ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability.

These properties are essential in the context of database management systems to ensure the reliability and consistency of data.

Here's a brief overview of each of the ACID properties:

1. **Atomicity:**
   * Atomicity ensures that a transaction is treated as a single, indivisible unit of work. It means that either all the operations within a transaction are completed successfully, or none of them are.
   * If any part of a transaction fails (e.g., due to an error or a system crash), the entire transaction is rolled back to its previous state to maintain data integrity.
2. **Consistency:**
   * Consistency ensures that a transaction brings the database from one consistent state to another. In other words, a transaction should ensure that the database follows a set of predefined rules and constraints.
   * If a transaction violates any of these rules or constraints, it is rolled back, and the database remains unchanged.
3. **Isolation:**
   * Isolation guarantees that multiple transactions can be executed concurrently without interfering with each other. Each transaction appears to run in isolation from other transactions.
   * Isolation is essential to prevent issues such as dirty reads, non-repeatable reads, and phantom reads, which can occur when multiple transactions access and modify the same data simultaneously.
4. **Durability:**
   * Durability ensures that once a transaction is committed and acknowledged as successful, its changes to the database will persist, even in the event of system crashes or failures.
   * Databases use various mechanisms such as write-ahead logging and periodic checkpoints to ensure that committed data is stored permanently and can be recovered in the face of failures.

These ACID properties collectively provide a strong guarantee of data reliability and consistency in database systems. However, it's important to note that enforcing strict ACID properties can sometimes introduce performance overhead, especially in highly concurrent systems. In some cases, database systems may relax these properties to improve performance, depending on the specific requirements of the application. These relaxed systems are often referred to as "NoSQL" databases, and they may prioritize certain aspects of data management, such as scalability and flexibility, over strict ACID compliance.

**Atomicity**

Atomicity is one of the four principles of the ACID properties in database systems.

It ensures that a transaction is treated as a single, indivisible unit, meaning either all operations in the transaction are completed successfully, or none of them are applied.

This prevents partial updates to the database that could lead to data inconsistency.

Here are some examples of tables and data scenarios that explain **Atomicity**:

**Example 1: Bank Transaction Tables**

**Tables:**

1. **Accounts Table**

| **Account\_ID** | **Account\_Holder** | **Balance** |
| --- | --- | --- |
| 101 | Alice | 5000 |
| 102 | Bob | 3000 |

**Scenario:**

A transaction to transfer $1000 from Alice's account to Bob's account.

**Transaction Steps:**

1. **Step 1**: Deduct $1000 from Alice's account.

UPDATE Accounts

SET Balance = Balance - 1000

WHERE Account\_ID = 101;

1. **Step 2**: Add $1000 to Bob's account.

UPDATE Accounts

SET Balance = Balance + 1000

WHERE Account\_ID = 102;

**Atomicity Behavior:**

* If **both steps succeed**, the transaction is committed, and the balances are updated as:
  + Alice: $4000
  + Bob: $4000
* If **Step 1 succeeds but Step 2 fails**, the transaction is rolled back, and both accounts retain their original balances:
  + Alice: $5000
  + Bob: $3000

**Example 2: E-commerce Order Processing**

**Tables:**

1. **Orders Table**

| **Order\_ID** | **Customer\_ID** | **Status** |
| --- | --- | --- |
| 1 | 201 | Pending |
| 2 | 202 | Pending |

1. **Inventory Table**

| **Product\_ID** | **Product\_Name** | **Stock** |
| --- | --- | --- |
| 301 | Laptop | 5 |
| 302 | Smartphone | 10 |

**Scenario:**

A customer places an order for a Laptop.

**Transaction Steps:**

1. **Step 1**: Add the order to the Orders table with Status = 'Pending'.

INSERT INTO Orders (Order\_ID, Customer\_ID, Status)

VALUES (3, 203, 'Pending');

1. **Step 2**: Deduct 1 unit from the stock of the Laptop in the Inventory table.

UPDATE Inventory

SET Stock = Stock - 1

WHERE Product\_ID = 301;

**Atomicity Behavior:**

* If **both steps succeed**, the order is placed, and the stock is updated:
  + Orders:

| **Order\_ID** | **Customer\_ID** | **Status** |
| --- | --- | --- |
| 3 | 203 | Pending |

* + Inventory:

| **Product\_ID** | **Stock** |
| --- | --- |
| 301 | 4 |

* If **Step 2 fails** (e.g., due to a stock shortage), the transaction is rolled back, and no changes are made:
  + Orders:
    - No new order is added.
  + Inventory:
    - Stock remains unchanged.

**Example 3: Student Registration**

**Tables:**

1. **Students Table**

| **Student\_ID** | **Name** | **Courses\_Registered** |
| --- | --- | --- |
| 1 | John | 2 |
| 2 | Sarah | 1 |

1. **Courses Table**

| **Course\_ID** | **Course\_Name** | **Available\_Seats** |
| --- | --- | --- |
| 101 | Math | 5 |
| 102 | Science | 10 |

**Scenario:**

A student (John) registers for the Math course.

**Transaction Steps:**

1. **Step 1**: Increment the Courses\_Registered for the student in the Students table.

UPDATE Students

SET Courses\_Registered = Courses\_Registered + 1

WHERE Student\_ID = 1;

1. **Step 2**: Deduct 1 from the Available\_Seats in the Courses table.

UPDATE Courses

SET Available\_Seats = Available\_Seats - 1

WHERE Course\_ID = 101;

**Atomicity Behavior:**

* If **both steps succeed**, the student's registration is completed, and the course seats are updated:
  + Students:

| **Student\_ID** | **Courses\_Registered** |
| --- | --- |
| 1 | 3 |

* + Courses:

| **Course\_ID** | **Available\_Seats** |
| --- | --- |
| 101 | 4 |

* If **Step 1 succeeds but Step 2 fails** (e.g., due to no available seats), the transaction is rolled back, and no changes are made:
  + Students:

| **Student\_ID** | **Courses\_Registered** |
| --- | --- |
| 1 | 2 |

* + Courses:

| **Course\_ID** | **Available\_Seats** |
| --- | --- |
| 101 | 5 |

**Key Takeaways:**

* Atomicity ensures that all steps in a transaction are treated as a single unit.
* If any part of the transaction fails, the entire transaction is rolled back to maintain database consistency.
* This behavior is particularly critical in real-world scenarios like banking, e-commerce, and registration systems to prevent partial or inconsistent updates.

**Consistency:**

**Consistency** is one of the ACID properties in database systems.

It ensures that a transaction brings the database from one valid state to another, maintaining all defined rules, constraints, and relationships.

Consistency guarantees that the data remains correct and adheres to the database schema and business logic, regardless of the outcome of the transaction.

**Key Points About Consistency:**

1. **Maintains Rules**: It enforces database constraints such as primary keys, foreign keys, and unique constraints.
2. **Preserves Integrity**: Ensures compliance with triggers, stored procedures, and user-defined business rules.
3. **Validation on Commit**: The database validates the changes made during a transaction before committing them.

**Examples of Consistency:**

**Example 1: Bank Transaction**

**Tables:**

1. **Accounts Table**

| **Account\_ID** | **Account\_Holder** | **Balance** |
| --- | --- | --- |
| 101 | Alice | 5000 |
| 102 | Bob | 3000 |

**Scenario:**

A transaction transfers $1000 from Alice to Bob.

1. Deduct $1000 from Alice's account:

UPDATE Accounts

SET Balance = Balance - 1000

WHERE Account\_ID = 101;

1. Add $1000 to Bob's account:

UPDATE Accounts

SET Balance = Balance + 1000

WHERE Account\_ID = 102;

**Consistency Rule:**

The total balance of both accounts before and after the transaction must remain the same (invariant).

* **Before Transaction**: Alice ($5000) + Bob ($3000) = $8000
* **After Transaction**: Alice ($4000) + Bob ($4000) = $8000

If any error (e.g., step 2 fails) occurs, the transaction is rolled back, preserving the consistency of the total balance.

**Example 2: Referential Integrity**

**Tables:**

1. **Orders Table**

| **Order\_ID** | **Customer\_ID** | **Order\_Amount** |
| --- | --- | --- |
| 1 | 201 | 500 |
| 2 | 202 | 300 |

1. **Customers Table**

| **Customer\_ID** | **Customer\_Name** |
| --- | --- |
| 201 | Alice |
| 202 | Bob |

**Scenario:**

A new order is placed for a customer with ID 203.

1. Add a new order to the Orders table:

INSERT INTO Orders (Order\_ID, Customer\_ID, Order\_Amount)

VALUES (3, 203, 700);

**Consistency Rule:**

The Customer\_ID in the Orders table must exist in the Customers table (foreign key constraint).

* If 203 is not in the Customers table, the database will reject the transaction, maintaining consistency.

**Example 3: Stock Management**

**Tables:**

1. **Products Table**

| **Product\_ID** | **Product\_Name** | **Stock** |
| --- | --- | --- |
| 301 | Laptop | 5 |
| 302 | Smartphone | 10 |

1. **Sales Table**

| **Sale\_ID** | **Product\_ID** | **Quantity** |
| --- | --- | --- |
| 1 | 301 | 2 |

**Scenario:**

A sale is made for 6 units of the Laptop.

1. Add the sale to the Sales table:

INSERT INTO Sales (Sale\_ID, Product\_ID, Quantity)

VALUES (2, 301, 6);

1. Deduct the sold quantity from the stock in the Products table:

UPDATE Products

SET Stock = Stock - 6

WHERE Product\_ID = 301;

**Consistency Rule:**

The Stock cannot go below zero (business rule). If the sale quantity exceeds the available stock, the transaction will be rolled back to maintain consistency.

**Example 4: Salary Constraints**

**Tables:**

1. **Employees Table**

| **Employee\_ID** | **Name** | **Salary** |
| --- | --- | --- |
| 101 | Alice | 5000 |
| 102 | Bob | 3000 |

**Scenario:**

An employee's salary is updated.

1. Update Bob's salary:

UPDATE Employees

SET Salary = 10000

WHERE Employee\_ID = 102;

**Consistency Rule:**

Business logic may enforce that no employee's salary can exceed $8000.

* If the updated salary exceeds this limit, the transaction fails, preserving the consistency of the data.

**Importance of Consistency:**

Consistency ensures:

* **Data Integrity**: Prevents the database from ending up in an invalid or corrupt state.
* **Validation of Rules**: Enforces database constraints and business logic.
* **Error Recovery**: Rolls back invalid transactions, maintaining correct data.

**Key Takeaway:**

Consistency works hand in hand with atomicity and ensures that only valid data is committed to the database. If any part of a transaction violates the defined rules, the entire transaction is rolled back.

**Isolation**

Isolation is one of the ACID properties of database transactions.

It ensures that multiple transactions occurring simultaneously do not interfere with each other.

Each transaction is executed as if it is the only one in the system, even though other transactions might be running concurrently.

This guarantees data consistency and correctness, regardless of the execution order of transactions.

**Key Concepts of Isolation:**

1. **Concurrent Transactions**: Transactions execute in parallel but do not impact each other's results.
2. **No Dirty Reads**: A transaction cannot see intermediate changes made by other uncommitted transactions.
3. **Database Isolation Levels**: Control how strictly isolation is enforced (e.g., Read Uncommitted, Read Committed, Repeatable Read, Serializable).

**Examples of Isolation**

**Example 1: Bank Accounts**

**Tables:**

**Accounts Table**

| **Account\_ID** | **Account\_Holder** | **Balance** |
| --- | --- | --- |
| 101 | Alice | 5000 |
| 102 | Bob | 3000 |

**Scenario:**

* **Transaction 1 (T1)**: Alice transfers $1000 to Bob.
* **Transaction 2 (T2)**: Bob withdraws $500.

**Steps:**

1. **T1 begins**: Deduct $1000 from Alice's account.

UPDATE Accounts

SET Balance = Balance - 1000

WHERE Account\_ID = 101;

1. Before T1 completes, **T2 begins**: Reads Bob's balance.

SELECT Balance

FROM Accounts

WHERE Account\_ID = 102;

1. **Isolation ensures**:
   * **T2** cannot see the intermediate state of Bob's balance from T1.
   * T2 will only see Bob's balance as $3000 (before the effect of T1 is committed).

**Example 2: Online Shopping Cart**

**Tables:**

**Inventory Table**

| **Product\_ID** | **Product\_Name** | **Stock** |
| --- | --- | --- |
| 201 | Laptop | 5 |
| 202 | Phone | 10 |

**Scenario:**

* **Transaction 1 (T1)**: Customer A buys 2 laptops.
* **Transaction 2 (T2)**: Customer B tries to buy 4 laptops.

**Steps:**

1. **T1 begins**: Reads current stock (5 units) and deducts 2.

UPDATE Inventory

SET Stock = Stock - 2

WHERE Product\_ID = 201;

1. Before T1 completes, **T2 begins**: Reads stock.

SELECT Stock

FROM Inventory

WHERE Product\_ID = 201;

1. **Isolation ensures**:
   * **T2** does not see the reduced stock (3 units) until T1 commits.
   * If T1 fails or rolls back, T2 will see the original stock (5 units).

**Example 3: Flight Booking**

**Tables:**

**Flights Table**

| **Flight\_ID** | **Seats\_Available** |
| --- | --- |
| F101 | 3 |
| F102 | 5 |

**Scenario:**

* **Transaction 1 (T1)**: Customer A books 2 seats on Flight F101.
* **Transaction 2 (T2)**: Customer B books 2 seats on Flight F101.

**Steps:**

1. **T1 begins**: Reads available seats (3) and deducts 2.

UPDATE Flights

SET Seats\_Available = Seats\_Available - 2

WHERE Flight\_ID = 'F101';

1. Before T1 commits, **T2 begins**: Reads available seats.

SELECT Seats\_Available

FROM Flights

WHERE Flight\_ID = 'F101';

1. **Isolation ensures**:
   * **T2** will not see the reduced seat count (1 seat) until T1 commits.
   * If T1 rolls back, **T2** will see the original seat count (3 seats).

**Example 4: Library System**

**Tables:**

**Books Table**

| **Book\_ID** | **Title** | **Quantity** |
| --- | --- | --- |
| 101 | Database Systems | 3 |
| 102 | Programming Basics | 5 |

**Scenario:**

* **Transaction 1 (T1)**: A student borrows 1 copy of "Database Systems."
* **Transaction 2 (T2)**: Another student tries to borrow 2 copies of "Database Systems."

**Steps:**

1. **T1 begins**: Reads available quantity (3) and deducts 1.

UPDATE Books

SET Quantity = Quantity - 1

WHERE Book\_ID = 101;

1. Before T1 completes, **T2 begins**: Reads quantity.

SELECT Quantity

FROM Books

WHERE Book\_ID = 101;

1. **Isolation ensures**:
   * **T2** does not see the reduced quantity (2) until T1 commits.
   * If T1 fails or rolls back, **T2** will see the original quantity (3).

**Database Isolation Levels**

1. **Read Uncommitted**:
   * Allows dirty reads (transactions can see uncommitted changes).
   * Least strict isolation.
2. **Read Committed**:
   * Prevents dirty reads (transactions only see committed changes).
   * Default in many databases.
3. **Repeatable Read**:
   * Prevents dirty and non-repeatable reads (data read by a transaction cannot change during its execution).
4. **Serializable**:
   * Highest level of isolation.
   * Ensures transactions are executed sequentially, avoiding all concurrency issues.

**Key Takeaway:**

Isolation ensures that concurrent transactions do not interfere with each other and maintain data consistency. The strictness of isolation depends on the chosen **isolation level**, which balances performance and data integrity.

**Durability**

Durability is the property of a database system that ensures once a transaction is committed, its changes are permanently stored, even in the event of a system crash, power failure, or other issues.

This guarantees that the database will retain the committed data and ensures reliability for users and applications relying on the system.

**How Durability is Achieved**

* **Transaction Logs**: Changes are recorded in logs before being committed, allowing recovery in case of failures.
* **Write-Ahead Logging (WAL)**: Ensures changes are written to persistent storage before confirming a transaction.
* **Backups and Replication**: Regular backups and replication protect against data loss.
* **Crash Recovery Mechanisms**: Databases use recovery protocols to ensure committed transactions are restored after a crash.

**Examples of Durability**

**Example 1: Bank Transactions**

**Scenario**: A user transfers $500 from their account to another account.

1. **Initial State**:
   * User A's balance: $1000
   * User B's balance: $2000
2. **Transaction**:
   * Deduct $500 from User A’s account.
   * Add $500 to User B’s account.
   * Commit the transaction.
3. **Durability Ensures**:
   * Even if the system crashes after the transaction is committed, the changes ($500 deducted from User A, added to User B) will persist after recovery.

**Example 2: E-commerce Order Placement**

**Scenario**: A customer places an order online.

1. **Steps**:
   * Order is created and marked as "pending."
   * Payment is processed successfully.
   * Inventory is updated, and the order is marked as "confirmed."
   * Commit the transaction.
2. **Durability Ensures**:
   * If the system crashes after committing, the order remains confirmed, payment is processed, and inventory changes persist.
   * The customer will not lose their order details or payment information.

**Example 3: Flight Booking System**

**Scenario**: A customer books a flight ticket.

1. **Steps**:
   * Deduct the ticket price from the customer's balance.
   * Mark the seat as "booked" in the system.
   * Commit the transaction.
2. **Durability Ensures**:
   * Even if the system crashes after committing, the seat remains booked and the ticket price remains deducted when the system is restored.

**Example 4: Library System**

**Scenario**: A student borrows a book.

1. **Steps**:
   * Mark the book as "borrowed."
   * Record the due date in the database.
   * Commit the transaction.
2. **Durability Ensures**:
   * If the system crashes after committing, the book remains marked as borrowed, and the due date remains recorded upon recovery.

**Example 5: Online Shopping Cart**

**Scenario**: A user checks out their cart and completes a purchase.

1. **Steps**:
   * Payment is processed.
   * Inventory is updated.
   * Order is created and marked as "completed."
   * Commit the transaction.
2. **Durability Ensures**:
   * If the system crashes after committing, the order status remains "completed," and inventory changes persist after recovery.

**Key Takeaway**

Durability ensures the reliability and trustworthiness of a database system by safeguarding committed transactions, even in the event of unexpected failures. It is critical for systems where data integrity and persistence are non-negotiable, such as banking, e-commerce, and flight booking systems.