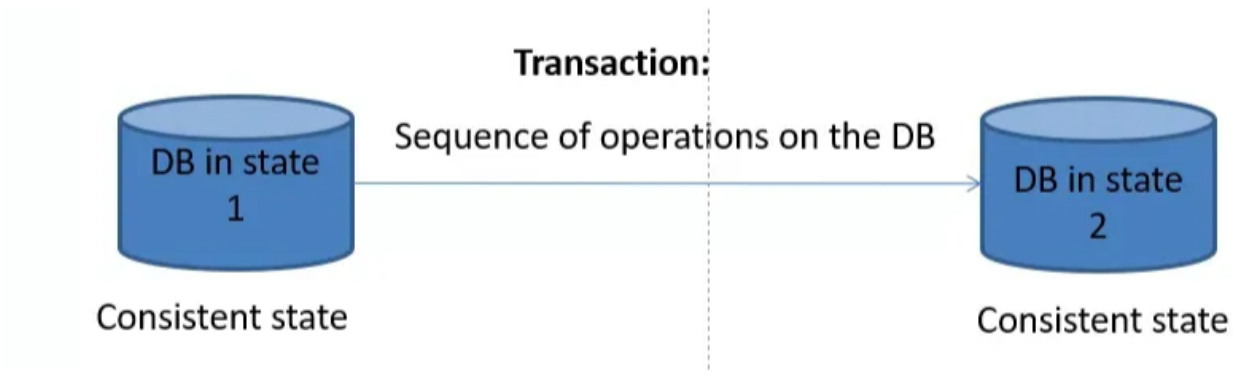
Lecture 2

Type	Lecture

Transactions

In order to keep the database or the business logic consistent, it is often crucial that all commands of a sequence run successfully or no command runs at all. A sequence of commands that necessarily needs to run together is called a transaction, it is a sequence of (read or write) operations that transfers a database from one consistent state to another consistent, not necessarily different state. The commands of a transaction are seen as one unit: Either all of the commands succeed or none.



A lot of issues can prevent the necessary operations from running successfully:

- Duplicate key error.
- Violation of FK or referential integrity.
- Violation of CHECK or trigger conditions.
- other application errors.
- connection gets lost in the middle of the commands' execution.
- Electricity shuts down after the first write operation.
- Database crashes after first write operation.

▼ Example

Textbook example of a typical transaction - bank transfer - in a banking application:

1. Read the account balance of *A* into the variable *a*: **read**(*A*,*a*);
2. Reduce the account balance by 50: *a*:= *a* - 50;
3. Write the new account balance to the database: **write**(*A*,*a*);
4. Read the account balance of *B* into the variable *b*: **read**(*B*,*b*);
5. Increase the account balance by 50: *b* := *b* + 50;
6. Write the new account balance to the database: **write**(*B*, *b*);

Transaction Concept

A transaction is a way for an application to group several reads and writes together into a logical unit. Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (commit) or it fails (abort, rollback). if it fails, application can safely retry.

Advantages

- error handling becomes much simpler for an application (it does not need to worry about partial failure-i.e., the case where some operations succeed and some fail).
- simplify the programming model for applications accessing a database (safety guarantees).

Disadvantages

- lowers performance
- you need to have a centralized system.

SQL Commit and Rollback

COMMIT and ROLLBACK are performed on transactions. A transaction is the smallest unit of work that is performed against a database. Its a sequence of instructions in a logical order. A transaction can be performed manually by a programmer or it can be triggered using an automated program.

SQL Commit

COMMIT is the SQL command that is used for storing changes performed by a transaction. When a COMMIT command is issued it saves all the changes since last COMMIT or ROLLBACK.

SQL RollBack

ROLLBACK is the SQL command that is used for reverting changes performed by a transaction. When a ROLLBACK command is issued it reverts all the changes since last COMMIT or ROLLBACK.

https://www.digitalocean.com/community/tutorials/sql-commit-sql-rollback#sql-commit-and-rollback

---

Transaction Commands

- Start Transaction / Begin / Begin of Transaction / BoT
- Commit: Ends a transaction successfully: All writes made by the transaction are durable (persistent).
- Rollback / abort: The transaction is aborted. The DBMS restores the database to exactly the consistent state that existed before the transaction started.
  The DBMS automatically performs a rollback if it could not run all commands of the transaction. That is, the DBMS allows a transaction to fail and performs a rollback on its own.
- Rollback set in application code: Of course, a rollback can also be set in the application code by the application developer for error handling.

▼ Example

```
-- Start a transaction
BEGIN TRANSACTION;

-- Insert a new employee
INSERT INTO Employees (ID, Name, Salary)
VALUES (101, 'John Doe', 60000);

-- Check the inserted salary and decide whether to commit or rollback
IF (SELECT Salary FROM Employees WHERE ID = 101) < 50000
BEGIN
   -- Undo the changes if salary is too low
   ROLLBACK;
   PRINT 'Transaction rolled back because salary is too low.';
END
ELSE
BEGIN
   -- Save the changes permanently
```

```
    COMMIT;
    PRINT 'Transaction committed successfully.';
END
```

ACID principle

if a database supports ACID principle, it is considered a transactional database.

`Atomicity` - a transaction is invisible. Either all operations are executed successfully or none is executed. All-or-Nothing-Principle.

▼ example

Bank Transfer
Suppose we are transferring $100 from Account A to Account B. The transaction should ensure that either both operations succeed or neither happens.

```
BEGIN TRANSACTION;

-- Deduct $100 from Account A
UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 1;

-- Add $100 to Account B
UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 2;

-- Check if both operations are valid
IF @@ERROR = 0
    COMMIT; -- Save both operations
ELSE
    ROLLBACK; -- Undo all changes if any error occurs
```

If an error occurs (e.g., Account A has insufficient funds), the transaction **rolls back** and Account B does not receive money.

`Consistency` - a transaction transfers a database from one consistent state to another (not necessarily different) consistent state, preserving all defined rules (referential integrity, constraints, cascades, triggers, etc.). Within a transaction, all rules or constraints, placed on the data need to be kept.

▼ example

Enforcing Constraints
Let's say we have a
`Products` table where `StockQuantity` should never be negative.

```
BEGIN TRANSACTION;

UPDATE Products
SET StockQuantity = StockQuantity - 10
WHERE ProductID = 1;

-- If stock goes negative, rollback to maintain consistency
IF (SELECT StockQuantity FROM Products WHERE ProductID = 1) < 0
    ROLLBACK;
ELSE
    COMMIT;
```

This ensures the database remains in a valid state where `StockQuantity` is never negative.

`Durability` - with successful commit, all changes made within a transaction are made permanent.

▼ example

Power Failure Handling
Imagine a system crash happens after a
`COMMIT`.

```
BEGIN TRANSACTION;

INSERT INTO Orders (OrderID, CustomerID, TotalAmount)
VALUES (1234, 567, 250.00);

COMMIT;
```

Even if the database crashes right after `COMMIT`, the data remains safe because it is written to disk.

`Isolation` - concurrent transactions run as if each transaction had the database completely ‘to itself’. concurrently executing transactions are isolated from each other: they cannot step on each other’s toes... the database ensures that when the transactions have committed, the result is the same as if they had run one after another, even though in reality they may have run concurrently.

▼ example

Preventing Dirty Reads
Let's say two users try to
read and update the same row at the same time.

Transaction 1 (User A)

```
BEGIN TRANSACTION;
UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;
SELECT pg_sleep(15); -- Simulate delay
COMMIT;
```

Transaction 2 (User B)

```
BEGIN TRANSACTION;
SELECT Balance FROM Accounts WHERE AccountID = 1; -- Should not see uncommitted changes from User A
COMMIT;
```

Without proper isolation, User B might see a wrong balance. Using `SET TRANSACTION ISOLATION LEVEL READ COMMITTED;` ensures that **User B only sees committed data**.

concurrent transactions - concurrent transactions significantly increase the performance of a system, but they may lead to inconsistency.

Transaction Isolation Levels

strongest isolation level - serializable - guarantees that the results of concurrently executing transactions are equivalent to the results of executing these transactions in serial order.

weaker isolation levels in transactions boost performance. developers work with them if inconsistencies can be tolerated or if the application itself takes care of consistency issues. a database has an isolation level set as a default for transactions, it is a weaker isolation level.

higher isolation level needs to be specified, if needed at the beginning of the transaction.

Concurrent Write operations

example: both sequence run as transactions with isolation level repeatable read.

Isolation Level	Dirty Write Anomaly	Lost Update Anomaly
	overwrite uncommitted data	overwrite committed data
Read Committed	does not happen (is prevented)	happpens (is not prevented)
Repeatable Read	does not happen (is prevented)	does not happen (is prevented)

Dirty Write anomaly

Transaction T1 modifies a data item. Another transaction T2 then further modifies that data item before T1 performs a COMMIT or ROLLBACK.

https://surfingcomplexity.blog/2024/07/05/dirty-writes/

Lost Update Anomaly

when two transactions are updating the same record at the same time in a DBMS then a lost update problem occurs. The first transaction updates a record and the second transaction updates the same record again, which nullifies the update of the first transaction.

Read Committed (Default)

PostgreSQL's default isolation level, **Read Committed**, ensures transactions **only see committed data** (**prevents dirty reads**). However, each query within a transaction sees the latest committed data at execution, allowing **non-repeatable reads** and **phantom reads**. If one transaction reads data, another transaction updates and commits changes before the first one re-reads, it will see the updated values. This provides a balance of **consistency and performance** for most applications.

▼ Example

```
--T1
BEGIN;
SELECT balance FROM accounts WHERE id = 1; -- Reads 100
SELECT pg_sleep(10); -- Simulates delay
SELECT balance FROM accounts WHERE id = 1; -- Might read an updated value if another transaction commits a char
COMMIT;

--T2
BEGIN;
UPDATE accounts SET balance = 200 WHERE id = 1;
COMMIT;
```

T1 initially reads `balance = 100`. T2 updates and commits `balance = 200`. T1's second `SELECT` sees the new value (`200`) because each statement reads the latest committed data.

Repeatable Read

Repeatable Read ensures a **consistent snapshot** of the database for the entire transaction, preventing **dirty reads** and **non-repeatable reads**, but allowing **phantom reads**. A transaction always sees the **same data** for rows it has read, even if another transaction updates or deletes them before it commits. However, new rows inserted by other transactions may still appear. This level is useful for **financial transactions** where consistency is critical.

▼ Example

Two transactions (`T1` and `T2`) both work with the same data, but `T1` runs in **Repeatable Read** isolation.

```
--T1
BEGIN ISOLATION LEVEL REPEATABLE READ;
-- Read balance for account 1
SELECT balance FROM accounts WHERE id = 1;  -- Reads 100
-- Simulate delay
SELECT pg_sleep(10);
-- Read balance again
SELECT balance FROM accounts WHERE id = 1;  -- Still reads 100 (unchanged, even if T2 commits)
COMMIT;
```

```
--T2
BEGIN;
-- Update balance
UPDATE accounts SET balance = 200 WHERE id = 1;
COMMIT;
```

T1 always sees `balance = 100`, even though **T2 commits** `balance = 200`. **T1 works with a consistent snapshot** taken at the start of the transaction. **T2 updates and commits changes, but T1 does not see them** until after it commits.