
- **Sorted Files:** Records are ordered based on a key field. Efficient for searching with the key but inserts require reordering data. (e.g., phonebook)
- **Hashed Files:** Records are stored in buckets based on a hash function of a key field. Fast for retrieving specific records using the key but less efficient for range queries. (e.g., password files)

2. Indexed Organization:

- **B-Trees:** Hierarchical tree structure for storing sorted data. Efficient for searching ranges and specific data. (e.g., most database tables)
- **ISAM (Indexed Sequential Access Method):** Combines features of sorted files and B-Trees. Good for retrieval based on key and sequential access. (e.g., transaction logs)

Choosing the right structure depends on factors like:

- Access patterns (frequent searches, inserts etc.)
- Type of queries (range, specific value)
- Performance requirements (speed, efficiency)

Concurrent Executions

Concept: Concurrent execution refers to the ability of a Database Management System (DBMS) to process multiple transactions simultaneously in a shared database.

Benefits:

- **Increased Throughput:** More transactions can be completed in a given timeframe, improving overall system performance.
- **Reduced Wait Time:** Transactions don't have to wait for others to finish, leading to faster response times.
- **Improved Resource Utilization:** Processors can be kept busy even if one transaction is waiting for I/O operations.

Challenges: Concurrent execution can lead to data inconsistencies if not managed properly. Here's why:

- **Interleaved Execution:** Transactions might access and modify the same data items at the same time, leading to unpredictable results.

Example: Consider two transactions (T1 and T2) modifying the account balance of Sandy (A_Sandy) which is currently $100:

- T1 reads A_Sandy ($100), withdraws $50, and updates the balance.
- T2 (concurrently) reads A_Sandy ($100), deposits $70, and updates the balance.

If not controlled, the final balance could be incorrect (e.g., $120 instead of the expected $150).

- **Solutions:** DBMS employs concurrency control mechanisms to ensure data integrity. These mechanisms include locking and optimistic concurrency control.

In essence, concurrent execution offers performance benefits but requires careful management to avoid data inconsistencies.

Serializability

- Ensures that concurrent transactions appear to execute in a specific order, even though they might be happening simultaneously.
- Prevents inconsistencies in the database that can arise due to uncoordinated execution.
- Maintains data integrity by guaranteeing predictable outcomes.

Types of Serializability:

1. Conflict Serializability:

- Focuses on preventing data inconsistencies arising from conflicting operations on the same data item.
- Two operations conflict if:
    - Both transactions read and write the same data item (read-write conflict).
    - Both transactions write to the same data item (write-write conflict).
- Ensures that a concurrent schedule produces the same outcome as if transactions were executed sequentially, with conflicting operations appearing in a serial order.
- **Example:**
    - Transaction T1 reads a value (X) from a data item.
    - Transaction T2 writes a new value (Y) to the same data item.
    - In a conflict serializable schedule, either T1 reads X before T2 writes Y, or T2 writes Y before T1 reads (but not both).

2. View Serializability:

- Less restrictive than conflict serializability.
- Focuses on the final outcome of transactions, ensuring they produce the same results as if executed sequentially in some valid order.
- Does not consider the exact order of conflicting operations within a transaction, as long as the final result is consistent.
- Allows for more concurrency compared to conflict serializability.
- **Example: (Only god can help you understand this one but i can try)**

    - Transaction T1 reads a value (X) from a data item.
    - Transaction T2 writes a new value (Y) to the same data item.
    - In a view serializable schedule, the final database state reflects:
    - Either: T1 reads the old value of X, and then T2 writes the new value of Y. (Basically, T1 doesn't see the change T2 makes.)
    - Or: T2 writes the new value of Y, and then T1 transaction reads the new value Y. (No transaction reads the old value of X after the change by T2.)

Key point: The final state never shows both transactions reading the old value of X. There's always a clear order of reading the old value (X) and seeing the update (Y).

Concurrency Control

- **What it is:** Concurrency control is a mechanism in DBMS that manages concurrent access to data by multiple users or applications. It ensures data consistency and integrity in a multi-user environment.
- **Why it's important:** Concurrent execution of transactions can lead to inconsistencies if not controlled. Imagine two bank transactions happening at the same time, both trying to update the same account balance. Concurrency control prevents such conflicts.

Goals of Concurrency Control:

- ACID properties
- Serializability

Concurrency Problems

- Issues that arise when multiple users or processes try to access and modify the same data at the same time in a concurrent (overlapping) manner.
- Can occur in various systems, but particularly common in database management systems (DBMS).

Types of Concurrency Problems:

- **Lost Update:** Two transactions update the same data, but only one change saves.
- **Unrepeatable Read:** Reading the same data twice gives different results due to another transaction's update.
- **Phantom Read:** A new data item appears (or disappears) during a read due to another transaction's insert (or delete).
- **Dirty Read:** Reading uncommitted data that might be rolled back later.

Preventing Concurrency Problems:

- **Locking Mechanisms:** Locks prevent conflicts during data updates.
- **Timestamp Ordering:** Timestamps ensure a specific execution order.
- **Optimistic Concurrency Control:** Allows concurrency but checks for conflicts later.
- **Database Isolation Levels:** Define the degree of isolation between transactions.

Concurrency Control Protocols

Concurrency control protocols are mechanisms used in DBMS to manage concurrent access to data by multiple users or processes. Their goal is to ensure data consistency and integrity despite these simultaneous operations.

Here's a breakdown of the main types:

1. Lock-Based Protocols

- Transactions acquire locks on specific data items (records, tables) before modifying them.
- Prevents other transactions from conflicting modifications.
    - **Example:** Two users trying to update the same bank account balance. One user acquires a lock, preventing the other from modifying it until the first user finishes.

Modes of Locks

- **Exclusive Lock (X-lock):** Grants exclusive access to a data item. Only the holding transaction can modify the data. Other transactions can only read the data if the lock is granted in a specific mode (covered later).
- **Shared Lock (S-lock):** Grants read access to a data item. Multiple transactions can hold shared locks on the same data item concurrently.

Granting of Locks

The lock manager determines if a lock request can be granted based on the current locking state and the type of lock requested (X or S). Here are some common scenarios:

- **Granting an X-lock:** If no other transaction holds any lock (S or X) on the data item, the X-lock is granted.
- **Granting an S-lock:** If only other transactions hold S-locks, the new S-lock is granted (maintaining read concurrency). An X-lock on the data item blocks granting an S-lock.

Two-Phase Locking Protocol (2PL)

2PL ensures a transaction's locking actions have a well-defined order, preventing inconsistencies. It operates in two phases:

- **Growing Phase:** The transaction acquires locks (X or S) on data items it needs. It cannot release any locks during this phase.
- **Shrinking Phase:** The transaction releases locks it no longer needs. It cannot acquire new locks in this phase.

Example: Transaction B wants to transfer funds between two accounts. In the growing phase, it acquires X-locks on both accounts. Then, in the shrinking phase, it releases the lock on the first account after updating its balance and acquires an X-lock on the second account before updating it. This ensures no other transaction can interfere with the transfer process.

Lock Point (Additional): The Point at which the growing phase ends, i.e., when a transaction takes the final lock it needs to carry on its work.

2. Timestamp-Based Protocols

- Each transaction receives a unique timestamp when it starts.
- Transactions are serialized based on their timestamps, ensuring a specific execution order.
- Useful for ensuring a specific order of execution when needed.
    - **Example:** Multiple travel booking transactions with timestamps. Ensures transactions are processed in the order they were received, preventing double-booking.

i) Timestamp Ordering Protocol

- Assigns a unique timestamp (TS) to each transaction.
- Ensures serializability by ordering transactions based on their timestamps.
    - Transaction with a smaller timestamp executes first.

Rules:

- Reads are processed before writes based on timestamps (Ri(X) before Wj(X) only if TS(Ti) < TS(Tj)).
- Conflicting writes are not allowed (writes to the same data item by different transactions).
- Any violation leads to transaction rollback.

- **Example:** Consider transactions T1 and T2 with TS(T1) < TS(T2). If T1 reads X before T2 writes X, the protocol ensures T1 reads the value written by T2, maintaining consistency.

ii) Thomas Write Rule (Modification of Timestamp Ordering Protocol)

- The Thomas write rule is a concurrency control mechanism used in database systems to ensure data consistency.
- Improves efficiency over basic timestamp ordering.
- Allows for some harmless operations to improve concurrency.

Rules:

Similar to Timestamp Ordering Protocol for reads.

For Writes:

- If TS(Ti) < R-TS(X) (Timestamp of last read on X), then the write operation is invalid and Ti is rolled back. This is because Ti is trying to write a value that was previously read, violating consistency.
- If TS(Ti) < W-TS(X) (Timestamp of last write on X), the write is ignored. This is because the write from Ti is outdated and won't affect the final outcome.
- Otherwise, the write operation is executed, and W-TS(X) is updated to TS(Ti).

Benefit: Allows ignoring outdated writes, improving performance compared to strictly rolling back transactions.

Example: Transaction T1 reads X (R-TS(X) = TS(T1)) and then another transaction T2 writes X (W-TS(X) = TS(T2) > TS(T1)). If T3 tries to write X with TS(T3) < TS(T2), the write is ignored as it's outdated.

Note: Thomas Write Rule provides better concurrency compared to the basic Timestamp Ordering Protocol.

3. Validation-Based Protocols

- Transactions are validated after their execution, checking for conflicts with previously committed transactions.
- If a conflict is found, the transaction is rolled back (undone).
- Suitable for high-concurrency environments with frequent reads.
    - **Example:** An online auction where multiple users might bid on the same item at once. Validation ensures only the highest unique bid is committed.

Process:

- **Read Phase:** Transaction reads data and stores it in local copies.
- **Validation Phase:** Local copies are compared against actual data for conflicts.
    - If a conflict exists (e.g., another transaction modified the data), the transaction is restarted.
- **Write Phase (if validation succeeds):** Local copies update the actual database.

Deadlock Handling

Deadlocks occur in a DBMS when two or more transactions wait indefinitely for resources (locks) held by each other. This creates a circular dependency, halting all involved transactions.

1. Deadlock Prevention:

- **Prevents deadlocks altogether.**
- DBMS analyzes transactions to ensure resource allocation never leads to a deadlock scenario.
- **Example:** Enforce a fixed ordering for acquiring resources (e.g., always lock tables in alphabetical order).
- **Drawback:** Can restrict concurrency and system performance.

2. Deadlock Avoidance:

- **Prevents potential deadlocks during transaction execution.**
- DBMS maintains information about resource requests and uses algorithms to predict and avoid deadlocks.
- **Example:** Banker's Algorithm tracks resource availability and grants requests only if they won't lead to a deadlock.
- **Overhead:** Requires additional bookkeeping and analysis by the DBMS.

3. Deadlock Detection and Recovery:

- **Allows deadlocks to occur but detects them for resolution.**
- DBMS monitors transactions and identifies wait-for graphs to detect deadlocks.
- **Recovery options:**
    - **Rollback:** Abort one or more involved transactions, releasing their locks. (**Example:** Choose the transaction that has completed the least work to minimize rollback cost.)
    - **Wait timeout:** Set a time limit for transactions waiting for locks. If the timeout occurs, rollback the waiting transaction.