**Types of DBMS**

* A DBMS is single-user if at most one user can use the system at a time.
* A DBMS is multi-user if many users can use the system concurrently.

**Database Operations**

**1. Database Operation (Read and Write)**

* **Read Operation:**
  + SELECT balance FROM accounts WHERE account\_id = 123;
* **Write Operation:** 
  + **INSERT** INTO orders (order\_id, customer\_id, order\_date) VALUES (456, 101, SYSDATE);
  + **UPDATE** accounts SET balance = balance - 50 WHERE account\_id = 123;
  + **DELETE** FROM products WHERE product\_id = 789;

**2. Non-Database Operation (Calculations/Formula)**

These are operations performed by the application or within the transaction's logic that do not directly interact with the database to read or write persistent data. Example: Data validation - Checking if a user's input for an email address matches a regex pattern before even attempting to save it to the database.

**3. Database Consistent State**

A database is said to be in a consistent state if it satisfies all defined integrity constraints, rules, and business logic. These constraints ensure the accuracy, validity, and reliability of the data.

* **Key/Entity Integrity:** Ensures uniqueness and non-nullability of primary keys. (e.g., CustId in Customer table must be unique and not null).
* **Referential Integrity:** Ensures that foreign key values in one table correctly reference existing primary key values in another table, or are null. (e.g., Order.CustId must exist in Customer.CustId or be null). This prevents "orphan" records.
* **Domain Integrity:** Ensures that column values fall within a specified domain (e.g., data type, length, value range). (e.g., SAL in EMP must be >= 0).
* **Check Constraints:** Custom rules applied to columns or tables. (e.g., SAL >= 0).
* **Unique Constraints:** Ensure all values in a column or set of columns are unique (e.g., DriverLicenseNo in PERSON is unique).

**4. Database Inconsistent State**

A database is in an inconsistent state if it violates one or more of the defined integrity constraints, rules, or business logic. This can lead to inaccurate, invalid, or unreliable data.

* Inserting duplicate primary keys
* Inserting foreign keys that don't exist in the parent table
* Violating a check constraint

**Database Transaction**

A white paper with black text

AI-generated content may be incorrect.Database transaction is a **logical unit of work** that involves one or more database operations (e.g., SELECT, INSERT, UPDATE, DELETE). These operations are treated as a single, indivisible unit. This means that either all the operations within a transaction are successfully completed and their changes are permanently saved to the database (**committed**), or none of them are completed, and the database is left in its original state before the transaction began (**rolled** **back**).

**Types of transactions**

* Read transaction
* write transaction

**Read-set of T:** all data items that transaction T reads

**Write-set of T:** all data items that transaction T writes

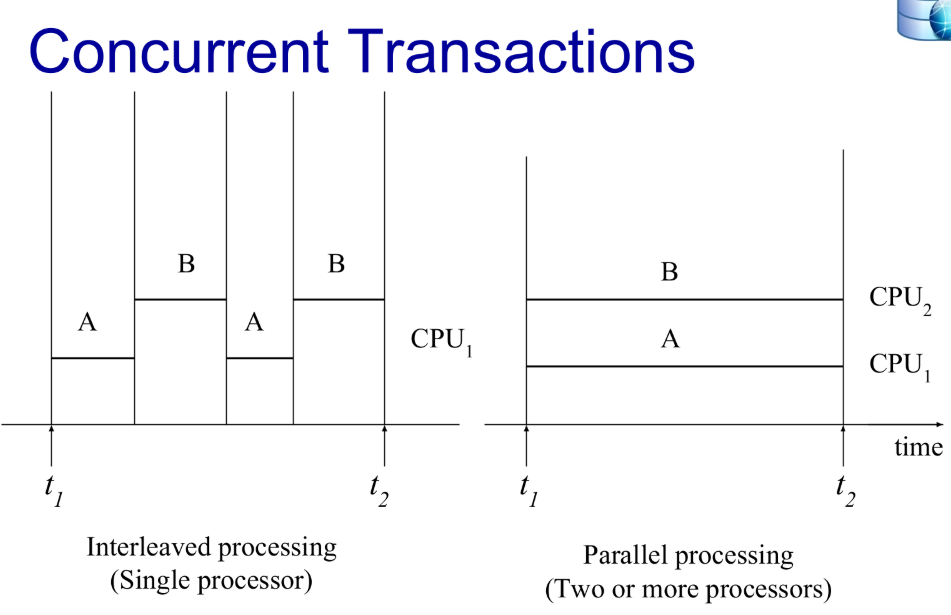
**Concurrent Transactions**

**Inter-leaved Execution:** This is the actual execution of multiple processes (which could be parts of different transactions) on a **single CPU** under **multiprogramming**. The operating system allocates small **time slices** to each process. The CPU rapidly switches between these processes, creating an interleaved pattern of execution.

* **Relevance to Transaction Processing:** When transactions are interleaved, their operations are executed sequentially with breaks. The DBMS's concurrency control mechanisms (like locking) are essential to prevent one transaction from interfering with the intermediate state of another, thus preserving Isolation.
* **Concurrency Problems:** lost update and inconsistent reads

**Parallel processing:** This is the actual simultaneous execution of multiple computations. In a **multi-processor** system, **different CPUs** can work on different parts of the same transaction or different transactions at the exact same time.

* **Relevance to Transaction Processing:** Parallel processing can improve transaction throughput (the number of transactions processed per unit of time) and potentially reduce the response time for individual transactions. However, it also introduces more complex challenges for concurrency control and recovery management to ensure ACID properties are maintained across multiple processing units.



**Reasons for concurrency:**

* Increased throughput
* Reduces average response time

**Key Properties of Transactions: ACID**

To ensure reliability, database transactions must adhere to a set of properties collectively known as **ACID**:

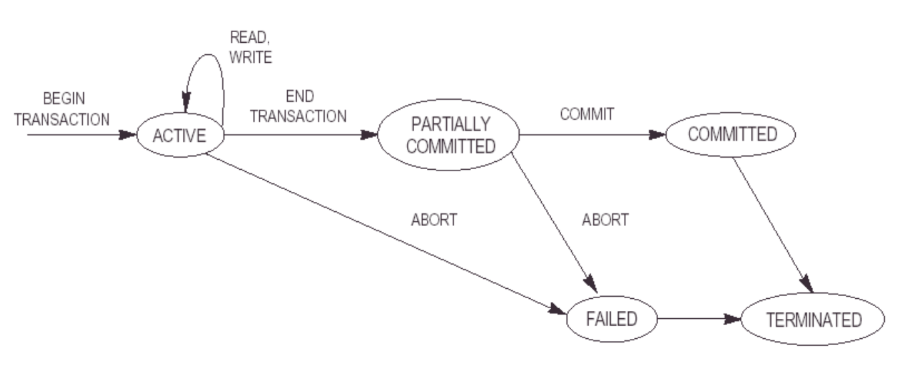
* **Atomicity:** This property ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are completed successfully (**committed**), or none of them are (**rolled back**). There's no in-between state where some operations have succeeded and others haven't. Atomicity is the responsibility of the **Recovery Subsystem** of the DBMS.
  + **Example:** In the money transfer scenario, atomicity guarantees that either both the debit and the credit happen, or neither happens.
* **Consistency:** A transaction must take the database from one consistent/valid state to another consistent/valid state. It must preserve all the database's integrity **constraints, rules, triggers and error handling**. If a transaction violates any of these rules, it must be rolled back, and the database must return to its state before the transaction began. Consistency is the responsibility of **Programmers**.
  + **Example:** If a database has a rule that the balance of a bank account cannot go below zero, a transaction that attempts to debit an account beyond its available balance will be rolled back to maintain consistency.
* **Isolation:** When multiple transactions are executing concurrently, the isolation property ensures that each transaction appears to be executing in complete isolation from all other transactions. The intermediate results of one transaction should not be visible to other concurrently running transactions. This prevents issues like **lost updates, dirty reads, and incorrect summaries**. Isolation is the responsibility of the **Concurrency Control Subsystem** of the DBMS.
  + **Example:** If two transactions are trying to update the same bank account balance concurrently, the isolation property ensures that the final balance is the result of applying both transactions in some serial order, as if they happened one after the other.
* **Durability:** Once a transaction has been committed, the changes made to the database are permanent and will survive even system failures (e.g., power outages, crashes). The database system must have mechanisms in place to ensure that committed transactions are recorded persistently. Durability is the responsibility of the **Recovery Subsystem** of the DBMS.
  + **Example:** Once a money transfer transaction is committed, the debit and credit to the respective accounts are permanently recorded in the database and will not be lost even if the server crashes immediately afterward.

**Transaction States**

A transaction goes through various states during its lifecycle:

* **Active:** The initial state where the transaction begins execution.
* **Partially Committed:** After the final operation of the transaction has been executed.
* **Committed:** The transaction has been successfully completed, and its changes have been permanently recorded in the database.
* **Failed:** The transaction cannot proceed to normal completion due to some error (e.g., constraint violation, system failure).
* **Aborted (Rolled Back):** The transaction has been terminated, and the database has been restored to its state before the transaction began. This happens after a transaction enters the failed state.

**Transition Diagram:**



**Transaction Management**

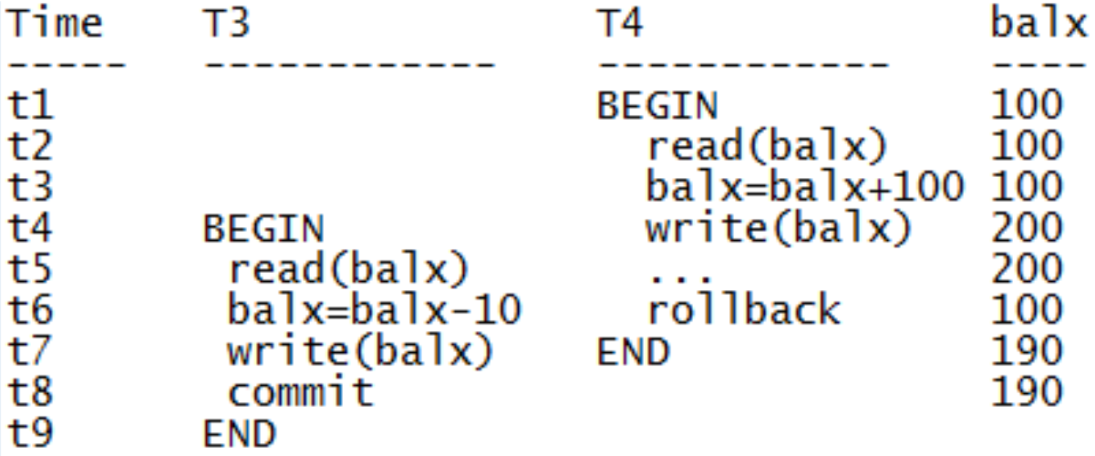
Database Management Systems (DBMS) are responsible for managing transactions and ensuring the ACID properties are maintained. This involves several key functions:

* **Transaction Boundaries:** Defining the start and end of a transaction (usually using commands like BEGIN/END TRANSACTION, COMMIT, ROLLBACK).
* **Concurrency Control Subsystem:** Managing the simultaneous execution of multiple transactions to ensure **isolation** and prevent conflicts. This is typically achieved using techniques like locking, timestamping, and multi-version concurrency control (MVCC). This subsystem ensures database remains in **consistent state** despite concurrent execution of transactions
* **Recovery Subsystem:** Ensuring **durability** by implementing mechanisms like **transaction logs** (recording all changes made by transactions) and **checkpointing** (periodically saving the database state to disk). In case of a system failure, the DBMS uses the transaction logs to redo committed transactions and undo uncommitted ones.

**Concurrency Control Issues (Without Proper Isolation)**

When multiple transactions access and modify the same data concurrently **without proper isolation**, several problems can arise:

* **Dirty Read (Temporary Update):** Transaction 2 reads data that has been modified by Transaction 1, which has not yet been committed. If Transaction 1 is rolled back, Transaction 2 has read inconsistent data.

A screenshot of a graph

AI-generated content may be incorrect.

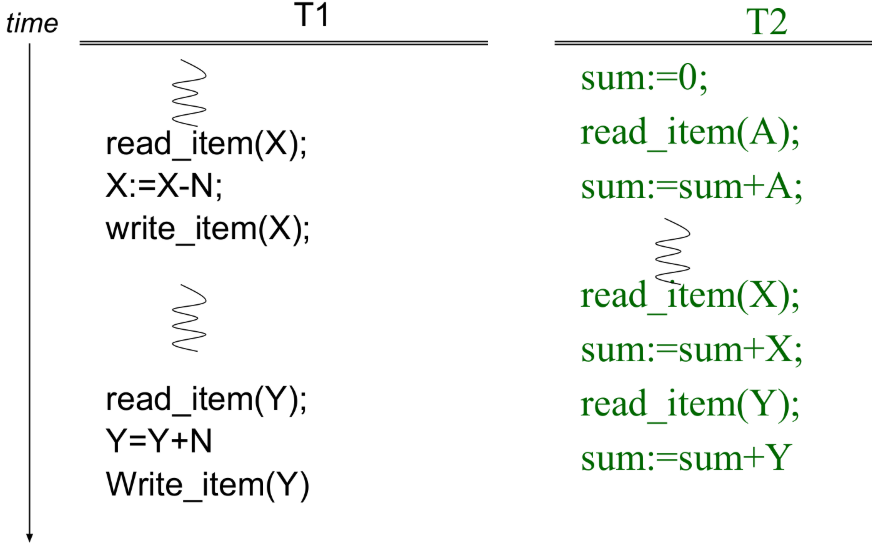
* **Lost Update:** Two transactions read the same data item, both modify it, and then both write it back. The changes made by one transaction are lost because the other transaction overwrites them without seeing the first transaction's changes.

A group of boxes with text

AI-generated content may be incorrect.A table of mathematical equations

AI-generated content may be incorrect.

* **Incorrect Summary:** When one transaction is performing an aggregate function (like SUM, AVG, COUNT) on a number of records while other transactions are concurrently updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated, leading to an incorrect summary or inconsistent results.



**Isolation Levels**

To control the level of isolation between concurrent transactions and to manage the trade-off between consistency and performance, SQL standards define different isolation levels:

* **Read Uncommitted:** The lowest level of isolation. Transactions can read uncommitted changes made by other transactions (leading to dirty reads). Non-repeatable reads and phantom reads can also occur.
* **Read Committed:** Transactions can only read data that has been committed by other transactions (preventing dirty reads). However, incorrect summary can still occur. This is often the **default isolation level** in many DBMS.
* **Repeatable Read:** It guarantees that within a single transaction, any data read by that transaction will not change if the transaction reads it again (preventing non-repeatable reads). However, incorrect summary can still occur. When one transaction reads a row, it acquires a **shared lock** on that row, other transactions are can also acquire shared locks on the same row. When a transaction wants to write to a row, it needs an **exclusive lock**. This lock cannot be acquired if any other transaction holds a shared or exclusive lock on that row. If there is only one transaction that holds the shared lock and it itself needs an exclusive lock then there is no issue.
  + By holding shared locks on the read data until the end of the transaction, Repeatable Read ensures that other transactions cannot modify the data that has been read.
  + This guarantees that within a transaction, if a data item is read multiple times, the transaction will see the same value (unless the transaction itself modifies it).
* **Serializable:** The highest level of isolation. Transactions execute as if they were running in a serial order (one after the other). This level prevents all concurrency issues (dirty reads, non-repeatable reads, phantom reads) but can impact performance due to increased locking.

|  | **CONCURRENCY ISSUES** | | |
| --- | --- | --- | --- |
| **ISOLATION LEVEL** | **Dirty Read** | **Lost Update** | **Incorrect summary** |
| **Read Uncommitted** | Yes | Yes | Yes |
| **Read Committed** | No | Yes | Yes |
| **Repeatable Read** | No | No | Yes |
| **Serializable** | No | No | No |

A computer screen shot of a log

AI-generated content may be incorrect.**Transaction Logs**

Transaction logs are crucial for ensuring **atomicity** and **durability**. They are a history of all the operations performed by transactions in the database, including the changes made to the data. The transaction log typically records:

* Transaction start and end times.
* The type of operation (e.g., INSERT, UPDATE, DELETE).
* The data items affected.
* The before and after values of the data items.

During recovery after a system failure, the DBMS uses the transaction log to:

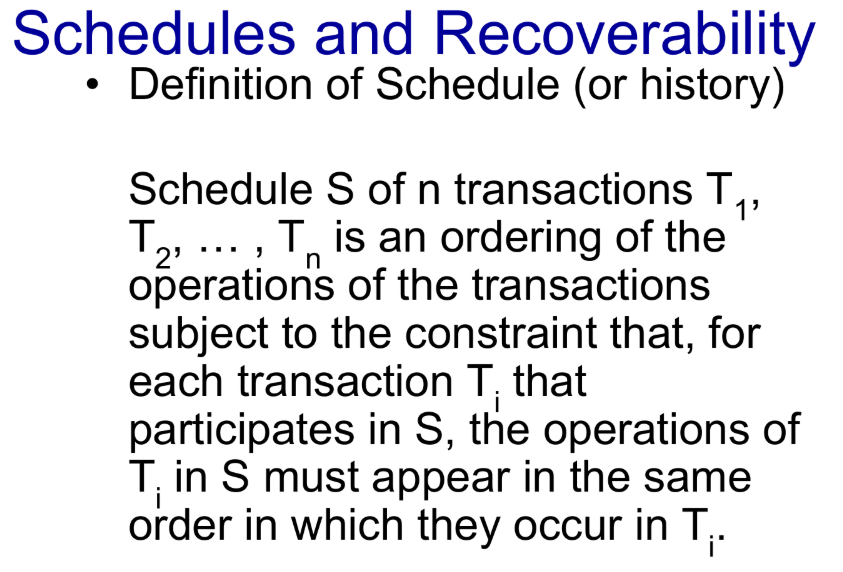
* **Redo (Roll Forward):** Redo works on committed transactions to make sure they are not lost. Redo happens when the system crashes before the data is written to the disk after commit. It reapplies the changes/transactions of all previously committed transactions that might not have been fully written to the disk before the system crash. It ensures **Durability** bypreserving the results of successfully committed transactions, even after a crash.
  + **Example:** After a system restarts/recovered, SQL Server **REDO logs** are checked. Committed changes are **re-applied** to ensure they are present on disk.
* **Undo (Roll Back):** Undo works on transactions that are not committed.Itreverts the changes of all uncommitted or active transactions at the time of the failure, removing partially applied changes which ensures **atomicity**.

**Checkpoints**

Checkpoints are snapshots of the database state at a particular point in time. They help in reducing the amount of work needed during recovery. When a checkpoint occurs, the DBMS typically:

* Writes all modified data from memory to disk.
* Writes information about the current state of transactions to the log file.

After a failure, the DBMS only needs to process the transaction log from the last checkpoint, significantly speeding up the recovery process.



A schedule of a schedule

AI-generated content may be incorrect.

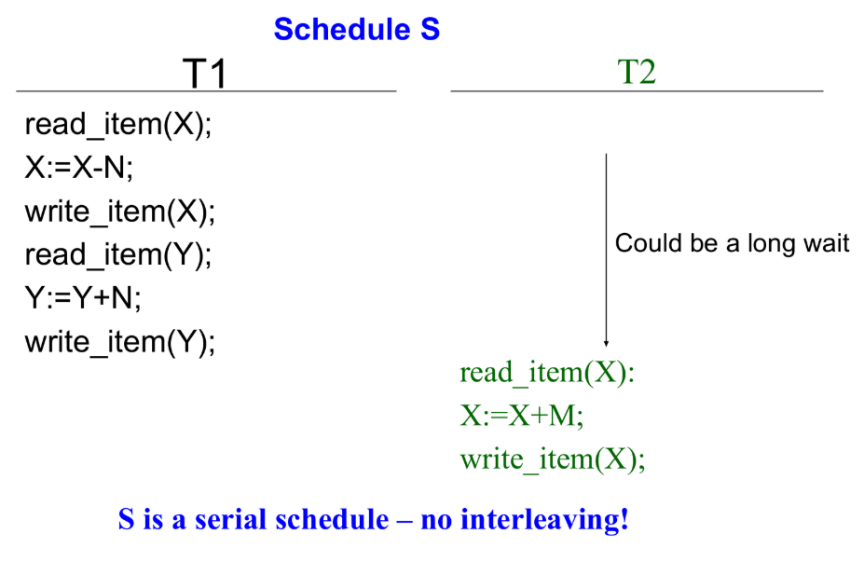
A schedule of transaction with text

AI-generated content may be incorrect.

**Serial Schedule**

A **serial schedule** is one where transactions execute one after another, without any interleaving of their operations. For example, if you have Transaction A and Transaction B, a serial schedule would be:

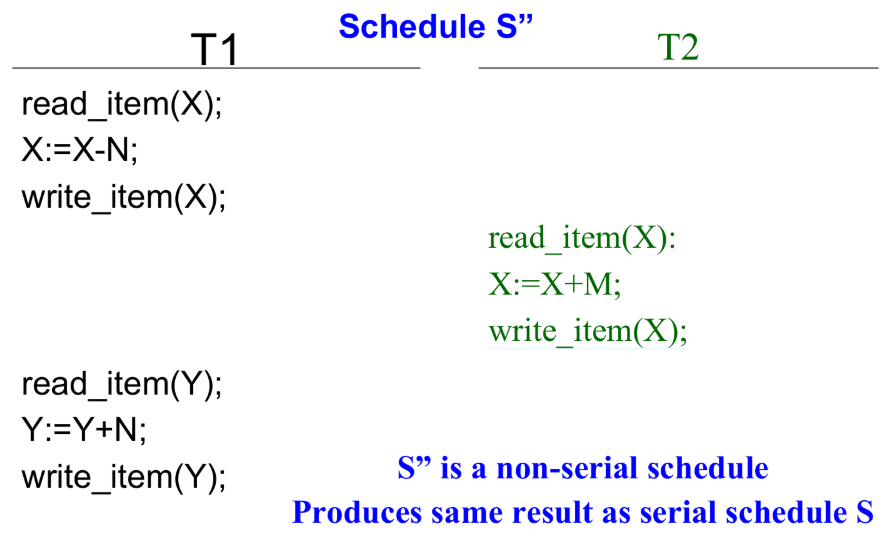
* Schedule 1: All operations of A, then all operations of B.
* Schedule 2: All operations of B, then all operations of A.

A white background with black text and green text

AI-generated content may be incorrect.Serial schedules are inherently correct and maintain consistency because there's no possibility of interference between transactions. But serial schedules are **inefficient** because other transactions have to wait for their turn.

**Serializable Schedule**

A **serializable schedule** is an interleaved schedule (meaning operations from different transactions can be mixed) that produces the same result as some serial schedule. Even though operations are interleaved, the DBMS ensures that the final state of the database is exactly what it would have been if the transactions had run sequentially. We want **increased through put** and **reduced average response time** that is why we try use concurrency and with serializable schedules**.**



**Serializability**

It ensures the execution of concurrent transactions is consistent and results in the same outcome as if transactions were executed serially.

**1. Conflict Serializability**

* It occurs when two or more transactions try to read and write same data item. A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
* **Conflicting operations** are operations from different transactions that access the same data item and at least one of them is a WRITE operation.
* READ-WRITE conflict (Non-Repeatable Read)
* WRITE-READ conflict (dirty read)
* WRITE-WRITE conflict (lost update)
* **Assumption:** Every serial schedule is correct
* **Goal:** Find non-serial schedules which are also correct
* If there exists a **loop/cycle** in a precedence graph, then there is no conflict serializability. If there is **no cycle**, use topological sort to obtain serializable schedule (linear order consistent with precedence order of graph). Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering *The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).*

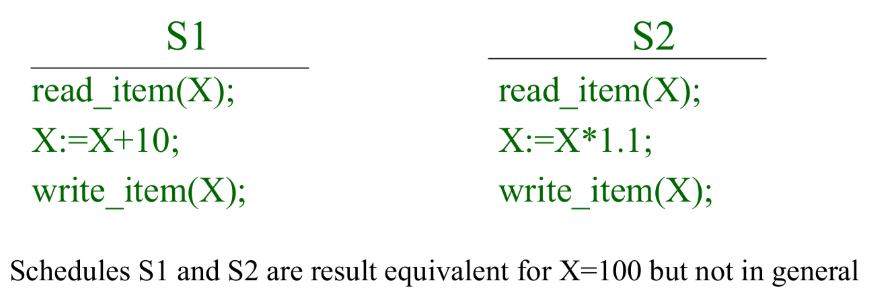
A screenshot of a white text

AI-generated content may be incorrect.

A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.

**When are two schedules equivalent?**

* **Result Equivalent Schedules**: They lead to same result. Two schedules are result equivalent if they produce the same final state of the database.



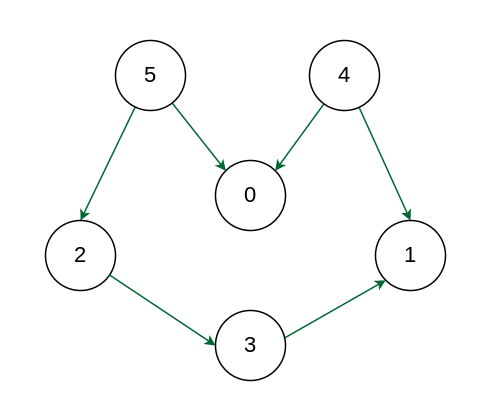
* **Conflict Equivalent Schedules**: Two schedules are conflict equivalent, if the order of any two conflicting operations is the same in both schedules.

Schedule S is conflict serializable if it is **conflict equivalent** to some serial schedule S. We can reorder the **non-conflicting operations** to improve efficiency.

**Non-conflicting operations:**

* Reads and writes from same transaction
* Reads from different transactions
* Reads and writes from different transactions on different data items

**Precedence Graph**



***Output:*** *A topological sorting of the following graph is "****5 4 2 3 1 0".*** *There can be more than one topological sorting for a graph. Another topological sorting of the following graph is* ***"4 5 2 3 1 0".***

If a schedule is **conflict-serializable**, its equivalent serial schedule(s) can be found by a topological sort of its precedence graph.

**2. View Serializability**

* A schedule is view serializable if it produces the same final state of the database as some serial schedule, even if the intermediate reads might differ.
* It's a more relaxed definition than conflict serializability. **All conflict-serializable schedules are view-serializable, but not all view-serializable schedules are conflict-serializable.**
* View serializability is harder to implement and detect than conflict serializability.

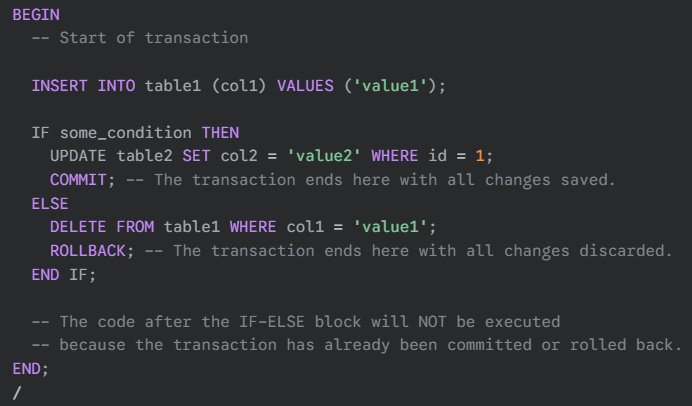
**Transaction Control Language (TCL)**

| **Statement** | **Purpose** |
| --- | --- |
| BEGIN TRAN | Start a transaction |
| COMMIT | Save all changes made by the transaction |
| ROLLBACK | Undo all changes made by the transaction |
| SAVEPOINT | Create a point within a transaction to roll back to |

**COMMIT:** The COMMIT statement signals the successful completion of a transaction. It makes all changes made by the transaction permanent in the database. Once a transaction is committed, the changes are saved and visible to other transactions (subject to isolation levels).

**ROLLBACK:** The ROLLBACK statement undoes all changes made by the current transaction. It restores the database to the state it was in before the transaction began. Any modifications made within the transaction are discarded.

**A transaction has a single outcome: either it is fully committed, or it is fully rolled back.** You **cannot** perform both a COMMIT and a ROLLBACK within a single, logical transaction in a way that both actions have a permanent effect on the database for that transaction.



**-- Problems –**

**Question:**

| **Time** | **Transaction (REPEATABLE READ)** | **Operation** | **Notes** |
| --- | --- | --- | --- |
| 1 | T1 | READ(x = 100) | Reads original value |
| 2 | T2 | READ(x = 100) | Also reads same original value |
| 3 | T1 | x = 100 + 50 | Calculates 150 in memory |
| 4 | T2 | x = 100 + 20 | Calculates 120 in memory |
| 5 | T1 | WRITE(x = 150) | Tries to write — **gets blocked** |
| 6 | T2 | WRITE(x = 120) | Tries to write — **gets blocked** |

This scenario is just theoretical, and this doesn’t happen in reality because programmers prevent concurrent writes manually.

**Problem Analysis:**

Both transactions **read the same value**, compute their own updates, and then **both try to write back**. This is a classic **lost update** situation, because one of the writes will overwrite the other without seeing the updated value.

BUT here's the **critical twist**:

**🔒 Under REPEATABLE READ in SQL Server:**

* A **shared lock** is held on x for the entire duration of T1 and T2.
* When **T1 reads x**, it locks it **shared (S) mode**.
* When **T1 tries to write x**, it **requests an exclusive (X) lock**, which **conflicts** with T2’s shared lock (still active).
* T2 also does the same.
* Now both transactions **wait on each other** = **Deadlock or blocking**.

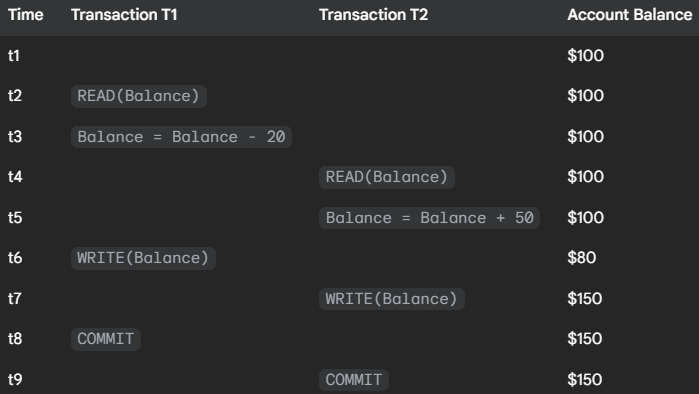
**🧠 What Actually Happens:**

* **T1 and T2 will both be blocked** when they try to upgrade to **exclusive locks** to perform the write.
* SQL Server detects this as a **deadlock**, and **one transaction is rolled back** automatically.
* The **other proceeds and completes its write**.

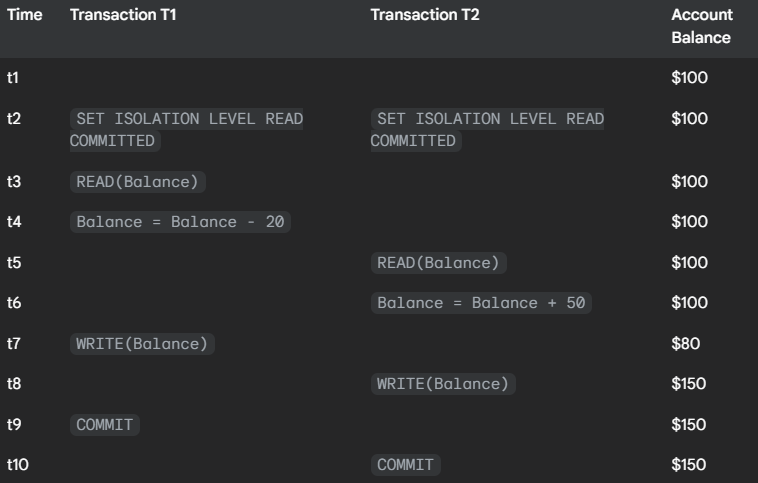
**Question:**

The **lost update anomaly** is said to occur if a transaction Tj reads a data item, then another transaction Tk writes the data item (possibly based on a previous read), after which Tj writes the data item. The update performed by Tk has been lost, since the update done by Tj ignored the value written by Tk .

* Give an example of a schedule showing the lost update anomaly.



* Give an example schedule to show that the lost update anomaly is possible with the read committed isolation level.



* Explain why the lost update anomaly is not possible with the repeated read isolation level.

Under **Repeatable Read Isolation Level**:

* Once a transaction **reads** a data item, **no other transaction can write to it** until the first transaction is done.
* It ensures **read consistency** and **prevents concurrent writes** on read items.

**👨‍🔧 Mechanism:**

* A **shared lock** is held on read data **until the transaction ends**.
* No other transaction can **write** that data item until the lock is released.

A screenshot of a computer

AI-generated content may be incorrect.

**Question:**

Consider the following two transactions:

**T13:**

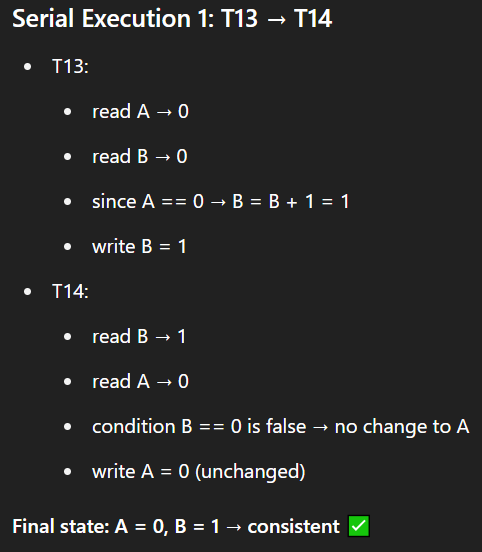
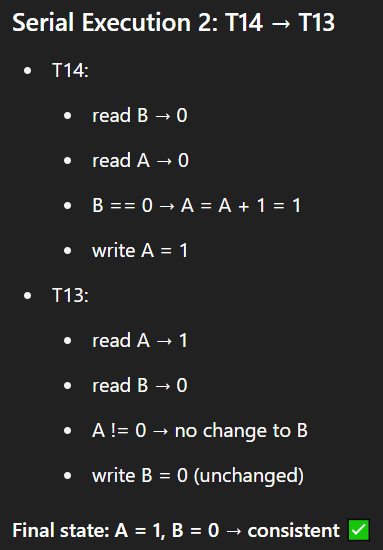
* read(A);
* read(B);
* if A = 0 then B := B + 1;
* write(B).

**T14:**

* read(B);
* read(A);
* if B = 0 then A := A + 1;
* write(A).

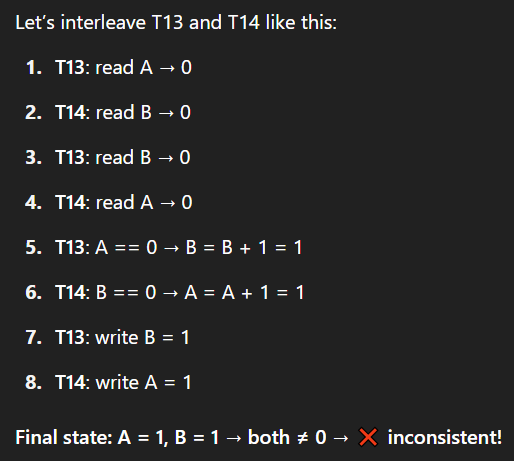
Let the consistency requirement be A = 0 ∨/OR B = 0, with A = B = 0 the initial values.

1. Show that every serial execution involving these two transactions preserves the consistency of the database.



1. Show a concurrent execution of T13 and T14 that produces a non-serializable schedule.



This interleaved execution results in a state **not possible in any serial execution** → so it is **non-serializable**.

1. Is there a concurrent execution of T13 and T14 that produces a serializable schedule.

**Question:**

For each of the following isolation levels, give an example of a schedule that respects the specified level of isolation, but is not serializable:

a. Read uncommitted

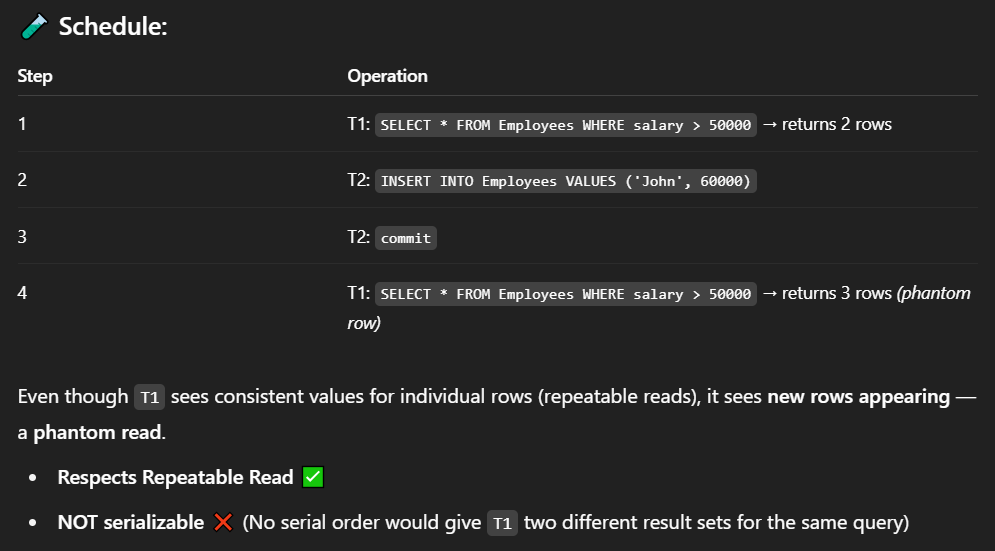


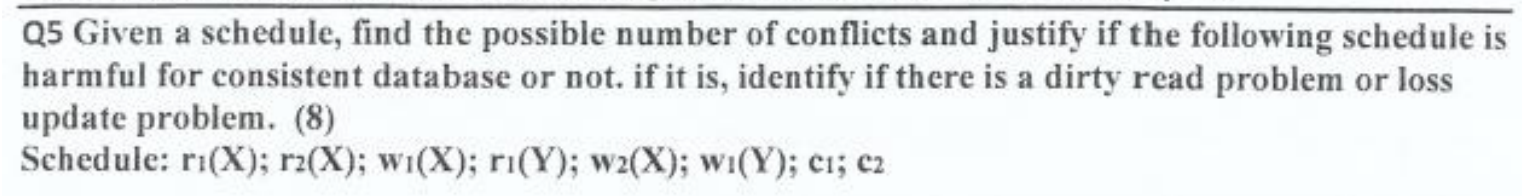
b. Read committed

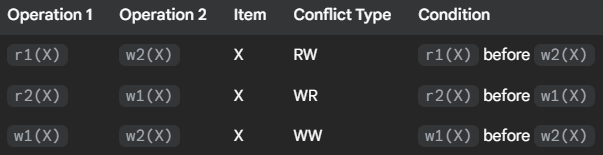
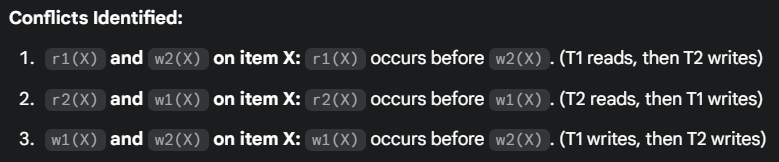
A screenshot of a computer

AI-generated content may be incorrect.

c. Repeatable read

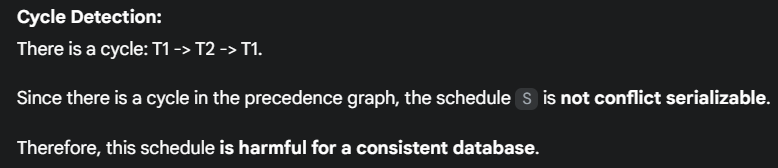
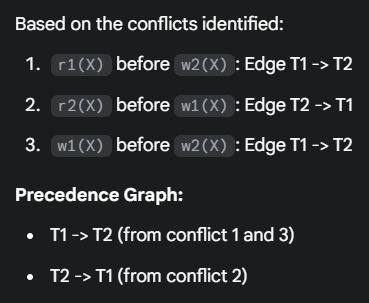






A black screen with white text

AI-generated content may be incorrect.



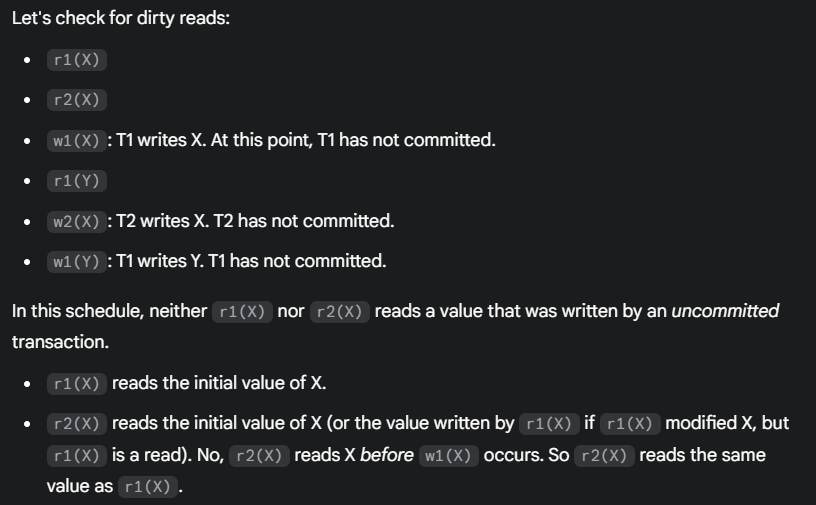
A black background with white text

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.