POSTGRESQL TRANSACTION

A **database transaction** is a single unit of work that consists of one or more operations.

A **database transaction** is a collection of SQL queries that are treated as one unit of work. Basically, you begin a transaction and then do some queries, and then end the transaction, this is one unit of work.

PostgreSQL transactions adhere to the ACID principles (Atomicity, Consistency, Isolation, and Durability) to manage concurrent access and prevent data corruption.

WHEN TO USE TRANSACTIONS

- **Transactions are most beneficial in scenarios requiring data:**
- Financial applications (banking, e-commerce)
- Inventory management systems
- Reservation systems (airlines, hotels)
- Applications with multi-user access to shared data

ACID

- **Atomicity** guarantees that the transaction completes in an all-or-nothing manner.
- **Consistency** ensures the change to data written to the database must be valid and follow predefined rules.
- **Isolation** determines how transaction integrity is visible to other transactions.
- **Durability** makes sure that transactions that have been committed will be stored in the database permanently.

ATOMICITY

- **Atomicity** ensures that all the operations within a transaction are treated as a single, indivisible unit of work. This means that if any operation within the transaction fails, the entire transaction will be rolled back and none of the changes will be committed to the database.

CONSISTENCY

- **Consistency** ensures that the transaction brings the database from one valid state to another. This means that the transaction must follow all the rules and constraints defined in the database schema, such as unique keys, foreign keys, and check constraints.

CONSISTENCY IN DATA

This represented the state that actually persisted in the data. This mainly involves enforcing.

-**foreign keys** referential integrity between two tables or documents an example can be seen when creating a user-like system, when a user(s) like an image or blog, the blog or image should persist the actual number of likes it has got from the user table.

- **atomicity** – data should persist across the database.

-**Isolation** – based on the isolation level which we will talk about later should return correct reads from two concurrent parallel tables.

ISOLATION

- **Isolation** ensures that concurrent transactions do not interfere with each other. Each transaction is executed as if it is the only transaction in the system, even though other transactions may be executing simultaneously. It is the result of having transactions as a separate entity (isolation) from other concurrent transactions which may lead to reads phenomena and Isolation levels.

DURABILITY

- **Durability** ensures that the changes made by a committed transaction are permanent and will survive any subsequent failures. This is typically achieved by writing or persisting the changes to disk or other non-volatile storage. This involves a system that can recover all writes and see all changes after committed even when the system crashes or loss of power.

TRANSACTION LIFECYCLE

- **Begin**: Starts the transaction.
- **Execute**: Performs database operations like reads, writes, or updates.
- **Commit**: Confirms and saves changes if all operations succeed.
- **Rollback**: Reverts changes if an error occurs, maintaining data integrity.

BEGIN A TRANSACTION

BEGIN TRANSACTION;

or

BEGIN WORK;

or just:

BEGIN;

COMMIT A TRANSACTION

COMMIT WORK;

or

COMMIT TRANSACTION;

or simply:

COMMIT;

ROLLING BACK A TRANSACTION

To roll back or undo the change of the current transaction, you use any of the following statement:

ROLLBACK WORK;

or

ROLLBACK TRANSACTION;

or in short:

ROLLBACK;

BASIC TRANSACTION CONTROL STATEMENTS

- **BEGIN**: Starts a transaction.
- **COMMIT**: Saves changes made in the transaction to the database.
- **ROLLBACK**: Undoes all changes made within the transaction.

```
BEGIN;
    INSERT INTO users (user_id, balance) VALUES (1, 100);
    UPDATE users SET balance = balance - 10 WHERE user_id = 1;
    UPDATE users SET balance = balance + 10 WHERE user_id = 2;
COMMIT;
```

If an error occurs at any point, you can call ROLLBACK instead of COMMIT

EXAMPLES

```sql
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    stock INT
);
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    product_id INT,
    quantity INT,
    order_date TIMESTAMP
);
INSERT INTO products (name, stock) VALUES
('Laptop', 10),
('Phone', 20);

INSERT INTO orders (product_id, quantity, order_date) VALUES
(1, 2, '2024-11-12 10:00:00'),
(2, 5, '2024-11-12 10:15:00');
```

	id integer	name character varying (100)	stock integer	id integer	product_id integer	quantity integer	order_date timestamp without time zone
1	1	Laptop	10	1	1	2	2024-11-12 10:00:00
2	2	Phone	20	1	1	2	2024-11-12 10:00:00
3	1	Laptop	10	2	2	5	2024-11-12 10:15:00
4	2	Phone	20	2	2	5	2024-11-12 10:15:00

EXAMPLES

```sql
BEGIN;
-- 1. adding  new order
INSERT INTO orders (product_id, quantity, order_date)
VALUES (1, 3, '2024-11-12 12:00:00');

-- 2. updating orders quantity
UPDATE products
SET stock = stock - 3
WHERE id = 1;

-- 3. deleting canceled order
DELETE FROM orders
WHERE id = 2;

--ending transaction
COMMIT;
```

	id integer	name character varying (100)	stock integer	id integer	product_id integer	quantity integer	order_date timestamp without time zone
1	2	Phone	20	1	1	2	2024-11-12 10:00:00
2	1	Laptop	7	1	1	2	2024-11-12 10:00:00
3	2	Phone	20	3	1	3	2024-11-12 12:00:00
4	1	Laptop	7	3	1	3	2024-11-12 12:00:00

ISOLATION LEVELS

- **Read Uncommitted**: Allows transactions to read uncommitted changes.
- **Read Committed** (Default): Only reads committed data.
- **Repeatable Read**: Guarantees consistent data for the duration of a transaction, even if other transactions modify it.
- **Serializable**: The strictest level, ensuring transactions behave as if executed sequentially.

CONCURRENCY ANOMALIES

- Each anomaly happens when multiple transactions access the same data without proper isolation.
- **Dirty Read:**
- A transaction reads data that another transaction has written but NOT committed yet.
- **Non-Repeatable Read:**
- A transaction reads the same row twice, but gets different values because another committed transaction modified it in between.
- **Phantom Read:**
- A transaction executes the same query twice, and new rows appear or disappear between reads.
- **Lost Update:**
- Two transactions both read the same value, then both write based on the old value.
- **Write Skew:**
- Two transactions read overlapping data, both decide to modify different rows based on the *same snapshot*, and the combined updates violate a business rule.

Anomaly	Description	Read Committed	Repeatable Read	Serializable
Dirty Read	Read uncommitted data	❌ (prevented)	❌	❌
Non-Repeatable Read	Re-read row → changed	✅	❌	❌
Phantom Read	Re-read query → new/missing rows	✅	❌*	❌
Lost Update	One update overwrites another	✅	✅	❌
Write Skew	Combined logical violation	✅	✅	❌

SET AN ISOLATION LEVEL:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
--transaction statements
COMMIT;
```

SAVEPOINTS

- **Savepoints**: Savepoints allow you to create a checkpoint within a transaction, which can be rolled back to without affecting the entire transaction.

```
BEGIN;
    INSERT INTO users (user_id, balance) VALUES (3, 200);
    SAVEPOINT save1;
    UPDATE users SET balance = balance + 50 WHERE user_id = 3;
    ROLLBACK TO save1; --return to the savepoint
COMMIT;
```

BENEFITS OF SAVEPOINTS

- **Flexibility in Transactions:** Savepoints allow you to manage complex transactions that may require only partial rollbacks rather than full transaction rollbacks.
- **Improved Error Management:** By using savepoints, you can isolate and handle errors within sections of a transaction, retry specific steps, or skip them if needed.
- **Enhanced Modularity:** Savepoints break a large transaction into parts, making it easier to understand, test, and debug, particularly useful in complex applications like financial processing or booking systems.

EXAMPLES

Update Flight Status to ‘Boarding’:

```
BEGIN;
SELECT status FROM flights WHERE flight_no= 'US-KS';
UPDATE flights SET status = 'Boarding' WHERE flight_no = 'US-KS';
COMMIT;
```

EXAMPLES

```
BEGIN;
SELECT status FROM flights WHERE flight_no = 'US-KS';
UPDATE flights SET status = 'full' WHERE flight_no = 'US-KS';
INSERT INTO booking(passenger_id, flight_id)
VALUES ((SELECT passenger_id FROM passengers WHERE first_name = 'Zere' AND last_name = 'Ali'),
        (SELECT flight_id FROM flights WHERE flight_no = 'US-KS'));
COMMIT;
```

- The transaction starts with BEGIN.
- The status of **flight UK-KS** is checked. Since the flight is initially available, it will return 'available'.
- Once Zere's books her ticket, the flight's status is updated to 'full', indicating no more seats are available
- A record is inserted for Zere in the **booking** table.
- The transaction is committed, which saves the changes to the database.

EXAMPLES

```
BEGIN;
SELECT status FROM flights WHERE flight_no= 'US-KS';
UPDATE flights SET status = 'Available' WHERE flight_no = 'US-KS';
DELETE FROM booking WHERE booking_id=501;
COMMIT;
```

- Mark a passenger's baggage_check as 'Checked', but only if the passenger exists and the airline is from China.

EXAMPLE

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
-- 1. Read the airline (ensure it is from China)
SELECT airline_country
FROM airline
WHERE airline_id = 20
FOR SHARE;

-- 2. Read the baggage check record
SELECT check_result, passenger_id
FROM baggage_check
WHERE baggage_check_id = 1
FOR UPDATE;

-- 3. Update baggage_check only if it is currently "Not checked"
UPDATE baggage_check
SET check_result = 'Checked',
    update_at = CURRENT_DATE
WHERE baggage_check_id = 1
  AND check_result = 'Not checked';

COMMIT;
```

ADVANTAGES OF USING TRANSACTIONS

- **ACID**
- **Error Handling and Rollback Capability:** Transactions allow error handling through rollbacks. If an error or failure occurs, changes are reverted to the initial state, protecting the database from corrupt or partial data entries.
- **Improved Concurrency Control:** By isolating transactions, databases can manage multiple users and operations concurrently without compromising data consistency. This is particularly beneficial for multi-user environments where several transactions are processed simultaneously.

DISADVANTAGES OF USING TRANSACTIONS

- **Increased Complexity and Overhead**:
- Managing transactions and implementing isolation levels adds complexity to application design and introduces computational overhead.
- **Reduced Concurrency Due to Locking**:
- Depending on the isolation level, transactions may require extensive locking, reducing the level of concurrency by blocking access to certain data until the transaction is complete.
- **Resource Consumption**:
- Transactions often consume significant resources, especially in long or complex transactions
- **Difficulty in Debugging and Troubleshooting**:
- Errors in transactions can be challenging to diagnose and troubleshoot due to the potential for complex dependencies between operations within the transaction.