ACID Transactions in Databases

ACID is a set of four essential properties that ensure reliable database transactions. The term stands for:

1. **Atomicity** – All operations in a transaction are completed successfully or none of them are applied.
2. **Consistency** – The database remains valid before and after the transaction.
3. **Isolation** – Transactions are executed independently without interfering with each other.
4. **Durability** – Once a transaction is committed, it remains permanent even in case of a system failure.

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).

Example: Bank Money Transfers can fail or succeed

Consider an online banking system where a user transfers ₹10,000 from **Account A** (**sender**) **to Account B** (**receiver**).

This transaction consists of two dependent operations:

1. Deduct ₹10,000 from Account A.
2. Add ₹10,000 to Account B.

If an error occurs after deducting money but before adding it to the recipient's account, the money could be lost!
Atomicity prevents this scenario by ensuring that:

- **If both operations succeed** → The transaction is committed.
- **If any operation fails** → The transaction is rolled back, and the database remains unchanged.

ACID Transactions in Databases

Bank Transaction Example in Django

1. Without Atomicity (Incorrect Approach)

Let's first see what happens **without atomicity**.

```
def transfer_money(sender, receiver, amount):
    # Deduct money from sender
    sender.balance -= amount
    sender.save()

    # Simulating an error (e.g., system crash)
    if amount > 5000:
        raise ValueError("Transaction failed due to limit exceeded")

    # Add money to receiver
    receiver.balance += amount
    receiver.save()

    print("Transaction Successful")
```

ACID Transactions in Databases

Problems in the Above Code:

- If an error occurs after sender.save() but before receiver.save(), the money is deducted but not added to the recipient.
- This results in data inconsistency, violating atomicity.
- If the database crashes midway, Account A loses money, and Account B never receives it.

---

2. Correct Approach: Using Atomic Transactions

To fix this, we wrap the operations inside a transaction block using **transaction.atomic()**.

```python
from django.db import transaction

def transfer_money(sender, receiver, amount):
    try:
        with transaction.atomic():  # Start an atomic transaction block
            sender.balance -= amount  # Deduct from sender
            sender.save()

            # Simulating an error (e.g., system crash)
            if amount > 5000:
                raise ValueError("Transaction failed due to limit exceeded")

            receiver.balance += amount  # Add to receiver
            receiver.save()

        print("Transaction Successful")

    except Exception as e:
        print(f"Transaction Rolled Back: {e}")
```

ACID Transactions in Databases

Deep Explanation of the Code

Step 1: Start an Atomic Transaction

```
with transaction.atomic():
```

- This ensures all operations inside the block are treated as a single transaction.
- If any statement inside the block fails, everything is rolled back.

Step 2: Deduct Money from the Sender

```
sender.balance -= amount

sender.save()
```

- The sender's balance is updated in memory, but not yet permanently saved.
- If the transaction fails later, this change will be reversed.

Step 3: Simulate an Error (Failure Case)

```
if amount > 5000:
    raise ValueError("Transaction failed due to limit exceeded")
```

- We artificially introduce an error for transactions over ₹5000.

- This forces a rollback, ensuring no money is deducted without being credited.

Step 4: Add Money to the Receiver

```
receiver.balance += amount

receiver.save()
```

- The receiver's account is credited with the transferred amount.
- If this step fails, Django automatically rolls back all previous steps.

Step 5: Commit or Rollback the Transaction

- **I**f no errors occur, Django commits the transaction, making changes permanent.
- If an error occurs, Django rolls back all previous operations.

Advantages of Using `transaction.atomic()`

✅ **Prevents Data Loss** – Ensures the sender's money is not deducted without being received.
✅ **Maintains Database Consistency** – No incomplete transactions.
✅ **Handles System Failures** – Transactions remain reliable even in case of crashes.
✅ **Easy to Implement** – Just wrap your code in `with transaction.atomic():`.

ACID Transactions in Databases

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:

Ensuring Consistency in Movie Ticket Booking System with ACID Transactions

Scenario: Booking a Movie Ticket

In a movie booking system, when a user books a ticket, the system must:

Check seat availability.

Deduct the seat from available seats.

Create a booking record.

1. Correct Approach: Using transaction.atomic()

ACID Transactions in Databases

Step 1: Start a Transaction Block

```python
from django.db import transaction

def book_ticket(user, show, seat_count):
    try:
        with transaction.atomic():  # Ensuring atomicity
            # Step 1: Check available seats
            if show.available_seats < seat_count:
                raise ValueError("Not enough seats available!")

            # Step 2: Deduct seats
            show.available_seats -= seat_count
            show.save()

            # Step 3: Simulating a failure (e.g., payment failure)
            if seat_count > 5:
                raise Exception("Payment gateway timeout!")

            # Step 4: Create the booking record
            Booking.objects.create(user=user, show=show, seat_count=seat_count)

        print("Booking successful!")

    except Exception as e:
        print(f"Booking failed, transaction rolled back: {e}")
```

with transaction.atomic():

✅ Ensures that all operations inside the block execute together.

✅ If any step fails, Django rolls back all changes.

Step 2: Check Available Seats

if show.available_seats < seat_count:

raise ValueError("Not enough seats available!")

✅ Prevents overbooking by ensuring sufficient seats are available before proceeding.

Step 3: Deduct the Seats

```
show.available_seats -= seat_count

show.save()
```

✅ Reduces the number of available seats only if booking succeeds.

Step 4: Simulating an Error

```
if seat_count > 5:
    raise Exception("Payment gateway timeout!")
```

✅ If a payment failure occurs, all changes are rolled back, and no seats are deducted.

Step 5: Create the Booking Record

```
Booking.objects.create(user=user, show=show, seat_count=seat_count)
```

✅ Ensures booking is only recorded if seat deduction succeeds.

ACID Transactions in Databases

Isolation in ACID Transactions:

Isolation ensures that concurrent transactions do not interfere with each other, maintaining data accuracy and integrity. When multiple transactions occur at the same time, their execution should behave as if they were executed sequentially, even if they run in parallel.

Example of Isolation in ACID Transactions

Scenario: Airline Ticket Booking System

Imagine two users trying to book the last seat on a flight at the same time.

User A starts booking a ticket.

User B also starts booking the same ticket simultaneously.

If isolation is not maintained, both users might get a confirmation, causing double booking (race condition).

Correct Approach: Using select_for_update()

```python
from django.db import transaction

def book_ticket(user, flight):
    try:
        with transaction.atomic():
            # Step 1: Lock the flight row to prevent simultaneous modifications
            flight = Flight.objects.select_for_update().get(id=flight.id)

            # Step 2: Check if seats are available
            if flight.available_seats <= 0:
                raise ValueError("No seats available!")

            # Step 3: Deduct the seat
            flight.available_seats -= 1
            flight.save()

            # Step 4: Confirm booking
            Booking.objects.create(user=user, flight=flight)

        print("Booking successful!")

    except Exception as e:
        print(f"Booking failed, transaction rolled back: {e}")
```

ACID Transactions in Databases

Why This Ensures Isolation?

✅ Prevents double booking – The select_for_update() lock prevents another transaction from accessing the seat count before the first transaction commits.

✅ Avoids race conditions – Users cannot modify the same data simultaneously.

✅ Ensures fairness – The first user to complete the transaction gets the seat.

Durability in ACID Transactions

What is Durability?

Durability ensures that once a transaction **is committed, it remains in the database permanently**, even if a system crash, power failure, or error occurs.

Why is Durability Important?

Without durability, a committed transaction **might be lost** if the system crashes right after execution.

Scenario: Online Shopping Order System

- A user places an order.
- The system **deducts stock** and **creates an order record**.
- A **power failure occurs** immediately after.

Without durability, the order might be lost, leading to **stock mismatches**.

Correct Approach: Using `transaction.atomic()`

Why This Ensures Durability?

✅ **Data persists even after crashes** – If the system crashes **after commit**, the order remains in the database.

ACID Transactions in Databases

✅ **Prevents data loss** – Once an order is placed, it cannot be lost.
✅ **Ensures accurate stock updates** – The stock count always matches actual orders.