What are ACID Transactions in Databases?

Explained with Examples

ASHISH PRATAP SINGH
FEB 04, 2025

♡ 207 5 9 Share

Imagine you're running an e-commerce application.

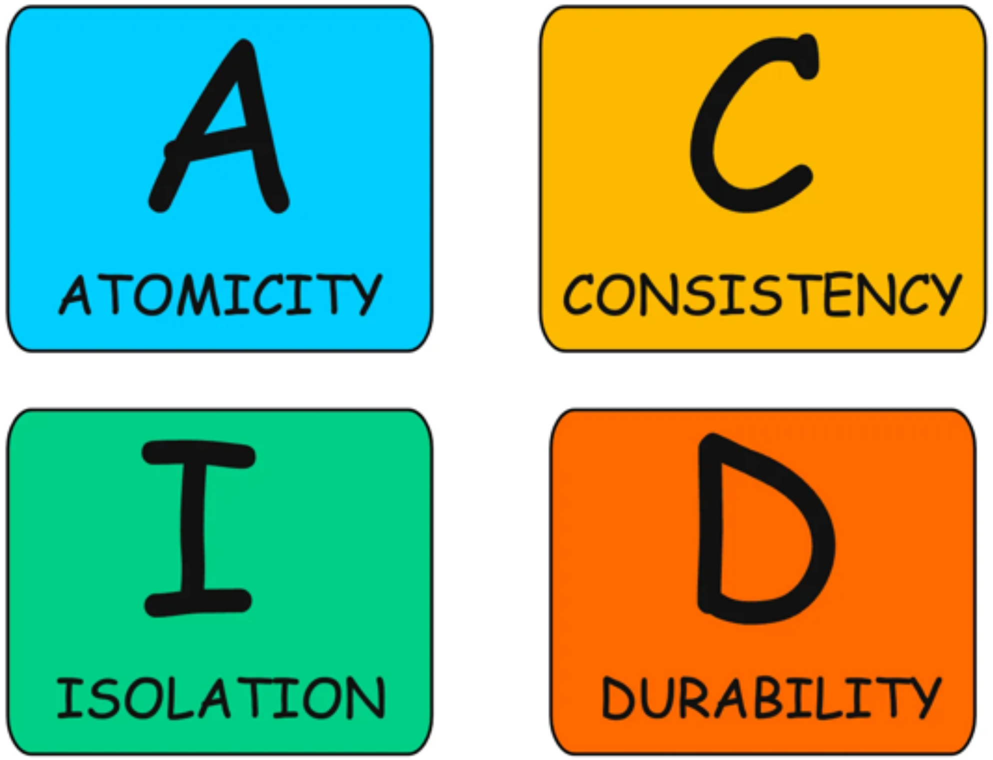
A customer places an order, and your system needs to deduct the item from inventory, charge the customer's credit card, and record the sale in your accounting system—all at once.

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 a transaction:

Atomicity, **Consistency**, **Isolation**, and **Durability**.



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.

If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

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).

Now, lets looks at each of the ACID properties.

---

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.

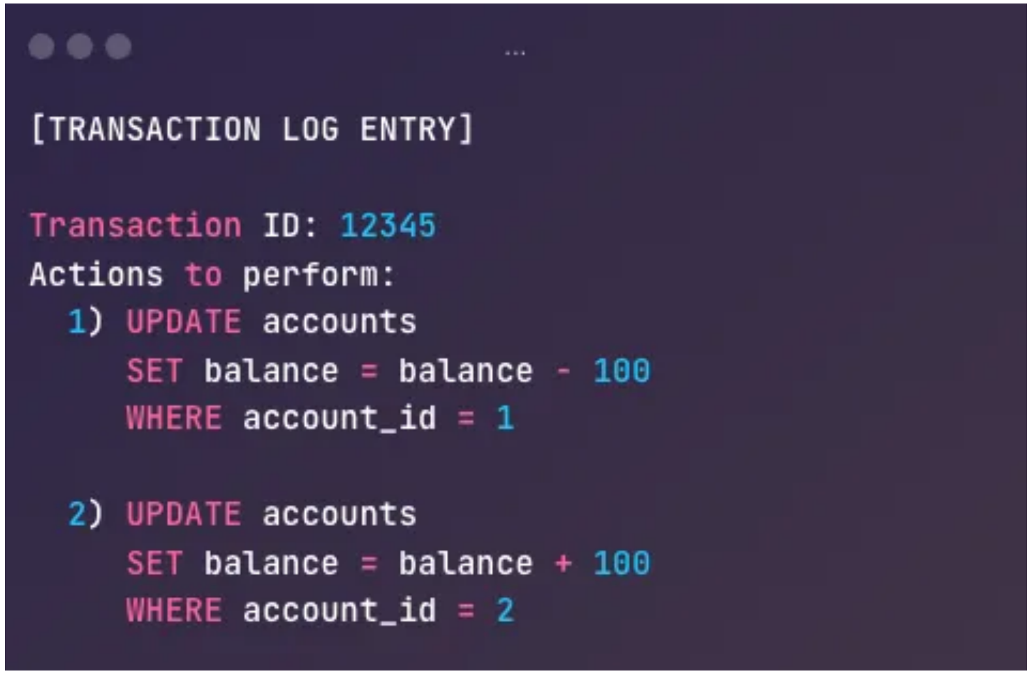
How Databases Implement Atomicity

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.

Example:



```
[TRANSACTION LOG ENTRY]

Transaction ID: 12345
Actions to perform:
  1) UPDATE accounts
     SET balance = balance - 100
     WHERE account_id = 1

  2) UPDATE accounts
     SET balance = balance + 100
     WHERE account_id = 2
```

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:

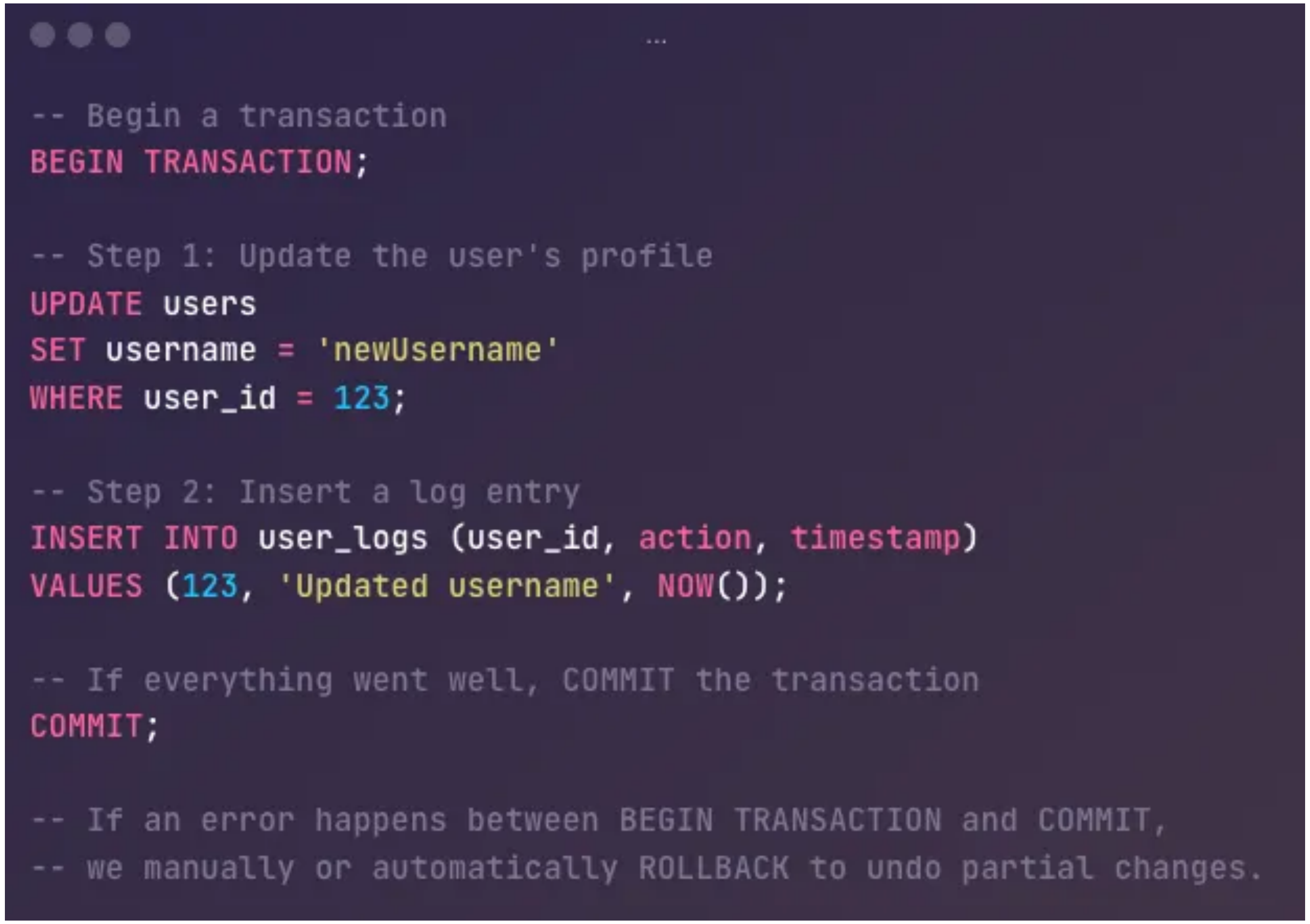
- On restart, the database checks the WAL.
- It sees **Transaction 12345** was committed.
- It reapplies the **UPDATE** operations to ensure the final balances are correct in the data files.

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.

Example:



```sql
-- Begin a transaction
BEGIN TRANSACTION;

-- Step 1: Update the user's profile
UPDATE users
SET username = 'newUsername'
WHERE user_id = 123;

-- Step 2: Insert a log entry
INSERT INTO user_logs (user_id, action, timestamp)
VALUES (123, 'Updated username', NOW());

-- If everything went well, COMMIT the transaction
COMMIT;

-- If an error happens between BEGIN TRANSACTION and COMMIT,
-- we manually or automatically ROLLBACK to undo partial changes.
```

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.

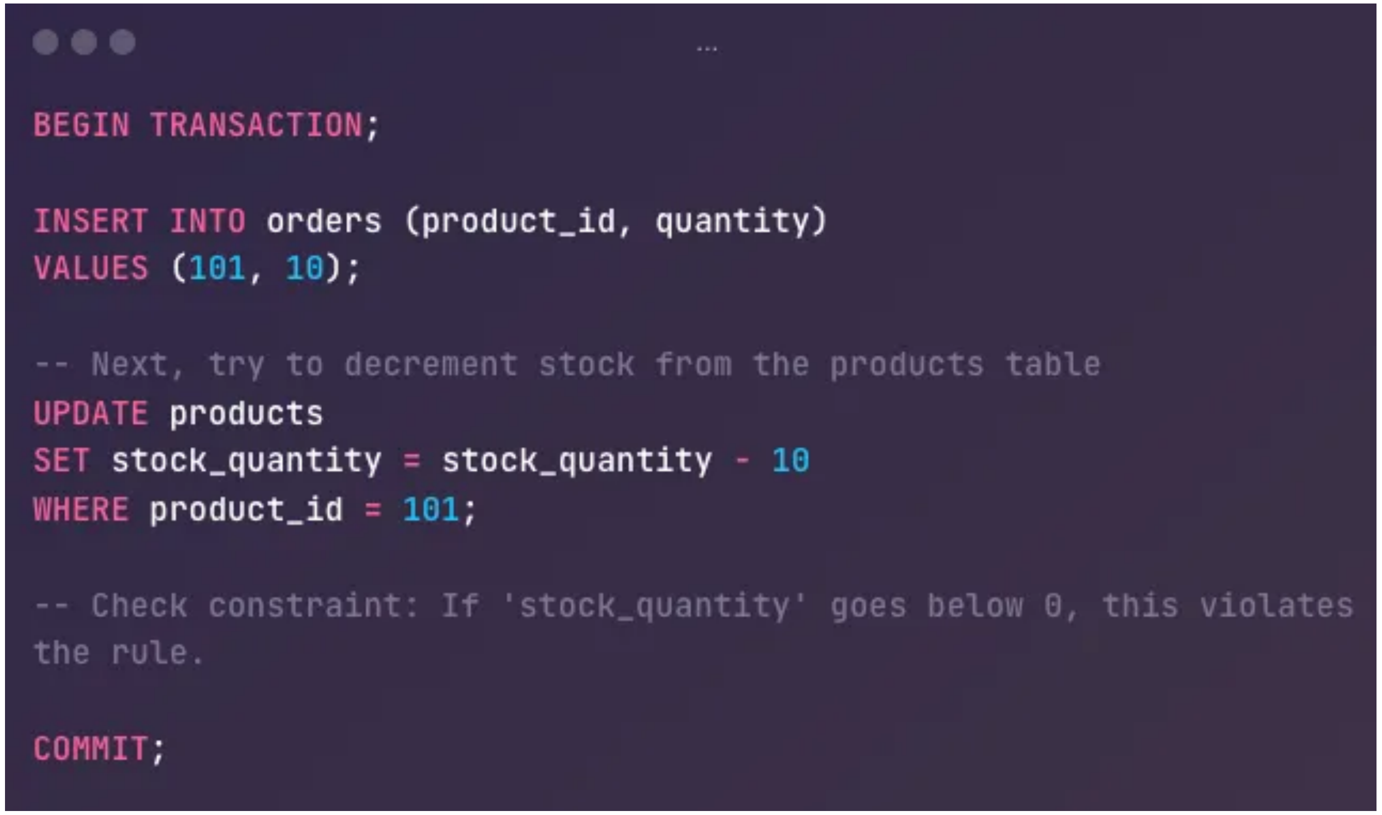
Example:

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



```
BEGIN TRANSACTION;

INSERT INTO orders (product_id, quantity)
VALUES (101, 10);

-- Next, try to decrement stock from the products table
UPDATE products
SET stock_quantity = stock_quantity - 10
WHERE product_id = 101;

-- Check constraint: If 'stock_quantity' goes below 0, this violates
the rule.

COMMIT;
```

- 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

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”).

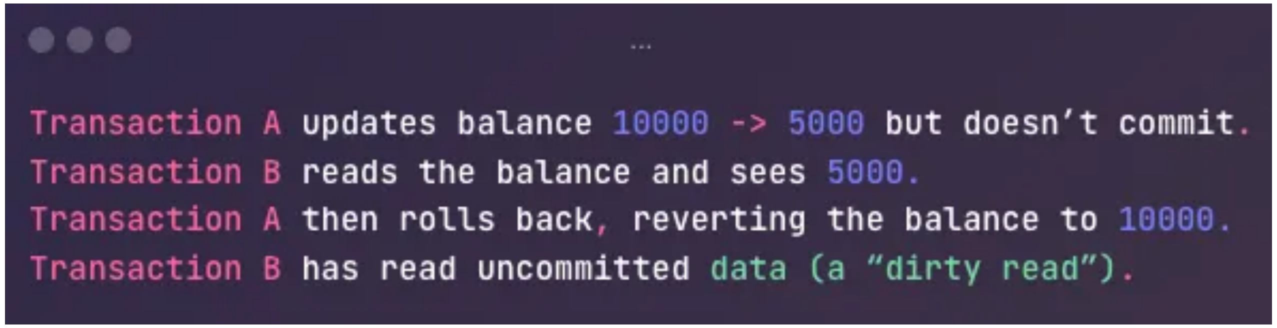
Isolation Levels

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.

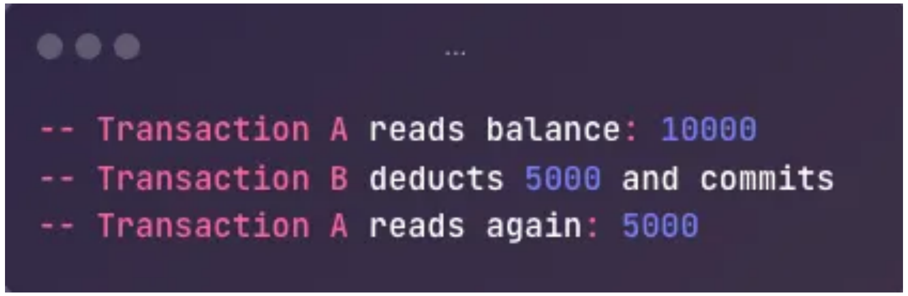
Let's explore the four common isolation levels:

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



```
Transaction A updates balance 10000 -> 5000 but doesn’t commit.
Transaction B reads the balance and sees 5000.
Transaction A then rolls back, reverting the balance to 10000.
Transaction B has read uncommitted data (a “dirty read”).
```

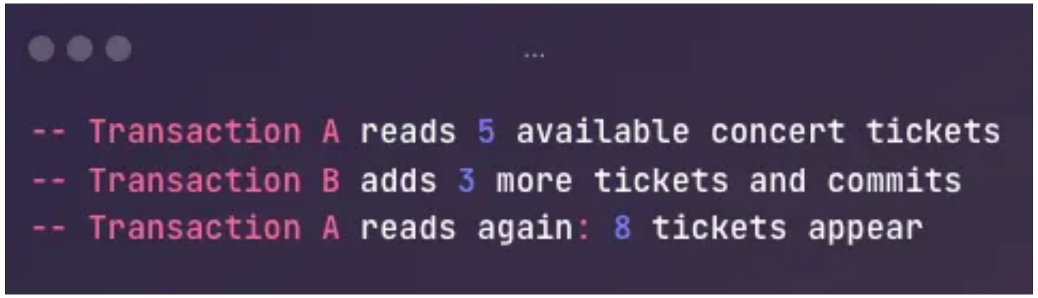
2. **Read Committed**
    - A transaction sees only data that has been committed at the moment of reading.
    - Prevents dirty reads, but non-repeatable reads and phantom reads can still occur.



```
-- Transaction A reads balance: 10000
-- Transaction B deducts 5000 and commits
-- Transaction A reads again: 5000
```

3. **Repeatable Read**

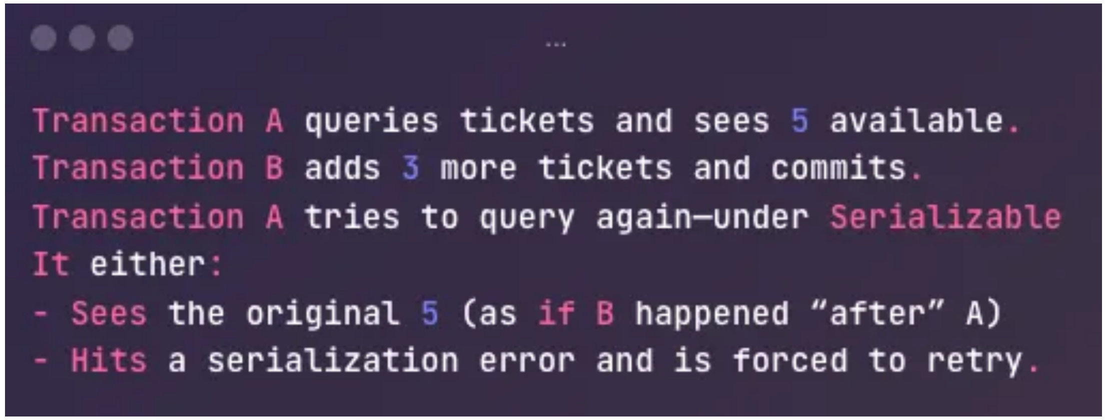
    - Ensures if you read the same rows multiple times within a transaction, you'll get the same values (unless you explicitly modify them).
    - Prevents dirty reads and non-repeatable reads, but phantom reads may still happen (depending on the database engine).



```
-- Transaction A reads 5 available concert tickets
-- Transaction B adds 3 more tickets and commits
-- Transaction A reads again: 8 tickets appear
```

4. **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.



```
Transaction A queries tickets and sees 5 available.
Transaction B adds 3 more tickets and commits.
Transaction A tries to query again—under Serializable
It either:
- Sees the original 5 (as if B happened "after" A)
- Hits a serialization error and is forced to retry.
```

How Databases Enforce Isolation

1. Locking

- **Pessimistic 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 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!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

There are group discounts, gift options, and referral bonuses available.

Checkout my Youtube channel for more in-depth content.

Follow me on LinkedIn, X and Medium to stay updated.

Checkout my [GitHub repositories]() for free interview preparation resources.

I hope you have a lovely day!

See you soon,
Ashish

207 Likes · 9 Restacks

← Previous Next →

Discussion about this post

Comments Restacks

Write a comment...

Ashwani Yadav 4 Feb *Edited*

Liked by Ashish Pratap Singh

Thanks, Ashish for this article.

I would like to point out that Snapshot Isolation (SI) prevents Phantom reads but write skew may still occur unless the isolation level is fully serializable.

I would like to know some examples of phantom reads in SI if they occur.

Example of Write Skew under Snapshot Isolation

Scenario: Doctor On-Call Scheduling System

Suppose a hospital enforces a rule that at least one doctor must be on call at any time. The system stores on-call doctors in a database:

CREATE TABLE on_call_doctors (

doctor_id INT PRIMARY KEY,

is_on_call BOOLEAN

);

Currently, there are two doctors, both on call:

doctor_id => is_on_call

1 => true

2 => true

Step-by-Step Breakdown of Write Skew:

Expand full comment

LIKE (4) REPLY SHARE

LIKE (4) REPLY SHARE

1 reply by Ashish Pratap Singh

Sajid 6 Feb

Liked by Ashish Pratap Singh

Awesome post! Really helpful for interview prep

LIKE (2) REPLY SHARE

3 more comments...