**What are ACID Transactions in Databases?**

Imagine you’re running an e-commerce application.

A customer places an order, and your system needs to deduct the item from inventory, charge the customer’s credit card, and record the sale in your accounting system—all at once.

What happens if the payment fails but your inventory count has already been reduced? Or if your application crashes halfway through the process?

This is where **ACID transactions** come into play. They ensure that all the steps in such critical operations happen reliably and consistently.

ACID is an acronym that refers to the set of 4 key properties that define transaction: **Atomicity** (পারমাণবিক শক্তি)**, Consistency**(ধারাবাহিকতা), **Isolation**(আলাদা করা), and **Durability**(স্থায়িত্ব).

[[A group of colorful signs with letters

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!zpL2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fa9ceb65c-70e6-4f3e-9511-f6bc5da93d13_1308x1086.png)](https://substackcdn.com/image/fetch/$s_!zpL2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fa9ceb65c-70e6-4f3e-9511-f6bc5da93d13_1308x1086.png" \t "_blank)

In this article, we’ll dive into what each of the ACID properties mean, why they are important, and how they are implemented in databases.

**What is a Database Transaction?**

A **transaction** in the context of databases is a sequence of one or more operations (such as inserting, updating, or deleting records) that the database treats as **one single action**. It either fully succeeds or fully fails, with no in-between states.

**Example: Bank Transfer**

When you send money to a friend, two things happen:

1. Money is deducted from your account.
2. Money is added to their account.

These two steps form **one transaction**. If either step fails, both are canceled.

Without transactions, databases could end up in inconsistent states.

For example:

* **Partial updates**: Your money is deducted, but your friend never receives it.
* **Conflicts**: Two people booking the last movie ticket at the same time.

Transactions solve these problems by enforcing rules like **ACID properties** (Atomicity, Consistency, Isolation, Durability).

**1. Atomicity**

Atomicity ensures that a transaction—comprising multiple operations—executes as a **single and indivisible**unit of work: it either **fully** succeeds (commits) or **fully** fails (rolls back).

If any part of the transaction fails, the entire transaction is rolled back, and the database is restored to a state exactly as it was before the transaction began.

**Example:**In a money transfer transaction, if the credit step fails, the debit step cannot be allowed to stand on its own. This prevents inconsistent states like “money disappearing” from one account without showing up in another.

Atomicity abstracts away the complexity of manually undoing changes if something goes wrong.

**How Databases Implement Atomicity**

Databases use two key mechanisms to guarantee atomicity.

**1. Transaction Logs (Write-Ahead Logs)**

* Every operation is recorded in a **write-ahead log** before it’s applied to the actual database table.
* If a failure occurs, the database uses this log to **undo** incomplete changes.

[[A computer screen with text and numbers

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!q5CP!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F7182af1e-62ac-4883-9434-6082cabe4328_478x312.png)](https://substackcdn.com/image/fetch/$s_!q5CP!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F7182af1e-62ac-4883-9434-6082cabe4328_478x312.png" \t "_blank)

Once the WAL entry is safely on disk, the database proceeds with modifying the in-memory pages that contain rows for **Account A** and **Account B**.

When the operations succeed:

1. The database marks **Transaction ID 12345** as **committed** in the transaction log.
2. The newly updated balances for A and B will eventually get flushed from memory to their respective data files on disk.

If the database crashes **after** the log entry is written but **before** the data files are fully updated, the WAL provides a way to recover:

* On restart, the database checks the WAL.
* It sees **Transaction 12345** was committed.
* It reapplies the **UPDATE** operations to ensure the final balances are correct in the data files.

If the transaction had not committed (or was marked as “in progress”) at the time of the crash, the database would **roll back** those changes using information in the log, leaving the table as if the transaction never happened.

**2. Commit/Rollback Protocols**

* Databases provide commands like BEGIN TRANSACTION, COMMIT, and ROLLBACK
* Any changes made between BEGIN TRANSACTION and COMMIT are considered “in-progress” and won’t be permanently applied unless the transaction commits successfully.
* If any step fails, or if you explicitly issue a ROLLBACK, all changes since the start of the transaction are undone.

[[A computer screen shot of a program code

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!5HO2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F12443c03-6f81-4eb4-a913-0c5eaddebbbb_622x439.png)](https://substackcdn.com/image/fetch/$s_!5HO2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F12443c03-6f81-4eb4-a913-0c5eaddebbbb_622x439.png" \t "_blank)

**2. Consistency**

**Consistency** in the context of ACID transactions ensures that any transaction will bring the database from one valid state to another valid state—never leaving it in a broken or “invalid” state.

It means that all the data integrity constraints, such as **primary key** **constraints** (no duplicate IDs), **foreign key** **constraints** (related records must exist in parent tables), and **check constraints**(age can’t be negative), are satisfied before and after the transaction.

If a transaction tries to violate these rules, it will not be committed, and the database will revert to its previous state.

**Example:**

You have two tables in an e-commerce database:

1. products (with columns: product\_id, stock\_quantity, etc.)
2. orders (with columns: order\_id, product\_id, quantity, etc.)

* **Constraint**: You can’t place an order for a product if quantity is greater than the stock\_quantity in the products table.

**Transaction Flow**

[[A computer screen with text

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!rEA_!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F43f22710-56e1-4bbb-af2d-d4808353edbc_642x378.png)](https://substackcdn.com/image/fetch/$s_!rEA_!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F43f22710-56e1-4bbb-af2d-d4808353edbc_642x378.png" \t "_blank)

* If the product’s stock\_quantity was 8 (less than what we’re trying to order), the database sees that the new value would be -2 which breaks the consistency rule (it should not go negative).
* The transaction fails or triggers a rollback, preventing the database from ending in an invalid state.

**How to Implement Consistency**

1. **Database Schema Constraints**
   * **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY**, **CHECK** constraints, and other schema definitions ensure no invalid entries are allowed.
2. **Triggers and Stored Procedures**
   * Triggers can automatically check additional rules whenever rows are inserted, updated, or deleted.
   * Stored procedures can contain logic to validate data before committing.
3. **Application-Level Safeguards**
   * While the database enforces constraints at a lower level, applications often add extra checks—like ensuring business rules are followed or data is validated before it even reaches the database layer.

**3. Isolation**

**Isolation** ensures that concurrently running transactions do not interfere with each other’s intermediate states.

Essentially, while a transaction is in progress, its updates (or intermediate data) remain invisible to other ongoing transactions—giving the illusion that each transaction is running sequentially, one at a time.

Without isolation, two or more transactions could read and write partial or uncommitted data from each other, causing incorrect or inconsistent results.

With isolation, developers can reason more reliably about how data changes will appear to other transactions.

**Concurrency Anomalies** (অসঙ্গতি, ব্যতিক্রম)

To understand how isolation works, it helps to see what can go wrong without proper isolation. Common concurrency anomalies include:

1. **Dirty Read**
   * Transaction A reads data that Transaction B has modified but not yet committed.
   * If Transaction B then rolls back, Transaction A ends up holding an invalid or “dirty” value that never truly existed in the committed state.
2. **Non-Repeatable Read**
   * Transaction A reads the same row(s) multiple times during its execution but sees different data because another transaction updated or deleted those rows in between A’s reads.
3. **Phantom Read**
   * Transaction A performs a query that returns a set of rows. Another transaction inserts, updates, or deletes rows that match A’s query conditions.
   * If A re-runs the same query, it sees a different set of rows (“phantoms”).

**Isolation Levels**

Databases typically allow you to choose an **isolation level**, which balances data correctness with performance.

Higher isolation levels provide stronger data consistency but can reduce system performance by increasing the wait times for transactions.

Let's explore the four common isolation levels:

1. **Read Uncommitted**
   * Allows dirty reads; transactions can see uncommitted changes.
   * Rarely used, as it can lead to severe anomalies.

[[A screenshot of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!grD2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F45a40f9b-380a-472c-bd2d-49d0e1ebe771_593x150.png)](https://substackcdn.com/image/fetch/$s_!grD2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F45a40f9b-380a-472c-bd2d-49d0e1ebe771_593x150.png" \t "_blank)

1. Read Committed
   * A transaction sees only data that has been committed at the moment of reading.
   * Prevents dirty reads, but non-repeatable reads and phantom reads can still occur.

[[A screenshot of a computer screen

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!Si5W!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F0f4f73b9-df9a-4880-8d74-ad1581665342_420x135.png)](https://substackcdn.com/image/fetch/$s_!Si5W!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F0f4f73b9-df9a-4880-8d74-ad1581665342_420x135.png" \t "_blank)

1. Repeatable Read
   * Ensures if you read the same rows multiple times within a transaction, you’ll get the same values (unless you explicitly modify them).
   * Prevents dirty reads and non-repeatable reads, but phantom reads may still happen (depending on the database engine).

[[A screenshot of a computer program

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!X-U8!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbe6068dd-a64f-4005-8343-4df514804e52_481x136.png)](https://substackcdn.com/image/fetch/$s_!X-U8!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbe6068dd-a64f-4005-8343-4df514804e52_481x136.png" \t "_blank)

1. Serializable
   * The highest level of isolation, acting as if all transactions happen sequentially one at a time.
   * Prevents dirty reads, non-repeatable reads, and phantom reads.
   * Most expensive in terms of performance and concurrency because it can require more locking or more conflict checks.

[[A computer screen with white text

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!Forv!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6eefdcc0-ecd6-4375-bb45-73df83f451fb_513x193.png)](https://substackcdn.com/image/fetch/$s_!Forv!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6eefdcc0-ecd6-4375-bb45-73df83f451fb_513x193.png" \t "_blank)

**How Databases Enforce Isolation**

**1. Locking**

* **Pessimistic**(হতাশাবাদী) **Concurrency Control**
  + Rows or tables are locked so that no other transaction can read or write them until the lock is released.
  + Can lead to blocking or deadlocks if multiple transactions compete for the same locks.

**2. MVCC (Multi-Version Concurrency Control)**

* **Optimistic Concurrency Control**
  + Instead of blocking reads, the database keeps multiple versions of a row.
  + Readers see a consistent snapshot of data (like a point-in-time view), while writers create a new version of the row when updating.
  + This approach reduces lock contention but requires carefully managing row versions and cleanup (vacuuming in PostgreSQL, for example).

**3. Snapshot Isolation**

* A form of MVCC where each transaction sees data as it was at the start (or a consistent point) of the transaction.
* Prevents non-repeatable reads and dirty reads. Phantom reads may still occur unless the isolation level is fully serializable.

**4. Durability**

**Durability** ensures that once a transaction has been committed, the changes it made will survive, even in the face of power failures, crashes, or other catastrophic(বিপর্যয়কর, সর্বনাশা, আকস্মিক বিপত্তিমূলক) events.

In other words, once a transaction says “done,” the data is permanently recorded and cannot simply disappear.

**How Databases Ensure Durability**

**1. Transaction Logs (Write-Ahead Logging)**

Most relational databases rely on a **Write-Ahead Log (WAL)** to preserve changes before they’re written to the main data files:

1. **Write Changes to WAL**: The intended operations (updates, inserts, deletes) are recorded in the WAL on durable storage (disk).
2. **Commit the Transaction**: Once the WAL entry is safely persisted, the database can mark the transaction as committed.
3. **Apply Changes to Main Data Files**: The updated data eventually gets written to the main files—possibly first in memory, then flushed to disk.

If the database crashes, it uses the WAL during **recovery**:

* **Redo**: Any committed transactions not yet reflected in the main files are reapplied.
* **Undo**: Any incomplete (uncommitted) transactions are rolled back to keep the database consistent.

**2. Replication / Redundancy**

In addition to WAL, many systems use replication to ensure data remains durable even if hardware or an entire data center fails.

* **Synchronous Replication**: Writes are immediately copied to multiple nodes or data centers. A transaction is marked committed only if the primary and at least one replica confirm it’s safely stored.
* **Asynchronous Replication**: Changes eventually sync to other nodes, but there is a (small) window where data loss can occur if the primary fails before the replica is updated.

**3. Backups**

Regular **backups** provide a safety net beyond logs and replication. In case of severe corruption, human error, or catastrophic failure:

* **Full Backups**: Capture the entire database at a point in time.
* **Incremental/Differential Backups**: Store changes since the last backup for faster, more frequent backups.
* **Off-Site Storage**: Ensures backups remain safe from localized disasters, allowing you to restore data even if hardware is damaged.

Thank you for reading!

**Basic Questions with Answers**

1. **What does ACID stand for in database transactions, and why is it important?**  
   **Answer:**  
   ACID stands for **Atomicity, Consistency, Isolation, Durability**.
   * **Atomicity:** Ensures that all operations in a transaction complete successfully or none are applied.
   * **Consistency:** Ensures the database moves from one valid state to another according to defined rules.
   * **Isolation:** Ensures concurrent transactions do not interfere with each other.
   * **Durability:** Ensures that committed transactions remain saved even after a system crash.  
     It’s important because it guarantees **data reliability and correctness** in multi-user environments.
2. **Can you explain the "Atomicity" property with an example?**  
   **Answer:**  
   Atomicity means “all or nothing.”  
   Example: In a bank transfer of $100 from Account A to Account B:
   * Deduct $100 from Account A
   * Add $100 to Account B  
     If the system fails after deducting but before adding, the transaction is rolled back so both accounts remain unchanged.
3. **What is the difference between Consistency and Isolation in ACID?**  
   **Answer:**
   * **Consistency** ensures data integrity rules are maintained before and after a transaction.
   * **Isolation** ensures that simultaneous transactions do not affect each other’s intermediate states.  
     Example: Consistency checks whether an account balance can’t be negative. Isolation ensures two concurrent withdrawals don’t read outdated balances.
4. **What happens if a transaction violates the Consistency property?**  
   **Answer:**  
   The database rejects the transaction and rolls back changes. Violating consistency means the result would leave the database in an invalid state, breaking constraints, triggers, or business rules.
5. **How does a database ensure the Durability of a committed transaction?**  
   **Answer:**  
   Durability is ensured by writing transaction logs and changes to **non-volatile storage** (disk, SSD, or distributed storage) before confirming a commit. Even if the system crashes, recovery processes use logs to restore committed data.

**Intermediate Questions with Answers**

1. **How is Atomicity typically implemented in relational databases?**  
   **Answer:**  
   It is implemented using a **transaction log** (Write-Ahead Log). The DB engine records intended changes before applying them. If any operation fails, the log is used to roll back all operations.
2. **What are some real-life scenarios where the Isolation property is critical?**  
   **Answer:**
   * **Banking:** Multiple ATMs withdrawing from the same account simultaneously.
   * **E-commerce:** Two customers purchasing the last unit of a product at the same time.
   * **Ticket booking:** Preventing double booking of the same seat.
3. **Can you describe different isolation levels and how they relate to the ACID properties?**  
   **Answer:**  
   Common isolation levels:
   * **Read Uncommitted:** Allows dirty reads (weak isolation).
   * **Read Committed:** Prevents dirty reads.
   * **Repeatable Read:** Prevents dirty and non-repeatable reads.
   * **Serializable:** Prevents dirty reads, non-repeatable reads, and phantom reads (strongest isolation).  
     Higher isolation gives stronger ACID compliance but may reduce concurrency.
4. **What is the difference between a committed transaction and a rolled-back transaction in terms of ACID?**  
   **Answer:**
   * **Committed:** Changes are permanent (Durability) and visible to others.
   * **Rolled-back:** Changes are undone (Atomicity), leaving the database unchanged.
5. **How do write-ahead logs (WAL) or redo logs help maintain ACID properties?**  
   **Answer:**  
   They record intended changes before applying them to the database. If a crash occurs:
   * **Atomicity:** The log is used to roll back incomplete transactions.
   * **Durability:** The log is replayed to ensure committed transactions are applied.

**Advanced Questions with Answers**

1. **How can database replication affect ACID compliance?**  
   **Answer:**
   * **Synchronous replication** maintains strong ACID compliance (commit waits until data is written on all replicas).
   * **Asynchronous replication** may temporarily violate Consistency or Durability during a failover because changes may not yet be on replicas.
2. **What challenges do distributed databases face in maintaining ACID properties?**  
   **Answer:**
   * Network latency and partitions make synchronous commits slower.
   * Coordinating multiple nodes requires **distributed transactions** (e.g., two-phase commit).
   * Balancing ACID with availability and scalability is challenging.
3. **How does the CAP theorem relate to ACID transactions?**  
   **Answer:**  
   CAP theorem states that in a distributed system, you can only guarantee **two of Consistency, Availability, and Partition tolerance** at the same time.  
   ACID systems prioritize **Consistency** over Availability in case of a network partition.
4. **In a high-concurrency system, how can ACID properties cause performance bottlenecks, and how would you mitigate them?**  
   **Answer:**
   * High isolation levels (e.g., Serializable) can cause locking, reducing throughput.
   * Mitigation: Use **optimistic concurrency**, **lower isolation levels** (if acceptable), **sharding**, or **read replicas**.
5. **Can you explain how a database handles partial failures to preserve Atomicity and Durability?**  
   **Answer:**
   * **Atomicity:** If a crash occurs mid-transaction, uncommitted changes are rolled back using the log.
   * **Durability:** Committed transactions are reapplied using the log during recovery.  
     This is why commit records are written to stable storage before confirming to the client.

**Scenario-Based ACID Questions**

1. **Bank Transfer Failure**  
   You are transferring $500 from Account A to Account B. The debit from A is successful, but the credit to B fails due to a network error.
   * **Question:** Which ACID property ensures both accounts remain unchanged?
   * **Expected Answer:** **Atomicity** — ensures “all or nothing.”
2. **E-Commerce Double Order**  
   Two customers order the last item in stock at the same time. Without proper isolation, both might get confirmation.
   * **Question:** Which isolation level would prevent this, and why?
   * **Expected Answer:** **Serializable** isolation — ensures transactions run as if sequential.
3. **Hotel Booking Conflict**  
   Two agents try to book the same hotel room simultaneously. One booking is committed, but the second should fail.
   * **Question:** Which ACID property enforces this?
   * **Expected Answer:** **Isolation** — prevents conflicting writes.
4. **Ticket Reservation Rollback**  
   A flight ticket booking transaction deducts a seat count and charges the customer. The payment gateway fails after the seat count update.
   * **Question:** What should the database do, and which property applies?
   * **Expected Answer:** Roll back the seat deduction (**Atomicity**).
5. **Medical Records Consistency**  
   A hospital updates a patient’s diagnosis and prescription. The new prescription violates a database constraint (e.g., drug allergy).
   * **Question:** Which property forces rejection of the update?
   * **Expected Answer:** **Consistency**.
6. **Shopping Cart Checkout Crash**  
   During checkout, payment is processed and the order is created, but the server crashes before confirmation.
   * **Question:** Which property ensures the order is still recorded after restart?
   * **Expected Answer:** **Durability**.
7. **Inventory Update Race Condition**  
   Multiple warehouse workers update stock levels at the same time, causing incorrect totals.
   * **Question:** Which ACID property is violated here, and how would you fix it?
   * **Expected Answer:** **Isolation** — use proper locking or higher isolation levels.
8. **Stock Trading Platform**  
   A user buys shares, but due to concurrency issues, the same shares are sold twice to different users.
   * **Question:** Which ACID property would prevent this?
   * **Expected Answer:** **Isolation** (Serializable level).
9. **Restaurant Reservation App**  
   The app books a table and updates customer points in two separate steps. If one step fails, the other still commits.
   * **Question:** Which property is missing?
   * **Expected Answer:** **Atomicity** — should wrap both in one transaction.
10. **Distributed Payment System**  
    A payment is processed in one database, but confirmation is stored in another database. The second database is down.
    * **Question:** What technique ensures ACID in this case?
    * **Expected Answer:** **Two-Phase Commit (2PC)** in distributed transactions.
11. **Order Processing with Logging**  
    Your system writes an order to the DB and a log entry for auditing. After commit, the server crashes before sending confirmation to the user.
    * **Question:** Will ACID guarantee the log is there?
    * **Expected Answer:** Yes — **Durability** ensures committed changes remain.
12. **Bank Withdrawal with Low Isolation**  
    A customer checks their balance and withdraws money. Meanwhile, another withdrawal is processed before the first one finishes.
    * **Question:** Which read phenomenon can occur?
    * **Expected Answer:** **Non-repeatable read** or **dirty read** depending on isolation level.
13. **E-Wallet Transfer Across Time Zones**  
    A user sends funds at 11:59 PM from one country, received at 12:01 AM in another. Business rules require same-day transactions only.
    * **Question:** Which property ensures the date rules are enforced?
    * **Expected Answer:** **Consistency** — validates business constraints.
14. **Hotel PMS System Crash**  
    While processing group check-in, the system updates room statuses and creates billing entries. The crash happens mid-way.
    * **Question:** Which property ensures either all rooms and bills are updated or none?
    * **Expected Answer:** **Atomicity**.
15. **Read vs Write Delay in Analytics**  
    Your analytics dashboard reads from a replica database for performance. A recent transaction isn’t yet reflected there.
    * **Question:** Which ACID property might be temporarily broken here?
    * **Expected Answer:** **Consistency** — especially in asynchronous replication.